## TABLE OF CONTENTS

**Chapter 1. Overview**

1. Overview ........................................................................................................... 1
   1. What is DCGM ................................................................................................. 1
   1. Focus Areas .................................................................................................. 2
   1. Target Users ................................................................................................. 3

**Chapter 2. Getting Started**

2. Overview .......................................................................................................... 4
   2. Supported Platforms ..................................................................................... 4
   2. Installation..................................................................................................... 5
   2. Basic Components ....................................................................................... 7
   2. Modes of Operation ..................................................................................... 7
      2.4.1. Embedded Mode ................................................................................ 8
      2.4.2. Standalone Mode ............................................................................... 9
   2. Static Library ............................................................................................... 9

**Chapter 3. Feature Overview**

3. Overview .......................................................................................................... 10
   3. Groups ........................................................................................................ 11
   3. Configuration ............................................................................................... 13
   3. Policy .......................................................................................................... 15
      3.3.1. Notifications .................................................................................... 15
      3.3.2. Actions ........................................................................................... 17
   3.4. Job Stats .................................................................................................. 18
   3.5. Health and Diagnostics ........................................................................... 20
      3.5.1. Background Health Checks .............................................................. 20
      3.5.2. Active Health Checks ....................................................................... 21
   3.6. Topology .................................................................................................. 23
   3.7. NVLink Counters ..................................................................................... 24
   3.8. Field Groups ............................................................................................ 24
   3.9. Link Status ................................................................................................25
   3.10. Profiling Metrics .................................................................................... 26
      3.10.1. Profiling Metrics ............................................................................. 27
      3.10.2. CUDA Test Generator (dcgmproftester) ......................................... 28
      3.10.3. Platform Support ........................................................................... 29

**Chapter 4. Integrating with DCGM**

4. Overview .......................................................................................................... 30
   4. Integrating with DCGM Reader .................................................................... 30
      4.1.1. Reading Using the Dictionary ......................................................... 30
      4.1.2. Reading Using Inheritance ............................................................... 31
      4.1.3. Completing the Proof of Concept .................................................... 32
      4.1.4. Additional Customization ............................................................... 32
   4. Integrating with Prometheus and Grafana .................................................. 33
      4.2.1. Starting the Prometheus Server ....................................................... 33
      4.2.2. Starting the Prometheus Client ....................................................... 34
      4.2.3. Integrating with Grafana ............................................................... 35
4.2.4. Customizing the Prometheus Client ............................................................. 39

Chapter 5. DCGM Diagnostics................................................................................ 41
  5.1. Overview .................................................................................................. 41
    5.1.1. DCGM Diagnostics Goals ................................................................. 41
    5.1.2. Beyond the Scope of the DCGM Diagnostics ...................................... 42
    5.1.3. Dependencies .................................................................................. 42
    5.1.4. Supported Products ......................................................................... 42
  5.2. Using DCGM Diagnostics ........................................................................... 43
    5.2.1. Command line options ..................................................................... 43
    5.2.2. Usage Examples ............................................................................. 47
    5.2.3. Configuration file ........................................................................... 48
    5.2.4. Global parameters .......................................................................... 48
    5.2.5. GPU parameters ............................................................................ 49
    5.2.6. Test Parameters ............................................................................. 50
  5.3. Overview of Plugins .................................................................................. 51
    5.3.1. Deployment Plugin .......................................................................... 51
    5.3.2. PCIe - GPU Bandwidth Plugin ........................................................... 52
    5.3.3. Memory Bandwidth Plugin ................................................................. 58
    5.3.4. SM Stress Plugin ............................................................................. 59
    5.3.5. Hardware Diagnostic Plugin ............................................................... 62
    5.3.6. Targeted Stress Plugin ..................................................................... 65
    5.3.7. Power Plugin ................................................................................ 68
  5.4. Test Output ............................................................................................. 70
    5.4.1. JSON Output ................................................................................ 70

Chapter 6. DCGM Modularity ............................................................................ 71
  6.1. Module List .......................................................................................... 71
  6.2. Blacklisting Modules ............................................................................ 72
Chapter 1.
OVERVIEW

1.1. What is DCGM

The NVIDIA® Data Center GPU Manager (DCGM) simplifies administration of NVIDIA Tesla GPUs in cluster and datacenter environments. At its heart, DCGM is an intelligent, lightweight user space library/agent that performs a variety of functions on each host system:

- GPU behavior monitoring
- GPU configuration management
- GPU policy oversight
- GPU health and diagnostics
- GPU accounting and process statistics
- NVSwitch configuration and monitoring

This functionality is accessible programmatically through public APIs and interactively through CLI tools. It is designed to be run either as a standalone entity or as an embedded library within management tools.

This document is intended as an overview of DCGM’s main goals and features and is intended for system administrators, ISV developers, and individual users managing groups of Tesla GPUs.

TERMINOLOGY

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCGM</td>
<td>NVIDIA’s Datacenter GPU Manager</td>
</tr>
<tr>
<td>NVIDIA Host Engine</td>
<td>Standalone executable wrapper for DCGM shared library</td>
</tr>
<tr>
<td>Host Engine daemon</td>
<td>Daemon mode of operation for the NVIDIA Host Engine</td>
</tr>
</tbody>
</table>
### 1.2. Focus Areas

DCGM’s design is geared towards the following key functional areas.

**Manage GPUs as collections of related resources.** In the majority of large-scale GPU deployments there are multiple GPUs per host, and often multiple hosts per job. In most cases there is a strong desire to ensure homogeneity of behavior across these related resources, even as specific expectations may change from job to job or user to user, and even as multiple jobs may use resources on the same host simultaneously. DCGM applies a group-centric philosophy to node level GPU management.

**Configure NVSwitches.** On DGX-2 or HGX-2, all GPUs communicate by way of NVSwitch. The Fabric Manager component of DCGM configures the switches to form a single memory fabric among all participating GPUs, and monitors the NVLinks that support the fabric.

**Define and enforce GPU configuration state.** The behavior of NVIDIA GPUs can be controlled by users to match requirements of particular environments or applications. This includes performance characteristics such as clock settings, exclusivity constraints like compute mode, and environmental controls like power limits. DCGM provides enforcement and persistence mechanisms to ensure behavioral consistency across related GPUs.

**Automate GPU management policies.** NVIDIA GPUs have advanced capabilities that facilitate error containment and identify problem areas. Automated policies that define
GPU response to certain classes of events, including recovery from errors and isolation of bad hardware, ensure higher reliability and a simplified administration environment. DCGM provides policies for common situations that require notification or automated action.

*Provide robust, online health and diagnostics.* The ability to ascertain the health of a GPU and its interaction with the surrounding system is a critical management need. This need comes in various forms, from passive background monitoring to quick system validation to extensive hardware diagnostics. In all cases it is important to provide these features with minimal impact on the system and minimal additional environmental requirements. DCGM provides extensive automated and non-automated health and diagnostic capabilities.

*Enable job-level statistics and accounting.* Understanding GPU usage is important for schedulers and resource managers. Tying this information together with RAS events, performance information and other telemetry, especially at the boundaries of a workload, is very useful in explaining job behavior and root-causing potential performance or execution issues. DCGM provides mechanism to gather, group and analyze data at the job level.

### 1.3. Target Users

DCGM is targeted at the following users:

- OEMs and ISVs wishing to improve GPU integration within their software.
- Datacenter admins managing their own GPU enabled infrastructure.
- Individual users and FAEs needing better insight into GPU behavior, especially during problem analysis.
- All DGX-2 and HGX-2 users will use the Fabric Manager to configure and monitor the NVSwitch fabric.

DCGM provides different interfaces to serve different consumers and use cases. Programmatic access via C and Python is geared towards integration with 3rd-party software. Python interfaces are also geared towards admin-centric scripting environments. CLI-based tools are present to provide an interactive out-of-the-box experience for end users. Each interface provides roughly equivalent functionality.
Chapter 2. GETTING STARTED

2.1. Supported Platforms

DCGM currently supports the following products and environments:

- All K80 and newer Tesla GPUs
- NVSwitch on DGX A100, HGX A100. Note that for DGX-2 and HGX-2 systems, while a minimum version of DCGM 1.7 is required, DCGM 2.0 is recommended.
- All Maxwell and newer non-Tesla GPUs

Starting with v1.3, limited DCGM functionality is available on non-Tesla GPUs. More details are available in the section Feature Overview.

- CUDA 7.5+ and NVIDIA Driver R418+
  NVIDIA Driver R450 and later is required on systems using NVSwitch, such as DGX A100 or HGX A100. Starting with DCGM 2.0, Fabric Manager (FM) for NVSwitch systems is no longer bundled with DCGM packages. FM is a separate artifact that can be installed using the CUDA network repository. For more information, see the Fabric Manager User Guide.
- Bare metal and virtualized (full passthrough only)
- Supported Linux distributions and architectures are shown in the table below:

<table>
<thead>
<tr>
<th>Linux Distribution</th>
<th>x86 (x86_64)</th>
<th>Arm (aarch64)</th>
<th>POWER (ppc64le)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHEL 8.y</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RHEL/CentOS 7.y</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLES 15</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Ubuntu 20.04 LTS</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Ubuntu 18.04 LTS</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
2.2. Installation

To run DCGM the target system must include the following NVIDIA components, listed in dependency order:

1. Supported NVIDIA Datacenter Driver
2. Supported CUDA Toolkit
3. DCGM Runtime and SDK
4. DCGM Python bindings (if desired)

All of the core components are available as RPMs/DEBs from NVIDIA’s website. The Python bindings are available in the `/usr/src/dcgm/bindings` directory after installation. The user must be root or have sudo privileges for installation, as for any such packaging operations.

DCGM is tested and designed to run with NVIDIA Datacenter Drivers. Attempting to run on other drivers, such as a developer driver, could result in missing functionality.

To remove the previous installation (if any), perform the following steps (e.g. on an RPM-based system).

1. Make sure that the nv-hostengine is not running. You can stop it using the following command

   ```
   # sudo nv-hostengine -t
   ```

2. Remove the previous installation.

   ```
   # sudo yum remove datacenter-gpu-manager
   ```

It is recommended to install DCGM via Linux package managers from the CUDA network repository.

- Install DCGM on Ubuntu

  1. Install repository meta-data. Note substitute `<architecture>` as `x86_64`, `sbsa` or `ppc64le` as appropriate.

     ```
     $ distribution=$(./etc/os-release;echo $ID$VERSION_ID | sed -e 's/\./\//g')
     $ echo "deb http://developer.download.nvidia.com/compute/cuda/repos/$distribution/$arch/ /" | sudo tee /etc/apt/sources.list.d/cuda.list
     ```

  2. Trust the CUDA public GPG key

     When installing on Ubuntu 20.04/18.04:

     ```
     $ sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/repos/$distribution/$arch/7fa2af80.pub
     ```
When installing on Ubuntu 16.04:

```
$ sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/$distribution/$architecture/7fa2af80.pub
```

Pin file to prioritize CUDA repository:

```
$ sudo mv cuda-$distribution.pin /etc/apt/preferences.d/cuda-repository-pin-600
```

3. Update the Apt repository cache

```
$ sudo apt-get update
```

4. Install DCGM

```
$ sudo apt-get install -y datacenter-gpu-manager
```

5. Enable DCGM systemd service (on reboot) and start it now

```
$ sudo systemctl --now enable nvidia-dcgm
```

Install DCGM on RHEL

1. Install repository meta-data and GPG key. Note substitute <architecture> as x86_64, sbsa or ppc64le as appropriate.

```
$ distribution=$(. /etc/os-release;echo $ID'rpm -E "$(%{?rhel}%{?fedora})")

```

2. Update metadata

```
$ sudo dnf clean expire-cache
```

3. Install DCGM

```
$ sudo dnf install -y datacenter-gpu-manager
```

4. Enable DCGM systemd service (on reboot) and start it now

```
$ sudo systemctl --now enable nvidia-dcgm
```

Note that the default nvidia-dcgm.service files included in the installation package use the systemd format. If DCGM is being installed on OS distributions that use the init.d format, then these files may need to be modified.

To verify installation, start the standalone host engine and use dcgmi to query the system. You should see a listing of all supported GPUs:

```
# nv-hostengine
Starting host engine using port number : 5555

# dcgmi discovery -l
2 GPUs found.
+--------+-------------------------------------------------------------------+
| GPU ID | Device Information                                                |
+--------+===================================================================+
| 0      | Name: Tesla K80                                                   |
```
2.3. Basic Components

DCGM shared library

The user space shared library, libdcgm.so, is the core component of DCGM. This library implements the major underlying functionality and exposes this as a set of C-based APIs. It sits on top of the NVIDIA driver, NVML, and the CUDA Toolkit.

NVIDIA Host Engine

The NVIDIA host engine, nv-hostengine, is a thin wrapper around the DCGM shared library. Its main job is to instantiate the DCGM library as a persistent standalone process, including appropriate management of the monitoring and management activities.

- DCGM can run as root or non-root. Some DCGM functionality, such as configuration management, are not allowed to be run as non-root.
- On DGX-2 or HGX-2, nv-hostengine must run as root to enable the Fabric Manager.

DCGM CLI Tool

The command line interface to DCGM, dcgmi, is a network-capable interface into the NVIDIA host engine. It exposes much of the DCGM functionality in a simple, interactive format. It is intended for users and admins who want to control DCGM, or gather relevant data, without needing to build against the programmatic interfaces. It is not intended for scripting.

Python Bindings

The Python bindings are included with the DCGM package and installed in /usr/src/dcgm/bindings.

Software Development Kit

The DCGM SDK includes examples of how to leverage major DCGM features, alongside API documentation and headers. The SDK includes coverage for both C and Python based APIs, and include examples for using DCGM in both standalone and embedded modes. These are installed in /usr/src/dcgm/sdk_samples.

2.4. Modes of Operation

The core DCGM library can be run as a standalone process or be loaded by an agent as a shared library. In both cases it provides roughly the same class of functionality and has...
the same overall behavior. The choice of mode depends on how it best fits within the user’s existing environment.

2.4.1. Embedded Mode

In this mode the agent is loaded as a shared library. This mode is provided for the following situations:

- A 3rd-party agent already exists on the node, and
- Extra jitter associated with an additional autonomous agent needs to be managed

By loading DCGM as a shared library and managing the timing of its activity, 3rd-party agents can control exactly when DCGM is actively using CPU and GPU resources.

In this mode the 3rd-party agent should generally load the shared library at system initialization and manage the DCGM infrastructure over the lifetime of the host. Since DCGM is stateful, it is important that the library is maintained over the life of the 3rd-party agent, not invoked in a one-off fashion. In this mode all data gathering loops, management activities, etc. can be explicitly invoked and controlled via library interfaces. A 3rd-party agent may choose, for example, to synchronize DCGM activities across an entire multi-node job in this way.

Caution In this mode it is important that the various DCGM management interfaces be executed by the 3rd-party within the designated frequency ranges, as described in the API definitions. Running too frequently will waste resources with no noticeable gain. Running too infrequently will allow for gaps in monitoring and management coverage.

Working in this mode requires a sequence of setup steps and a management thread within the 3rd-party agent that periodically triggers all necessary DCGM background work. The logic is roughly as follows:

- **On Agent Startup**
  
  ```
  dcgmInit()
  ```
  System or job-level setup, e.g.
  call dcgmGroupCreate() to set up GPU groups
  call dcgmWatchFields() to manage watched metrics
  call dcgmPolicySet() to set policy

- **Periodic Background Tasks (managed)**
  
  Trigger system management behavior, i.e.
  ```
  call dcgmUpdateAllFields() to manage metrics
  call dcgmPolicyTrigger() to manage policies
  ```
  Gather system data, e.g.
  ```
  call dcgmHealthCheck() to check health
  call dcgmGetLatestValues() to get metric updates
  ```

- **On Agent Shutdown**
2.4.2. Standalone Mode

In this mode the DCGM agent is embedded in a simple daemon provided by NVIDIA, the NVIDIA Host Engine. This mode is provided for the following situations:

- DCGM clients prefer to interact with a daemon rather than manage a shared library resource themselves
- Multiple clients wish to interact with DCGM, rather than a single node agent
- Users wish to leverage the NVIDIA CLI tool, DCGMI
- Users of DGX-2 or HGX-2 systems will need to run the Host Engine daemon to configure and monitor the NVSwitches

Generally, NVIDIA prefers this mode of operation, as it provides the most flexibility and lowest maintenance cost to users. In this mode the DCGM library management routines are invoked transparently at default frequencies and with default behaviors, in contrast to the user control provided by the Embedded Mode. Users can either leverage DCGMI tool to interact with the daemon process or load the DCGM library with daemon's IP address during initialization for programmatic interaction.

The daemon leverages a socket-based interface to speak with external processes, e.g. DCGMI. Users are responsible for configuring the system initialization behavior, post DCGM install, to ensure the daemon is properly executed on startup.

- Helper installation scripts for daemon setup will be included in the next Release Candidate package.
- On DGX-2 or HGX-2 systems, nv-hostengine is automatically started at system boot time, so that the Fabric Manager can configure and monitor the NVSwitches.

2.5. Static Library

A statically-linked stub version of the DCGM library has been included for the purposes of being able to remove an explicit dependency on the DCGM shared library. This library provides wrappers to the DCGM symbols and uses `dlopen()` to dynamically access `libdcgm.so`. If the shared library is not installed, or cannot be found in the `LD_LIBRARY_PATH`, an error code is returned. When linking against this library `libdl` must be included in the compile line which is typically done using:

```
# gcc foo.c -o foo -ldcgm_stub -ldl
```
The following sections review key DCGM features, along with examples of input and output using the DCGMI tool. Common usage scenarios and suggested best practices are included as well. Starting with v1.3, DCGM is supported on non-Tesla GPUs. The following table lists the features available on different GPU products.

<table>
<thead>
<tr>
<th>Feature Group</th>
<th>Support Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tesla</td>
</tr>
<tr>
<td>Field Value Watches (GPU metrics)</td>
<td>X</td>
</tr>
<tr>
<td>Configuration Management</td>
<td>X</td>
</tr>
<tr>
<td>Active Health Checks (GPU subsystems)</td>
<td>X</td>
</tr>
<tr>
<td>Job Statistics</td>
<td>X</td>
</tr>
<tr>
<td>Topology</td>
<td>X</td>
</tr>
<tr>
<td>Introspection</td>
<td>X</td>
</tr>
<tr>
<td>Policy Notification</td>
<td>X</td>
</tr>
<tr>
<td>GPU Diagnostics (Diagnostic Levels - 1, 2, 3)</td>
<td>All Levels</td>
</tr>
</tbody>
</table>

While DCGM interfaces are shown, all functionality below is accessible via the C and Python APIs as well.
3.1. Groups

Almost all DCGM operations take place on groups. Users can create, destroy and modify collections of GPUs on the local node, using these constructs to control all subsequent DCGM activities.

Groups are intended to help the user manage collections of GPUs as a single abstract resource, usually correlated to the scheduler’s notion of a node-level job. By working in this way clients can ask question about the entire job, such as job-level health, without needing to track the individual resources.

Note: Today DCGM does not enforce group behavior beyond itself, e.g. through OS isolation mechanisms like cgroups. It is expected that clients do this externally. The ability for clients to opt-in to DCGM enforcement of this state is likely in the future.

In machines with only one GPU the group concept can be ignored altogether, as all DCGM operations that require a group can use one containing that sole GPU. For convenience, at init, DCGM creates a default group representing all supported GPUs in the system.

Groups in DCGM need not be disjoint. In many cases it may make sense to maintain overlapping groups for different needs. Global groups, consisting of all GPUs in the system, are useful for node-level concepts such as global configuration or global health. Partitioned groups, consisting of only a subset of GPUs, are useful for job-level concepts such as job stats and health.

Tip: It is recommended that the client maintain one long-lived global group for node-level activities. For systems with multiple transient concurrent workloads it is recommended that additional partitioned groups be maintained on a per-job basis.

For example, a group created to manage the GPUs associated with a single job might have the following lifecycle. During prologue operations the group is created, configured, and used to verify the GPUs are ready for work. During epilogue operations the groups is used to extract target information. And while the job is running, DCGM works in the background to handle the requested behaviors.

Managing groups is very simple. Using the dcgmi group subcommand, the following example shows how to create, list and delete a group.
To add GPUs to a group it is first necessary to identify them. This can be done by first asking DCGM for all supported GPUs in the system.

```
# dcmi discovery -l
2 GPUs found.
+-----------------+---------------------------------------------------------------+
| GPU ID | Device Information                   |
|-------+=================================================================
| 0     | Name: Tesla K80                                                              |
|       | PCI Bus ID: 0000:07:00.0                                                      |
|       | Device UUID: GPU-0000000000000000000000000000000000000000000000000000000000000000 |
| 1     | Name: Tesla K80                                                              |
|       | PCI Bus ID: 0000:08:00.0                                                      |
|       | Device UUID: GPU-1111111111111111111111111111111111111111111111111111111111111111 |
```

```
# dcmi group -g 1 -a 0,1
Add to group operation successful.
```
3.2. Configuration

An important aspect of managing GPUs, especially in multi-node environments, is ensuring consistent configuration across workloads and across devices. In this context the term *configuration* refers to the set of administrative parameters exposed by NVIDIA to tune GPU behavior. DCGM makes it easier for clients to define target configurations and ensure those configurations are maintained over time.

It is important to note that different GPU properties have different levels of persistence. There are two broad categories:

- **Device InfoROM lifetime**
  - Non-volatile memory on each board, holding certain configurable firmware settings.
  - Persists indefinitely, though firmware can be flashed.
- **GPU initialization lifetime**
  - Driver level data structures, holding volatile GPU runtime information.
  - Persists until the GPU is de-initialized by the kernel mode driver.

DCGM is primarily focused on maintaining configuration settings that fall into the second category. These settings are normally volatile, potentially resetting each time a GPU becomes idle or is reset. By using DCGM a client can ensure that these settings persist over the desired lifetime.

In most common situations the client should be expected to define a configuration for all GPUs in the system (global group) at system initialization, or define individual partitioned group settings on a per-job basis. Once a configuration has been defined DCGM will enforce that configuration, for example across driver restarts, GPU resets or at job start.

DCGM currently supports the follows configuration settings:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync Boost</td>
<td>Coordinate Auto Boost across GPUs in the group</td>
<td>None</td>
</tr>
<tr>
<td>Target Clocks</td>
<td>Attempt to maintain fixed clocks at the target values</td>
<td>None</td>
</tr>
<tr>
<td>ECC Mode</td>
<td>Enable ECC protection throughout the GPU’s memory</td>
<td>Usually On</td>
</tr>
</tbody>
</table>
Feature Overview

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Limit</td>
<td>Set the maximum allowed power consumption</td>
<td>Varies</td>
</tr>
<tr>
<td>Compute Mode</td>
<td>Limit concurrent process access to the GPU</td>
<td>No restrictions</td>
</tr>
</tbody>
</table>

To define a target configuration for a group, use the `dcgmi config` subcommand. Using the group created in the section above, the following example shows how to set a compute mode target and then list the current configuration state.

```bash
# dcgmi config -g 1 --set -c 2
Configuration successfully set.
# dcgmi config -g 1 --get
```

```
+--------------------------+------------------------+------------------------+
| GPU_Group                |                        |                        |
| Group of 2 GPUs          | TARGET CONFIGURATION   | CURRENT CONFIGURATION  |
+==========================+========================+========================+
| Sync Boost               | Not Specified          | Disabled               |
| SM Application Clock     | Not Specified          | ****                   |
| Memory Application Clock | Not Specified          | ****                   |
| ECC Mode                 | Not Specified          | ****                   |
| Power Limit              | Not Specified          | ****                   |
| Compute Mode             | E. Process             | E. Process             |
+--------------------------+------------------------+------------------------+
**** Non-homogenous settings across group. Use with --v flag to see details.
```

```bash
#dcgmi config -g 1 --get --verbose
```

```
+--------------------------+------------------------+------------------------+
| GPU_ID: 0                |                        |                        |
| Tesla K20c               | TARGET CONFIGURATION   | CURRENT CONFIGURATION  |
+==========================+========================+========================+
| Sync Boost               | Not Specified          | Disabled               |
| SM Application Clock     | Not Specified          | 705                    |
| Memory Application Clock | Not Specified          | 2600                   |
| ECC Mode                 | Not Specified          | Disabled               |
| Power Limit              | Not Specified          | 225                    |
| Compute Mode             | E. Process             | E. Process             |
+--------------------------+------------------------+------------------------+
```

```
+--------------------------+------------------------+------------------------+
| GPU_ID: 1                |                        |                        |
| GeForce GT 430           | TARGET CONFIGURATION   | CURRENT CONFIGURATION  |
+==========================+========================+========================+
| Sync Boost               | Not Specified          | Disabled               |
| SM Application Clock     | Not Specified          | 562                    |
| Memory Application Clock | Not Specified          | 2505                   |
| ECC Mode                 | Not Specified          | Enabled                |
| Power Limit              | Not Specified          | 200                    |
| Compute Mode             | E. Process             | E. Process             |
+--------------------------+------------------------+------------------------+
```

Once a configuration is set, DCGM maintains the notion of Target and Current state. Target tracks the user’s request for configuration state while Current tracks the actual state of the GPU and group. These are generally maintained such that they are equivalent with DCGM restoring current state to target in situations where that state is lost or changed. This is common in situations where DCGM has executed some invasive policy like a health check or GPU reset.
3.3. Policy

DCGM provides a way for clients to configure automatic GPU behaviors in response to various conditions. This is useful for event->action situations, such as GPU recovery in the face of serious errors. It’s also useful for event->notification situations, such as when a client wants to be warned if a RAS event occurs. In both scenarios the client must define a condition on which to trigger further behavior. These conditions are specified from a predefined set of possible metrics. In some cases the client must also provide a threshold above/below which the metric condition is triggered. Generally, conditions are fatal and non-fatal RAS events, or performance-oriented warnings. These include the following examples:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Type</th>
<th>Threshold</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe/NVLink Errors</td>
<td>Fatal</td>
<td>Hardcoded</td>
<td>Uncorrected, or corrected above SDC threshold</td>
</tr>
<tr>
<td>ECC Errors</td>
<td>Fatal</td>
<td>Hardcoded</td>
<td>Single DBE, multiple co-located SBEs</td>
</tr>
<tr>
<td>Page Retirement Limit</td>
<td>Non-Fatal</td>
<td>Settable</td>
<td>Lifetime limit for ECC errors, or above RMA rate</td>
</tr>
<tr>
<td>Power Excursions</td>
<td>Performance</td>
<td>Settable</td>
<td>Excursions above specified board power threshold</td>
</tr>
<tr>
<td>Thermal Excursions</td>
<td>Performance</td>
<td>Settable</td>
<td>Excursions above specified GPU thermal threshold</td>
</tr>
<tr>
<td>XIDs</td>
<td>All</td>
<td>Hardcoded</td>
<td>XIDs represent several kinds of events within the NVIDIA driver such as pending page retirements or GPUs falling off the bus. See <a href="http://docs.nvidia.com/deploy/xid-errors/index.html">http://docs.nvidia.com/deploy/xid-errors/index.html</a> for details.</td>
</tr>
</tbody>
</table>

3.3.1. Notifications

The simplest form of a policy is to instruct DCGM to notify a client when the target condition is met. No further action is performed beyond this. This is primarily interesting as a callback mechanism within the programmatic interfaces, as a way to avoid polling.

When running DCGM in embedded mode such callbacks are invoked automatically by DCGM each time a registered condition is hit, at which point the client can deal with that event as desired. The client must register through the appropriate API calls to
receive these callbacks. Doing so transparently instructs DCGM to track the conditions that trigger those results.

Once a callback has been received for a particular condition, that notification registration is terminated. If the client wants repeated notifications for a condition it should re-register after processing each callback.

The `dcgmi policy` subcommand does allow access to some of this functionality from the command line via setting of conditions and via a blocking notification mechanism. This can be useful when watching for a particular problem, e.g. during a debugging session.

As an example, the following shows setting a notification policy for PCIe fatal and non-fatal events:

```
# dcgmi policy -g 2 --set 0,0 -p
Policy successfully set.

# dcgmi policy -g 2 --get
Policy information
+---------------------------+------------------------------------------------+
| GPU_Group                 | Policy Information                             |
|----------------------------+------------------------------------------------+
| Violation conditions      | PCI errors and replays                         |
| Isolation mode            | Manual                                         |
| Action on violation       | None                                           |
| Validation after action   | None                                           |
| Validation failure action | None                                           |
+---------------------------+------------------------------------------------+

**** Non-homogenous settings across group. Use with -v flag to see details.

# dcgmi policy -g 2 --get --verbose
Policy information
+---------------------------+------------------------------------------------+
| GPU ID: 0                 | Policy Information                             |
|----------------------------+------------------------------------------------+
| Violation conditions      | PCI errors and replays                         |
| Isolation mode            | Manual                                         |
| Action on violation       | None                                           |
| Validation after action   | None                                           |
| Validation failure action | None                                           |
+---------------------------+-----------------------------------------------+

| GPU ID: 1                 | Policy Information                             |
|----------------------------+------------------------------------------------+
| Violation conditions      | PCI errors and replays                         |
| Isolation mode            | Manual                                         |
| Action on violation       | None                                           |
| Validation after action   | None                                           |
| Validation failure action | None                                           |
+---------------------------+------------------------------------------------+

Once such a policy is set the client will receive notifications accordingly. While this is primarily interesting for programmatic use cases, `dcgmi policy` can be invoked to wait for policy notifications:

```
# dcgmi policy -g 2 --reg
Listening for violations
...
A PCIe error has violated policy manager values.
...
3.3.2. Actions

Action policies are a superset of the notification policies above.

Some clients may find it useful to tie a condition to an action that DCGM performs automatically as soon as the condition is met. This is most pertinent when the condition is a RAS event that prevents the GPU from otherwise operating normally.

Policies that are defined as actions include three additional components:

1. **Isolation mode** – whether DCGM grabs exclusive access to the GPU before performing the subsequent policy steps.
2. **Action** – The DCGM invasive behavior to perform.
3. **Validation** – Any follow-up validation of GPU state, post action.

A common action based policy is to configure DCGM to automatically retire a memory page after an ECC DBE has occurred. By retiring the page and re-initializing the GPU, DCGM can isolate the hardware fault and prepare the GPU for the next job. Since this operation involves resetting the GPU, a quick system validation is a follow-up step to ensure the GPU is healthy.

Clients setting action policies receive two notifications each time the policy is run.

1. Notification callback when condition is hit and policy enacted.
2. Notification callback when action completes, i.e. after validation step.

Using the `dcgmi policy` subcommand, this kind of action-based policy can be configured as follows:

```
# dcgmi policy -g 1 --set 1,1 -e
Policy successfully set.

# dcgmi policy -g 1 --get
Policy information for group 1
+----------------------------------------------------------------------------+
| GPU ID: 0                 | Policy Information                             |
|===========================|================================================|
| Violation Conditions      | Double-bit ECC errors                          |
| Isolation mode            | Manual                                         |
| Action on violation       | Reset GPU                                      |
| Validation after action   | NVVS (Short)                                   |
| Validation failure action | None                                           |
+----------------------------------------------------------------------------+
```
As shown in the previous section, `dcgmi policy` can also be used to watch for notifications associated with this policy.

### 3.4. Job Stats

DCGM provides background data gathering and analysis capabilities, including the ability to aggregate data across the lifetime of a target workload and across the GPUs involved. This makes it easy for clients to gather job level data, such as accounting, in a single request.

To request this functionality a client must first enable stats recording for the target group. This tells DCGM that all relevant metrics must be periodically watched for those GPUs, along with process activity on the devices. This need only be done once at initialization for each job-level group.

```bash
# dcgmi stats -g 1 --enable
Successfully started process watches on group 1.
```

Stats recording must be enabled prior to the start of the target workload(s) for reliable information to be available.

Once a job has completed DCGM can be queried for information about that job, both at the summary level of a group and, if needed, broken down individually between the GPUs within that group. The suggested behavior is that clients perform this query in epilogue scripts as part of job cleanup.

An example of group-level data provided by `dcgmi stats`:

```bash
dcgmi stats --pid 1234 -v
Successfully retrieved process info for pid: 1234. Process ran on 1 GPUs.

| GPU ID: 0 |

---------- Execution Stats ----------
| Start Time * | Tue Nov 3 17:36:43 2015 |
| End Time * | Tue Nov 3 17:38:33 2015 |
| Total Execution Time (sec) * | 110.33 |
| No. of Conflicting Processes * | 0 |

--------- Performance Stats --------
| Energy Consumed (Joules) | 15758 |
| Power Usage (Watts) | Avg: 150, Max: 250, Min: 100 |
| Max GPU Memory Used (bytes) * | 213254144 |
| SM Clock (MHz) | Avg: 837, Max: 875, Min: 679 |
| Memory Clock (MHz) | Avg: 2505, Max: 2505, Min: 2505 |
| SM Utilization (%) | Avg: 99, Max: 100, Min: 99 |
| Memory Utilization (%) | Avg: 2, Max: 3, Min: 0 |
| PCIe Rx Bandwidth (megabytes) | Avg: N/A, Max: N/A, Min: N/A |
| PCIe Tx Bandwidth (megabytes) | Avg: N/A, Max: N/A, Min: N/A |

--------- Event Stats ----------
| Single Bit ECC Errors | 0 |
| Double Bit ECC Errors | 0 |
| PCIe Replay Warnings | 0 |
| Critical XID Errors | 0 |

--------- Slowdown Stats ----------
| Due to - Power (%) | 0 |
| - Thermal (%) | 0 |
```
For certain frameworks the processes and their PIDs cannot be associated with a job directly, and the process associated with a job may spawn many children. In order to get job-level stats for such a scenario, DCGM must be notified when a job starts and stops. It is required that the client notifies DCGM with the user defined job id and the corresponding GPU group at job prologue, and notifies with the job id at the job epilogue. The user can query the job stats using the job id and get aggregated stats across all the pids during the window of interest.

An example of notifying DCGM at the beginning and end of the job using dcgmi:

```
# dcgmi stats -g 1 -s <user-provided-jobid>
Successfully started recording stats for <user-provided-jobid>
# dcgmi stats -x <user-provided-jobid>
Successfully stopped recording stats for <user-provided-jobid>
```

The stats corresponding to the job id already watched can be retrieved using dcgmi:

```
# dcgmi stats -j <user-provided-jobid>
Successfully retrieved statistics for <user-provided-jobid>
```

<table>
<thead>
<tr>
<th>GPU ID: 0</th>
</tr>
</thead>
</table>

|------- Execution Stats ----------+-----------------------------------------|
| Start Time                       | Tue Nov 3 17:36:43 2015                 |
| End Time                         | Tue Nov 3 17:38:33 2015                 |
| Total Execution Time (sec)       | 110.33                                  |
| No. of Processes                 | 0                                       |

|----- Performance Stats ----------+-----------------------------------------|
| Energy Consumed (Joules)         | 15758                                   |
| Power Usage (Watts)              | Avg: 150, Max: 250, Min 100             |
| Max GPU Memory Used (bytes)      | 213254144                               |
| SM Clock (MHz)                   | Avg: 837, Max: 875, Min: 679           |
| Memory Clock (MHz)               | Avg: 2505, Max: 2505, Min: 2505        |
| SM Utilization (%)               | Avg: 99, Max: 100, Min: 99             |
| Memory Utilization (%)           | Avg: 2, Max: 3, Min: 0                 |
| PCIe Rx Bandwidth (megabytes)    | Avg: N/A, Max: N/A, Min: N/A           |
| PCIe Tx Bandwidth (megabytes)    | Avg: N/A, Max: N/A, Min: N/A           |

|----- Event Stats ----------------+-----------------------------------------|
| Single Bit ECC Errors            | 0                                       |
| Double Bit ECC Errors            | 0                                       |
| PCIe Replay Warnings             | 0                                       |
| Critical XID Errors              | 0                                       |

|----- Slowdown Stats -------------+-----------------------------------------|
| Due to - Power (%)                | 0                                       |
| - Thermal (%)                     | 0                                       |
| - Reliability (%)                 | 0                                       |
| - Board Limit (%)                 | 0                                       |
| - Low Utilization (%)             | 0                                       |
| - Sync Boost (%)                  | Not Supported                           |
3.5. Health and Diagnostics

DCGM provides several mechanisms for understanding GPU health, each targeted at different needs. By leveraging each of these interfaces it is easy for clients to determine overall GPU health non-invasively -- while workloads are running, and actively -- when the GPU(s) can run dedicated tests. A new major feature of DCGM is the ability to run online hardware diagnostics.

More detailed targeted use cases are as follows:

- **Background health checks.**
  
  These are non-invasive monitoring operations that occur while jobs are running, and can be queried at any time. There is no impact on application behavior or performance.

- **Prologue health checks.**
  
  Quick, invasive health checks that take a few seconds and are designed to verify that a GPU is ready for work prior to job submission.

- **Epilogue health checks.**
  
  Medium duration invasive health checks, taking a few minutes, that can be run when a job has failed or a GPU’s health is otherwise suspect.

- **Full system validation.**
  
  Long duration invasive health checks, taking tens of minutes, that can be run when a system is being actively investigated for hardware problems or other serious issues.

**Caution** All of these are online diagnostics, meaning they run within the current environment. There is potential for factors beyond the GPU to influence behavior in negative ways. While these tools try to identify those situations, full offline diagnostics delivered via a different NVIDIA tool are required for complete hardware validation, and are required for RMA.

3.5.1. Background Health Checks

This form of health check is based on passive, background monitoring of various hardware and software components. The objective is to identify problems in key areas without impact on application behavior or performance. These kinds of checks can catch serious problems, such as unresponsive GPUs, corrupted firmware, thermal escapes, etc.

When such issues are identified they are reported by DCGM as warnings or errors. Each situation can require different client responses, but the following guidelines are usually true:
Feature Overview

- **Warning** – an issue has been detected that won’t prevent current work from completing, but the issue should be examined and potentially addressed in the future.
- **Error** – a critical issue has been detected and the current work is likely compromised or interrupted. These situations typically correspond to fatal RAS events and usually indicate the need for job termination and GPU health analysis.

Background health checks can be set and watched via simple DCGM interfaces. Using `dcgmi health` as the interface, the following code sets several health checks for a group and then verifies that those checks are currently enabled:

```
# dcgmi health -g 1 -s mpi
Health monitor systems set successfully.
```

To view the current status of all GPUs in the group the client can simply query for the overall group health. The result is an overall health score for the group as well as individual results for each impacted GPU, identifying key problems.

For example, DCGM would show the following when excessive PCIe replay events or InfoROM issues are detected:

```
# dcgmi health -g 1 -c
Health Monitor Report
+----------------------------------------------------------------------------+
| Group 1          | Overall Health: Warning |
+==================+================================+ |
| GPU ID: 0        | Warning                     |
|                  | PCIe system: Warning - Detected more than 8 PCIe replays per minute for GPU 0: 13 |
+------------------+--------------------------------------------------------------------------------+ |
| GPU ID: 1        | Warning                     |
|                  | InfoROM system: Warning - A corrupt InfoROM has been detected in GPU 1. |
+------------------+--------------------------------------------------------------------------------+ |
```

The `dcgmi` interfaces above only report current health status. The underlying data, exposed via other interfaces, captures more information about the timeframe of the events and their connections to executing processes on the GPU.

### 3.5.2. Active Health Checks

This form of health check is invasive, requiring exclusive access to the target GPUs. By running real workloads and analyzing the results, DCGM is able to identify common problems of a variety of types. These include:

- **Deployment and Software Issues**
  - NVIDIA library access and versioning
  - 3rd-party software conflicts
- **Integration Issues**
  - Correctable/uncorrectable issues on PCIe/NVLINK busses
  - Topological limitations
  - OS-level device restrictions, cgroups checks
  - Basic power and thermal constraint checks
Stress Checks

- Power and thermal stress
- PCIe/NVLINK throughput stress
- Constant relative system performance
- Maximum relative system performance

Hardware Issues and Diagnostics

- GPU hardware and SRAMs
- Computational robustness
- Memory
- PCIe/NVLINK busses

DCGM exposes these health checks through its diagnostic and policy interfaces. DCGM provides three levels of diagnostic capability (see `dcgmi diag help` on the command line). DCGM runs more in-depth tests to verify the health of the GPU at each level. The test names and tests run at each level are provided in the table below:

<table>
<thead>
<tr>
<th>Test Suite Name</th>
<th>Run Level</th>
<th>Test Duration</th>
<th>Test Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick</td>
<td>1</td>
<td>~ seconds</td>
<td>Software: Deployment</td>
</tr>
<tr>
<td>Medium</td>
<td>2</td>
<td>~ 2 minutes</td>
<td>Deployment: Memory Test</td>
</tr>
<tr>
<td>Long</td>
<td>3</td>
<td>~15 minutes</td>
<td>Deployment: Memory Test, Memory Bandwidth, HW Diagnostic Tests</td>
</tr>
</tbody>
</table>

While simple tests of runtime libraries and configuration are possible on non-Tesla GPUs (Run Level 1), DCGM is also able to perform hardware diagnostics, connectivity diagnostics, and a suite of stress tests on Tesla GPUs to help validate health and isolate problems. The actions in each test type are further described in the section GPU Parameters.

For example, running the full system validation (long test):

```bash
# dcgmi diag -g 1 -r 3
Successfully ran diagnostic for group 1.
+---------------------------------------------+---------------------------------------------+
| Diagnostic                                 | Result                                      |
+=============================================+=============================================+
|------- Deployment --------------------------+---------------------------------------------+
| Blacklist                                  | Pass                                       |
| NVML Library                               | Pass                                       |
| CUDA Main Library                          | Pass                                       |
| CUDA Toolkit Libraries                     | Pass                                       |
| Permissions and OS Blocks                  | Pass                                       |
| Persistence Mode                          | Pass                                       |
| Environment Variables                     | Pass                                       |
| Page Retirement                           | Pass                                       |
| Graphics Processes                         | Pass                                       |
+---------------------------------------------+---------------------------------------------+
```
<table>
<thead>
<tr>
<th>GPU Memory</th>
<th>Pass - All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostic</td>
<td>Pass - All</td>
</tr>
<tr>
<td>Integration</td>
<td>Pass - All</td>
</tr>
<tr>
<td>PCIe</td>
<td>Pass - All</td>
</tr>
<tr>
<td>Performance</td>
<td>Pass - All</td>
</tr>
<tr>
<td>SM Stress</td>
<td>Pass - All</td>
</tr>
<tr>
<td>Targeted Stress</td>
<td>Pass - All</td>
</tr>
<tr>
<td>Targeted Power</td>
<td>Pass - All</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>Pass – All</td>
</tr>
</tbody>
</table>

Diagnostic configuration options, as well as verbose output with a description of failures/actions will be included in the next Release Candidate package.

The diagnostic tests can also be run as part of the validation phase of action-based policies. A common scenario, for example, would be to run the short version of the test as a validation to a DBE page retirement action.

DCGM will store logs from these tests on the host file system. Two types of logs exist:

- Hardware diagnostics include an encrypted binary log, only viewable by NVIDIA.
- System validation and stress checks provide additional time series data via JSON text files. These can be viewed in numerous programs to see much more detailed information about GPU behavior during each test.

For complete details about the active health checks, including descriptions of the plugins and their various failure conditions, please read chapter 5.

### 3.6. Topology

DCGM provides several mechanisms for understanding GPU topology both at a verbose device-level view and non-verbose group-level view. These views are designed to give a user information about connectivity to other GPUs in the system as well as NUMA/affinity information.

For the device-level view:

```bash
# dcmd topo --gpuid 0
```

<table>
<thead>
<tr>
<th>GPU ID: 0</th>
<th>Topology Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core Affinity</td>
<td>0 - 11</td>
</tr>
<tr>
<td>To GPU 1</td>
<td>Connected via an on-board PCIe switch</td>
</tr>
<tr>
<td>To GPU 2</td>
<td>Connected via a PCIe host bridge</td>
</tr>
</tbody>
</table>

And for the group-level view:

```bash
# dcmd topo -g 1
```

<table>
<thead>
<tr>
<th>MyGroup</th>
<th>Topology Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core Affinity</td>
<td>0 - 11</td>
</tr>
<tr>
<td>NUMA Optimal</td>
<td>True</td>
</tr>
<tr>
<td>Worst Path</td>
<td>Connected via a PCIe host bridge</td>
</tr>
</tbody>
</table>
3.7. NVlink Counters

DCGM provides a way to check the nvlink error counters for various links in the system. This makes it easy for clients to catch abnormalities and watch the health of the communication over nvlink. There are multiple types of nvlink errors that are accounted for by DCGM as follows:

1. **CRC FLIT Error**: Data link receive flow control digit CRC error
2. **CRC Data Error**: Data link receive data CRC error.
3. **Replay Error**: Transmit replay error.
4. **Recovery Error**: Transmit recovery error.

To check the nvlink counters for all the nvlink present in gpu with gpu Id 0:

```
# dcgmi nvlink --errors -g 0
```

<table>
<thead>
<tr>
<th>GPU ID: 0</th>
<th>NVLINK Error Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link 0</td>
<td>CRC FLIT Error</td>
</tr>
<tr>
<td>Link 0</td>
<td>CRC Data Error</td>
</tr>
<tr>
<td>Link 0</td>
<td>Replay Error</td>
</tr>
<tr>
<td>Link 0</td>
<td>Recovery Error</td>
</tr>
<tr>
<td>Link 1</td>
<td>CRC FLIT Error</td>
</tr>
<tr>
<td>Link 1</td>
<td>CRC Data Error</td>
</tr>
<tr>
<td>Link 1</td>
<td>Replay Error</td>
</tr>
<tr>
<td>Link 1</td>
<td>Recovery Error</td>
</tr>
<tr>
<td>Link 2</td>
<td>CRC FLIT Error</td>
</tr>
<tr>
<td>Link 2</td>
<td>CRC Data Error</td>
</tr>
<tr>
<td>Link 2</td>
<td>Replay Error</td>
</tr>
<tr>
<td>Link 2</td>
<td>Recovery Error</td>
</tr>
<tr>
<td>Link 3</td>
<td>CRC FLIT Error</td>
</tr>
<tr>
<td>Link 3</td>
<td>CRC Data Error</td>
</tr>
<tr>
<td>Link 3</td>
<td>Replay Error</td>
</tr>
<tr>
<td>Link 3</td>
<td>Recovery Error</td>
</tr>
</tbody>
</table>

3.8. Field Groups

DCGM provides predefined groups of fields like job statistics, process statistics, and health for ease of use. Additionally, DCGM allows users to create their own custom groups of fields called field groups. Users can watch a group of fields on a group of GPUs and then retrieve either the latest values or a range of values of every field in a field group for every GPU in a group.

Field groups are not used directly in DCGMI, but you can still look at them and manage them from DCGMI.

To see all of the active field groups on a system, run:

```
# dcgmi fieldgroup -l
4 field groups found.
```

<table>
<thead>
<tr>
<th>FIELD GROUPS</th>
</tr>
</thead>
</table>
If you want to create your own field group, pick a unique name for it, decide which field IDs you want inside of it, and run:

```
# dcgmi fieldgroup -c mygroupname -f 50,51,52
```

Successfully created field group "mygroupname" with a field group ID of 5

Note that field IDs come from dcgm_fields.h and are the macros that start with DCGM_FI_.

Once you have created a field group, you can query its info:

```
#dcgmi fieldgroup -i --fieldgroup 5
```

If you want to delete a field group, run the following command:

```
# dcgmi fieldgroup -d -g 5
```

Successfully removed field group 5

Note that DCGM creates a few field groups internally. Field groups that are created internally, like the ones above, cannot be removed. Here is an example of trying to delete a DCGM-internal field group:

```
# dcgmi fieldgroup -d -g 1
Error: Cannot destroy field group 1. Return: No permission.
```

### 3.9. Link Status

In DCGM 1.5, you can query the status of the NVLinks of the GPUs and NVSwitches attached to the system with the following command:

```
# dcgmi nvlink --link-status
```
3.10. Profiling Metrics

As GPU-enabled servers become more common in the datacenter, it becomes important to better understand applications’ performance and the utilization of GPU resources in
the cluster. Profiling metrics in DCGM enables the collection of a set of metrics using the hardware counters on the GPU. DCGM provides access to device-level metrics at low performance overhead in a continuous manner. This feature is currently in beta starting with DCGM 1.7.

DCGM includes a new profiling module to provide access to these metrics. The new metrics are available as new fields (i.e., new IDs) via the regular DCGM APIs (such as the C, Python, Go bindings or the dcgmi command line utility). The installer packages also include an example CUDA based test load generator (called dcgmproftester) to demonstrate the new capabilities.

3.10.1. Profiling Metrics

In this release of DCGM, the following new device-level profiling metrics are supported. The definitions and corresponding DCGM field IDs are listed. By default, DCGM provides the metrics at a sample rate of 1Hz (every 1000ms). Users can query the metrics at any configurable frequency (minimum is 100ms) from DCGM (for example, see dcgmi dmon -d ).

Table 2 Device Level GPU Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
<th>DCGM Field Name (DCGM_FI_*) and ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics Engine Activity</td>
<td>Ratio of time the graphics engine is active. The graphics engine is active if a graphics/compute context is bound and the graphics pipe or compute pipe is busy.</td>
<td>PROF_GR_ENGINE_ACTIVE (ID: 1001)</td>
</tr>
<tr>
<td>SM Activity</td>
<td>The ratio of cycles an SM has at least 1 warp assigned (computed from the number of cycles and elapsed cycles)</td>
<td>PROF_SM_ACTIVE (ID: 1002)</td>
</tr>
<tr>
<td>SM Occupancy</td>
<td>The ratio of number of warps resident on an SM. (number of resident warps as a percentage of the theoretical maximum number of warps per elapsed cycle)</td>
<td>PROF_SM_OCCUPANCY (ID: 1003)</td>
</tr>
<tr>
<td>Tensor Activity</td>
<td>The ratio of cycles the tensor (HMMA) pipe is active (off the peak sustained elapsed cycles)</td>
<td>PROF_PIPE_TENSOR_ACTIVE (ID: 1004)</td>
</tr>
<tr>
<td>Memory BW Utilization</td>
<td>The ratio of cycles the device memory interface is active sending or receiving data.</td>
<td>PROF_DRAM_ACTIVE (ID: 1005)</td>
</tr>
<tr>
<td>Engine Activity</td>
<td>Ratio of cycles the fp64 /fp32 /fp16 /HMMA</td>
<td>IMMA pipes are active.</td>
</tr>
</tbody>
</table>
## Feature Overview

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
<th>DCGM Field Name (DCGM_FI_*) and ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVLink Activity</td>
<td>The number of bytes of active NVLink rx or tx data including both header and payload.</td>
<td>DEV_NVLINK_BANDWIDTH_L0</td>
</tr>
<tr>
<td>PCIe Bandwidth</td>
<td>pci__bytes_{rx, tx} - The number of bytes of active PCIe rx or tx data including both header and payload.</td>
<td>PROF_PCIE_[T</td>
</tr>
</tbody>
</table>

Some metrics require multiple passes to be collected and therefore all metrics cannot be collected together. Due to hardware limitations on the Tesla V100, only certain groups of metrics can be read together. For example, SM Activity | SM Occupancy cannot be collected together with Tensor Utilization. To overcome these limitations, DCGM supports automatic multiplexing of metrics by statistically sampling the requested metrics and performing the groupings internally. This may be transparent to users who requested metrics that may not have been able to be collected together.

Profiling of the GPU counters requires administrator privileges starting with Linux drivers 418.43 or later. This is documented here. When using profiling metrics from DCGM, ensure that nv-hostengine is started with superuser privileges.

### 3.10.2. CUDA Test Generator (dcgmproftester)

dcgmproftester is a CUDA load generator. It can be used to generate deterministic CUDA workloads for reading and validating GPU metrics. The tool is shipped as a simple x86_64 Linux binary along with the CUDA kernels compiled to PTX. Customers can use the tool in conjunction with dcgmi to quickly generate a load on the GPU and view the metrics reported by DCGM via dcgmi dmon on stdout.

`dcgmproftester` takes two important arguments as input: `-t` for generating load for a particular metric (for example use 1004 to generate a half-precision matrix-multiply-accumulate for the Tensor Cores) and `-d` for specifying the test duration. Add `--no-dcgm-validation` to let dcgmproftester generate test loads only.

For a list of all the field IDs that can be used to generate specific test loads, see the table in the Profiling Metrics section. The rest of this section includes some examples using the dcgmi command line utility.

Before starting to use dcgmproftester copy the utilities into a local directory. You should find the dcgmproftester binary and DcgmProfTesterKernels.ptx for the PTX of the kernels.

```
$ cp -r /usr/local/dcgm/sdk_samples/dcgmproftester $HOME
```

For example in a console, generate a load for the TensorCores on V100 for 30 seconds. As can be seen, the V100 is able to achieve close to 94TFLOps of FP16 performance using the TensorCores.

```
$ ./dcgmproftester --no-dcgm-validation -t 1004 -d 30
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR: 2048
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT: 80
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR: 98304
```

www.nvidia.com
Data Center GPU Manager
In another console, use the `dcgmi dmon -e` command to view the various performance metrics (streamed to stdout) reported by DCGM as the CUDA workload runs on the GPU. In this example, DCGM reports the GPU activity, TensorCore activity and Memory utilization at a frequency of 1Hz (or 1000ms). As can be seen, the GPU is busy doing work (~99% of Graphics Activity showing that the SMs are busy), with the TensorCore activity pegged to ~80%. Note that `dcgmi` is currently returning the metrics for GPU ID: 0. On a multi-GPU system, you can specify the GPU ID for which DCGM should return the metrics. By default, the metrics are returned for all the GPUs in the system.

```
$ dcgmi dmon -e 1001,1004,1005
# GPU/Sw  GRACT  TENSO  DRAMA
Id
0  0.000  0.000  0.000
0  1.000  0.909  0.213
0  0.994  0.797  0.251
0  0.992  0.796  0.251
0  0.992  0.796  0.250
0  0.992  0.796  0.250
0  0.993  0.797  0.251
0  0.994  0.797  0.251
0  0.994  0.797  0.251
0  0.993  0.796  0.251
```

### 3.10.3. Platform Support

Profiling metrics are currently supported for the following GPU products on Linux x86_64 and POWER (ppc64le) platforms:

- Tesla V100
- Tesla T4

R418.87.00+ Tesla Linux drivers downloaded from the NVIDIA driver downloads site.
4.1. Integrating with DCGM Reader

DcgmReader.py is a class meant to facilitate gathering telemetry from DCGM so that the information can be viewed directly or integrated elsewhere. The sdk_samples directory contains a simple script which uses DcgmReader in DcgmReaderExample.py. The directory also contains other examples of how DcgmReader can be used to integrate into tools such as collectd (dcgm_collectd.py), Prometheus (dcgm_prometheus.py).

This section will walk through two simple ways of using DcgmReader to easily gather information from DCGM.

First, let's imagine that you want to publish some telemetry from DCGM on a message bus such as ZMQ. We can access this through two simple methods, the first of which asks for a dictionary.

All examples here assume that you have DCGM installed and nv-hostengine active locally.

4.1.1. Reading Using the Dictionary

```python
# dictionary_reader_example.py
from DcgmReader import DcgmReader
import zmq
import time

def main():
    dr = DcgmReader()
    context = zmq.Context()
    socket = context.socket(zmq.PUB)
    socket.bind("tcp://*:4096")
    while True:
        GetLatestGpuValuesAsFieldDict() gives us a dictionary that maps
        gpu ids to
        field name to
        the GPU value dictionary. The GPU value dictionary maps each
        the value for that field. This will publish each in the format
        GPU:<GPU ID>:fieldTag=value
```
NOTE: if you prefer to use field ids instead of field tags

```python
data = dr.GetLatestGpuValuesAsFieldDict()

for gpuId in data:
    for fieldTag in data[gpuId]:
        msg = "GPU:%s:%s" % (str(gpuId), fieldTag, val.value)
        self.m_zmqSocket.send("%s %s" % (fieldTag, message))
```

This method permits you to use DcgmReader without learning much about it.

## 4.1.2. Reading Using Inheritance

Using inheritance allows finer-grained controls.

```python
# inheritance_reader_example.py
from DcgmReader import DcgmReader
import zmq
import time
class DcgmPublisher(DcgmReader):
    # Have our constructor also open a ZMQ socket for publishing.
    def __init__(self, port=4096):
        DcgmReader.__init__(self)
        context = zmq.Context()
        self.m_zmqSocket = context.socket(zmq.PUB)
        self.m_zmqSocket.bind("tcp://*:4096")

        Publish the fieldTag as the topic and the message data as
        "GPU:<GPU ID>:fieldTag=value"
        This overrides the method in DcgmReader for what to do with each
        field. If you want additional controls, consider overriding
        CustomDataHandler from the parent class.

        def CustomFieldHandler(self, gpuId, fieldId, fieldTag, val):
            topic = fieldTag
            message = "GPU:%s:%s" % (str(gpuId), fieldTag, val.value)
            self.m_zmqSocket.send("%s %s" % (topic, message))

    def main():
        dp = DcgmPublisher()

        while True:
            dp.Process()
            time.sleep(15)
```

This method grants you more control over DcgmReader.
4.1.3. Completing the Proof of Concept

Either script – when run with DCGM installed and an active nv-hostengine - will publish data on port 4096. The following script will subscribe and print that data.

To run the subscriber from a different host, simply change localhost on line 5 to the IP address of the remote machine that is publishing data from DCGM. As is, the script will connect locally.

```python
# subscriber_example.py
import zmq
port = 4096
context = zmq.Context()
socket = context.socket(zmq.SUB)
# Change localhost to the ip addr of the publisher if remote
socket.connect("tcp://localhost:%d" % (port))
socket.setsockopt(zmq.SUBSCRIBE, '')
while True:
    update = socket.recv()
    topic, message = update.split()
    print message
```

4.1.4. Additional Customization

DcgmReader gathers the following fields by default:

```python
defaultFieldIds = [
    dcgm_fields.DCGM_FI_DRIVER_VERSION,
    dcgm_fields.DCGM_FI_NVML_VERSION,
    dcgm_fields.DCGM_FI_PROCESS_NAME,
    dcgm_fields.DCGM_FI_DEV_POWER_USAGE,
    dcgm_fields.DCGM_FI_DEV_GPU_TEMP,
    dcgm_fields.DCGM_FI_DEV_SM_CLOCK,
    dcgm_fields.DCGM_FI_DEV_GPU_UTIL,
    dcgm_fields.DCGM_FI_DEV_RETIRED_PENDING,
    dcgm_fields.DCGM_FI_DEV_RETIRED_SBE,
    dcgm_fields.DCGM_FI_DEV_RETIRED_DBE,
    dcgm_fields.DCGM_FI_DEV_ECC_SBE_VOL_TOTAL,
    dcgm_fields.DCGM_FI_DEV_ECC_DBE_VOL_TOTAL,
    dcgm_fields.DCGM_FI_DEV_ECC_SBE_AGG_TOTAL,
    dcgm_fields.DCGM_FI_DEV_ECC_DBE_AGG_TOTAL,
    dcgm_fields.DCGM_FI_DEV_FB_TOTAL,
    dcgm_fields.DCGM_FI_DEV_FB_FREE,
    dcgm_fields.DCGM_FI_DEV_FB_USED,
    dcgm_fields.DCGM_FI_DEV_PCIE_REPLAY_COUNTER,
    dcgm_fields.DCGM_FI_DEV_COMPUTE_PIDS,
    dcgm_fields.DCGM_FI_DEV_POWER_VIOLATION,
    dcgm_fields.DCGM_FI_DEV_THERMAL_VIOLATION,
    dcgm_fields.DCGM_FI_DEV_XID_ERRORS,
    dcgm_fields.DCGM_FI_DEV_NVLINK_CRC_FLIT_ERROR_COUNT_TOTAL,
    dcgm_fields.DCGM_FI_DEV_NVLINK_CRC_DATA_ERROR_COUNT_TOTAL,
    dcgm_fields.DCGM_FI_DEV_NVLINK_REPLAY_ERROR_COUNT_TOTAL,
    dcgm_fields.DCGM_FI_DEV_NVLINK_RECOVERY_ERROR_COUNT_TOTAL
]```
There are hundreds more fields that DCGM provides, and you may be interested in monitoring different ones. You can control this in your script by instantiating DcgmReader with parameters:

```python
# custom_fields_example.py
myFieldIds = [
    dcgm_fields.DCGM_FI_DEV_SM_CLOCK,
    dcgm_fields.DCGM_FI_DEV_MEM_CLOCK,
    dcgm_fields.DCGM_FI_DEV_APP_CLOCK,
    dcgm_fields.DCGM_FI_DEV_MEMORY_TEMP,
    dcgm_fields.DCGM_FI_DEV_GPU_TEMP,
    dcgm_fields.DCGM_FI_DEV_POWER_USAGE,
    dcgm_fields.DCGM_FI_DEV_GPU_UTIL,
    dcgm_fields.DCGM_FI_DEV_MEM_COPY_UTIL,
    dcgm_fields.DCGM_FI_DEV_COMPUTE_PIDS
]
...
# In main(), change the instantiation:
dr = DcgmReader(fieldIds=myFieldIds)
...
```

You can control other behaviors of DcgmReader using these additional parameters:

- **hostname**: defaults to localhost. Controls the hostname[:port] where we connected to DCGM.
- **fieldIds**: explained above. Controls the fields we are going to watch and read in DCGM.
- **updateFrequency**: defaults to 10 seconds (specified in microseconds). Controls how often DCGM refreshes each field value.
- **maxKeepAge**: defaults to 1 hour (specified in seconds). Controls how long DCGM keeps data for each of the specified fields.
- **ignoreList**: defaults to an empty array. Specifies field ids that should be retrieved but should be ignored for processing. Usually used for metadata.
- **fieldGroupName**: defaults to dcgm_fieldgroupData. Controls the name that DcgmReader gives to the group of fields we are watching. This is useful for running multiple instances of DcgmReader simultaneously.

### 4.2. Integrating with Prometheus and Grafana

#### 4.2.1. Starting the Prometheus Server

On the server side, configure Prometheus to read (scrape) the data being published by the dcgm_prometheus client. Just add a job to the scrape_configs section of the yaml Prometheus configuration file. See the following section of a working configuration:

```yaml
# A scrape configuration containing exactly one endpoint to scrape:
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'dcgm'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
```
4.2.2. Starting the Prometheus Client

The script `dcgm_prometheus.py` is provided as a fully functional Prometheus client that will publish timeseries data on a port to be read (scraped) by Prometheus. By default, this script will publish common fields read from a DCGM instance running locally every 10 seconds to localhost:8000. Information on controlling what is published, how often, and on what port will be in the section on customization.

On the client side, start this script. It can either connect to a standalone host engine or run on embedded in the script. To start an embedded host engine and check that it is publishing:

```
$ python dcgm_prometheus.py -e
$ curl localhost:8000 > tmp.txt
```
# HELP dcgm_sm_clock DCGM_PROMETHEUS
# TYPE dcgm_sm_clock gauge
dcgm_sm_clock{GpuBusID="00000000:08:00.0",GpuID="1"} 1480.0
dcgm_sm_clock{GpuBusID="00000000:07:00.0",GpuID="0"} 1480.0
dcgm_sm_clock{GpuBusID="00000000:0E:00.0",GpuID="2"} 1480.0
dcgm_sm_clock{GpuBusID="00000000:0F:00.0",GpuID="3"} 1480.0

# HELP dcgm_nvlink_flit_crc_error_count_total DCGM_PROMETHEUS
# TYPE dcgm_nvlink_flit_crc_error_count_total gauge
dcgm_nvlink_flit_crc_error_count_total{GpuBusID="00000000:08:00.0",GpuID="1"} 0.0
dcgm_nvlink_flit_crc_error_count_total{GpuBusID="00000000:0E:00.0",GpuID="2"} 0.0
dcgm_nvlink_flit_crc_error_count_total{GpuBusID="00000000:0F:00.0",GpuID="3"} 0.0

# HELP dcgm_power_usage DCGM_PROMETHEUS
# TYPE dcgm_power_usage gauge
dcgm_power_usage{GpuBusID="00000000:08:00.0",GpuID="1"} 294.969
dcgm_power_usage{GpuBusID="00000000:07:00.0",GpuID="0"} 273.121
dcgm_power_usage{GpuBusID="00000000:0E:00.0",GpuID="2"} 280.484
dcgm_power_usage{GpuBusID="00000000:0F:00.0",GpuID="3"} 281.301

# HELP dcgm_nvlink_data_crc_error_count_total DCGM_PROMETHEUS
# TYPE dcgm_nvlink_data_crc_error_count_total gauge
dcgm_nvlink_data_crc_error_count_total{GpuBusID="00000000:08:00.0",GpuID="1"} 0.0
dcgm_nvlink_data_crc_error_count_total{GpuBusID="00000000:0E:00.0",GpuID="2"} 0.0
dcgm_nvlink_data_crc_error_count_total{GpuBusID="00000000:0F:00.0",GpuID="3"} 0.0
...

The number of GPUs may vary, and the published field ids can be changed through configuration, but the output should conform to the above format.

### 4.2.3. Integrating with Grafana

Grafana offers additional features such as configurable dashboards and integration with Grafana is straightforward. Install and launch Grafana, and then open a browser to http://localhost:3000. The default login / password is admin / admin. After logging in, clock on the Grafana icon and select the ‘Data Sources’ option. Configure the Prometheus server as a data source:
Click ‘Add’ and then create a dashboard using the data that is scraped from the DCGM Prometheus client. Click the Grafana icon again and then Dashboards -> New. There are a lot of ways to customize dashboards; to create a dashboard with graphs, click the 'Graph' option at the top. Select 'Panel Title' and then 'Edit':
Type `dcgm` into the metric name box as shown below, and Grafana will offer you auto-completion options for the DCGM fields you have configured. If this doesn't happen, then the data source wasn't configured correctly, or Prometheus has stopped running.
Use the different tabs to customize the graph as desired. This graph has the metric `dcgm_gpu_temp` selected, and the title changed via the 'General', plus the units set to Celsius via the Axes tab.
4.2.4. Customizing the Prometheus Client

The DCGM Prometheus client can be controlled using command line parameters:

- **-n, --hostname**: specifies the hostname of the DCGM instance we’re querying for data. Default: localhost. Mutually exclusive with -e.
- **-e, --embedded**: start an embedded hostengine from this script instead of connecting to a standalone hostengine. Mutually exclusive with -n.
- **-p, --publish-port**: specifies the port where data is published. Default: 8000. Please note that if you change this you’ll need to change the Prometheus configuration accordingly.
- **-i, --interval**: specifies the interval at which DCGM is queried and data is published in seconds. Default: 10.
- **-l, --ignore-list**: specifies fields queried but not published. Default: DCGM_FI_DEV_PCI_BUSID (57).
- **--log-file**: Specifies the path to a log file. If this is used without –log-level, then only critical information is logged.
- **--log-level**: One of CRITICAL (0), ERROR (1), WARNING (2), INFO (3), or DEBUG (4) to specify what kind of information should be logged. If this is used without –log-file, then the information is logged to stdout.
- **-f, --field-ids**: specifies the list of fields queried and published from DCGM. Default:

```
DCGM_FI_DEV_PCI_BUSID (57)
DCGM_FI_DEV_POWER_USAGE (155)
DCGM_FI_DEV_GPU_TEMP (150)
DCGM_FI_DEV_SM_CLOCK (100)
```
<table>
<thead>
<tr>
<th>Metric Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCGM_FI_DEV_GPU_UTIL</td>
<td>203</td>
</tr>
<tr>
<td>DCGM_FI_DEV_RETIRED_PENDING</td>
<td>392</td>
</tr>
<tr>
<td>DCGM_FI_DEV_RETIRED_SBE</td>
<td>390</td>
</tr>
<tr>
<td>DCGM_FI_DEV_RETIRED_DBE</td>
<td>391</td>
</tr>
<tr>
<td>DCGM_FI_DEV_ECC_SBE_AGG_TOTAL</td>
<td>312</td>
</tr>
<tr>
<td>DCGM_FI_DEV_ECC_DBE_AGG_TOTAL</td>
<td>313</td>
</tr>
<tr>
<td>DCGM_FI_DEV_PCIE_REPLAY_COUNTER</td>
<td>202</td>
</tr>
<tr>
<td>DCGM_FI_DEV_ECC_SBE_VOL_TOTAL</td>
<td>310</td>
</tr>
<tr>
<td>DCGM_FI_DEV_ECC_DBE_VOL_TOTAL</td>
<td>311</td>
</tr>
<tr>
<td>DCGM_FI_DEV_POWER_VIOLATION</td>
<td>240</td>
</tr>
<tr>
<td>DCGM_FI_DEV_THERMAL_VIOLATION</td>
<td>241</td>
</tr>
<tr>
<td>DCGM_FI_DEV_XID_ERRORS</td>
<td>230</td>
</tr>
<tr>
<td>DCGM_FI_DEV_NVLINK_CRC_FLIT_ERROR_COUNT_TOTAL</td>
<td>409</td>
</tr>
<tr>
<td>DCGM_FI_DEV_NVLINK_CRC_DATA_ERROR_COUNT_TOTAL</td>
<td>419</td>
</tr>
<tr>
<td>DCGM_FI_DEV_NVLINK_REPLAY_ERROR_COUNT_TOTAL</td>
<td>429</td>
</tr>
<tr>
<td>DCGM_FI_DEV_NVLINK_RECOVERY_ERROR_COUNT_TOTAL</td>
<td>439</td>
</tr>
</tbody>
</table>

Example Usages

```bash
# Change the DCGM host to one named sel:
$ python dcgm\_prometheus.py -n sel
# Change the port:
$ python dcgm\_prometheus.py -p 10101
# Change the interval
$ python dcgm\_prometheus.py -i 20
# Change the ignore list and publish list:
$ python dcgm\_prometheus.py -l 523 -f 523,310,311,312
```
Chapter 5.
DCGM DIAGNOSTICS

5.1. Overview

The NVIDIA Validation Suite (NVVS) is now called DCGM Diagnostics. As of DCGM v1.5, running NVVS as a standalone utility is now deprecated and all the functionality (including command line options) is available via the DCGM command-line utility (‘dcgmi’). For brevity, the rest of the document may use DCGM Diagnostics and NVVS interchangeably.

5.1.1. DCGM Diagnostics Goals

DCGM Diagnostics are designed to:

1. Provide a system-level tool, in production environments, to assess cluster readiness levels before a workload is deployed.
2. Facilitate multiple run modes:
   - Interactive via an administrator or user in plain text.
   - Scripted via another tool with easily parseable output.
3. Provide multiple test timeframes to facilitate different preparedness or failure conditions:
   - Level 1 tests to use as a readiness metric
   - Level 2 tests to use as an epilogue on failure
   - Level 3 tests to be run by an administrator as post-mortem
4. Integrate the following concepts into a single tool to discover deployment, system software and hardware configuration issues, basic diagnostics, integration issues, and relative system performance.
   - Deployment and Software Issues
     - NVML library access and versioning
     - CUDA library access and versioning
     - Software conflicts
5. Provide troubleshooting help
6. Easily integrate into Cluster Scheduler and Cluster Management applications
7. Reduce downtime and failed GPU jobs

5.1.2. Beyond the Scope of the DCGM Diagnostics

DCGM Diagnostics are not designed to:

1. Provide comprehensive hardware diagnostics
2. Actively fix problems

5.1.3. Dependencies

- DCGM Diagnostics require a NVIDIA Linux driver to be installed. Both the standard display driver and Tesla Recommended Driver will work. You can obtain a driver from http://www.nvidia.com/object/unix.html.
- DCGM Diagnostics require the standard C++ runtime library with GLIBCXX of at least version 3.4.5 or greater.

5.1.4. Supported Products

DCGM Diagnostics support NVIDIA Data center GPUs running on 64-bit Linux (bare metal) operating systems. NVIDIA® Tesla™ Line:

- All Kepler, Maxwell, Pascal, Volta, Turing and NVIDIA Ampere architecture GPUs
5.2. Using DCGM Diagnostics

The various command line options of DCGM Diagnostics are designed to control general execution parameters, whereas detailed changes to execution behavior are contained within the configuration files detailed in the next chapter.

5.2.1. Command line options

The various options for dcgmi diag are as follows:

<table>
<thead>
<tr>
<th>Short option</th>
<th>Long option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s</td>
<td>--statspath</td>
<td>Write the plugin statistics to a given path rather than the current directory.</td>
</tr>
<tr>
<td>-a</td>
<td>--appendLog</td>
<td>When generating a debug logfile, do not overwrite the contents of a current log. Used in conjunction with the -d and -l options.</td>
</tr>
<tr>
<td>-c</td>
<td>--configfile</td>
<td>Specify the configuration file to be used. The default is that no config file is used.</td>
</tr>
<tr>
<td>-i</td>
<td>--gpuList</td>
<td>Comma separated list of the indexes of the GPUs to run NVVS on. This cannot be used in conjunction with -g (group). If neither are specified, then the diagnostic is run on all GPUs.</td>
</tr>
<tr>
<td>-j</td>
<td>--jsonOutput</td>
<td>Provide the output in JSON instead of text.</td>
</tr>
<tr>
<td>-d</td>
<td>--debugLevel</td>
<td>Specify the debug level for the output log. The range is 0 to 5 with 5 being the most verbose.</td>
</tr>
<tr>
<td></td>
<td>--debugLogFile</td>
<td>Specify the logfile for debug information. This will produce an encrypted log file intended to be returned to NVIDIA for post-run analysis after an error.</td>
</tr>
<tr>
<td></td>
<td>--plugin-path</td>
<td>Specify a custom path for the NVVS plugins. The plugins must have the same permissions and owner as the NVVS executable.</td>
</tr>
<tr>
<td>-r</td>
<td>--run</td>
<td>Run a specific test or test suite. Test suites are 1 (short), 2 (medium), or 3 (long). Use quotes to specify individual tests, and use commas to separate multiple tests. For example: -p &quot;diagnostic,sm stress&quot;. This argument is mandatory.</td>
</tr>
<tr>
<td>Short option</td>
<td>Long option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>-p</td>
<td>--parameters</td>
<td>Specify test parameters via the command-line. Use quotes around the argument, and separate the arguments using ':'. For example: -p 'sm stress.test_duration=150;diagnostic.test_duration=150' would set the test duration for the hardware Diagnostic and the SM Stress test to 150 seconds each.</td>
</tr>
<tr>
<td>-p</td>
<td>--parameters</td>
<td>Specifies host of the nv-hostengine we want to connect to. Defaults to localhost.</td>
</tr>
<tr>
<td>--host</td>
<td></td>
<td>Output statistic logs only if a test failure is encountered.</td>
</tr>
<tr>
<td>--statsonfail</td>
<td></td>
<td>Specify which throttling reasons should be ignored. You can provide a comma separated list of reasons. For example, specifying 'HW_SLOWDOWN,SW_THERMAL' would ignore the HW_SLOWDOWN and SW_THERMAL throttling reasons. Alternatively, you can specify the integer value of the ignore bitmask. For the bitmask, multiple reasons may be specified by the sum of their bit masks. For example, specifying '40' would ignore the HW_SLOWDOWN and SW_THERMAL throttling reasons. Valid throttling reasons and their corresponding bitmasks (given in parentheses) are: &quot;HW_SLOWDOWN (8) SW_THERMAL (32) HW_THERMAL (64) HW_POWER_BRAKE (128)&quot;</td>
</tr>
<tr>
<td>--train</td>
<td></td>
<td>Runs the diagnostic iteratively and generate a configure file of the golden values for this system based on the results of multiple runs of the diagnostic tests. This should be used when you wish to restrict what the acceptable standards for the plugins are to ensure that your system conforms to a higher standard. This is intended for systems which have a known, stable configuration. This is a beta feature for the 1.6 release.</td>
</tr>
<tr>
<td>--force</td>
<td></td>
<td>Ignore non-fatal errors and train the diagnostic. For example, if too great coefficient of variance is found among the test runs, then the results would normally be thrown out and no file would...</td>
</tr>
</tbody>
</table>
### Short option | Long option | Description
--- | --- | ---
--force | create the file and ignore these conditions.
--training-iterations | Specifies the number of times the diagnostic tests are each run during training. The default is 4.
--training-variance | Specifies the acceptable coefficient of variance required to trust the data and generate a golden values configuration file in training. The default is 5, which means the coefficient of variance has to be .05 or less to trust the data.
--training-tolerance | The percentage that the golden value should be scaled for tolerance when producing the file. For example, if 10 were specified, then values would be adjusted to tolerate 10% off of the standard before writing the config file so that runs do not have to achieve exactly the same result (or better) as the training runs. Default is 5.
--golden-values-filename | Specify the filename of the golden values file. These files are written to the /tmp dir, and the default name is golden_values.yml.
-v | --verbose | Enable verbose output.
-g | --group | Specifies the list of GPUs on which the diagnostic should run. If no group is specified, the diagnostic is run on all GPUs.
-h | --help | Display usage information and exit.

NVVS has been deprecated. The various options for NVVS are as follows:

### Short option | Long option | Description
--- | --- | ---
--statspath | Write the plugin statistics to a given path rather than the current directory.
-a | --appendLog | When generating a debug logfile, do not overwrite the contents of a current log. Used in conjunction with the -d and -l options.
-c | --config | Specify the configuration file to be used. The default is /etc/nvidia-validation-suite/nvvs.conf
<table>
<thead>
<tr>
<th>Short option</th>
<th>Long option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>--configless</td>
<td>Run NVVS in a configless mode. Executes a &quot;long&quot; test on all supported GPUs.</td>
</tr>
<tr>
<td>-d</td>
<td>--debugLevel</td>
<td>Specify the debug level for the output log. The range is 0 to 5 with 5 being the most verbose. Used in conjunction with the -l flag.</td>
</tr>
<tr>
<td>-g</td>
<td>--listGpus</td>
<td>List the GPUs available and exit. This will only list GPUs that are supported by NVVS.</td>
</tr>
<tr>
<td>-i</td>
<td>--indexes</td>
<td>Comma separated list of indexes to run NVVS on.</td>
</tr>
<tr>
<td>-j</td>
<td>--jsonOutput</td>
<td>Instructs nvvs to format the output as JSON.</td>
</tr>
<tr>
<td>-l</td>
<td>--debugLogFile</td>
<td>Specify the logfile for debug information. This will produce an encrypted log file intended to be returned to NVIDIA for post-run analysis after an error.</td>
</tr>
<tr>
<td></td>
<td>--quiet</td>
<td>No console output given. See logs and return code for errors.</td>
</tr>
<tr>
<td>-p</td>
<td>--pluginpath</td>
<td>Specify a custom path for the NVVS plugins.</td>
</tr>
<tr>
<td>-s</td>
<td>--scriptable</td>
<td>Produce output in a colon-separated, more script-friendly and parseable format.</td>
</tr>
<tr>
<td></td>
<td>--specifiedtest</td>
<td>Run a specific test in a configless mode. Multiple word tests should be in quotes, and if more than one test is specified it should be comma-separated.</td>
</tr>
<tr>
<td></td>
<td>--parameters</td>
<td>Specify test parameters via the command-line. For example: <code>--parameters &quot;sm stress.test_duration=300&quot;</code> would set the test duration for the SM Stress test to 300 seconds.</td>
</tr>
<tr>
<td></td>
<td>--statsonfail</td>
<td>Output statistic logs only if a test failure is encountered.</td>
</tr>
<tr>
<td>-t</td>
<td>--listTests</td>
<td>List the tests available to be executed through NVVS and exit. This will list only the readily loadable tests given the current path and library conditions.</td>
</tr>
<tr>
<td>-v</td>
<td>--verbose</td>
<td>Enable verbose reporting.</td>
</tr>
<tr>
<td></td>
<td>--version</td>
<td>Displays the version information and exits.</td>
</tr>
</tbody>
</table>
5.2.2. Usage Examples

To display the list of GPUs available on the system.

```bash
user@hostname
$ nvvs -g

NVIDIA Validation Suite (version 352.00)

Supported GPUs available:
[0000:01:00.0] -- Tesla K40c
[0000:05:00.0] -- Tesla K20c
[0000:06:00.0] -- Tesla K20c
```

An example "quick" test (explained later) using a custom configuration file.

```bash
user@hostname
$ nvvs -c Tesla_K40c_quick.conf

NVIDIA Validation Suite (version 352.00)

Software
Blacklist ......................................... PASS
NVML Library ...................................... PASS
CUDA Main Library ................................. PASS
CUDA Toolkit Libraries ............................. PASS
Permissions and OS-related Blocks ................. PASS
Persistence Mode .................................. PASS
Environmental Variables ........................... PASS

Hardware
Memory GPU0 ....................................... PASS

Integration
PCIe .............................................. PASS
```

To output an encrypted debug file at the highest debug level to send to NVIDIA for analysis after a problem.

```bash
user@hostname
$ nvvs -c Tesla_K40c_medium.conf -d 5 -l debug.log

NVIDIA Validation Suite (version 352.00)

Software
Blacklist ......................................... PASS
NVML Library ...................................... PASS
CUDA Main Library ................................. PASS
CUDA Toolkit Libraries ............................. PASS
Permissions and OS-related Blocks ................. PASS
Persistence Mode .................................. PASS
Environmental Variables ........................... PASS

Hardware
Memory GPU0 ....................................... PASS

Integration
PCIe .............................................. PASS

*** GPU 0 is running at PCI link width 8X, which is below the minimum allowed link width of 16X (parameter: min_pci_width)"
```

The output file, debug.log would then be returned to NVIDIA.
5.2.3. Configuration file

The NVVS configuration file is a YAML-formatted (e.g. human-readable JSON) text file with three main stanzas controlling the various tests and their execution.

The general format of a configuration file consists of:

```
---
globals:
  key1: value
  key2: value

test_suite_name:
  - test_class_name1:
    test_name1:
      key1: value
      key2: value
    subtests:
      subtest_name1:
        key1: value
        key2: value
  test_name2:
    key1: value
    key2: value
  - test_class_name2:
    test_name3:
      key1: value
      key2: value

gpus:
  - gpuset: name
    properties:
      key1: value
      key2: value
    tests:
      name: test_suite_name
```

There are three distinct sections: `globals`, `test_suite_name`, and `gpus` each with its own subsection of parameters and as is with any YAML document, indentation is important thus if errors are generated from your own configuration files please refer to this example for indentation reference.

5.2.4. Global parameters

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logfile</td>
<td>String</td>
<td>The prefix for all detailed test data able to be used for post-processing.</td>
</tr>
<tr>
<td>logfile_type</td>
<td>String</td>
<td>Can be <code>json</code>, <code>text</code>, or <code>binary</code>. Used in conjunction with the <code>logfile</code> global parameter. Default is <code>JSON</code>.</td>
</tr>
<tr>
<td>scriptable</td>
<td>Boolean</td>
<td>Accepts <code>true</code>, or <code>false</code>. Produces a script-friendly, colon-separated</td>
</tr>
</tbody>
</table>
5.2.5. GPU parameters

The `gpus` stanza may consist of one or more `gpusets` which will each match zero or more GPUs on the system based on their `properties`(a match of zero will produce an error).

GPUs are matched based on the following criteria with their configuration file keywords in parenthesis:

- Name of the GPU, i.e. Tesla K40c (`name`)
- Brand of the GPU, i.e. Tesla (`brand`)
- A comma separated list of indexes (`index`)
- The GPU UUID (`uuid`)
- or the PCIe Bus ID (`busid`)

The matching rules are based off of exclusion. First, the list of supported GPUs is taken and if no `properties` tag is given then all GPUs will be used in the test. Because a UUID or PCIe Bus ID can only match a single GPU, if those properties are given then only that GPU will be used if found. The remaining properties, `index`, `brand`, and `name` work in an "AND" fashion such that, if specified, the result must match at least one GPU on the system for a test to be performed.

For example, if `name` is set to "Tesla K40c" and `index` is set to "0" NVVS will error if index 0 is not a Tesla K40c. By specifying both `brand` and `index` a user may limit a test to specific "Tesla" cards for example. In this version of NVVS, all matching GPUs must be homogeneous.

The second identifier for a `gpuset` is `tests`. This parameter specifies either the suite of tests that a user wishes to run or the test itself.

At present the following suites are available:

- **Quick** -- meant as a pre-run sanity check to ensure that the GPUs are ready for a job. Currently runs the Deployment tests described in the next chapter.
- **Medium** -- meant as a quick, post-error check to make sure that nothing very obvious such as ECC enablement or double-bit errors have occurred on a GPU.
Currently runs the Deployment, Memory/Hardware, and PCIe/Bandwidth tests. The Hardware tests are meant to be relatively short to find obvious issues.

- **Long** -- meant as a more extensive check to find potential power and/or performance problems within a cluster. Currently runs an extensive test that involves Deployment, Memory/Hardware, PCI/Bandwidth, Power, Stress, and Memory Bandwidth. The Hardware tests will run in a longer-term iterative mode that are meant to try and capture transient failures as well as obvious issues.

An individual test can also be specified. Currently the keywords are: **Memory, Diagnostic, Targeted Stress, Targeted Power, PCIe, SM Stress, and Memory Bandwidth**. Please see the "custom" section in the next subchapter to configure and tweak the parameters when this method is used.

### 5.2.6. Test Parameters

The format of the NVVS configuration file is designed for extensibility. Each test suite above can be customized in a number of ways described in detail in the following chapter for each test. Individual tests belong to a specific class of functionality which, when wanting to customize specific parameters, must also be specified.

The classes and the respective tests they perform are as follows:

<table>
<thead>
<tr>
<th>Class name</th>
<th>Tests</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>Deployment</td>
<td>Checks for various runtime libraries, persistence mode, permissions, environmental variables, and blacklisted drivers.</td>
</tr>
<tr>
<td>Hardware</td>
<td>Diagnostic</td>
<td>Execute a series of hardware diagnostics meant to exercise a GPU or GPUs to their factory specified limits.</td>
</tr>
<tr>
<td>Integration</td>
<td>PCIe</td>
<td>Test host to GPU, GPU to host, and P2P (if possible) bandwidth. P2P between GPUs occurs over NVLink (if possible) or PCIe.</td>
</tr>
<tr>
<td>Stress</td>
<td>Targeted Stress</td>
<td>Sustain a specific targeted stress level for a given amount of time.</td>
</tr>
<tr>
<td></td>
<td>Targeted Power</td>
<td>Sustain a specific targeted power level for a given amount of time.</td>
</tr>
<tr>
<td></td>
<td>SM Stress</td>
<td>Sustain a workload on the Streaming Multiprocessors (SMs) of the GPU for a given amount of time.</td>
</tr>
<tr>
<td></td>
<td>Memory Bandwidth</td>
<td>Verify that a certain memory bandwidth can be achieved on the framebuffer of the GPU.</td>
</tr>
</tbody>
</table>
Some tests also have subtests that can be enabled by using the `subtests` keyword and then hierarchically adding the subtest parameters desired beneath. An example would be the PCIe Bandwidth test which may have a section that looks similar to this:

```yaml
long:
  - integration:
    pcie:
      test_unpinned: false
    subtests:
      h2d_d2h_single_pinned:
        min_bandwidth: 20
        min_pci_width: 16
```

When only a specific test is given in the GPU set portion of the configuration file, both the suite and class of the test are `custom`. For example:

```yaml
%YAML 1.2
---
globals:
  logfile: nvvs.log
custom:
  - custom:
    targeted stress:
      test_duration: 60
gpus:
  - gpuset: all_K40c
    properties:
      name: Tesla K40c
tests:
  - name: targeted stress
```

## 5.3. Overview of Plugins

The NVIDIA Validation Suite consists of a series of plugins that are each designed to accomplish a different goal.

### 5.3.1. Deployment Plugin

The deployment plugin's purpose is to verify the compute environment is ready to run Cuda applications and is able to load the NVML library.

**Preconditions**

- `LD_LIBRARY_PATH` must include the path to the cuda libraries, which for version X.Y of Cuda is normally `/usr/local/cuda-X.Y/lib64`, which can be set by running `export LD_LIBRARY_PATH=/usr/local/cuda-X.Y/lib64`
- The linux nouveau driver must not be running, and should be blacklisted since it will conflict with the nvidia driver
Configuration Parameters

None at this time.

Stat Outputs

None at this time.

Failure

The plugin will fail if:

- The corresponding device nodes for the target GPU(s) are being blocked by the operating system (e.g. cgroups) or exist without r/w permissions for the current user.
- The NVML library libnvidia-ml.so cannot be loaded
- The Cuda runtime libraries cannot be loaded
- The nouveau driver is found to be loaded
- Any pages are pending retirement on the target GPU(s)
- Any other graphics processes are running on the target GPU(s) while the plugin runs

5.3.2. PCIe - GPU Bandwidth Plugin

The GPU bandwidth plugin’s purpose is to measure the bandwidth and latency to and from the GPUs and the host.

Preconditions

None

Sub Tests

The plugin consists of several self-tests that each measure a different aspect of bandwidth or latency. Each subtest has either a pinned/unpinned pair or a p2p enabled/p2p disabled pair of identical tests. Pinned/unpinned tests use either pinned or unpinned memory when copying data between the host and the GPUs.

This plugin will use NvLink to communicate between GPUs when possible. Otherwise, communication between GPUs will occur over PCIe.

Each sub test is represented with a tag that is used both for specifying configuration parameters for the sub test and for outputting stats for the sub test. P2p enabled/p2p disabled tests enable or disable GPUs on the same card talking to each other directly rather than through the PCIe bus.
<table>
<thead>
<tr>
<th>Sub Test Tag</th>
<th>Pinned/Unpinned</th>
<th>P2P Enabled/P2P Disabled</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2d_d2h_single_pinned</td>
<td>Pinned</td>
<td></td>
<td>Device &lt;-&gt; Host Bandwidth, one GPU at a time</td>
</tr>
<tr>
<td>h2d_d2h_single_unpinned</td>
<td>Unpinned</td>
<td></td>
<td>Device &lt;-&gt; Host Bandwidth, one GPU at a time</td>
</tr>
<tr>
<td>h2d_d2h_concurrent_pinned</td>
<td>Pinned</td>
<td></td>
<td>Device &lt;-&gt; Host Bandwidth, all GPUs concurrently</td>
</tr>
<tr>
<td>h2d_d2h_concurrent_unpinned</td>
<td>Unpinned</td>
<td></td>
<td>Device &lt;-&gt; Host Bandwidth, all GPUs concurrently</td>
</tr>
<tr>
<td>h2d_d2h_latency_pinned</td>
<td>Pinned</td>
<td></td>
<td>Device &lt;-&gt; Host Latency, one GPU at a time</td>
</tr>
<tr>
<td>h2d_d2h_latency_unpinned</td>
<td>Unpinned</td>
<td></td>
<td>Device &lt;-&gt; Host Latency, one GPU at a time</td>
</tr>
<tr>
<td>p2p_bw_p2p_enabled</td>
<td>P2P Enabled</td>
<td></td>
<td>Device &lt;-&gt; Device bandwidth one GPU pair at a time</td>
</tr>
<tr>
<td>p2p_bw_p2p_disabled</td>
<td>P2P Disabled</td>
<td></td>
<td>Device &lt;-&gt; Device bandwidth one GPU pair at a time</td>
</tr>
<tr>
<td>p2p_bw_concurrent_p2p_enabled</td>
<td>P2P Enabled</td>
<td></td>
<td>Device &lt;-&gt; Device bandwidth, concurrently, focusing on bandwidth between GPUs between GPUs likely to be directly connected to each other -&gt; for each (index / 2) and (index / 2)+1</td>
</tr>
<tr>
<td>p2p_bw_concurrent_p2p_disabled</td>
<td>P2P Disabled</td>
<td></td>
<td>Device &lt;-&gt; Device bandwidth, concurrently, focusing on bandwidth between GPUs between GPUs likely to be directly connected to each other -&gt; for each (index / 2) and (index / 2)+1</td>
</tr>
<tr>
<td>1d_exch_bw_p2p_enabled</td>
<td>P2P Enabled</td>
<td></td>
<td>Device &lt;-&gt; Device bandwidth, concurrently, focusing on bandwidth between gpus, every GPU either sending to the gpu with the index higher than itself</td>
</tr>
<tr>
<td>Sub Test Tag</td>
<td>Pinned/Unpinned</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------</td>
<td>------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2P Enabled/P2P Disabled</td>
<td>(l2r) or to the gpu with the index lower than itself (r2l)</td>
<td></td>
</tr>
<tr>
<td>1d_exch_bw_p2p_disabled</td>
<td>P2P Disabled</td>
<td>Device &lt;-&gt; Device bandwidth, concurrently, focusing on bandwidth between gpus, every GPU either sending to the gpu with the index higher than itself (l2r) or to the gpu with the index lower than itself (r2l)</td>
<td></td>
</tr>
<tr>
<td>p2p_latency_p2p_enabled</td>
<td>P2P Enabled</td>
<td>Device &lt;-&gt; Device Latency, one GPU pair at a time</td>
<td></td>
</tr>
<tr>
<td>p2p_latency_p2p_disabled</td>
<td>P2P Disabled</td>
<td>Device &lt;-&gt; Device Latency, one GPU pair at a time</td>
<td></td>
</tr>
</tbody>
</table>

**Configuration Parameters- Global**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_pinned</td>
<td>Bool</td>
<td>True</td>
<td>True/False</td>
<td>Include subtests that test using pinned memory.</td>
</tr>
<tr>
<td>test_unpinned</td>
<td>Bool</td>
<td>True</td>
<td>True/False</td>
<td>Include subtests that test using unpinned memory.</td>
</tr>
<tr>
<td>test_p2p_on</td>
<td>Bool</td>
<td>True</td>
<td>True/False</td>
<td>Include subtests that require peer to peer (P2P) memory transfers between cards to occur.</td>
</tr>
<tr>
<td>test_p2p_off</td>
<td>Bool</td>
<td>True</td>
<td>True/False</td>
<td>Include subtests that do not require peer to peer (P2P) memory transfers between cards to occur.</td>
</tr>
</tbody>
</table>
### Parameter Table

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_pcie_replays</td>
<td>Float</td>
<td>80.0</td>
<td>1.0 - 1000000.0</td>
<td>Maximum number of PCIe replays to allow per GPU for the duration of this plugin. This is based on an expected replay rate &lt;8 per minute for PCIe Gen 3.0, assuming this plugin will run for less than a minute and allowing 10x as many replays before failure.</td>
</tr>
</tbody>
</table>

### Configuration Parameters - Sub Test

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Default (Range)</th>
<th>Affected Sub Tests</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_bandwidth</td>
<td>Null (0.0 - 100.0)</td>
<td>h2d_d2h_single_pinned, h2d_d2h_single_unpinned, h2d_d2h_concurrent_pinned, h2d_d2h_concurrent_unpinned</td>
<td>Minimum bandwidth in GB/s that must be reached for this sub-test to pass.</td>
</tr>
<tr>
<td>max_latency</td>
<td>100,000.0 (0.0 - 1,000,000.0)</td>
<td>h2d_d2h_latency_pinned, h2d_d2h_latency_unpinned</td>
<td>Latency in microseconds that cannot be exceeded for this sub-test to pass.</td>
</tr>
<tr>
<td>min_pci_generation</td>
<td>1.0 (1.0 - 3.0)</td>
<td>h2d_d2h_single_pinned, h2d_d2h_single_unpinned</td>
<td>Minimum allowed PCI generation that the GPU must be at or exceed for this sub-test to pass.</td>
</tr>
<tr>
<td>min_pci_width</td>
<td>1.0 (1.0 - 16.0)</td>
<td>h2d_d2h_single_pinned, h2d_d2h_single_unpinned</td>
<td>Minimum allowed PCI width that the GPU must be at or exceed for this sub-test to pass. For example, 16x = 16.0.</td>
</tr>
</tbody>
</table>

### Stat Outputs - Global

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Stat Scope</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcie_replay_count</td>
<td>GPU</td>
<td>Float</td>
<td>The per second reading of PCIe replays that have</td>
</tr>
</tbody>
</table>
### Stat Outputs - Sub Test

Stats for the GPU Bandwidth test are also output on a test by test basis, using the sub test name as the group name key. The following stats sections are organized by sub test.

**h2d_d2h_single_pinned/h2d_d2h_single_unpinned**

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_h2d</td>
<td>Float</td>
<td>Average bandwidth from host to device for device N</td>
</tr>
<tr>
<td>N_d2h</td>
<td>Float</td>
<td>Average bandwidth from device to host for device N</td>
</tr>
<tr>
<td>N_bidir</td>
<td>Float</td>
<td>Average bandwidth from device to host and host to device at the same time for device N</td>
</tr>
</tbody>
</table>

**h2d_d2h_concurrent_pinned/h2d_d2h_concurrent_unpinned**

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_h2d</td>
<td>Float</td>
<td>Average bandwidth from host to device for device N</td>
</tr>
<tr>
<td>N_d2h</td>
<td>Float</td>
<td>Average bandwidth from device to host for device N</td>
</tr>
<tr>
<td>N_bidir</td>
<td>Float</td>
<td>Average bandwidth from device to host and host to device at the same time for device N</td>
</tr>
<tr>
<td>sum_bidir</td>
<td>Float</td>
<td>Sum of the average bandwidth from device to host and host to device for all devices.</td>
</tr>
<tr>
<td>sum_h2d</td>
<td>Float</td>
<td>Sum of the average bandwidth from host to device for all devices.</td>
</tr>
<tr>
<td>sum_d2h</td>
<td>Float</td>
<td>Sum of the average bandwidth from device to host for all devices.</td>
</tr>
</tbody>
</table>
### h2d_d2h_latency_pinned/h2d_d2h_latency_unpinned

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_h2d</td>
<td>Float</td>
<td>Average latency from host to device for device N</td>
</tr>
<tr>
<td>N_d2h</td>
<td>Float</td>
<td>Average latency from device to host for device N</td>
</tr>
<tr>
<td>N_bidir</td>
<td>Float</td>
<td>Average latency from device to host and host to device at the same time for device N</td>
</tr>
</tbody>
</table>

### p2p_bw_p2p_enabled/p2p_bw_p2p_disabled

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_M_onedir</td>
<td>Float</td>
<td>Average bandwidth from device N to device M, copying one direction at a time.</td>
</tr>
<tr>
<td>N_M_bidir</td>
<td>Float</td>
<td>Average bandwidth from device N to device M, copying both directions at the same time.</td>
</tr>
</tbody>
</table>

### p2p_bw_concurrent_p2p_enabled/p2p_bw_concurrent_p2p_disabled

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l2r_N_M</td>
<td>Float</td>
<td>Average bandwidth from device N to device M</td>
</tr>
<tr>
<td>r2l_N_M</td>
<td>Float</td>
<td>Average bandwidth from device M to device N</td>
</tr>
<tr>
<td>bidir_N_M</td>
<td>Float</td>
<td>Average bandwidth from device M to device N, copying concurrently</td>
</tr>
<tr>
<td>r2l_sum</td>
<td>Float</td>
<td>Sum of average bandwidth for all right (M) to left (N) copies</td>
</tr>
<tr>
<td>r2l_sum</td>
<td>Float</td>
<td>Sum of average bidirectional bandwidth for all right (M) to left (N) and left to right copies copies</td>
</tr>
</tbody>
</table>

### 1d_exch_bw_p2p_enabled/1d_exch_bw_p2p_disabled
### 5.3.3. Memory Bandwidth Plugin

The purpose of the Memory Bandwidth plugin is to validate that the bandwidth of the framebuffer of the GPU is above a preconfigured threshold.

#### Preconditions

*This plugin only runs on GV100 GPUs at this time.*

#### Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimum_bandwidth</td>
<td>Float</td>
<td>Differs per GPU</td>
<td>1.0 - 1000000.0</td>
<td>Minimum framebuffer bandwidth threshold that must be achieved in order to pass</td>
</tr>
</tbody>
</table>
### Parameter Name | Type | Default | Value Range | Description
---|---|---|---|---
| | | | | this test in MB/sec.

### Stat Outputs

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Stat Scope</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>power_usage</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second power usage of each GPU in watts. Note that for multi-GPU boards, each GPU gets a fraction of the power budget of the board.</td>
</tr>
<tr>
<td>memory_clock</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second clock rate of the GPU’s memory in MHZ</td>
</tr>
<tr>
<td>nvml_events</td>
<td>GPU</td>
<td>Time series Int64</td>
<td>Any events that were read with nvmlEventSetWait during the test and the timestamp it was read it.</td>
</tr>
</tbody>
</table>

### Failure

The plugin will fail if:

- the minimum bandwidth specified in `minimum_bandwidth` cannot be achieved.
- If GPU double bit ECC errors occur or the configured amount of SBE errors occur.
- If a critical XID occurs

### 5.3.4. SM Stress Plugin

The SM performance plugin’s purpose is to bring the Streaming Multiprocessors (SMs) of the target GPU(s) to a target performance level in gigaflops by doing large matrix multiplications using cublas. Unlike the Targeted Stress plugin, the SM stress plugin does not copy the source arrays to the GPU before every matrix multiplication. This allows the SM performance plugin’s performance to not be capped by device to host bandwidth. The plugin calculates how many matrix operations per second are necessary to achieve the configured performance target and fails if it cannot achieve that target.

This plugin should be used to watch for thermal, power and related anomalies while the target GPU(s) are under realistic load conditions. By setting the appropriate parameters...
a user can ensure that all GPUs in a node or cluster reach desired performance levels. Further analysis of the generated stats can also show variations in the required power, clocks or temperatures to reach these targets, and thus highlight GPUs or nodes that are operating less efficiently.

**Preconditions**

*None*

**Configuration Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_duration</td>
<td>Float</td>
<td>90.0</td>
<td>30.0 - 3600.0</td>
<td>How long the performance test should run for in seconds. It is recommended to set this to at least 30 seconds for performance to stabilize.</td>
</tr>
<tr>
<td>temperature_max</td>
<td>Float</td>
<td>Null</td>
<td>30.0 - 120.0</td>
<td>The maximum temperature in °C the card is allowed to reach during the test. Note that this check is disabled by default. Use nvidia-smi -q to see the normal temperature limits of your device.</td>
</tr>
<tr>
<td>target_stress</td>
<td>Float</td>
<td>Null</td>
<td>SKU dependent</td>
<td>The maximum relative performance each card will attempt to achieve.</td>
</tr>
</tbody>
</table>

**Stat Outputs**

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Stat Scope</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>power_usage</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second power usage of each GPU in</td>
</tr>
<tr>
<td>Stat Name</td>
<td>Stat Scope</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>watts</td>
<td></td>
<td></td>
<td>Note that for multi-GPU boards, each GPU gets a fraction of the power budget of the board.</td>
</tr>
<tr>
<td>graphics_clock</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second clock rate of each GPU in MHZ</td>
</tr>
<tr>
<td>memory_clock</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second clock rate of the GPU’s memory in MHZ</td>
</tr>
<tr>
<td>nvml_events</td>
<td>GPU</td>
<td>Time series Int64</td>
<td>Any events that were read with nvmlEventSetWait - including single or double bit errors or XID errors - during the test.</td>
</tr>
<tr>
<td>power_violation</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Percentage of time this GPU was violating power constraints.</td>
</tr>
<tr>
<td>gpu_temperature</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second temperature of the GPU in degrees C</td>
</tr>
<tr>
<td>perf_gflops</td>
<td>GPU</td>
<td>Time series Float</td>
<td>The per second reading of average gflops since the test began.</td>
</tr>
<tr>
<td>flops_per_op</td>
<td>GPU</td>
<td>Float</td>
<td>Flops (floating point operations) per operation queued to the GPU stream. One operation is one call to cublasSgemm or cublasDgemm</td>
</tr>
<tr>
<td>bytes_copied_per_op</td>
<td>GPU</td>
<td>Float</td>
<td>How many bytes are copied to + from the GPU per operation</td>
</tr>
<tr>
<td>Stat Name</td>
<td>Stat Scope</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>num_cuda_streams</td>
<td>GPU</td>
<td>Float</td>
<td>How many cuda streams were used per gpu to queue operations to the GPUs</td>
</tr>
<tr>
<td>try_ops_per_sec</td>
<td>GPU</td>
<td>Float</td>
<td>Calculated number of ops/second necessary to achieve target gigaflops</td>
</tr>
</tbody>
</table>

**Failure**

The plugin will fail if:

- The GPU temperature exceeds a user-provided threshold.
- If thermal violation counters increase
- If the target performance level cannot be reached
- If GPU double bit ECC errors occur or the configured amount of SBE errors occur.
- If a critical XID occurs

### 5.3.5. Hardware Diagnostic Plugin

The HW Diagnostic Plugin is designed to identify HW failures on GPU silicon and board-level components, extending out to the PCIE and NVLINK interfaces. It is not intended to identify HW or system level issues beyond the NVIDIA-provided HW. Nor is it intended to identify SW level issues above the HW, e.g. in the NVIDIA driver stack. The plugin runs a series of tests that target GPU computational correctness, GDDR/HBM memory resiliency, GPU and SRAM high power operation, SM stress and NVLINK/PCIE correctness. The plugin can run with several combinations of tests corresponding to medium and long NVVS operational modes. This plugin will take about three minutes to execute.

The plugin produces a simple pass/fail output. A failing output means that a potential HW issue has been found. However, the NVVS HW Diagnostic Plugin is not by itself a justification for GPU RMA. Any failure in the plugin should be followed by execution of the full NVIDIA Field Diagnostic after the machine has been taken offline. Only a failure of the Field Diagnostic tool constitutes grounds for RMA. Since the NVVS HW Diagnostic Plugin is a functional subset of the Field Diagnostic a failure in the plugin is a strong indicator of a future Field Diagnostic failure.

**Preconditions**

- No other GPU processes can be running.
## Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_duration</td>
<td>Float</td>
<td>180.0</td>
<td>30.0 - 3600.0</td>
<td>How long the performance test should run for in seconds. It is recommended to set this to at least 30 seconds to make sure you actually get some stress from the test.</td>
</tr>
<tr>
<td>use_doubles</td>
<td>Boolean</td>
<td>False</td>
<td>True or False</td>
<td>If set to true, tells the test to use double-point precision in its calculations. By default, it is false and the test will use floating point precision.</td>
</tr>
<tr>
<td>temperature_max</td>
<td>Float</td>
<td>100.0</td>
<td>30.0 - 120.0</td>
<td>The maximum temperature in C that the card is allowed to reach during the test. Use nvidia-smi -q to see the normal temperature limits of your device.</td>
</tr>
</tbody>
</table>

## Stat Outputs

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Stat Scope</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>power_usage</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second power usage of each GPU in watts. Note that for multi-GPU boards, each GPU gets a fraction of the power budget of the board.</td>
</tr>
</tbody>
</table>
### DCGM Diagnostics

#### Stat Name | Stat Scope | Type | Description
--- | --- | --- | ---
graphics_clock | GPU | Time series Float | Per second clock rate of each GPU in MHZ
memory_clock | GPU | Time series Float | Per second clock rate of the GPU’s memory in MHZ
nvml_events | GPU | Time series Int64 | Any events that were read with nvmlEventSetWait - including single or double bit errors or XID errors - during the test.
powerViolation | GPU | Time series Float | Percentage of time this GPU was violating power constraints.
gpu_temperature | GPU | Time series Float | Per second temperature of the GPU in degrees C
thermalViolation | GPU | Time series Float | Percentage of time this GPU was violating thermal constraints.
perf_gflops | GPU | Time Series Float | The per second reading of average gflops since the test began.

#### Failure

The plugin will fail if:

- The corresponding device nodes for the target GPU(s) are being blocked by the operating system (e.g. cgroups) or exist without r/w permissions for the current user.
- Other GPU processes are running
- A hardware issue has been detected. *This is not an RMA actionable failure but rather an indication that more investigation is required.*
- The temperature reaches unacceptable levels during the test.
- If GPU double bit ECC errors occur or the configured amount of SBE errors occur.
- If a critical XID occurs
5.3.6. Targeted Stress Plugin

The Targeted Stress plugin’s purpose is to bring the GPU to a target performance level in gigaflops by doing large matrix multiplications using cublas. The plugin calculates how many matrix operations per second are necessary to achieve the configured performance target and fails if it cannot achieve that target.

This plugin should be used to watch for thermal, power and related anomalies while the target GPU(s) are under realistic load conditions. By setting the appropriate parameters a user can ensure that all GPUs in a node or cluster reach desired performance levels. Further analysis of the generated stats can also show variations in the required power, clocks or temperatures to reach these targets, and thus highlight GPUs or nodes that are operating less efficiently.

Preconditions

None

Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_duration</td>
<td>Float</td>
<td>120.0</td>
<td>30.0 - 3600.0</td>
<td>How long the Targeted Stress test should run for in seconds. It is recommended to set this to at least 30 seconds for performance to stabilize.</td>
</tr>
<tr>
<td>temperature_max</td>
<td>Float</td>
<td>Null</td>
<td>30.0 - 120.0</td>
<td>The maximum temperature in °C the card is allowed to reach during the test. Note that this check is disabled by default. Use nvidia-smi -q to see the normal temperature limits of your device.</td>
</tr>
<tr>
<td>target_stress</td>
<td>Float</td>
<td>Null</td>
<td>SKU dependent</td>
<td>The maximum relative stress each</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Type</td>
<td>Default</td>
<td>Value Range</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------</td>
<td>---------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>max_pcie_replays</td>
<td>Float</td>
<td>160.0</td>
<td>1.0 - 1000000.0</td>
<td>Maximum number of PCIe replays to allow per GPU for the duration of this plugin. This is based on an expected replay rate &lt;8 per minute for PCIe Gen 3.0, assuming this plugin will run for 2 minutes (configurable) and allowing 10x as many replays before failure.</td>
</tr>
</tbody>
</table>

**Stat Outputs**

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Stat Scope</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>power_usage</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second power usage of each GPU in watts. Note that for multi-GPU boards, each GPU gets a fraction of the power budget of the board.</td>
</tr>
<tr>
<td>graphics_clock</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second clock rate of each GPU in MHZ</td>
</tr>
<tr>
<td>memory_clock</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second clock rate of the GPU’s memory in MHZ</td>
</tr>
<tr>
<td>nvml_events</td>
<td>GPU</td>
<td>Time series Int64</td>
<td>Any events that were read with nvmlEventSetWait during the test and</td>
</tr>
<tr>
<td>Stat Name</td>
<td>Stat Scope</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>timestamp</td>
<td></td>
<td></td>
<td>the timestamp it was read.</td>
</tr>
<tr>
<td>power_violation</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Percentage of time this GPU was violating power constraints.</td>
</tr>
<tr>
<td>gpu_temperature</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second temperature of the GPU in degrees C.</td>
</tr>
<tr>
<td>perf_gflops</td>
<td>GPU</td>
<td>Time series Float</td>
<td>The per second reading of average gflops since the test began.</td>
</tr>
<tr>
<td>flops_per_op</td>
<td>GPU</td>
<td>Float</td>
<td>Flops (floating point operations) per operation queued to the GPU stream. One operation is one call to cublasSgemm or cublasDgemm</td>
</tr>
<tr>
<td>bytes_copied_per_op</td>
<td>GPU</td>
<td>Float</td>
<td>How many bytes are copied to + from the GPU per operation</td>
</tr>
<tr>
<td>num_cuda_streams</td>
<td>GPU</td>
<td>Float</td>
<td>How many cuda streams were used per gpu to queue operations to the GPUs.</td>
</tr>
<tr>
<td>try_ops_per_sec</td>
<td>GPU</td>
<td>Float</td>
<td>Calculated number of ops/second necessary to achieve target gigaflops.</td>
</tr>
<tr>
<td>pcie_replay_count</td>
<td>GPU</td>
<td>Float</td>
<td>The per second reading of PCIe replays that have occurred since the start of the Targeted Stress plugin.</td>
</tr>
</tbody>
</table>
Failure

The plugin will fail if:

- The GPU temperature exceeds a user-provided threshold.
- If temperature violation counters increase
- If the target stress level cannot be reached
- If GPU double bit ECC errors occur or the configured amount of SBE errors occur.
- If the number of PCIe retransmits exceeds a user-provided threshold.
- A critical XID occurs

5.3.7. Power Plugin

The purpose of the power plugin is to bring the GPUs to a preconfigured power level in watts by gradually increasing the compute load on the GPUs until the desired power level is achieved. This verifies that the GPUs can sustain a power level for a reasonable amount of time without problems like thermal violations arising.

Preconditions

None

Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Default</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_duration</td>
<td>Float</td>
<td>120.0</td>
<td>30.0 - 3600.0</td>
<td>How long the performance test should run for in seconds. It is recommended to set this to at least 60 seconds for performance to stabilize.</td>
</tr>
<tr>
<td>temperature_max</td>
<td>Float</td>
<td>Null</td>
<td>30.0 - 120.0</td>
<td>The maximum temperature in C the card is allowed to reach during the test. Note that this check is disabled by default. Use nvidia-smi -q to see the normal temperature</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Type</td>
<td>Default</td>
<td>Value Range</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>---------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>target_power</td>
<td>Float</td>
<td>Differs per GPU</td>
<td>Differs per GPU. Defaults to TDP - 1 watt.</td>
<td>What power level in wattage we should try to maintain. If this is set to greater than the enforced power limit of the GPU, then we will try to power cap the device.</td>
</tr>
</tbody>
</table>

**Stat Outputs**

<table>
<thead>
<tr>
<th>Stat Name</th>
<th>Stat Scope</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>power_usage</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second power usage of each GPU in watts. Note that for multi-GPU boards, each GPU gets a fraction of the power budget of the board.</td>
</tr>
<tr>
<td>graphics_clock</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second clock rate of each GPU in MHZ</td>
</tr>
<tr>
<td>memory_clock</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second clock rate of the GPU’s memory in MHZ</td>
</tr>
<tr>
<td>nvml_events</td>
<td>GPU</td>
<td>Time series Int64</td>
<td>Any events that were read with nvmlEventSetWait during the test and the timestamp it was read it.</td>
</tr>
<tr>
<td>power_violation</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Percentage of time this GPU was violating power constraints.</td>
</tr>
<tr>
<td>Stat Name</td>
<td>Stat Scope</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gpu_temperature</td>
<td>GPU</td>
<td>Time series Float</td>
<td>Per second temperature of the GPU in degrees C</td>
</tr>
</tbody>
</table>

**Failure**

The plugin will fail if:

- The GPU temperature exceeds a user-provided threshold.
- If temperature violation counters increase
- If the target performance level cannot be reached
- If GPU double bit ECC errors occur or the configured amount of SBE errors occur.
- If a critical XID occurs

### 5.4. Test Output

The output of tests can be collected by setting the "logfile" global parameter which represents the prefix for the detailed outputs produced by each test. The default type of output is JSON but text and binary outputs are available as well. The latter two are meant more for parsing and direct reading by custom consumers respectively so this portion of the document will focus on the JSON output.

#### 5.4.1. JSON Output

The JSON output format is keyed based off of the "stats" keys given in each test overview from Chapter 3. These standard JSON files can be processed in any number of ways but two example Python scripts have been provided to aid in visualization in the default installation directory. The first is a JSON to comma-separated value script (json2csv.py) which can be used to import key values in to a graphing spreadsheet. Proper usage would be:

```
user@hostname
$ python json2csv.py -i stats_targeted_performance.json -o stats.csv -k gpu_temperature,power_usage
```

Also provided is an example Python script that uses the pygal library to generate readily viewable scalar vector graphics charts (json2svg.py), able to be opened in any browser. Proper usage would be:

```
user@hostname
$ python json2svg.py -i stats_targeted_performance.json -o stats.svg -k gpu_temperature,power_usage
```
Chapter 6.
DCGM MODULARITY

DCGM 1.5 introduces a concept called modularity to DCGM where different functional areas are separated into different shared libraries. These shared libraries are lazy-loaded by DCGM when first used by a corresponding API or DCGMI call. If you never want a module to be loaded, you can blacklist that module using dcgmi, API calls, or nv-hostengine command-line arguments. Additionally, you can remove the so.1 file of the module you want to permanently blacklist, and DCGM will behave as if the library is blacklisted.

### 6.1. Module List

The following modules have been added to DCGM.

<table>
<thead>
<tr>
<th>Module ID</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DcgmModuleIdNvSwitch</td>
<td>1</td>
<td>Manages NVSwitches and is required in order for DGX-2 / HGX-2 systems to function</td>
</tr>
</tbody>
</table>
6.2. Blacklisting Modules

Users may prevent DCGM from loading modules by providing a command-line option to nv-hostengine when they run it. The argument to this command line is the # column in the table above.

For instance, to start nv-hostengine with the introspection (3) and health (4) modules blacklisted, you would change nv-hostengine’s service file to pass the following arguments: `nv-hostengine --blacklist-modules 3,4`

Note that NVSwitch must be explicitly loaded with the -l and -g options. To only load the NVSwitch module and blacklist all others, use the following command line:

`nv-hostengine -l -g --blacklist-modules 2,3,4,5,6,7`

You can query the status of all of the dcgm modules with the following command:

```
#dcgmi modules -l
```

<table>
<thead>
<tr>
<th>Module ID</th>
<th>Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Core</td>
<td>Loaded</td>
</tr>
<tr>
<td>1</td>
<td>NvSwitch</td>
<td>Not loaded</td>
</tr>
<tr>
<td>2</td>
<td>VGPU</td>
<td>Not loaded</td>
</tr>
<tr>
<td>3</td>
<td>Introspection</td>
<td>Not loaded</td>
</tr>
<tr>
<td>4</td>
<td>Health</td>
<td>Not loaded</td>
</tr>
<tr>
<td>5</td>
<td>Policy</td>
<td>Not loaded</td>
</tr>
<tr>
<td>6</td>
<td>Config</td>
<td>Not loaded</td>
</tr>
<tr>
<td>7</td>
<td>Diag</td>
<td>Not loaded</td>
</tr>
</tbody>
</table>

Only modules that are Not Loaded can be blacklisted.
To blacklist a module, take note of its module name from the table above. We'll blacklist module Policy for this example:

```
#dcgmi modules --blacklist Policy
```

```
| Blacklist Module                                                                 |
| Status: Success                                                                 |
| Successfully blacklisted module Policy                                          |
```

Once a module has been blacklisted, you can verify that by listing modules again:

```
dcgmi modules -l
```

```
| List Modules                                                                 |
| Status: Success                                                                 |
```

```
<table>
<thead>
<tr>
<th>Module ID</th>
<th>Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Core</td>
<td>Loaded</td>
</tr>
<tr>
<td>1</td>
<td>NvSwitch</td>
<td>Not loaded</td>
</tr>
<tr>
<td>2</td>
<td>VGPU</td>
<td>Not loaded</td>
</tr>
<tr>
<td>3</td>
<td>Introspection</td>
<td>Not loaded</td>
</tr>
<tr>
<td>4</td>
<td>Health</td>
<td>Not loaded</td>
</tr>
<tr>
<td>5</td>
<td>Policy</td>
<td>Blacklisted</td>
</tr>
<tr>
<td>6</td>
<td>Config</td>
<td>Not loaded</td>
</tr>
<tr>
<td>7</td>
<td>Diag</td>
<td>Not loaded</td>
</tr>
</tbody>
</table>
```
Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, 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 product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide 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 guide. 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 guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide 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.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the Unites States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2018-2021 NVIDIA Corporation. All rights reserved.