

Overview

The NVIDIA® Data Center GPU Manager (DCGM) simplifies administration of NVIDIA Datacenter (previously “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 NVIDIA GPUs.

Terminology

Terms used in this document

Term	Meaning
DCGM	NVIDIA’s Datacenter GPU Manager
NVIDIA Host Engine	Standalone executable wrapper for DCGM shared library
Host Engine daemon	Daemon mode of operation for the NVIDIA Host Engine
Fabric Manager	A module within the Host Engine daemon that supports NVSwitch fabric on DGX-2 or HGX-2.
3rd-party DCGM Agent	Any node-level process from a 3rd-party that runs DCGM in Embedded Mode
Embedded Mode	DCGM executing as a shared library within a 3rd-party DCGM agent

Standalone Mode	DCGM executing as a standalone process via the Host Engine
System Validation	Health checks encompassing the GPU, board and surrounding environment
HW diagnostic	System validation component focusing on GPU hardware correctness
RAS event	Reliability, Availability, Serviceability event. Corresponding to both fatal and non-fatal GPU issues
NVML	NVIDIA Management Library

Focus Areas

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

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 continuous GPU telemetry

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. DCGM also provides continuous GPU telemetry at very low performance overheads.

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.

Note

As of v2.x, DCGM no longer includes the Fabric Manager, which is a separate component that needs to be installed for NVSwitch based systems.

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.

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, Python and Go 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.

Getting Started 🌱

Supported Platforms

DCGM currently supports the following products and environments:

- All Kepler (K80) and newer NVIDIA datacenter (previously, 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, a minimum of DCGM 2.0 is recommended.
- All Maxwell and newer non-datacenter (e.g. GeForce or Quadro) GPUs
- CUDA 7.5+ and NVIDIA Driver R450+
- Bare metal and virtualized (full passthrough only)

Note

- 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](#).
- Starting with v1.3, limited DCGM functionality is available on non-datacenter GPUs. More details are available in the section Feature Overview.

Supported Linux Distributions

Linux Distributions and Architectures

Linux Distribution	x86 (x86_64)	Arm64 (aarch64)	POWER (ppc64le)
Debian 10	X		
Debian 11	X		
RHEL 8.y/Rocky Linux 8.y	X	X	X
RHEL/CentOS 7.y	X		
SLES/OpenSUSE 15.y	X	X	

Ubuntu 22.04 LTS	X	X	
Ubuntu 20.04 LTS	X	X	
Ubuntu 18.04 LTS	X	X	

Installation

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

- Supported **NVIDIA Datacenter Driver**
- Supported CUDA Toolkit
- DCGM Runtime and SDK
- 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/local/dcgm/bindings` directory after installation. The user must be `root` or have `sudo` privileges for installation, as for any such packaging operations.

Note

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.

Remove Older Installations

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

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

```
$ sudo nv-hostengine -t
```

Remove the previous installation:

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

Installation

Note

Since the CUDA repository GPG keys were rotated, any old GPG keys should be removed from the system prior to installing packages from the CUDA network repository:

Ubuntu LTS

RHEL/CentOS/Rocky Linux/SLES/OpenSUSE

```
$ sudo apt-key del 7fa2af80
```

Ubuntu LTS and Debian

Determine the distribution name:

```
$ distribution=$(. /etc/os-release;echo $ID$VERSION_ID | sed -e 's/\./_/g')
```

Download the meta-package to setup the CUDA network repository:

x86_64

arm64

ppc64le

```
$ wget  
https://developer.download.nvidia.com/compute/cuda/repos/$distribution/x86_64,  
keyring_1.0-1_all.deb
```

Install the repository meta-data and the CUDA GPG key:

```
$ sudo dpkg -i cuda-keyring_1.0-1_all.deb
```

Update the Apt repository cache

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

Now, install DCGM

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

RHEL / CentOS / Rocky Linux

Note substitute `<architecture>` as `x86_64`, `sbsa` or `ppc64le` as appropriate:

Determine the distribution name:

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

Install the repository meta-data and the CUDA GPG key:

x86_64

arm64

ppc64le

```
$ sudo dnf config-manager \
  --add-repo
http://developer.download.nvidia.com/compute/cuda/repos/$distribution/x86_64/rhel8.repo
```

Update the repository metadata

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

Now, install DCGM

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


SUSE SLES / OpenSUSE

Note substitute `<architecture>` as `x86_64`, `sbsa` or `ppc64le` as appropriate:

Determine the distribution name:

```
$ distribution=$(cat /etc/os-release;echo $ID$VERSION_ID | sed -e 's/\.[0-9]//')
```

Install the repository meta-data and the CUDA GPG key:

x86_64

arm64

ppc64le

```
$ sudo zypper ar \
```

```
http://developer.download.nvidia.com/compute/cuda/repos/$distribution/x86_64/
$distribution.repo
```

Update the repository metadata

```
$ sudo zypper refresh
```

Now, install DCGM

```
$ sudo zypper install datacenter-gpu-manager
```

Post-Install

Note

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.

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

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

```
● dcgm.service - DCGM service
   Loaded: loaded (/usr/lib/systemd/system/dcgm.service; disabled; vendor preset:
   enabled)
   Active: active (running) since Mon 2020-10-12 12:18:57 PDT; 14s ago
 Main PID: 32847 (nv-hostengine)
    Tasks: 7 (limit: 39321)
   CGroup: /system.slice/dcgm.service
           └─32847 /usr/bin/nv-hostengine -n

Oct 12 12:18:57 ubuntu1804 systemd[1]: Started DCGM service.
Oct 12 12:18:58 ubuntu1804 nv-hostengine[32847]: DCGM initialized
Oct 12 12:18:58 ubuntu1804 nv-hostengine[32847]: Host Engine Listener Started
```

To verify installation, use `dcgmi` to query the system. You should see a listing of all supported GPUs (and any NVSwitches) found in the system:

```
$ dcgmi discovery -l
```

```
8 GPUs found.
```

```
+-----+-----+
+
| GPU ID | Device Information
|
+-----+-----+
+
| 0      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:07:00.0
|
|        | Device UUID: GPU-1d82f4df-3cf9-150d-088b-52f18f8654e1
|
+-----+-----+
+
| 1      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:0F:00.0
|
|        | Device UUID: GPU-94168100-c5d5-1c05-9005-26953dd598e7
|
+-----+-----+
+
| 2      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:47:00.0
|
|        | Device UUID: GPU-9387e4b3-3640-0064-6b80-5ace1ee535f6
```

```

|
+-----+
+
| 3      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:4E:00.0
|
|        | Device UUID: GPU-cefd0e59-c486-c12f-418c-84ccd7a12bb2
|
+-----+
+
| 4      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:87:00.0
|
|        | Device UUID: GPU-1501b26d-f3e4-8501-421d-5a444b17eda8
|
+-----+
+
| 5      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:90:00.0
|
|        | Device UUID: GPU-f4180a63-1978-6c56-9903-ca5aac8af020
|
+-----+
+
| 6      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:B7:00.0
|
|        | Device UUID: GPU-8b354e3e-0145-6cfc-aec6-db2c28dae134
|
+-----+
+
| 7      | Name: A100-SXM4-40GB
|
|        | PCI Bus ID: 00000000:BD:00.0
|
|        | Device UUID: GPU-a16e3b98-8be2-6a0c-7fac-9cb024dbc2df
|
+-----+
+
6 NvSwitches found.
+-----+
| Switch ID |
+-----+
| 11        |
| 10        |
| 13        |
| 9         |
| 12        |
| 8         |
+-----+

```

Basic Components

The DCGM SDK contains these 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.

Note

- 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/local/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/local/dcgm/sdk_samples`.

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.

Note

In both modes the DCGM library should be run as root. Many features will not work without privileged access to the GPU, including various configuration settings and diagnostics.

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.

Warning

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

```
dcgmShutdown()
```

Note

For a more complete example see the Embedded Mode example in the DCGM SDK

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.

Note

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.

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
```

Feature Overview

The following sections review key DCGM features, along with examples of input and output using the `dcgmcli` CLI. Common usage scenarios and suggested best practices are included as well. Starting with v1.3, DCGM is supported on non-Tesla branded GPUs. The following table lists the features available on different GPU products.

Features Supported on Different GPU Products

Feature Group	Tesla	Titan	Quadro	GeForce
Field Value Watches (GPU metrics)	X	X	X	X
Configuration Management	X	X	X	X
Active Health Checks (GPU subsystems)	X	X	X	X
Job Statistics	X	X	X	X
Topology	X	X	X	X
Introspection	X	X	X	X
Policy Notification	X			
GPU Diagnostics (Diagnostic Levels - 1, 2, 3)	All Levels	Level 1	Level 1	Level 1

 **Note**

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

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.

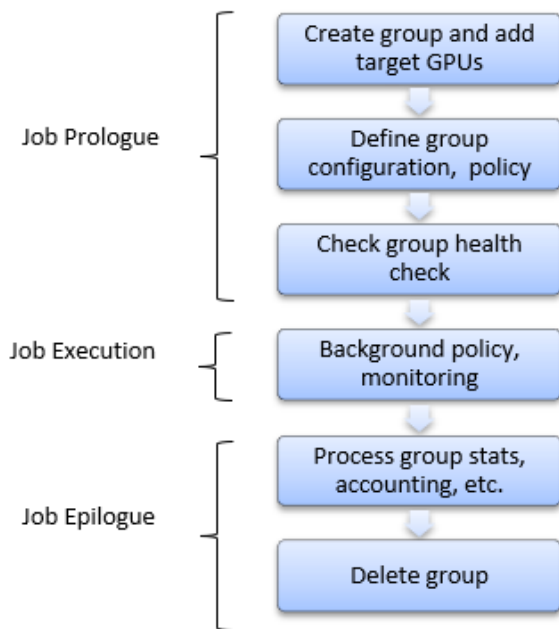
In machines with only one GPU the group concept can be ignored altogether, as all DCGM operations that require a group can use one containing that sole GPU. For convenience, at init, DCGM creates a default group representing all supported GPUs in the system. Groups in DCGM need not be disjoint. In many 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.



```
$ dcgmi group -c GPU_Group
```

Successfully created group "GPU_Group" with a group ID of 1

```
$ dcgmi group -l
```

1 group found.

GROUPS	
Group ID	1
Group Name	GPU_Group
GPU ID(s)	None

```
$ dcgmi group -d 1
```

Successfully removed group 1

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.

```
$ dcgmi discovery -l
```

```
2 GPUs found.
```

GPU ID		Device Information
0		Name: Tesla K80 PCI Bus ID: 0000:07:00.0 Device UUID: GPU-00000000000000000000000000000000
1		Name: Tesla K80 PCI Bus ID: 0000:08:00.0 Device UUID: GPU-11111111111111111111111111111111

```
$ dcgmi group -g 1 -a 0,1
```

```
Add to group operation successful.
```

```
$ dcgmi group -g 1 -i
```

GROUPS	
Group ID	1
Group Name	GPU_Group
GPU ID(s)	0, 1

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:

DCGM Configuration Settings

Setting	Description	Defaults
Sync Boost	Coordinate Auto Boost across GPUs in the group	None
Target Clocks	Attempt to maintain fixed clocks at the target values	None
ECC Mode	Enable ECC protection throughout the GPU's memory	Usually On
Power Limit	Set the maximum allowed power consumption	Varies
Compute Mode	Limit concurrent process access to the GPU	No restrictions

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.

```
$ dcgmi config -g 1 --set -c 2
```

```
Configuration successfully set.
```

```
$ dcgmi config -g 1 --get
```

GPU_Group	TARGET CONFIGURATION	CURRENT CONFIGURATION
Group of 2 GPUs		
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.

```
$ dcgmi config -g 1 --get --verbose
```

GPU ID: 0	TARGET CONFIGURATION	CURRENT CONFIGURATION
Tesla K20c		
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	TARGET CONFIGURATION	CURRENT CONFIGURATION
GeForce GT 430		
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.

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:

Error Conditions

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

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.

Note

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
```

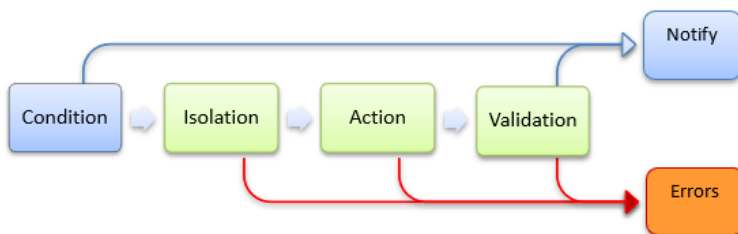
```
Listening for violations
...
A PCIe error has violated policy manager values.
...
```

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.

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.

Job Statistics

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.

```
$ 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 `dcmgi stats`:

```
$ 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                           |
| - Reliability (%)             | 0                           |
| - Board Limit (%)            | 0                           |
| - Low Utilization (%)         | 0                           |
| - Sync Boost (%)              | Not Supported               |
+-----+-----+

```

(*) Represents a process statistic. Otherwise device statistic during process lifetime listed.

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>

```
+-----+
| 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 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 |
+-----+
```

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 active investigated for hardware problems or other serious issues.

⚠ Warning

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.

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:

- **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.

Note

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.

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:

Test Suite Name	Run Level	Test Duration	Test Classes			
			Software	Hardware	Integration	Stress
Quick	-r 1	~ seconds	Deployment	-	-	
Medium	-r 2	~ 2 minutes	Deployment	Memory Test	PCIe/NVLink	-
Long	-r 3	~ 15 minutes	Deployment	<ul style="list-style-type: none"> • Memory Test • Memory Bandwidth • HW Diagnostic Tests 	PCIe/NVLink	<ul style="list-style-type: none"> • SM Stress • Targeted Stress • Targeted Power

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):

```
$ 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
----- Hardware -----	
GPU Memory	Pass - All
Diagnostic	Pass - All
----- Integration -----	
PCIe	Pass - All
----- Performance -----	
SM Stress	Pass - All
Targeted Stress	Pass - All
Targeted Power	Pass - All
Memory Bandwidth	Pass - All

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.

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:

```
$ dcgmi topo --gpuid 0
```

GPU ID: 0	Topology Information
CPU Core Affinity	0 - 11
To GPU 1	Connected via an on-board PCIe switch
To GPU 2	Connected via a PCIe host bridge

And for the group-level view:

```
$ dcgmi topo -g 1
```

MyGroup	Topology Information
CPU Core Affinity	0 - 11
NUMA Optimal	True
Worst Path	Connected via a PCIe host bridge
.....	

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
```


+-----+-----+-----+			
GPU ID: 0 NVLINK Error Counts			
+-----+-----+-----+			
Link 0	CRC FLIT Error	0	
Link 0	CRC Data Error	0	
Link 0	Replay Error	0	
Link 0	Recovery Error	0	
Link 1	CRC FLIT Error	0	
Link 1	CRC Data Error	0	
Link 1	Replay Error	0	
Link 1	Recovery Error	0	
Link 2	CRC FLIT Error	0	
Link 2	CRC Data Error	0	
Link 2	Replay Error	0	
Link 2	Recovery Error	0	
Link 3	CRC FLIT Error	0	
Link 3	CRC Data Error	0	
Link 3	Replay Error	0	
Link 3	Recovery Error	0	
+-----+-----+-----+			

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.

FIELD GROUPS	
ID	1
Name	DCGM_INTERNAL_1SEC
Field IDs	38, 73, 86, 112, 113, 119, 73, 51, 47, 46, 66, 72, 61, 118,...
ID	2
Name	DCGM_INTERNAL_30SEC
Field IDs	124, 125, 126, 130, 131, 132, 133, 134, 135, 136, 137, 138,...
ID	3
Name	DCGM_INTERNAL_HOURLY
Field IDs	117, 55, 56, 64, 62, 63, 6, 5, 26, 8, 17, 107, 22, 108, 30, 31
ID	4
Name	DCGM_INTERNAL_JOB
Field IDs	111, 65, 36, 37, 38, 101, 102, 77, 78, 40, 41, 121, 115, 11...

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
```

FIELD GROUPS	
ID	5
Name	mygroupname
Field IDs	50, 51, 52

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
```

Link Status

Starting with 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
```

```
+-----+  
|  NvLink Link Status  |  
+-----+
```

GPUs:

```
gpuId 0:
  U U U U U U
gpuId 1:
  U U U U U U
gpuId 2:
  U U U U U U
gpuId 3:
  U U U U U U
gpuId 4:
  U U U U U U
gpuId 5:
  U U U U U U
gpuId 6:
  U U U U U U
gpuId 7:
  U U U U U U
gpuId 8:
  U U U U U U
gpuId 9:
  U U U U U U
gpuId 10:
  U U U U U U
gpuId 11:
  U U U U U U
gpuId 12:
  U U U U U U
gpuId 13:
  U U U U U U
gpuId 14:
  U U U U U U
gpuId 15:
  U U U U U U
```

NvSwitches:

```
physicalId 8:
  U U U U U U X X U U U U U U U U
physicalId 9:
  U U U U U U U U U U U U U X X U U
physicalId 10:
  U U U U U U U U U U U X U U U X U
physicalId 11:
  U U U U U U X X U U U U U U U U U
physicalId 12:
  U U U U X U U U U U U U U U X U
physicalId 13:
  U U U U X U U U U U U U U U X U
physicalId 24:
  U U U U U U X X U U U U U U U U
physicalId 25:
  U U U U U U U U U U U U U X X U U
physicalId 26:
  U U U U U U U U U U U X U U U X U
physicalId 27:
  U U U U U U X X U U U U U U U U U
physicalId 28:
  U U U U X U U U U U U U U U X U
physicalId 29:
  U U U U X U U U U U U U U U X U
```

Key: Up=U, Down=D, Disabled=X, Not Supported=_

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 supported in production 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.

Metrics

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`).

Device Level GPU Metrics

Metric	Definition	DCGM Field Name (DCGM_FI_*) and ID
Graphics Engine Activity	The fraction of time any portion of the graphics or compute engines were active. The graphics engine is active if a graphics/compute context is bound and the graphics/compute pipe is busy. The value represents an average over a time interval and is not an instantaneous value.	PROF_GR_ENGINE_ACTIVE (ID: 1001)
SM Activity	The fraction of time at least one warp was active on a multiprocessor, averaged over all multiprocessors. Note that "active" does not necessarily mean a warp is actively computing. For instance, warps waiting on memory requests are considered active. The value represents an average over a time interval and is not an instantaneous value. A value of 0.8 or greater is necessary, but not sufficient, for effective use of the GPU. A value less than 0.5 likely indicates ineffective GPU usage. Given a simplified GPU architectural view, if a GPU has N SMs then a kernel using N blocks that runs over the entire time interval will correspond to an activity of 1 (100%). A kernel using N/5 blocks that runs over the entire time interval will correspond to an activity of 0.2 (20%). A kernel using N blocks that runs over one fifth of the time interval, with the SMs otherwise idle, will also have an activity of 0.2 (20%). The value is insensitive to the number of threads per block (see <code>DCGM_FI_PROF_SM_OCCUPANCY</code>).	PROF_SM_ACTIVE (ID: 1002)

SM Occupancy	<p>The fraction of resident warps on a multiprocessor, relative to the maximum number of concurrent warps supported on a multiprocessor. The value represents an average over a time interval and is not an instantaneous value. Higher occupancy does not necessarily indicate better GPU usage. For GPU memory bandwidth limited workloads (see DCGM_FI_PROF_DRAM_ACTIVE), higher occupancy is indicative of more effective GPU usage. However if the workload is compute limited (i.e. not GPU memory bandwidth or latency limited), then higher occupancy does not necessarily correlate with more effective GPU usage.</p> <p>Calculating occupancy is not simple and depends on factors such as the GPU properties, the number of threads per block, registers per thread, and shared memory per block. Use the CUDA Occupancy Calculator to explore various occupancy scenarios.</p>	PROF_SM_OCCUPANCY (ID: 1003)
Tensor Activity	<p>The fraction of cycles the tensor (HMMA / IMMA) pipe was active. The value represents an average over a time interval and is not an instantaneous value. Higher values indicate higher utilization of the Tensor Cores. An activity of 1 (100%) is equivalent to issuing a tensor instruction every other cycle for the entire time interval. An activity of 0.2 (20%) could indicate 20% of the SMs are at 100% utilization over the entire time period, 100% of the SMs are at 20% utilization over the entire time period, 100% of the SMs are at 100% utilization for 20% of the time period, or any combination in between (see DCGM_FI_PROF_SM_ACTIVE to help disambiguate these possibilities).</p>	PROF_PIPE_TENSOR_ACTIVE (ID: 1004)
FP64 Engine Activity	<p>The fraction of cycles the FP64 (double precision) pipe was active. The value represents an average over a time interval and is not an instantaneous value. Higher values indicate higher utilization of the FP64 cores. An activity of 1 (100%) is equivalent to a FP64 instruction on every SM every fourth cycle on Volta over the entire time interval. An activity of 0.2 (20%) could indicate 20% of the SMs are at 100% utilization over the entire time period, 100% of the SMs are at 20% utilization over the entire time period, 100% of the SMs are at 100% utilization for 20% of the time period, or any combination in between (see DCGM_FI_PROF_SM_ACTIVE to help disambiguate these possibilities).</p>	PROF_PIPE_FP64_ACTIVE (ID: 1006)
FP32 Engine Activity	<p>The fraction of cycles the FMA (FP32 (single precision), and integer) pipe was active. The value represents an average over a time interval and is not an instantaneous value. Higher values indicate higher utilization of the FP32 cores. An activity of 1 (100%) is equivalent to a FP32 instruction every other cycle over the entire time interval. An activity of 0.2 (20%) could indicate 20% of the SMs are at 100% utilization over the entire time period,</p>	PROF_PIPE_FP32_ACTIVE (ID: 1007)

	100% of the SMs are at 20% utilization over the entire time period, 100% of the SMs are at 100% utilization for 20% of the time period, or any combination in between (see <code>DCGM_FI_PROF_SM_ACTIVE</code> to help disambiguate these possibilities).	
FP16 Engine Activity	The fraction of cycles the FP16 (half precision) pipe was active. The value represents an average over a time interval and is not an instantaneous value. Higher values indicate higher utilization of the FP16 cores. An activity of 1 (100%) is equivalent to a FP16 instruction every other cycle over the entire time interval. An activity of 0.2 (20%) could indicate 20% of the SMs are at 100% utilization over the entire time period, 100% of the SMs are at 20% utilization over the entire time period, 100% of the SMs are at 100% utilization for 20% of the time period, or any combination in between (see <code>DCGM_FI_PROF_SM_ACTIVE</code> to help disambiguate these possibilities).	PROF_PIPE_FP16_ACTIVE (ID: 1008)
Memory BW Utilization	The fraction of cycles where data was sent to or received from device memory. The value represents an average over a time interval and is not an instantaneous value. Higher values indicate higher utilization of device memory. An activity of 1 (100%) is equivalent to a DRAM instruction every cycle over the entire time interval (in practice a peak of ~0.8 (80%) is the maximum achievable). An activity of 0.2 (20%) indicates that 20% of the cycles are reading from or writing to device memory over the time interval.	PROF_DRAM_ACTIVE (ID: 1005)
NVLink Bandwidth	The rate of data transmitted / received over NVLink, not including protocol headers, in bytes per second. The value represents an average over a time interval and is not an instantaneous value. The rate is averaged over the time interval. For example, if 1 GB of data is transferred over 1 second, the rate is 1 GB/s regardless of the data transferred at a constant rate or in bursts. The theoretical maximum NVLink Gen2 bandwidth is 25 GB/s per link per direction.	PROF_NVLINK_TX_BYTES (1011) and PROF_NVLINK_RX_BYTES (1012)
PCIe Bandwidth	The rate of data transmitted / received over the PCIe bus, including both protocol headers and data payloads, in bytes per second. The value represents an average over a time interval and is not an instantaneous value. The rate is averaged over the time interval. For example, if 1 GB of data is transferred over 1 second, the rate is 1 GB/s regardless of the data transferred at a constant rate or in bursts. The theoretical maximum PCIe Gen3 bandwidth is 985 MB/s per lane.	PROF_PCIE_[T R]X_BYTES (ID: 1009 (TX); 1010 (RX))

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.

Multiplexing of Profiling Counters

Some metrics require multiple passes to be collected and therefore all metrics cannot be collected together. Due to hardware limitations on the GPUs, only certain groups of metrics can be read together. For example, SM Activity | SM Occupancy cannot be collected together with Tensor Utilization on V100 but can be done on T4. To overcome these hardware 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.

A side-effect of multiplexing is that collection at higher frequencies will result in zeroes returned as DCGM attempts to group metrics together for collection.

The metrics that can be collected together for a specific GPU can be determined by running the following command:

```
$ dcgmi profile -l -i 0
```

Group.Subgroup	Field ID	Field Tag
A.1	1002	sm_active
A.1	1003	sm_occupancy
A.1	1004	tensor_active
A.1	1007	fp32_active
A.2	1006	fp64_active
A.3	1008	fp16_active
B.0	1005	dram_active
C.0	1009	pcie_tx_bytes
C.0	1010	pcie_rx_bytes
D.0	1001	gr_engine_active
E.0	1011	nvlink_tx_bytes
E.0	1012	nvlink_rx_bytes

From the output above, we can determine that for this GPU (in this example, an NVIDIA T4), a metric from each letter group can be collected without multiplexing. From this example, a metric from [A.1](#) can be collected with another metric from [A.1](#) without multiplexing. A metric from [A.1](#) will be multiplexed with another metric from [A.2](#) or [A.3](#). Metrics from different letter groups can be combined for concurrent collection (without requiring multiplexing by DCGM).

Building on this example further, on T4 these metrics can be collected together without multiplexing:

```
sm_active + sm_occupancy + tensor_active + fp32_active
```

The above DCGM command will show what groupings are supported by the hardware for concurrent collection.

Profiling Sampling Rate

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. In general, DCGM is not designed for very high frequency sampling due to the total time involved in the collection/processing of metrics from the hardware counters and the driver.

The update frequency can be modified by setting the value in the `dcgmProfUnwatchFields_v1` structure that is passed to `dcgmProfWatchFields` when watching the `profiling metrics`.

Concurrent Usage of NVIDIA Profiling Tools Due to current hardware limitations, collection of profiling metrics with DCGM will conflict with usage of other developer tools from NVIDIA such as Nsight Systems or Nsight Compute. Users may encounter an error from DCGM (either with `dcgmi` or when using the APIs) such as:

```
Error setting watches. Result: The requested operation could not be completed because the affected resource is in use.
```

To allow DCGM to co-exist with the usage of other profiling tools, it is recommended to pause metrics collection with DCGM when the tools are in use and then resume after the usage of the tools is complete.

With `dcgmi profile`, the `--pause` and `--resume` options can be used:

```
$ dcgmi profile --pause
$ dcgmi profile --resume
```

When using DCGM APIs, the following APIs can be called from the monitoring process:

`dcgmProfPause()` and `dcgmProfResume()`

When paused, DCGM will publish `BLANK` values for profiling metrics. These `BLANK` values can be tested with `DCGM_FP64_IS_BLANK(value)` in the C or Python bindings.

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.

For example in a console, generate a load for the TensorCores on A100 for 30seconds. As can be seen, the A100 is able to achieve close to 253TFlops of FP16 performance using the TensorCores.

```
$ /usr/bin/dcgmproftester11 --no-dcgm-validation -t 1004 -d 10
```

```
Skipping CreateDcgmGroups() since DCGM validation is disabled
Skipping CreateDcgmGroups() since DCGM validation is disabled
Skipping WatchFields() since DCGM validation is disabled
Skipping CreateDcgmGroups() since DCGM validation is disabled
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (250362.2 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (252917.0 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (253971.7 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (253700.2 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (252599.0 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (253134.6 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (252676.7 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (252861.4 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (252764.1 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm 0.000 (253109.4 gflops)
Worker 0:0[1004]: Message: Bus ID 00000000:00:04.0 mapped to cuda device ID 0
DCGM CudaContext Init completed successfully.
```

```
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR: 2048
CUDA_VISIBLE_DEVICES:
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT: 108
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR: 167936
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR: 8
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR: 0
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH: 5120
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE: 1215
Max Memory bandwidth: 1555200000000 bytes (1555.2 GiB)
CU_DEVICE_ATTRIBUTE_ECC_SUPPORT: true
```

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 ~93%. 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
```

#	Entity Id	GRACT	TENSO	DRAMA
	GPU 0	0.000	0.000	0.000
	GPU 0	0.000	0.000	0.000
	GPU 0	0.000	0.000	0.000
	GPU 0	0.552	0.527	0.000
	GPU 0	0.969	0.928	0.000
	GPU 0	0.973	0.931	0.000
	GPU 0	0.971	0.929	0.000
	GPU 0	0.969	0.927	0.000
	GPU 0	0.971	0.929	0.000
	GPU 0	0.971	0.930	0.000
	GPU 0	0.973	0.931	0.000
	GPU 0	0.974	0.931	0.000
	GPU 0	0.971	0.930	0.000
	GPU 0	0.974	0.932	0.000
	GPU 0	0.972	0.930	0.000

Metrics on Multi-Instance GPU

The Multi-Instance GPU (MIG) feature allows supported NVIDIA GPUs to be securely partitioned into up to seven separate GPU Instances for CUDA applications, providing multiple users with separate GPU resources for optimal GPU utilization. This feature is particularly beneficial for workloads that do not fully saturate the GPU's compute capacity and therefore users may want to run different workloads in parallel to maximize utilization. For more information on MIG, refer to the [MIG User Guide](#).

DCGM can provide metrics for workloads that are running on MIG devices. DCGM offers two views of the metrics for MIG:

- GPU device-level metrics
- MIG device (either GPU Instance or Compute Instance) granularity of the metrics

Example 1

In this example, let's generate a CUDA workload using `dcgmproftester` and observe metrics using `dcgmi dmon`.

In this example, we follow these steps to demonstrate the collection of metrics for MIG devices:

- Create MIG devices (assumes that the GPU has MIG mode enabled)
- Verify DCGM can list the devices
- Create a group of devices for DCGM to monitor
- Run CUDA workloads on the desired MIG device(s)
- Use `dcgmi dmon` to stream metrics

Step 1: Create MIG Devices Notice that we have partitioned the GPU into two MIG devices with the 3g.20gb profiles.

```
$ sudo nvidia-smi mig -cgi 9,9 -C
```

```
...
+-----+
| MIG devices: |
+-----+
| GPU  GI  CI  MIG |      Memory-Usage | SM  Vol |      Shared | | | | | | |
|      ID ID  Dev |      BAR1-Usage   |     Unc| CE  ENC  DEC  OFA  JPG |
|      |      |      |      |      |      |      |      |      |      |
+-----+-----+-----+-----+-----+-----+-----+
|  0    1    0    0 | 11MiB / 20096MiB | 42    0 | 3    0    2    0    0 |
|      |      |      |      |      |      |      |      |      |      |
+-----+-----+-----+-----+-----+-----+
|  0    2    0    1 | 11MiB / 20096MiB | 42    0 | 3    0    2    0    0 |
|      |      |      |      |      |      |      |      |      |      |
+-----+-----+-----+-----+-----+-----+

```

Step 2: Verify enumeration by DCGM We can also observe the devices enumerated by DCGM:

```
$ dcgmi discovery -c
```

```
+-----+
-+
| Instance Hierarchy |
+-----+
| GPU 0              | GPU GPU-5fd15f35-e148-2992-4ecb-9825e534f253 (EntityID: 0) |
| -> I 0/1           | GPU Instance (EntityID: 0) |
|   -> CI 0/1/0       | Compute Instance (EntityID: 0) |
| -> I 0/2           | GPU Instance (EntityID: 1) |
|   -> CI 0/2/0       | Compute Instance (EntityID: 1) |
+-----+
-+

```

Step 3: Creation of MIG device groups Create groups of devices for DCGM to monitor. In this example, we will add the GPU and the two GPU Instances (using entity IDs 0 and 1) to the group:

```
$ dcgmi group -c mig-ex1 -a 0,i:0,i:1
```

```
Successfully created group "mig-ex1" with a group ID of 8
Specified entity ID is valid but unknown: i:0. ParsedResult: ParsedGpu(i:0)
Specified entity ID is valid but unknown: i:1. ParsedResult: ParsedGpu(i:1)
Add to group operation successful.
```

Now, we can list the devices added to the group and see that the group contains the GPU (GPU:0), GPU Instances (0 and 1):

```
$ dcgmi group -l
```

```
+-----+
| GROUPS                                     |
| 1 group found.                           |
+=====+
| Groups                                    |
| -> 8                                     |
|   -> Group ID      | 8                  |
|   -> Group Name    | mig-ex1             |
|   -> Entities      | GPU 0, GPU_I 0, GPU_I 1 |
+-----+
```

Step 4: Run CUDA workloads Launch `dcgmproftester`. Note that `dcgmproftester` will run on all GPU instances available on the GPU. There is currently no way to limit the GPU instances from `dcgmproftester` (though one could do this by running a `dcgmproftester` container, which we will see in the next example).

```
$ sudo dcgmproftester11 --no-dcgm-validation -t 1004 -d 120
```

```
Skipping CreateDcgmGroups() since DCGM validation is disabled
Skipping CreateDcgmGroups() since DCGM validation is disabled
Skipping CreateDcgmGroups() since DCGM validation is disabled
Skipping WatchFields() since DCGM validation is disabled
Skipping CreateDcgmGroups() since DCGM validation is disabled
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm { GPU: 0.000, GI: 0.000, CI:
0.000 } (102659.5 gflops)
Worker 0:1[1004]: TensorEngineActive: generated ???, dcgm { GPU: 0.000, GI: 0.000, CI:
0.000 } (102659.8 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm { GPU: 0.000, GI: 0.000, CI:
0.000 } (107747.3 gflops)
Worker 0:1[1004]: TensorEngineActive: generated ???, dcgm { GPU: 0.000, GI: 0.000, CI:
0.000 } (107787.3 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm { GPU: 0.000, GI: 0.000, CI:
0.000 } (108107.6 gflops)
Worker 0:1[1004]: TensorEngineActive: generated ???, dcgm { GPU: 0.000, GI: 0.000, CI:
0.000 } (108102.3 gflops)
Worker 0:0[1004]: TensorEngineActive: generated ???, dcgm { GPU: 0.000, GI: 0.000, CI:
0.000 } (108001.2 gflops)
```

snip...snip

Worker 0:0[1004]: Message: DCGM CudaContext Init completed successfully.

```
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR: 2048
CUDA_VISIBLE_DEVICES: MIG-GPU-5fd15f35-e148-2992-4ecb-9825e534f253/1/0
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT: 42
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR: 167936
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR: 8
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR: 0
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH: 5120
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE: 1215
Max Memory bandwidth: 1555200000000 bytes (1555.2 GiB)
CU_DEVICE_ATTRIBUTE_ECC_SUPPORT: true
```

Worker 0:1[1004]: Message: DCGM CudaContext Init completed successfully.

```
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR: 2048
CUDA_VISIBLE_DEVICES: MIG-GPU-5fd15f35-e148-2992-4ecb-9825e534f253/2/0
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT: 42
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR: 167936
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR: 8
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR: 0
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH: 5120
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE: 1215
Max Memory bandwidth: 1555200000000 bytes (1555.2 GiB)
CU_DEVICE_ATTRIBUTE_ECC_SUPPORT: true
```

Skipping UnwatchFields() since DCGM validation is disabled

Step 5: Stream metrics via `dcgmi dmon` Now in another window, we can observe the metrics being attributed to each MIG device using `dcgmi dmon`. Notice that in this example, we show the GPU Activity (1001) and the Tensor Core Utilization (1004) for the group 8 that we created in Step 3.

```
$ dcgmi dmon -e 1001,1004 -g 8
```

```
# Entity   GRACT  TENS0
   Id
GPU 0      0.000  0.000
GPU-I 0     0.000  0.000
GPU-I 1     0.000  0.000
GPU 0      0.000  0.000
GPU-I 0     0.000  0.000
GPU-I 1     0.000  0.000
GPU 0      0.457  0.442
GPU-I 0     0.534  0.516
GPU-I 1     0.533  0.515
GPU 0      0.845  0.816
GPU-I 0     0.986  0.953
GPU-I 1     0.985  0.952
GPU 0      0.846  0.817
GPU-I 0     0.987  0.953
GPU-I 1     0.986  0.953
GPU 0      0.846  0.817
GPU-I 0     0.987  0.954
GPU-I 1     0.986  0.953
GPU 0      0.843  0.815
GPU-I 0     0.985  0.951
GPU-I 1     0.983  0.950
GPU 0      0.845  0.817
GPU-I 0     0.987  0.953
GPU-I 1     0.985  0.952
GPU 0      0.844  0.816
GPU-I 0     0.985  0.952
GPU-I 1     0.984  0.951
GPU 0      0.845  0.816
GPU-I 0     0.986  0.952
GPU-I 1     0.985  0.952
```

Understanding Metrics

To understand the metrics attribution, GRACT is calculated based on the number of allocated compute resources to the total number of compute resources available on the GPU. We can see the total number of compute resources (SMs) using the `nvidia-smi` command:

```
$ sudo nvidia-smi mig -lgip
```

GPU instance profiles:									
GPU	Name	ID	Instances Free/Total	Memory GiB	P2P	SM CE	DEC JPEG	ENC OFA	
0	MIG 1g.5gb	19	0/7	4.75	No	14 1	0 0	0 0	
0	MIG 2g.10gb	14	0/3	9.75	No	28 2	1 0	0 0	
0	MIG 3g.20gb	9	0/2	19.62	No	42 3	2 0	0 0	
0	MIG 4g.20gb	5	0/1	19.62	No	56 4	2 0	0 0	
0	MIG 7g.40gb	0	0/1	39.50	No	98 7	5 1	0 1	

Because the MIG geometry in this example is 3g.20gb, the compute resources are = $2 \cdot 42 / 98$ or 85.71%. We can now interpret GRACT from the `dcgmi dmon` output:

```
GPU 0  0.845  0.816
GPU-I 0  0.986  0.952
GPU-I 1  0.985  0.952
```

Since each GPU instance is 98.6% active, the entire GPU = $0.8571 \cdot 0.986$ or 84.5% utilized. The same interpretation can be extended to Tensor Core utilization.

Platform Support

Profiling metrics are currently supported on datacenter products starting with the Volta architecture on Linux x86_64, Arm64 (aarch64) and POWER (ppc64le) platforms.

DCGM Diagnostics

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.

DCGM Diagnostic 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
 - Hardware Issues and Diagnostics

- Pending Page Retirements
- PCIe interface checks
- NVLink interface checks
- Framebuffer and memory checks
- Compute engine checks
- Integration Issues
 - PCIe replay counter checks
 - Topological limitations
 - Permissions, driver, and cgroups checks
 - Basic power and thermal constraint checks
- Stress Checks
 - Power and thermal stress
 - Throughput stress
 - Constant relative system performance
 - Maximum relative system performance
 - Memory Bandwidth

5. Provide troubleshooting help
6. Easily integrate into Cluster Scheduler and Cluster Management applications
7. Reduce downtime and failed GPU jobs

Beyond the Scope of the DCGM Diagnostics

DCGM Diagnostics are not designed to:

1. Provide comprehensive hardware diagnostics
2. Actively fix problems
3. Replace the field diagnosis tools. Please refer to <http://docs.nvidia.com/deploy/hw-field-diag/index.html> for that process.
4. Facilitate any RMA process. Please refer to <http://docs.nvidia.com/deploy/rma-process/index.html> for those procedures.

Overview of Plugins

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

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 pending row remaps or failed row remappings on the target GPU(s).
- Any other graphics processes are running on the target GPU(s) while the plugin runs

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.

Sub Test Tag	Pinned/Unpinned P2P Enabled/P2P Disabled	Description
h2d_d2h_single_pinned	Pinned	Device <-> Host Bandwidth, one GPU at a time
h2d_d2h_single_unpinned	Unpinned	Device <-> Host Bandwidth, one GPU at a time
h2d_d2h_latency_pinned	Pinned	Device <-> Host Latency, one GPU at a time
h2d_d2h_latency_unpinned	Unpinned	Device <-> Host Latency, one GPU at a time
p2p_bw_p2p_enabled	P2P Enabled	Device <-> Device bandwidth one GPU pair at a time
p2p_bw_p2p_disabled	P2P Disabled	Device <-> Device bandwidth one GPU pair at a time
p2p_bw_concurrent_p2p_enabled	P2P Enabled	Device <-> Device bandwidth, concurrently, focusing on bandwidth between GPUs between GPUs likely to be directly connected to each other -> for each (index / 2) and (index / 2)+1
p2p_bw_concurrent_p2p_disabled	P2P Disabled	Device <-> Device bandwidth, concurrently, focusing on bandwidth between GPUs between

		GPUs likely to be directly connected to each other -> for each (index / 2) and (index / 2)+1
1d_exch_bw_p2p_enabled	P2P Enabled	Device <-> 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)
1d_exch_bw_p2p_disabled	P2P Disabled	Device <-> 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)
p2p_latency_p2p_enabled	P2P Enabled	Device <-> Device Latency, one GPU pair at a time
p2p_latency_p2p_disabled	P2P Disabled	Device <-> Device Latency, one GPU pair at a time

Pulse Test Diagnostic

Overview

The Pulse Test is part of the new level 4 tests. The pulse test is meant to fluctuate the power usage to create spikes in current flow on the board to ensure that the power supply is fully functional and can handle wide fluctuations in current.

Test Description

By default, the test runs kernels with high transiency in order to create spikes in the current running to the GPU. Default parameters have been verified to create worst-case scenario failures by measuring with oscilloscopes.

The test iteratively runs different kernels while tweaking internal parameters to ensure that spikes are produced; work across GPU is synchronized to create extra stress on the power supply.

Supported Parameters

--	--	--

Parameter	Description	Default
test_duration	seconds for an internal step, not full time	500
kernel	kernel to execute	sgemm
exit_on_error	Exit on error	0
internal_loops	kernel calls between checks	1024
alpha	Alpha	2.0
beta	Beta	-1.0
waves	Minimum saturation factor	1
k_size	K value	4096
min_k_size	Minimum k schmoo value	32
freq0	Frequency in Hz	22000
duty0	Duty as a fraction	0.5
freq1	Frequency in Hz	22000
duty1	Duty as a fraction	0.5
sync_timeout	Wait time when syncing GPUs	10000
random_seed	Random seed	0xDEADCAFE
matrix_size_mode	standard, max_alloc, square, forced	standard
force_m	Forced M value	64
force_n	Forced N value	64
inject_errors	Number of errors to inject in test results	0
debug0	Debug flag	0
check_mode	crc or diff	diff
use_curand	Use curand for random numbers	1

Sample Commands

Run the entire diagnostic suite, including the pulse test:

```
$ dcgmi diag -r 4
```

Run just the pulse test:

```
$ dcgmi diag -r pulse_test
```

Run just the pulse test, but at a lower frequency:

```
$ dcgmi diag -r pulse_test -p pulse_test.freq0=3000
```

Run just the pulse test at a lower frequency and for a shorter time:

```
$ dcgmi diag -r pulse_test -p  
"pulse_test.freq0=5000;pulse_test.test_duration=180"
```

Failure Conditions

- The pulse test will fail if the power supply unit cannot handle the spikes in the current.
- It will also fail if unrecoverable memory errors, temperature violations, or XIDs occur during the test.

Memtest Diagnostic

Overview

Beginning with 2.4.0 DCGM diagnostics support an additional level 4 diagnostics (`-r4`). The first of these additional diagnostics is memtest. Similar to `memtest86`, the DCGM memtest will exercise GPU memory with various test patterns. These patterns each given a separate test and can be enabled and disabled by administrators.

Test Descriptions

Note

Test runtimes refer to average seconds per single iteration on a single A100 40gb GPU.

Test0 [Walking 1 bit] - This test changes one bit at a time in memory to see if it goes to a different memory location. It is designed to test the address wires. Runtime: ~3 seconds.

Test1 [Address check] - Each Memory location is filled with its own address followed by a check to see if the value in each memory location still agrees with the address. Runtime: < 1 second.

Test 2 [Moving inversions, ones&zeros] - This test uses the moving inversions algorithm from memtest86 with patterns of all ones and zeros. Runtime: ~4 seconds.

Test 3 [Moving inversions, 8 bit pat] - Same as test 1 but uses a 8 bit wide pattern of “walking” ones and zeros. Runtime: ~4 seconds.

Test 4 [Moving inversions, random pattern] - Same algorithm as test 1 but the data pattern is a random number and it's complement. A total of 60 patterns are used. The random number sequence is different with each pass so multiple passes can increase effectiveness. Runtime: ~2 seconds.

Test 5 [Block move, 64 moves] - This test moves blocks of memory. Memory is initialized with shifting patterns that are inverted every 8 bytes. Then these blocks of memory are moved around. After the moves are completed the data patterns are checked. Runtime: ~1 second.

Test 6 [Moving inversions, 32 bit pat] - This is a variation of the moving inversions algorithm that shifts the data pattern left one bit for each successive address. To use all possible data patterns 32 passes are made during the test. Runtime: ~155 seconds.

Test 7 [Random number sequence] - A 1MB block of memory is initialized with random patterns. These patterns and their complements are used in moving inversion tests with rest of memory. Runtime: ~2 seconds.

Test 8 [Modulo 20, random pattern] - A random pattern is generated. This pattern is used to set every 20th memory location in memory. The rest of the memory location is set to the compliment of the pattern. Repeat this for 20 times and each time the memory location to set the pattern is shifted right. Runtime: ~10 seconds.

Test 9 [Bit fade test, 2 patterns] - The bit fade test initializes all memory with a pattern and then sleeps for 1 minute. Then memory is examined to see if any memory bits have changed. All ones and all zero patterns are used. Runtime: ~244 seconds.

Test10 [Memory stress] - A random pattern is generated and a large kernel is launched to set all memory to the pattern. A new read and write kernel is launched immediately after the previous write kernel to check if there is any errors in memory and set the

memory to the compliment. This process is repeated for 1000 times for one pattern. The kernel is written as to achieve the maximum bandwidth between the global memory and GPU. Runtime: ~6 seconds.

Note

By default Test7 and Test10 alternate for a period of 10 minutes. If any errors are detected the diagnostic will fail.

Supported Parameters

Parameter	Syntax	Default
test0	boolean	false
test1	boolean	false
test2	boolean	false
test3	boolean	false
test4	boolean	false
test5	boolean	false
test6	boolean	false
test7	boolean	true
test8	boolean	false
test9	boolean	false
test10	boolean	true
test_duration	seconds	600

Sample Commands

Run test7 and test10 for 10 minutes (this is the default):

```
$ dcgmi diag -r 4
```

Run each test serially for 1 hour then display results:

```
$ dcgmi diag -r 4 -p \
```

```
memtest.test0=true\;memtest.test1=true\;memtest.test2=true\;memtest.test3=true\;me
```

Run test0 for one minute 10 times, displaying the results each minute:

```
$ dcgmi diag --iterations 10 \  
-r 4 -p
```

```
memtest.test0=true\;memtest.test7=false\;memtest.test10=false\;memtest.test_durati
```