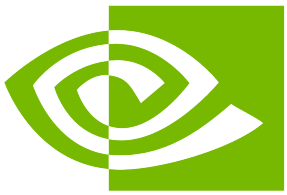


Multi-Process Service

Release 615

NVIDIA Corporation

Sep 09, 2026

Contents

The Multi-Process Service (MPS) is a lightweight runtime service, designed to transparently enable co-operative CUDA multi-process and multi-application workflows on NVIDIA GPUs. It consists of several components:

- `nvidia-cuda-mps-control` (Control Daemon Process) – The control daemon is responsible for starting and stopping the server, as well as coordinating connections between clients and servers.
- `nvidia-cuda-mps-server` (Server Process) – The server is the clients' shared connection to the GPU and owner of the GPU scheduling resources.
- `libcuda.so` (Client Runtime) – The MPS client runtime is built into the CUDA Driver library and may be used transparently by any CUDA application.

Enabling MPS provides the benefit of improved GPU utilization and reduced GPU context switching. MPS also provides memory and SM partitioning capabilities, as well as priority and dynamic resource adjustment. To learn more, start with the *Quick Start*.

MPS v3 is a new, opt-in control daemon interface that replaces the interactive shell of Legacy MPS v2 with a scriptable module verb command syntax, named servers and namespaces, and file-based configuration. All existing Legacy MPS v2 functionality continues to work unchanged; MPS v3 is selected explicitly, see *MPS v3 Interface* for more details.

Chapter 1. Organization of This Document

This document is organized as follows:

- **Quick Start** – describes how to get started.
- **When to Use MPS** – describes what factors to consider when choosing to run an application with or choosing to deploy MPS for your users.
- **Architecture** – describes the client-server architecture of MPS in detail and how it multiplexes clients onto the GPU.
- **Common Tasks** – describes how to start, stop, and administer MPS, showing the Legacy MPS v2 and MPS v3 commands side by side for each task.
- **Troubleshooting Guide** – lists common MPS server and client errors along with their causes and recommended actions.
- **Legacy MPS v2 Interface** – reference for the Legacy MPS v2 control daemon commands, environment variables, and utilities.
- **MPS v3 Interface** – reference for the MPS v3 command-line interface, covering servers, clients, devices, SM partitions, and features.
- **MPS v3 Namespaces** – explains how namespaces subdivide a server's resources and how clients are routed to them.
- **MPS v3 Configuration File** – describes the TOML file used to define servers, namespaces, partitions, and features at daemon startup.
- **MPS v3 Memory Partitioning** – describes how MPS v3 partitions device memory across clients.
- **Appendix: Environment Variables** – reference for the environment variables that configure MPS clients and daemons.
- **Appendix: Logging** – describes the MPS log files, their contents, and their locations.

1.1. See Also

- Manpage for `nvidia-cuda-mps-control (1)`

1.1.1. Quick Start

For an in-depth guide on MPS usage, refer to *Common Tasks*.

For an in-depth guide on commands, environment variables, and more, refer to *Legacy MPS v2 Interface*.

1.1.1.1 Starting the MPS controller

To start the MPS controller as a daemon (recommended):

Legacy MPS v2

```
nvidia-cuda-mps-control -d
```

MPS v3

Protocol version 3 can be selected with the `-p/--protocol` flag:

```
nvidia-cuda-mps-control -d -p 3
```

Or with the `CUDA_MPS_PROTOCOL_VERSION` environment variable:

```
export CUDA_MPS_PROTOCOL_VERSION=3
nvidia-cuda-mps-control -d
```

1.1.1.2 Launching an application under MPS

When the MPS controller is active, CUDA applications will use MPS:

```
./cuda_application
```

This can be verified with the `ps` command while the application is running:

Legacy MPS v2

```
echo ps | nvidia-cuda-mps-control
```

MPS v3

```
nvidia-cuda-mps-control client list --all
```

This can also be verified by using `nvidia-smi` after the application is launched to verify the existence of the `nvidia-cuda-mps-server` process. While the application is running, it will also appear in `nvidia-smi` with the `M+C` type specified.

1.1.1.3 Quitting MPS

To quit the MPS controller and any associated MPS servers:

Legacy MPS v2

```
echo quit | nvidia-cuda-mps-control
```

MPS v3

```
nvidia-cuda-mps-control -q
```

1.1.1.4 Explicitly starting an MPS server

To explicitly start an MPS server for user $UID:

Legacy MPS v2

```
echo start_server -uid $UID | nvidia-cuda-mps-control
```

MPS v3

```
nvidia-cuda-mps-control server create --uid=$UID
```

Note that servers are started implicitly when a CUDA application is launched while the MPS controller is active.

1.1.1.5 Starting multiple control/server pairs

Legacy MPS v2

To start multiple MPS servers for the same user, you can use different CUDA MPS directories to start different controllers.

```
export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps-0 # Select a location that's
↪accessible to the given $UID
export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log-0 # Select a location that's
↪accessible to the given $UID
nvidia-cuda-mps-control -d # Start an MPS controller on pipe 0
```

```
export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps-1 # Select a location that's
↪accessible to the given $UID
export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log-1 # Select a location that's
↪accessible to the given $UID
nvidia-cuda-mps-control -d # Start an MPS controller on pipe 1
```

MPS v3

A single MPS v3 daemon manages multiple named servers directly, so there is no need to start multiple control daemons on separate pipe directories:

```
nvidia-cuda-mps-control server create pipe0
nvidia-cuda-mps-control server create pipe1
```

Each server gets its own pipe directory automatically:

```
$ nvidia-cuda-mps-control server get pipe0 pipe-directory
/run/nvidia-mps/pipe0

$ nvidia-cuda-mps-control server get pipe1 pipe-directory
/run/nvidia-mps/pipe1
```

To start an application under one of the controllers, the same CUDA_MPS_PIPE_DIRECTORY must be set for the application as the controller.

Legacy MPS v2

```
CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps-0 CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-
↪log-0 ./cuda_application # This will use the MPS controller on pipe 0
CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps-1 CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-
↪log-1 ./cuda_application # This will use the MPS controller on pipe 1
```

MPS v3

```
# This will use the "pipe0" server
CUDA_MPS_PIPE_DIRECTORY=/run/nvidia-mps/pipe0 ./cuda_application

# This will use the "pipe1" server
CUDA_MPS_PIPE_DIRECTORY=/run/nvidia-mps/pipe1 ./cuda_application
```

1.1.1.6 Starting an MPS server with Locality Domains

To start a server configured to use MLOPart on supported devices for user $UID:

Legacy MPS v2

```
echo start_server -uid $UID -mlopart | nvidia-cuda-mps-control
```

MPS v3

MLOPart is renamed to locality domains in MPS v3, and is enabled per-server instead of per-server-start:

```
nvidia-cuda-mps-control server create --uid=$UID --locality-domains=true
```

For more information, refer to *Locality Domains* and *Locality Domains*.

1.1.1.7 Enabling static SM partitioning

Legacy MPS v2

Static SM partitioning must be enabled at MPS controller start time:

```
nvidia-cuda-mps-control -d -S
```

MPS v3

MPS v3 does not require a daemon-startup flag; SM partitions are created per-device once the daemon and server are running. The default UID-based server (auto-named `uid_$UID`) can be used directly:

```
nvidia-cuda-mps-control -d -p 3
nvidia-cuda-mps-control server create --uid=$UID
nvidia-cuda-mps-control sm-partition create large --server=uid_$UID --
↪device=0 --chunks=4
```

For more information, refer to *Static SM Partitioning* and *SM Partitions*.

1.1.2. When to Use MPS

1.1.2.1 Identifying Candidate Applications

MPS is useful when each application process does not generate enough work to saturate the GPU. Multiple processes can be run per node using MPS to enable more concurrency. Applications like this are identified by having a small number of blocks-per-grid.

Further, if the application shows a low GPU occupancy because of a small number of threads-per-grid, performance improvements may be achievable with MPS.Using fewer blocks-per-grid in the kernel invocation and more threads-per-block to increase the occupancy per block is recommended. MPS allows the leftover GPU capacity to be occupied with CUDA kernels running from other processes.

These cases arise in strong-scaling situations, where the compute capacity (node, CPU core and/or GPU count) is increased while the problem size is held fixed. Though the total amount of computation work stays the same, the work per process decreases and may underutilize the available compute capacity while the application is running. With MPS, the GPU will allow kernel launches from different processes to run concurrently and remove an unnecessary point of serialization from the computation.

1.1.2.2 Considerations

1.1.2.2.1 System Considerations

1.1.2.2.1.1 Limitations

- MPS is only supported on the Linux and QNX operating systems. The MPS server will fail to start when launched on an operating system other than Linux.
- Exclusive-mode restrictions are applied to the MPS server, not MPS clients. GPU compute modes are not supported on Tegra platforms.
- Only one user on a system may have an active MPS server.
- The MPS control daemon will queue MPS server activation requests from separate users, leading to serialized exclusive access of the GPU between users regardless of GPU exclusivity settings.
- All MPS client behavior will be attributed to the MPS server process by system monitoring and accounting tools (for example, `nvidia-smi`, NVML API).

1.1.2.2.1.2 GPU Compute Modes

Three Compute Modes are supported via settings accessible in `nvidia-smi`:

- `PROHIBITED` – the GPU is not available for compute applications.

- EXCLUSIVE_PROCESS — the GPU is assigned to only one process at a time, and individual process threads may submit work to the GPU concurrently.
- DEFAULT – multiple processes can use the GPU simultaneously. Individual threads of each process may submit work to the GPU simultaneously.

Using MPS effectively causes EXCLUSIVE_PROCESS mode to behave like DEFAULT mode for all MPS clients using the same CUDA_MPS_PIPE_DIRECTORY. MPS will always allow multiple clients to use the GPU via the MPS server.

When using MPS it is recommended to use EXCLUSIVE_PROCESS mode to ensure that only a single MPS server is using the GPU, which provides additional insurance that the MPS server is the single point of arbitration between all CUDA processes for that GPU.

1.1.2.2.2 Application Considerations

- Only 64-bit applications are supported. The MPS server will fail to start if the CUDA application is not 64-bit. The MPS client will fail CUDA initialization.
- If an application uses the CUDA driver API, then it must use headers from CUDA 4.0 or later (that is, it must not have been built by setting CUDA_FORCE_API_VERSION to an earlier version). Context creation in the client will fail if the context version is older than 4.0.
- Dynamic parallelism is not supported. CUDA module load will fail if the module uses dynamic parallelism features.
- MPS server only supports clients running with the same UID as the server. The client application will fail to initialize if the server is not running with the same UID. Volta MPS may be launched in `-multiuser-server` mode to allow clients under different UIDs to connect to a single MPS server launched under the root user while dropping isolation between users. Refer to *Server* for details regarding `-multiuser-server` mode.
- Terminating an MPS client without synchronizing with all outstanding GPU work (via Ctrl-C / program exception such as segfault / signals, etc.) can leave the MPS server and other MPS clients in an undefined state, which may result in hangs, unexpected failures, or corruptions.
- MPS Client Termination, CUDA IPC and cooperative launches are not supported with MPS on Tegra platforms.

1.1.2.2.3 Memory Protection and Error Containment

MPS client processes have fully isolated GPU address spaces. MPS supports a limited form of error containment:

- A fatal GPU fault generated by a Volta MPS client process will be contained within the subset of GPUs shared between all clients with the fatal fault-causing GPU.
- A fatal GPU fault generated by a Volta MPS client process will be reported to all the clients running on the subset of GPUs in which the fatal fault is contained, without indicating which client generated the error. Note that it is the responsibility of the affected clients to exit after being informed of the fatal GPU fault.
- Clients running on other GPUs remain unaffected by the fatal fault and will run as normal until completion.
- Once a fatal fault is observed, the MPS server will wait for all the clients associated with the affected GPUs to exit, prohibiting new client connecting to those GPUs from joining. The status of the MPS server changes from ACTIVE to FAULT. When all the existing clients associated with the affected GPUs have exited, the MPS server will recreate the GPU contexts on the affected

GPUs and resume processing client requests to those GPUs. The MPS server status changes back to ACTIVE, indicating that it is able to process new clients.

For example, if your system has devices 0, 1, and 2, and if there are four clients client A, client B, client C, and client D connected to the MPS server: client A runs on device 0, client B runs on device 0 and 1, client C runs on device 1, client D runs on device 2. If client A triggers a fatal GPU fault:

- Since device 0 and device 1 share a comment client, client B, the fatal GPU fault is contained within device 0 and 1.
- The fatal GPU fault will be reported to all the clients running on device 0 and 1, that is, client A, client B, and client C.
- Client D running on device 2 remain unaffected by the fatal fault and continue to run as normal.
- The MPS server will wait for client A, client B, and client C to exit and reject any new client requests will be rejected with error CUDA_ERROR_MPS_SERVER_NOT_READY while the server status is FAULT. After client A, client B, and client C have exited, the server recreates the GPU contexts on device 0 and device 1 and then resumes accepting client requests on all devices. The server status becomes ACTIVE again.

Information about the fatal GPU fault containment will be logged, including:

- If the fatal GPU fault is a fatal memory fault, the PID of the client which triggered the fatal GPU memory fault.
- The device IDs of the devices which are affected by this fatal GPU fault.
- The PIDs of the clients which are affected by this fatal GPU fault. The status of each affected client becomes INACTIVE and the status of the MPS server becomes FAULT.
- The messages indicating the successful recreation of the affected devices after all the affected clients have exited.

CUDA API errors generated on the CPU in the CUDA Runtime or CUDA Driver are delivered only to the calling client.

1.1.2.2.4 MPS on Multi-GPU Systems

The MPS server supports using multiple GPUs. On systems with more than one GPU, you can use CUDA_VISIBLE_DEVICES to enumerate the GPUs you would like to use. Refer to *Appendix: Environment Variables* for more details.

1.1.2.2.5 Dynamic Execution Resource Provisioning

MPS supports limited execution resource provisioning. The client contexts can be set to only use a portion of the available threads. The provisioning capability is commonly used to achieve two goals:

- **Reduce client memory footprint:** Since each MPS client process has fully isolated address space, each client context allocates independent context storage and scheduling resources. Those resources scale with the amount of threads available to the client. By default, each MPS client has all available threads useable. As MPS is usually used with multiple processes running simultaneously, making all threads accessible to every client is often unnecessary, and therefore wasteful to allocate full context storage. Reducing the number of threads available will effectively reduce the context storage allocation size.
- **Improve QoS:** The provisioning mechanism can be used as a classic QoS mechanism to limit available compute bandwidth. Reducing the portion of available threads will also concentrate the work submitted by a client to a set of SMs, reducing destructive interference with other clients' submitted work.

Setting the limit does not reserve dedicated resources for any MPS client context. It simply limits how much resources can be used by a client context. Kernels launched from different MPS client contexts may execute on the same SM, depending on load-balancing.

By default, each client is provisioned to have access to all available threads. This will allow the maximum degree of scheduling freedom, but at a cost of higher memory footprint due to wasted execution resource allocation. The memory usage of each client process can be queried through `nvidia-smi`.

The provisioning limit can be set via a few different mechanisms for different effects. These mechanisms are categorized into two mechanisms: active thread percentage and programmatic interface. In particular, partitioning via active thread percentage are categorized into two strategies: uniform partitioning and non-uniform partitioning.

The limit constrained by the uniform active thread percentage is configured for a client process when it starts and cannot be changed for the client process afterwards. The executed limit is reflected through device attribute `cudaDevAttrMultiProcessorCount` whose value remains unchanged throughout the client process.

- The MPS control utility provides 2 sets of commands to set/query the limit of all future MPS clients. Refer to *nvidia-cuda-mps-control* for more details.
- Alternatively, the limit for all future MPS clients can be set by setting the environment variable `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` for the MPS control process. Refer to *MPS control daemon level* for more details.
- The limit can be further constrained for new clients by solely setting the environment variable `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` for a client process. Refer to *client process level* for more details.

The limit constrained by the non-uniform active thread percentage is configured for every client CUDA context and can be changed throughout the client process. The executed limit is reflected through device attribute `cudaDevAttrMultiProcessorCount` whose value returns the portion of available threads that can be used by the client CUDA context current to the calling thread.

- The limit constrained by the uniform partitioning mechanisms can be further constrained for new client CUDA contexts by setting the environment variable `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` in conjunction with the environment variable `CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PARTITIONING`. Refer to *client CUDA context level* and *CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PARTITIONING* for more details.

The limit constrained by the programmatic partitioning is configured for a client CUDA context created via `cuCtxCreate_v3()` with the execution affinity `CUexecAffinityParam` which specifies the number of SMs that the context is limited to use. The executed limit of the context can be queried through `cuCtxGetExecAffinity()`. Refer to *Best Practices for SM Partitioning* for more details.

A common provisioning strategy is to uniformly partition the available threads equally to each MPS client processes (i.e., set active thread percentage to 100% / n, for n expected MPS client processes). This strategy will allocate close to the minimum amount of execution resources, but it could restrict performance for clients that could occasionally make use of idle resources.

A more optimal strategy is to uniformly partition the portion by half of the number of expected clients (i.e., set active thread percentage to 100% / 0.5n) to give the load balancer more freedom to overlap execution between clients when there are idle resources.

The near optimal provision strategy is to non-uniformly partition the available threads based on the workloads of each MPS clients (i.e., set active thread percentage to 30% for client 1 and set active thread percentage to 70 % client 2 if the ratio of the client 1 workload and the client2 workload is 30%: 70%). This strategy will concentrate the work submitted by different clients to disjoint sets of the SMs and effectively minimize the interference between work submissions by different clients.

The most optimal provision strategy is to precisely limit the number of SMs to use for each MPS clients knowing the execution resource requirements of each client (i.e., 24 SMs for client1 and 60 SMs for client 2 on a device with 84 SMs). This strategy provides finer grained and more flexible control over the set of SMs the work will be running on than the active thread percentage.

If the active thread percentage is used for partitioning, the limit will be internally rounded down to the nearest hardware supported thread count limit. If the programmatic interface is used for partitioning, the limit will be internally rounded up to the nearest hardware supported SM count limit.

1.1.2.2.6 Limits

1.1.2.2.6.1 Client-Server Connection Limits

Each MPS server supports up to 60 client CUDA contexts per-device when using the default CUDA_DEVICE_MAX_CONNECTIONS limit of 2. These contexts may be distributed over multiple processes but it's recommended to use a single context (the primary device context) per process. Increasing CUDA_DEVICE_MAX_CONNECTIONS will decrease the total number of MPS clients possible per server. If the connection limit is exceeded, the CUDA application will fail to create a CUDA Context and return an API error from `cuCtxCreate()` or the first CUDA Runtime API call that triggers context creation. Failed connection attempts will be logged by the MPS server.

1.1.2.2.6.2 MPS Device Memory Limit

Users can enforce clients to adhere to allocate device memory up to a preset limit. This mechanism provides a facility to fractionalize GPU memory across MPS clients that run on the specific GPU, which enables scheduling and deployment systems to make decisions based on the memory usage for the clients. If a client attempts to allocate memory beyond the preset limit, the cuda memory allocation calls will return out of memory error. The memory limit specific will also account for CUDA internal device allocations which will help users make scheduling decisions for optimal GPU utilization. This can be accomplished through a hierarchy of control mechanisms for users to limit the pinned device memory on MPS clients. The `default` limit setting would enforce a device memory limit on all the MPS clients of all future MPS Servers spawned. The `per server` limit setting allows finer grained control on the memory resource limit whereby users have the option to set memory limit selectively using the server PID and thus all clients of the server. Additionally, MPS clients can further constrain the memory limit setting from the server by using the `CUDA_MPS_PINNED_DEVICE_MEM_LIMIT` environment variable.

1.1.2.2.7 Interaction with Tools

1.1.2.2.7.1 CUDA-GDB

GPU coredumps can be generated and debugged using CUDA-GDB. Refer to the CUDA-GDB documentation for usage instructions.

Debugging with cuda-gdb can only be done without MPS. To bypass MPS, set the `CUDA_MPS_PIPE_DIRECTORY` environment variable to an empty string or a non-existent directory.

1.1.2.2.7.2 Compute Sanitizer

The `compute-sanitizer` tool is supported on MPS. Refer to the `compute-sanitizer` documentation for usage instructions.

1.1.2.2.7.3 Profiling

CUDA profiling tools (such as nvprof and Nvidia Visual Profiler) and CUPTI-based profilers are supported under MPS. Refer to the MPS Profiling documentation for more details.

1.1.2.2.8 Client Early Termination

Terminating an MPS client via CTRL-C or signals is not supported and will lead to undefined behavior. The user must guarantee that the MPS client is idle, by calling either `cudaDeviceSynchronize` or `cudaStreamSynchronize` on all streams, before the MPS client can be terminated. Early termination of a MPS client without synchronizing all outstanding GPU work may leave the MPS server in an undefined state and result in unexpected failures, corruptions, or hangs; as a result, the affected MPS server and all its clients must be restarted.

A user can instruct the MPS server to terminate the MPS client process, regardless of whether the CUDA contexts are idle or not, by using the control command `terminate_client <server PID> <client PID>` in Legacy MPS v2, or `client terminate <client PID>` in MPS v3. This mechanism is safe to invoke without affecting the MPS server or its other MPS clients. The command sends a request to the MPS server which terminates the CUDA contexts of the target MPS client process on behalf of the user and returns after the MPS server has completed the request. The return value is `CUDA_SUCCESS` if the CUDA contexts of the target MPS client process have been successfully terminated; otherwise, a CUDA error describing the failure state. When the MPS server starts handling the request, each MPS client context running in the target MPS client process becomes `INACTIVE`; the status changes will be logged by the MPS server. Upon successful completion of the client termination, the target MPS client process will observe a sticky error `CUDA_ERROR_MPS_CLIENT_TERMINATED`, and it becomes safe to kill the target MPS client process with signals such as `SIGKILL` without affecting the rest of the MPS server and its MPS clients. Note that the MPS server is not responsible for killing the target MPS client process after the sticky error.

If the user wants to terminate the GPU work of a MPS client process that is running inside a PID namespace different from the MPS control's PID namespace, such as an MPS client process inside a container, the user must use the PID of the target MPS client process translated into the MPS control's PID namespace. For example, the PID of an MPS client process inside the container is 6, and the PID of this MPS client process in the host PID namespace is 1024; the user must use 1024 to terminate the GPU work of the target MPS client process.

The common workflow for terminating the client application `nbody`:

Get the status of the current active MPS clients.

Legacy MPS v2

```
$ echo "ps" | nvidia-cuda-mps-control
PID ID SERVER DEVICE NAMESPACE COMMAND
9741 0 6472 GPU-cb1213a3-d6a4-be7f 4026531836 ./nbody
9743 0 6472 GPU-cb1213a3-d6a4-be7f 4026531836 ./matrixMul
```

MPS v3

```
$ nvidia-cuda-mps-control client list --all
PID   SERVER  DEVICE                  MAWS-NS  CMD
9741  uid_0   GPU-cb1213a3-d6a4-be7f  default  ./nbody
9743  uid_0   GPU-cb1213a3-d6a4-be7f  default  ./matrixMul
```

Terminate the target client by its PID in the host PID namespace:

Legacy MPS v2

In Legacy MPS v2, `terminate_client` takes both the server PID and the client PID:

```
$ echo "terminate_client 6472 9741" | nvidia-cuda-mps-control
#wait until terminate_client to return
#upon successful termination 0 is returned
0
```

MPS v3

In MPS v3, `client terminate` takes only the client PID:

```
$ nvidia-cuda-mps-control client terminate 9741
```

Now it is safe to kill nbody:

```
$ kill -9 9741
```

1.1.2.2.9 Client Priority Level Control

Users are normally only able to control the GPU priority level of their kernels by using the `cudaStreamCreateWithPriority()` API while the program is being written. The user can use the control command `set_default_client_priority <Priority Level>` to map the stream priorities of a given client to a different range of internal CUDA priorities. Changes to this setting do not take effect until the next client connection to the server is opened. The user can also set the `CUDA_MPS_CLIENT_PRIORITY` environment variable before starting the control daemon or any given client process to set this value.

The allowed priority level values are `0` (normal) and 1 (below normal). Lower numbers map to higher priorities to match the behavior of the Linux kernel scheduler.

> **Note**
>
> CUDA priority levels are not guarantees of execution order–they are only a performance hint to the CUDA Driver.

For example:

- Process A is launched at Normal priority and only uses the default CUDA Stream, which has the lowest priority of 0.
- Process B is launched at Below Normal priority and uses streams with custom Stream priority values, such as -3.

Without this feature, the streams from Process B would be executed first by the CUDA driver. However, with the Client Priority Level feature, the streams from Process A will take precedence.

1.1.2.2.10 Locality Domains

On some Blackwell and newer GPUs, users are able to create locality domains.

Note

On Legacy MPS v2, locality domains and their associated commands and output are called MLOPart (for example, the `-mlopart` server option and the `MD` device attribute).

Legacy MPS v2

Enable locality domains with the `-mlopart` server option:

```
echo "start_server -uid <UID> -mlopart" | nvidia-cuda-mps-control
```

MPS v3

Enable locality domains per-server. Refer to *Locality Domains* for details.

```
nvidia-cuda-mps-control server create --uid=<UID> --locality-domains=true
```

When a server is created with this option, GPUs that are capable of creating locality domains will do so. Locality domain devices are CUDA devices that are derived from another GPU, and have been optimized for lower-latency and/or higher-bandwidth by ensuring that compute and memory resources are physically colocated.

When using locality domain devices, users will note that there are multiple such devices for each underlying device, with the locality domain devices having fewer compute resources and available memory, and being optimized for memory locality.

1.1.2.2.10.1 Locality Domain Device Enumeration

Because a client created by a locality-domain enabled server may have more CUDA devices than it would otherwise, there is ambiguity in the device enumeration. This has consequences for other MPS control commands, as well as `CUDA_VISIBLE_DEVICES`. To resolve this ambiguity, use the device enumeration command to determine the device layout after taking locality domains into account.

Legacy MPS v2

```
$ echo device_query | nvidia-cuda-mps-control
Default
Device Ordinal  PCI IDs        UUID               Name
 ↪Attributes
0               0000:20.00.00  GPU-1d925fce-3b7d  NVIDIA B300
1               0000:48.00.00  GPU-468af2de-d4f0  NVIDIA B300
2               0000:57.00.00  GPU-a74f1c76-1ca2  NVIDIA B300
3               0000:66.00.00  GPU-4fe2ee0e-71e4  NVIDIA B300
4               0000:a2.00.00  GPU-ced3eaa1-26b0  NVIDIA B300
5               0000:c8.00.00  GPU-118a6ac9-86d2  NVIDIA B300
6               0000:d6.00.00  GPU-1ec93f24-564f  NVIDIA B300

Server 908527
Device Ordinal  PCI IDs        UUID               Name
 ↪Attributes
N/A             0000:20.00.00  GPU-1d925fce-3b7d  NVIDIA B300            M
0               0000:20.00.00  GPU-3b861d38-5e0c  NVIDIA B300 MLOPart 0  MD
```

(continues on next page)

(continued from previous page)

```
1               0000:20.00.00  GPU-d74bb67a-2db6  NVIDIA B300 MLOPart 1  MD
N/A             0000:48.00.00  GPU-468af2de-d4f0  NVIDIA B300            M
2               0000:48.00.00  GPU-5b669fd8-1170  NVIDIA B300 MLOPart 0  MD
3               0000:48.00.00  GPU-1e489f1a-6f85  NVIDIA B300 MLOPart 1  MD
N/A             0000:57.00.00  GPU-a74f1c76-1ca2  NVIDIA B300            M
4               0000:57.00.00  GPU-0384bea6-6424  NVIDIA B300 MLOPart 0  MD
5               0000:57.00.00  GPU-be428d68-2cea  NVIDIA B300 MLOPart 1  MD
N/A             0000:66.00.00  GPU-4fe2ee0e-71e4  NVIDIA B300            M
6               0000:66.00.00  GPU-2b3a2dae-3c97  NVIDIA B300 MLOPart 0  MD
7               0000:66.00.00  GPU-405be3d0-f1b5  NVIDIA B300 MLOPart 1  MD
N/A             0000:a2.00.00  GPU-ced3eaa1-26b0  NVIDIA B300            M
8               0000:a2.00.00  GPU-2b3fa6b7-1e62  NVIDIA B300 MLOPart 0  MD
9               0000:a2.00.00  GPU-ddf3743f-bed6  NVIDIA B300 MLOPart 1  MD
N/A             0000:c8.00.00  GPU-118a6ac9-86d2  NVIDIA B300            M
10              0000:c8.00.00  GPU-2dcf8afb-6f79  NVIDIA B300 MLOPart 0  MD
11              0000:c8.00.00  GPU-1458f72b-8fdb  NVIDIA B300 MLOPart 1  MD
N/A             0000:d6.00.00  GPU-1ec93f24-564f  NVIDIA B300            M
12              0000:d6.00.00  GPU-9cfe5596-8fec  NVIDIA B300 MLOPart 0  MD
13              0000:d6.00.00  GPU-6bef62f3-40d9  NVIDIA B300 MLOPart 1  MD
```

MPS v3

In MPS v3, use `device list`. Output shown for one underlying GPU; the remaining GPUs follow the same pattern.

```
$ nvidia-cuda-mps-control device list
Ordinal  Server  PCI ID         Device UUID        Name
 ↪    Total SMs  Total Chunks  Attributes
N/A      uid_0   0000:20.00.00  GPU-1d925fce-3b7d  NVIDIA B300
 ↪    128        16            LE
0        uid_0   0000:20.00.00  GPU-3b861d38-5e0c  NVIDIA B300 Locality
 ↪Domain 0  64         8             LD
1        uid_0   0000:20.00.00  GPU-d74bb67a-2db6  NVIDIA B300 Locality
 ↪Domain 1  64         8             LD
```

The device enumeration displays important information, including the device ordinals after taking locality domains into account, the PCI IDs (which can be used to determine which devices belong to the same GPU), the device UUIDs, device names, and attributes. In Legacy MPS v2, the attribute `MD` indicates a locality domain device while `M` indicates the underlying GPU; in MPS v3, `LD` indicates a locality domain device and `LE` indicates the underlying GPU with locality domains enabled.

The client listing is also capable of showing which device a process is using.

Legacy MPS v2

```
$ while1 -a &
[1] 52845
$ echo ps | nvidia-cuda-mps-control
PID       ID    SERVER    DEVICE             NAMESPACE      COMMAND
 ↪ATTRIBUTES
52845     1     52837     GPU-b13add01-c28c  4026531836     while1      MD
```

The attribute `MD` indicates that this is a locality domain device.

MPS v3

```
$ while1 -a &
[1] 52845
$ nvidia-cuda-mps-control client list --all
PID    SERVER  DEVICE             MAWS-NS  CMD
52845  uid_0   GPU-b13add01-c28c  default  while1
```

The MPS v3 client listing does not have an attributes column. The locality domain device is identified by its DEVICE UUID, which appears with the LD attribute in `device list`.

1.1.2.2.10.2 P2P with Locality Domains

Locality domain devices that are derived from the same underlying GPU have the special property of sharing GPU memory. They are capable of directly addressing each other's memory without enabling peer-to-peer access.

When interacting with memory on another device (a locality domain device or otherwise), the access modifier set by `cuMemSetAccess` is shared for all locality domain devices belonging to the same underlying GPU, and is the least restrictive access type that is set. For example, if a pointer is desired to be set as read-only by a locality domain device, that access type should be set for all locality domain devices belonging the same underlying GPU. If it is only set for a single locality domain device, then it will have no effect since other locality domain devices still have the default read/write permission. Similarly, if there are two locality domain devices, and one of them has memory with read-only permission to a pointer while another has no-access to a pointer, they will both have read-only permission.

1.1.2.2.10.3 Limitations of Locality Domains

There are some limitations when using locality domain devices:

- `nvidia-smi` is not aware of locality domain devices. For example, using `nvidia-smi -L` will not show locality domain devices, their UUIDs, or ordinals.
- Host allocations or allocations created through `cuMemAllocManaged` will see no benefit from using locality domain devices.
- When using NVLink Multicast, multiple locality domain devices from the same underlying device cannot join the same multicast group.
- Locality domain devices have less total memory than their underlying device. Each locality domain device tracks their free memory (queried through `cuMemGetInfo`) independently, however allocating memory on one locality domain device may affect the remaining free memory on another. Additionally, using `cuMemAllocManaged` may result in less free memory on any or all locality domain devices.

Additionally, there are some configurations in which locality domains are not supported:

- In driver version 590, locality domains are only supported on x86-based systems.
- In driver version 595 and newer, locality domains are also supported on ARM-based systems when CDMM is enabled (kernel module paramater `NVreg_CoherentGPUMemoryMode=driver`).
- Locality domains cannot be used in conjunction with static SM partitioning. On Legacy MPS v2, the `-mlopart` option of `start_server` is ignored if static partitioning is enabled.
- Locality domains cannot be used in conjunction with NVIDIA vGPU.
- MIG devices do not support locality domains. Using MIG on one GPU does not prevent using locality domains on another GPU.

1.1.2.2.11 Static SM Partitioning

On NVIDIA Ampere architecture and newer GPUs, users can create static SM partitions for MPS clients. This feature provides deterministic resource allocation and spatial isolation between clients by allowing users to explicitly control which SMs each client can access.

Legacy MPS v2

Static partitioning mode is enabled at MPS control daemon launch time using the `-S` or `--static-partitioning` command line parameter.

```
nvidia-cuda-mps-control -d -S
```

MPS v3

There is no special launch mode. Start the daemon and create a server to own the partitions.

```
nvidia-cuda-mps-control -d -p 3
nvidia-cuda-mps-control server create <name>
```

Static partitioning mode uses "chunks" as the partitioning unit. The size of a chunk depends on the GPU architecture:

- **dGPU**: 8 SMs on Hopper and newer GPUs (Hopper+ chunks are cluster-capable). On pre-Hopper GPUs, a chunk is 4 SMs on Legacy MPS v2 and 2 SMs on MPS v3.
- **iGPU**: 2 SMs on all architectures

1.1.2.2.11.1 Creating and Managing Partitions

Users create SM partitions by specifying the number of chunks. Upon successful creation, a unique partition ID is returned that can be used to assign clients to that partition:

Legacy MPS v2

Use the `sm_partition add` command with the full device UUID or a unique partial UUID of the device.

```
$ echo "sm_partition add GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65 7" | nvidia-
↪cuda-mps-control
GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

MPS v3

Use the `sm-partition create` command, naming the partition and its server.

```
$ nvidia-cuda-mps-control sm-partition create large --server=training --
↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65 --chunks=7
SM partition 'large' created on device GPU-74d43ed3-cdf7-e667-3644-
↪bf5b4f46ed65 (sm-count=56)
CUDA_MPS_SM_PARTITION=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Attempting to oversubscribe will result in an error message indicating the requested and available SM counts.

Legacy MPS v2

```
$ echo "sm_partition add GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65 20" |
 ↪nvidia-cuda-mps-control
Failed to fulfill the requested SM partition of 20 chunks, error CUDA_ERROR_
 ↪INVALID_RESOURCE_CONFIGURATION
```

MPS v3

```
$ nvidia-cuda-mps-control sm-partition create big --server=training --
 ↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65 --chunks=20
Error: Not enough free chunks on device GPU-74d43ed3-cdf7-e667-3644-
 ↪bf5b4f46ed65 to create SM partition 'big'.
```

To assign this partition to a client application, set the `CUDA_MPS_SM_PARTITION` environment variable:

```
$ export CUDA_MPS_SM_PARTITION=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
 ↪Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ./nbody
```

One or more MPS clients can run on the same partition. However, assigning more than one partition from the same device to the same client is prohibited and will fail with `CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION`. When static partitioning mode is enabled, MPS client applications must set this environment variable before initializing the MPS client. Attempting to run a client without setting the SM partition will fail on client initialization with `CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION`. Similarly, if the partition ID is invalid, client initialization will also fail with `CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION`.

View the current partitioning configuration:

Legacy MPS v2

Use the `lspart` command.

```
$ echo "lspart" | nvidia-cuda-mps-control
GPU          Partition                                free    used    free
 ↪used  clients
                                                      chunk   chunk   SM    SM
GPU-74d43ed3  -                                       0       8       74    56
 ↪   -
GPU-74d43ed3  Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA      -       7        -    56
 ↪   Yes
```

MPS v3

Use the `sm-partition list` command.

```
$ nvidia-cuda-mps-control sm-partition list --server=training --all
SERVER    DEVICE        NAME   CHUNKS  SM-COUNT  INCLUDE-REMAINDER  STATUS
 ↪    VISIBLE-UUID
training  GPU-74d43ed3  large  7       56        false              created
 ↪    Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
training  GPU-74d43ed3  -      8       74        -
 ↪unallocated  -
```

Remove the partition when it is no longer needed. The command will fail if the partition is still in use.

Legacy MPS v2

Specify the device UUID and the partition ID using `sm_partition rm`. It supports both `rm <device UUID>/<partition ID>` and `rm <device UUID> <partition ID>` formats.

```
$ echo "sm_partition rm GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" | nvidia-cuda-mps-control
Partition GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA removed
```

If the partition is still in use:

```
$ echo "sm_partition rm GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" | nvidia-cuda-mps-control
Partition GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA in use. Terminate all clients before
↪removing.
```

MPS v3

Specify the partition name, its server, and the device using `sm-partition delete`.

```
$ nvidia-cuda-mps-control sm-partition delete large --server=training --
↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65
SM partition 'large' deleted from device GPU-74d43ed3-cdf7-e667-3644-
↪bf5b4f46ed65
```

If the partition is still in use:

```
$ nvidia-cuda-mps-control sm-partition delete large --server=training --
↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65
Error: Cannot delete SM partition 'large': device GPU-74d43ed3-cdf7-e667-3644-
↪bf5b4f46ed65 has active clients.
```

1.1.2.2.11.2 Partition ID Determinism

Partition IDs are determined by the partition size, the command sequence order, and the distribution of reserved SMs at the time of partitioning. This means that repeating the same command sequence from the same initial state on systems with identical GPU SKUs will reliably produce the same partition IDs, enabling reproducible configurations across different systems.

1.1.2.2.11.3 Resource Distribution

Legacy MPS v2

When static partitioning mode is enabled, after all chunks are allocated, any remaining SMs are automatically distributed to the least recently created partition when the first client connects.

MPS v3

Remaining SMs are not distributed automatically. To include the remainder in a partition, create it with `--include-remainder=true`. Only one partition per device can claim the remainder; attempting to claim it from another partition fails with an error.

1.1.2.2.11.4 API Behavior with Static Partitioning

When a client is running on a static SM partition, CUDA API calls reflect the resources available to that partition:

- `cuDeviceGetAttribute` with `CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT` returns the total SM count of the partition.
- `cuOccupancyMaxActiveClusters` returns the cluster occupancy of the partition on Hopper and newer GPUs.

1.1.2.2.11.5 Limitations of Static SM Partitioning

Static SM partitioning has the following limitations and requirements:

- Static partitioning mode is only supported on NVIDIA Ampere architecture and newer GPUs.
- When static partitioning mode is enabled, all MPS clients must set the `CUDA_MPS_SM_PARTITION` environment variable before creating a CUDA context. Failure to do so will result in context creation failing with `CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION`.
- Assigning multiple partitions from the same device to a single client is prohibited and will fail with `CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION`.
- Partitions cannot be removed while clients are actively using them.
- Attempting to create partitions that exceed available SM resources will fail with an error message indicating the requested and available SM counts.
- Static SM partitioning cannot be used in conjunction with locality domains. On Legacy MPS v2, the `-mlopart` option of `start_server` is ignored if static partitioning is enabled.
- When static partitioning mode is enabled, dynamic resource provisioning will be ignored, this includes but not limited to MPS active thread percentage and per-context device multiprocessor partitioning.

1.1.3. Architecture

1.1.3.1 Background

CUDA is a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

A CUDA program starts by creating a CUDA context, either explicitly using the driver API or implicitly using the runtime API, for a specific GPU. The context encapsulates all the hardware resources necessary for the program to be able to manage memory and launch work on that GPU.

Launching work on the GPU typically involves copying data over to previously allocated regions in GPU memory, running a CUDA kernel that operates on that data, and then copying the results back from GPU memory into system memory. A CUDA kernel consists of a hierarchy of thread groups that execute in parallel on the GPU's compute engine.

All work on the GPU launched using CUDA is launched either explicitly into a CUDA stream, or implicitly using a default stream. A stream is a software abstraction that represents a sequence of commands, which may be a mix of kernels, copies, and other commands, that execute in order. Work launched in two different streams can execute simultaneously, allowing for coarse grained parallelism.

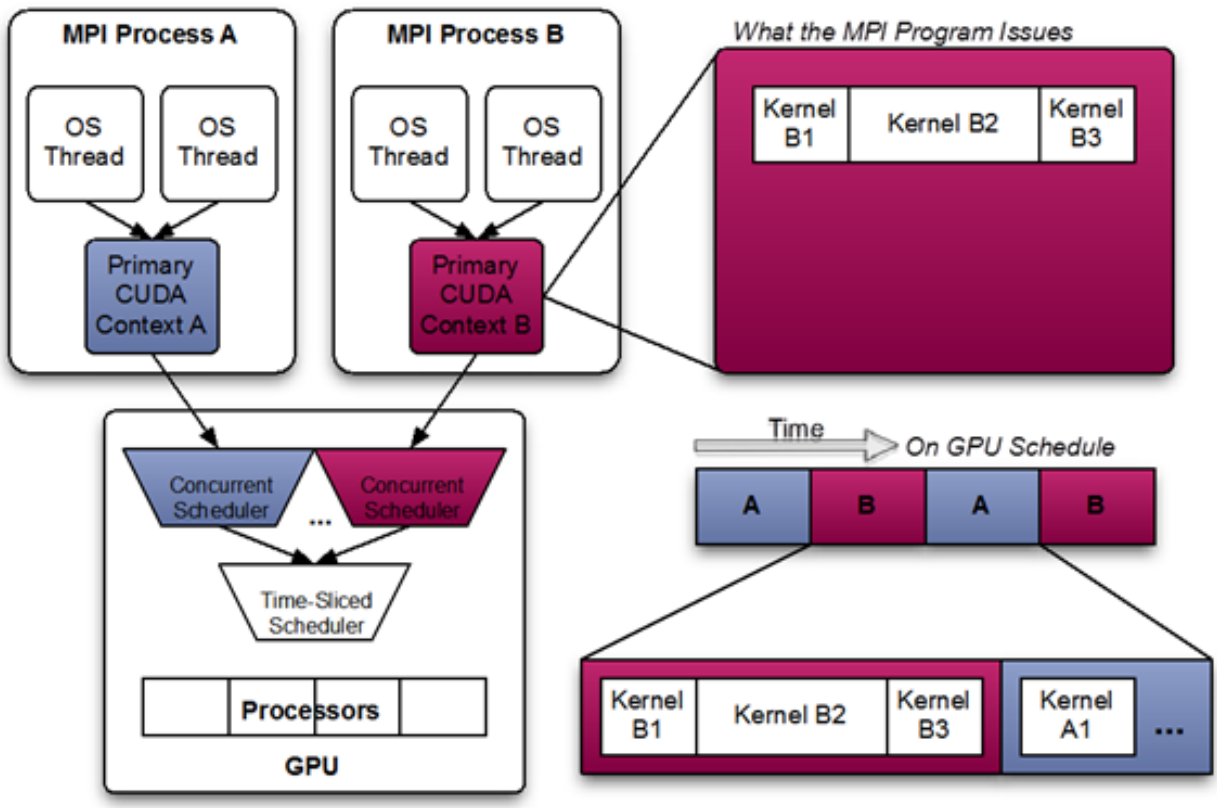
CUDA streams are aliased onto one or more 'work queues' on the GPU by the driver. Work queues are hardware resources that represent an in-order sequence of the subset of commands in a stream to be executed by a specific engine on the GPU, such as the kernel executions or memory copies. GPUs with Hyper-Q have a concurrent scheduler to schedule work from work queues belonging to a single CUDA context. Work launched to the compute engine from work queues belonging to the same CUDA context can execute concurrently on the GPU.

The GPU also has a time sliced scheduler to schedule work from work queues belonging to different CUDA contexts. Work launched to the compute engine from work queues belonging to different CUDA contexts cannot execute concurrently. This can cause underutilization of the GPU's compute resources if work launched from a single CUDA context is not sufficient to use up all resource available to it.

Additionally, within the software layer, to receive asynchronous notifications from the OS and perform asynchronous CPU work on behalf of the application the CUDA driver may create internal threads: an upcall handler thread and potentially a user callback executor thread.

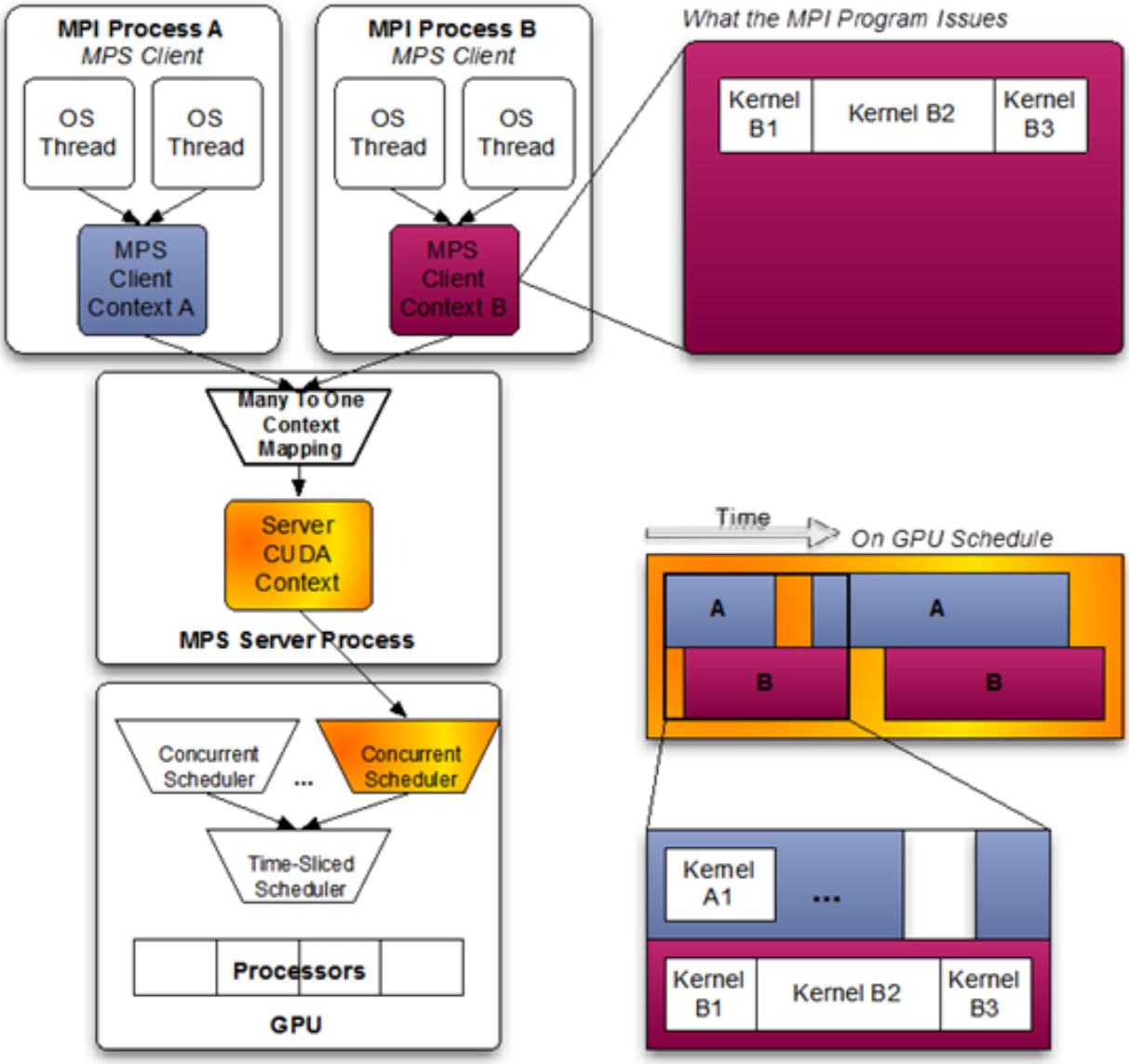
MPS clients submit work directly to the GPU without passing through the MPS server. Each MPS client owns its own GPU address space instead of sharing GPU address space with all other MPS clients and supports limited execution resource provisioning for Quality of Service (QoS).

1.1.3.2 Client-server Architecture



This diagram shows a likely schedule of CUDA kernels when running an MPI application consisting of multiple OS processes without MPS. Note that while the CUDA kernels from within each MPI process

may be scheduled concurrently, each MPI process is assigned a serially scheduled time-slice on the whole GPU.



Volta provides new hardware capabilities to reduce the types of hardware resources the MPS server must managed. A client CUDA context manages most of the hardware resources on Volta, and submits work to the hardware directly. The Volta MPS server mediates the remaining shared resources required to ensure simultaneous scheduling of work submitted by individual clients, and stays out of the critical execution path.

The communication between the MPS client and the MPS server is entirely encapsulated within the CUDA driver behind the CUDA API. As a result, MPS is transparent to the CUDA program.

MPS clients CUDA contexts retain their upcall handler thread and any asynchronous executor threads. The MPS server creates an additional upcall handler thread and creates a worker thread for each client.

1.1.3.3 Provisioning Sequence

1.1.3.3.1 Server

The MPS control daemon is responsible for the startup and shutdown of MPS servers. The control daemon allows at most one MPS server to be active at a time. When an MPS client connects to the control daemon, the daemon launches an MPS server if there is no server active. The MPS server is launched with the same user id as that of the MPS client.

If there is an MPS server already active and the user ID of the server and client match, then the control daemon allows the client to proceed to connect to the server. If there is an MPS server already active, but the server and client were launched with different user IDs, the control daemon requests the existing server to shutdown once all its clients have disconnected. Once the existing server has shutdown, the control daemon launches a new server with the same user ID as that of the new user's client process. This is shown in the figure above where user Bob starts client C' before a server is available. Only once user Alice's clients exit is a server created for user Bob and client C'.

The MPS control daemon does not shutdown the active server if there are no pending client requests. This means that the active MPS server process will persist even if all active clients exit. The active server is shutdown when either a new MPS client, launched with a different user id than the active MPS server, connects to the control daemon or when the work launched by the clients has caused a fault. This is shown in the example above, where the control daemon issues a server exit request to Alice's server only once user Bob starts client C, even though all of Alice's clients have exited.

The restriction of one Linux user per MPS server may be relaxed to avoid reprovisioning the MPS server on each new user request. Under this mode, clients from all Linux users will appear as clients from the root user and connect to the root MPS server. It is important to make sure that isolation between different users (including the root user) can be safely disregarded before enabling this mode. Clients from all users will share the same MPS log files. The same error containment rules (refer to *Memory Protection and Error Containment*) also apply in this mode across clients from all users. For example, a fatal fault from one client may bring down a different user's client that shares any GPU with the faulting client. To allow multiple Linux users share one MPS server, start the control daemon under superuser with the `-multiuser-server` option. This option is not supported on Tegra platforms.

An MPS server may be in one of the following states: `INITIALIZING`, `ACTIVE` or `FAULT`. The `INITIALIZING` state indicates that the MPS server is busy initializing and the MPS control will hold the new client requests in its queue. The `ACTIVE` state indicates the MPS server is able to process new client requests. The `FAULT` state indicates that the MPS server is blocked on a fatal fault caused by a client. Any new client requests will be rejected with error `CUDA_ERROR_MPS_SERVER_NOT_READY`.

A newly launched MPS server will be in the `INITIALIZING` state first. After successful initialization, the MPS server goes into the `ACTIVE` state. When a client encounters a fatal fault, the MPS server will transition from `ACTIVE` to `FAULT`. On pre-Volta MPS, the MPS server shuts down after encountering a fatal fault. On Volta MPS, the MPS server becomes `ACTIVE` again after all faulting clients have disconnected.

The control daemon executable also supports an interactive mode where a user with sufficient permissions can issue commands, for example to see the current list of servers and clients or startup and shutdown servers manually.

1.1.3.3.2 Client Attach/Detach

When CUDA is first initialized in a program, the CUDA driver attempts to connect to the MPS control daemon. If the connection attempt fails, the program continues to run as it normally would without MPS. If however, the connection attempt succeeds, the MPS control daemon proceeds to ensure that an MPS server, launched with same user ID as that of the connecting client, is active before returning to the client. The MPS client then proceeds to connect to the server.

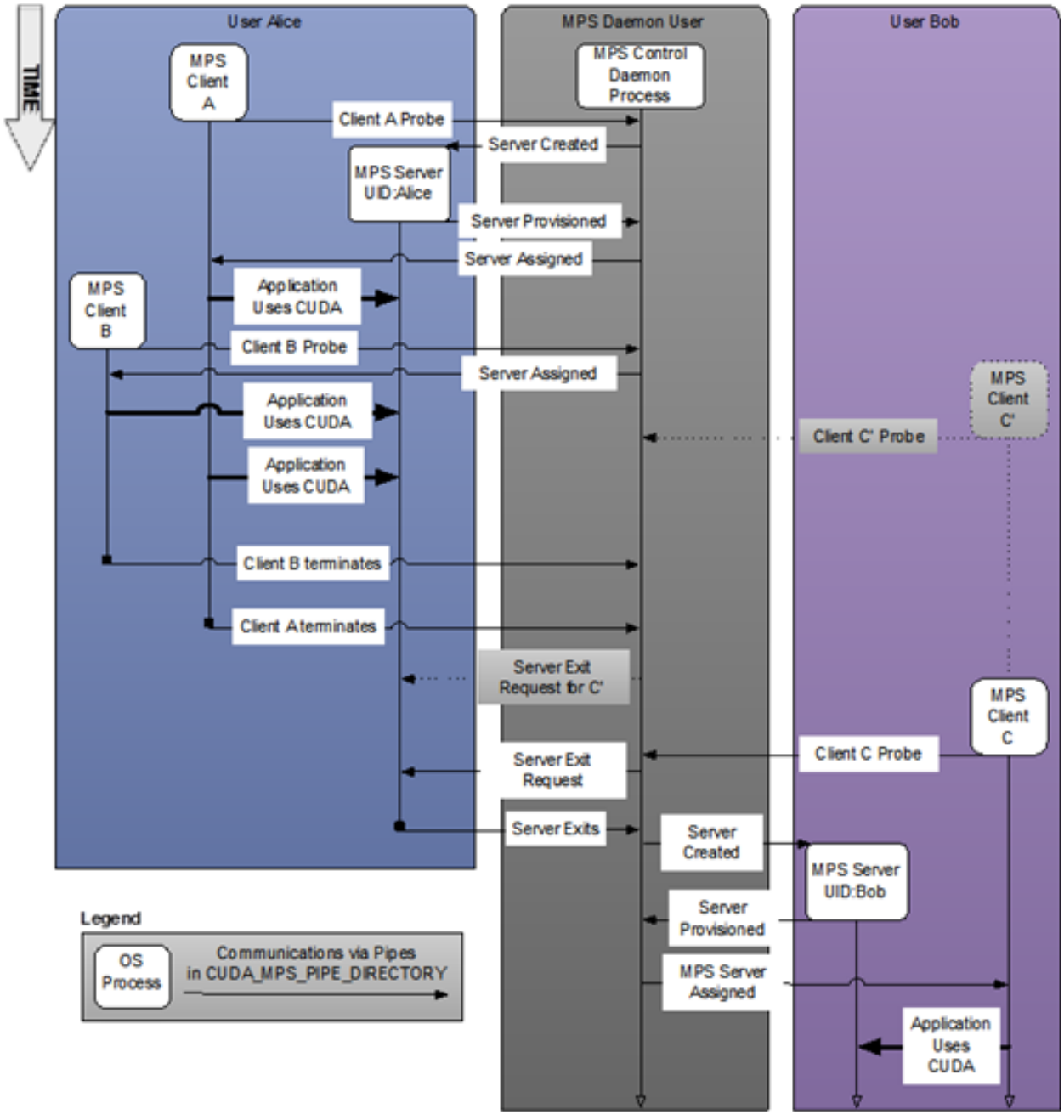


Figure 1: System-wide provisioning with multiple users.

All communication between the MPS client, the MPS control daemon, and the MPS server is done using UNIX domain sockets. The MPS server launches a worker thread to receive commands from the client. Successful client connection will be logged by the MPS server as the client status becomes ACTIVE. Upon client process exit, the server destroys any resources not explicitly freed by the client process and terminates the worker thread. The client exit event will be logged by the MPS server.

1.1.4. Common Tasks

This page covers common MPS administration and usage tasks. Where the commands differ between *Legacy MPS v2* and *MPS v3*, both are shown in the tabs below.

1.1.4.1 nvidia-cuda-mps-control

Typically stored under `/usr/bin` on Linux and QNX systems and typically run with superuser privileges, this control daemon is used to manage the `nvidia-cuda-mps-server` described in the following section. These are the relevant use cases:

Legacy MPS v2

```
$ nvidia-cuda-mps-control -d           # Start daemon in background process.

$ ps -ef | grep mps                    # Check if the MPS daemon is running,
↪for Linux.

$ pidin  | grep mps                    # See if the MPS daemon is running, for
↪QNX.

$ echo quit | nvidia-cuda-mps-control  # Shut the daemon down.

$ nvidia-cuda-mps-control -f           # Start daemon in foreground.

$ nvidia-cuda-mps-control -v           # Print version of control daemon
↪executable.

$ nvidia-cuda-mps-control -S           # Start daemon with static partitioning
↪mode enabled.
```

MPS v3

```
$ nvidia-cuda-mps-control -d -p 3      # Start daemon in background process.

$ nvidia-cuda-mps-control server list  # List running MPS servers.

$ nvidia-cuda-mps-control -q           # Shut the daemon down.

$ nvidia-cuda-mps-control -f -p 3      # Start daemon in foreground.

$ nvidia-cuda-mps-control -v           # Print version of control daemon
↪executable.
```

The nvidia-cuda-mps-control has a man page available as part of the installation:

```
man nvidia-cuda-mps-control
```

1.1.4.2 nvidia-cuda-mps-server

Typically stored under `/usr/bin` on Linux and QNX systems, this daemon is run under the same $UID as the client application running on the node. The `nvidia-cuda-mps-server` instances are created on-demand when client applications connect to the control daemon. The server binary should not be invoked directly, and instead the control daemon should be used to manage the startup and shutdown of servers.

The `nvidia-cuda-mps-server` process owns the CUDA context on the GPU and uses it to execute GPU operations for its client application processes. Due to this, when querying active processes via `nvidia-smi` (or any NVML-based application) `nvidia-cuda-mps-server` will appear as the active CUDA process rather than any of the client processes.

The version of the `nvidia-cuda-mps-server` executable can be printed with:

```
nvidia-cuda-mps-server -v
```

1.1.4.3 Starting and Stopping MPS on Linux

To view the daemon and server log files, refer to *Appendix: Logging*.

1.1.4.3.1 On a Multi-user System

To cause all users of the system to run CUDA applications via MPS you will need to set up the MPS control daemon to run when the system starts.

1.1.4.3.1.1 Starting MPS control daemon

As root, run the commands:

Legacy MPS v2

```
$ export CUDA_VISIBLE_DEVICES=0            # Select GPU 0.

$ nvidia-smi -i 0 -c EXCLUSIVE_PROCESS     # Set GPU 0 to exclusive mode.

$ nvidia-cuda-mps-control -d               # Start the daemon.
```

MPS v3

```
$ export CUDA_VISIBLE_DEVICES=0            # Select GPU 0.

$ nvidia-smi -i 0 -c EXCLUSIVE_PROCESS     # Set GPU 0 to exclusive mode.

$ nvidia-cuda-mps-control -d -p 3          # Start the daemon.
```

This will start the MPS control daemon that will spawn a new MPS Server instance for any $UID starting an application and associate it with the GPU visible to the control daemon. Only one instance of the `nvidia-cuda-mps-control` daemon should be run per node. Note that `CUDA_VISIBLE_DEVICES` should not be set in the client process's environment.

Under MPS v3, each connecting UID is automatically served by a `uid_<uid>` server. You can also create named servers explicitly; refer to *MPS v3 Interface*.

1.1.4.3.1.2 Shutting Down MPS Control Daemon

To shut down the daemon, as root, run:

Legacy MPS v2

```
$ echo quit | nvidia-cuda-mps-control
```

MPS v3

```
$ nvidia-cuda-mps-control -q
```

1.1.4.3.2 On a Single-User System

When running as a single user, the control daemon must be launched with the same user ID as that of the client process.

1.1.4.3.2.1 Starting MPS Control Daemon

As $UID, run the commands:

Legacy MPS v2

```
$ export CUDA_VISIBLE_DEVICES=0                       # Select GPU 0.

$ export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps    # Select a location that
↪'s accessible to the given $UID.

$ export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log     # Select a location that
↪'s accessible to the given $UID.

$ nvidia-cuda-mps-control -d                          # Start the daemon.
```

MPS v3

```
$ export CUDA_VISIBLE_DEVICES=0                       # Select GPU 0.

$ export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps    # Select a location that
↪'s accessible to the given $UID.

$ export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log     # Select a location that
↪'s accessible to the given $UID.

$ nvidia-cuda-mps-control -d -p 3                     # Start the daemon.
```

This will start the MPS control daemon that will spawn a new MPS Server instance for that $UID starting an application and associate it with GPU visible to the control daemon.

1.1.4.3.2.2 Starting MPS client application

Set the following variables in the client process's environment. Note that CUDA_VISIBLE_DEVICES should not be set in the client's environment.

```
$ export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps   # Set to the same location
↪as the MPS control daemon.

$ export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log    # Set to the same location
↪as the MPS control daemon.
```

1.1.4.3.2.3 Shutting Down MPS

To shut down the daemon, as $UID, run:

Legacy MPS v2

```
$ echo quit | nvidia-cuda-mps-control
```

MPS v3

```
$ nvidia-cuda-mps-control -q
```

1.1.4.4 Best Practices for SM Partitioning

Creating a context is a costly operation in terms of time, memory, and the hardware resources.

If a context with execution affinity is created at kernel launch time, the user will observe a sudden increase in latency and memory footprint as a result of the context creation. To avoid paying the latency of context creation and the abrupt increase in memory usage at kernel launch time, it is recommended that users create a pool of contexts with different SM partitions upfront and select context with the suitable SM partition on kernel launch:

```
int device = 0;
cudaDeviceProp prop;
const Int CONTEXT_POOL_SIZE = 4;
CUcontext contextPool[CONTEXT_POOL_SIZE];
int smCounts[CONTEXT_POOL_SIZE];
cudaSetDevice(device);
cudaGetDeviceProperties(&prop, device);
smCounts[0] = 1; smCounts[1] = 2;
smCounts[3] = (prop. multiProcessorCount - 3) / 3;
smCounts[4] = (prop. multiProcessorCount - 3) / 3 * 2;
for (int i = 0; i < CONTEXT_POOL_SIZE; i++) {
   CUexecAffinityParam affinity;
   affinity.type = CU_EXEC_AFFINITY_TYPE_SM_COUNT;
   affinity.param.smCount.val = smCounts[i];
   cuCtxCreate_v3(&contextPool[i], affinity, 1, 0, deviceOrdinal);
}

for (int i = 0; i < CONTEXT_POOL_SIZE; i++) {
   std::thread([i]() {
```

(continues on next page)

(continued from previous page)

```
      int numSms = 0;
      int numBlocksPerSm = 0;
      int numThreads = 128;
      CUexecAffinityParam affinity;
      cuCtxSetCurrent(contextPool[i]);
      cuCtxGetExecAffinity(&affinity, CU_EXEC_AFFINITY_TYPE_SM_COUNT);
      numSms = affinity.param.smCount.val;
      cudaOccupancyMaxActiveBlocksPerMultiprocessor(
         &numBlocksPerSm, kernel, numThreads, 0);
      void *kernelArgs[] = { /* add kernel args */ };

      dim3 dimBlock(numThreads, 1, 1);
      dim3 dimGrid(numSms * numBlocksPerSm, 1, 1);
      cudaLaunchCooperativeKernel((void*)my_kernel, dimGrid, dimBlock,
↪kernelArgs);
   };
}
```

1.1.4.4.1 Using Static SM Partitioning

Static SM partitioning mode allows users to create exclusive SM partitions for MPS clients on NVIDIA Ampere architecture and newer GPUs, providing deterministic resource allocation and improved isolation.

Starting with Driver version r610, partial error isolation is supported when static SM partitioning is enabled. Because an SM is owned only by one partition, the driver can attribute SM error state to the faulting partition/client. Clients in different SM partitions are isolated from each other's SM-triggered faults. On fault detection, the driver terminates work for the faulting client and prevents additional work from being submitted. Terminated operations may report `CUDA_ERROR_LAUNCH_FAILED` or a more specific CUDA error. It should be noted that this is a partial isolation not a guarantee that every possible GPU or system-level failure is isolated per process.

Performance note

This mode trades some MPS flexibility for isolation. SMs reserved for a partition remain exclusive to the clients assigned to that partition; other clients cannot automatically borrow idle SMs from it. It is best suited for workloads with known resource needs, workloads that can tolerate strict resource limits, or deployments where fault containment is more important than opportunistic sharing.

The following example demonstrates the minimal workflow for configuring and using static partitions.

1.1.4.4.1.1 Basic Workflow

Legacy MPS v2

```
# 1. Start the MPS control daemon with static partitioning enabled
$ nvidia-cuda-mps-control -d -S

# 2. Create an SM partition with 7 chunks. The first partitioning command
# will perform a lightweight CUDA initialization.
$ echo "sm_partition add GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65 7" | nvidia-
```

(continues on next page)

(continued from previous page)

```
↪cuda-mps-control
GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

# 3. View the current partitioning configuration
$ echo "lspart" | nvidia-cuda-mps-control
GPU          Partition                                free    used    free
↪used  clients
                                                      chunk   chunk    SM    SM
GPU-74d43ed3      -                                   3       7        24    56
↪   -
GPU-74d43ed3  Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  -       7        -     56
↪   -

# 4. Assign the partition to a client application. The MPS server will start
# on client application connection.
$ export CUDA_MPS_SM_PARTITION=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ./nbody

# 5. After the application completes, remove the partition
$ echo "sm_partition rm GPU-74d43ed3 Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" |
↪nvidia-cuda-mps-control
```

MPS v3

```
# 1. Start the MPS control daemon
$ nvidia-cuda-mps-control -d -p 3

# 2. Create a server to own the partitions
$ nvidia-cuda-mps-control server create training

# 3. Create an SM partition with 4 chunks
$ nvidia-cuda-mps-control sm-partition create large --server=training --
↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65 --chunks=4
SM partition 'large' created on device GPU-74d43ed3-cdf7-e667-3644-
↪bf5b4f46ed65 (sm-count=34)
CUDA_MPS_SM_PARTITION=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

# 4. View the current partitioning configuration
$ nvidia-cuda-mps-control sm-partition list --server=training --all
SERVER    DEVICE                                   NAME   CHUNKS  SM-COUNT
↪INCLUDE-REMAINDER  STATUS   VISIBLE-UUID
training  GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65  large  4       34
↪false              created  Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

# 5. Assign the partition to a client application. The MPS server will start
# on client application connection.
$ export CUDA_MPS_SM_PARTITION=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ./nbody
```

(continues on next page)

(continued from previous page)

```
# 6. After the application completes, remove the partition
$ nvidia-cuda-mps-control sm-partition delete large --server=training --
↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65
```

1.1.4.4.1.2 Multiple Partitions Example

The following example demonstrates creating multiple partitions for different workloads:

Legacy MPS v2

```
# Start MPS control with static partitioning
$ nvidia-cuda-mps-control -d -S

# View the current partitioning configuration
$ echo "lspart" | nvidia-cuda-mps-control
GPU          Partition                                free    used    free
↪used  clients
                                                      chunk   chunk    SM    SM
GPU-74d43ed3      -                                   10       0       92    92
↪   -

# Create three partitions with different sizes
$ echo "sm_partition add GPU-74d43ed3 5" | nvidia-cuda-mps-control
GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

$ echo "sm_partition add GPU-74d43ed3 3" | nvidia-cuda-mps-control
GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/Cx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

$ echo "sm_partition add GPU-74d43ed3 2" | nvidia-cuda-mps-control
GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/Bx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

# Run different applications on different partitions
$ CUDA_MPS_SM_PARTITION=GPU-74d43ed3/Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ./
↪large_workload &
$ CUDA_MPS_SM_PARTITION=GPU-74d43ed3/Cx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ./
↪medium_workload &
$ CUDA_MPS_SM_PARTITION=GPU-74d43ed3/Bx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ./
↪small_workload &

# Check partition usage
$ echo "lspart" | nvidia-cuda-mps-control
GPU          Partition                                free    used    free
↪used  clients
                                                      chunk   chunk    SM    SM
GPU-74d43ed3      -                                   0       10       0     80
↪   -
GPU-74d43ed3  Dx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  -       5        -     40
↪   Yes
GPU-74d43ed3  Cx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  -       3        -     24
↪   Yes
```

(continues on next page)

(continued from previous page)

```
GPU-74d43ed3  Bx4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  -       2        -     28
 ↪  Yes
```

MPS v3

```
# Start the MPS control daemon and create a server
$ nvidia-cuda-mps-control -d -p 3
$ nvidia-cuda-mps-control server create training

# Create three partitions with different sizes. Each prints the value to
# assign to CUDA_MPS_SM_PARTITION.
$ nvidia-cuda-mps-control sm-partition create large --server=training --
 ↪device=0 --chunks=5
$ nvidia-cuda-mps-control sm-partition create medium --server=training --
 ↪device=0 --chunks=3
$ nvidia-cuda-mps-control sm-partition create small --server=training --
 ↪device=0 --chunks=2

# View the current partitioning configuration
$ nvidia-cuda-mps-control sm-partition list --server=training --all
SERVER    DEVICE        NAME    CHUNKS  SM-COUNT  INCLUDE-REMAINDER  STATUS
 ↪VISIBLE-UUID
training  GPU-74d43ed3  large   5       40        false              created
 ↪Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
training  GPU-74d43ed3  medium  3       24        false              created
 ↪Jn6AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
training  GPU-74d43ed3  small   2       28        false              created
 ↪Hm2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

# Run different applications on different partitions
$ CUDA_MPS_SM_PARTITION=GPU-74d43ed3/Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ./
 ↪large_workload &
$ CUDA_MPS_SM_PARTITION=GPU-74d43ed3/Jn6AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ./
 ↪medium_workload &
$ CUDA_MPS_SM_PARTITION=GPU-74d43ed3/Hm2AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ./
 ↪small_workload &
```

1.1.5. Troubleshooting Guide

1.1.5.1 MPS Server Not Accepting New Client Requests

This error indicates that the MPS (Multi-Process Service) server is currently unable to accept new client connections. This is typically a temporary condition caused by one of the following reasons.

Possible Causes and Recommended Actions Server Recovery or Automatic Restart

Cause: The MPS server is recovering from a previous error or automatically restarting due to a kernel-level application failure.

Action: Wait for the server to complete its recovery. If these errors occur repeatedly, investigate the client application for underlying issues.

Server Initialization in Progress

Cause: The server is performing initialization operations equivalent to `cuInit() --> cuCtxCreate()`. On systems with multiple GPUs and no persistence daemon, this can take a significant amount of time. Any client connections attempted during this period will receive this error.

Action: Allow the initialization to complete. To speed up future startups, consider enabling the NVIDIA Persistence Daemon.

Client Termination Cleanup

Cause: A `terminate_client` operation is in progress. During this time, the server restricts new connections to clean up resources from a specific client.

Action: Wait for the cleanup process to finish. This does not affect other connected clients.

User ID Mismatch Without Multi-User Mode

Cause: A client with a different user ID is attempting to connect while the server is not started with the `-multiuser-server` flag.

Action: Either start the MPS server with the `-multiuser-server` option or ensure that clients are using the same user ID. The default MPS behavior is to restart the server for each unique user ID.

1.1.6. Legacy MPS v2 Interface

The following describes the commands, environment variables and utilities available to configure and interact with the MPS execution environment.

1.1.6.1 nvidia-cuda-mps-control

The control daemon executable has the following options:

- `-d` – Start daemon in background process
- `-f` – Start daemon in foreground.
- `-v` – Print version of control daemon executable
- `-S` or `--static-partitioning` – Start daemon with static partitioning mode enabled.

The control daemon has an interactive shell, which can accept the following commands:

- `get_server_list` – prints out a list of all PIDs of server instances.
- `get_server_status <PID>` – this will print out the status of the server with the given <PID>.
- `start_server -uid <user id> [-mlopart]` – manually starts a new instance of nvidia-cuda-mps-server with the given user ID. If `mlopart` is specified, then clients will create MLOPart devices if supported.
- `get_client_list <PID>` – lists the PIDs of client applications connected to a server instance assigned to the given PID.
- `quit` – terminates the `nvidia-cuda-mps-control` daemon.
- `get_device_client_list [<PID>]` – lists the devices and PIDs of client applications that enumerated this device. It optionally takes the server instance PID.
- `set_default_active_thread_percentage <percentage>` – overrides the default active thread percentage for MPS servers. If there is already a server spawned, this command will only affect the next server. The set value is lost if a `quit` command is executed. The default is 100.
- `get_default_active_thread_percentage` – queries the current default available thread percentage.

- `set_active_thread_percentage <PID> <percentage>` – overrides the active thread percentage for the MPS server instance of the given PID. All clients created with that server afterwards will observe the new limit. Existing clients are not affected.
- `get_active_thread_percentage <PID>` – queries the current available thread percentage of the MPS server instance of the given PID.
- `set_default_device_pinned_mem_limit <dev> <value>` – sets the default device pinned memory limit for each MPS client. If there is already a server spawned, this command will only affect the next server. The set value is lost if a `quit` command is executed. The dev argument may be a device UUID string or an integer ordinal. The value must be in the form of an integer followed by a qualifier, either "G" or "M" that specifies the value in Gigabyte or Megabyte respectively. For example, to set a limit of 10 gigabytes for device 0, use the following command:

  ```
  set_default_device_pinned_mem_limit 0 10G
  ```

  By default, there is no memory limit set.

  Note that for this command, the dev argument is not validated against available devices in the MPS server. Therefore, it is possible to set two memory limits for the same device: one by device UUID and another by ordinal. When an MPS server is started, whichever limit was set last will take effect. A limit set with an invalid device UUID or ordinal will be ignored when starting the MPS server.
- `get_default_device_pinned_mem_limit <dev>` – queries the current default pinned memory limit for the device. The `dev` argument may be device UUID string or an integer ordinal.

  Note that this command does not translate between device UUIDs or ordinals and will return the limit that was set for each device identifier via the `set_default_device_pinned_mem_limit` command.
- `set_device_pinned_mem_limit <PID> <dev> <value>` - overrides the device pinned memory limit for MPS servers. This sets the device pinned memory limit for each client of MPS server instance of the given PID for the device dev. All clients created with that server afterwards will observe the new limit. Existing clients are not affected. The `dev` argument may be a device UUID string or an integer ordinal. For example, to set a limit of 900MB for the server with pid 1024 for device 0, use the following command:

  ```
  set_device_pinned_mem_limit 1024 0 900M
  ```
- `get_device_pinned_mem_limit <PID> <dev>` – queries the current device pinned memory limit of the MPS server instance of the given PID for the device `dev`. The `dev` argument may be a device UUID string or an integer ordinal.
- `terminate_client <server PID> <client PID>` – terminates all the outstanding GPU work of the MPS client process `<client PID>` running on the MPS server denoted by `<server PID>`. For example, to terminate the outstanding GPU work for an MPS client process with PID 1024 running on an MPS server with PID 123, use the following command:

  ```
  terminate_client 123 1024
  ```
- `ps [-p PID]` – reports a snapshot of the current client processes. It optionally takes the server instance PID. It displays the PID, the unique identifier assigned by the server, the partial UUID of the associated device, the PID of the connected server, the namespace PID, and the command line of the client.
- `set_default_client_priority [priority]` – sets the default client priority that will be used for new clients. The value is not applied to existing clients. Priority values should be considered as hints to the CUDA Driver, not guarantees. Allowed values are `0 [NORMAL]` and `1 [BELOW NORMAL]`. The set value is lost if a `quit` command is executed. The default is `0 [NORMAL]`.

- `get_default_client_priority` – queries the current priority value that will be used for new clients.
- `device_query [<server PID>] [--csv]` – Queries the devices that are available to MPS clients. If a server PID is specified, then the command will output the device information for that server and ignore other servers. If `csv` is specified, then the command will output the device information in a comma-separated format.
- `sm_partition add <device UUID> <number of chunks>` – creates an SM partition with the specified number of chunks on the given device. Upon successful creation, the full partition ID is displayed. This command accepts unique partial UUIDs of devices.
- `sm_partition rm <device UUID> <partition>` – removes the specified SM partition from the given device.
- `lspart` – displays the current SM partitioning configuration. The output includes the device UUID, partition IDs, free and used chunks, free and used SMs, and whether the partition is in use. The display uses unique partial UUIDs of devices.

1.1.6.2 Environment Variables

Refer to *Appendix: Environment Variables* for the full list of environment variables used to configure Legacy MPS v2 (and MPS v3).

1.1.7. MPS v3 Interface

MPS v3 is a new, opt-in control daemon interface that replaces the interactive shell of *Legacy MPS v2* with a scriptable `module verb` command syntax, named servers and namespaces, and file-based configuration. All existing Legacy MPS v2 functionality continues to work unchanged; MPS v3 is selected explicitly per the instructions below.

1.1.7.1 Enabling MPS v3

MPS v3 is selected by protocol version. Version 2 is Legacy MPS and remains the default; version 3 selects MPS v3.

- `-p 3` or `--protocol 3` on the `nvidia-cuda-mps-control` command line.
- The `CUDA_MPS_PROTOCOL_VERSION` environment variable, set to 3. If both the flag and the environment variable are set, the environment variable takes precedence.

```
nvidia-cuda-mps-control -d -p 3
```

```
export CUDA_MPS_PROTOCOL_VERSION=3
nvidia-cuda-mps-control -d
```

If an MPS control daemon is already running, any new `nvidia-cuda-mps-control` invocation automatically follows that daemon's protocol version, regardless of its own `-p` flag or environment variable.

1.1.7.2 Control Daemon Options

The control daemon accepts the following top-level options:

- `-h, --help` – Print this message.
- `-d, --daemon` – Setup the daemon in the background.

- `-f`, `--foreground` – Setup the daemon in the foreground.
- `-m`, `--multiuser` – Enable multiuser mode.
- `-a`, `--apply <file>` – Apply configuration from a TOML file. Refer to *MPS v3 Configuration File*.
- `-p`, `--protocol <2|3>` – Protocol 2 (legacy, default) or 3.
- `-q`, `--quit` – Quit and stop the control daemon and all running servers.
- `-v`, `--version` – Print the version.

1.1.7.3 CLI Command Structure

MPS v3 commands follow a common `module verb [arguments]` syntax:

```
nvidia-cuda-mps-control <module> <verb> [positional-args] [--key=value ...] [-
↪-flag ...]
```

- **Modules**: `server`, `namespace`, `client`, `device`, `sm-partition`, `feature`, `memacct`, `context-share`.
- Arguments may be positional (up to 3), or given as `--key=value` / `key=value`, or as valueless flags (`--force`, `--all`).
- Memory limit values use unit suffixes: `4G`, `512M`, `1024K`.
- Client priority values are `0` (normal) or `1` (below normal).
- Server and namespace names must be lowercase letters, digits, and underscores only, up to 64 characters.
- `<module> help` or `<module> <verb> help` prints detailed usage for that module or verb.

1.1.7.3.1 Server

A server is a named, independently managed MPS server process. Unlike Legacy MPS v2, where exactly one server can be active per UID, MPS v3 supports multiple concurrently running named servers.

```
nvidia-cuda-mps-control server create <name> [--uid=<n>] [--allowed-devices=
↪<devices>]
                                             [--locality-domains=<true|false>
↪]
nvidia-cuda-mps-control server delete <name> [--force]
nvidia-cuda-mps-control server list [<name>] [--format=<table|csv>[,noheader]]
nvidia-cuda-mps-control server set <name> [--uid=<n>] [--pinned-memory-limit=
↪<limit>]
                                          [--active-thread-percentage=<pct>]
                                          [--locality-domains=<true|false>]
                                          [--client-priority=<priority>]
nvidia-cuda-mps-control server get <name> <field>
```

- `create` – Creates a named server. Either `<name>` or `--uid=<n>` must be provided; if `--uid` is omitted, the server is owned by the connecting user. In multiuser mode, only `--uid=0` is allowed and `--uid` otherwise defaults to the connecting user.
  - `--allowed-devices=<devices>` – Restrict the server to specific GPUs (default: all). Comma-separated GPU IDs or ordinals, for example `GPU-abc123` or `0,1`.

- `--locality-domains=<true|false>` – Enable or disable locality domains (see *Locality Domains*).
- `--uid=<n>` – Owner UID. Required without `<name>`; optional with `<name>`.

- `delete` – Deletes a server by name. Without `--force`, refuses if active clients remain. With `--force`, terminates active clients first.
- `list` – Lists all servers, or a specific server if `<name>` is given. Output columns: `name`, `uid`, `pid`, `status`, `pipe-directory`, `allowed-devices`, `context-ids`.
- `set` – Sets server-level limits, applied to the server's `default` namespace. At least one option flag is required.
  - `--pinned-memory-limit=<limit>` – Pinned memory limit.
  - `--active-thread-percentage=<pct>` – Active thread percentage, 1-100.
  - `--locality-domains=<true|false>` – Enable or disable locality domains.
  - `--client-priority=<priority>` – Client priority.
- `get` – Gets a single, unformatted field value from a server. Fields: `name`, `uid`, `pid`, `status`, `pipe-directory`, `allowed-devices`.

```
$ nvidia-cuda-mps-control server create training --allowed-devices=GPU-abcd-
↪1234,GPU-efgh-5678 --uid=1000
$ nvidia-cuda-mps-control server set training --pinned-memory-limit=8G
$ nvidia-cuda-mps-control server list --format=csv,noheader
training,1000,48201,Ready,/run/nvidia-mps/training,GPU-abcd-1234,-
$ nvidia-cuda-mps-control server get training pipe-directory
/run/nvidia-mps/training
$ nvidia-cuda-mps-control server delete training --force
```

1.1.7.3.2 Namespace

A namespace subdivides a server's resources for a group of clients. Every server has an implicit `default` namespace. Refer to *MPS v3 Namespaces* for more about what namespaces are for.

```
nvidia-cuda-mps-control namespace create <name> --server=<s>
nvidia-cuda-mps-control namespace delete <name> --server=<s> [--force]
nvidia-cuda-mps-control namespace list [<name>] [--server=<s>]
                                       [--format=<table|csv>[,noheader]]
nvidia-cuda-mps-control namespace set <name> --server=<s> [--pinned-memory-
↪limit=<limit>]
                                             [--active-thread-percentage=
↪<pct>]
                                             [--client-priority=<priority>]
nvidia-cuda-mps-control namespace get <name> <server> <field>
```

- `create` – Creates a namespace under the given server.
- `delete` – Deletes a namespace. `--force` allows deletion even with active clients.
- `list` – Lists namespaces. Output columns: `name`, `server`, `server-status`, `pinned-memory-limit`, `active-thread-percentage`, `client-priority`, `pipe-directory`.
- `set` – Sets namespace limits. At least one option flag is required.

- `get` – Gets a single field value from a namespace. Fields: `name`, `server`, `server-status`, `pipe-directory`, `pinned-memory-limit`, `active-thread-percentage`, `client-priority`.

```
$ nvidia-cuda-mps-control namespace create high_priority --server=training
$ nvidia-cuda-mps-control namespace set high_priority --server=training --
↪pinned-memory-limit=4G
$ nvidia-cuda-mps-control namespace get high_priority training pinned-memory-
↪limit
4G
$ nvidia-cuda-mps-control namespace delete high_priority --server=training
```

1.1.7.3.3 Client

```
nvidia-cuda-mps-control client list [--server=<s>] [--all] [--format=
↪<table|csv>[,noheader]]
nvidia-cuda-mps-control client get <pid> <field>
nvidia-cuda-mps-control client terminate <pid>
```

- `list` – Lists active clients connected to servers, along with namespace and device information. `--all` includes clients from all servers; `--server=<s>` filters to one server.
- `get` – Gets a single field value for a client. Fields: `pid`, `server`, `device`, `pci-id`, `linux-ns`, `maws-ns`, `cmd`.
- `terminate` – Terminates a client by PID.

```
$ nvidia-cuda-mps-control client list --server=training --format=csv,noheader
51023,training,0,default,python train.py
51087,training,0,high_priority,python eval.py
$ nvidia-cuda-mps-control client terminate 51023
```

1.1.7.3.4 Device

1.1.7.3.4.1 list

```
nvidia-cuda-mps-control device list [--server=<s>] [--format=<table|csv>[,
↪noheader]]
```

Enumerates devices. `--server=<s>` scopes the view to one server's device enumeration (default: all servers). Output columns: `Ordinal`, `Server`, `PCI ID`, `Device UUID`, `Name`, `Total SMs`, `Total Chunks`, `Attributes`.

1.1.7.4 SM Partitions

Static SM partitioning in MPS v3 works the same way conceptually as *Legacy MPS v2 static SM partitioning*, using the `sm-partition` module instead of the interactive `sm_partition` command.

Note

The partition ID (the `CUDA_MPS_SM_PARTITION` value) is encoded differently in MPS v3 than in Legacy MPS v2, so a partition ID generated by one is not valid in the other. Always use the ID reported by the version of MPS you are running.

```
nvidia-cuda-mps-control sm-partition create <name> --server=<s> --device=
↪<device>
                                                --chunks=<n> [--include-
↪remainder=<true|false>]
nvidia-cuda-mps-control sm-partition delete <name> --server=<s> --device=
↪<device>
nvidia-cuda-mps-control sm-partition list [--server=<s>] [--device=<device>]
↪[--all]
                                              [--format=<table|csv>[,noheader]]
```

- `create` – Creates an SM partition. `--device` accepts either a device ordinal or a GPU/MIG UUID. `--chunks` must be a positive integer. `--include-remainder` defaults to `false`. If the target server is not yet running, the partition is recorded and applied the next time the server starts. If the server is already running, the partition is created immediately and its assigned ID is printed.
- `delete` – Deletes an SM partition.
- `list` – Lists SM partitions. Output columns: `server`, `device`, `name`, `chunks`, `sm-count`, `include-remainder`, `status`, `visible-uuid`. A partition's `status` is `configured` if it has been recorded but not yet applied to a running server, or `created` once it has been carved on the live GPU. Devices with unclaimed capacity also show a synthetic row with status `unallocated`.

```
$ nvidia-cuda-mps-control sm-partition create large --server=training --
↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65 --chunks=4
SM partition 'large' created on device GPU-74d43ed3-cdf7-e667-3644-
↪bf5b4f46ed65 (sm-count=34)
CUDA_MPS_SM_PARTITION=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65/
↪Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ nvidia-cuda-mps-control sm-partition list --server=training --all
SERVER    DEVICE                                   NAME     CHUNKS  SM-COUNT
↪INCLUDE-REMAINDER  STATUS   VISIBLE-UUID
training  GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65  large    4       34
↪ false             created  Kp8AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
training  GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65  -        1       8
↪ -                 unallocated  -
$ nvidia-cuda-mps-control sm-partition delete large --server=training --
↪device=GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65
SM partition 'large' deleted from device GPU-74d43ed3-cdf7-e667-3644-
↪bf5b4f46ed65
```

Static partitioning is rejected if the target device already has active clients, if locality domains are enabled on the server (the two features are mutually exclusive), or if the device does not have enough free chunks remaining.

1.1.7.5 Locality Domains

Locality domains are the MPS v3 name for the feature formerly called MLOPart in Legacy MPS v2. Refer to *Locality Domains* for a conceptual overview. Enable them per-server, either at creation time or on an existing server:

```
nvidia-cuda-mps-control server create <name> --locality-domains=<true|false>
nvidia-cuda-mps-control server set <name> --locality-domains=<true|false>
```

Locality domains are mutually exclusive with static SM partitions on the same server.

1.1.7.6 Features

Optional server-side behaviors are exposed as toggleable features through the `feature` module. Feature names are given positionally.

```
nvidia-cuda-mps-control feature enable <memacct|context-share>
nvidia-cuda-mps-control feature disable <memacct|context-share>
nvidia-cuda-mps-control feature describe [memacct|context-share] [--format=
↪<table|csv>[,noheader]]
```

`feature describe` prints, for one or all registered features: `FEATURE`, `STATE` (`enabled`/`disabled`, with `(default)` appended when it matches the built-in default), `MODULE`, and `COVERS`.

```
$ nvidia-cuda-mps-control feature describe
FEATURE          STATE                MODULE           COVERS
memacct          enabled (default)    memacct          -
context-share    enabled (default)    context-share    -
```

1.1.7.6.1 memacct

The `memacct` feature governs automatic memory accounting and audit logging.

```
nvidia-cuda-mps-control memacct set audit_log=<true|false>
nvidia-cuda-mps-control memacct describe [--format=<table|csv>[,noheader]]
```

`audit_log` is the only supported key. It is disabled by default.

1.1.7.6.2 context-share

The `context-share` feature controls MPS's underlying context-sharing behavior, including whether the default (UID-routed) socket auto-spawns backward-compatible servers.

```
nvidia-cuda-mps-control context-share set default_socket=<on|off>
nvidia-cuda-mps-control context-share describe [--format=<table|csv>[,
↪noheader]]
```

`default_socket` is enabled by default. Disabling `context-share` itself, or changing it while any server exists, is rejected – with the exception of `default_socket`, which may be changed at any time. Disabling `context-share` entirely lets CUDA applications run without MPS-style context sharing.

1.1.7.7 Configuration File

MPS v3 can load its full startup configuration from a TOML file with `-a`/`--apply`, covering servers, namespaces, devices, SM partitions, and features in a single file. Refer to *MPS v3 Configuration File* for the full schema and an example.

```
nvidia-cuda-mps-control -d -p 3 -a /path/to/config.toml
```

1.1.7.8 Directories and Environment Variables

MPS v3 clients and servers use the same *Appendix: Environment Variables* as Legacy MPS v2, including `CUDA_VISIBLE_DEVICES`, `CUDA_MPS_PIPE_DIRECTORY`, `CUDA_MPS_LOG_DIRECTORY`, and `CUDA_DEVICE_MAX_CONNECTIONS`. `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`, `CUDA_MPS_PINNED_DEVICE_MEM_LIMIT`, and `CUDA_MPS_CLIENT_PRIORITY` also work the same on

the client side, but interact with MPS v3 namespace-level settings differently than with Legacy MPS v2's daemon-wide settings; see the notes under each variable's entry there.

MPS v3 uses the same pipe directory and log directory defaults as Legacy MPS v2:

- `CUDA_MPS_PIPE_DIRECTORY` – If set, used directly. Otherwise defaults to `/run/nvidia-mps` when the daemon runs as root, or `/tmp/nvidia-mps` otherwise. Each server and namespace gets its own subdirectory under the pipe directory containing its `control` socket; each server's subdirectory also contains its `log` pipe.
- `CUDA_MPS_LOG_DIRECTORY` – Defaults to `/var/log/nvidia-mps`. The daemon log is `control.log`; each server logs to `<server-name>/server.log`.

At startup, the daemon prints the pipe directory clients should connect through:

```
To connect CUDA applications to this daemon, set CUDA_MPS_PIPE_DIRECTORY=/run/
↪nvidia-mps
```

1.1.8. MPS v3 Namespaces

A namespace subdivides a *server*'s resources – pinned memory limit, active thread percentage, and client priority – for a group of clients. Servers are typically shared across many clients with different resource needs; namespaces let those groups of clients be managed independently without requiring a separate server per group.

Every server has an implicit `default` namespace, which is used by clients that do not target a specific namespace. Refer to *MPS v3 Interface* for the `namespace create/delete/list/set/get` command reference, and to *MPS v3 Configuration File* for configuring namespaces from a TOML file.

1.1.8.1 Namespace Routing

A client selects its namespace by which socket, under the server's pipe directory, it connects to:

```
$CUDA_MPS_PIPE_DIRECTORY/
    control                          # main daemon control socket
    training/                        # server "training"
        control                      # server listener socket (routes to the
↪default namespace)
        default/
            control                  # default namespace socket
        high_priority/               # custom namespace
            control                  # namespace socket
        batch/
            control
```

- `$CUDA_MPS_PIPE_DIRECTORY/training/high_priority/control` – routes to the `high_priority` namespace on server `training`.
- `$CUDA_MPS_PIPE_DIRECTORY/training/control` – routes to server `training`'s `default` namespace.
- `$CUDA_MPS_PIPE_DIRECTORY/control` – UID-based auto-routing, for backward compatibility with Legacy MPS v2 clients.

A client sets `CUDA_MPS_PIPE_DIRECTORY` to the namespace's directory to connect through that namespace's socket, in the same way Legacy MPS v2 clients set it to a server's pipe directory.

1.1.9. MPS v3 Configuration File

MPS v3 can load its full startup configuration from a TOML file with `-a`/`--apply`:

```
nvidia-cuda-mps-control -d -p 3 -a /path/to/config.toml
```

Configuration is applied once at startup; there is currently no persistence of subsequent CLI changes back to disk, so servers, namespaces, and partitions created after startup only exist for the lifetime of the daemon unless they are also added to the configuration file.

```
schema_version = "1.0"

[servers.training]
allowed_devices          = "GPU-abcd-1234,GPU-efgh-5678"
pinned_memory_limit      = "10G"
active_thread_percentage = "50"
client_priority          = "0"
uid                      = 1000
locality_domains         = false

[servers.training.namespaces.high_priority]
pinned_memory_limit      = "5G"
active_thread_percentage = "25"

[[servers.training.devices]]
uuid = "GPU-74d43ed3-cdf7-e667-3644-bf5b4f46ed65"

  [[servers.training.devices.sm_partitions]]
  name              = "large"
  chunks            = 4
  include_remainder = false

[features.memacct]
enabled   = true
audit_log = true

[features.context-share]
enabled        = true
default_socket = "on"
```

- `[servers.<name>]` accepts `allowed_devices`, `pinned_memory_limit`, `active_thread_percentage`, `client_priority`, `uid`, and `locality_domains` (boolean only). Any other key is a configuration error.
- `[servers.<name>.namespaces.<name>]` accepts only `pinned_memory_limit`, `active_thread_percentage`, and `client_priority`. A namespace named `default` refers to the server's built-in default namespace rather than creating a new one. Refer to *MPS v3 Namespaces* for the equivalent CLI commands.
- `[[servers.<name>.devices]]` requires `uuid` and optionally nests `[[servers.<name>.devices.sm_partitions]]` entries, each requiring `name` and `chunks`, with optional `include_remainder` (default `false`).
- `[features.<name>]` tables are read per-feature; see the Features section in *MPS v3 Interface* for the keys each feature accepts.

- TOML keys use underscores; the equivalent CLI flags use dashes (for example, `pinned_memory_limit` in the file corresponds to `--pinned-memory-limit` on the command line).

1.1.10. MPS v3 Memory Partitioning

MPS v3 memory partitioning fractionalizes a GPU's device memory across cgroups and containers, so that a single tenant cannot monopolize the GPU and schedulers such as Kubernetes can pack workloads by memory. Unlike the older userspace memory limits, this feature accounts for *all* device memory and integrates with the Linux kernel cgroup controllers.

This feature requires Linux with cgroup **v2** mounted at `/sys/fs/cgroup`, CUDA 13.4 or newer, and a non-MIG device. MIG is explicitly unsupported, with memory reporting left unaltered for MIG handles. Managed and UVM memory have additional limitations; see *Known Limitations*.

1.1.10.1 Soft and Hard Limits

Memory partitioning is expressed with two thresholds that define three zones:

```
-- HARD LIMIT --------
FAILURE  ZONE (allocations return OUT_OF_MEMORY)

-- SOFT LIMIT --------
PRESSURE ZONE (allocations still succeed; you are "borrowing" from others'
↪share.
               Under real contention MPS may evict/kill you.)

-- (0) ---------------
SAFE ZONE:    (guaranteed available)
```

Under the dmem controller:

- **Hard limit** maps to `dmem.max`. It is enforced at allocation time; an allocation that would exceed it fails with an out-of-memory error.
- **Soft limit** maps to `dmem.min`. It is a *target*, not instant failure. Crossing it makes a client an eviction *candidate*; the client is only actually evicted under real memory pressure.

1.1.10.2 Backends

Memory partitioning has two backends, chosen automatically when the driver loads:

- **dmem**: used when the kernel (6.14 or newer) has the DMEM cgroup controller present. The kernel `dmem` controller enforces the hard and soft limits natively.
- **misc** (the fallback): used on kernels without the DMEM controller. The driver does its own per-cgroup accounting internally.

Note

The dmem backend organizes reservations as a hierarchy per root cgroup. For deployments that share a single root cgroup across independent workloads follow one of the deployment models in *Hierarchy and Containers* so that each workload is partitioned independently. The misc backend, which has a flat one-limit-per-container model with no hierarchy, is a simple alternative for those cases.

Advanced: forcing a backend

The backend can be overridden with the `RmMemacctMode` registry key, set through the driver module parameter:

```
# /etc/modprobe.d/nvidia.conf, then reload nvidia.ko
options nvidia NVreg_RegistryDwords="RmMemacctMode=1"
```

Values: `0` off, `1` force misc, `2` force dmem, `3` auto-detected (default). The override can only downgrade. The misc backend can also be selected by disabling the kernel controller with the `cgroup_disable=dmem` kernel command-line parameter.

1.1.10.3 Setting Limits

Limits can be set through `nvidia-smi` or NVML, both set the same underlying state. Always set limits through `nvidia-smi` or NVML rather than writing the `dmem.*` files directly. These tools resolve the device key, handle the backend, and maintain the ancestor propagation described in *Hierarchy and Containers*, which manual writes do not. Limits are specified in **MiB** for `nvidia-smi` and in bytes for NVML. Both limits accept a "max" sentinel meaning "device capacity."

1.1.10.3.1 Using nvidia-smi

Recommended for manual and operational use.

```
# Set soft = 4 GiB, hard = 8 GiB on a cgroup, for GPU 0
$ sudo nvidia-smi memory-limits --set \
     --namespace /sys/fs/cgroup/mygroup \
     --soft-limit 4000 --hard-limit 8000 -i 0

# Read back
$ nvidia-smi memory-limits --get -n /sys/fs/cgroup/mygroup -i 0
namespace:     /sys/fs/cgroup/mygroup
soft limit:    4096 MiB
hard limit:    8192 MiB
current used:  512 MiB
```

1.1.10.3.2 Using the NVML API

Recommended for frameworks and orchestrators. Setting limits is privileged; reading them is not.

```
nvmlDeviceSetMemoryLimits_v1_t lim = {
    .nameSpace = "/sys/fs/cgroup/mygroup",   // full path to the cgroup dir
    .softLimit = 4ULL<<30,                    // bytes (or bSetToMax)
    .hardLimit = 8ULL<<30,
};
nvmlDeviceSetMemoryLimits_v1(device, &lim);          // privileged

nvmlDeviceGetMemoryLimits_v1_t got = { .nameSpace = "/sys/fs/cgroup/mygroup" }
↪;
nvmlDeviceGetMemoryLimits_v1(device, &got);          // non-privileged
// got.softLimit / hardLimit / currentLimit (+ bIsMax flags)
```

NVML opens the cgroup directory and applies the limit through the driver. It defaults to the

sysfs/dmem path and falls back to the driver's internal control when dmem is not present. MIG handles are rejected.

1.1.10.4 How Limits Are Reported

Once memory partitioning is active for a container's cgroup, memory queries reflect the **limit**, not the physical device.

- `cuMemGetInfo` (v1/v2): reports total and free memory capped to the hard limit and the remaining headroom under it.
- NVML `nvmlDeviceGetMemoryInfo_v2`: reports used, total, and free memory adjusted to account for the soft-limit headroom guarantee and reserved physical memory.
- MIG: reporting is **not** altered.

1.1.10.5 Enforcement

- **Hard limit:** an allocation that would exceed `dmem.max` fails. On dmem the kernel rejects it; on misc the driver checks against the group's available budget and rejects it. CUDA maps this to `CUDA_ERROR_OUT_OF_MEMORY`.
- **Soft limit:** crossing it fires a subdevice soft-limit-exceeded notifier carrying the offending PID. This only *marks it as a candidate*; it is not a kill. Actual eviction is reactive and handled by MPS; see *MPS Pressure Handling and Eviction*.

1.1.10.6 Hierarchy and Containers

1.1.10.6.1 Setting on a leaf versus a parent

- You can only set a limit on a cgroup that has the dmem controller enabled (has `dmem.*` files). Setting on a cgroup with no dmem files returns "not found."
- A cgroup with no dmem files (its parent did not add `+dmem` to `subtree_control`) inherits the nearest dmem-enabled ancestor's limit on *reads*. Set the limit on the enabled ancestor; fileless descendants read it back.
- **Docker:** setting the limit on the Docker root or pod slice might *not* reach the container, since a **new leaf** cgroup is created for the container with `+dmem` enabled (so the leaf has its own `dmem.max` of `max`). It reports the full GPU and never inherits the ancestor. The fix is to discover the container's real cgroup *after* creation (for example, `docker inspect` the container PID, then read its actual `/sys/fs/cgroup/...` path) and apply the limit on that leaf.

1.1.10.6.2 Ancestor reservation propagation (dmem)

Because the kernel computes effective protection hierarchically, a leaf's `dmem.min` is only honored if every ancestor reserves at least as much. The driver's set-limit path therefore propagates: it walks parents and adds the delta so that each `ancestor.dmem.min` equals the sum of its subtree's leaf soft limits.

- Setting a leaf's soft **raises** its ancestors' `dmem.min`; resetting it to 0 **lowers** them.
- The walk stops at the first ancestor without `dmem.min` (the cgroup root, or a gap where `+dmem` is not enabled). A gap breaks propagation and kernel protection above it so keep the dmem-enabled chain contiguous from the leaf to the reservation point.
- Only the driver path propagates. This is why limits must be set through `nvidia-smi` or NVML; writing the `dmem.*` files directly bypasses propagation and causes drift.

The ancestor propagation only *decrements* ancestors when the leaf soft is lowered or zeroed through NVML or `nvidia-smi`. If the leaf cgroup is **deleted directly** (the normal container exit/crash lifecycle), the driver never sees it, so the down-delta does not run and each ancestor keeps its leaf's contribution.

- Before deleting a leaf, zero its soft first (`nvidia-smi memory-limits --set ... --soft-limit 0 ...` or NVML with `bSetToMax`/0) so the ancestors are decremented.
- Or avoid relying on propagation at all; see *Deployment models*.

1.1.10.6.3 Deployment models

To workaround the shared-root aggregation with dmem, choose one of:

1. **Separate root per workload.** Each hierarchy is independent; propagation only walks up to its own root and cannot interfere with another.
2. **Force the root reservation to 0** after every set-limits call. A `dmem.min` of 0 at the root overrides the event system to use manual filesystem checks instead.
3. **Use the misc fallback** (most reliable today). misc has no hierarchy.

1.1.10.7 MPS Pressure Handling and Eviction

MPS is the component that reacts to memory pressure. The MPS server subscribes per-GPU to the soft-limit notifier at startup.

Handling is currently **reactive, not proactive**. The soft limit means "borrowing," so crossing it never kills a client outright. The flow is:

1. A client crosses its soft limit, and the server marks that PID a **candidate**.
2. Another client's allocation hits the **hard** wall and would OOM. Because that client is still within its own soft limit, the CUDA driver requests eviction from the server and blocks on the reply.
3. The server picks an over-soft victim, force-terminates it, and replies with a retry.
4. The requester retries the allocation with bounded back-off; on success it returns `CUDA_SUCCESS`, otherwise `CUDA_ERROR_OUT_OF_MEMORY`.
5. The evicted victim's next CUDA call returns `CUDA_ERROR_MPS_CLIENT_TERMINATED`.

The server will not evict the requester itself.

1.1.10.8 dmem versus misc

Aspect	dmem	misc
Limit store	kernel `dmem.*` sysfs	driver-internal map
Hierarchy	kernel effective protection plus driver `dmem.min` propagation	nearest limited ancestor; limits do not nest (a parent cap does not bound limited children)
Over-soft detection	MPS compares `dmem.current` against each cgroup's `dmem.min`	MPS is notified by the driver on charge and uncharge

1.1.10.9 Known Limitations

- **MIG:** unsupported; memory reporting is not altered for MIG handles.
- **UVM / managed memory:** `cudaMallocManaged` is only tracked by the memory controller in system memory.
- **Shared root cgroups (dmem):** the kernel aggregates usage up to the root. When a root cgroup is shared across independent workloads, use a deployment model from *Deployment models* to partition each workload independently, and zero a leaf's soft before deleting its cgroup so the ancestor reservation is released.
- **Multi-GPU:** `dmem.*` files are per-GPU; always use the exact device key. Resolve the key from the target device's PCI BDF.
- **Contiguous dmem chain:** a gap where `+dmem` is not enabled at some level breaks propagation and kernel protection above it.

1.1.10.10 Diagnostics and Troubleshooting

- **Which backend?** Check whether the target cgroup has a `dmem.max` file. If it does, you are on dmem; if not, you are on misc (or the feature is off):

```
if [ -e /sys/fs/cgroup/<cgroup>/dmem.max ]; then
    echo "dmem backend"
else
    echo "misc backend (or memory partitioning is off)"
fi
```

- **Walk the hierarchy (dmem):** inspect each level's `dmem.current`, `dmem.min`, and `dmem.max` to confirm the reservations propagated as expected up to the intended ancestor.
- **Limit not reflected in a container:** check whether the container's *own* cgroup has a `dmem.max` value (`max` means unlimited, which shadows the ancestor). Set the limit on the container's real leaf cgroup.
- **Eviction never fires for a within-soft requester:** verify the requester can reach the MPS daemon's control socket (pipe directory), and that its cgroup soft limit is actually being read.
- **MPS eviction events:** the MPS daemon logs per-device eviction events – eviction requests, evictions performed, and no-victim counts – useful for confirming eviction behavior under pressure.

1.1.10.11 Quick Reference

Set the soft and hard limits on a cgroup for a GPU:

```
sudo nvidia-smi memory-limits --set -n <cgroup> --soft-limit <MiB> --hard-
↪limit <MiB> -i <idx>
```

Read the limits back:

```
nvidia-smi memory-limits --get -n <cgroup> -i <idx>
```

1.1.11. Appendix: Environment Variables

The following environment variables apply identically to *Legacy MPS v2* and *MPS v3* clients and servers, except where a MPS v3-specific note is called out below.

1.1.11.1 CUDA_VISIBLE_DEVICES

CUDA_VISIBLE_DEVICES is used to specify which GPU's should be visible to a CUDA application. Only the devices whose index or UUID is present in the sequence are visible to CUDA applications and they are enumerated in the order of the sequence.

When CUDA_VISIBLE_DEVICES is set before launching the control daemon, the devices will be remapped by the MPS server. This means that if your system has devices 0, 1 and 2, and if CUDA_VISIBLE_DEVICES is set to 0,2, then when a client connects to the server it will see the remapped devices – device 0 and a device 1. Therefore, keeping CUDA_VISIBLE_DEVICES set to 0,2 when launching the client would lead to an error.

To avoid this ambiguity, we recommend using UUIDs instead of indices. These can be viewed by launching nvidia-smi -q. When launching the server, or the application, you can set CUDA_VISIBLE_DEVICES to UUID_1,UUID_2, where UUID_1 and UUID_2 are the GPU UUIDs. It will also work when you specify the first few characters of the UUID (including GPU-) rather than the full UUID.

The MPS server will fail to start if incompatible devices are visible after the application of CUDA_VISIBLE_DEVICES.

1.1.11.2 CUDA_MPS_PIPE_DIRECTORY

The MPS control daemon, the MPS server, and the associated MPS clients communicate with each other via named pipes and UNIX domain sockets. The default directory for these pipes and sockets is /tmp/nvidia-mps. The environment variable, CUDA_MPS_PIPE_DIRECTORY, can be used to override the location of these pipes and sockets. The value of this environment variable should be consistent across all MPS clients sharing the same MPS server, and the MPS control daemon.

The recommended location for the directory containing these named pipes and domain sockets is local folders such as /tmp. If the specified location exists in a shared, multi-node filesystem, the path must be unique for each node to prevent multiple MPS servers or MPS control daemons from using the same pipes and sockets. When provisioning MPS on a per-user basis, the directory should be set to a location such that different users will not end up using the same directory.

On Tegra platforms, there is no default directory setting for pipes and sockets. Users must set this environment variable such that only intended users have access to this location.

1.1.11.3 CUDA_MPS_LOG_DIRECTORY

The MPS control daemon maintains a control.log file which contains the status of its MPS servers, user commands issued and their result, and startup and shutdown notices for the daemon. The MPS server maintains a server.log file containing its startup and shutdown information and the status of its clients.

By default these log files are stored in the directory /var/log/nvidia-mps. The CUDA_MPS_LOG_DIRECTORY environment variable can be used to override the default value. This environment variable should be set in the MPS control daemon's environment and is automatically inherited by any MPS servers launched by that control daemon.

On Tegra platforms, there is no default directory setting for storing the log files. MPS will remain operational without the user setting this environment variable; however, in such instances, MPS logs will not be available. If logs are required to be captured, then the user must set this environment variable such that only intended users have access to this location.

1.1.11.4 CUDA_DEVICE_MAX_CONNECTIONS

When encountered in the MPS client's environment, `CUDA_DEVICE_MAX_CONNECTIONS` sets the preferred number of compute and copy engine concurrent connections (work queues) from the host to the device for that client. The number actually allocated by the driver may differ from what is requested based on hardware resource limitations or other considerations. Under MPS, each server's clients share one pool of connections, whereas without MPS each CUDA context would be allocated its own separate connection pool. Volta MPS clients exclusively owns the connections set aside for the client in the shared pool, so setting this environment variable under Volta MPS may reduce the number of available clients. The default value is 2 for Volta MPS clients.

1.1.11.5 CUDA_MPS_ACTIVE_THREAD_PERCENTAGE

On Volta GPUs, this environment variable sets the portion of the available threads that can be used by the client contexts. The limit can be configured at different levels:

- Setting this environment variable in an MPS control's environment will configure the default active thread percentage when the MPS control daemon starts. All the MPS servers spawned by the MPS control daemon will observe this limit. Once the MPS control daemon has started, changing this environment variable cannot affect the MPS servers.
- Setting this environment variable in an MPS client's environment will configure the active thread percentage when the client process starts. The new limit will only further constrain the limit set by the control daemon (via `set_default_active_thread_percentage` or `set_active_thread_percentage` control daemon commands or this environment variable at the MPS control daemon level). If the control daemon has a lower setting, the control daemon setting will be obeyed by the client process instead.

  All the client CUDA contexts created within the client process will observe the new limit. Once the client process has started, changing the value of this environment variable cannot affect the client CUDA contexts.
- By default, configuring the active thread percentage at the client CUDA context level is disabled. User must explicitly opt-in via environment variable `CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PARTITIONING`. Refer to *CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PARTITIONING* for more details.

  Setting this environment variable within a client process will configure the active thread percentage when creating a new client CUDA context. The new limit will only further constraint the limit set at the control daemon level and the client process level. If the control daemon or the client process has a lower setting, the lower setting will be obeyed by the client CUDA context instead. All the client CUDA contexts created afterwards will observe the new limit. Existing client CUDA contexts are not affected.

Note

Under MPS v3, if the client's namespace has an active thread percentage set (see *MPS v3 Interface*), that value overrides the client-process-level setting from this environment variable, rather than only further constraining it as the control-daemon-level setting does under Legacy MPS v2.

1.1.11.6 CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PARTITIONING

By default, users can only partition the available threads uniformly. An explicit opt-in via this environment variable is required to enable non-uniform partitioning capability. To enable non-uniform partitioning capability, this environment variable must be set before the client process starts.

When non-uniform partitioning capability is enabled in an MPS client's environment, client CUDA contexts can have different active thread percentages within the same client process via setting `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` before context creations. The device attribute `cudaDevAttrMultiProcessorCount` will reflect the active thread percentage and return the portion of available SMs that can be used by the client CUDA context current to the calling thread.

1.1.11.7 CUDA_MPS_PINNED_DEVICE_MEM_LIMIT

The pinned memory limit control limits the amount of GPU memory that is allocatable by CUDA APIs by the client process. On Volta GPUs, this environment variable sets a limit on pinned device memory that can be allocated by the client contexts. Setting this environment variable in an MPS client's environment will set the device's pinned memory limit when the client process starts. The new limit will only further constrain the limit set by the control daemon (via `set_default_device_pinned_mem_limit` or `set_device_pinned_mem_limit control` daemon commands or this environment variable at the MPS control daemon level). If the control daemon has a lower value, the control daemon setting will be obeyed by the client process instead. This environment variable will have the same semantics as `CUDA_VISIBLE_DEVICES` i.e. the value string can contain comma-separated device ordinals and/or device UUIDs with per device memory limit separated by an equals. Example usage:

```
$ export CUDA_MPS_PINNED_DEVICE_MEM_LIMIT=''0=1G,1=512MB''
```

The following example highlights the hierarchy and usage of the MPS memory limiting functionality.

```
# Set the default device pinned mem limit to 3G for device 0. The default limit
 ↪constrains the memory allocation limit of all the MPS clients of future MPS
 ↪servers to 3G on device 0.

$ nvidia-cuda-mps-control set_default_device_pinned_mem_limit 0 3G

# Start daemon in background process

$ nvidia-cuda-mps-control -d

# Set device pinned mem limit to 2G for device 0 for the server instance of the
# given PID. All the MPS clients on this server will observe this new limit of
 ↪2G
# instead of the default limit of 3G when allocating pinned device memory on
 ↪device 0.
# Note -- users are allowed to specify a server limit (via set_device_pinned_
 ↪mem_limit)
# greater than the default limit previously set by set_default_device_pinned_
 ↪mem_limit.

$ nvidia-cuda-mps-control set_device_pinned_mem_limit <pid> 0 2G

# Further constrain the device pinned mem limit for a particular MPS client to
 ↪1G for
# device 0. This ensures the maximum amount of memory allocated by this client
 ↪is capped
# at 1G.
# Note - setting this environment variable to a value greater than value
 ↪observed by the
# server for its clients (through set_default_device_pinned_mem_limit/ set_
```

(continues on next page)

(continued from previous page)

```
↪device_pinned_mem_limit)
# will not set the limit to the higher value and thus will be ineffective and
↪the eventual
# limit observed by the client will be that observed by the server.

$ export CUDA_MPS_DEVICE_MEM_LIMIT="0=1G"
```

Note

Under MPS v3, the enforced limit is sourced from the client's namespace (`--pinned-memory-limit=`, see *MPS v3 Interface*) instead of a single daemon-wide default and per-server override. This environment variable still works identically on the client side, further constraining whatever limit the namespace has set.

1.1.11.8 CUDA_MPS_CLIENT_PRIORITY

The client priority level variable controls the initial default server value for the MPS Control Daemon if used to launch that, or the client priority level value for a given client if used in a client launch. The following examples demonstrate both usages.

```
# Set the default client priority level for new servers and clients to Below
↪Normal
$ export CUDA_MPS_CLIENT_PRIORITY=1

$ nvidia-cuda-mps-control -d

# Set the client priority level for a single program to Normal without changing
↪the priority level for future clients
$ CUDA_MPS_CLIENT_PRIORITY=0 <program>
```

Note

CUDA priority levels are not guarantees of execution order – they are only a performance hint to the CUDA driver.

Note

Under MPS v3, if a client leaves this environment variable unset, its namespace's client priority (`--client-priority=`, see *MPS v3 Interface*) is used as the default instead of the daemon-wide default. If the client does set this environment variable, it takes effect the same way as under Legacy MPS v2.

1.1.12. Appendix: Logging

The MPS control daemon maintains a `control.log` file which contains the status of its MPS servers, user commands issued and their result, and startup and shutdown notices for the daemon. Each MPS

server maintains a `server.log` file containing its startup and shutdown information and the status of its clients.

1.1.12.1 Log File Locations

By default, these log files are stored in `/var/log/nvidia-mps`. The `CUDA_MPS_LOG_DIRECTORY` environment variable can be used to override the location; it should be set in the MPS control daemon's environment and is automatically inherited by any MPS servers it launches. Refer to *Appendix: Environment Variables* for details.

Legacy MPS v2

```
$CUDA_MPS_LOG_DIRECTORY/control.log

$CUDA_MPS_LOG_DIRECTORY/server.log
```

MPS v3

```
$CUDA_MPS_LOG_DIRECTORY/control.log

$CUDA_MPS_LOG_DIRECTORY/<server-name>/server.log
```

These log files are typically only visible to users with administrative privileges unless `CUDA_MPS_LOG_DIRECTORY` was set up elsewhere.

On Tegra platforms, there is no default directory setting for storing the log files. MPS will remain operational without the user setting `CUDA_MPS_LOG_DIRECTORY`; however, in such instances, MPS logs will not be available.

1.1.13. Notices

product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform 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 document. NVIDIA accepts no 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 document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

1.1.13.1 Trademarks

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

Copyright