MULTI-PROCESS SERVICE

vR331 | March 2014

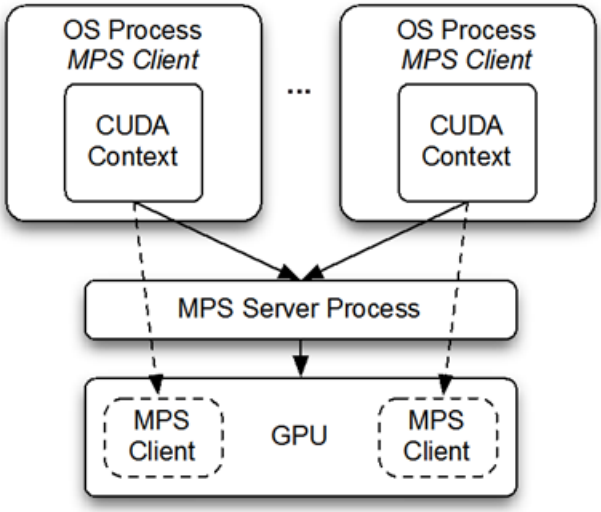
Multi-Process Service

INTRODUCTION

1.1. AT A GLANCE

1.1.1. MPS

The Multi-Process Service (MPS) is an alternative, binary-compatible implementation of the CUDA Application Programming Interface (API). The MPS runtime architecture is designed to transparently enable co-operative multi-process CUDA applications, typically MPI jobs, to utilize Hyper-Q capabilities on the latest NVIDIA (Kepler-based) Tesla and Quadro GPUs. Hyper-Q allows CUDA kernels to be processed concurrently on the same GPU; this can benefit performance when the GPU compute capacity is underutilized by a single application process.



1.1.2. Intended Audience

This document is a comprehensive guide to MPS capabilities and usage. It is intended to be read by application developers & users who will be running GPU calculations and intend to achieve the greatest level of execution performance. It is also intended to

be read by system administrators who will be enabling the MPS capability in a user-friendly way, typically on multi-node clusters.

1.1.3. Organization of This Document

The order of the presentation is as follows:

Introduction and Concepts – describes why MPS is needed and how it enables Hyper-Q for multi-process applications.

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.

Appendices – Reference information for the tools and interfaces used by the MPS system and guidance for common use-cases.

1.2. Prerequisites

Portions of this document assume that you are already familiar with:

the structure of CUDA applications and how they utilize the GPU via the CUDA Runtime and CUDA Driver software libraries.

concepts of modern operating systems, such as how processes and threads are scheduled and how inter-process communication typically works

the Linux command-line shell environment

configuring and running MPI programs via a command-line interface

1.3. Concepts

1.3.1. Why MPS is needed

To balance workloads between CPU and GPU tasks, MPI processes are often allocated individual CPU cores in a multi-core CPU machine to provide CPU-core parallelization of potential Amdahl bottlenecks. As a result, the amount of work each individual MPI process is assigned may underutilize the GPU when the MPI process is accelerated using CUDA kernels. While each MPI process may end up running faster, the GPU is being used inefficiently. The Multi-Process Service takes advantage of the inter-MPI rank parallelism, increasing the overall GPU utilization.

1.3.2. What MPS is

MPS is a binary-compatible client-server runtime implementation of the CUDA API which consists of several components.

Control Daemon Process – The control daemon is responsible for starting and stopping the server, as well as coordinating connections between clients and servers.

Client Runtime – The MPS client runtime is built into the CUDA Driver library and may be used transparently by any CUDA application.
Server Process – The server is the clients' shared connection to the GPU and provides concurrency between clients.

1.4. See Also

Manpage for nvidia-cuda-mps-control (1)
Manpage for nvidia-smi (1)
Blog “Unleash Legacy MPI Codes With Kepler’s Hyper-Q” by Peter Messmer
(http://blogs.nvidia.com/2012/08/unleash-legacy-mpi-codes-with-keplers-hyper-q)

WHEN TO USE MPS

2.1. The Benefits of MPS

2.1.1. GPU utilization

A single process may not utilize all the compute and memory-bandwidth capacity available on the GPU. MPS allows kernel and memcopy operations from different processes to overlap on the GPU, achieving higher utilization and shorter running times.

2.1.2. Reduced on-GPU context storage

Without MPS each CUDA processes using a GPU allocates separate storage and scheduling resources on the GPU. In contrast, the MPS server allocates one copy of GPU storage and scheduling resources shared by all of its clients.

2.1.3. Reduced GPU context switching

Without MPS, when processes share the GPU their scheduling resources must be swapped on and off the GPU. The MPS server shares one set of scheduling resources between all of its clients, eliminating the overhead of swapping when the GPU is scheduling between those clients.

2.2. 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-block, 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.

2.3. Considerations

2.3.1. System Considerations

2.3.1.1. Limitations

MPS is only supported on the Linux operating system. The MPS server will fail to start when launched on an operating system other than Linux.

MPS requires a Tesla or Quadro GPU with compute capability version 3.5 or higher. The MPS server will fail to start if the GPU visible after applying CUDA_VISIBLE_DEVICES is not of compute capability 3.5 or higher.

The Unified Virtual Addressing (UVA) feature of CUDA must be available, which is the default for any 64-bit CUDA program running on a GPU with compute capability version 2.0 or higher. If UVA is unavailable, the MPS server will fail to start.

Only single GPU in a multi-GPU system may be managed by an MPS server. The MPS server will fail to start if more than one GPU is visible after applying the CUDA_VISIBLE_DEVICES filter.

The amount of page-locked host memory that can be allocated by MPS clients is limited by the size of the tmpfs filesystem (/dev/shm).

Exclusive-mode restrictions are applied to the MPS server, not MPS clients.

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 (e.g. nvidia-smi, NVML API)

2.3.1.2. GPU Compute Modes

Fermi- and Kepler- based GPUs support four Compute Modes via settings accessible in nvidia-smi.

PROHIBITED – the GPU is not available for compute applications.

EXCLUSIVE_THREAD – the GPU is assigned to only one process at a time, and will only perform work from one thread of that process. The CUDA stream construct can be used to run multiple CUDA kernels (and memcopy operations) concurrently.

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. 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.

Using EXCLUSIVE_THREAD mode with MPS is unsupported and will cause undefined behavior.

2.3.2. 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 (i.e. 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 client applications have access to only a single CUDA device. MPS clients only see the single device exposed by the MPS server, and it is always device 0.

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.

Stream callbacks are not supported. Calling any stream callback APIs will return an error.

The amount of page-locked host memory that MPS client applications can allocate is limited by the size of the tmpfs filesystem (/dev/shm). Attempting to allocate more page-locked memory than the allowed size using any of relevant CUDA APIs will fail.

2.3.3. Memory Protection and Error Containment

MPS is only recommended for running cooperative processes effectively acting as a single application, such as multiple ranks of the same MPI job, such that the severity of the following memory protection and error containment limitations is acceptable.

2.3.3.1. Memory Protection

MPS client processes allocate memory from different partitions of the same GPU virtual address space. As a result:

An out-of-range write in a CUDA Kernel can modify the CUDA-accessible memory state of another process, and will not trigger an error.

An out-of-range read in a CUDA Kernel can access CUDA-accessible memory modified by another process, and will not trigger an error, leading to undefined behavior.

This behavior is constrained to memory accesses from pointers within CUDA Kernels. Any CUDA API restricts MPS clients from accessing any resources outside of that MPS Client's memory partition. For example, it is not possible to overwrite another MPS client's memory using the cudaMemcpy() API.

2.3.3.2. Error Containment

MPS client processes share on-GPU scheduling and error reporting resources. As a result:

A GPU exception generated by any client will be reported to all clients, without indicating which client generated the error.

A fatal GPU exception triggered by one client will terminate the GPU activity of all clients.

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

2.3.4. MPS On Multi-GPU Systems

The MPS server only supports sharing a single GPU. On systems with more than one GPU MPS may only be used for one of the installed GPUs. If the MPS server is started and can enumerate more than one GPU it will terminate and log an error. To start an MPS server on a multi-GPU machine you must use CUDA_VISIBLE_DEVICES. See section 4.2 for more details.

2.3.5. Performance

2.3.5.1. Client-Server Connection Limits

The MPS Server supports 16 client CUDA contexts concurrently. These contexts may be distributed over up to 16 processes. If the connection limit is exceed 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.

2.3.5.2. Increased Launch Latency

MPS clients incur increased GPU kernel launch and memcopy initiation latency due to the necessary communication with the MPS server. Launch latency of MPS clients should be on the order of 15-25us, but may vary based on system load.

2.3.6. Interaction with Tools

2.3.6.1. Debugging and cuda-gdb

Under certain conditions applications invoked from within cuda-gdb (or any CUDA-compatible debugger, such as Allinea DDT) may be automatically run without using MPS, even when MPS automatic provisioning is active. To take advantage of this automatic fallback, no other MPS client applications may be running at the time. This enables debugging of CUDA applications without modifying the MPS configuration for the system.

Here's how it works:

cuda-gdb attempts to run an application and recognizes that it will become an MPS client.

The application running under cuda-gdb blocks in cuInit() and waits for all of the active MPS client processes to exit, if any are running.

Once all client processes have terminated, the MPS server will allow cuda-gdb and the application being debugged to continue.

If any new client processes attempt to connect to the MPS server while cuda-gdb is running, the new MPS client will block in cuInit() until the debugger has terminated.

The client applications will continue normally after the debugger has terminated.

2.3.6.2. cuda-memcheck

Cuda-memcheck is supported on MPS. See the cuda-memcheck documentation for usage instructions.

2.3.6.3. Profiling

CUDA profiling tools (such as nvprof and Nvidia Visual Profiler) and CUPTI based profilers are supported under MPS. See the profiler documentation for usage instructions.

ARCHITECTURE

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 GPUs compute engine.

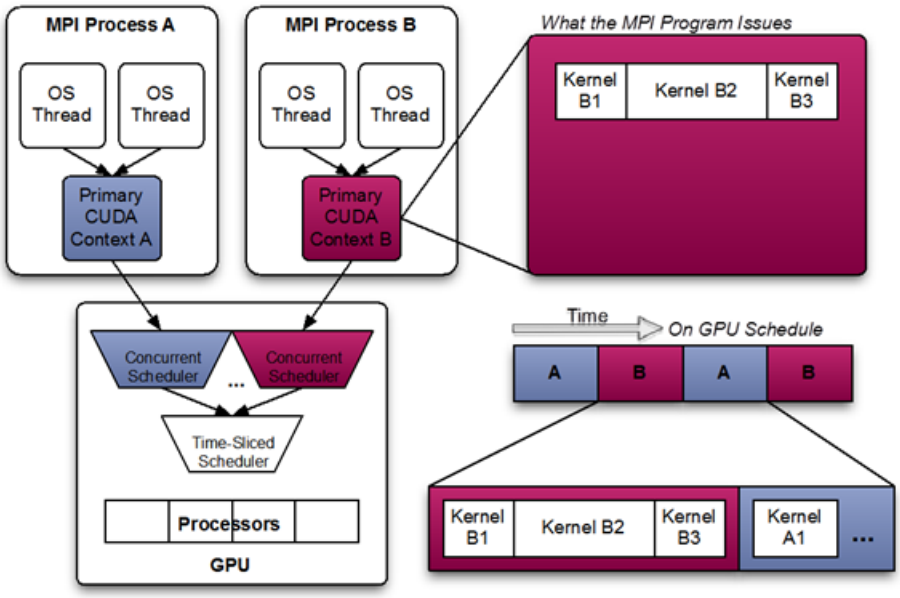
All work on the GPU launched using CUDA is launched either explicitly into a CUDA stream, or implicitly using a default stream. A stream is a software abstraction that represents a sequence of commands, which may be a mix of kernels, copies, and other commands, that execute in order. Work launched in 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. GPU's 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.

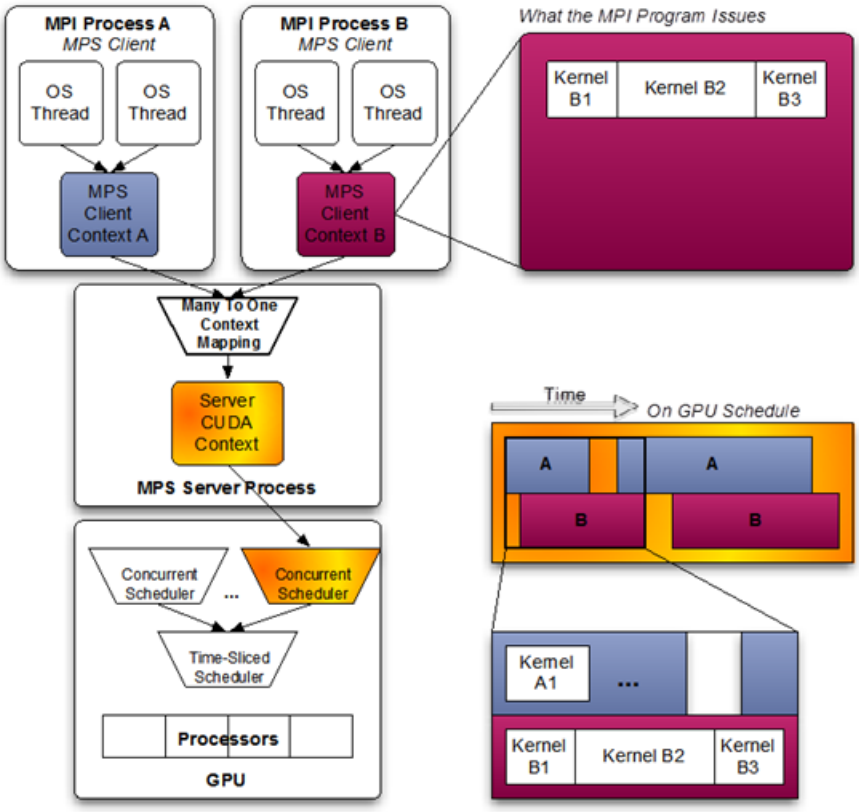
The GPU also has a time sliced scheduler to schedule work from work queues belonging to different CUDA contexts. Work launched to the compute engine from work queues belonging to different CUDA contexts cannot execute concurrently. This can cause underutilization of the GPU’s compute resources if work 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.

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.

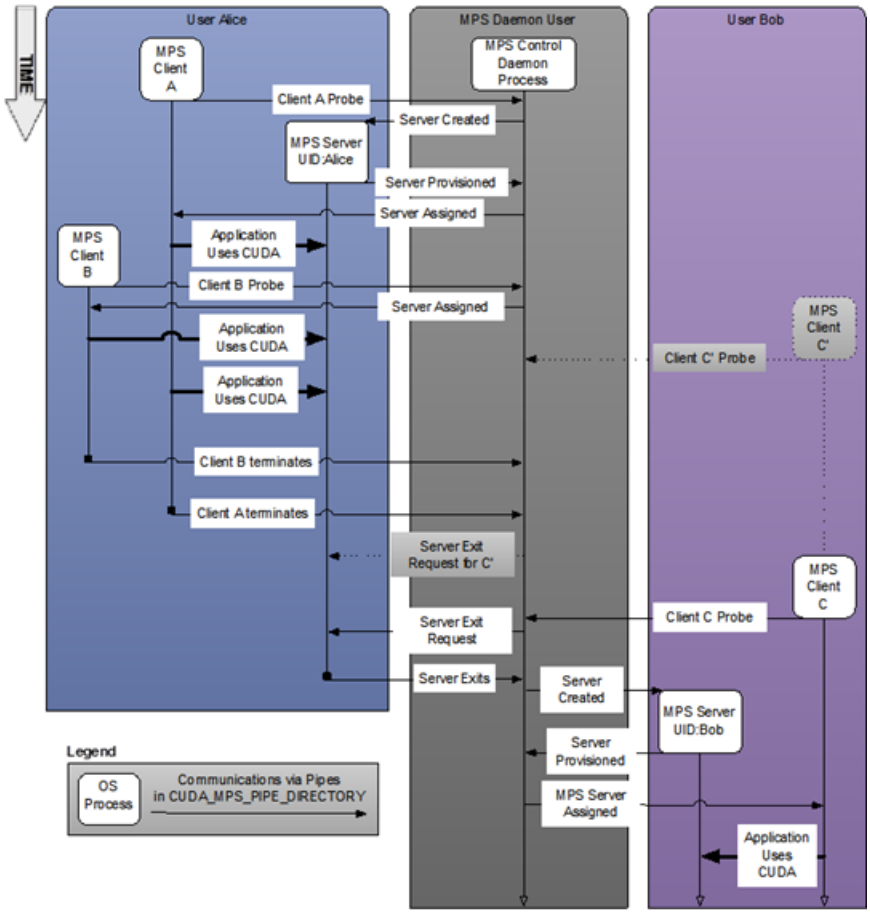


When using MPS, the server manages the hardware resources associated with a single CUDA context. The CUDA contexts belonging to MPS clients funnel their work through the MPS server. This allows the client CUDA contexts to bypass the hardware limitations associated with time sliced scheduling, and permit their CUDA kernels execute simultaneously.

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 MPI 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.

3.3. Provisioning Sequence



System-wide provisioning with multiple users.

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 id's, 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 avialable. 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 only when a new MPS client, launched with a different user id than the active MPS server ,connects to the control daemon. 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 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.

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.

All communication between the MPS client, the MPS control daemon, and the MPS server is done using named pipes. The MPS server launches a worker thread to receive commands from the client. Upon client process exit, the server destroys any resources not explicitly freed by the client process and terminates the worker thread.

APPENDIX: TOOLS AND INTERFACE REFERENCE

The following utility programs and environment variables are used to manage the MPS execution environment. They are described below, along with other relevant pieces of the standard CUDA programming environment.

4.1. Utilities and Daemons

4.1.1. nvidia-cuda-mps-control

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

man nvidia-cuda-mps-control # Describes usage of this utility.

nvidia-cuda-mps-control -d # Start daemon as a background process.

ps -ef | grep MPS # See if the MPS daemon is running.

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

nvidia-cuda-mps-control # Start in interactive mode.

When used in interactive mode, the available commands are

get_server_list – this will print out a list of all PIDs of server instances.
start_server –uid <user id> - this will manually start a new instance of nvidia-cuda-mps-server with the given user ID.
get_client_list <PID> - this lists the PIDs of client applications connected to a server instance assigned to the given PID
quit – terminates the nvidia-cuda-mps-control daemon

Only one instance of the nvidia-cuda-mps-control daemon should be run per node.

4.1.2. nvidia-cuda-mps-server

Typically stored under /usr/bin on Linux 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.

4.1.3. nvidia-smi

Typically stored under /usr/bin on Linux systems, this is used to configure GPU's on a node. The following use cases are relevant to managing MPS:

man nvidia-smi # Describes usage of this utility.

nvidia-smi -L # List the GPU's on node.

nvidia-smi -q # List GPU state and configuration information.

nvidia-smi -q -d compute # Show the compute mode of each GPU.

nvidia-smi -i 0 -c EXCLUSIVE_PROCESS # Set GPU 0 to exclusive mode, run as root.

nvidia-smi -i 0 -c DEFAULT # Set GPU 0 to default mode, run as root. (SHARED_PROCESS)

nvidia-smi -i 0 -r # Reboot GPU 0 with the new setting.

4.2. Environment Variables

4.2.1. 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 is present in the sequence are visible to CUDA applications and they are enumerated in the order of the sequence. If one of the indices is invalid, only the devices whose index precedes the invalid index are visible to CUDA applications.

For example, setting CUDA_VISIBLE_DEVICES to 2,1 causes device 0 to be invisible and device 2 to be enumerated before device 1. Setting CUDA_VISIBLE_DEVICES to 0,2,-1,1 causes devices 0 and 2 to be visible and device 1 to be invisible.

When running an application with MPS, this variable must be set in the environment of the MPS control daemon if the node contains more than one GPU. The MPS

server will fail to start if more than one device is visible after the application of CUDA_VISIBLE_DEVICES.

4.2.2. CUDA_MPS_PIPE_DIRECTORY

The MPS control daemon, the MPS server, and the associated MPS clients communicate with each other via named pipes. The default directory for these pipes is /tmp/nvidia-mps. The environment variable, CUDA_MPS_PIPE_DIRECTORY, can be used to override the location of these pipes. 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 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. When provisioning MPS on a per-user basis, the pipe directory should set to a location such that different users will not end up using the same directory.

4.2.3. 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.

4.2.4. 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.

4.3. MPS Logging Format

4.3.1. Control Log

The control daemon's log file contains information about the following:

Startup and shutdown of MPS servers identified by their process id's and the user id with which they are being launched.

```
[2013-08-05 12:50:23.347 Control 13894] Starting new server 13929 for user 500

[2013-08-05 12:50:24.870 Control 13894] NEW SERVER 13929: Ready

[2013-08-05 13:02:26.226 Control 13894] Server 13929 exited with status 0
```

New MPS client connections identified by the client process id and the user id of the user that launched the client process.

```
[2013-08-05 13:02:10.866 Control 13894] NEW CLIENT 19276 from user 500: Server
 already exists

[2013-08-05 13:02:10.961 Control 13894] Accepting connection...
```

User commands issued to the control daemon and their result.

```
[2013-08-05 12:50:23.347 Control 13894] Starting new server 13929 for user 500

[2013-08-05 12:50:24.870 Control 13894] NEW SERVER 13929: Ready
```

Error information such as failing to establish a connection with a client.

```
[2013-08-05 13:02:10.961 Control 13894] Accepting connection...

[2013-08-05 13:02:10.961 Control 13894] Unable to read new connection type
 information
```

4.3.2. Server Log

The server's log file contains information about the following:

New MPS client connections and disconnections identified by the client process id.

```
[2013-08-05 13:00:09.269 Server 13929] New client 14781 connected

[2013-08-05 13:00:09.270 Server 13929] Client 14777 disconnected
```

Error information such as the MPS server failing to start due to system requirements not being met.

```
[2013-08-06 10:51:31.706 Server 29489] MPS server failed to start

[2013-08-06 10:51:31.706 Server 29489] MPS is only supported on 64-bit Linux
 platforms, with an SM 3.5 or higher Tesla/Quadro GPU.

[2013-08-06 10:51:31.706 Server 29489] MPS is not supported on multi-GPU
 configurations. Please use CUDA_VISIBLE_DEVICES to select the device on which
 the MPS server should be run.
```

APPENDIX: COMMON TASKS

The convention for using MPS will vary between system environments. The Cray environment, for example, manages MPS in a way that is almost invisible to the user, whereas other Linux-based systems may require the user to manage activating the control daemon themselves. As a user you will need to understand which set of conventions is appropriate for the system you are running on. Some cases are described in this section.

5.1. Starting and Stopping MPS on LINUX

5.1.1. 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.

5.1.1.1. Starting MPS control daemon

As root, run the commands

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.

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

5.1.1.2. Shutting Down MPS control daemon

To shut down the daemon, as root, run

echo quit | nvidia-cuda-mps-control

5.1.1.3. Log Files

You can view the status of the daemons by viewing the log files in

/var/log/nvidia-mps/control.log

/var/log/nvidia-mps/server.log

These are typically only visible to users with administrative privileges.

5.1.2. 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

5.1.2.1. Starting MPS control daemon

As $UID, run the commands

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.

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.

5.1.2.2. 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

5.1.2.3. Shutting Down MPS

To shut down the daemon, as $UID, run

echo quit | nvidia-cuda-mps-control

5.1.2.4. Log Files

You can view the status of the daemons by viewing the log files in

$CUDA_MPS_LOG_DIRECTORY/control.log

$CUDA_MPS_LOG_DIRECTORY/server.log

5.1.3. Scripting a Batch Queuing System

5.1.3.1. Basic Principles

Chapters 3-4 describe the MPS components, software utilities, and the environment variables that control them. However, using MPS at this level puts a burden on the user since

At the application level, the user only cares whether MPS is engaged or not, and should not have to understand the details of environment settings etc. when they are unlikely to deviate from a fixed configuration.

There may be consistency conditions that need to be enforced by the system itself, such as clearing CPU- and GPU- memory between application runs, or deleting zombie processes upon job completion.

Root-access (or equivalent) is required to change the mode of the GPU.

We recommend you manage these details by building some sort of automatic provisioning abstraction on top of the basic MPS components. This section discusses how to implement a batch-submission flag in the PBS/Torque queuing environment and discusses MPS integration into a batch queuing system in-general.

5.1.3.2. Per-Job MPS Control: A Torque/PBS Example

Note: Torque installations are highly customized. Conventions for specifying job resources vary from site to site and we expect that, analogously, the convention for enabling MPS could vary from site to site as well. Check with your system's administrator to find out if they already have a means to provision MPS on your behalf.

Tinkering with nodes outside the queuing convention is generally discouraged since jobs are usually dispatched as nodes are released by completing jobs. It is possible to enable MPS on a per-job basis by using the Torque prologue and epilogue scripts to start and stop the nvidia-cuda-mps-control daemon. In this example, we re-use the “account” parameter to request MPS for a job, so that the following command.

qsub -A “MPS=true” …

will result in the prologue script starting MPS as shown:

```
# Activate MPS if requested by user

USER=$2

ACCTSTR=$7

echo $ACCTSTR | grep -i "MPS=true"

if [ $? -eq 0 ]; then

nvidia-smi -c 3

USERID=`id -u $USER`

export CUDA_VISIBLE_DEVICES=0

nvidia-cuda-mps-control -d && echo "MPS control daemon started"

sleep 1
```

```
echo "start_server -uid $USERID" | nvidia-cuda-mps-control && echo "MPS server started for $USER"
fi
```

and the epilogue script stopping MPS as shown:

```
# Reset compute mode to default
nvidia-smi -c 0
# Quit cuda MPS if it's running
ps aux | grep nvidia-cuda-mps-control | grep -v grep > /dev/null
if [ $? -eq 0 ]; then
echo quit | nvidia-cuda-mps-control
fi
# Test for presence of MPS zombie
ps aux | grep nvidia-cuda-mps | grep -v grep > /dev/null
if [ $? -eq 0 ]; then
logger "`hostname` epilogue: MPS refused to quit! Marking offline"
pbsnodes -o -N "Epilogue check: MPS did not quit" `hostname`
fi
# Check GPU sanity, simple check
nvidia-smi > /dev/null
if [ $? -ne 0 ]; then
logger "`hostname` epilogue: GPUs not sane! Marking `hostname` offline"
pbsnodes -o -N "Epilogue check: nvidia-smi failed" `hostname`
fi
```