MULTI-PROCESS SERVICE

vR550 | February 2024

Multi-Process Service

Introduction... 1
1.1. AT A GLANCE... 1
1.1.1. MPS.. 1
1.1.2. Volta MPS.. 1
1.1.3. Intended Audience.. 2
1.1.4. Organization of This Document... 2
1.2. Prerequisites... 2
1.3. Concepts.. 3
1.3.1. Why MPS is needed... 3
1.3.2. What MPS is.. 3
1.4. See Also... 3
When to use MPS.. 4
2.1. The Benefits of MPS.. 4
2.1.1. GPU utilization.. 4
2.1.2. Reduced on-GPU context storage... 4
2.1.3. Reduced GPU context switching.. 4
2.2. Identifying Candidate applications.. 4
2.3. Considerations.. 5
2.3.1. System Considerations.. 5
2.3.1.1. Limitations... 5
2.3.1.2. GPU Compute Modes.. 5
2.3.2. Application Considerations... 6
2.3.3. Memory Protection and Error Containment... 6
2.3.3.1. Memory Protection... 6
2.3.3.2. Error Containment.. 7
2.3.4. MPS on Multi-GPU Systems.. 8
2.3.5. Performance.. 8
2.3.5.1. Client-Server Connection Limits... 8
2.3.5.2. Volta MPS Execution Resource Provisioning.. 9
2.3.5.3. Threads & Linux Scheduling... 10
2.3.5.4. Volta MPS Device Memory Limit.. 11
2.3.6. Interaction with Tools... 11
2.3.6.1. Debugging and cuda-gdb... 11
2.3.6.2. cuda-memcheck... 12
2.3.6.3. Profiling... 12
2.3.7. Client Early Termination.. 12
2.3.8. Client Priority Level Control.. 13
Architecture... 14
3.1. Background... 14
3.2. Client-server Architecture... 15
3.3. Provisioning Sequence.. 17

3.3.1. Server...... 17
3.3.2. Client Attach/Detach...... 18
Appendix: Tools and Interface Reference...... 19
4.1. Utilities and Daemons...... 19
4.1.1. nvidia-cuda-mps-control...... 19
4.1.2. nvidia-cuda-mps-server...... 21
4.1.3. nvidia-smi...... 21
4.2. Environment Variables...... 21
4.2.1. CUDA_VISIBLE_DEVICES...... 21
4.2.2. CUDA_MPS_PIPE_DIRECTORY...... 22
4.2.3. CUDA_MPS_LOG_DIRECTORY...... 22
4.2.4. CUDA_DEVICE_MAX_CONNECTIONS...... 22
4.2.5. CUDA_MPS_ACTIVE_THREAD_PERCENTAGE...... 23
4.2.5.1. MPS Control Daemon Level...... 23
4.2.5.2. Client Process Level...... 23
4.2.5.3. Client CUDA Context Level...... 23
4.2.6. CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PARTITIONING...... 24
4.2.7. CUDA_MPS_PINNED_DEVICE_MEM_LIMIT...... 24
4.2.8. CUDA_MPS_CLIENT_PRIORITY...... 25
4.3. MPS Logging Format...... 25
4.3.1. Control Log...... 25
4.3.2. Server Log...... 26
4.4. MPS KNOWN ISSUES...... 27
Appendix: Common Tasks...... 28
5.1. Starting and Stopping MPS on LINUX...... 28
5.1.1. On a Multi-User System...... 28
5.1.1.1. Starting MPS control daemon...... 28
5.1.1.2. Shutting Down MPS control daemon...... 28
5.1.1.3. Log Files...... 28
5.1.2. On a Single-User System...... 29
5.1.2.1. Starting MPS control daemon...... 29
5.1.2.2. Starting MPS client application...... 29
5.1.2.3. Shutting Down MPS...... 29
5.1.2.4. Log Files...... 29
5.1.3. Scripting a Batch Queuing System...... 29
5.1.3.1. Basic Principles...... 30
5.1.3.2. Per-Job MPS Control: A Torque/PBS Example...... 30
5.2. BEST PRACTICE FOR SM PARITITIONING...... 31

INTRODUCTION

1.1. AT A GLANCE

1.1.1. MPS

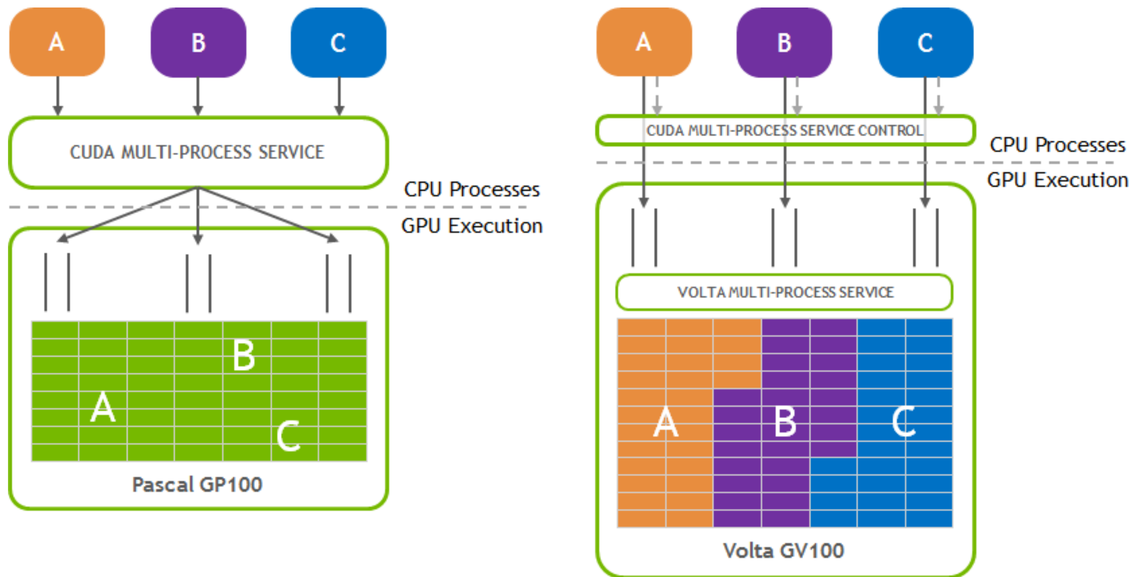
The Multi-Process Service (MPS) is an alternative, binary-compatible implementation of the CUDA Application Programming Interface (API). The MPS runtime architecture is designed to transparently enable co-operative multi-process CUDA applications, typically MPI jobs, to utilize Hyper-Q capabilities on the latest NVIDIA (Kepler and later) 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. Volta MPS

The Volta architecture introduced new MPS capabilities. Compared to MPS on pre-Volta GPUs, Volta MPS provides a few key improvements:

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

This document will introduce the new capabilities and note the differences between Volta MPS and MPS on pre-Volta GPUs. Running MPS on Volta will automatically enable the new capabilities.



1.1.3. 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.

1.1.4. 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 its clients. Volta MPS supports increased isolation between MPS clients, so the resource reduction is to a much lesser degree.

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-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.

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 is not supported on Tegra platforms. The MPS server will fail to start when launched on Tegra platforms.
- MPS requires a GPU with compute capability version 3.5 or higher. The MPS server will fail to start if one of the GPUs 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.
- 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

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. 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.

2.3.2. Application Considerations

- The NVIDIA Codec SDK: https://developer.nvidia.com/nvidia-video-codec-sdk is not supported under MPS on pre-Volta MPS clients.
- 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 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 on pre-Volta MPS clients. Calling any stream callback APIs will return an error.
- CUDA graphs with host nodes are not supported under MPS on pre-Volta MPS clients.
- The amount of page-locked host memory that pre-Volta 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.
- 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.
- CUDA IPC between CUDA contexts which are created by processes running as MPS clients and CUDA contexts which are created by processes not running as MPS clients is supported under Volta MPS.

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

Volta MPS client processes have fully isolated GPU address spaces.

Pre-Volta 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 pre-Volta MPS 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

Volta MPS supports limited 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 till 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, i.e., 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.

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

- A GPU fault generated by any client will be reported to all clients, without indicating which client generated the error.
- A fatal GPU fault triggered by one client will terminate the MPS server and 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 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. See section 4.2 for more details.

On systems with a mix of Volta / pre-Volta GPUs, if the MPS server is set to enumerate any Volta GPU, it will discard all pre-Volta GPUs. In other words, the MPS server will either operate only on the Volta GPUs and expose Volta capabilities or operate only on pre-Volta GPUs.

2.3.5. Performance

2.3.5.1. Client-Server Connection Limits

The pre-Volta MPS Server supports up to 16 client CUDA contexts per-device concurrently. Volta MPS server supports 48 client CUDA contexts per-device. These contexts may be distributed over multiple processes. 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.

2.3.5.2. Volta MPS Execution Resource Provisioning

Volta 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. See section 4.1.1 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. See section 4.2.5.1 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. See section 4.2.5.2 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. See sections 4.2.5.3 and 4.2.6 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(). See section 5.2 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 client1 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.

2.3.5.3. Threads & Linux Scheduling

On pre-Volta GPUs, launching more MPS clients than there are available logical cores on your machine will incur increased launch latency and will generally slow down client-server communication due to how the threads get scheduled by the Linux CFS (Completely Fair Scheduler). For setups where multiple GPUs are used with an MPS control daemon and server started per GPU, we recommend pinning each MPS server

to a distinct core. This can be accomplished by using the utility 'taskset', which allows binding a running program to multiple cores or launching a new one on them. To accomplish this with MPS, launch the control daemon bound to a specific core, e.g., `taskset –c 0 nvidia-cuda-mps-control –d`. The process affinity will be inherited by the MPS server when it starts up.

2.3.5.4. Volta MPS Device Memory Limit

On Volta MPS, 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.

2.3.6. Interaction with Tools

2.3.6.1. Debugging and cuda-gdb

On Volta MPS, GPU coredumps can be generated and debugged using cuda-gdb. See cuda-gdb documentation for usage instructions.

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.
- Any new client processes attempt to connect to the MPS daemon will be provisioned a server normally.

2.3.6.2. cuda-memcheck

The cuda-memcheck tool 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.

2.3.7. Client Early Termination

Terminating a 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.

On Volta MPS, user can instruct the MPS server to terminate the CUDA contexts of a MPS client process, regardless of whether the CUDA contexts are idle or not, by using the control command 'terminate_client <server PID> <client PID>'. This mechanism enables user to terminate the CUDA contexts of a given MPS client process, even when the CUDA contexts are non-idle, without affecting the MPS server or its other MPS clients. The control command 'terminate_client' 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 is set because the target MPS client process might want to:

Perform clean-up of its GPU or CPU state. This may include a device reset. Continue remaining CPU work.

Continue remaining CPU work.

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 a 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 a 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':

Use the control command 'ps' to get the status of the current active MPS clients

$ 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

Terminate using the PID of 'nbody' in the host PID namespace as reported by 'ps':

$ echo "terminate_client 6472 9741" | nvidia-cuda-mps-control

#wait until terminate_client to return

#upon successful termination 0 is returned

0

Now it is safe to kill 'nbody'

$kill -9 9741

2.3.8. 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. On Volta MPS, 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.

In this release, 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.

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.

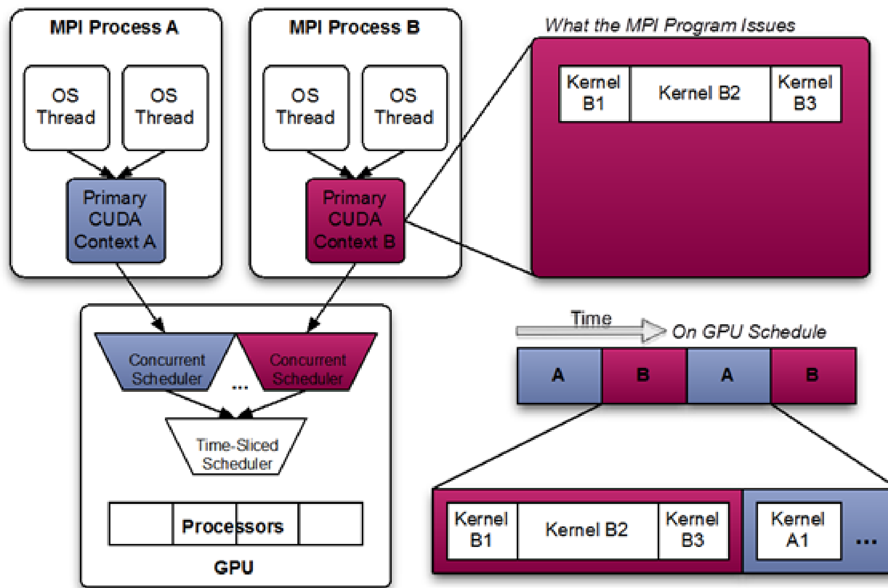
All work on the GPU launched using CUDA is launched either explicitly into a CUDA stream, or implicitly using a default stream. A stream is a software abstraction that represents a sequence of commands, which may be a mix of kernels, copies, and other commands, that execute in order. Work launched in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

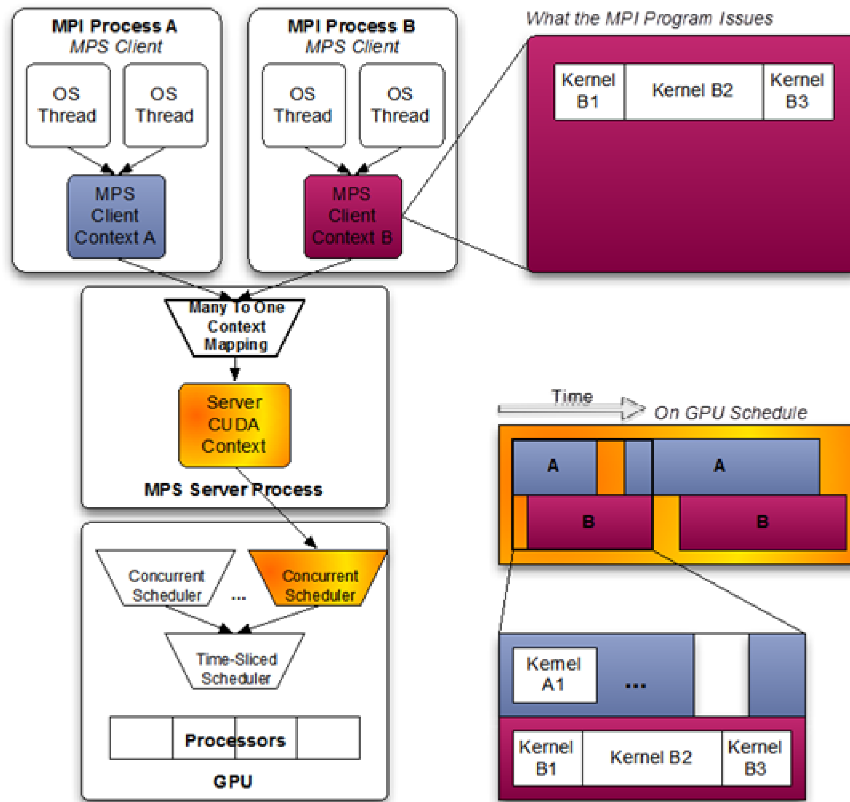
The GPU also has a time sliced scheduler to schedule work from work queues belonging to different CUDA contexts. Work launched to the compute engine from work queues belonging to different CUDA contexts cannot execute concurrently. This can cause underutilization of the GPU's compute resources if work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

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.



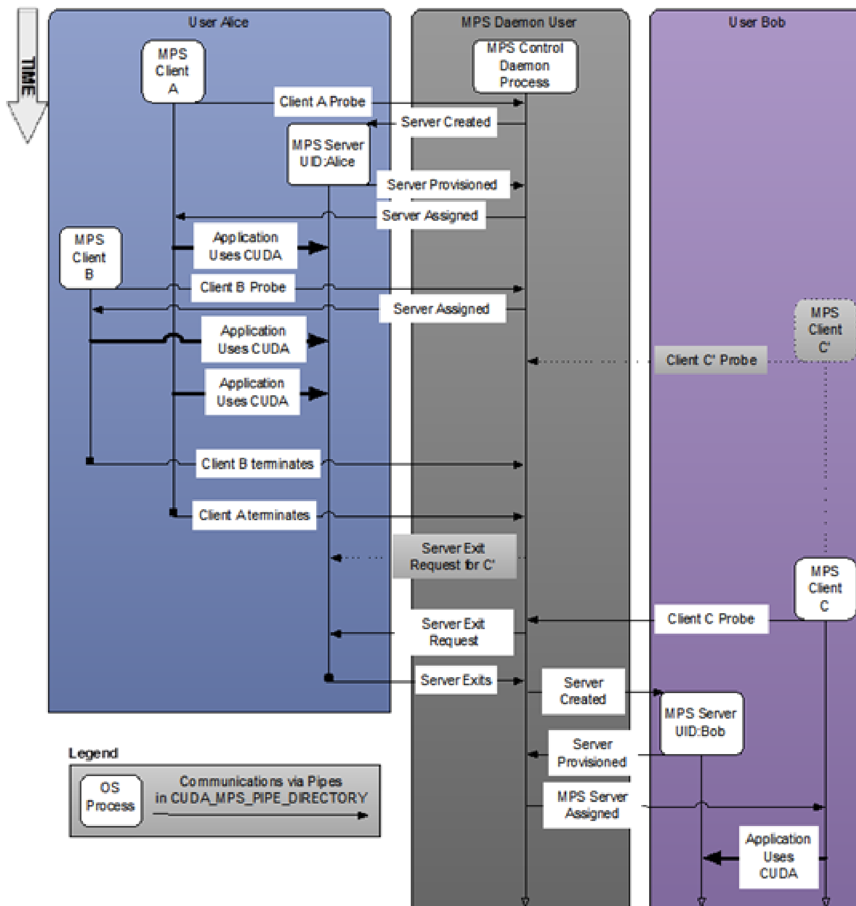
When using pre-Volta MPS, the server manages the hardware resources associated with a single CUDA context. The CUDA contexts belonging to MPS clients funnel their work through the MPS server. This allows the client CUDA contexts to bypass the hardware limitations associated with time sliced scheduling,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

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 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 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 an 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 active 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 and their status 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 and 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.

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 in 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 -f # Start daemon in foreground

The control daemon creates a nvidia-cuda-mps-control.pid file that contains the PID of the control daemon process in the CUDA_MPS_PIPE_DIRECTORY. When there are multiple instances of the control daemon running in parallel, one can target a specific instance by looking up its PID in the corresponding CUDA_MPS_PIPE_DIRECTORY. If CUDA_MPS_PIPE_DIRECTORY is not set, the nvidia-cuda-mps-control.pid file will be created at the default pipe directory at /tmp/nvidia-mps.

When used in interactive mode, the available commands are

get_server_list – this will print 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> - 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

Commands available to Volta MPS control:

get_device_client_list [<PID>] - this 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> - this 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> - this 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> - this 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 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: In order to set limit to 10 gigabytes for device 0, the command used is:

set_default_device_pinned_mem_limit 0 10G.

By default memory limiting is disabled.

get_default_device_pinned_mem_limit <dev> - queries the current default pinned memory limit for the device.

set_device_pinned_mem_limit <PID> <dev> <value> - this 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. Example usage to set memory limit of 900MB for server with pid 1024 for device 0.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.

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

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>. Example usage to terminate the outstanding GPU work for MPS client process with PID 1024 running on MPS server with PID 123: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]- Set 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 -Query the current priority value that will be used for new clients.

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 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.

The MPS control daemon will further filter-out any pre-Volta devices, if any visible device is Volta+.

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.

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 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.

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. 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.

4.2.5. 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.

4.2.5.1. MPS Control Daemon Level

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.

4.2.5.2. Client Process Level

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 constraint 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.

4.2.5.3. Client CUDA Context Level

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. See section 4.2.6 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.

4.2.6. CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PA

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.

4.2.7. 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_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"

4.2.8. 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.

4.3. MPS Logging Format

4.3.1. Control Log

Some of the example messages logged by the control daemon:
Startup and shutdown of MPS servers identified by their process ids 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

Some of the example messages logged by the MPS server:

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 GPU.
```

Information about fatal GPU error containment on Volta+ MPS.

```
[2022-04-28 15:56:07.410 Other 11570] Volta MPS: status of client {11661, 1} is
 ACTIVE
```

```
[2022-04-28 15:56:07.468 Other 11570] Volta MPS: status of client {11663, 1} is
 ACTIVE
```

```
[2022-04-28 15:56:07.518 Other 11570] Volta MPS: status of client {11643, 2} is
 ACTIVE
```

```
[2022-04-28 15:56:08.906 Other 11570] Volta MPS: Server is handling a fatal GPU
 error.
```

```
[2022-04-28 15:56:08.906 Other 11570] Volta MPS: Server status is FAULT.
```

```
[2022-04-28 15:56:08.906 Other 11570] Volta MPS: status of client {11641, 1} is
 INACTIVE
```

```
[2022-04-28 15:56:08.906 Other 11570] Volta MPS: status of client {11643, 1} is
 INACTIVE
```

```
[2022-04-28 15:56:08.906 Other 11570] Volta MPS: status of client {11643, 2} is
 INACTIVE
```

```
[2022-04-28 15:56:08.906 Other 11570] Volta MPS: The following devices
```

```
[2022-04-28 15:56:08.906 Other 11570] 0
```

```
[2022-04-28 15:56:08.907 Other 11570] 1
```

```
[2022-04-28 15:56:08.907 Other 11570] Volta MPS: The following clients have a
 sticky error set:
```

```
[2022-04-28 15:56:08.907 Other 11570] 11641
```

```
[2022-04-28 15:56:08.907 Other 11570] 11643
```

```
[2022-04-28 15:56:09.200 Other 11570] Client {11641, 1} exit
```

```
[2022-04-28 15:56:09.244 Other 11570] Client {11643, 1} exit
```

```
[2022-04-28 15:56:09.244 Other 11570] Client {11643, 2} exit
```

```
[2022-04-28 15:56:09.245 Other 11570] Volta MPS: Destroy server context on
 device 0
```

```
[2022-04-28 15:56:09.269 Other 11570] Volta MPS: Destroy server context on
 device 1
```

```
[2022-04-28 15:56:10.310 Other 11570] Volta MPS: Creating server context on
 device 0
```

```
[2022-04-28 15:56:10.397 Other 11570] Volta MPS: Creating server context on
 device 1
```

```
[2022-04-28 15:56:10.397 Other 11570] Volta MPS: Server status is ACTIVE.
```

4.4. MPS KNOWN ISSUES

Clients may fail to start, returning ERROR_OUT_OF_MEMORY when the first CUDA context is created, even though there are fewer client contexts than the hard limit of 16.

Comments: When creating a context, the client tries to reserve virtual address space for the Unified Virtual Addressing memory range. On certain systems, this can clash with the system linker and the dynamic shared libraries loaded by it. Ensure that CUDA initialization (e.g., cuInit() , or any cuda*() Runtime API function) is one of the first functions called in your code. To provide a hint to the linker and to the Linux kernel that you want your dynamic shared libraries higher up in the VA space (where it won't clash with CUDA's UVA range), compile your code as PIC (Position Independent Code) and PIE (Position Independent Executable). Refer to your compiler manual for instructions on how to achieve this.

Memory allocation API calls (including context creation) may fail with the following message in the server log: MPS Server failed to create/open SHM segment.

Comments: This is most likely due to exhausting the file descriptor limit on your system. Check the maximum number of open file descriptors allowed on your system and increase if necessary. We recommend setting it to 16384 and higher. Typically this information can be checked via the command 'ulimit –n'; refer to your operating system instructions on how to change the limit.

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
```

5.2. BEST PRACTICE FOR SM PARITITIONING

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]() {
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);
};
}
```

The hardware resources needed for client CUDA contexts is limited and support up to 48 client CUDA contexts per-device on Volta MPS. The size of the context

pool per-device is limited by the number of CUDA client contexts supported per-device. The memory footprint of each client CUDA context and the value of CUDA_DEVICE_MAX_CONNECTIONS may further reduce the number of available clients. Therefore, CUDA client contexts with different SM partitions should be created judiciously.

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.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

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.

© 2013-2024 NVIDIA Corporation. All rights reserved.