

INTRODUCTION TO CUDA's MULTI-PROCESS SERVICE (MPS)



MOTIVATING USE CASE

Given a fixed amount of work to do, divided evenly among N MPI ranks:

- What is the optimal value of N?
- How many GPUs should we distribute these N ranks across? -

global void kernel (double* x, int N) { int i = threadIdx.x + blockIdx.x * blockDim.x; if (i < N) { x[i] = 2 * x[i];



BASE CASE: 1 RANK Run with $N = 1024^3$

_rank_no_MPS_N_1e9.qdrep × 4_ranks_no_MPS_N_1e9.qdrep × ■ Timeline View -75 🗢 +425ms +430ms +435ms +440ms +445ms +450ms +455ms +460ms +465ms +470ms +475ms +480ms +485ms +490ms +495mt 27 498.05ms 2ms ▶ CPU (176) Processes (27) ▼ [162415] ./test CUDA HW (Unknown GPU) kern... kernel(double*,... kernel(double*... kernel(double*... kernel(double*... kernel(double*... 100.0% Kernels Ends: 7.508845 (+ 19 grid: <<<3906250, block: <<<256, 1, 1 Launch Type: Regul Static Shared Memo Threads (1) Dynamic Shared M Registers Per Threa Local Memory Per Local Memory Tota [162382] jsrun Shared Memory e Shared Memory Ba Launched from thr Latency: ←6.184 s Correlation ID: 419 Stream: Stream 7 [162409] ▶ [162401] 23 processes hidden... -+



								<u> </u>	Q 4x		- _ _	<u>1 warning, 9</u>	messages
+505ms +	+510ms	+515	ns	+520ms	+525ms	+530m	s +535	ōms	+540ms	+545ms	+550ms	+555	5ms
2 *	ke	rne	l(do	oub	le*	. ke	rnel	l(do	bub	le*,.	ke	rnel	••••
), 1, 1>>> 1>>> Jar nory: 0 bytes Aemory: 0 bytes ad: 16 Thread: 0 bytes al: 94,371,840 byt kecuted: 0 bytes ank Size: 4 B read: 162415	tes												
•													
													*

3

💿 NVIDIA.

GPU COMPUTE MODES

NVIDIA GPUs have several compute modes

Default: multiple processes can run at one time Exclusive Process: only one process can run at one time Prohibited: no processes can run

Controllable with nvidia-smi --compute-mode; generally needs elevated privileges (so e.g. bsub -alloc flags gpudefault on Summit)





SIMPLE OVERSUBSCRIPTION

The most common oversubscription case uses default mode We simply target the same GPU with N ranks



-	



OVERSUBSCRIPTION: 4 RANKS

Run with $N = 1024^3$

1_rank_no_MPS_N_1e9.qdrep × 4_ran	ks_no_MPS_N_1e9.qdrep ×								
85 -	+430ms +435m	s +440ms	+445ms	+450m	is +45	55ms	+460ms	+465ms 8s	468.10ms +470ms
▶ CPU (176)									
▼ Processes (30)									
▶ [163770] jsrun									
▼ [163802] ./test									
▼ CUDA HW (Unknown GPU)									
▶ 100.0% Kernels	kernel(doub	le*, int)		k	ærnel(double*,	int)			
▼ Threads (1)									
▶ [163802] test									
▼ [163803] ./test									
 CUDA HW (Unknown GPU) 									
▶ 100.0% Kernels		kern	el(double*, int)				kernel(double*, int)	kernel
 Threads (1) 									Begins: 8.45521s Ends: 8.47403s (+18.819 grid: <<<976563, 1, 1>
▼ [163804] ./test									Launch Type: Regular Static Shared Memory: Dynamic Shared Memory
▼ CUDA HW (Unknown GPU)									Registers Per Thread: 10 Local Memory Per Thre Local Memory Total: 94
▶ 100.0% Kernels	kernel(double*, int)		kernel(double*	, int)				kernel(doub	Shared Memory execut Shared Memory Bank S Launched from thread:
 Threads (1) 									Latency: ←7.096 s Correlation ID: 736 Stream: Stream 7
▼ [163805] ./test									
✓ CUDA HW (Unknown GPU)									
▶ 100.0% Kernels		kernel(doub	le*, int)			kernel(d	double*, int)		
 Threads (1) 									
25 processes hidden — +									

				🔤 Q 2x 💳 🗆	A warnings, 24 mess
+475	ms	+480ms	+485ms	+490ms	+495ms +5
ernel(do	ouble* int)			kernel(double	* int)
		ķ	kernel(double*,	int)	
				,	
tes bytes					
bytes 840 bytes					
bytes B			kernel(c	louble*, int)	ke
303					
		kernel(double*,	int)		kernel(double

🕺 NVIDIA.

SIMPLE OVERSUBSCRIPTION

Each rank operates fully independently of all other ranks

Individual processes operate in time slices

A performance penalty is paid for switching between time slices



🕺 NVIDIA

ASIDE: CUDA CONTEXTS

Every process creates its own CUDA context

The context is a stateful object required to run CUDA Automatically created for you when using the CUDA runtime API On V100, the size is ~300 MB + your GPU code size This limits the number of ranks we can fit on the GPU regardless of application data **Context size is partially controlled by** cudaLimitStackSize (more on that later)







Timeslice 1





CPU Processes

GPU Interrupt











Full process isolation, peak throughput optimized for each process



WHEN DOES OVERSUBSCRIPTION HELP? Perhaps a smaller case where launch latency is relevant? (N = 10⁶)

	1_rank_no_MPS_N_1e9.qdrep × 4_ra	anks_no_MPS_N_1e9.qdrep × 1	1_rank_no_MPS_N_1e6.qdr	rep ×																	
x 1000 1100	■ Timeline View ▼																	2003	Q 2x	<u>^ 1 wa</u>	arning, 9 messages
• MORT	25	+112.315ms +112.32ms	+112.325ms +1	112.33ms +112.335m	ns +112.34ms	+112.345ms	+112.35ms	+112.355ms +112.36m	ns +1 <mark>2s 112.</mark>	36637ms -112.37ms	+112.375ms	+112.38ms	+112.385ms +112	39ms +112.395m	is +112.4ms	+112.405ms	+112.41ms +	112.415ms +112	42ms +112.425n	s +112.43ms	+112.435ms
- lossing - Marking Marken Ma	► CPU (176)																				
- Korzy Add Construction of the second of	 Processes (27) 																				
· Not score Seme Sem Se	▼ [106672]./test														_						
• Non server emel(double*, int) kernel(double*, int) kernel(double*, int) kernel(double*, int) • Nata 00	▼ CUDA HW (Unknown GPU)																				
• Next () Performance • Next () Pe	▶ 100.0% Kernels	kernel(doubl		kernel(double	e*, int)			kernel(double	e*, int)	kernel		kernel(d	ouble*, int)			kernel(do	ouble*, int)		kerr	el(double*, i	nt)
 Next Part Part Part Part Part Part Part Par	▶ Threads (1)									Begins: 2.11235s Ends: 2.11237s (+22 grid: <<<4096, 1, 1	2.460 μs) 1>>>										
• (D095) Note: Start Marry 100 Figure 200 Figure	▶ [106631] jsrun									block: <<<256, 1, 1 Launch Type: Regu	>>> Ilar										
• INNOT Image: Image	▶ [106659]									Dynamic Shared Merri Registers Per Threa Local Memory Per	lemory: 0 bytes lemory: 0 bytes id: 16 Thread: 0 bytes										
	▶ [106650]									Local Memory Tota Shared Memory ex Shared Memory Ba	al: 94,371,840 bytes ecuted: 0 bytes ank Size: 4 B										
	23 processes hidden — +	-								Launched from thr Latency: +3.675 m	read: 106672 s										
										Stream: Stream 7											
		4								:											•

WHEN DOES OVERSUBSCRIPTION HELP? Unfortunately, this isn't better.

1_rank_no_MPS_N_1e9.qdrep × 4_ranks_no_MPS_N_1e9.qdrep × 1_rank_no_MPS_N_1e6.qdrep × 4_ranks_no_MPS_N_1e6.qdrep >		
		🖾 Q 2x — <u>A 4 warnings, 24 mess</u>
1s ▼ +310.75ms +310.8ms +310.85ms 1s 31	0.8869ms 0.9ms +310.95ms +311.05ms +311.05ms +311.1ms +311.15ms +311.2ms +311.25ms +311.25ms +311.35ms	+311.35ms +311.4ms
► CPU (176)		
▼ Processes (30)		
▶ [111188] jsrun		
▼ [111221] ./test		
▼ CUDA HW (Unknown GPU)		
▶ 100.0% Kernels	kernel	
▼ Threads (1)	Begins: 1.31088s Ends: 1.31089s (+6.396 μs) grid: <<<1024, 1, 1>>>	
▶ [111221] test	block: <<<2b, 1, 1>>> Launch Type: Regular Static Shared Memory: 0 bytes	
▼ [111222] ./test	Registers Per Thread: 16 Local Memory Per Thread: 0 bytes Local Memory Total: 94,371,840 bytes	
✓ CUDA HW (Unknown GPU)	Shared Memory executed: 0 bytes Shared Memory Bank Size: 4 B Launched from thread: 111221	
▶ 100.0% Kernels	Correlation ID: 1167 Stream: Stream 7	
Threads (1)		
▼ [111223] ./test		
▼ CUDA HW (Unknown GPU)		
▶ 100.0% Kernels		
Threads (1)		
▼ [111224] ./test		
✓ CUDA HW (Unknown GPU)		
▶ 100.0% Kernels		
Threads (1)		
25 processes hidden — +		

14

💿 NVIDIA.

OVERSUBSCRIPTION CONCLUSIONS (when running with the default compute mode)

No free lunch theorem applies: if GPU is fully utilized, cannot get faster answers

For cases that don't fully utilize the GPU, we'd like to fill in gaps in the timeline But with GPU-only workloads, this rarely works out just right to be beneficial Typically performs better when there is CPU-only work to interleave



SCHEDULING: HOW COULD WE DO BETTER?

Pre-emptive scheduling

Processes share GPU through time-slicing Scheduling managed by system





Concurrent scheduling

Processes run on GPU simultaneously User creates & manages scheduling streams





MULTI-PROCESS SERVICE

NVIDIA <u>MPS</u> (Multi-Process Service) improves the situation by allowing multiple process to (instantaneously) share GPU compute resources (SMs)

Designed to **concurrently** map multiple MPI ranks onto a single GPU

Used when each rank is **too small** to fill the GPU on its own





MULTI-PROCESS SERVICE Improving on what we had before!



OVERSUBSCRIPTION WITH MPS

Same case as earlier with $N = 10^9$

MPS mostly recovers performance losses due to context switching

But again, no free lunch theorem applies (no significant speedup either)





OVERSUBSCRIPTION WITH MPS

A smaller case: $N = 2 * 10^7$

Whether or not there's a speedup depends substantially on precise timing





OVERSUBSCRIPTION WITH MPS

A much smaller case: $N = 10^5$

Splitting up work is a clear loser here (quickly get hit by launch latency)





OVERSUBSCRIPTION CONCLUSIONS REDUX

No free lunch theorem still applies: if GPU is fully utilized, cannot get faster answers

Strive to write your application so that you don't need MPS

If you are unable to write kernels that fully saturate the GPU, then consider oversubscription, and MPS is usually always worth turning on for that case

Profile your code to understand why MPS did or did not help





COMPARISON OF PRE- AND POST-VOLTA MPS



Software work submission Limited isolation 16 clients per GPU No provisioning Faster, hardware-accelerated work submission



Hardware memory isolation 48 clients per GPU Execution resource provisioning

KEY DIFFERENCES BETWEEN PRE- AND POST-VOLTA MPS

More MPS clients per GPU: 48 instead of 16

Less overhead: Volta MPS clients submit work directly to the GPU without passing through the MPS server.

More security: Each Volta MPS client owns its own GPU address space instead of sharing GPU address space with all other MPS clients.

More control: Volta MPS supports limited execution resource provisioning for Quality of Service (QoS). -> CUDA MPS ACTIVE THREAD PERCENTAGE

Independent work submission: Each process has private work queues, allowing concurrent submission without contending over locks.



USING MPS

No application modifications necessary

Not limited to MPI applications

MPS control daemon spawns MPS server upon CUDA application startup

Profiling tools are MPS-aware; cuda-gdb doesn't support attaching but you can dump core files

Manually

nvidia-smi -c EXCLUSIVE PROCESS

nvidia-cuda-mps-control -d

On Summit

bsub -alloc flags gpumps

Compute modes

On shared systems, recommended to use EXCLUSIVE_PROCESS mode to ensure that only a single MPS server is using the GPU

• **PROHIBITED** (cannot set device)

• **EXCLUSIVE_PROCESS** (single shared device)

• **DEFAULT** (per-process device)

MPS CONTROL: ENVIRONMENT VARIABLES

These are set per-process; can also manage MPS system-wide via control daemon

CUDA_VISIBLE_DEVICES

Sets devices which an application can see. When set on MPS daemon, limits visible GPUs for all clients.

CUDA_MPS_PIPE_DIRECTORY

Directory where MPS control daemon pipes are created. Clients & daemon must set to same value. Default is /var/log/nvidia-mps.

CUDA_MPS_LOG_DIRECTORY

Directory where MPS control daemon log is created. Default is /tmp/nvidia-mps.

CUDA_DEVICE_MAX_CONNECTIONS

Sets number of hardware work queues that CUDA streams map to. MPS clients all share the same pool, so if set in an MPS-attached process sets this it may limit the max number of MPS processes.

CUDA_MPS_ACTIVE_THREAD_PERCENTAGE

Controls what fraction of GPU may be used by a process - see next slides.



EXECUTION RESOURCE PROVISIONING WITH MPS Using MPS, applications can assign fractions of a GPU to each process

export CUDA MPS ACTIVE THREAD PERCENTAGE=percentage \$

- Environment variable: configures maximum fraction of a GPU available to an MPS-attached process
- Guarantees a process will use at most *percentage* execution resources (SMs)
- Over-provisioning is permitted: sum across all MPS processes may exceed 100%
- Provisions only execution resources (SMs) does not provision memory bandwidth or capacity
- Before CUDA 11.2, all processes be set to the same percentage
- Since CUDA 11.2, percentage may be different for each process

Full details at: https://docs.nvidia.com/deploy/mps/index.html#topic_5_2_5



GPU PROVISIONING WITH MPS Using MPS, applications can assign fractions of a GPU to each process

Image: select			
Image: select			
Image: select			
Image: select			

Fractional Provisioning

Process C could use more, but is limited to just 33% of execution resources

Process B is guaranteed space if needed



Process B is not using all of its allocation Process C may grow to fill available space Additional B work may have to wait for resources

← 3 concurrent MPS processes



Using Oversubscription



THINGS TO WATCH OUT FOR See https://docs.nvidia.com/deploy/mps/index.html for more details

Memory Footprint

To provide a per-thread stack, CUDA reserves 1kB of GPU memory per thread This is (2048 threads per SM x 1kB per thread) = 2 MB per SM used, or **164 MB per client** for V100 (221 MB for A100) CUDA_MPS_ACTIVE_THREAD_PERCENTAGE reduces max SM usage, and so reduces memory footprint Each MPS process also uploads a new copy of the executable code, which adds to the memory footprint

Work Queue Sharing

CUDA maps streams onto CUDA_DEVICE_MAX_CONNECTIONS hardware work queues Queues are normally per-process, but MPS allows 96 hardware queues to be shared among up to 48 clients MPS automatically reduces connections-per-client unless environment variable is set If CUDA_DEVICE_MAX_CONNECTIONS is set (e.g. to enable more concurrency within a process), this can reduce the maximum number of concurrent clients



MPS LOGICAL VS. MIG PHYSICAL PARTITIONING



Multi-Process Service Dynamic contention for GPU resources Single tenant



Hierarchy of instances with guaranteed resource allocation Multiple tenants

Multi-Instance GPU



MULTI-INSTANCE GPU (MIG) Divide a Single A100 GPU Into Multiple Instances, Each With Isolated Paths Through the Entire Memory System



All MIG instances run in parallel with predictable throughput & latency, fault & error isolation

Diverse Deployment Environments Supported with Bare metal, Docker, Kubernetes Pod, Virtualized Environments

Up To 7 GPU Instances In a Single A100

Full software stack enabled on each instance, with dedicated SM, memory, L2 cache & bandwidth

Simultaneous Workload Execution With **Guaranteed Quality Of Service**



CUDA CONCURRENCY MECHANISMS

	Streams	MPS	MIG
Partition Type	Single process	Logical	Physical
Max Partitions	Unlimited	48	7
Performance Isolation	No	By percentage	Yes
Memory Protection	No	Yes	Yes
Memory Bandwidth QoS	No	No	Yes
Error Isolation	No	No	Yes
Cross-Partition Interop	Always	IPC	Limited IPC
Reconfigure	Dynamic	Process launch	When idle

MPS: Multi-Process Service MIG: Multi-Instance GPU





