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

MOTIVATING USE CASE

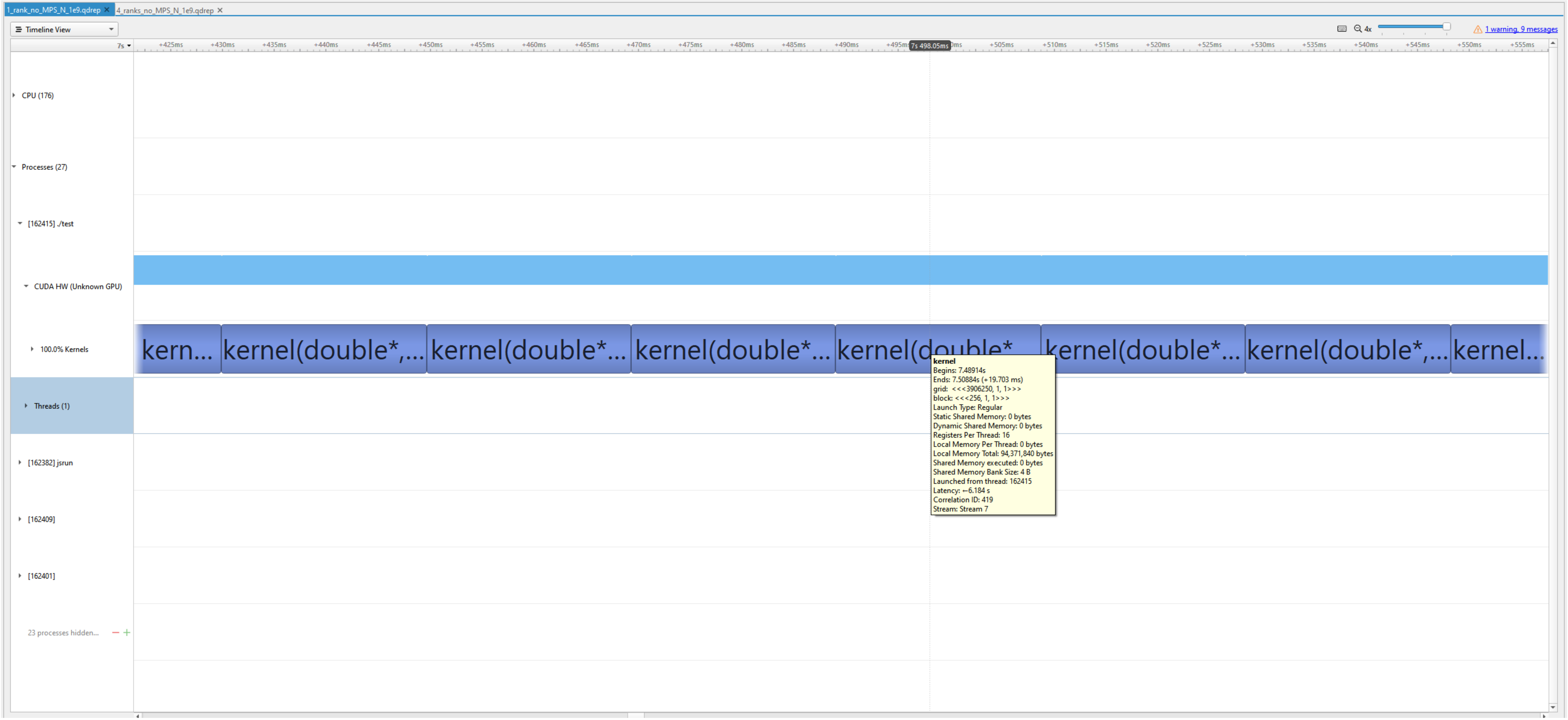
Given a fixed amount of work to do, divided evenly among N MPI ranks:

- 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



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

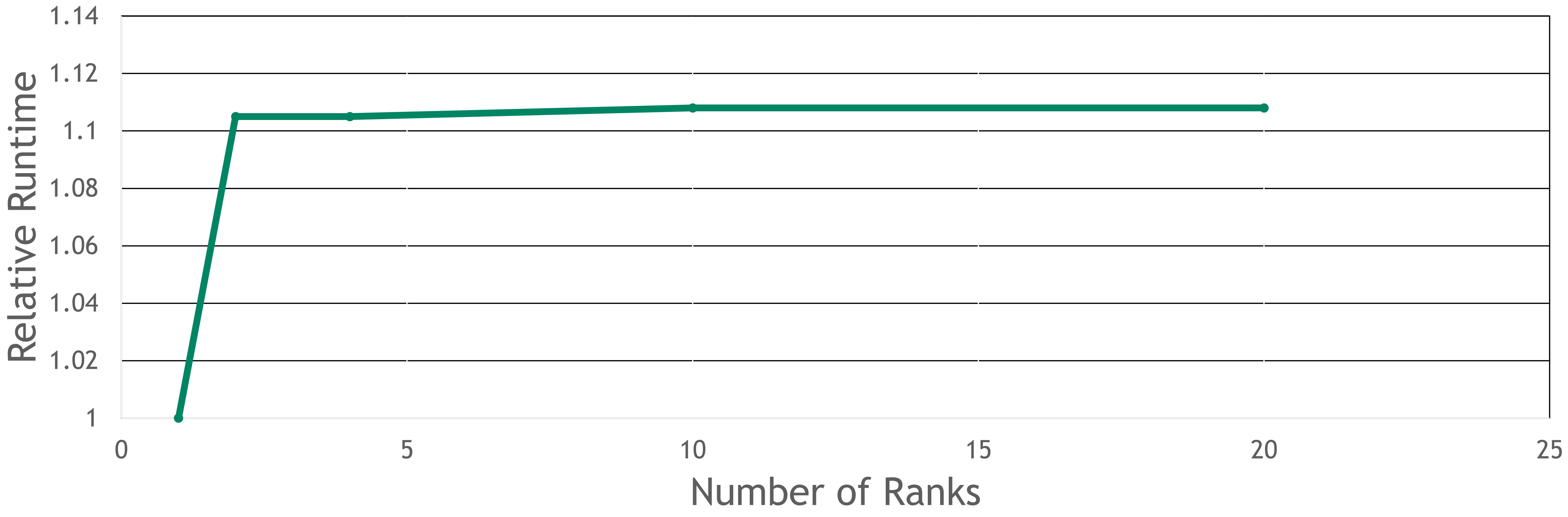
Controllable with `nvidia-smi --compute-mode`; generally needs elevated privileges (so e.g. `bsub -alloc_flags gpudefault` on Summit)

SIMPLE OVERSUBSCRIPTION

The most common oversubscription case uses default mode

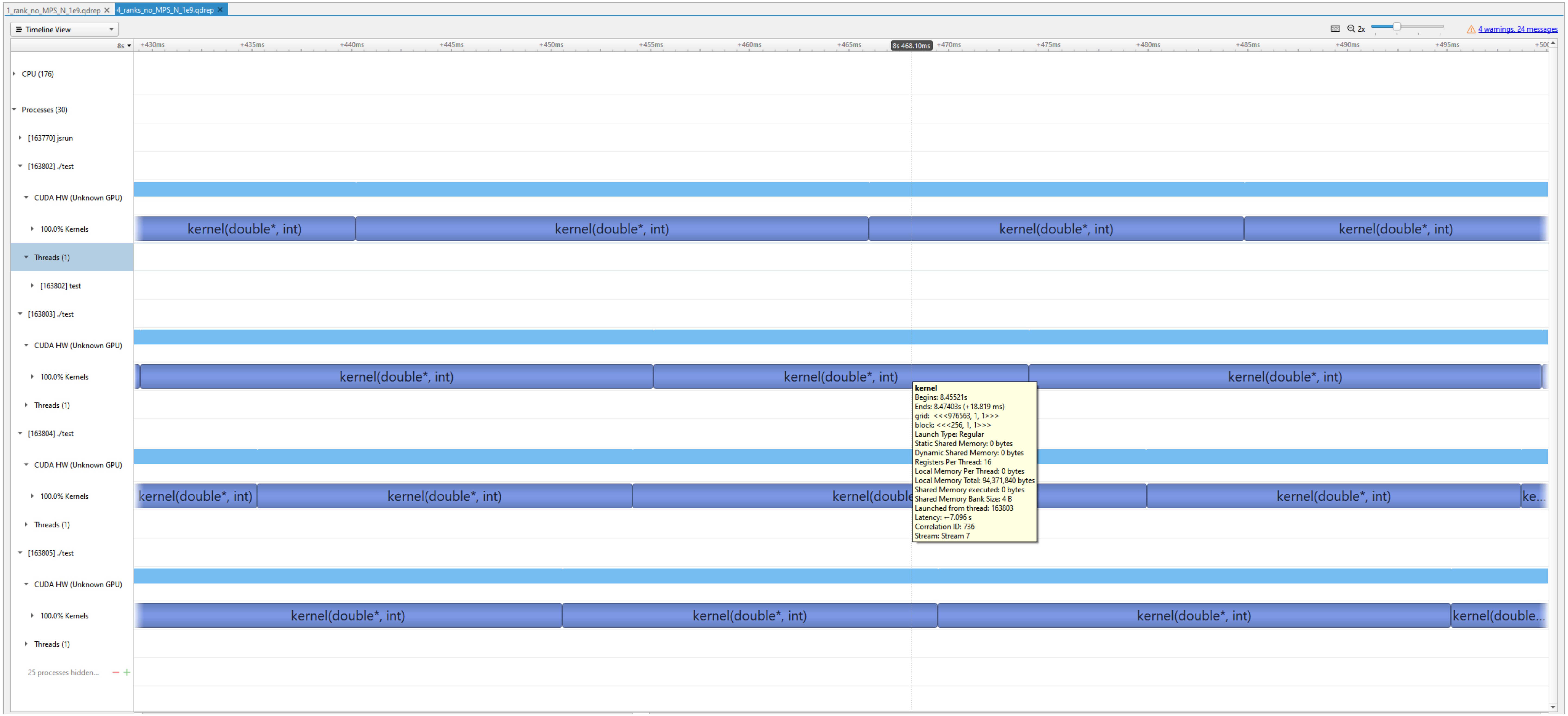
We simply target the same GPU with N ranks

```
$ jsrun -n 1 -a <NUM_RANKS> -g 1 -c <NUM_RANKS> ./test 1073741824
```



OVERSUBSCRIPTION: 4 RANKS

Run with N = 1024^3

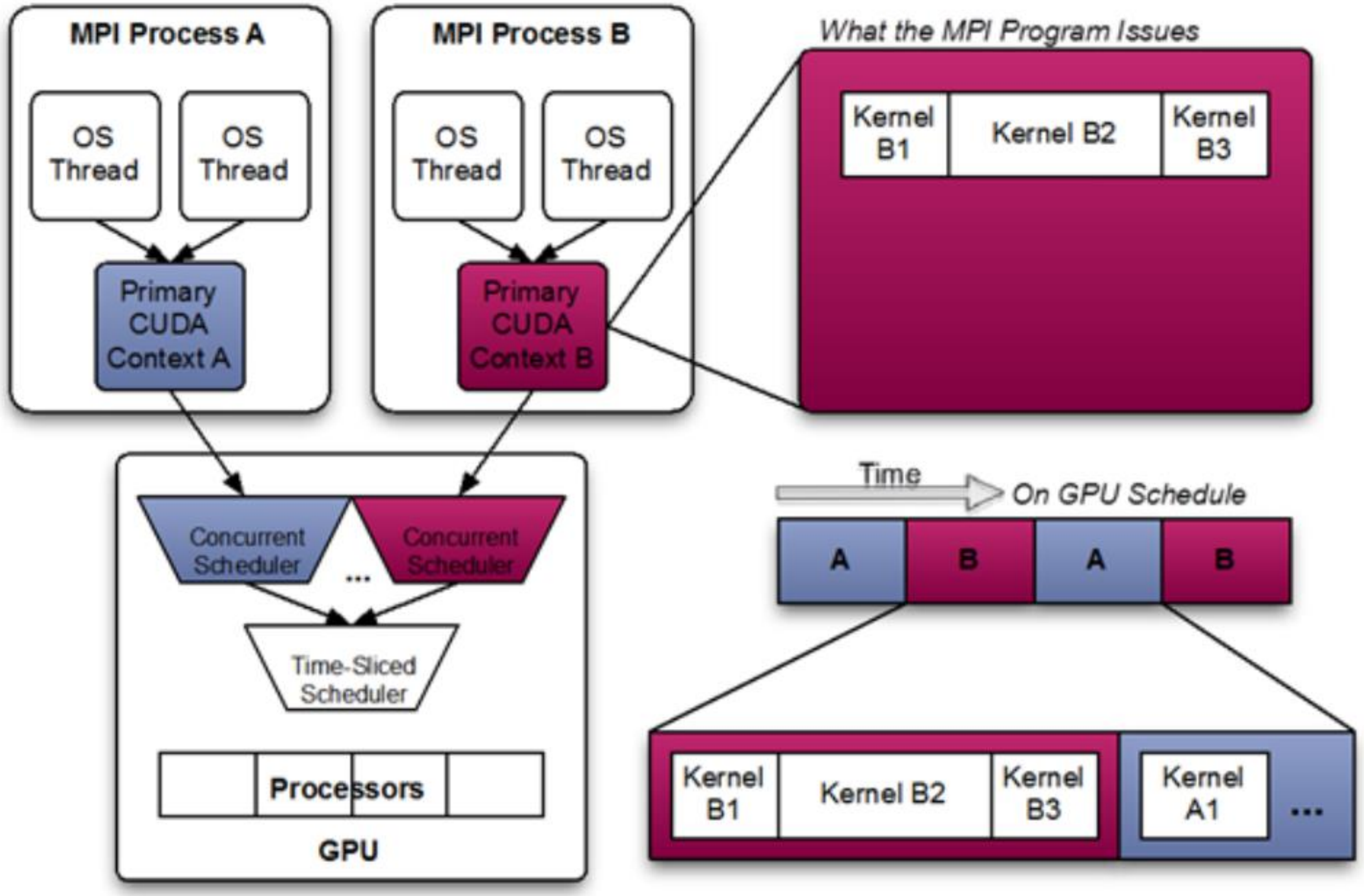


SIMPLE OVERSUBSCRIPTION

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

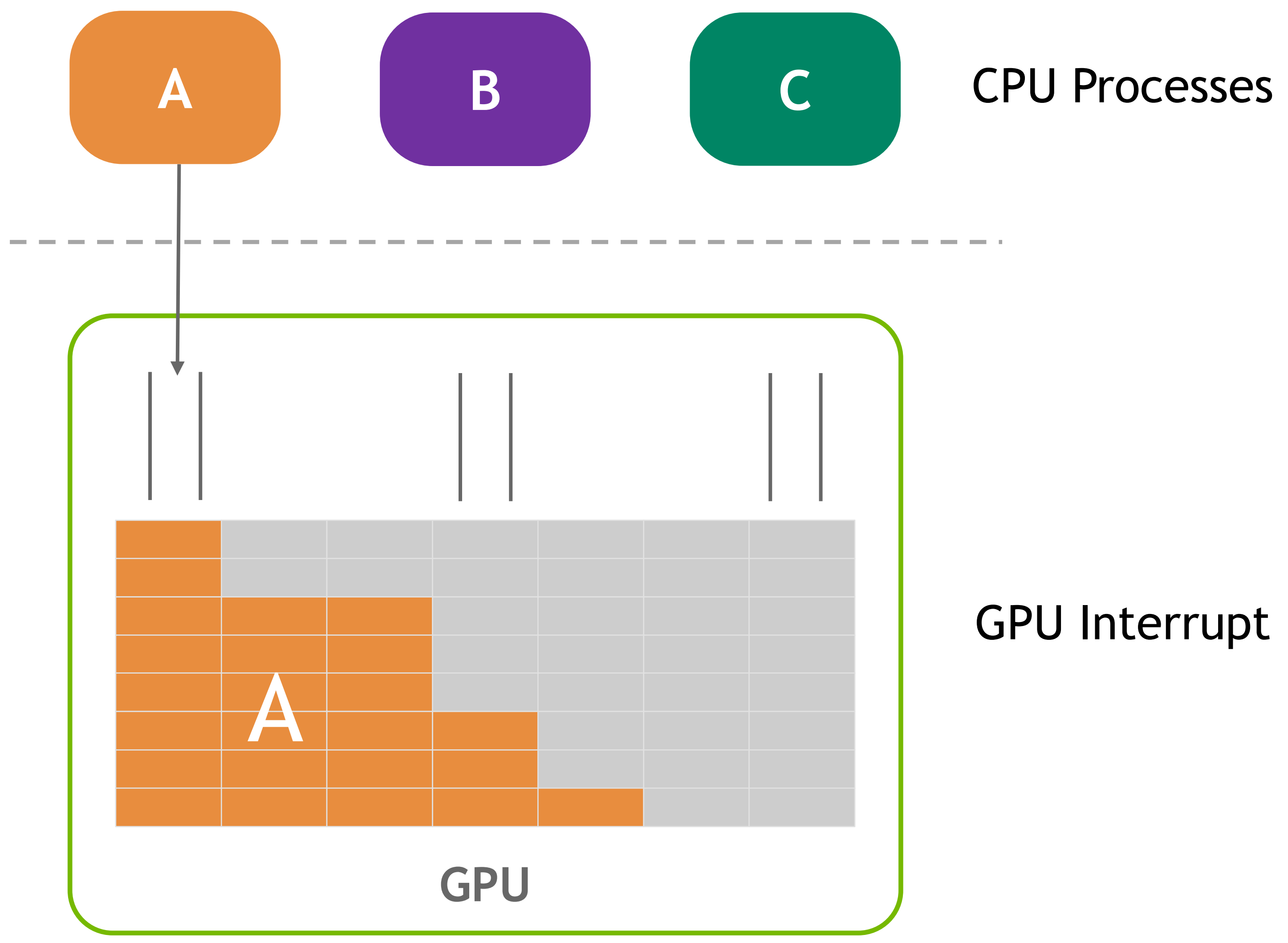


ASIDE: CUDA CONTEXTS

Every process creates its own *CUDA context*

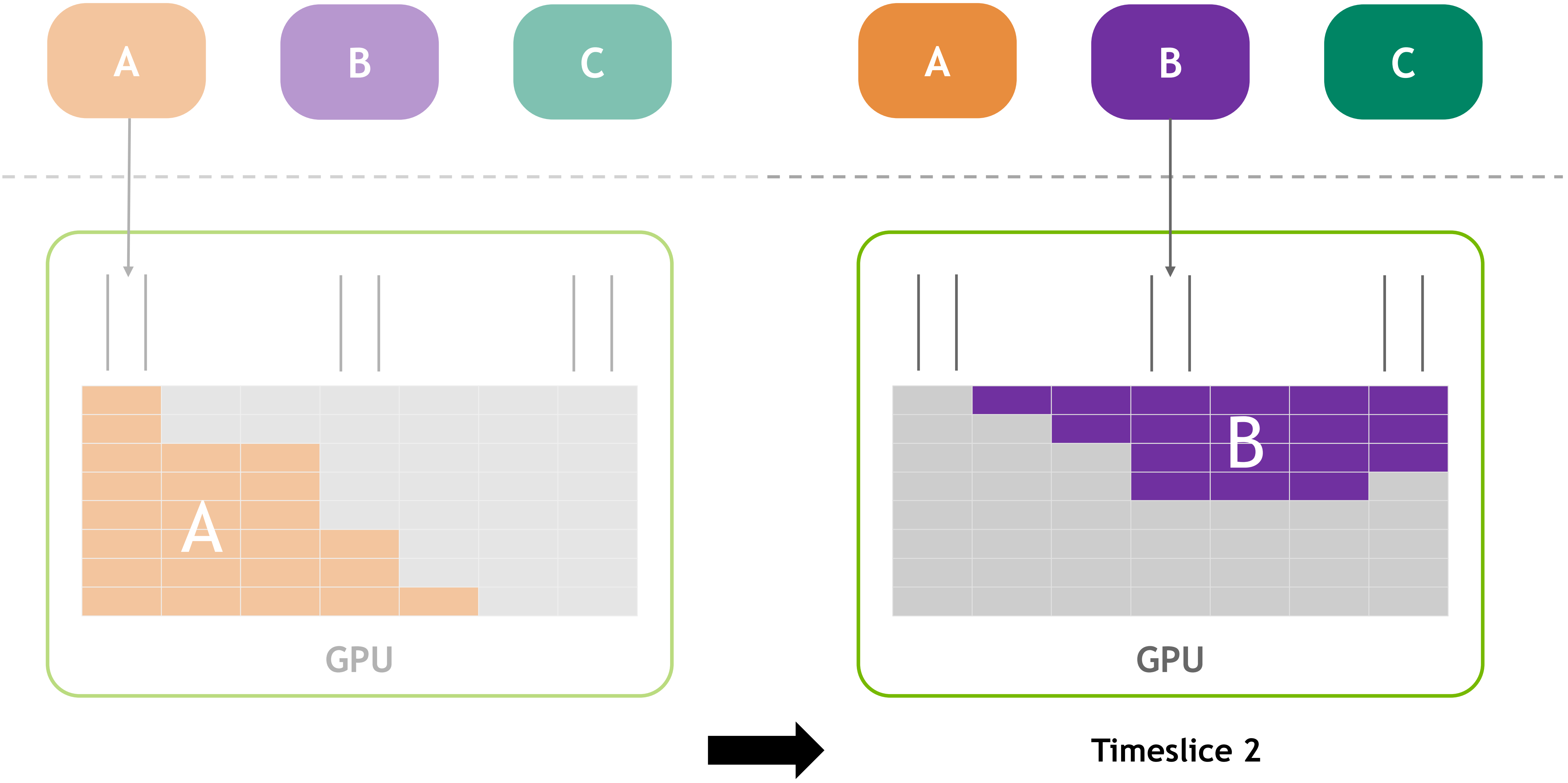
- The context is a stateful object required to run CUDA
- Automatically created for you when using the CUDA runtime API
- On V100, the size is ~300 MB + your GPU code size
- This limits the number of ranks we can fit on the GPU regardless of application data
- Context size is partially controlled by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

MULTI-PROCESS TIMESLICING



Timeslice 1

MULTI-PROCESS TIMESLICING

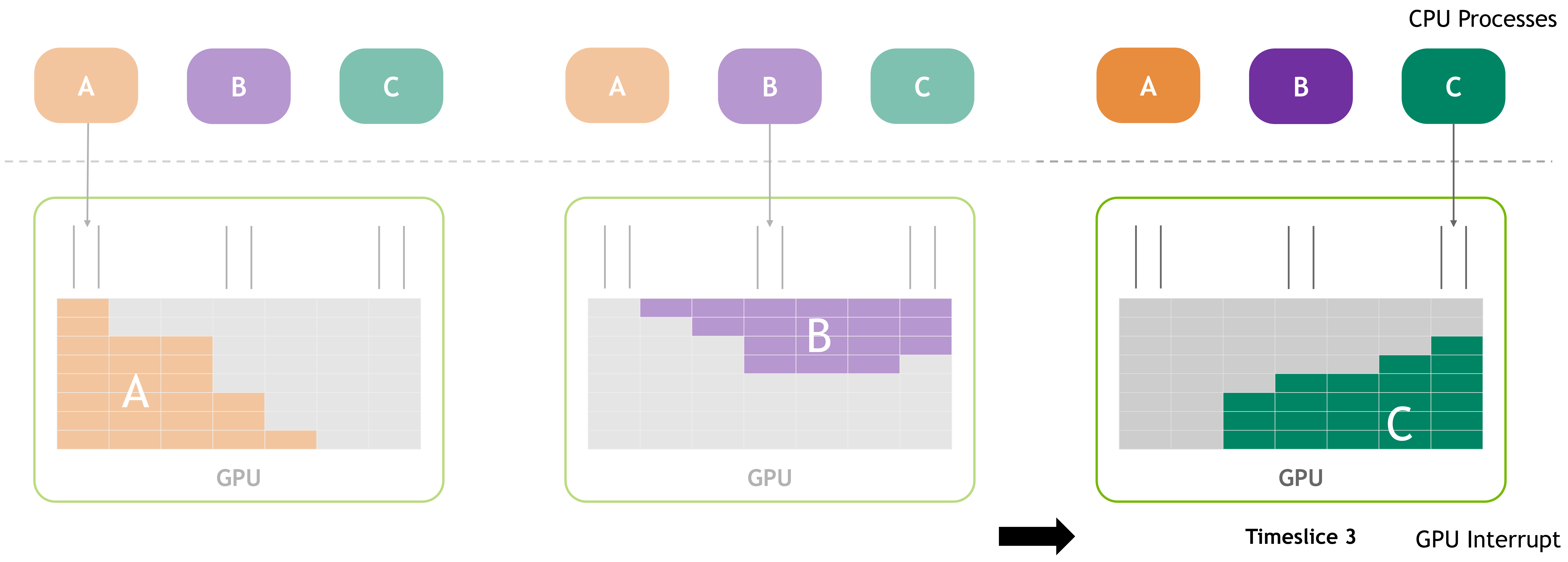


CPU Processes

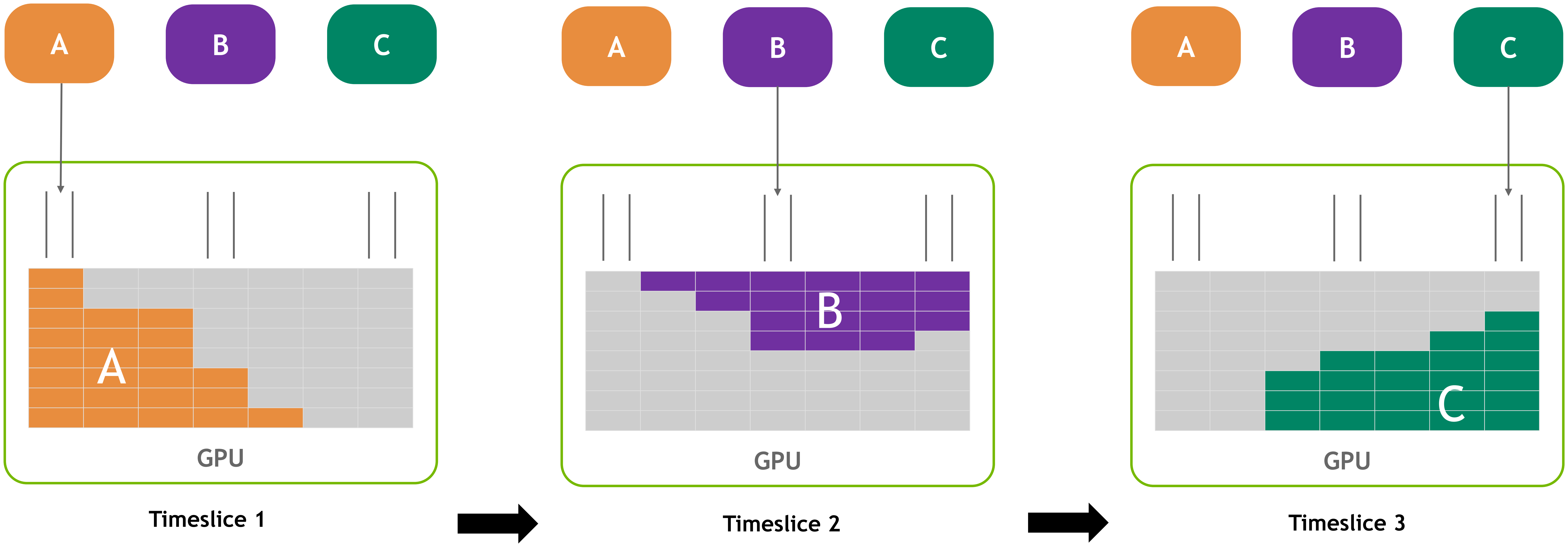
GPU Interrupt

Timeslice 2

MULTI-PROCESS TIMESLICING



MULTI-PROCESS TIMESLICING

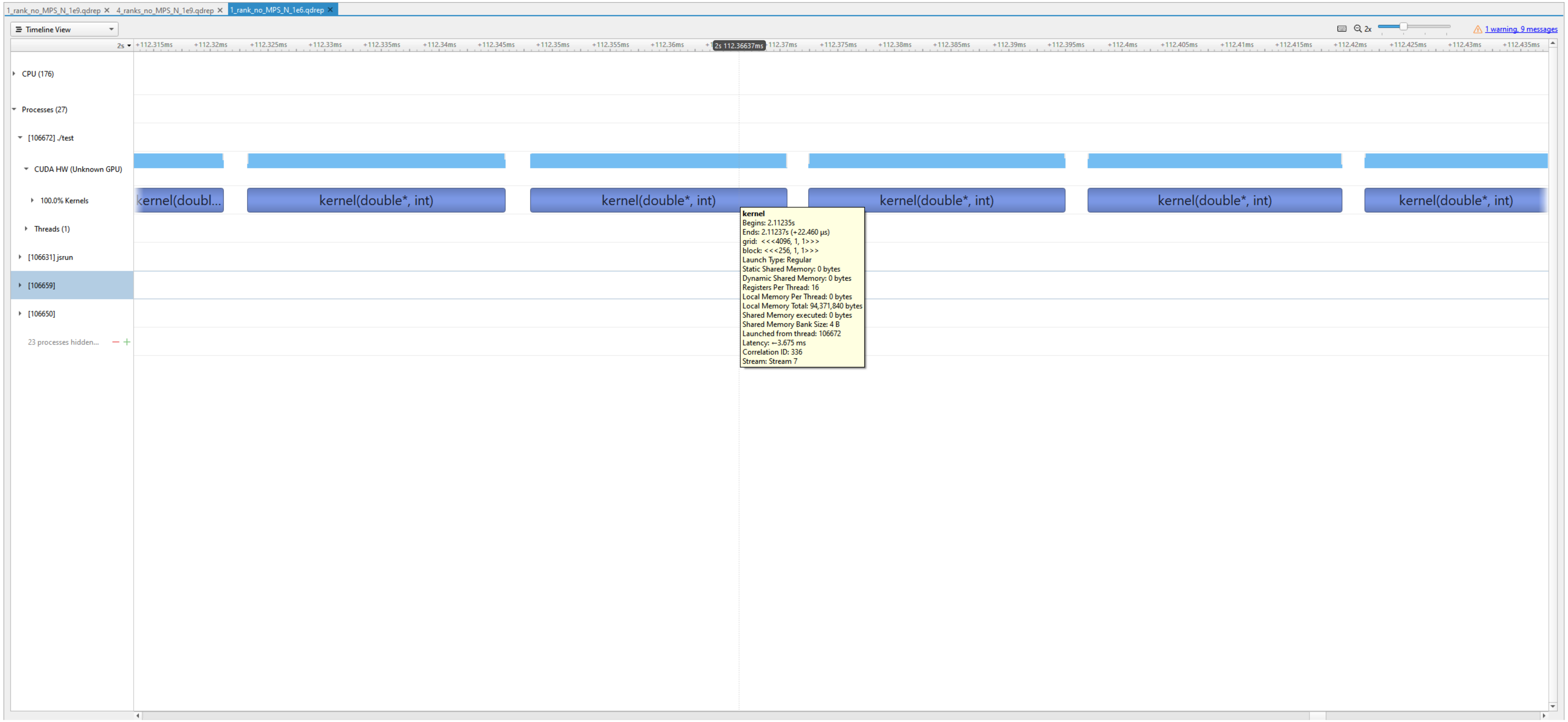


Full process isolation, peak throughput optimized for each process

NVIDIA.

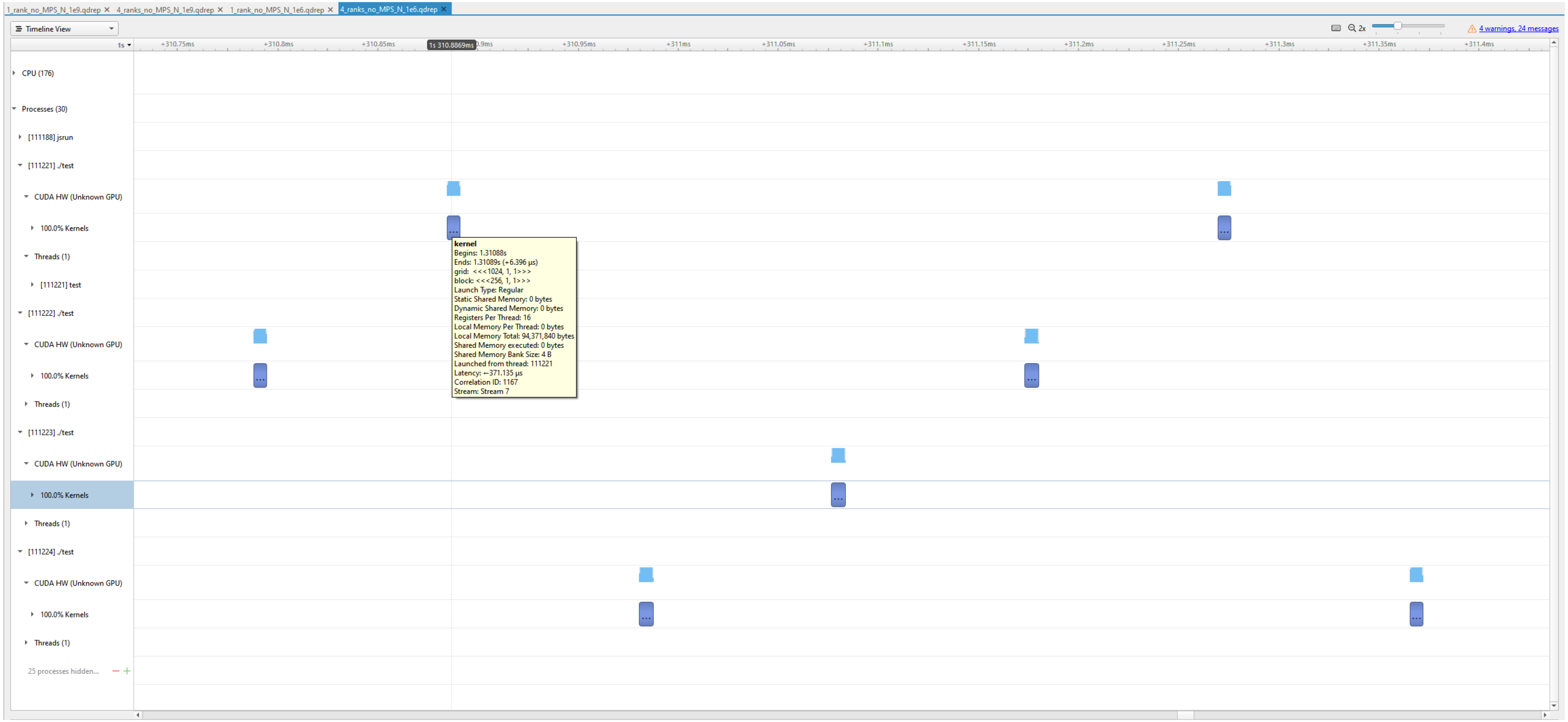
WHEN DOES OVERSUBSCRIPTION HELP?

Perhaps a smaller case where launch latency is relevant? ($N = 10^6$)



WHEN DOES OVERSUBSCRIPTION HELP?

Unfortunately, this isn't better.



OVERSUBSCRIPTION CONCLUSIONS

(when running with the default compute mode)

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

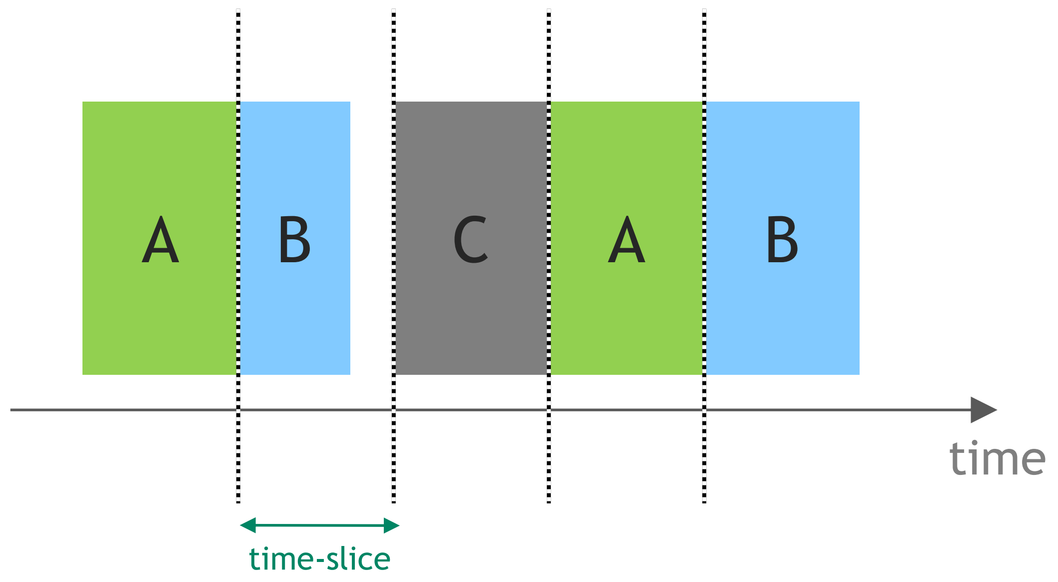
For cases that don’t fully utilize the GPU, we’d like to fill in gaps in the timeline

- 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?

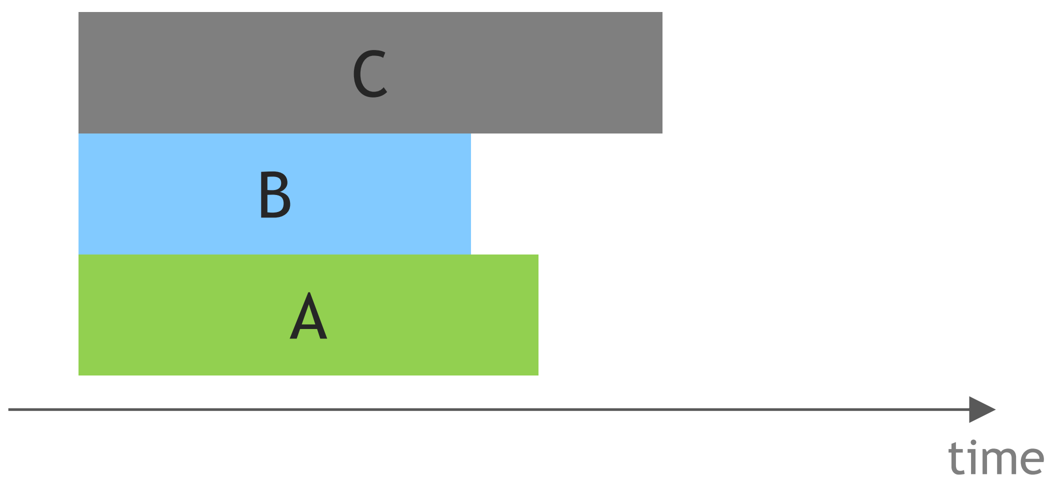
Pre-emptive scheduling

Processes share GPU through time-slicing
Scheduling managed by system



Concurrent scheduling

Processes run on GPU simultaneously
User creates & manages scheduling streams

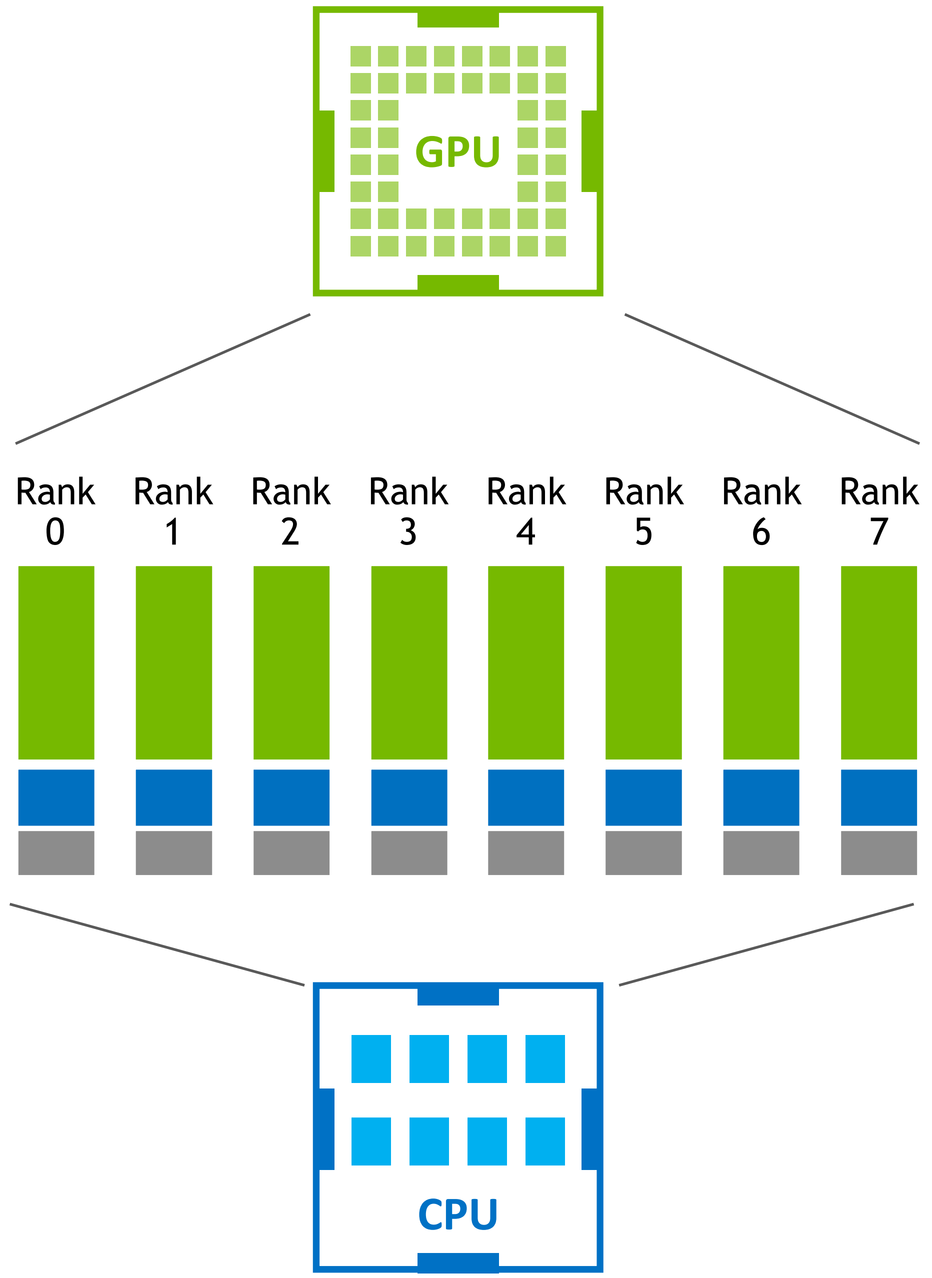


MULTI-PROCESS SERVICE

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

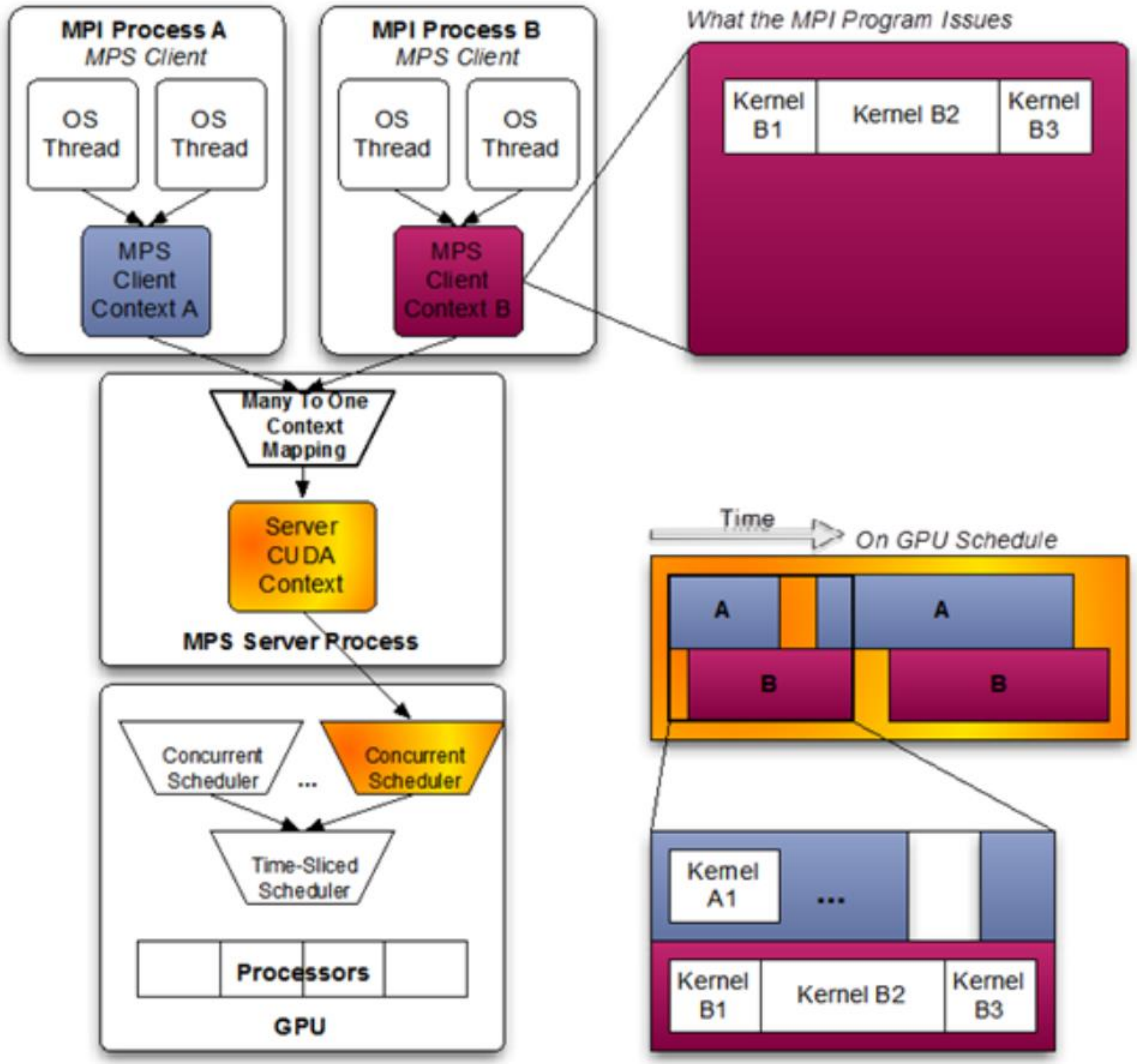
Designed to **concurrently** map multiple MPI ranks onto a single GPU

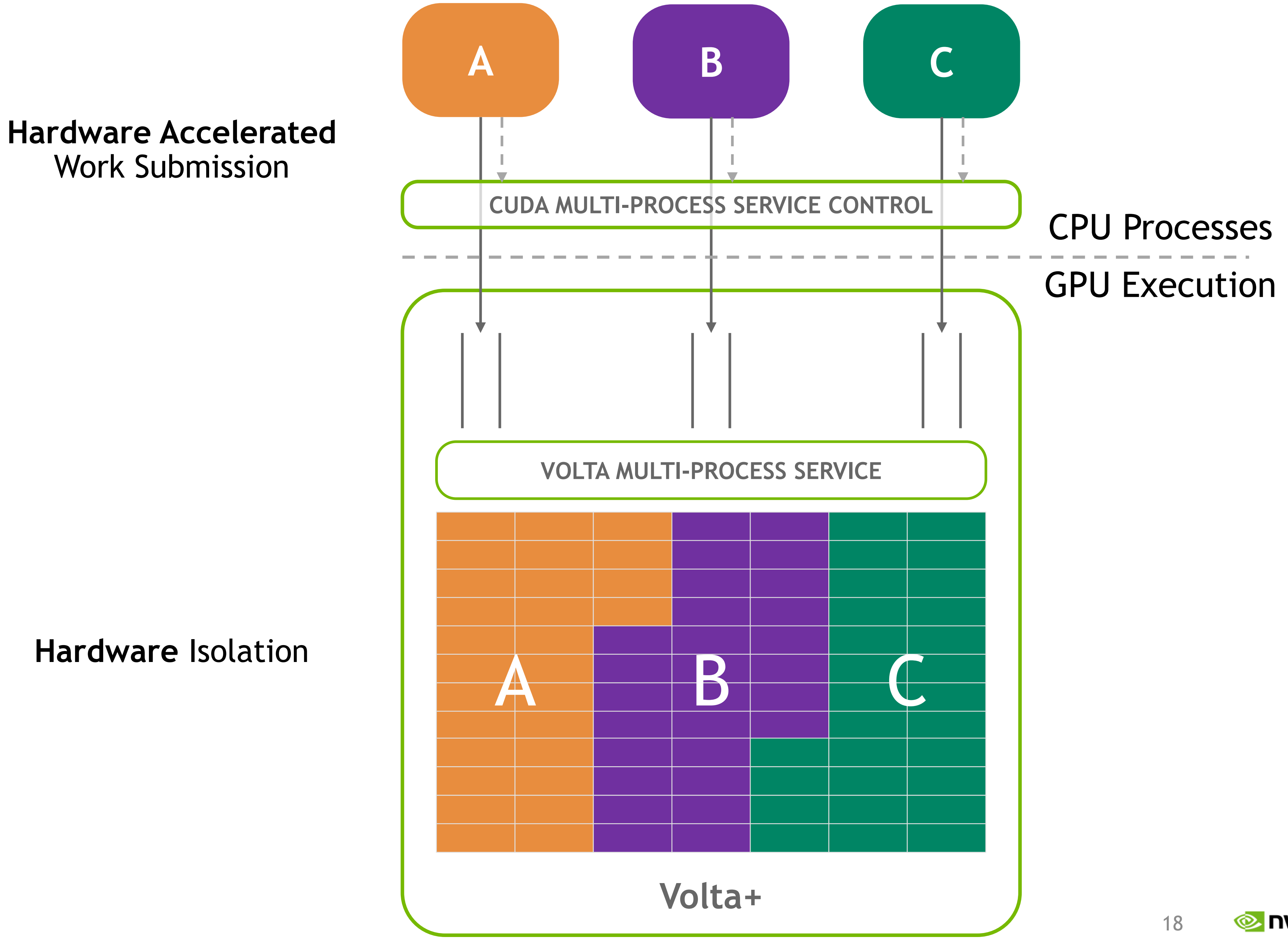
Used when each rank is **too small** to fill the GPU on its own



MULTI-PROCESS SERVICE

Improving on what we had before!



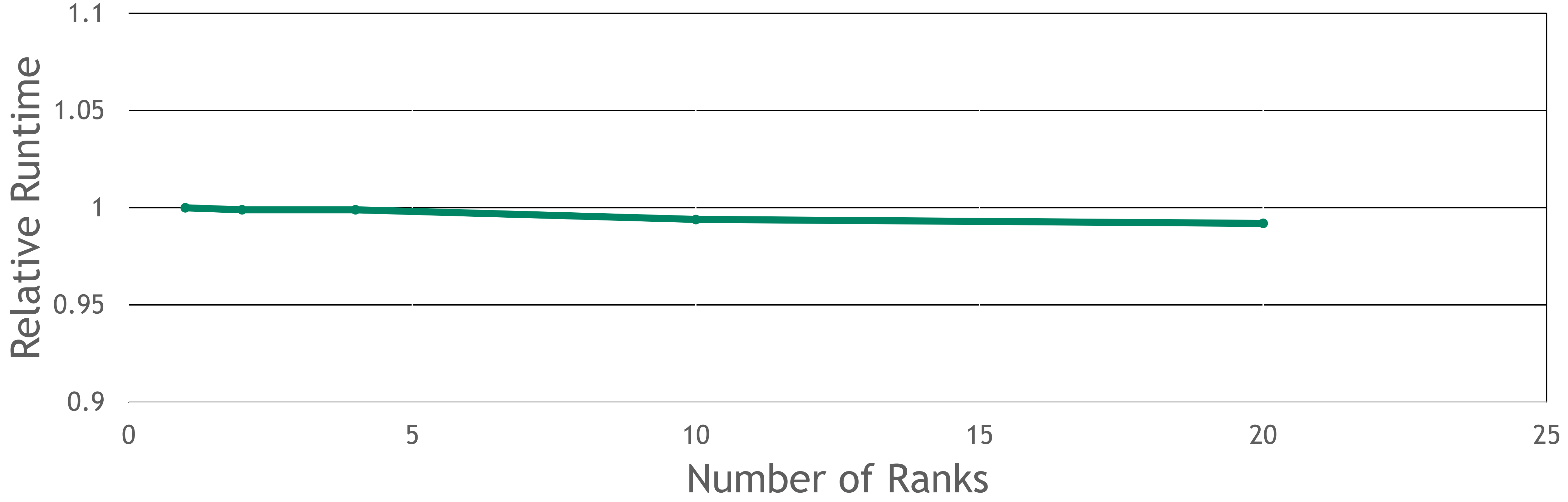


OVERSUBSCRIPTION WITH MPS

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)

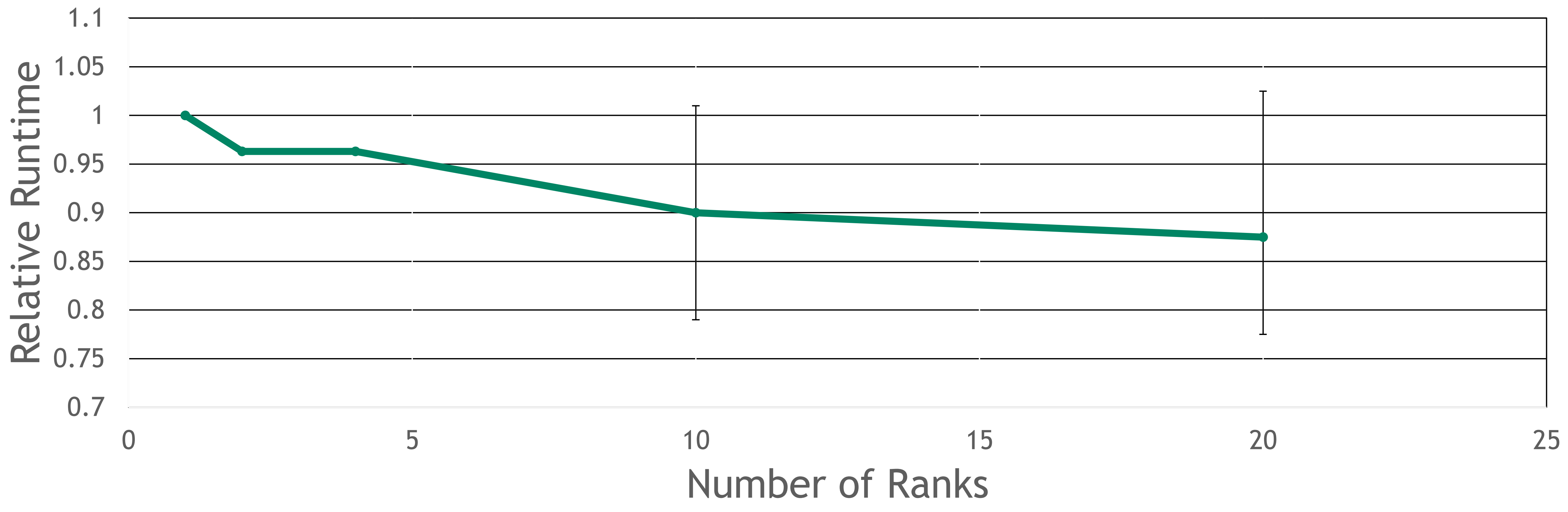


NVIDIA.

OVERSUBSCRIPTION WITH MPS

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

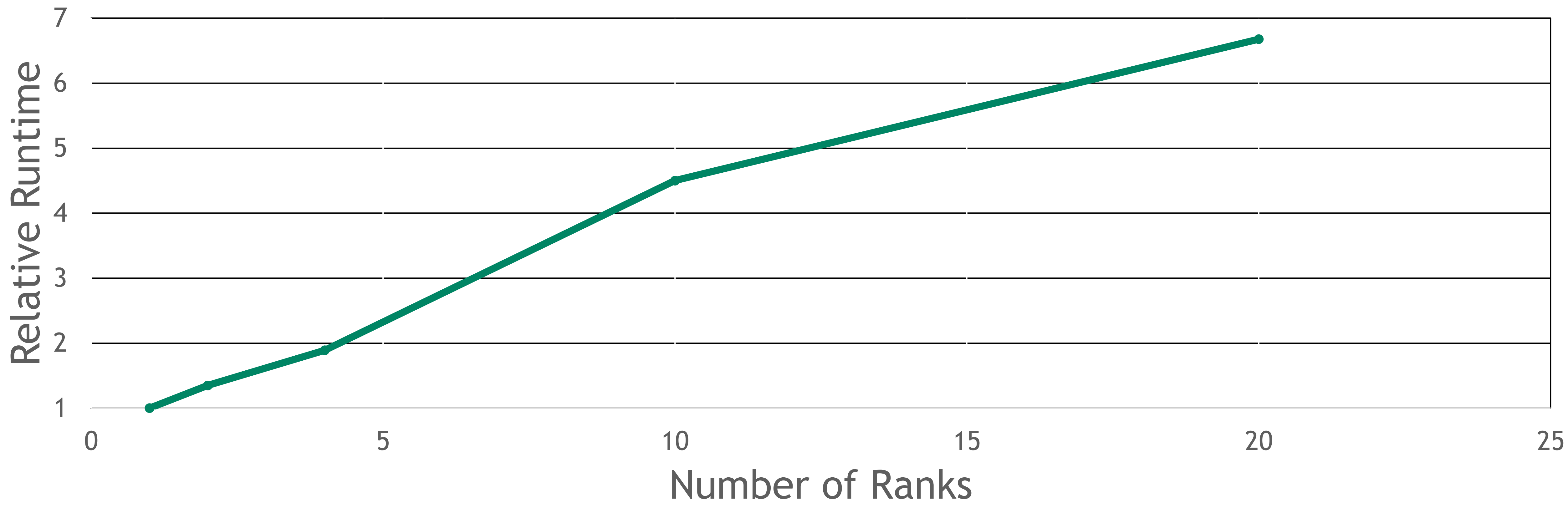
Whether or not there's a speedup depends substantially on precise timing



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

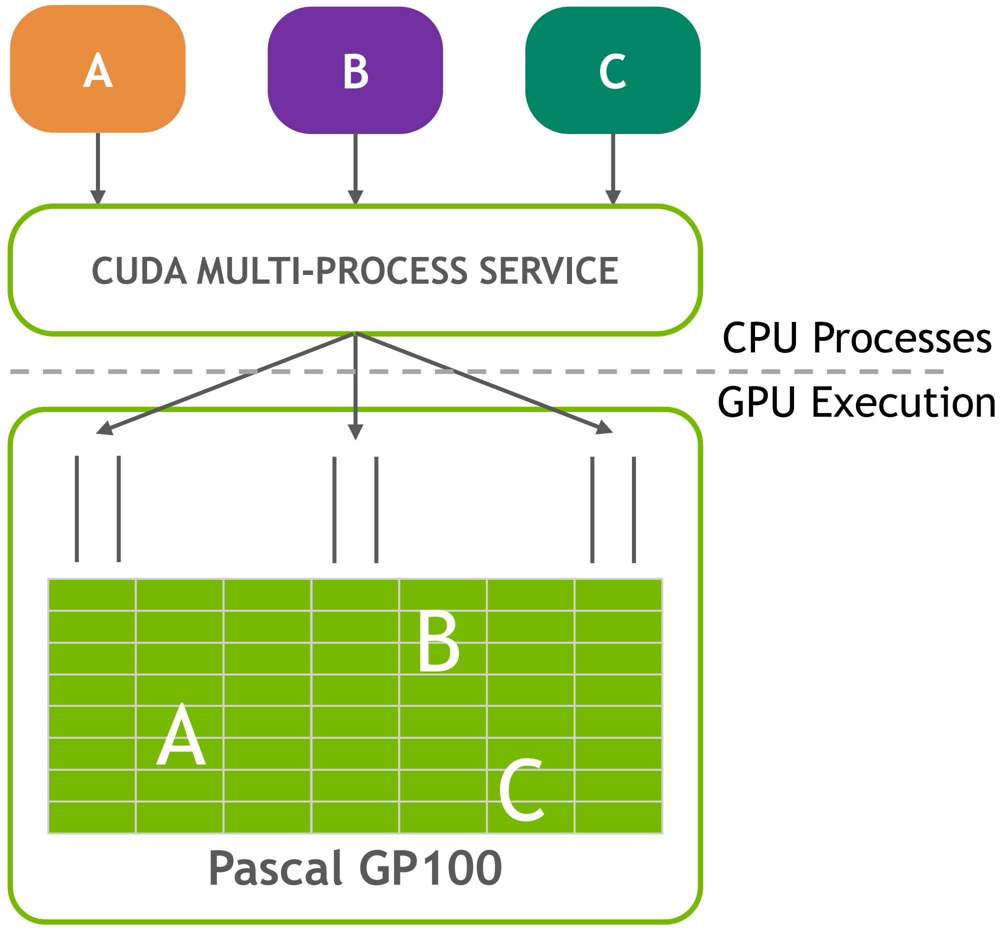
No free lunch theorem still applies: if GPU is fully utilized, cannot get faster answers

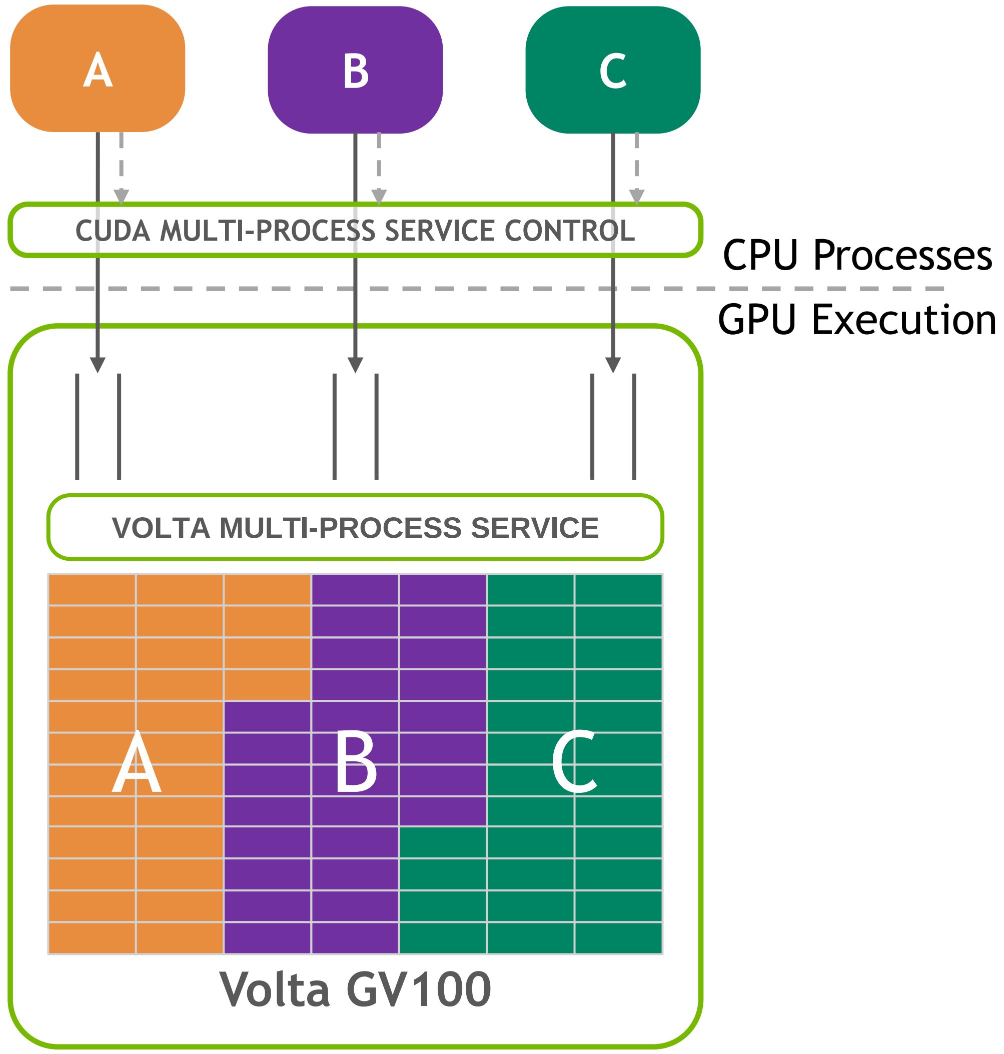
Strive to write your application *so that you don’t need MPS*

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

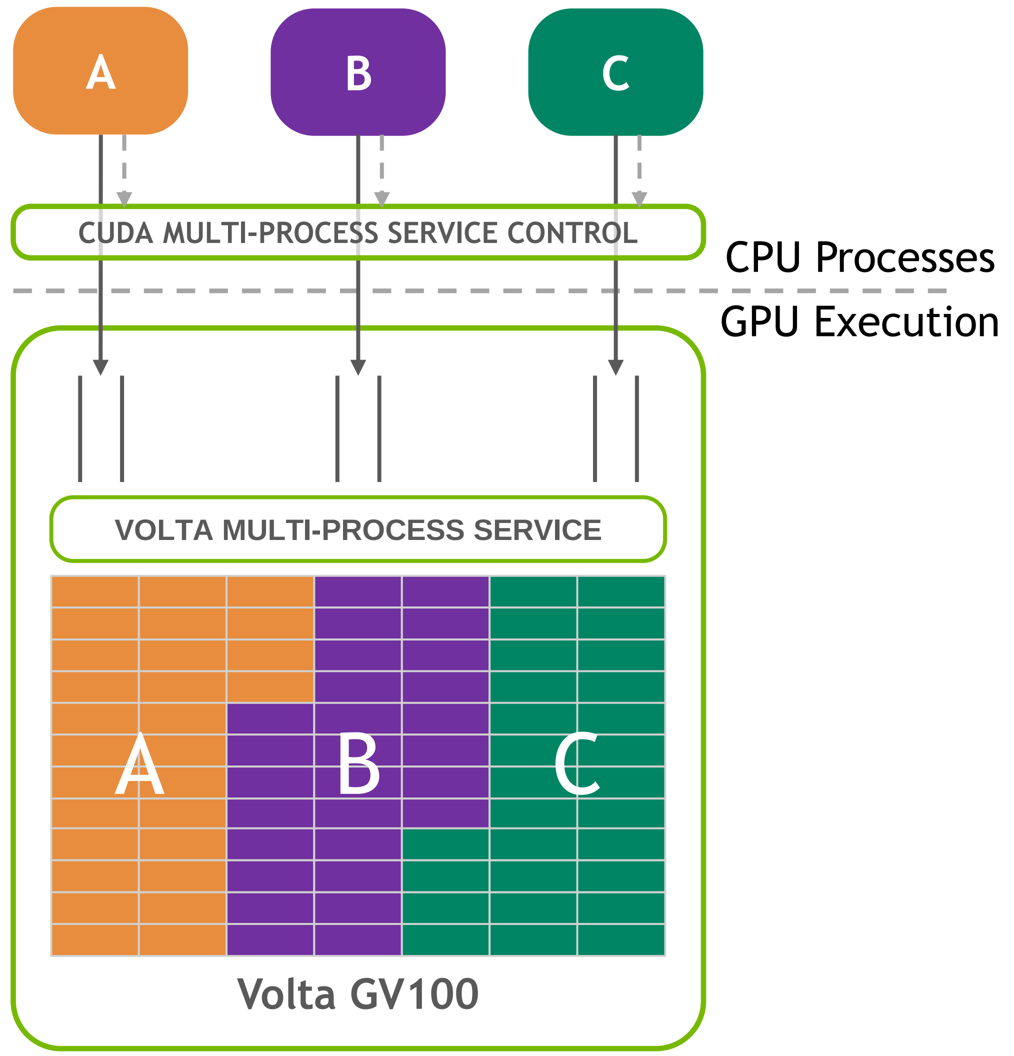
More MPS clients per GPU: 48 instead of 16

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

- **PROHIBITED** (cannot set device)
- **EXCLUSIVE_PROCESS** (single shared device)
- **DEFAULT** (per-process device)

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

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

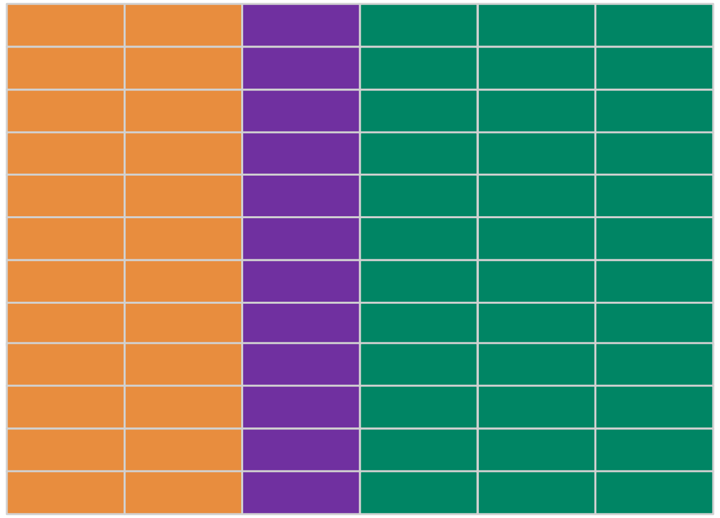
Using MPS, applications can assign fractions of a GPU to each process

A=33%, B=33%, C=33%

Fractional Provisioning

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

Process B is guaranteed space if needed



A=33%, B=33%, C=100%

Using Oversubscription

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

A B C ← 3 concurrent MPS processes

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

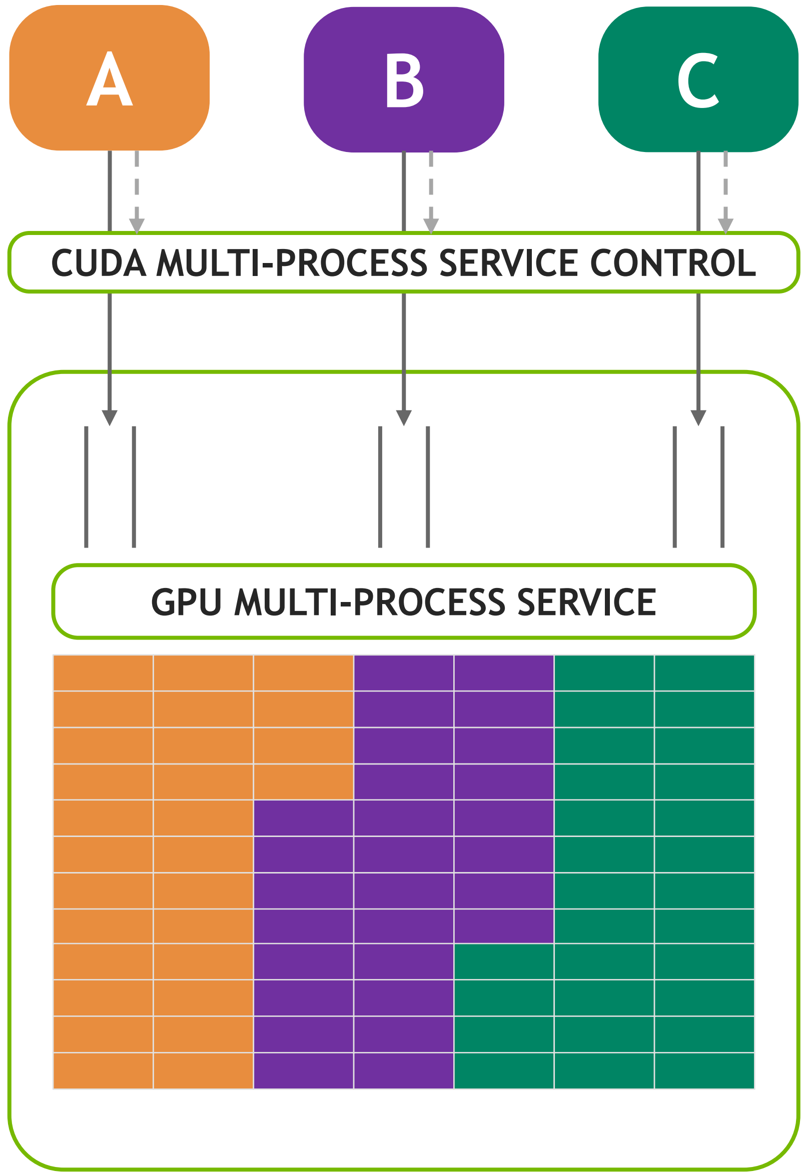
CUDA maps streams onto CUDA_DEVICE_MAX_CONNECTIONS hardware work queues

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

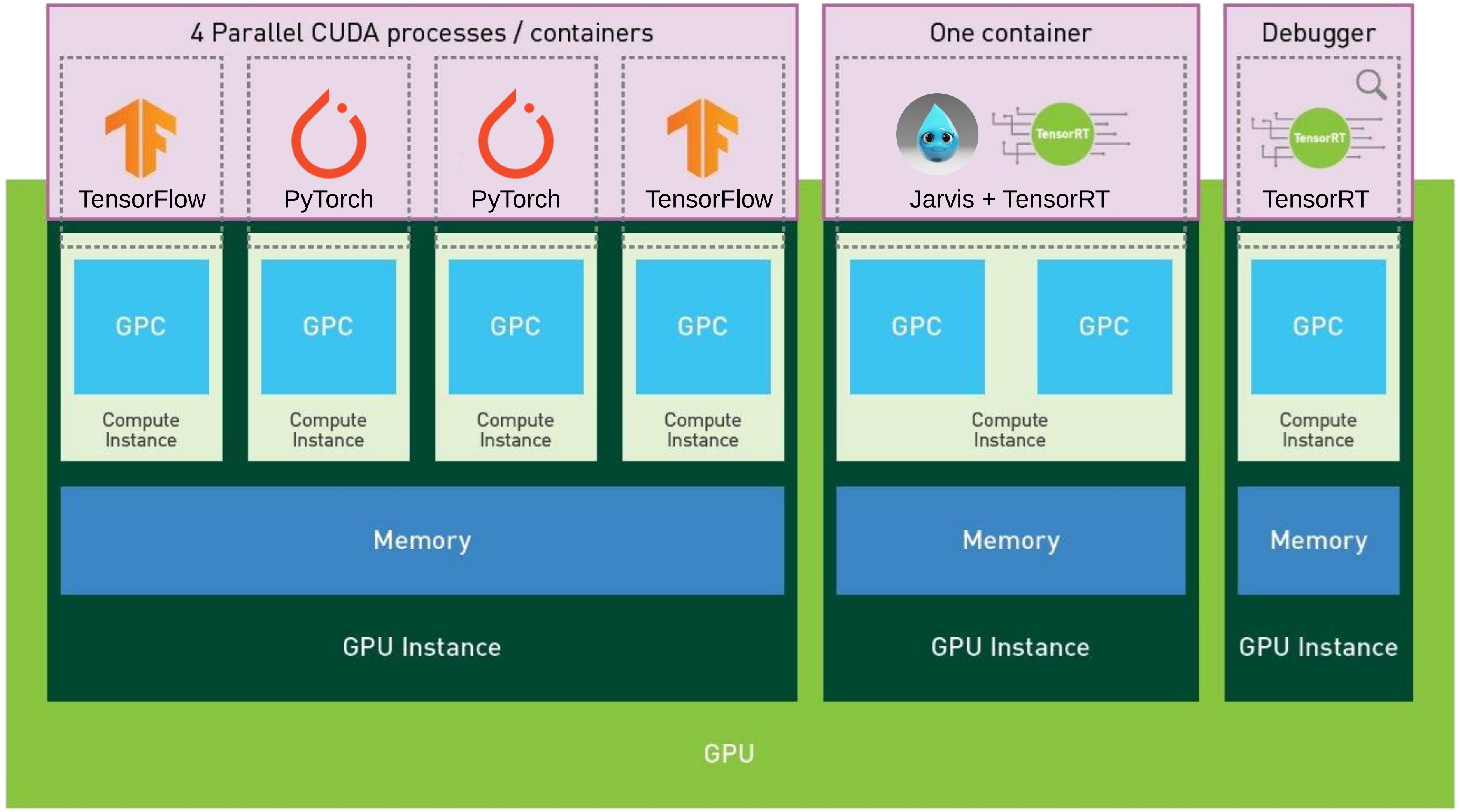
MPS LOGICAL VS. MIG PHYSICAL PARTITIONING



Multi-Process Service

Dynamic contention for GPU resources

Single tenant



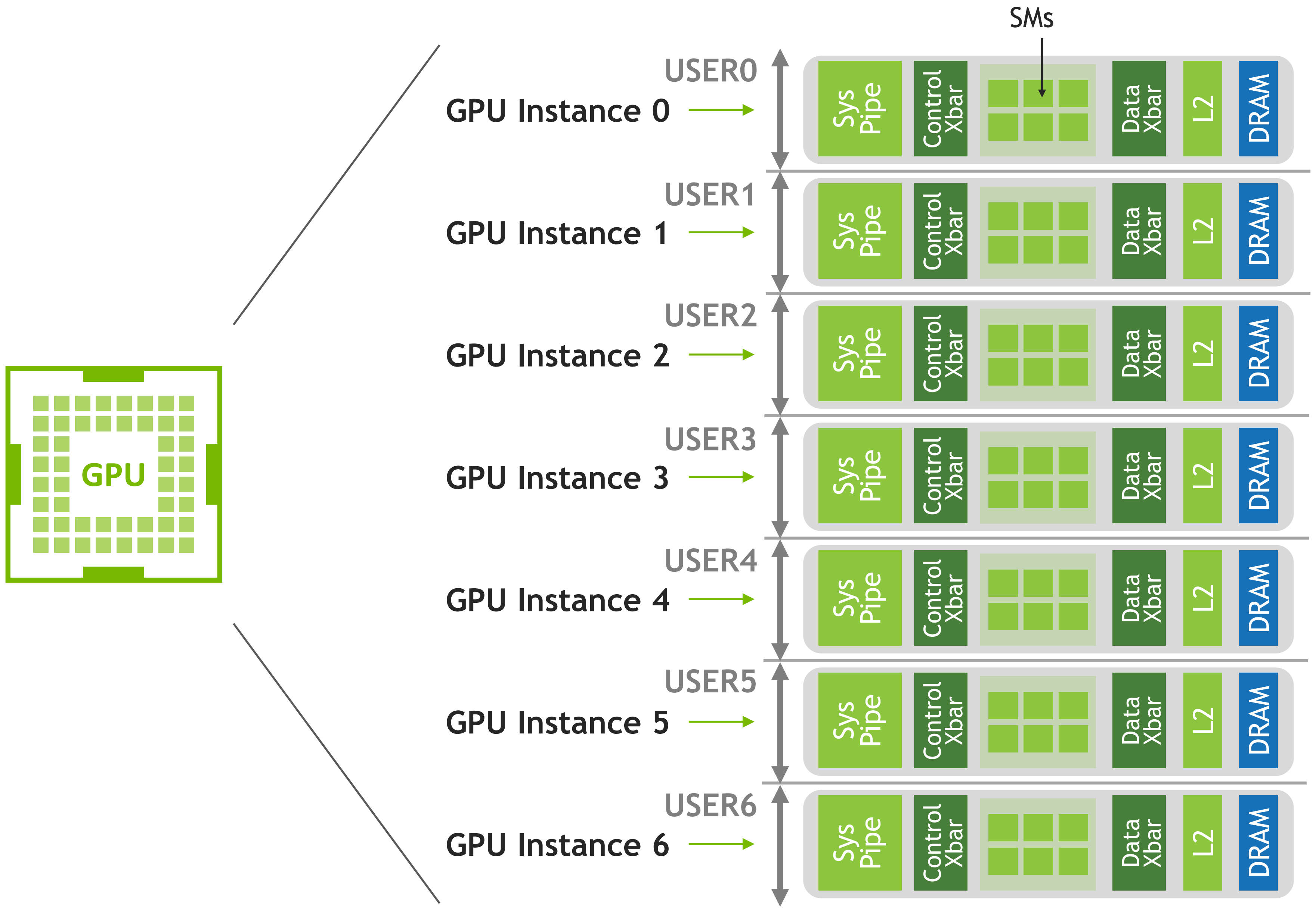
Multi-Instance GPU

Hierarchy of instances with guaranteed resource allocation

Multiple tenants

MULTI-INSTANCE GPU (MIG)

Divide a Single A100 GPU Into Multiple *Instances*, Each With Isolated Paths Through the **Entire** Memory System



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

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

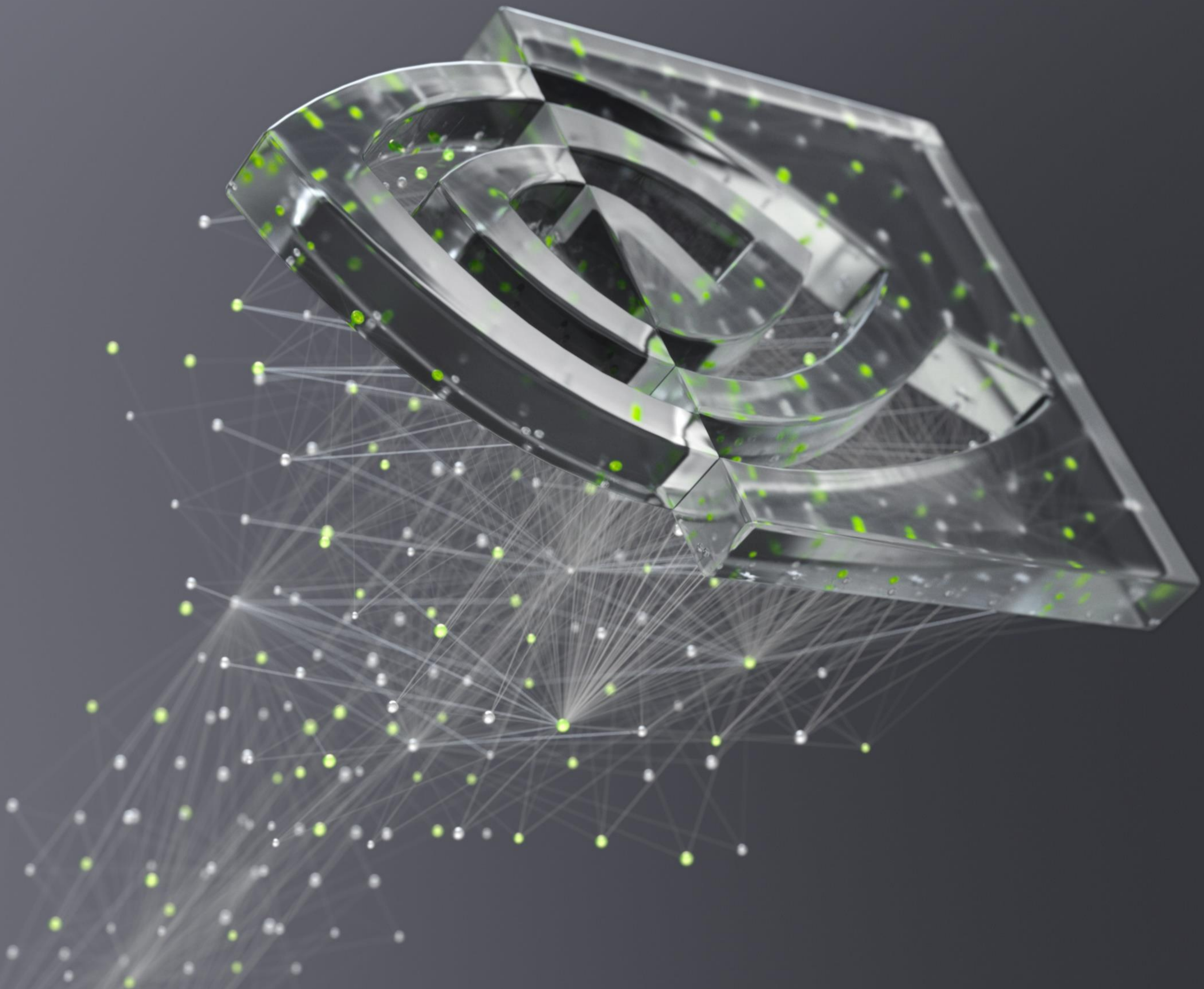
Diverse Deployment Environments

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

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



NVIDIA