SNIA DEVELOPER CONFERENCE
SDC 25
By Developers FOR Developers

Hyatt Regency Santa Clara, CA
September 15-17, 2025

Why does NVMe need to evolve for efficient storage access from GPUs?

Chandra Guda(SMTS), Suresh Rajgopal(DMTS), Pierre Labat(SMTS)
Micron Technology

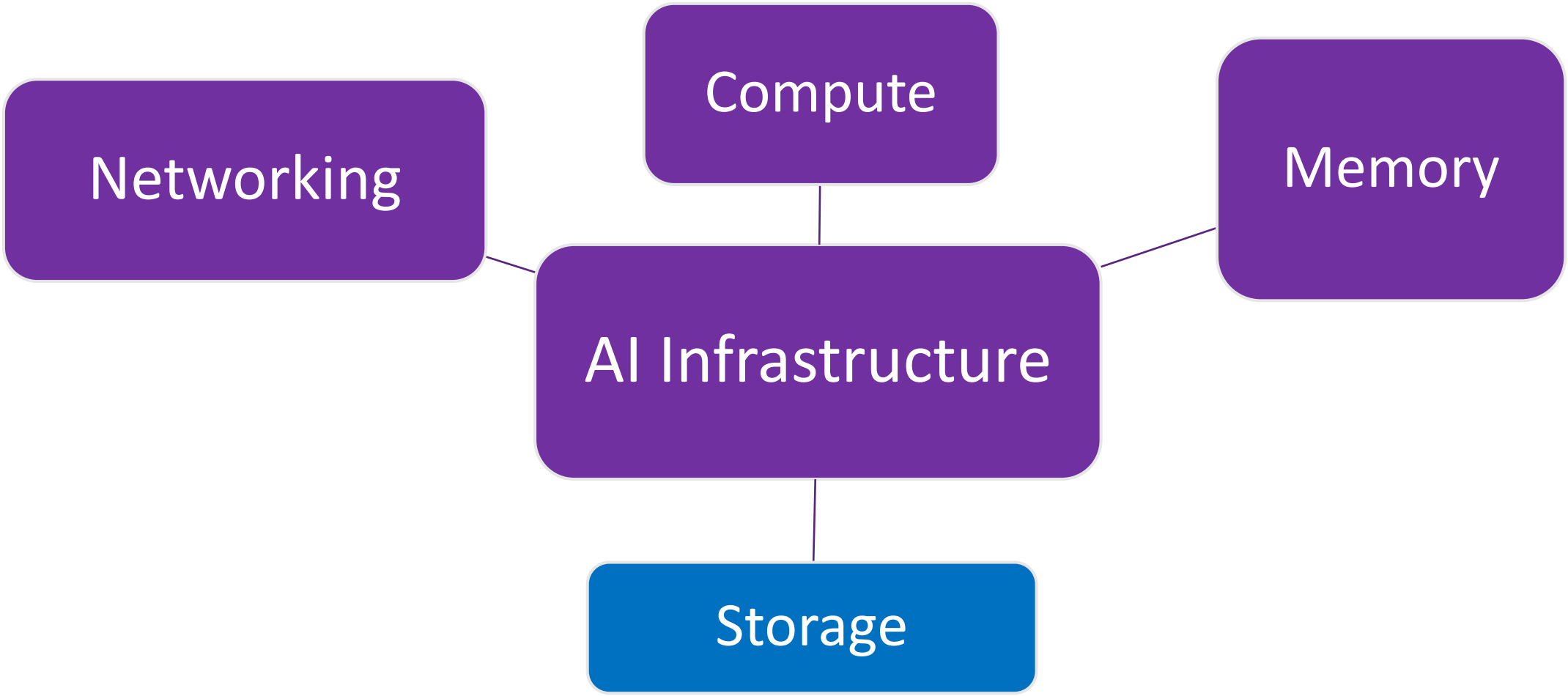
Preface

Traditional NVMe Protocol was designed with CPU cores and hyperthreading in mind. With advent of GPUs and NV-Link/ UA-links speeds reaching TB/s connection to the fastest SSDs which are still in multi-GB/s. Storage bottlenecks need to be addressed where the storage protocol needs to be reviewed.(NVMe Protocol)

micron

SDC 25

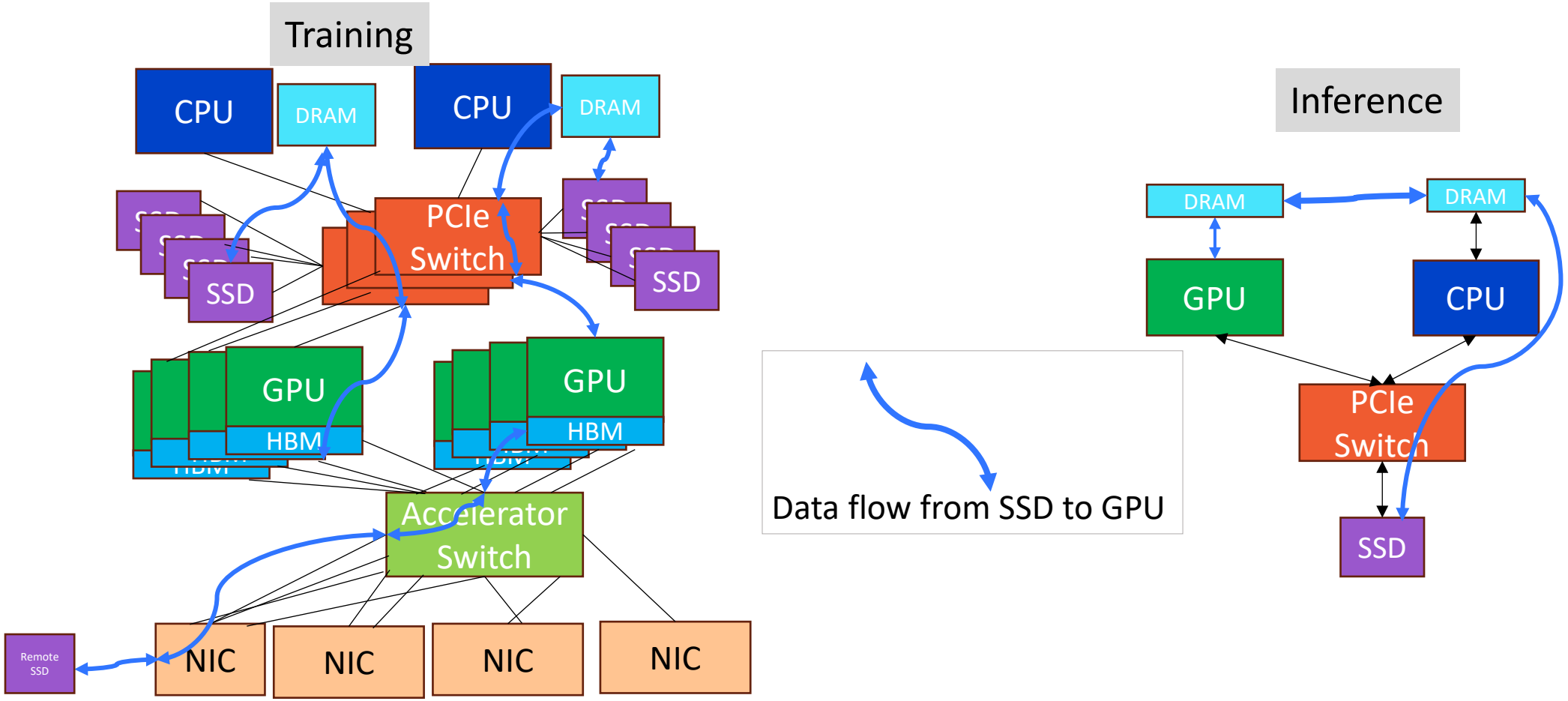
Motivation why are we here - AI



micron

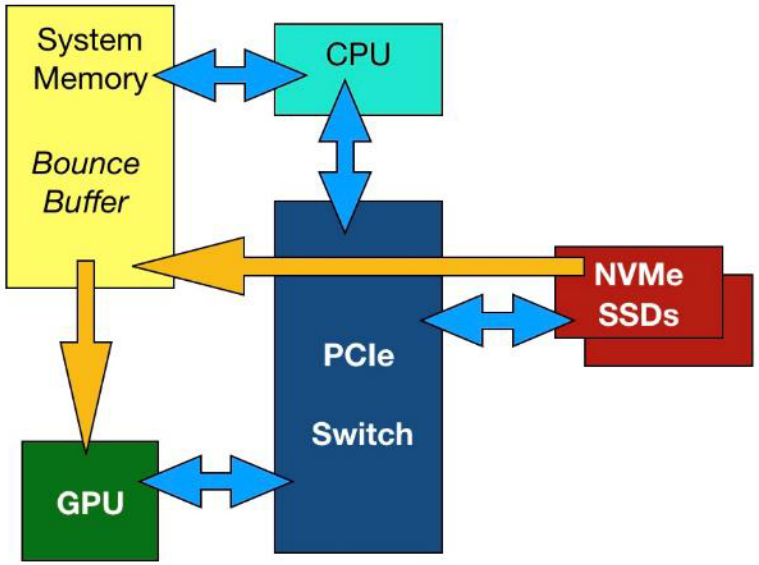
SDC 25

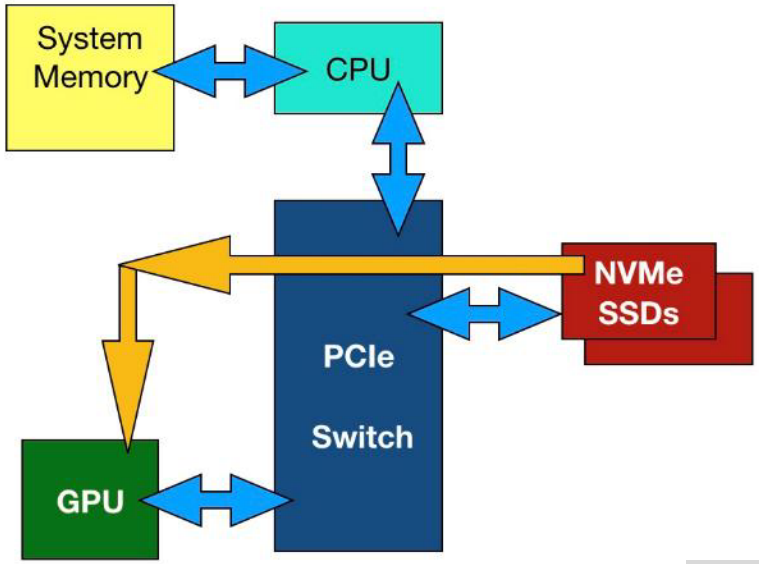
GPUs are at the heart of AI Training and Inference



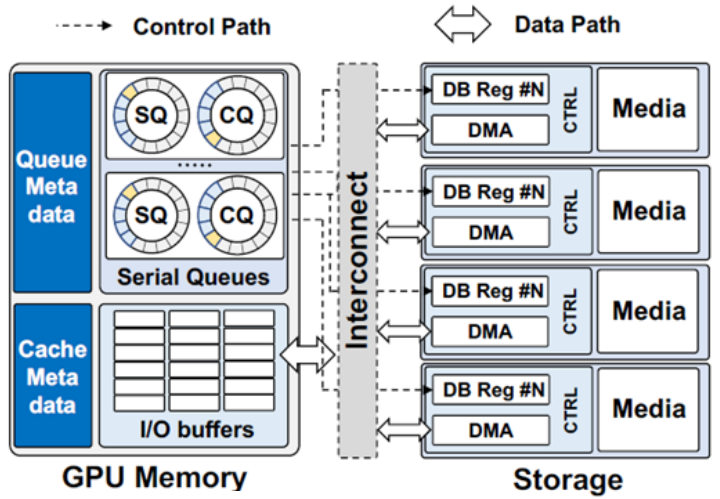
- Whether its training or inference, the GPU needs to be fed with data from storage-near or remote

Making Storage Accessible to GPUs





With and Without peer-to-peer (SSD to GPU) DMA data path



NVMe control path from GPU to SSD

- If data is needed by the GPU, then can the IO be initiated from GPU instead of the CPU?

micron

SDC 25

Quick Introduction to CPU vs GPUs I/O Workloads

CPU vs GPU

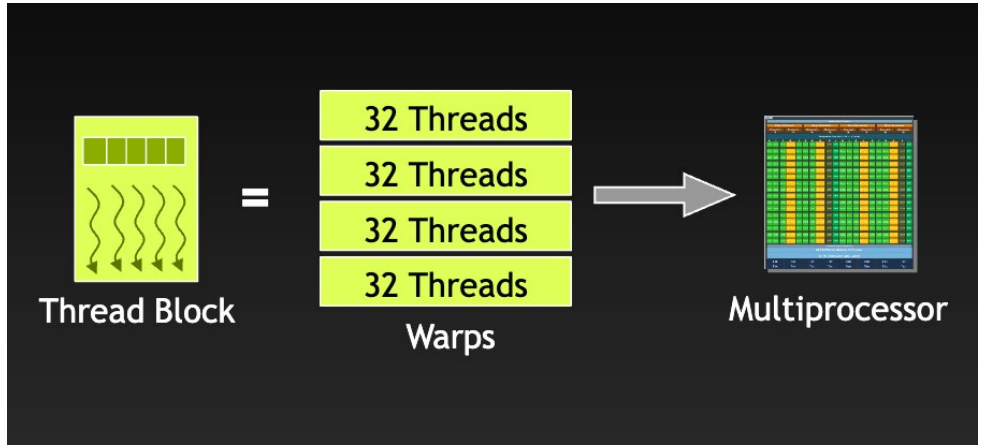
Performance Metrics

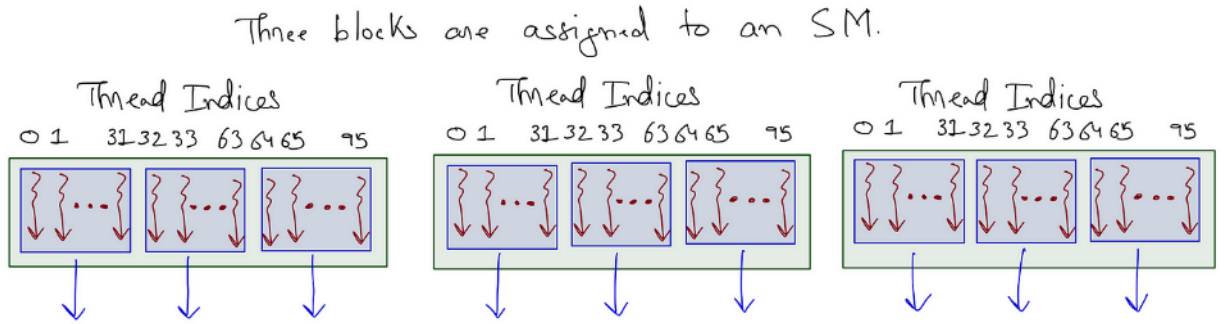
Metric	CPU driven I/O workload	GPU driven I/O workload
Throughput	500MB – 1GB/s	3-14+GB/s
IOPS	Small Block transfers ~1MIOPS	Largely BW focused, smaller blk extremely high IOPS use cases ~100MIOPS
Latency	100us-1ms	Command timeout sensitivity
Utilization Pattern	Continuous	Bursty(per batch)
Use cases	High IOPS, QoS Sensitive applications small reads/writes like Database applications	Training - large chunks Batching, caching , bursty prefill type operations where CPUs can be bypassed

micron

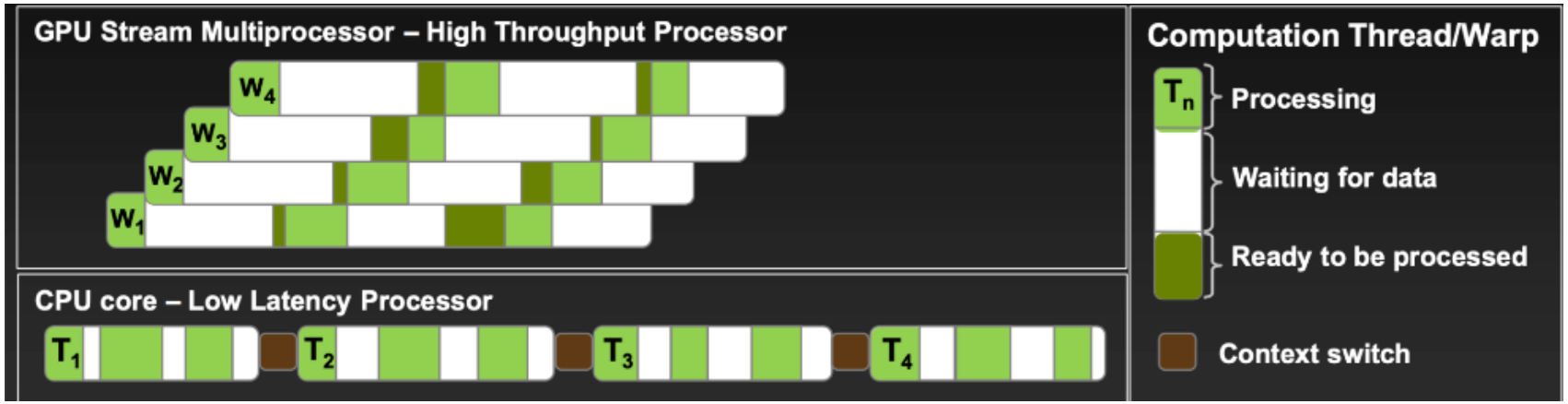
SDC 25

GPU Threads, Warps, SM Core & Pipelining





- Warp scheduling hides latencies
- CPU core scheduling optimizes for latency and QoS



https://wordpress.cels.anl.gov/atpesc/wp-content/uploads/sites/96/2015/08/Sakharnykh_GPGPU_trends.pdf

micron

SDC 25

GPUs SIMT architecture

- GPUs execute groups of threads known as ***warps/waves*** in **SIMT (Single Instruction, Multiple Thread)** fashion.
- Many CUDA programs achieve high performance by taking advantage of warp execution
- In a SIMT architecture, rather than a single thread issuing vector instructions applied to data vectors, **multiple threads issue common instructions to arbitrary data.** These are all tied to a single **Warp.**
- Today GPUs perform a warp of 32 parallel threads using SIMT. This allows each thread to access its own registers to load and store.
- GPU code compilers and GPUs work together to ensure the threads of a warp execute the same instruction sequences together at max performance.
- When these Warps are submitted to NVMe SSDs. They get scattered across several QPs to enable parallel execution and allow GPUs to operate on Next Warp.

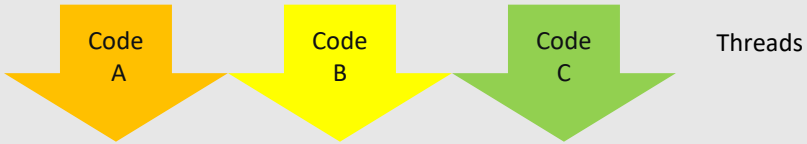
NVMe QPs design needs to align better with GPU scheduling

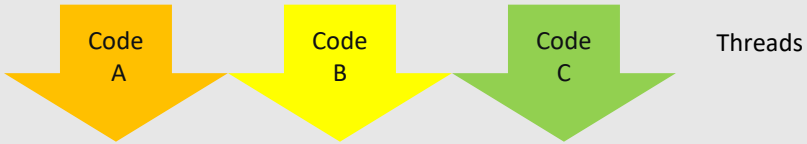
Using CUDA Warp-Level Primitives | NVIDIA Technical Blog

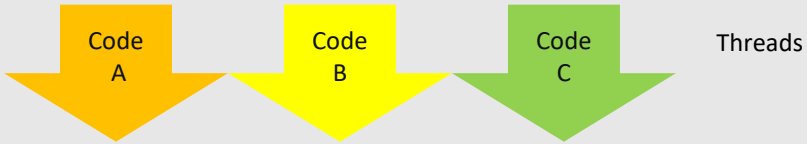
micron

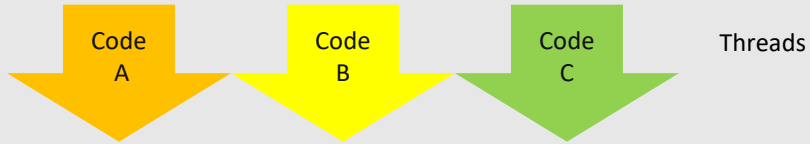
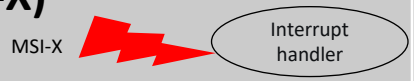
SDC 25

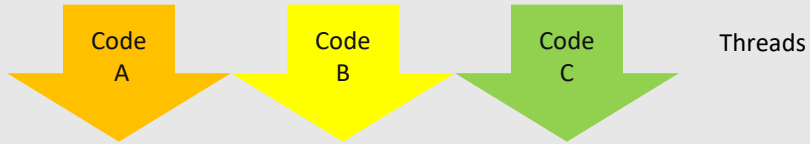
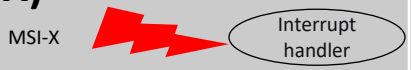
CPU	GPU
Number threads accessing a Nvme Q at time t = 1 One Q assigned to one CPU only, no pre-emption when in critical section of code accessing the queue => only one thread can access the queue. 1 Nvme Q per CPU is enough to achieve zero contention.	**Number threads potentially accessing a same Nvme Q at time t = O(100000)** Thousands of threads can run in // per SM/CU and they can be pre-empted at any time.

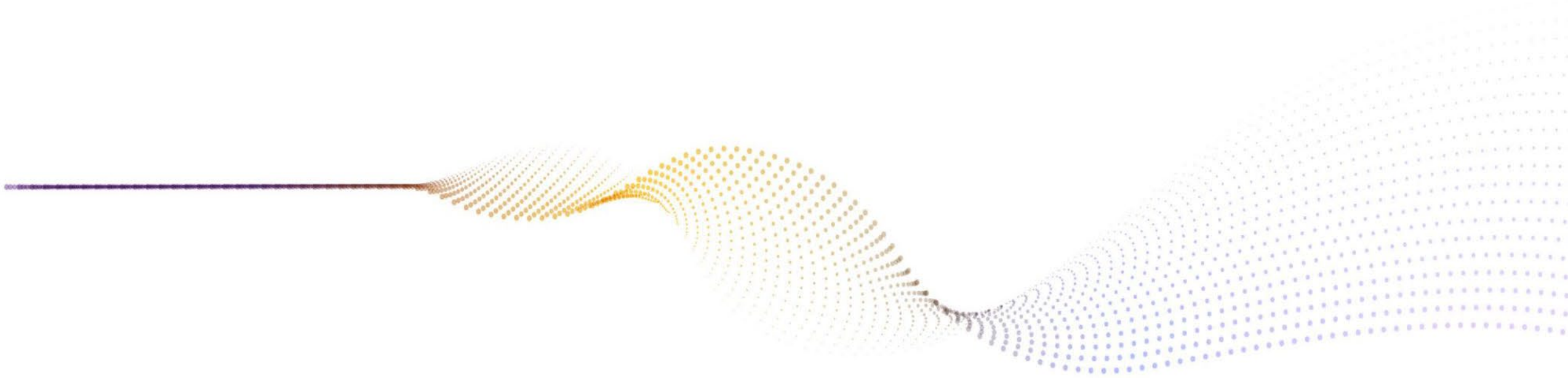
CPU	GPU
Number threads accessing a Nvme Q at time t = 1 One CPU only assigned to a Q, no pre-emption when in critical section of code accessing the queue => only one thread can access the queue	**Number threads potentially accessing a Nvme Q at time t = O(100000)** Thousands of threads can run in // per SM/CU and they can be pre-empted at any time.
No downside with threads running different code. 	**Optimized/very efficient for threads running same code.** 

CPU	GPU
Number threads accessing a Nvme Q at time t = 1 One CPU only assigned to a Q, no pre-emption when in critical section of code accessing the queue => only one thread can access the queue	**Number threads potentially accessing a Nvme Q at time t = O(100000)** Thousands of threads can run in // per SM/CU and they can be pre-empted at any time.
No downside with threads running different code. 	**Optimized/very efficient for threads running same code.** 
Poor at handling memory access latency CPU/core stalls wasting compute Hyperthreading when avail is of very limited help (toggling between 2 threads / core)	**Excellent at handling memory access latency** GPU doesn't stall but schedule other threads at very large scale

CPU	GPU
Number threads accessing a Nvme Q at time t = 1 One CPU only assigned to a Q, no pre-emption when in critical section of code accessing the queue => only one thread can access the queue	**Number threads potentially accessing a Nvme Q at time t = O(100000)** Thousands of threads can run in // per SM/CU and they can be pre-empted at any time.
No downside with threads running different code. 	**Optimized/very efficient for threads running same code.** 
Poor at handling memory access latency Cpu/core stalls wasting compute Hyperthreading when avail is of very limited help (toggling between 2 threads / core)	**Excellent at handling memory access latency** GPU doesn't stall but schedule other threads at very large scale
User can assign threads to CPU	**User can't assign threads to GPU SM/CU, assignment decided by hardware only**

CPU	GPU
Number threads accessing a Nvme Q at time t = 1 One CPU only assigned to a Q, no pre-emption when in critical section of code accessing the queue => only one thread can access the queue	**Number threads potentially accessing a Nvme Q at time t = O(100000)** Thousands of threads can run in // per SM/CU and they can be pre-empted at any time.
No downside with threads running different code. 	**Optimized/very efficient for threads running same code.** 
Poor at handling memory access latency Cpu/core stalls wasting compute Hyperthreading when avail is of very limited help (toggling between 2 threads / core)	**Excellent at handling memory access latency** GPU doesn't stall but schedule other threads at very large scale
User can assign threads to CPU	**User can't assign threads to GPU SM/CU, assignment decided by hardware only**
Programming can use Interrupts (MSI-X) 	**No interrupts available for programming** GPUs are designed for **massively parallel computation**, not for **event-driven control.** No hardware nor software available to handle interrupt handlers.

CPU	GPU
Number threads accessing a Nvme Q at time t = 1 One CPU only assigned to a Q, no pre-emption when in critical section of code accessing the queue => only one thread can access the queue	**Number threads potentially accessing a Nvme Q at time t = O(100000)** Thousands of threads can run in // per SM/CU and they can be pre-empted at any time.
No downside with threads running different code. 	**Optimized/very efficient for threads running same code.** 
Poor at handling memory access latency Cpu/core stalls wasting compute Hyperthreading when avail is of very limited help (toggling between 2 threads / core)	**Excellent at handling memory access latency** GPU doesn't stall but schedule other threads at very large scale
User can assign threads to CPU	**User can't assign threads to GPU SM/CU, assignment decided by hardware only**
Programming can use Interrupts (MSI-X) 	**No interrupts available for programming** GPUs are designed for **massively parallel computation**, not for **event-driven control.** No hardware nor software available to handle interrupt handlers.
Any threads has rich OS services at hand (memory allocation for example)	**On GPU typical host services are not available. The only services available are GPU specific (thread synchronization for example).**



Challenges in NVMe
– Thread Sync in NVMe hurts GPU utilization

micron

SDC 25

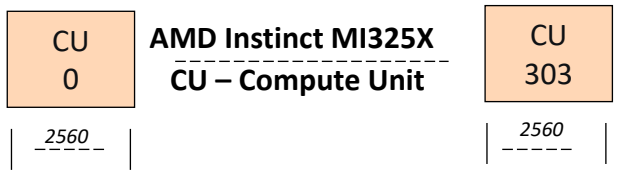
Thread Synchronization in NVMe hurts GPU utilization

Nvidia H200

SM - Streaming Multiprocessor

132*2048 = **~270K** threads in 1 GPU
can potentially access NVMe QPs in parallel

Many more threads in a GPU than CPU

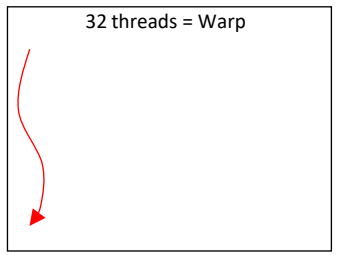


AMD Instinct MI325X

CU – Compute Unit

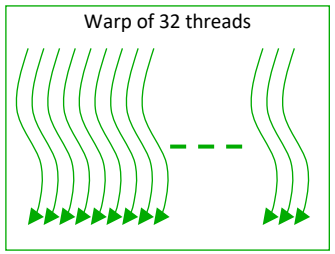
778K threads on 1 AMD GPU

When a thread is in sync code, SM can run only that thread at each cycle



Today

Ideally the SM should run 32 threads at each cycle

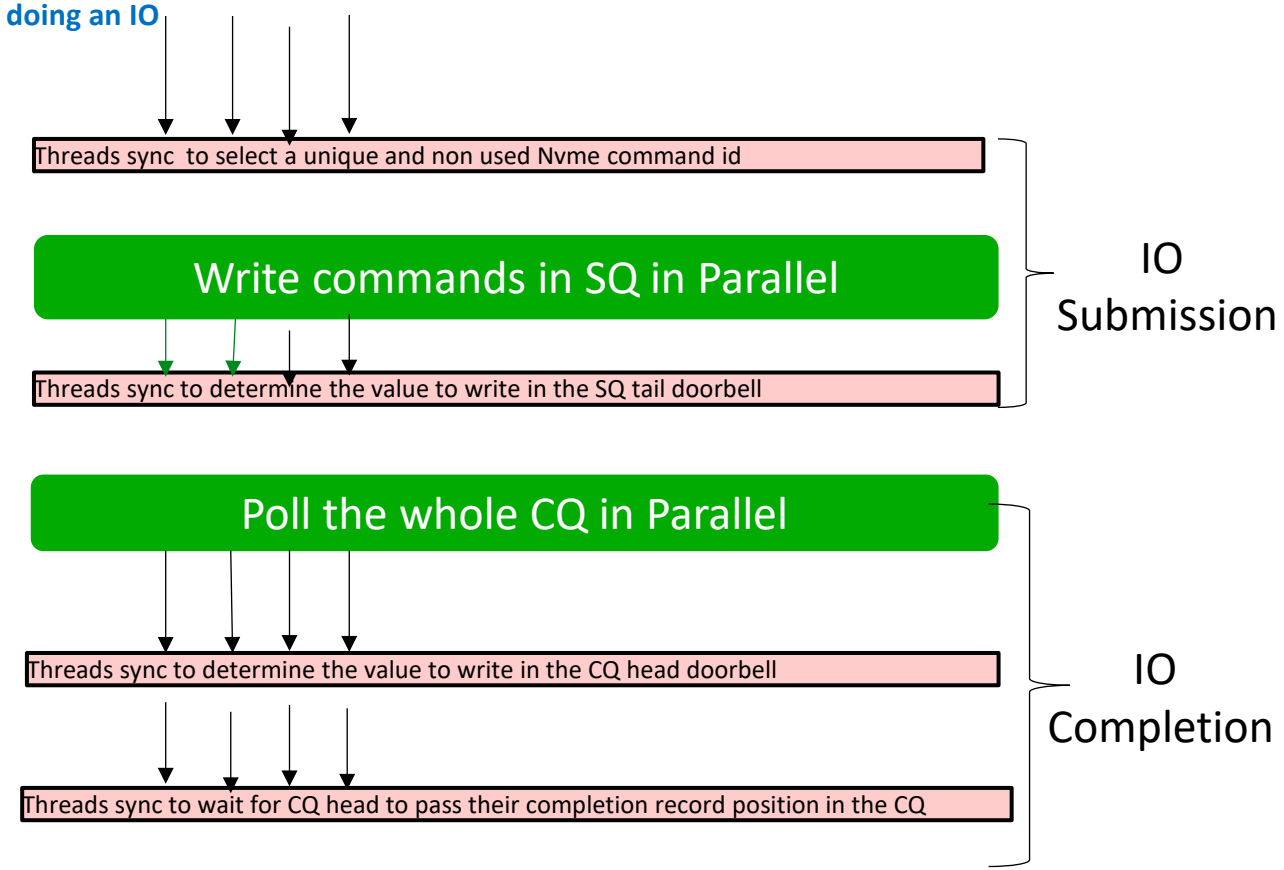


Future

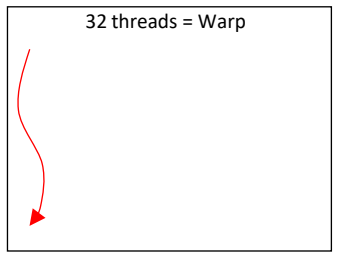
Thread Synchronization in NVMe hurts GPU utilization

Multiple Synchronization points during IO Submission and Completion

All threads accessing the same QP go through several synchronization loops (atomic operation+rescheduling, shown in red below) when doing an IO

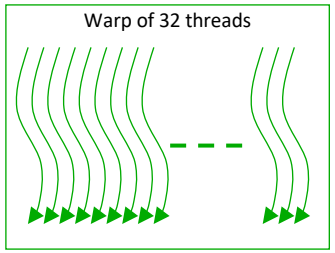


When a thread is in sync code, SM can run only that thread at each cycle



Today

Ideally the SM should run 32 threads at each cycle



Future

Thread Synchronization in NVMe hurts GPU utilization

Multiple Synchronization points during IO Submission and Completion

All threads accessing the same QP go through several synchronization loops (atomic operation+rescheduling, shown in red below) when doing an IO

Synchronization robs SM resources

Synchronizations handled via Loops sleep/ update GPU memory atomic variables.

- Loss of SM cycles due to thread serialization
- Impacts SM L1 Cache bandwidth
- Excessive re-scheduling of threads (each loop)
- Memory used to synchronize 16*(QP size) per QP + 256KB per SSD

Threads sync to select a unique and non used Nvme command id

Write commands in SQ in Parallel

Threads sync to determine the value to write in the SQ tail doorbell

IO Submission

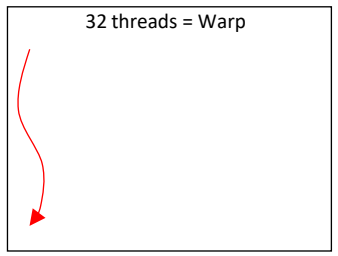
Poll the whole CQ in Parallel

Threads sync to determine the value to write in the CQ head doorbell

Threads sync to wait for CQ head to pass their completion record position in the CQ

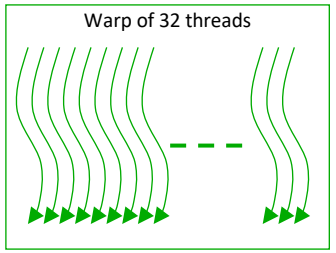
IO Completion

When a thread is in sync code, SM can run only that thread at each cycle



Today

Ideally the SM should run 32 threads at each cycle



Future

micron

SDC 25

Thread Synchronization in NVMe hurts GPU utilization

SM 0 | SM 131 — 2048 | 2048

Nvidia H200

SM - Streaming Multiprocessor

132*2048 = **~270K** threads in 1 GPU can potentially access NVMe QPs in parallel

Many more threads in a GPU than CPU

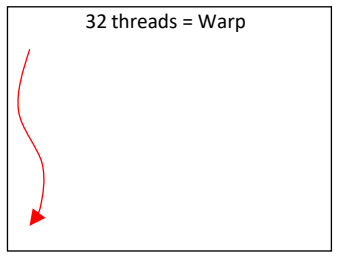
CU 0 | CU 303 — 2560 | 2560

AMD Instinct MI325X

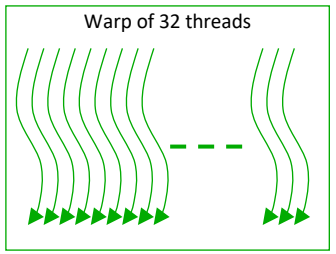
CU – Compute Unit

778K threads on 1 AMD GPU

When a thread is in sync code, SM can run only that thread at each cycle



Ideally the SM should run 32 threads at each cycle



Multiple Synchronization points during IO Submission and Completion

All threads accessing the same QP go through several synchronizationloops (atomic operation+rescheduling, shown in red below) whendoing an IO

Synchronization robs SM resources

Synchronizations handled via Loops sleep/ update GPU memory atomic variables.

- Loss of SM cycles due to thread serialization
- Impacts SM L1 Cache bandwidth
- Excessive re-scheduling of threads (each loop)
- Memory used to synchronize 16*(QP size) per QP + 256KB per SSD

Threads sync to select a unique and non used Nvme command id

Write commands in SQ in Parallel

Threads sync to determine the value to write in the SQ tail doorbell

IO Submission

Poll the whole CQ in Parallel

Threads sync to determine the value to write in the CQ head doorbell

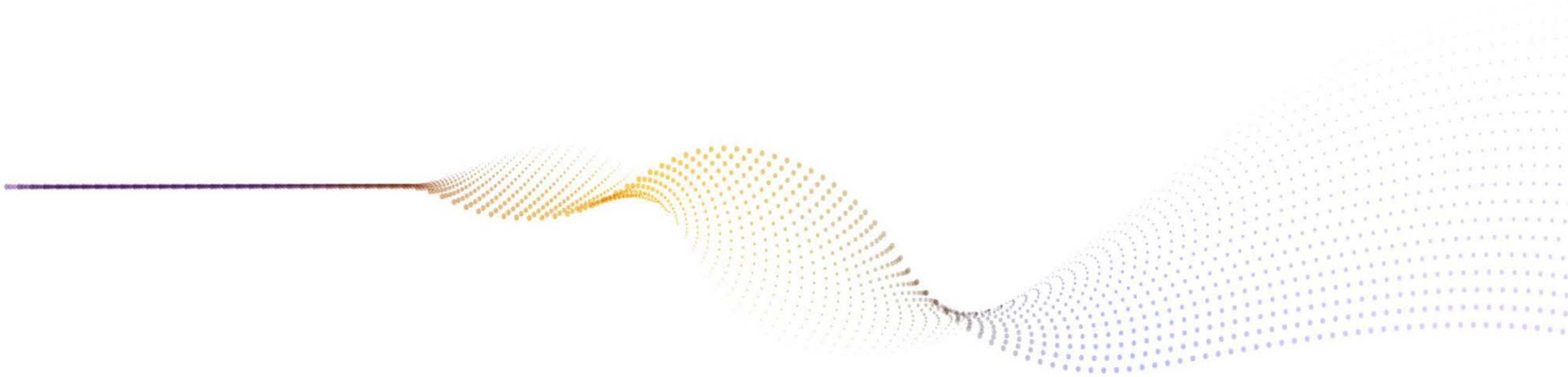
Threads sync to wait for CQ head to pass their completion record position in the CQ

IO Completion

NVMe Protocol revisions can reduce sync points, improve parallelization and improve GPU utilization

micron

SDC 25



Interrupts vs Polling

micron

SDC 25

Interrupts versus Polling

- Polling is well suited to parallelization and SIMT architecture. Polling allows threads to **stay executing in lock steps**, each thread polling a distinct completion entry in //. That maximizes the use of GPU SM scheduler, at each cycle it can run a high number of threads.
- If interrupts were to be used, an interrupt handler would break the parallelism.
  - The thread running the interrupt handler would evict 32 threads running in lock steps on a SM scheduler. Hence running on thread instead of 32 at each cycle.
  - That would be slow, only one thread to handle N completions.
  - That would require expensive synchronization between the threads initiating the IOs and the handler completing them.

micron

SDC 25

Challenges for NVMe to address

NVMe Challenges:

1) QPs scaling challenges
2) GPUs have thousands of cores and scaling - Doorbells, MSI-Xs, Interrupts on NVMe – scale for future ?
3) PCIe lanes extremely busy – is everything optimized for HW?
4) Sub 4K access @100M needs end to end evaluation of all Datapath
5) SSDs will need to support both CPUs and GPU flows while maintaining existing infrastructure such as configuration management, telemetry, error triage etc.

micron

SDC 25

Call to Action

NVMe Challenges:

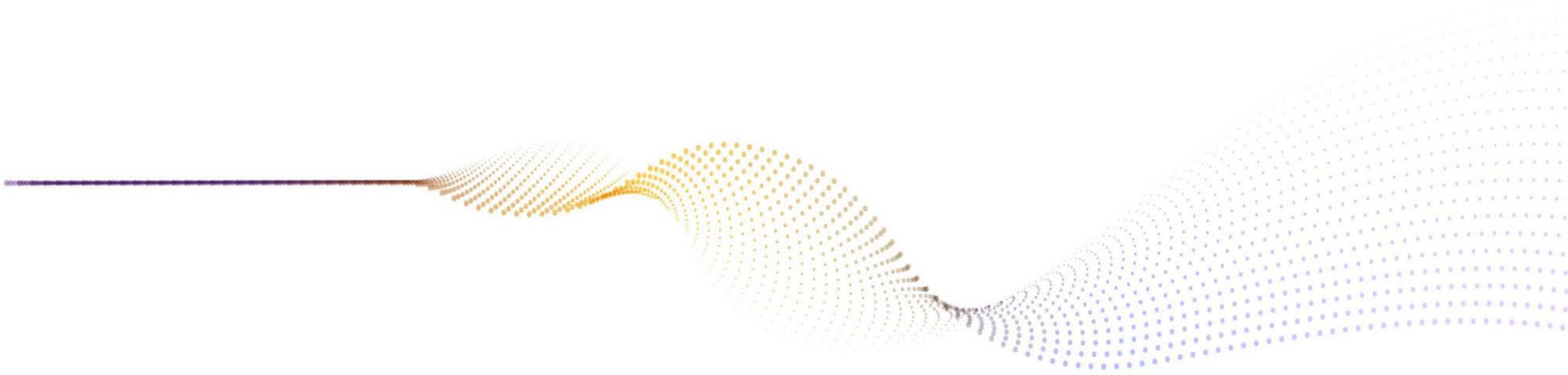
1) GPU needs are fundamentally changing the workloads and NVMe handling
2) Queue Arbitration, Doorbells have efficiency impacts when handling extremely small block I/O switching to polling
3) Extremely large usage of queues for parallel access from GPU threads needs to be examined
4) Interconnecting several GPUs to access storage behind NVMe would require deep reevaluation to reduce overheads down to PCIe transport.

Exciting times ahead!!!

micron

SDC 25

Questions???



Thank you for attending!

Please remember to rate this session. You get access the presentations at
http://sniadeveloper.org/conference

micron

SDC 25