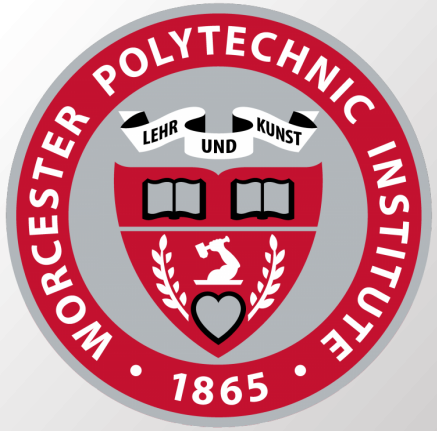
Characterizing Concurrency Mechanisms on NVIDIA GPUs for Deep Learning

Guin Gilman

Robert J. Walls



Concurrent Workloads on GPUs

- Data centers frequently experience low utilization periods with idle resources
- A solution is to do **concurrent execution** of multiple applications with spare resources
- **Concurrent Deep Learning:** Multiple deep learning inference or training tasks on a single GPU

This Talk

Are current GPU scheduling mechanisms sufficient to guarantee **predictable and low turnaround times** while also maintaining high **utilization**?

Our Experiments

- We analyzed three concurrency mechanisms on NVIDIA GPUs:
  - Priority streams
  - Time-slicing
  - Multi-Process Service (MPS)
- We characterized their performance for concurrent deep learning workloads

Our Findings

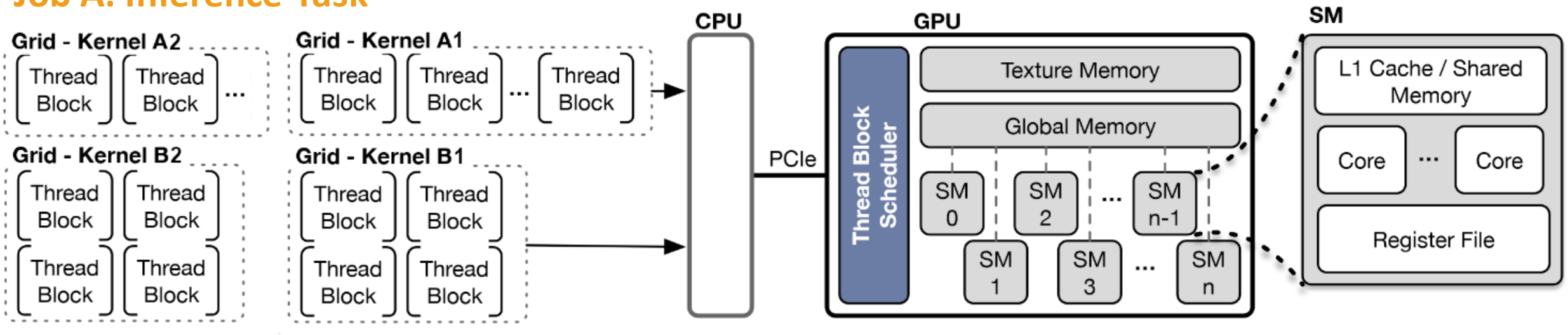
- Concurrent execution on GPUs is suboptimal without the ability to reassign resources dynamically
- Concurrency mechanisms on NVIDIA GPUs also lack sufficient task prioritization
- Without dynamic resource assignment and task prioritization, it is hard to keep turnaround times low and utilization high consistently
- Fine-grained preemption is a promising solution to improve predictability and utilization

Workload Characterization

Case Study: Deep Learning Concurrent Workloads

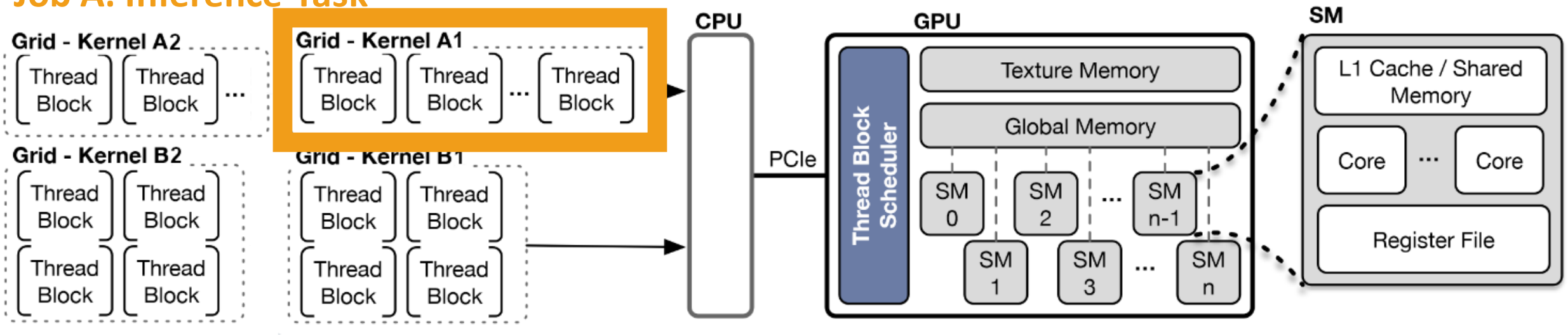
- A deep learning concurrent workload consists of:
  - **High-priority inference jobs**: a series of inference requests where low turnaround time and variance are important
  - **Low-priority training job**: training task for a single model run on the same GPU using spare resources to achieve higher GPU utilization

Job A: Inference Task



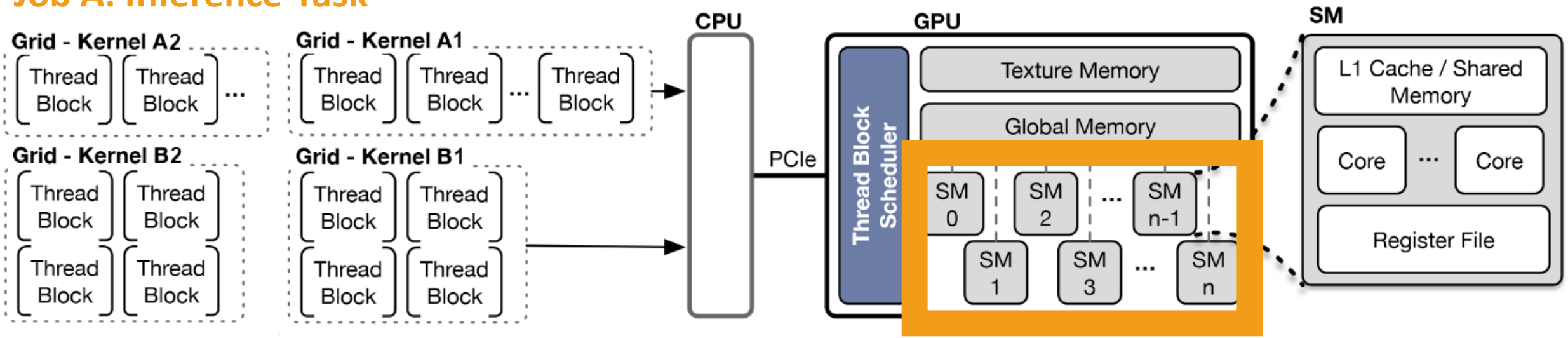
Job B: Training Task

Job A: Inference Task



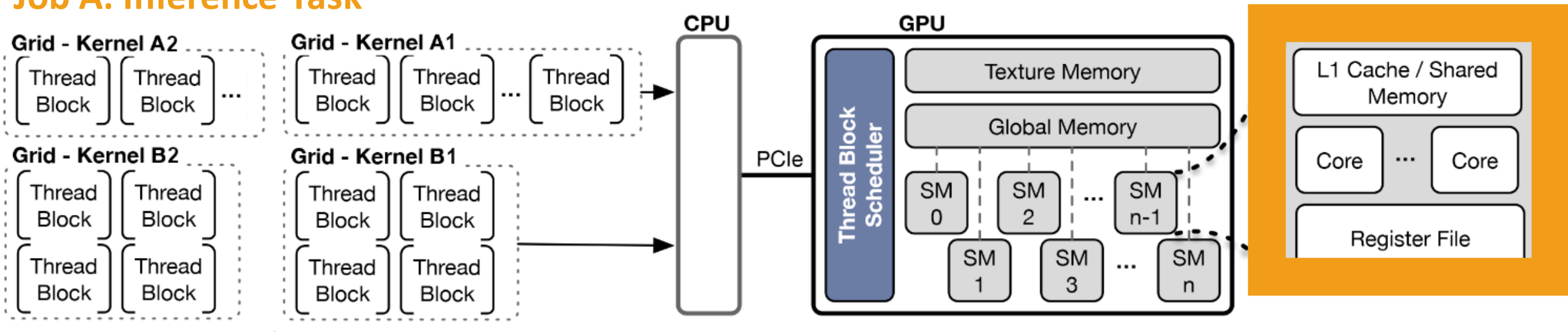
Job B: Training Task

Job A: Inference Task



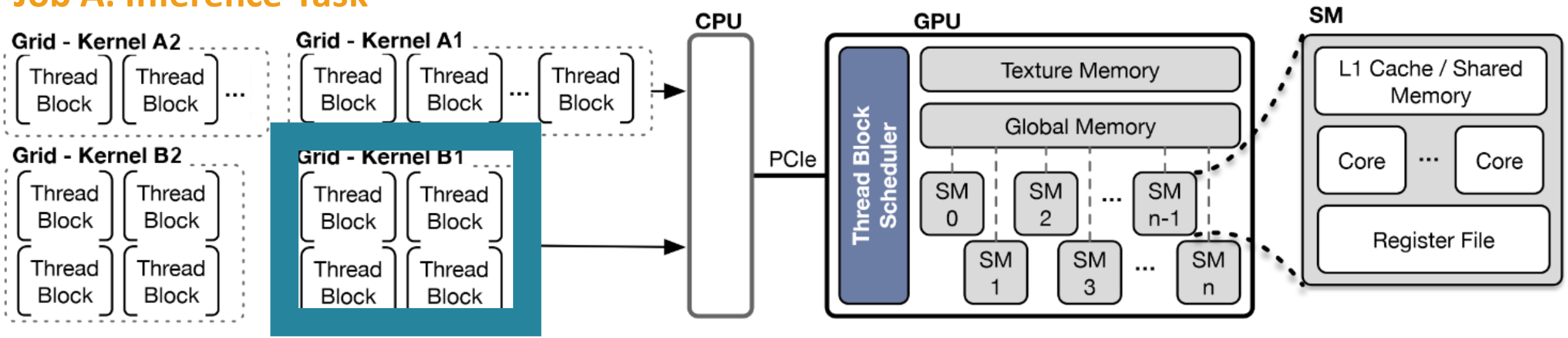
Job B: Training Task

Job A: Inference Task



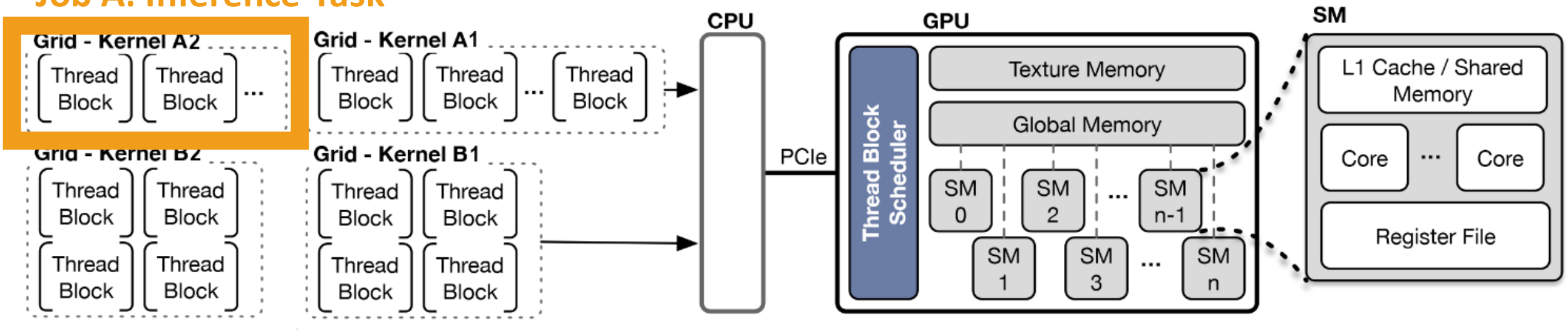
Job B: Training Task

Job A: Inference Task



Job B: Training Task

Job A: Inference Task



Job B: Training Task

Depending on which mechanism is used, you can change where and when thread blocks are scheduled

Our Empirical Workload

- Two sources of Models:
  - MLPerf inference and training benchmarks
  - PyTorch example models
- PyTorch inference tasks: 5000 consecutive inference requests
- MLPerf inference tasks had two modes:
  - Consecutive, i.e., single stream (same as PyTorch task)
  - Poisson, i.e., server (request arrivals followed a Poisson distribution)

Important Workload Characteristics

- Implemented as a **sequence of kernels** executed on the GPU
  - Resource requirements fluctuate as different kernels are launched

Important Workload Characteristics

- Implemented as a **sequence of kernels** executed on the GPU
- **Kernel runtime**
  - Long-running: >1ms, occupies resources for a long period of time without interruption
  - Training jobs either had a significant amount (40-60%) or almost none (2-6%)

Important Workload Characteristics

- Implemented as a **sequence of kernels** executed on the GPU
- **Kernel runtime**
- **Kernel size**
  - Large: Grid of blocks are unable to fit on the GPU, preventing other jobs from sharing resources
  - Almost all training jobs had a significant amount (35-70%)
  - Half of inference jobs had a significant amount (15-50%)

Existing Concurrency Mechanisms

Priority Streams

- From within the same process on different streams
- Three priority levels
- Uses the leftover and most-room policies
- No preemption of thread blocks
  - High-priority kernels wait for any current blocks to finish

Time-Slicing

- Each application runs for a fixed-length time-slice of ~4ms
- Applications' time slices are executed in a round-robin fasion
- Two separate applications are never executing at the same time
- Possibility for OoM error if combined resource usage exceeds GPU's available resources

Multi-Process Service (MPS)

- A server intercepts the kernels from different processes
- Launched to the GPU as if they're from the same context on different streams
- Can specify an upper limit on threads per client
- For running multiple processes when there are resources leftover
- Server performs load-balancing when there are spare resources

High-Level Summary

Features	Streams	Time-slicing	MPS
Separate application contexts			
Kernel concurrency			
Task prioritization			
Coarse-grained preemption			

Characterization of Existing Mechanisms

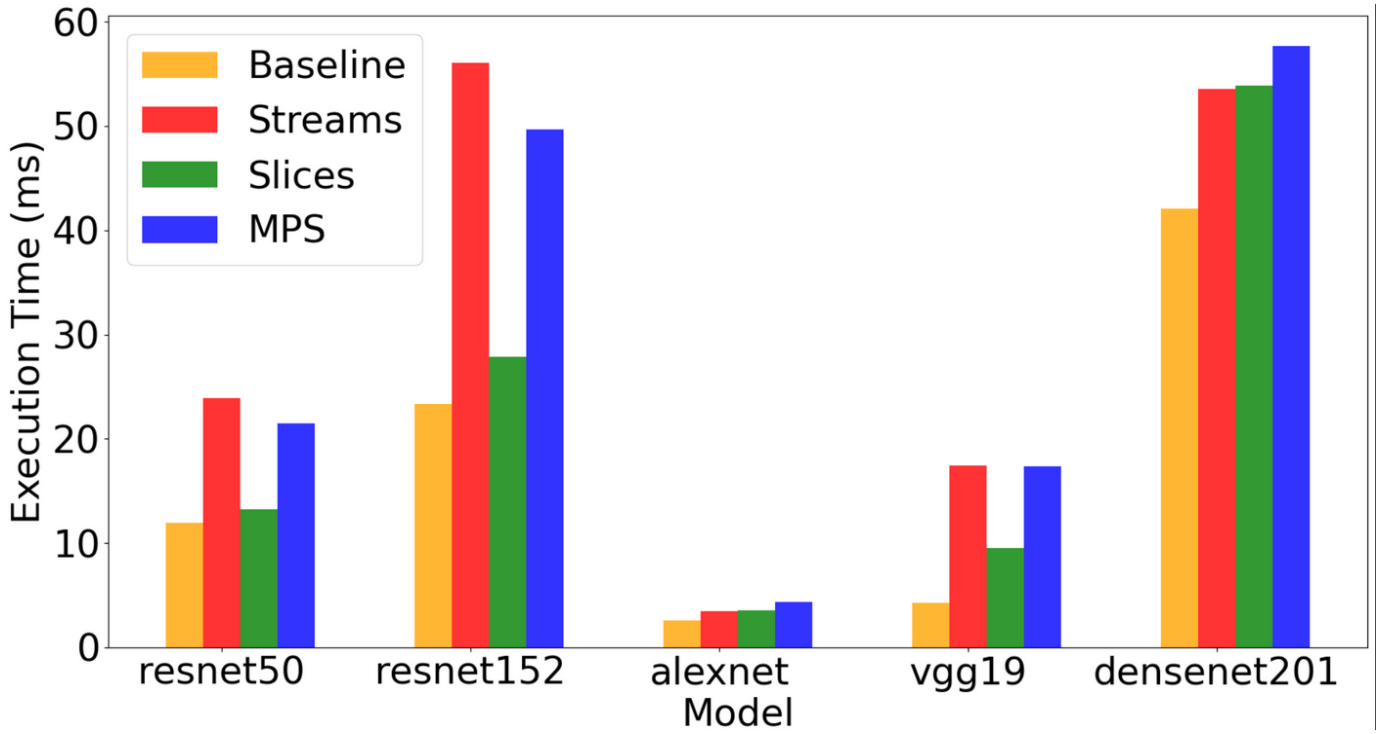
Metrics

- Turnaround time
  - Time to return results after arrival
- Utilization
  - Training execution time
- Variance
  - Variance in turnaround time

Observation 1

Priority streams cannot preempt executing thread blocks in the middle of execution, resulting in compounded delay and resource contention leading to high/less predictable turnaround times.

Compounded Delay

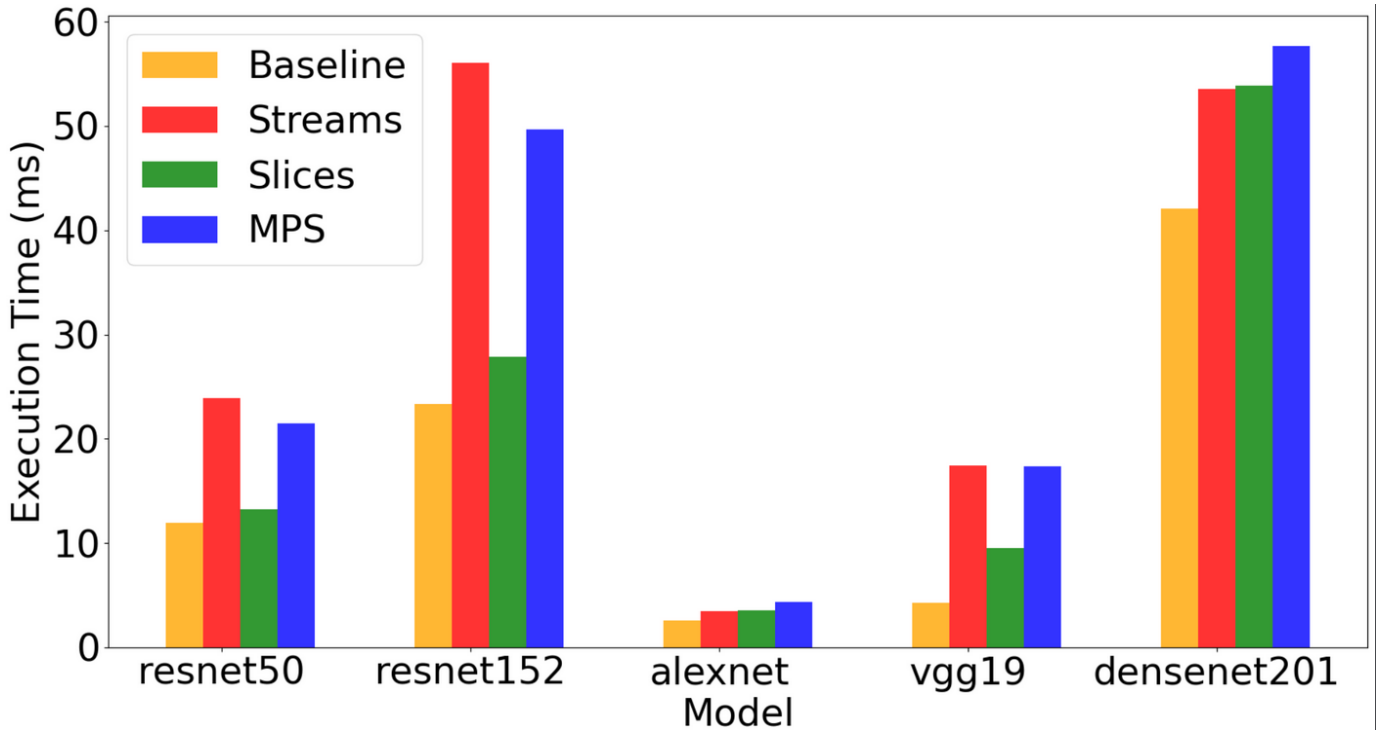


- **Compounded delay**, where kernels are forced to wait behind long-running and large ones

Observation 5

While MPS increased utilization overall, it also caused intra-SM resource contention that added to the execution times of both the training and inference tasks.

Compounded Delay/Resource Contention

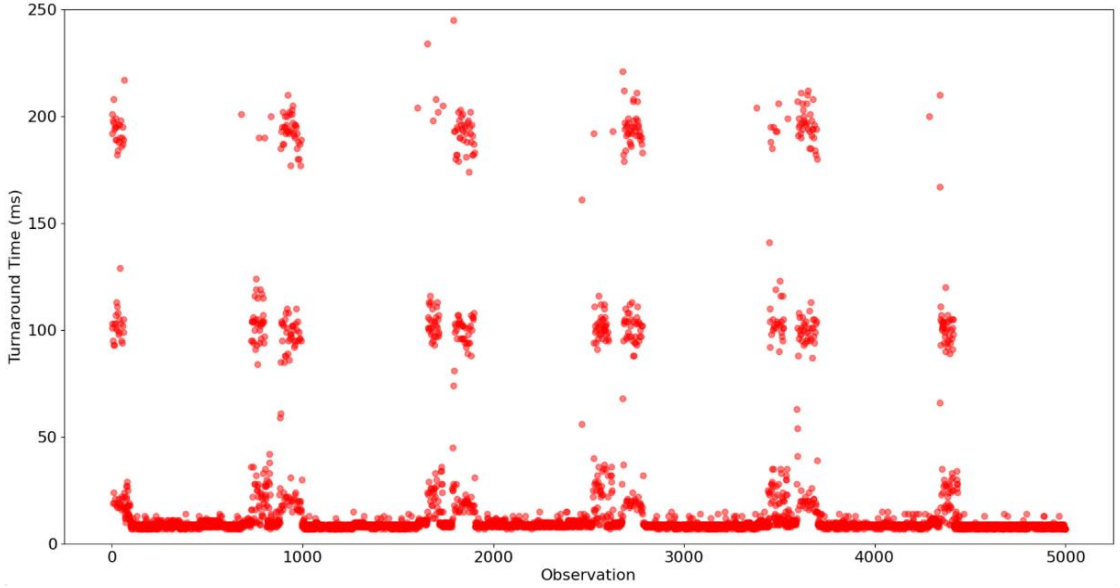


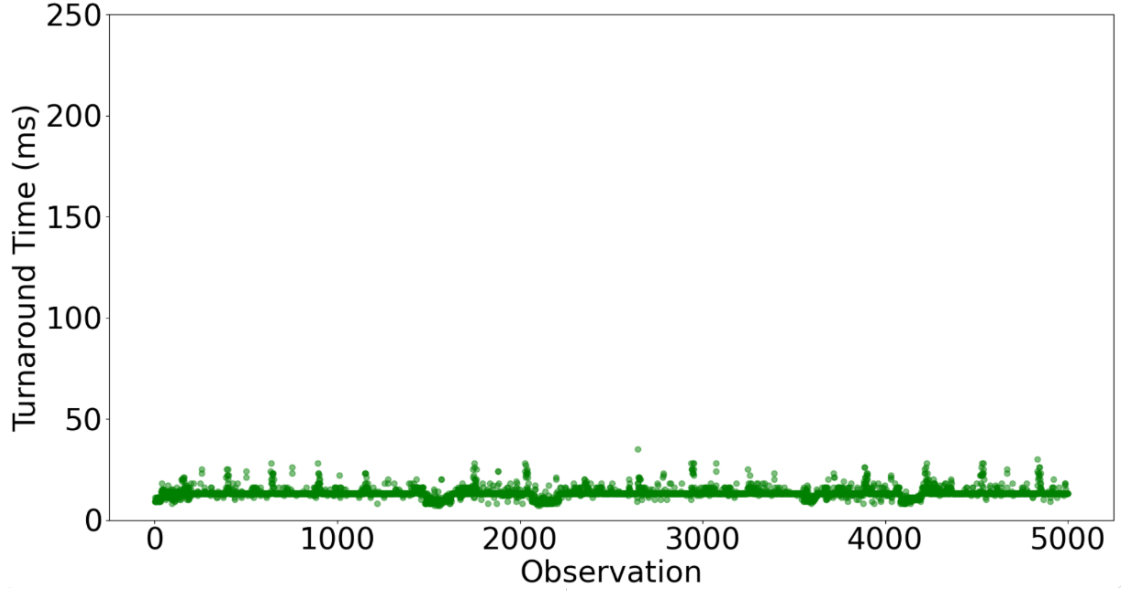
- **Compounded delay**, where kernels are forced to wait behind long-running and large ones
- **Resource contention**, where two smaller kernels share the GPU and interfere with each other

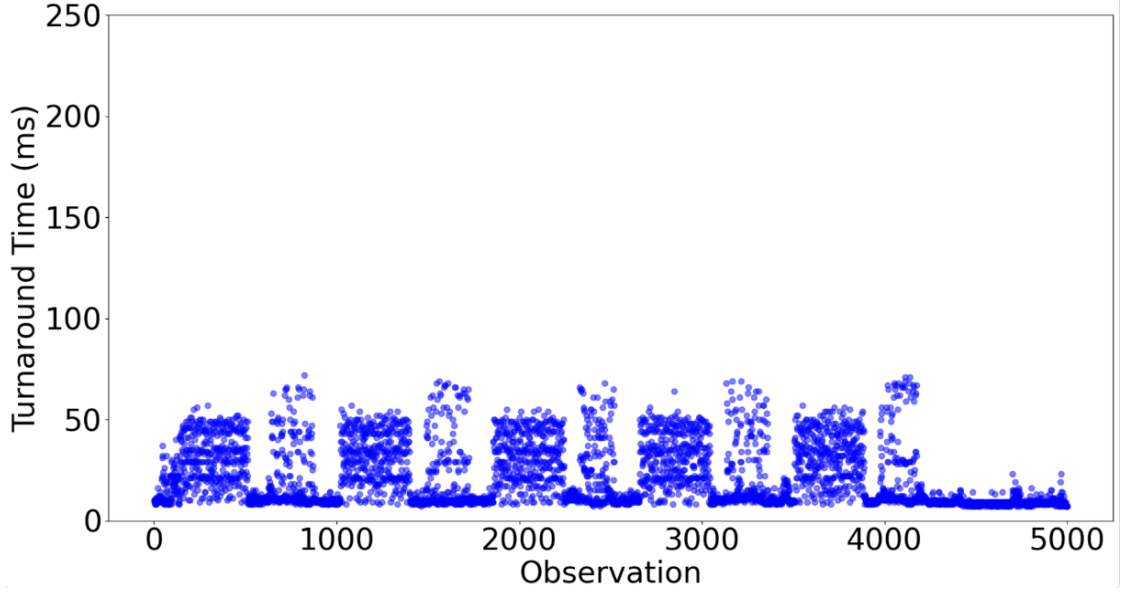
Observation 2

Time-slicing tended to exhibit predictable and low turnaround times for models with relatively low baseline turnaround times, unless there is memory transfer contention. This came at the cost of poor utilization, as the two tasks never actually executed on the GPU at the same time.

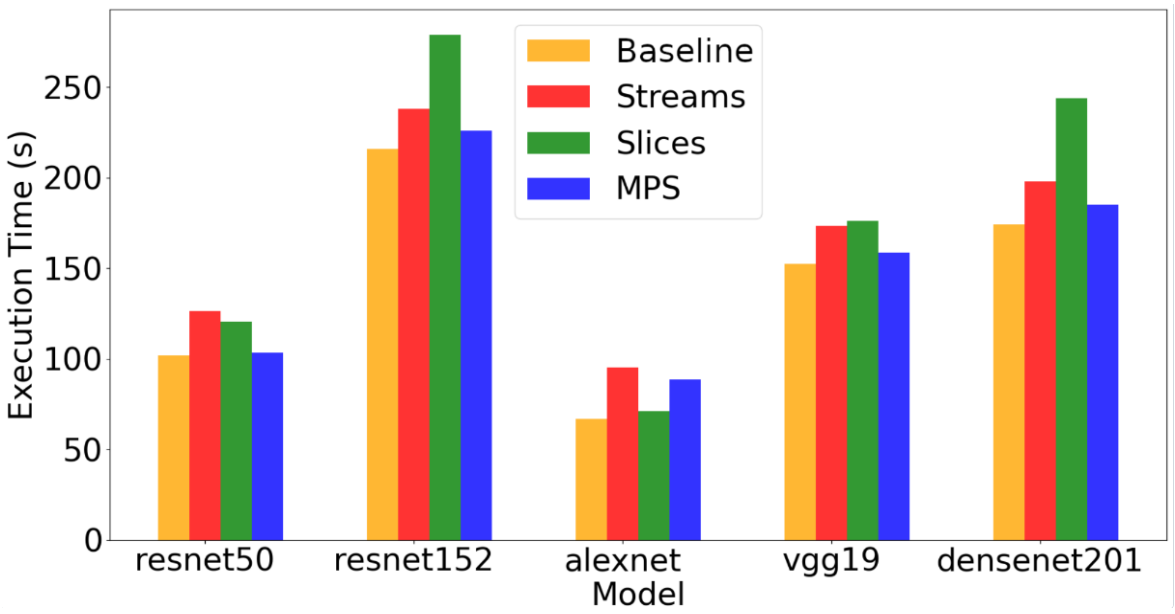
PyTorch Variance

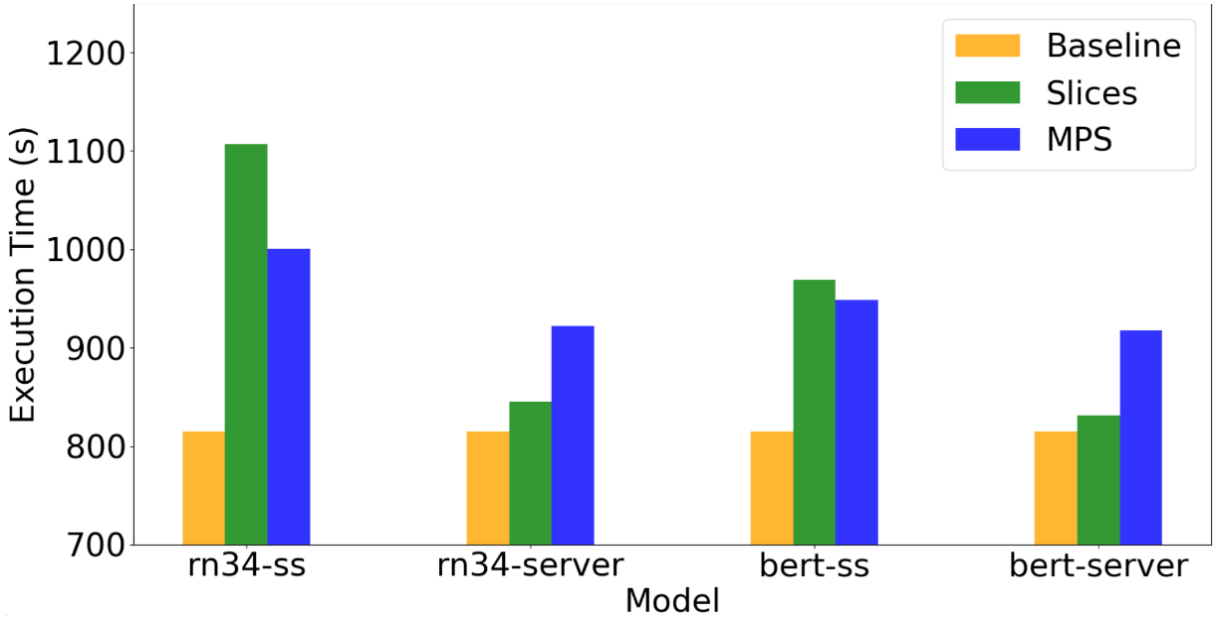






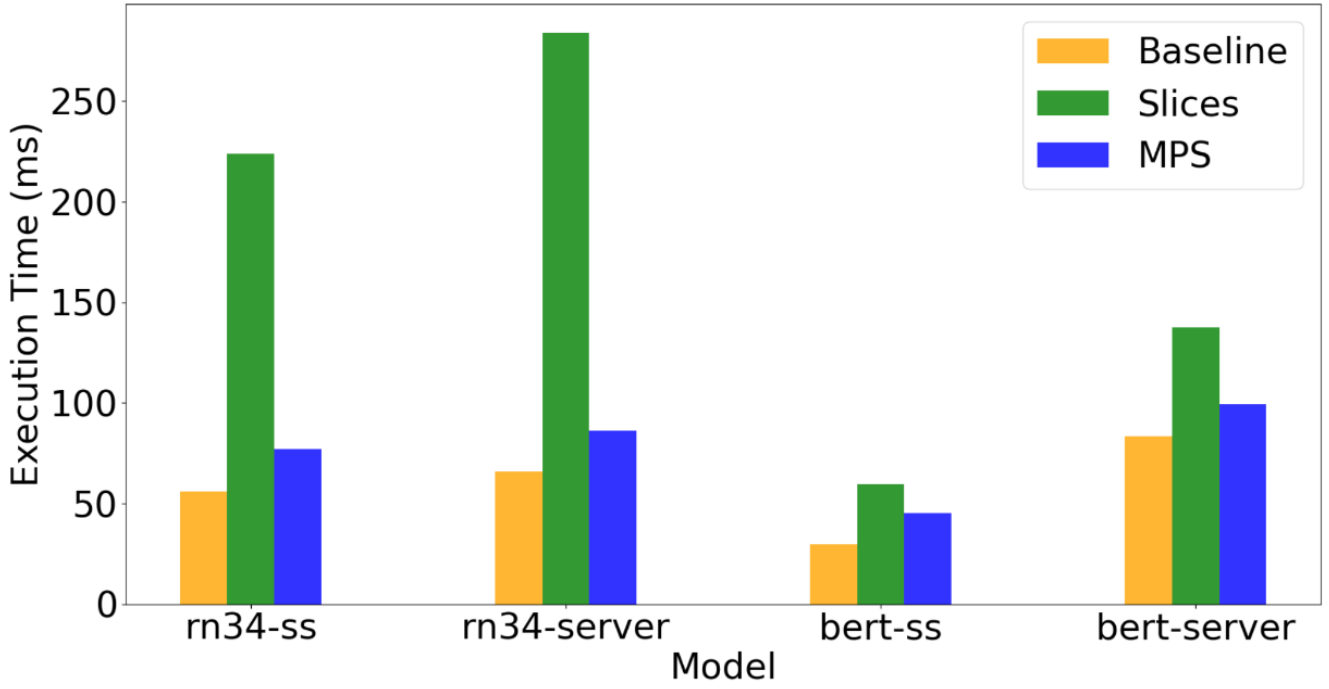
Utilization: PyTorch vs. MLPerf



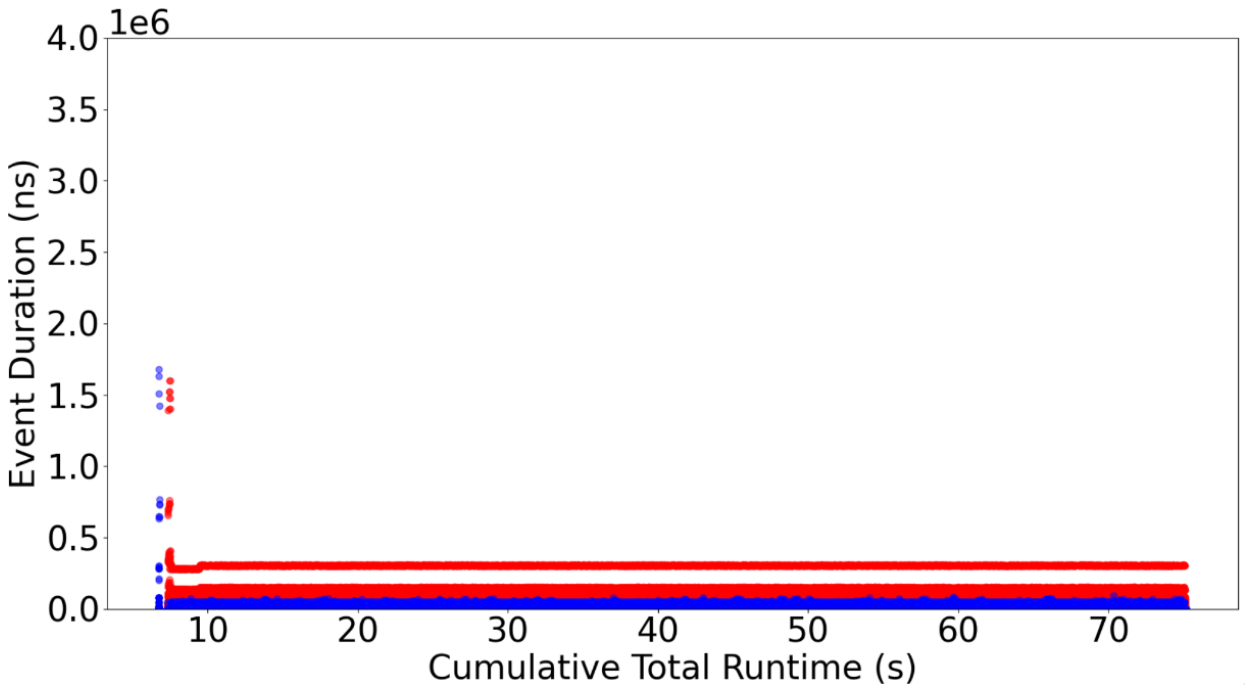


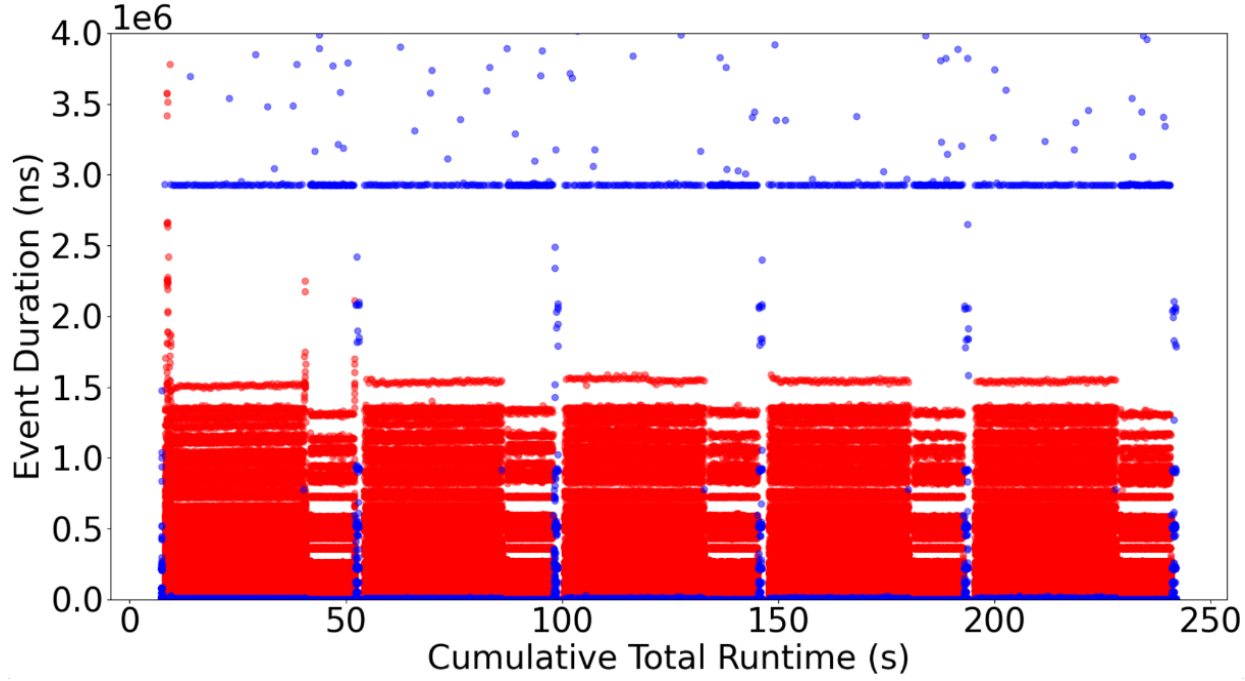
MLPerf Turnaround Times

- Compared to PyTorch models, time-slicing saw drastic increases in turnaround time
- One possible explanation is **memory transfer interference**



Inference Task Transfer Times: Baseline vs. Time Slices





The Potential of Fine-Grained Preemption

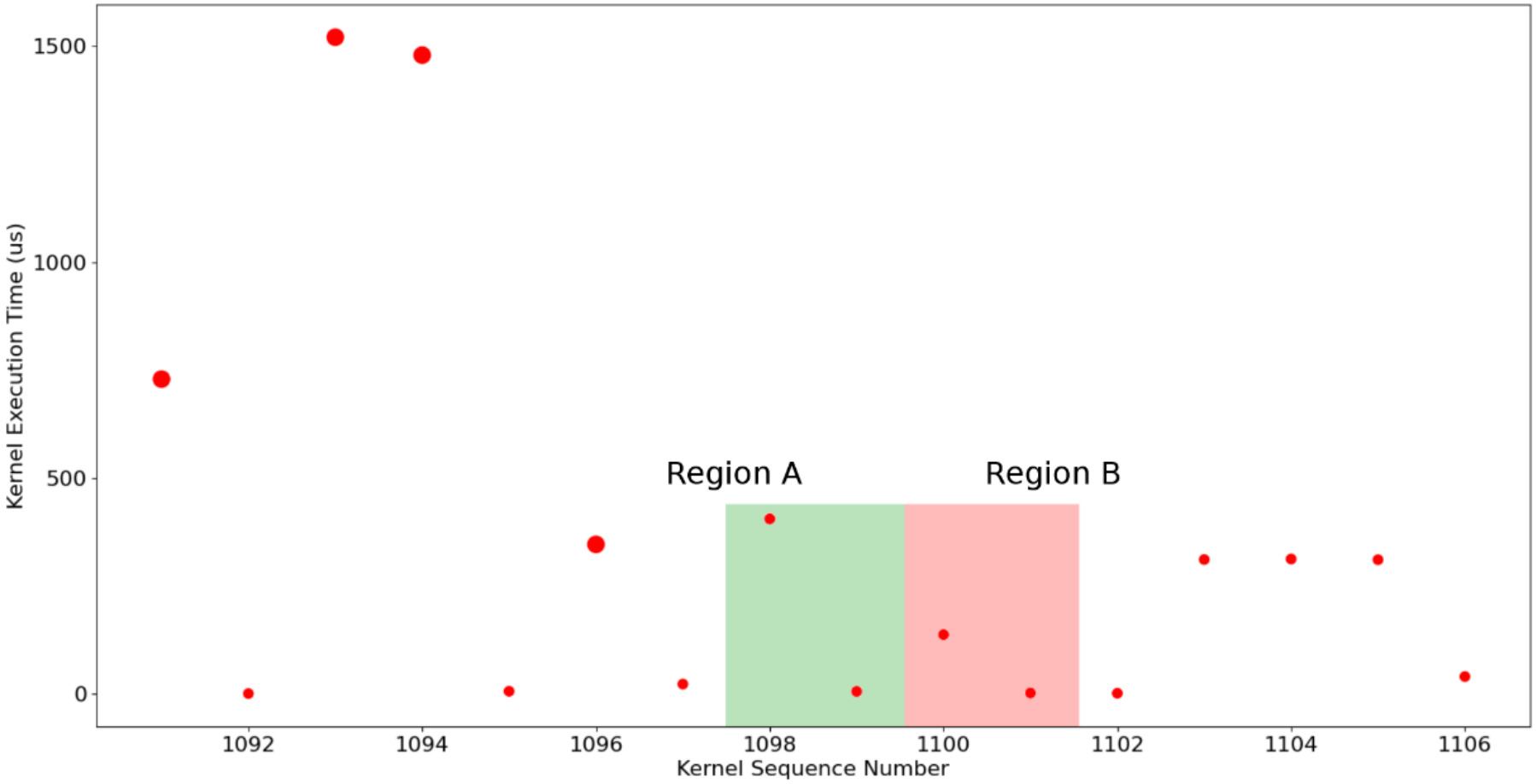
Fine-Grained Preemption

- Two main issues to address:
  - Unpredictability/inefficiency due to **stochastic inference requests**
  - Unpredictability due to **resource contention**
- Improve performance through **fine-grained preemption**:
  - Interrupt any set of thread blocks on the GPU
  - Relaunch those blocks later from where they left off

Benefits of Fine-Grained Preemption

- Could complement priority streams or MPS
  - Would eliminate the effects of compounded delay for inference task
  - Could allow for rearrangement and load-balancing to avoid resource contention
  - Makes task prioritization possible in the case of MPS
- Improve predictability (similarly to time-slicing) without sacrificing utilization

Opportunities to Hide/Reduce Preemption Cost



Summary

Our Findings

- Concurrent execution on GPUs is suboptimal without the ability to reassign resources dynamically
- Concurrency mechanisms on NVIDIA GPUs also lack sufficient task prioritization
- Without dynamic resource assignment and task prioritization, it is hard to keep turnaround times low and utilization high consistently
- Fine-grained preemption is a promising solution to improve predictability and utilization, and its costs can be offset by its potential benefits

Learn more about our work at: cake.wpi.edu