TReM: A Task Revocation Mechanism for GPUs

Manos Pavlidakis[1,2]
manospavl@ics.forth.gr

Stelios Mavridis[1]
mavridis@ics.forth.gr

Nikos Chrysos[1]
nchrysos@ics.forth.gr

Angelos Bilas[1,2]
bilas@ics.forth.gr

[1] Institute of Computer Science, Foundation for Research and Technology - Hellas, Greece
[2] Computer Science Department, University of Crete, Greece

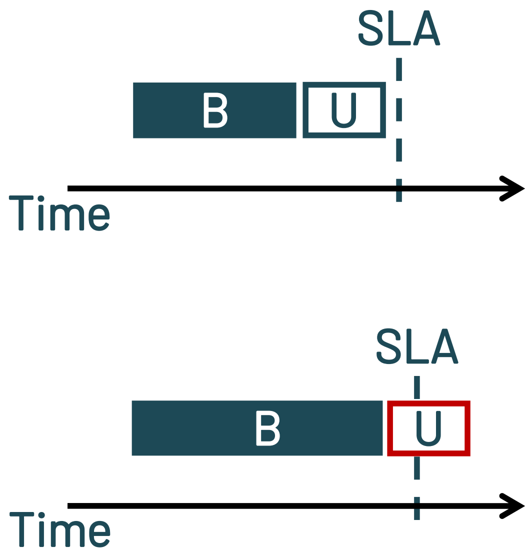
GPU sharing

- Today, GPUs are offered in a dedicated manner by cloud providers
- To ensure SLA for user-facing tasks
  - User-facing task's response time < SLA
- But GPUs are underutilized
- State of the art approaches increase GPU utilization
- By using idle GPUs for batch tasks
  - Batch task does not have strict response time requirements
- If batch task **execution time** adequately < SLA

➢ User-facing task can meet its SLA target

- **But** batch tasks execution time ≥ SLA

➢ Leads to SLA violation

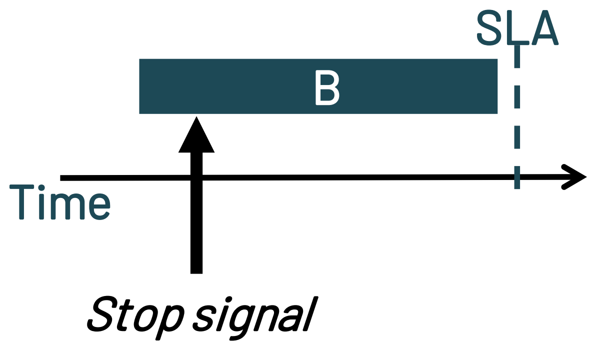


Preemption can reduce SLA violations

- GPU preemption approaches incur **variable** & **high** latency:

1. Rely on existing thread blocks **or** slice tasks to provide preemption points
    - Rare preemption points → high latency
    - Frequent preemption points → increase task execution time
2. Store stopped task's state
    - In GPU memory → memory monopolization
    - In Host memory → variable latency

- High preemption latency affects violations



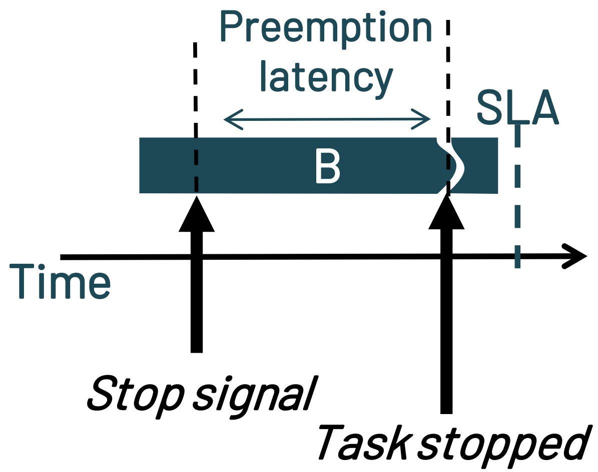
Preemption can reduce SLA violations

- GPU preemption approaches incur **variable** & **high** latency:

1. Rely on existing thread blocks **or** slice tasks to provide preemption points
   - Rare preemption points → high latency
   - Frequent preemption points → increase task execution time

2. Store stopped task's state
   - In GPU memory → memory monopolization
   - In Host memory → variable latency

- High preemption latency affects violations

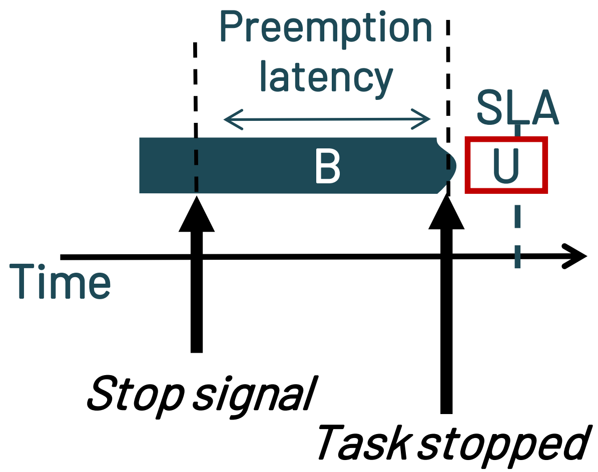


Preemption can reduce SLA violations

- GPU preemption approaches incur **variable** & **high** latency:

1. Rely on existing thread blocks **or** slice tasks to provide preemption points
    - Rare preemption points → high latency
    - Frequent preemption points → increase task execution time

2. Store stopped task's state
    - In GPU memory → memory monopolization
    - In Host memory → variable latency



- High preemption latency affects violations

❑ We need preemption mechanism with **constant** & **low** latency

- Much shorter than the SLA

TReM: Task Revocation Mechanism for GPUS

✓ With **constant** & **low** latency

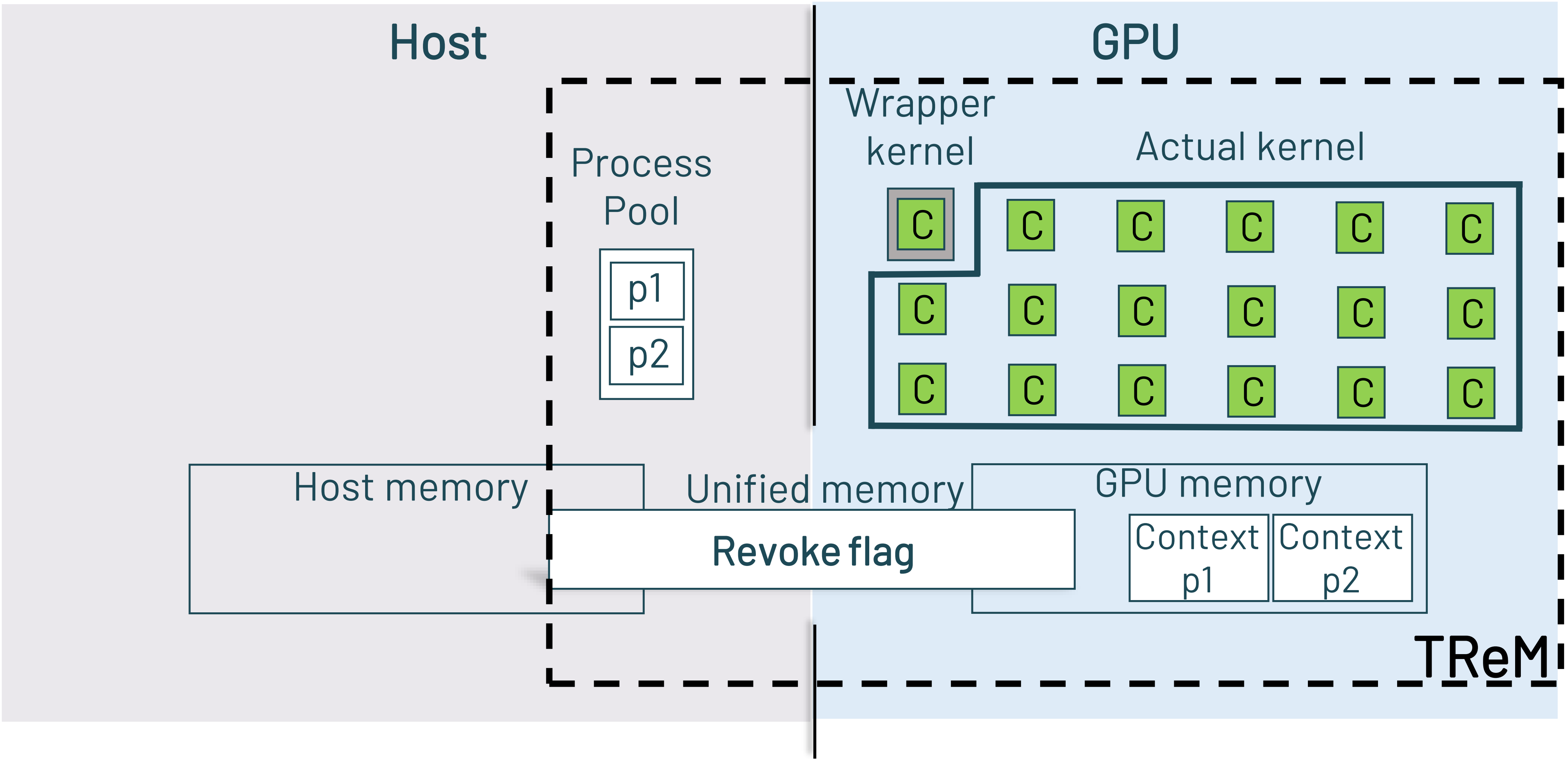
- To achieve that TReM:
  - Stops a task at any point of its execution → low & constant latency
  - Does not store the state of the revoked task → constant latency
  - Replays the revoked task later
- To stop a task TReM uses 3 mechanisms
  1. CUDA dynamic parallelism
  2. CUDA unified memory
  3. asm(trap)
- We examine the effectiveness of TReM on SLA violations
  - Using different scheduling policies
  - Focusing on long running batch tasks (i.e. execution time relative to SLA)

Why TReM?

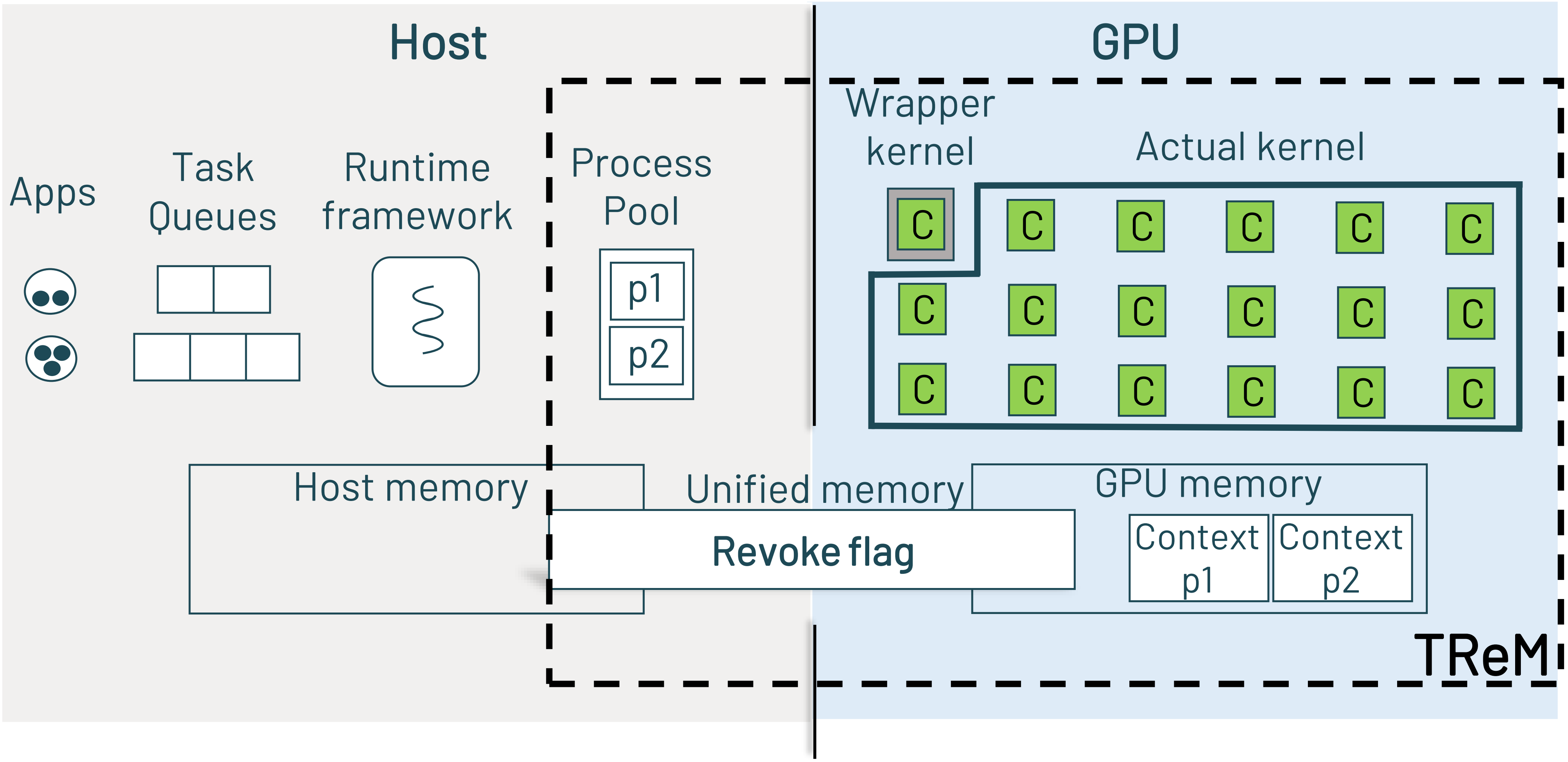
Desired features	FLEP	GPES	Pascal Preemption	Chimera	TReM
Preemption/Revocation	P	P	P	P	R
Provides Low & Constant preemption latency	–	–	–	+	+
Handles tasks with large memory footprint	–	–	+	+	+
Does not need kernel source code	–	–	+	–	+
Supports all NVIDIA GPUs	+	+	–	+	+

TReM Design Overview

TReM components

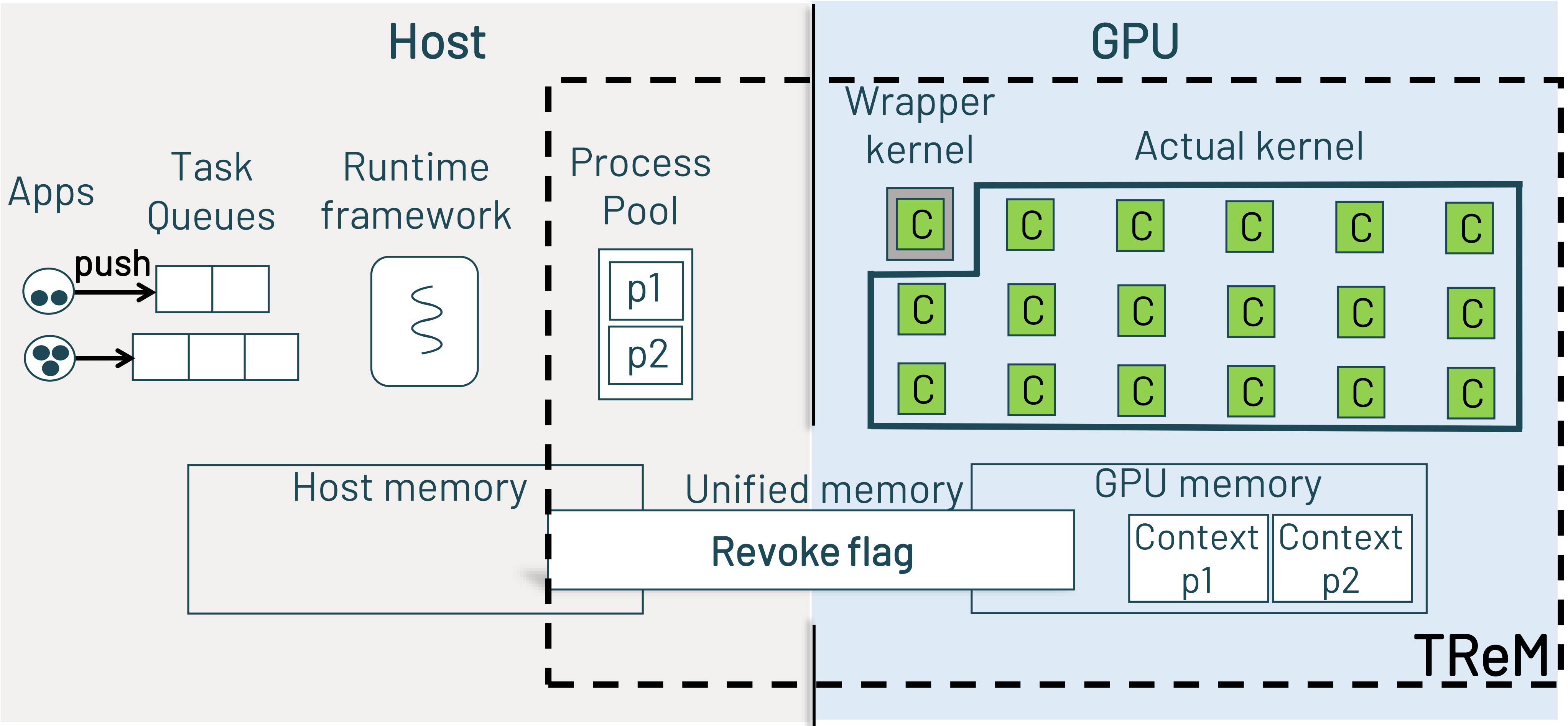


Overall system with TReM

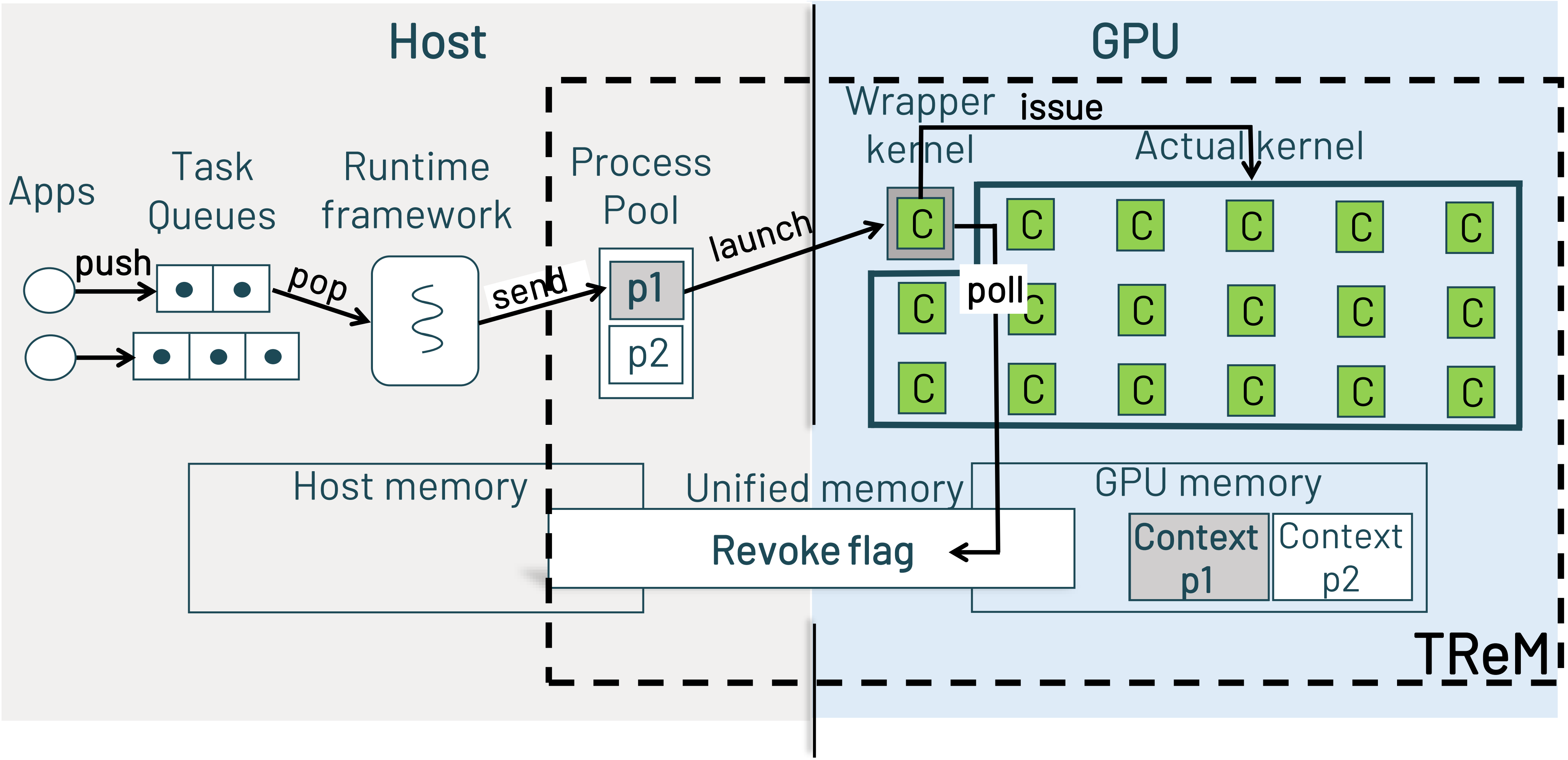


Start a kernel with TReM

- Using CUDA Dynamic Parallelism

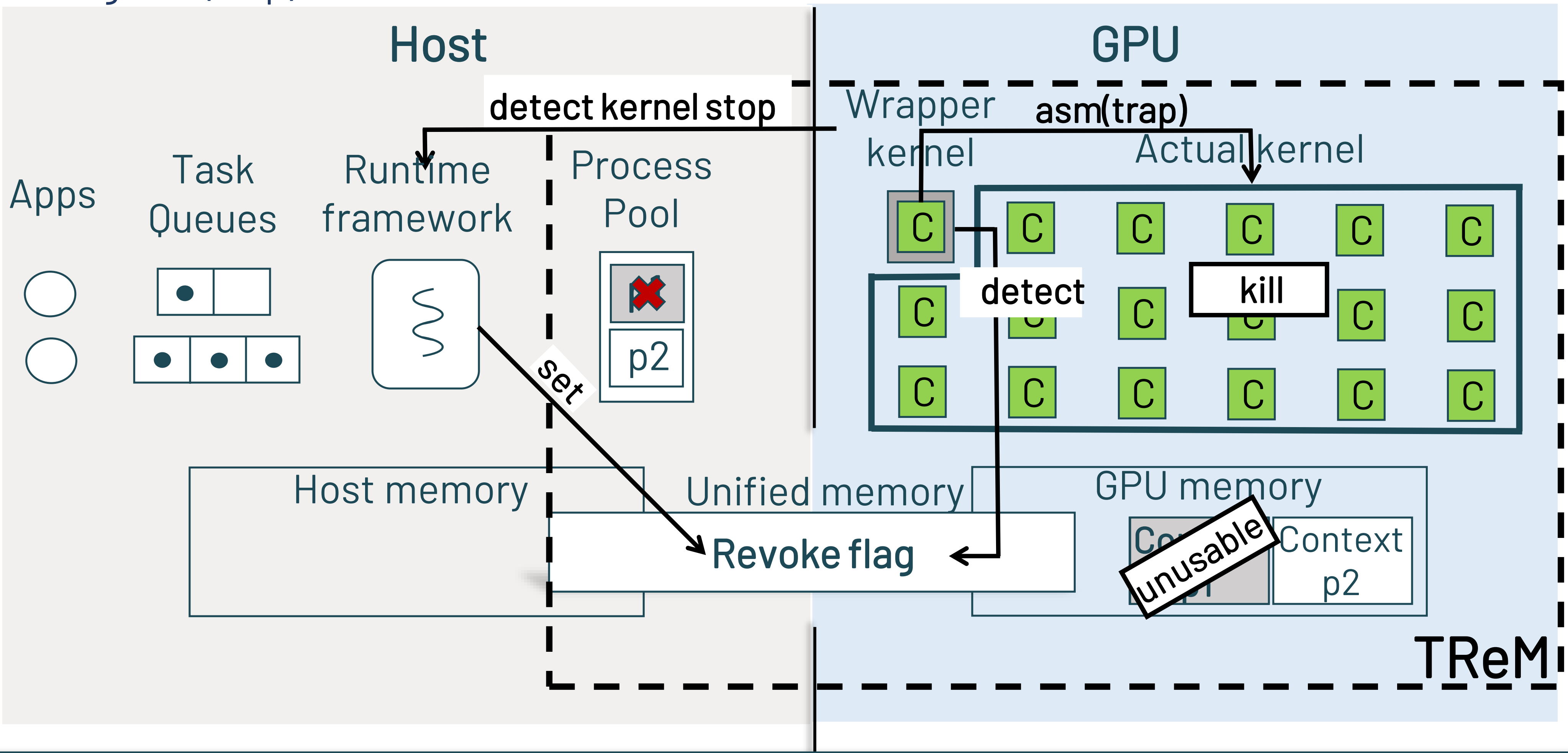


Start a kernel with TReM

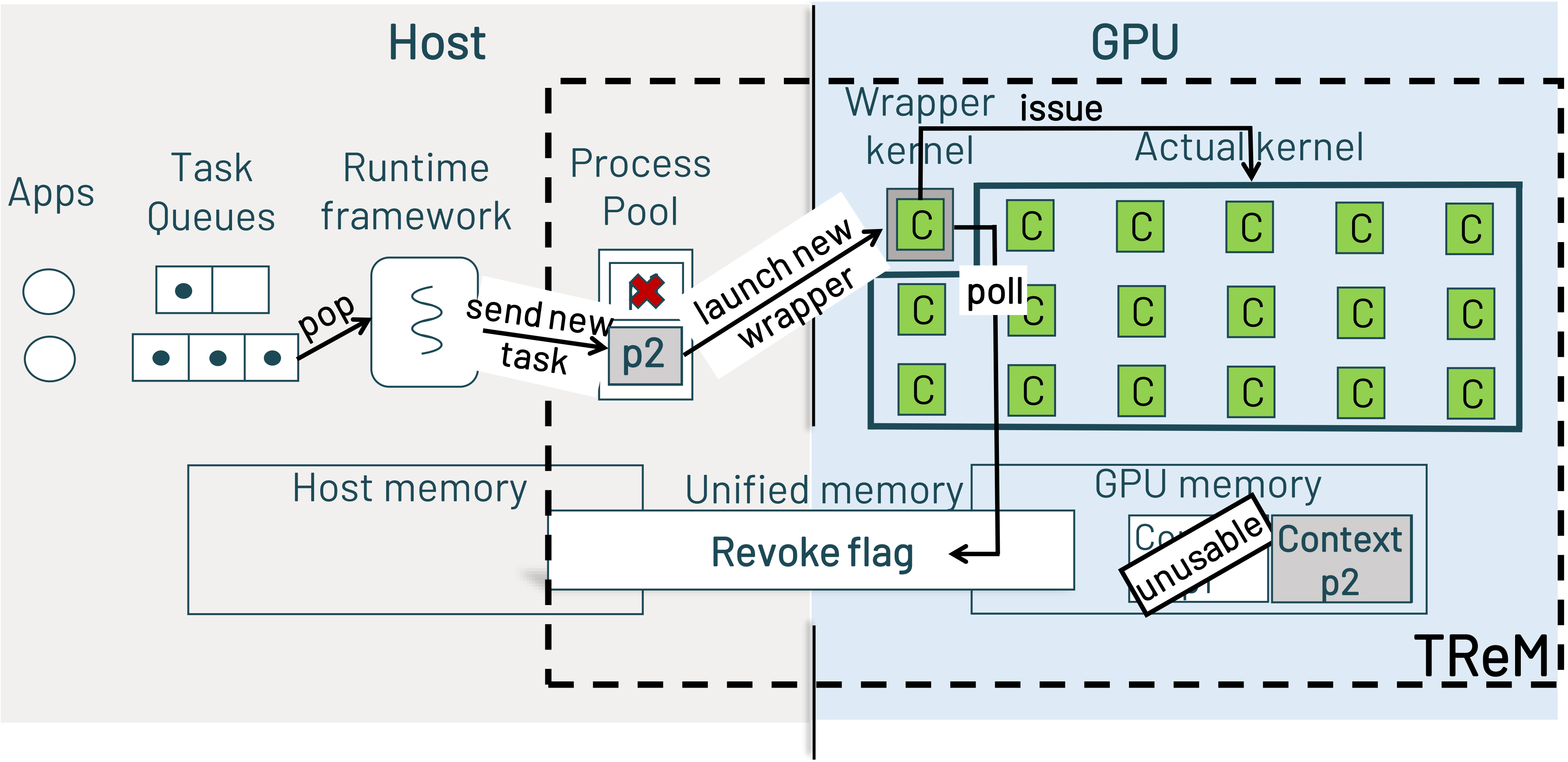


Revoke a kernel with TReM

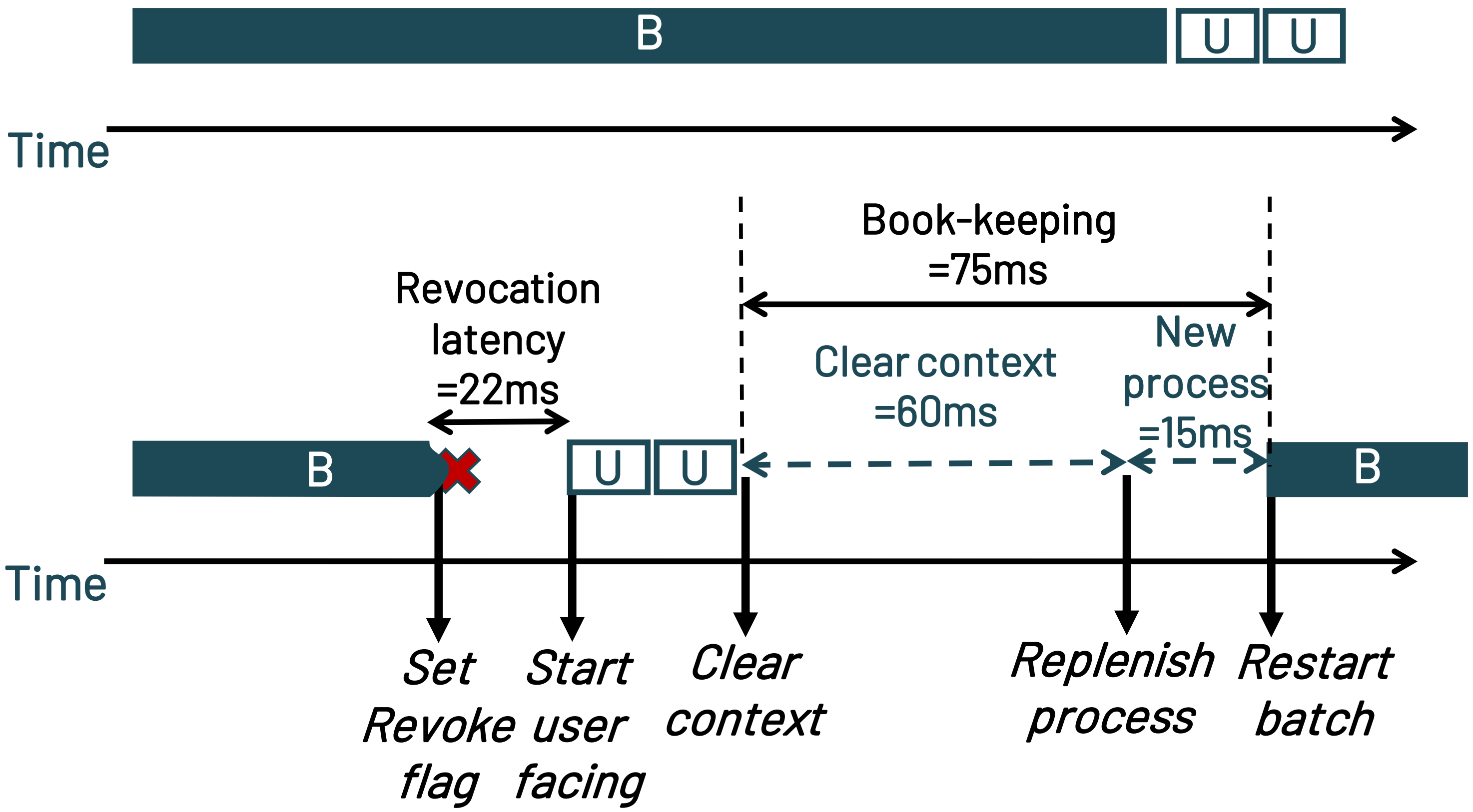
➤ Using asm(trap)



Revoke a kernel with TReM



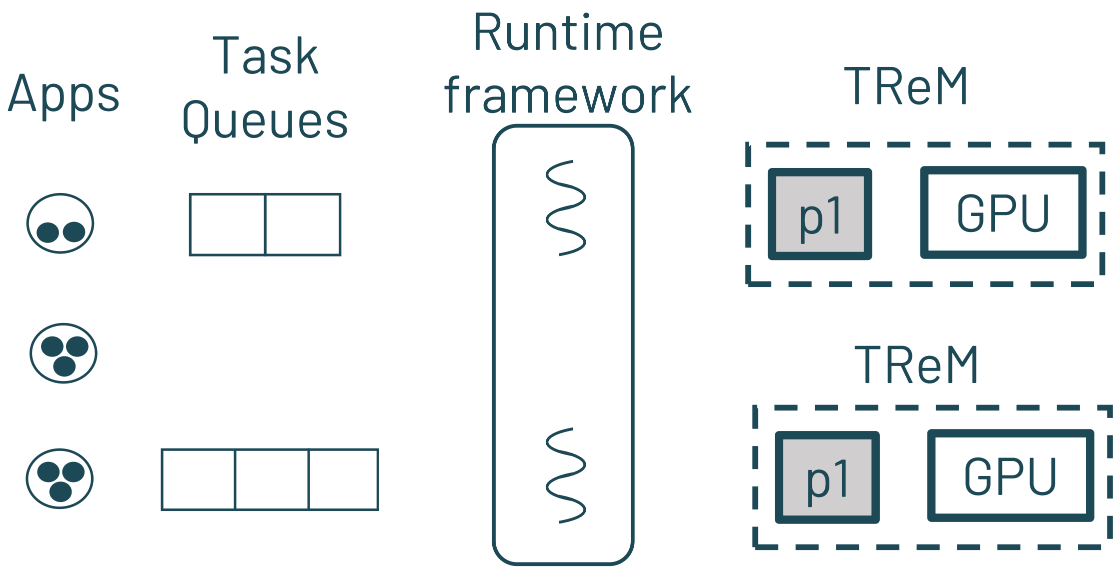
TReM breakdown



- Revocation time = 22ms
  - To stop the task: **5 ms**
  - To start the new task: **17 ms**
- Book-keeping time = 75 ms
  - Postponed until next batch task
  - To clear the GPU context: 60ms
  - To replenish the process pool: 15ms

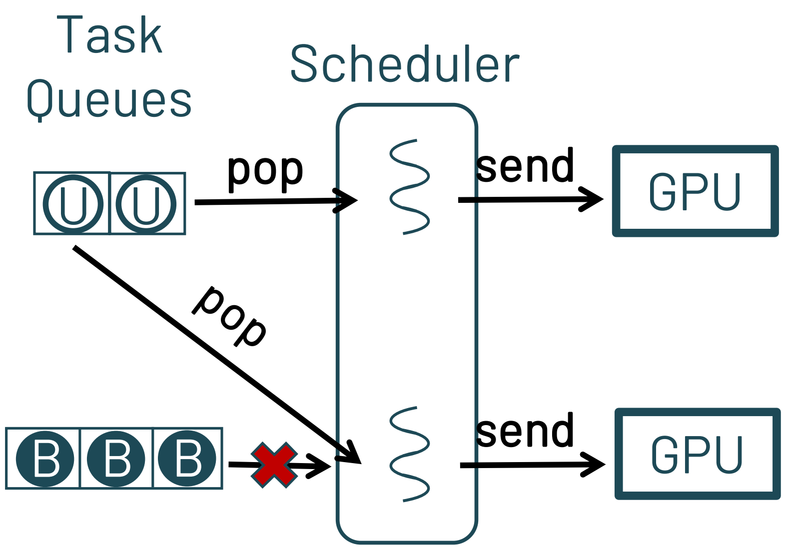
TReM with multiple GPUs

- Servers today
  - Have multiple GPUs & run multiple applications
- In such setups TReM runs in every GPU
- To handle multiple GPUs & apps
  - We design & implement a runtime framework
- The runtime framework
  - Instructs TReM when to revoke a kernel
  - Minimize lost work due to revocations
  - Selects which task queue to serve according to a scheduling policy
- We use two scheduling policies:
  - (Baseline) Priority: Prioritizes user-facing over batch tasks
  - Elastic: Packs user-facing tasks in a GPU → do not violate the SLA
    - Devotes the remaining GPUs to batch tasks



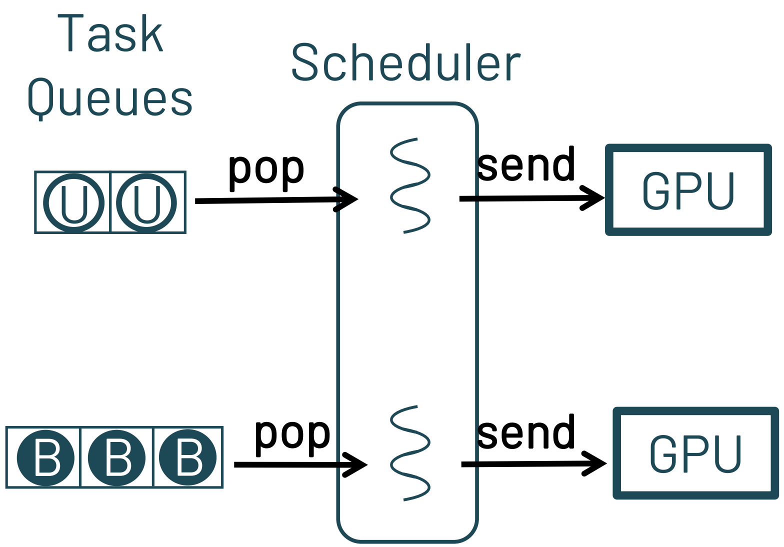
Priority vs Elastic scheduling policy

Priority



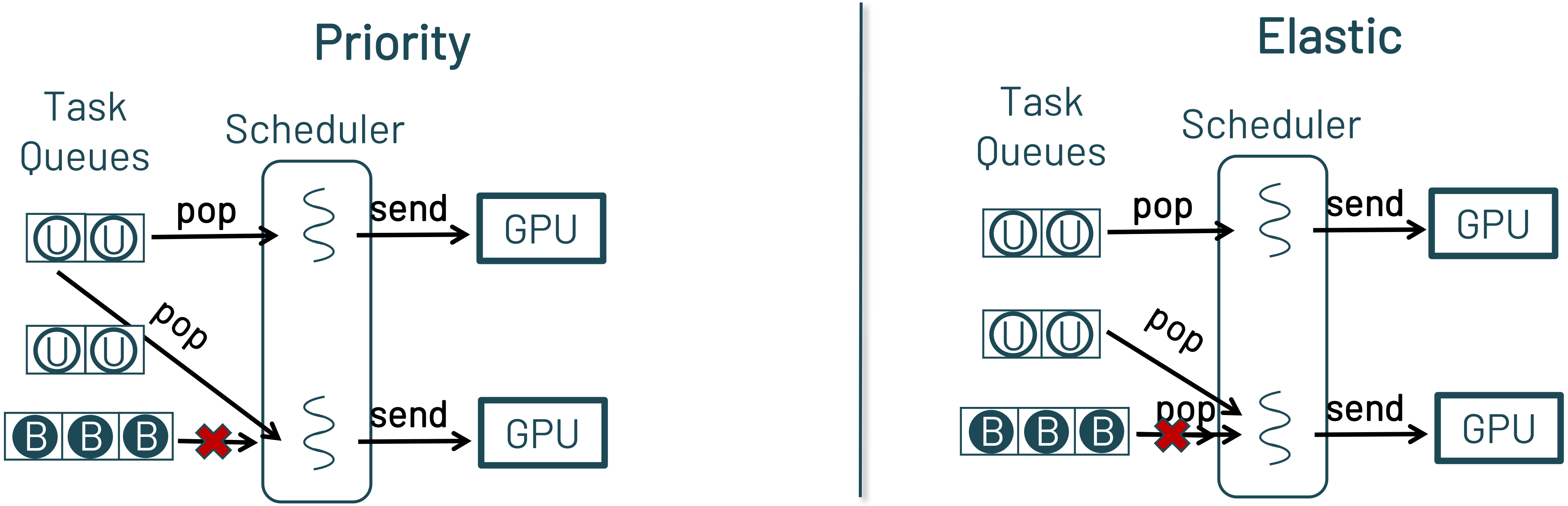
- Does not account the user-facing latency
- Assigns all GPUs to user-facing
  - As many as the number of user-facing tasks
- Postpones the execution of batch tasks

Elastic



- Assigns the minimum number of GPUs
  - As such user-facing response time < SLA
  - In our example 1xGPU is sufficient
- Provides the remaining GPUs to batch tasks

Priority vs Elastic scheduling policy

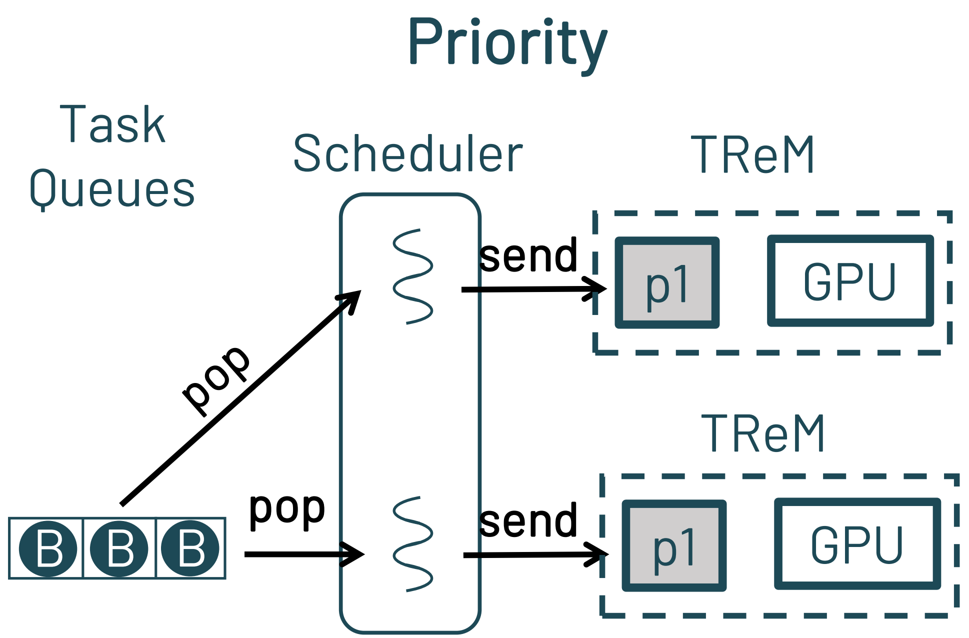


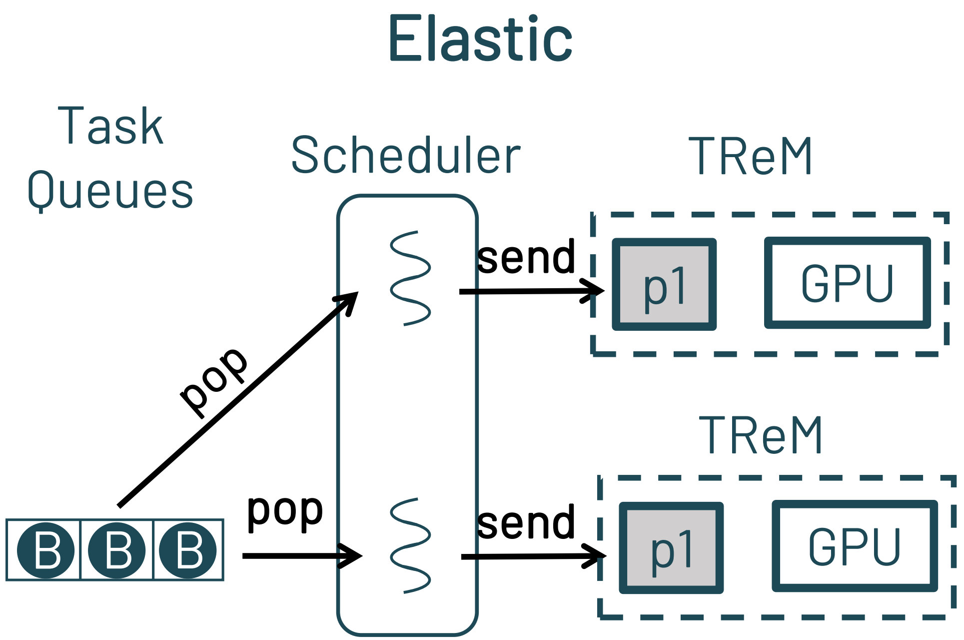
- When the user-facing load increases

- Wait for the currently executing user-facing
- Assigns the GPUs to new user-facing
- Postpones the execution of batch tasks

- Elastic assigns more GPUs for user-facing
  - In our example 1xGPU is sufficient
- Batch tasks are postponed

Incorporating TReM in Priority & Elastic



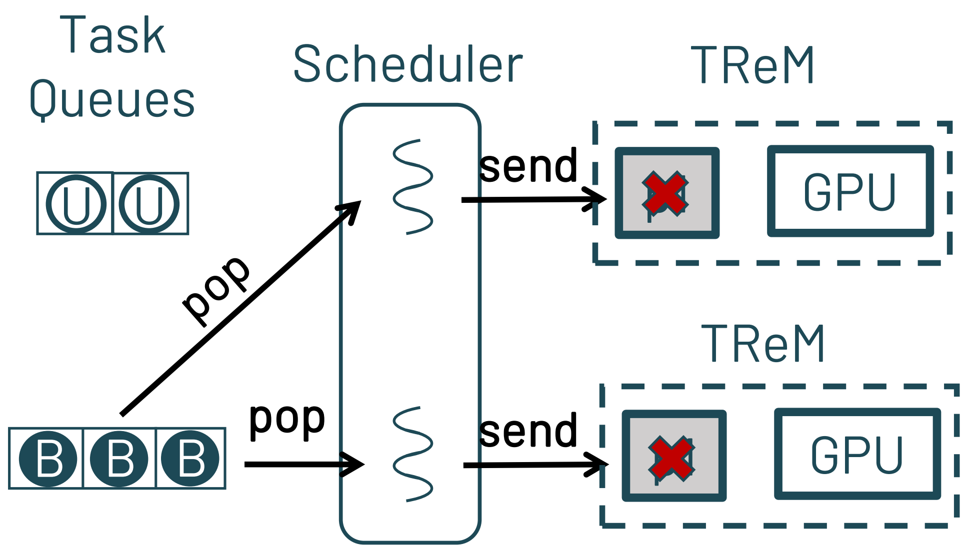


✓ Initially there are no user-facing tasks

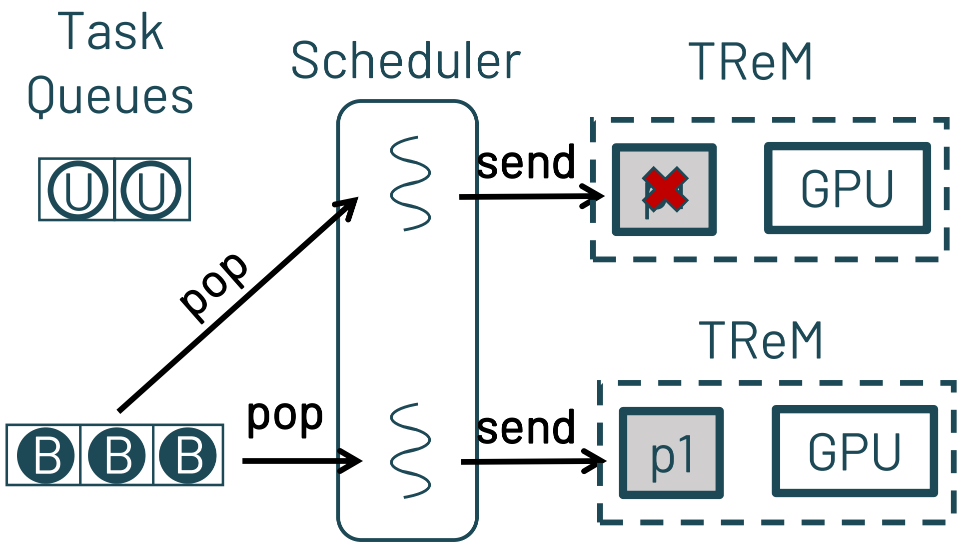
✓ All GPUs are provided to batch

Incorporating TReM in Priority & Elastic

Priority



Elastic

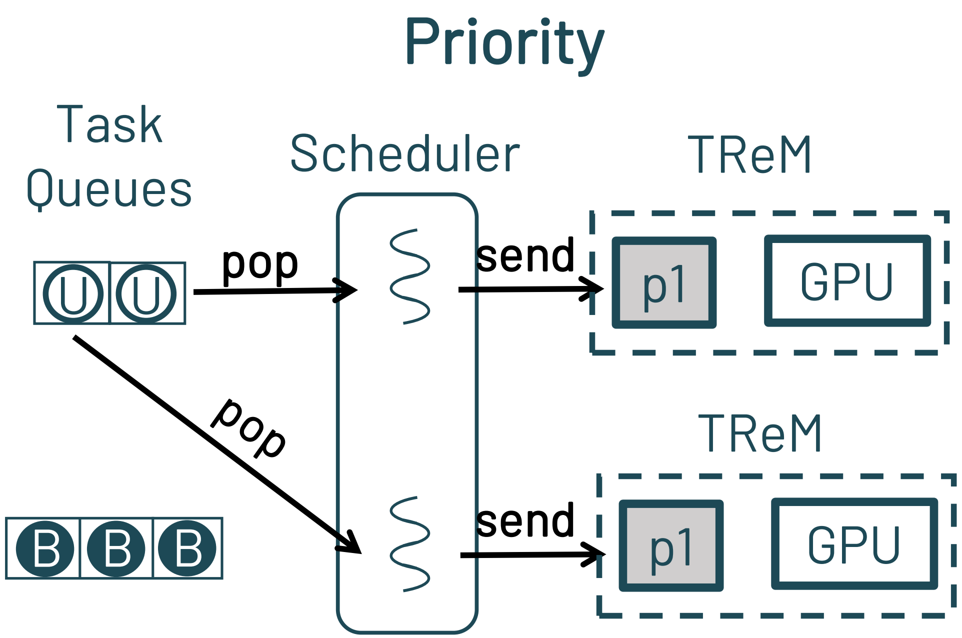


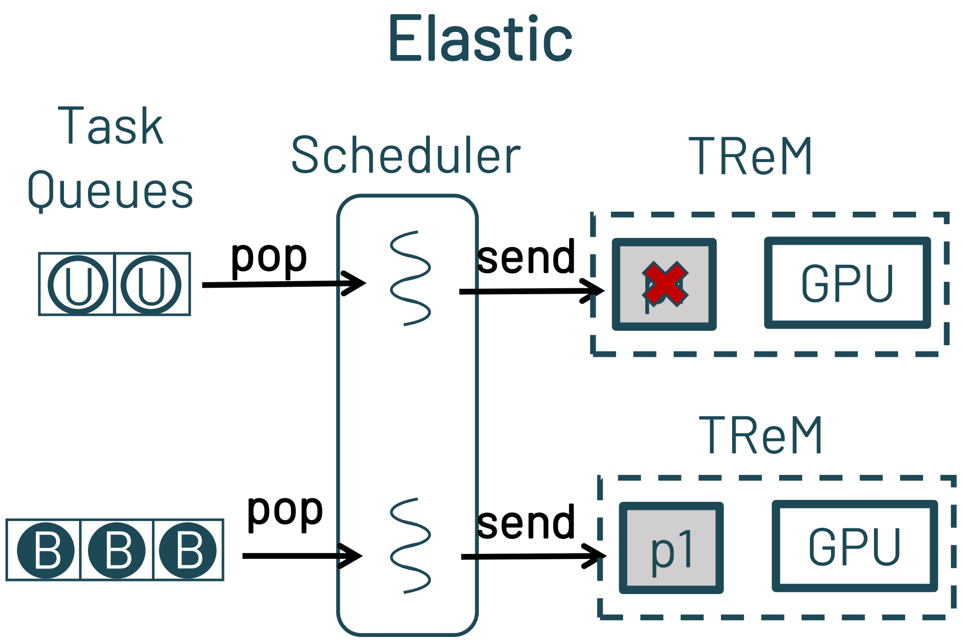
- ✓ Initially there are no user-facing tasks
  - ✓ All GPUs are provided to batch
- ✓ A burst of user-facing arrives

➢ Priority revokes both GPUs

➢ Elastic revokes one GPU

Incorporating TReM in Priority & Elastic





✓ Initially there are no user-facing tasks

✓ All GPUs are provided to batch

✓ A burst of user-facing arrives

➢ Both GPUs are provided to user-facing

➢ 1 GPU is provided to user-facing

Experimental Methodology

Testbed

- We use a server with:
  - Intel Xeon CPU E5-2630 v3 running at 2.40GHz
  - 128GB of DRAM
  - 4xNVIDIA P1000 GPUs (Pascal Architecture)
- Each GPU
  - Has 640 CUDA cores & 4GB of GDDR5
  - Connected with a 16 lanes PCIe gen3
- We use CUDA 9.0 to implement TReM

Workloads

- Micro-benchmarks
  - With a few tasks
  - To measure the overheads of TReM
- Datacenter-inspired synthetic workloads
  - With thousands of user-facing & batch tasks
  - To measure the performance of the overall system
- We use tasks from Rodinia 3.2 and NVIDIA SDK
  - SLA = 200ms
  - Tasks with execution time < SLA → user-facing
  - Tasks with execution time >> SLA → batch

user-facing

Tasks	AVG Exec. Time (ms)	Memory Footprint (MB)
Euclid	8	12
NW	38	44
Pathfinder	68	74
Monte Carlo	150	68
Lava MD	46000	1069
Hot Spot	130696	423
Gaussian	311000	1120

batch

Datacenter workloads

Workload specs	W 1	W 2
User-facing to batch ratio	50:50	80:20
User-facing job duration (mean)	5s	
Batch job duration (mean)	600s	
Total # of jobs	30	
Total # of tasks	1560	

- We implement a workload generator
  - *Mimics* traces from Google and Alibaba
  - Takes 3 parameters:
    1. Job duration → Pareto distribution
    2. Job inter-arrival time → Exponential distribution
    3. User-facing to batch job ratio → 50:50 (Alibaba), 80:20 (Google)
- We generate two workloads: W1 & W2

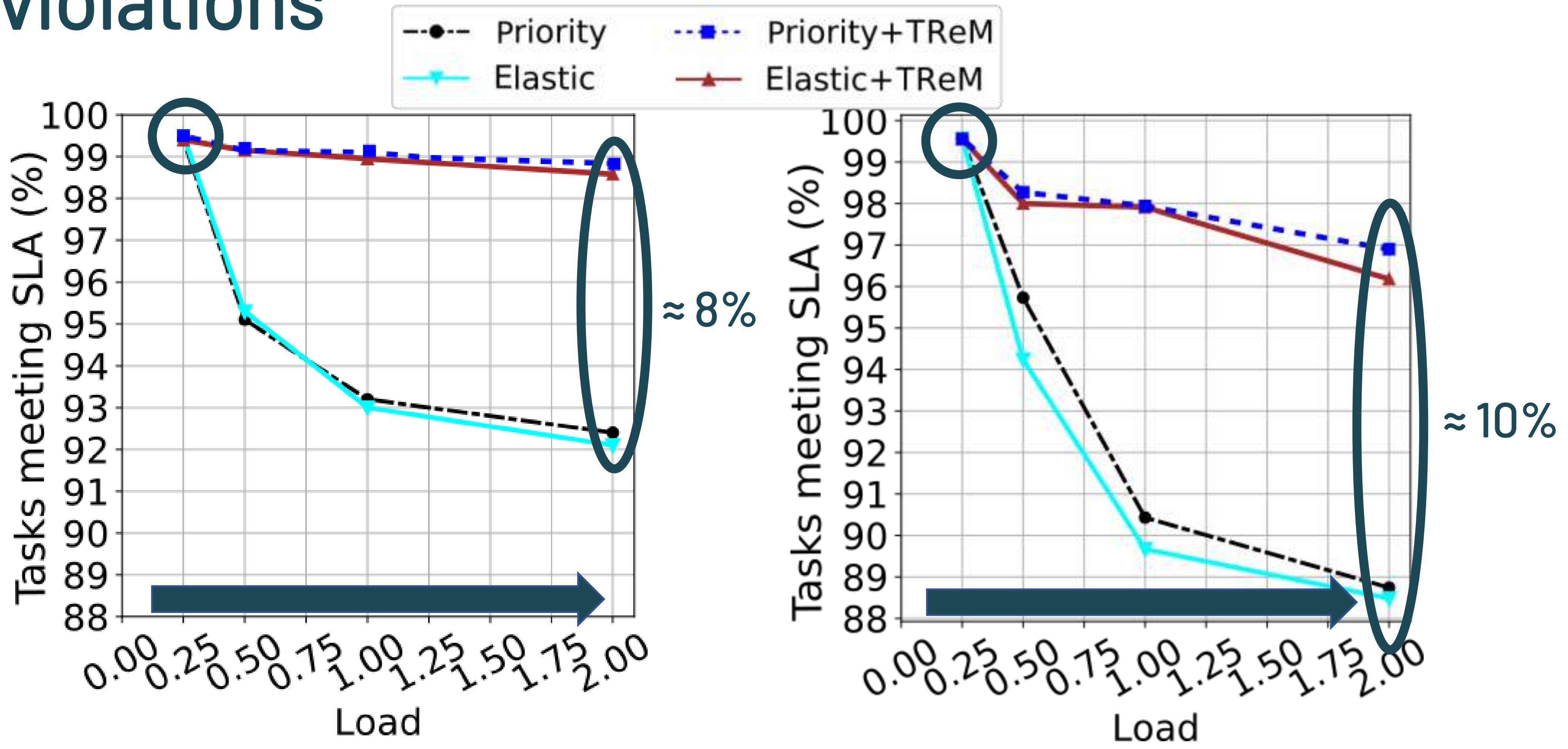
Datacenter workloads

Workload specs	W 1	W 2
User-facing to batch ratio	50:50	80:20
User-facing job duration (mean)	5s	
Batch job duration (mean)	600s	
Total # of jobs	30	
Total # of tasks	1560	
Load	0.25 - 2	

- We implement a workload generator
  - *Mimics* traces from Google and Alibaba
  - Takes 3 parameters:
    1. Job duration → Pareto distribution
    2. Job inter-arrival time → Exponential distribution
    3. User-facing to batch job ratio → 50:50 (Alibaba), 80:20 (Google)
- We generate two workloads: W1 & W2
- To emulate different Load
  - We use a scaling factor on the base inter-arrival mean
  - The scaling factor ranges from 0.25 (low load) to 2.0 (oversubscription)
    - Load 0.25 can fully utilize one GPU
    - Load 1 can fully utilize four GPUs

Experimental Analysis

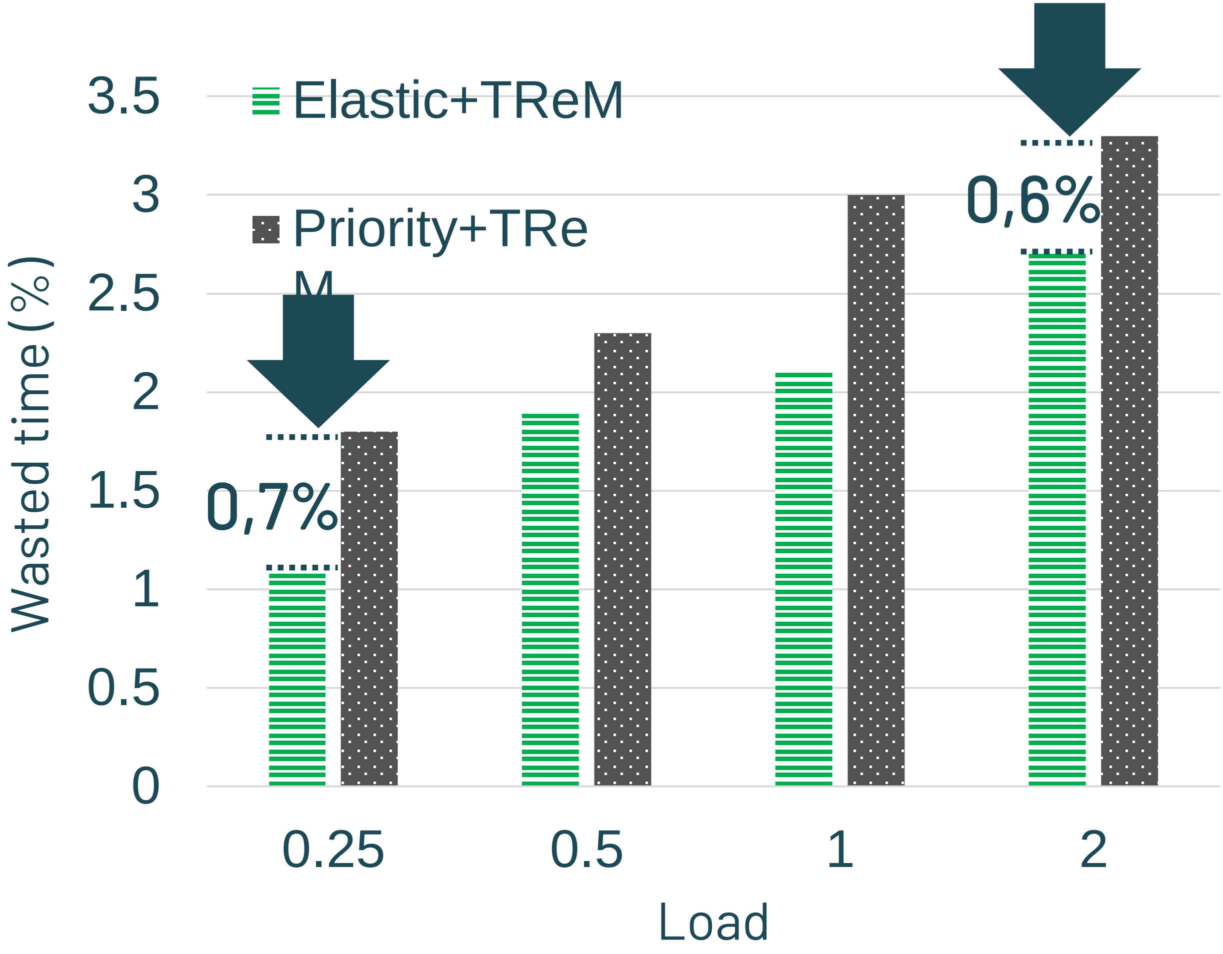
SLA violations



W1: 50% user-facing – 50% batch

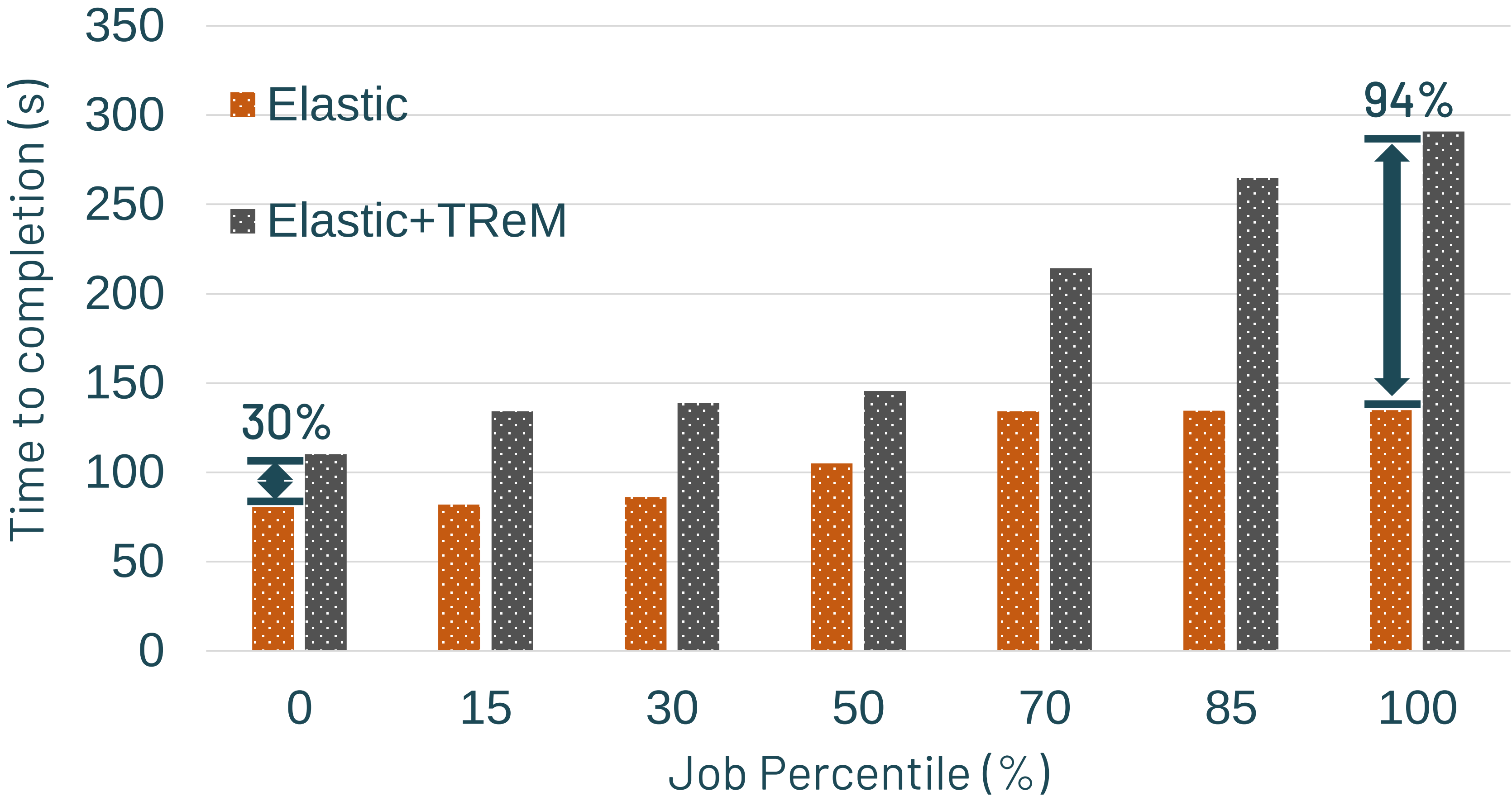
W2: 80% user-facing – 20% batch

Lost work due to revocations



- Both policies minimize wasted time
  - Revoke more recently started tasks
- Elastic minimize more wasted time
  - Uses minimum # GPUs for user-facing

PDF with batch job duration



Compare revocation mechanisms

Process kill:

- \+ Constant latency
- – High latency

Compare revocation mechanisms

		Latency (ms)		
Kernel dimensions	Total threads	Process kill	asm(exit)	asm(trap)
Kernel <16,16>	256	3000	130	
Kernel <32,32>	1024	3000	195	
Kernel <64,64>	4096	3000	600	
Kernel <128,128>	16384	3000	1430	

asm(exit):

- Variable latency
- High latency

Compare revocation mechanisms

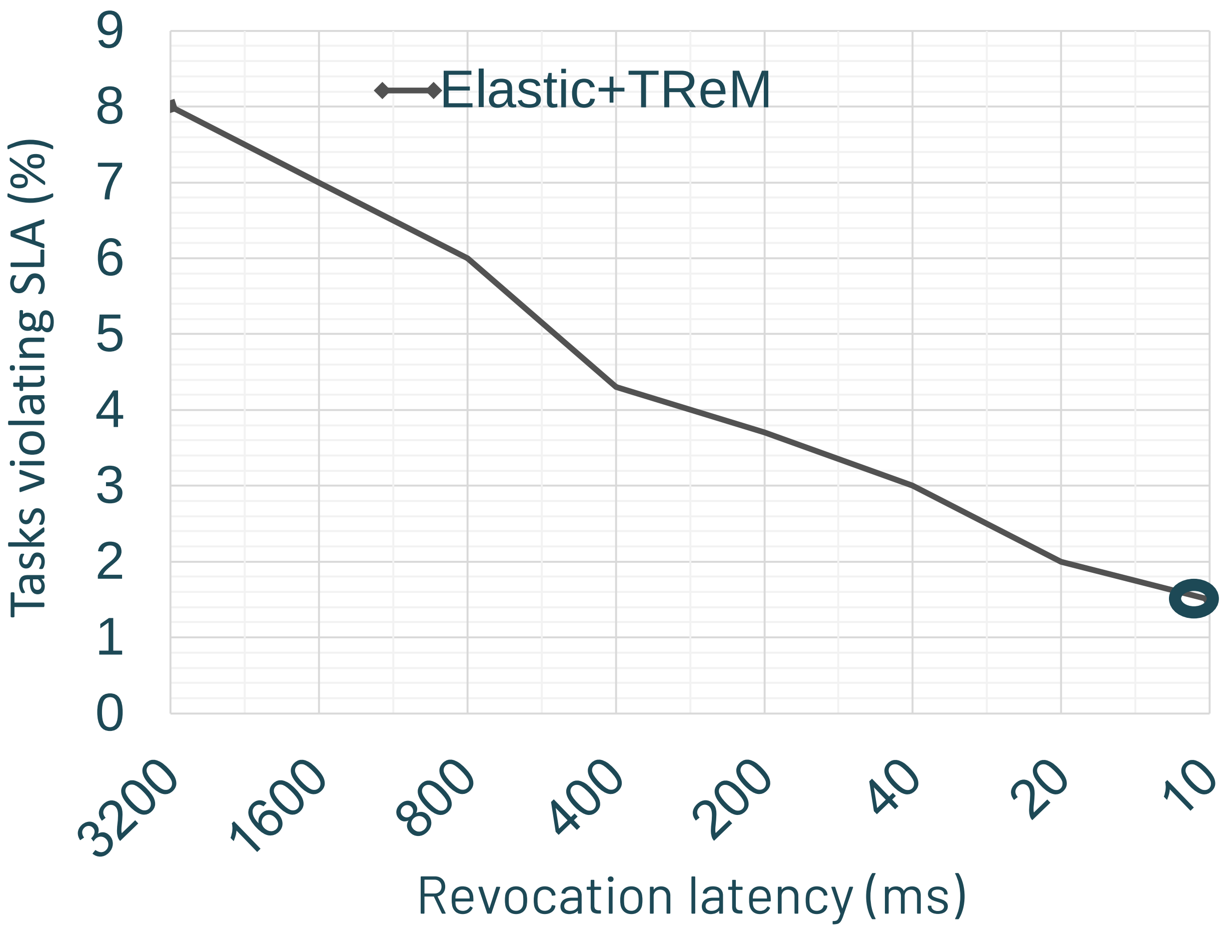
		Latency (ms)		
Kernel dimensions	Total threads	Process kill	asm(exit)	asm(trap)
Kernel <16,16>	256	3000	130	22
Kernel <32,32>	1024	3000	195	22
Kernel <64,64>	4096	3000	600	22
Kernel <128,128>	16384	3000	1430	22

✓ TReM uses asm(trap)

asm(trap):

- + Constant latency
- + Low latency

SLA violations vs. Revocation latency



Conclusions

TReM: A Task Revocation Mechanism for GPUs

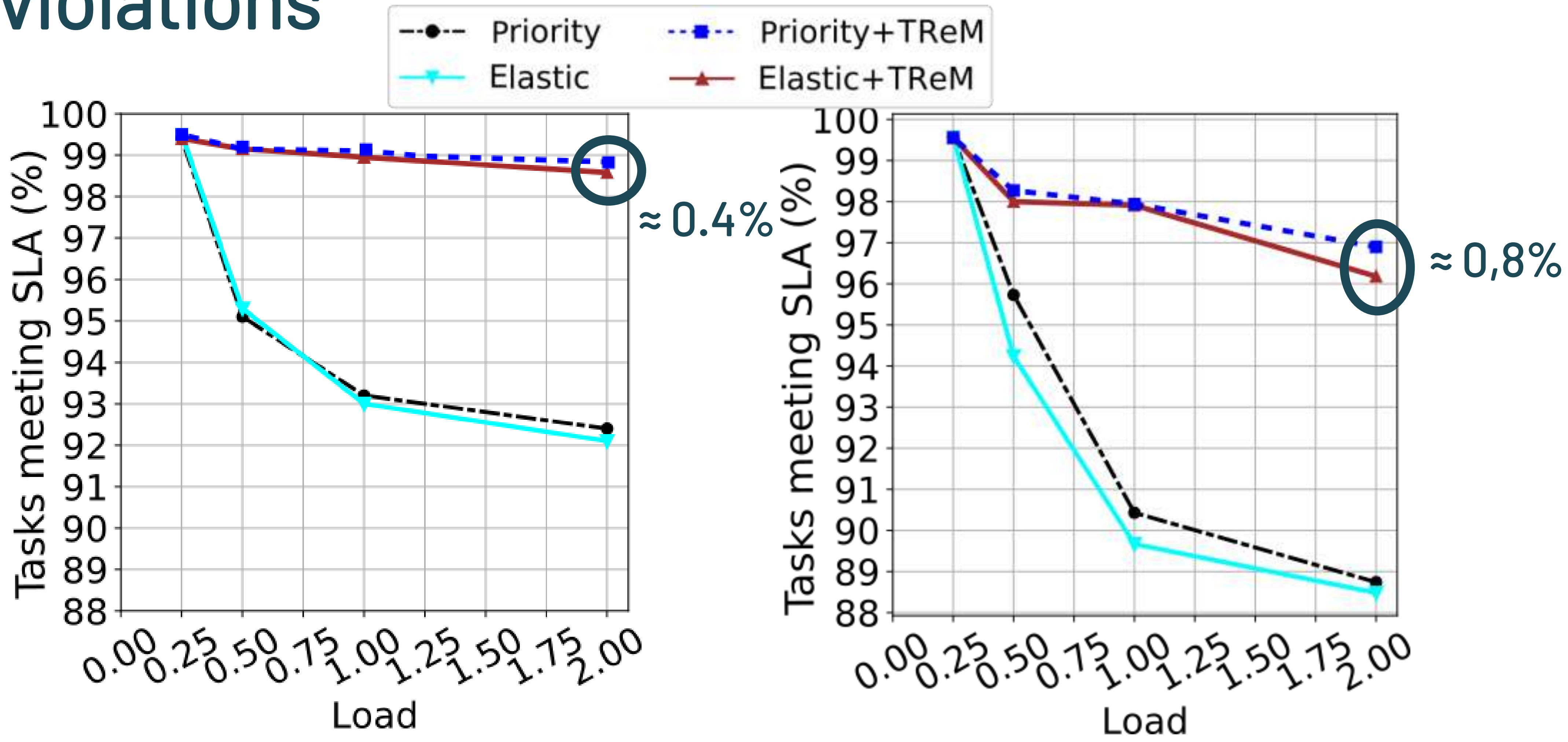
- To provide QoS under GPU sharing
  - We need a preemption or revocation mechanism
- **BUT** this mechanism should have constant and low latency (<<SLA)
- **TReM** is a **T**ask **Re**vocation **M**echanism
  - Stops a kernel at any point of its execution without storing state
  - Replays the revoked task later
- TReM revocation latency is 22ms
- TReM + Elastic
  - Ensure the SLA for 8% more user-facing tasks compared to Priority
  - Limits the lost work due to revocations to 2,1% on average

Thank you

Questions?

Manos Pavlidakis
manospavl@ics.forth.gr

SLA violations



W1: 50% user-facing – 50% batch

W2: 80% user-facing – 20% batch