Berkeley Architecture Research

SLICE

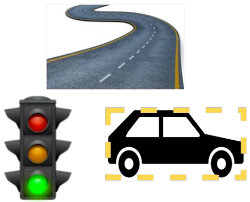
MoCA: Memory-Centric, Adaptive Execution for Multi-Tenant Deep Neural Networks

Seah Kim, Hasan Genc, Vadim Nikiforov,
Krste Asanovic, Borivoje Nikolic, Yakun Sophia Shao

Berkeley
UNIVERSITY OF CALIFORNIA

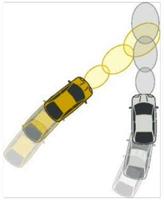
Multi-tenancy for DNN

Multiple tasks share system resources

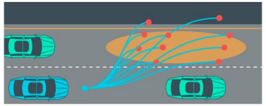


Detections

Tracking



Motion Prediction

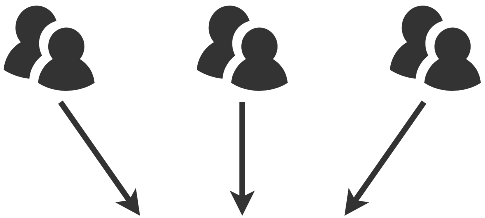


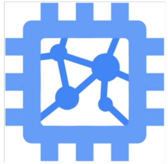
Trajectory Planning

Autonomous Vehicle

@ Edge

Clients

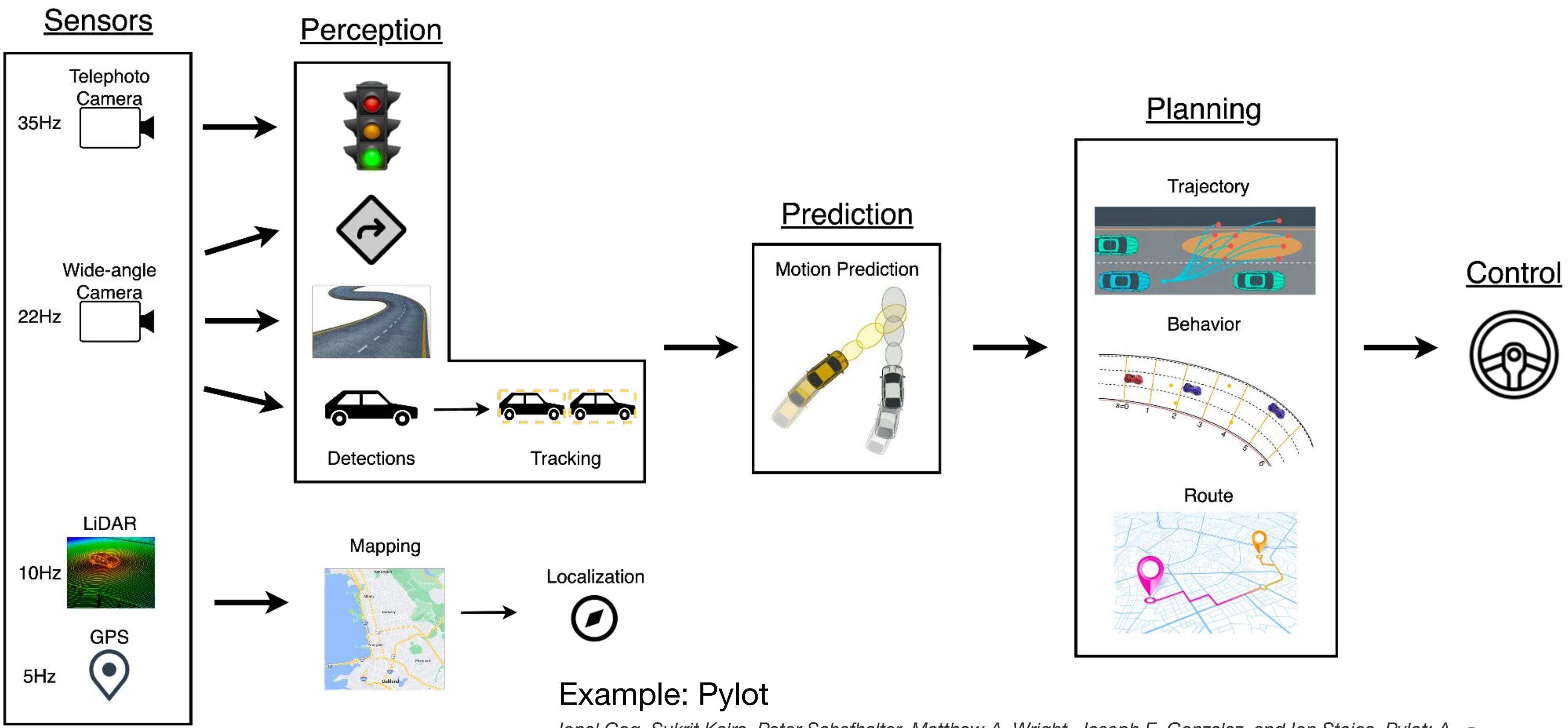




Cloud Accelerator

@ Cloud

Multi-tenancy Example

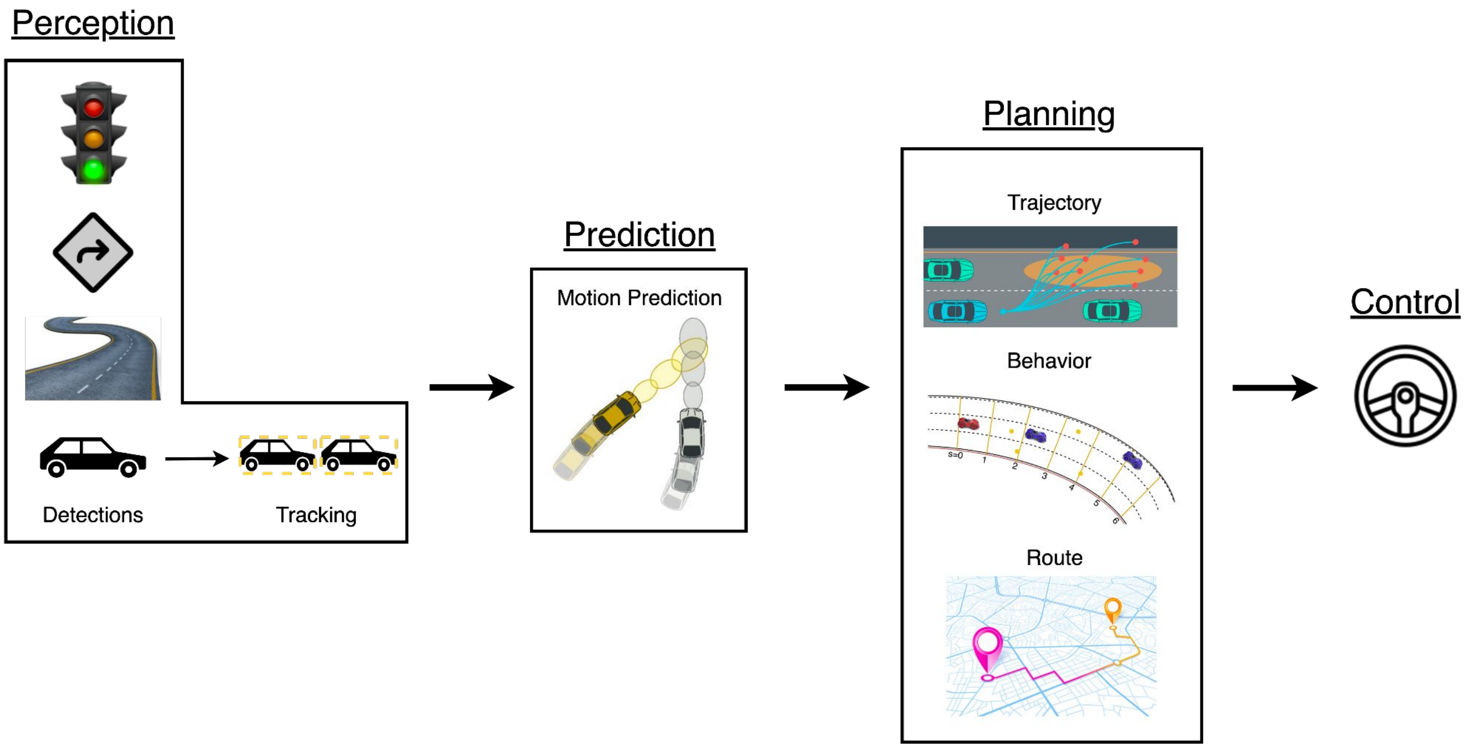


Example: Pylot

Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A. Wright, Joseph E. Gonzalez, and Ion Stoica. Pylot: A Modular Platform for Exploring Latency-Accuracy Tradeoffs in Autonomous Vehicles, ICRA 2021

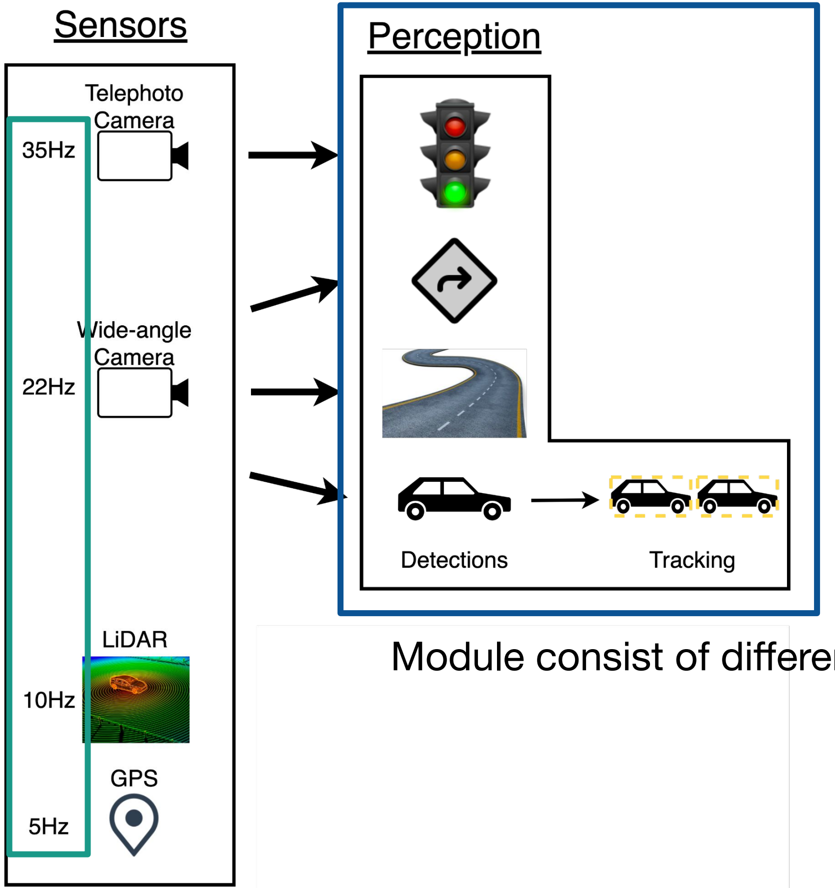
Multi-tenancy Example

- Consists of multiple different modules
  - Perception, Prediction, Planning, Control



Multi-tenancy Example

- Multiple tasks exists in a module

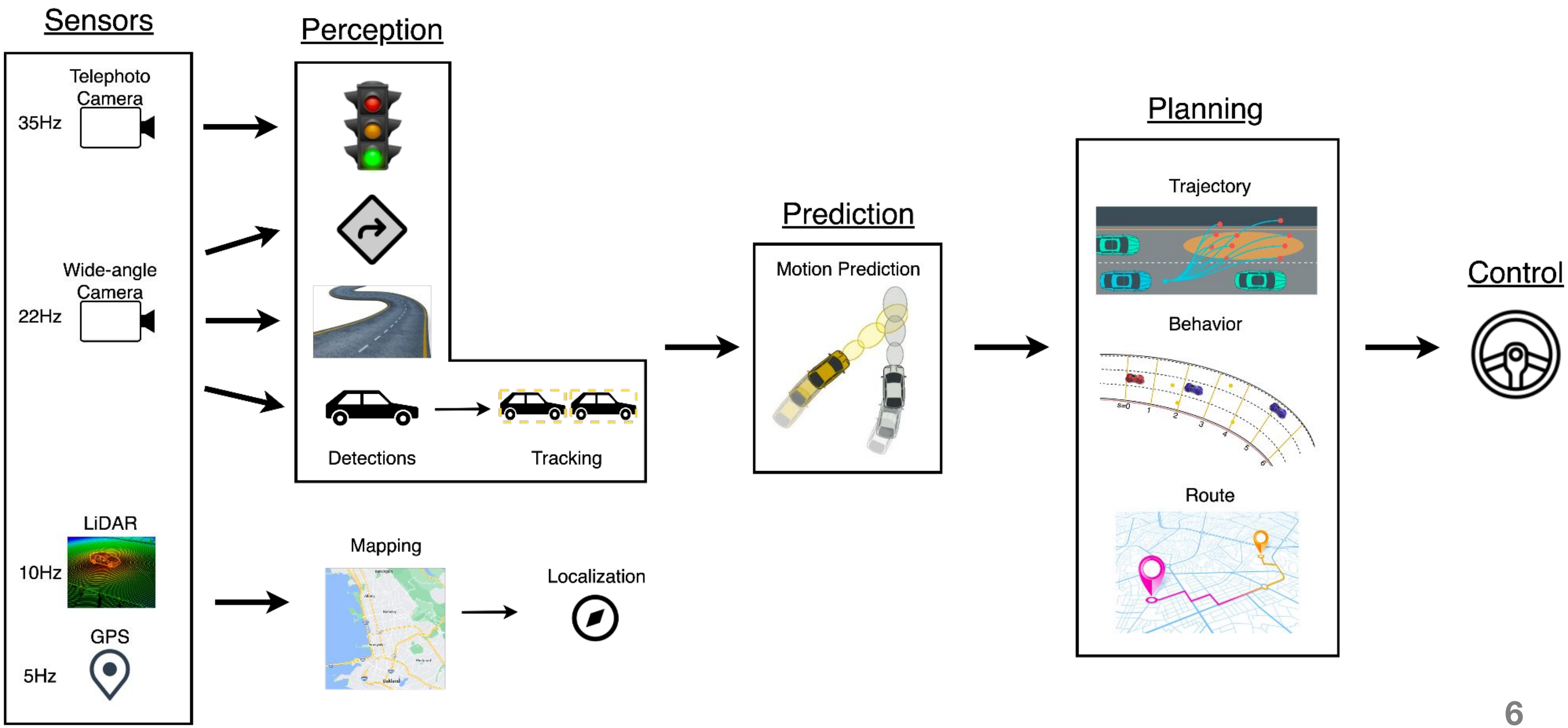


Models with different processing rate
-> different target deadline

Module consist of different models

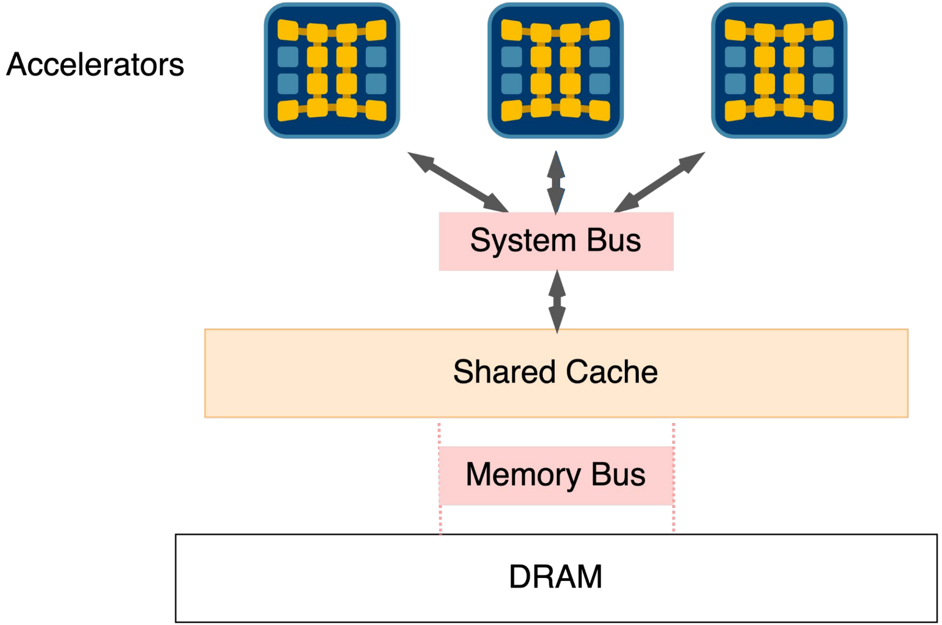
Multi-tenancy Example

- Need multi-tenancy support by co-running multiple models together



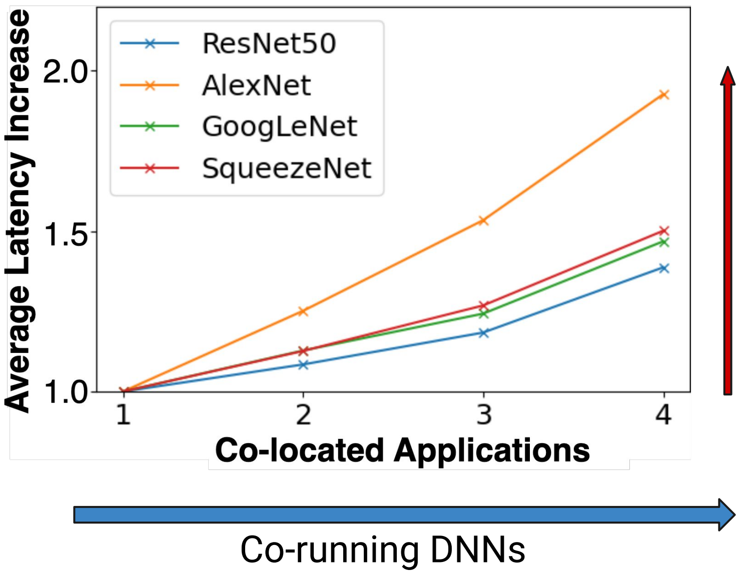
Challenge 1 - Interference

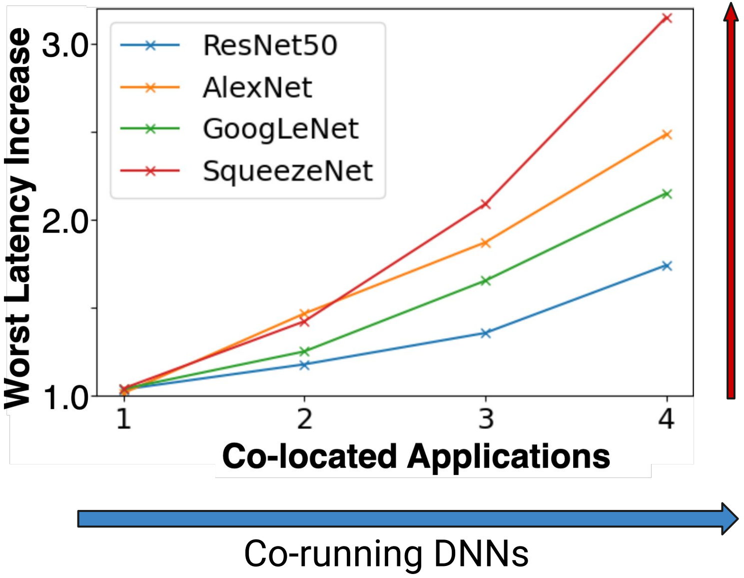
- Performance degradation due to shared resource contention
  - LLC, DRAM, IO, System bus



Challenge 1 - Interference

- Increased system-level interference cause significant performance degradation

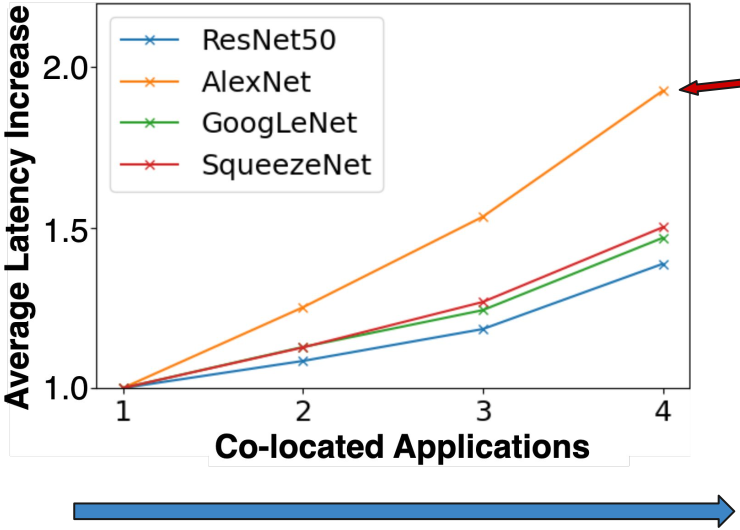




Up to 3x latency increase!

Challenge 1 - Interference

- Increased system-level interference cause significant performance degradation



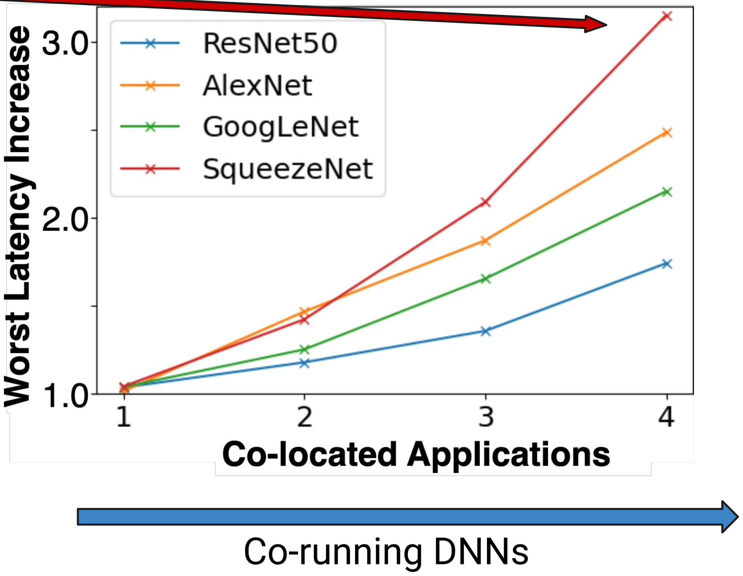
Co-running DNNs

Memory intensive FC layers
-> interference caused by memory contention

Challenge 1 - Interference

- Increased system-level interference cause significant performance degradation

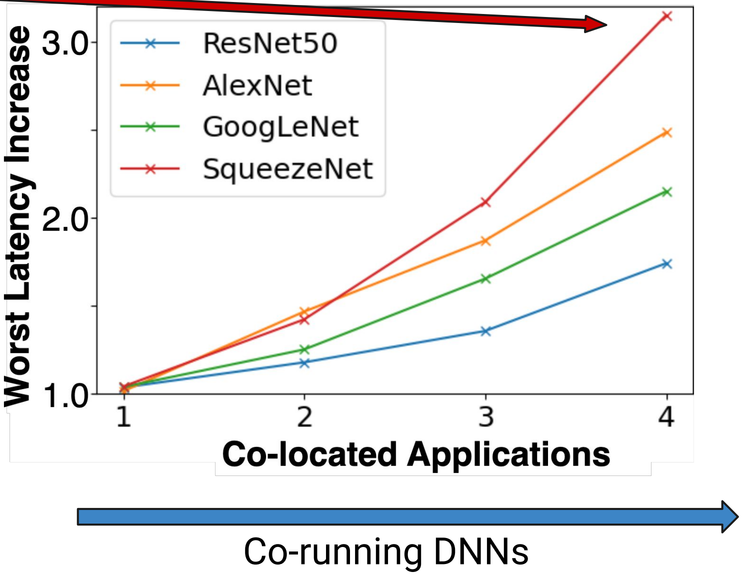
Short running network
-> depends on co-located workload characteristics of using shared resources



Challenge 1 - Interference

- Increased system-level interference cause significant performance degradation

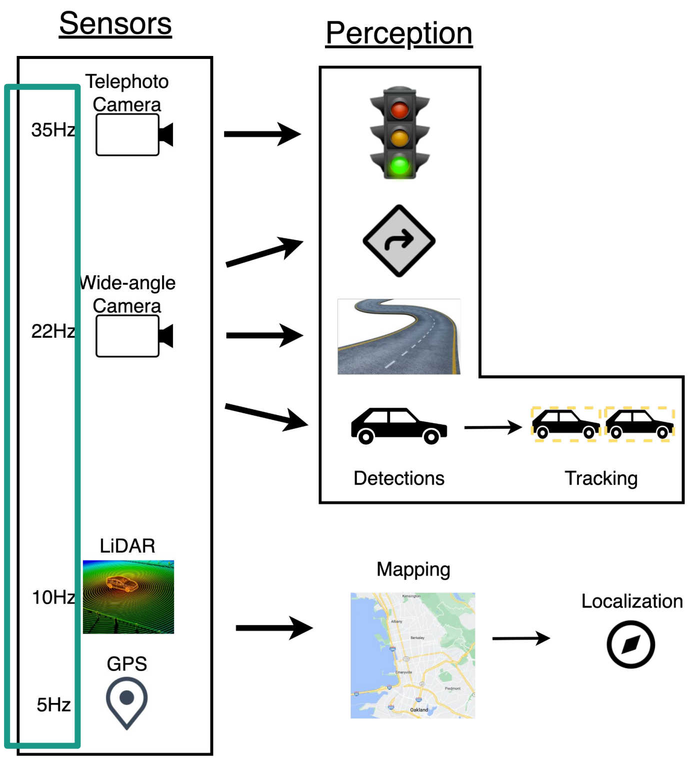
Short running network
-> depends on co-located workload
characteristics of using shared resources



Need runtime contention detection

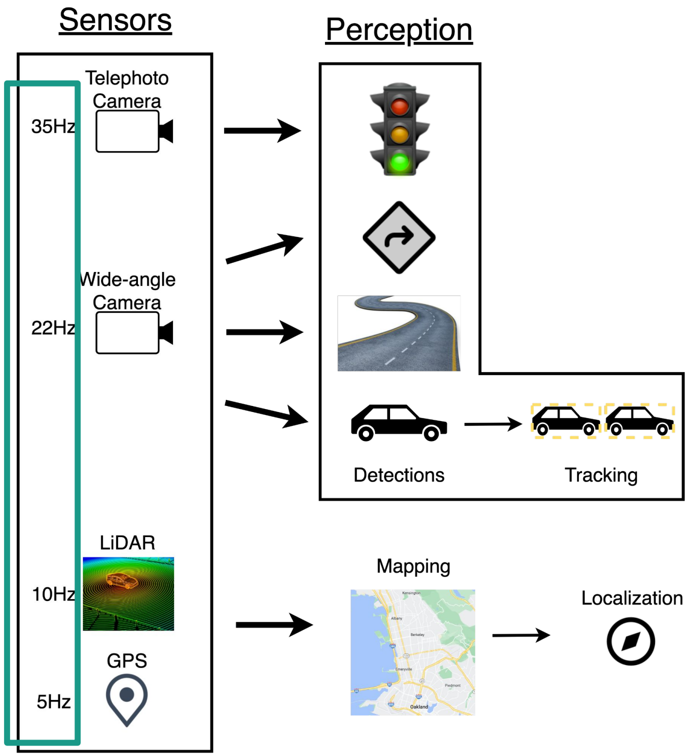
Need management and dynamic manipulation of shared resources

Challenge 2 - Scheduling



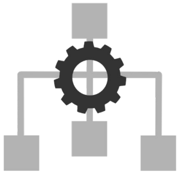
Different target latency: **target-aware**

Challenge 2 - Scheduling



Different target latency: **target-aware**

Varied user-given priority level: **priority-aware**



Infrastructure

Monitoring

Category	Priorities	Percentage of jobs
Free	0 or 1	33.63%
Other	2 to 8	56.30%
Production	9	9.91%
Monitoring	10	0.13%
Infrastructure	11	0.002%

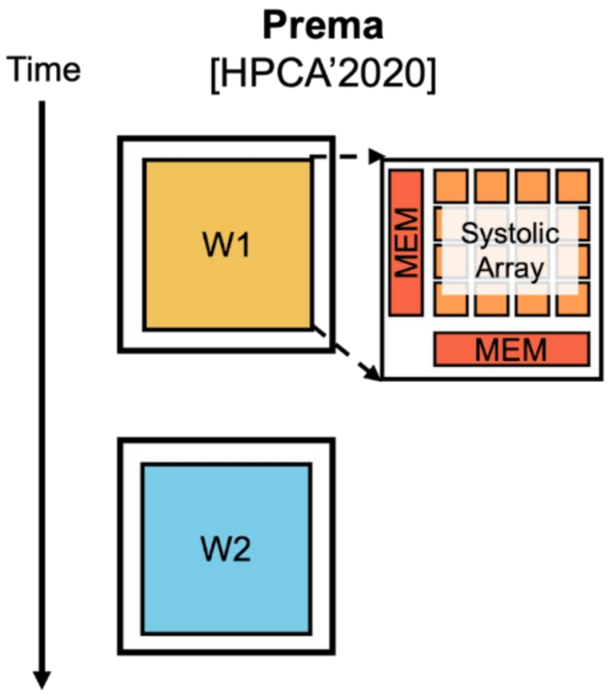
Others

P. Minet, É. Renault, I. Khoufi and S. Boumerdassi, "Analyzing Traces from a Google Data Center," 2018 IWCMC

Prior Works in Multi-tenancy

- Two challenges in Multi-tenancy execution
  - System level interference
  - Target & Priority aware scheduling
- How did prior works address the challenges?
- MoCA's advantage over prior works?

Prior Works in Multi-tenancy

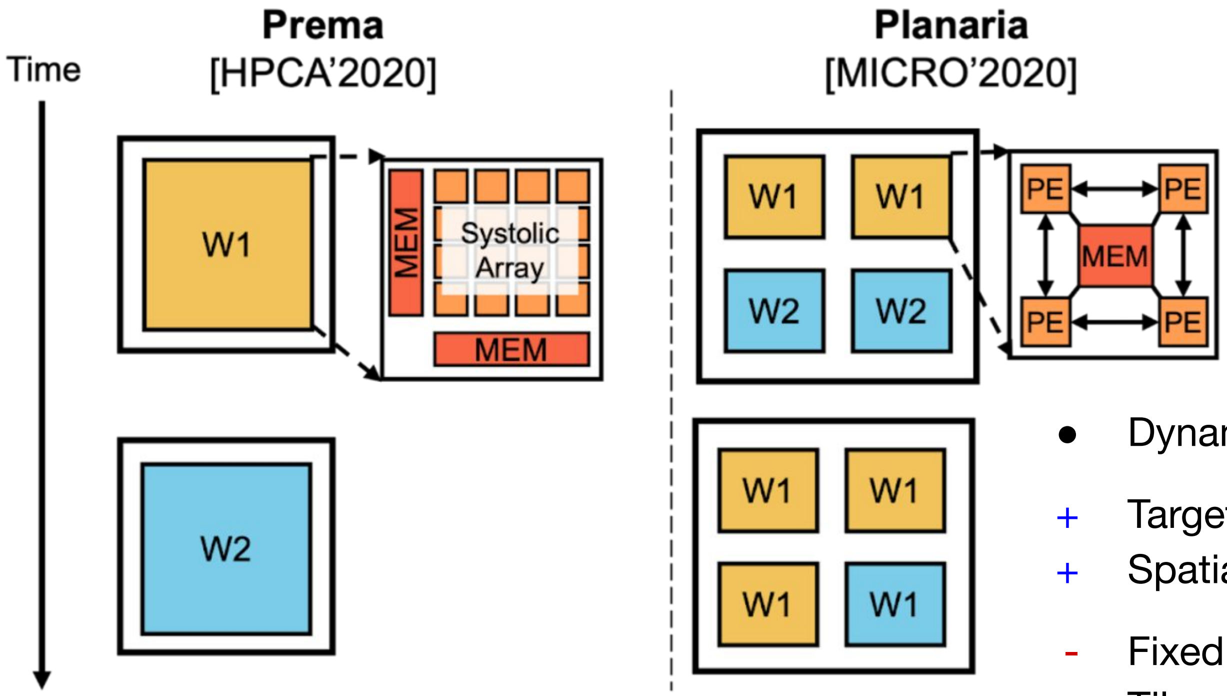


- Time multiplexing workloads using preemption

+ Target, priority aware scheduler

- No spatial co-location

Prior Works in Multi-tenancy

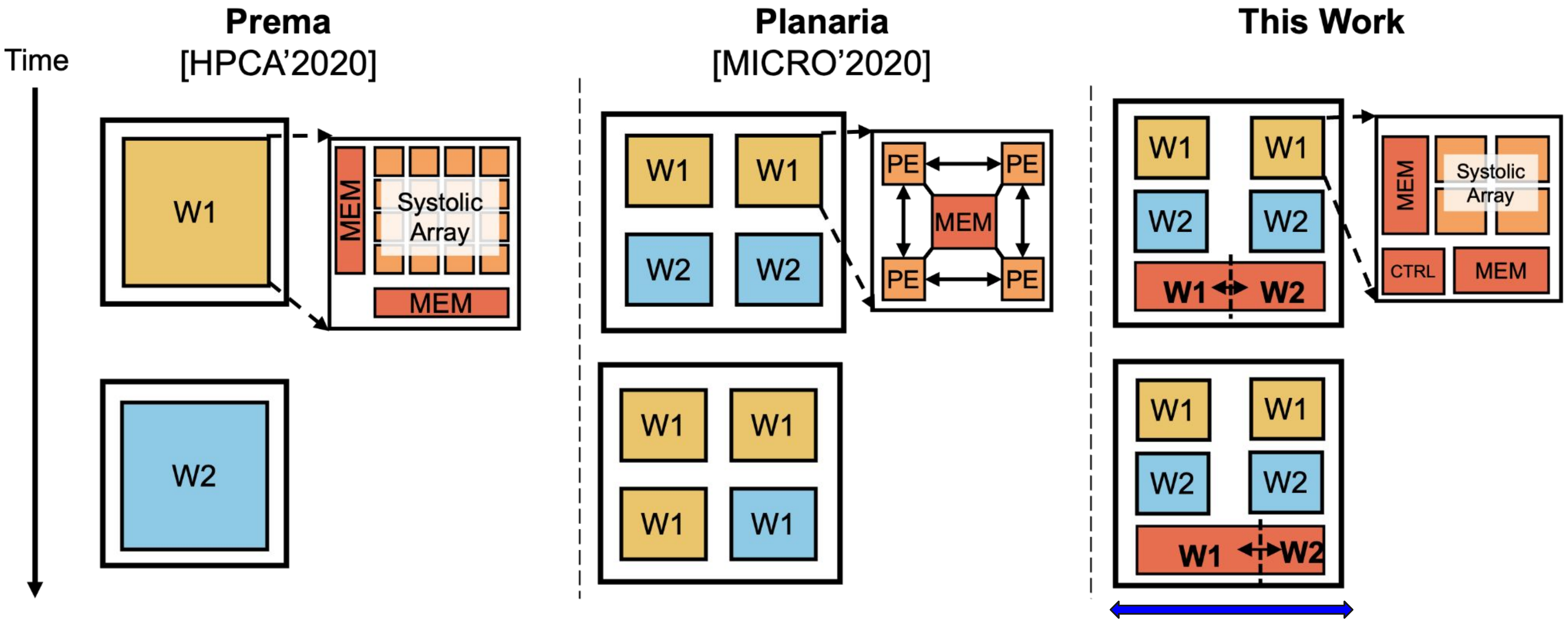


- Dynamic compute resource partitioning

\+ Target, priority aware scheduler
\+ Spatial co-location

\- Fixed compute-to-memory ratio
\- Tile granularity repartitioning (~1M cycles thread migration overhead)

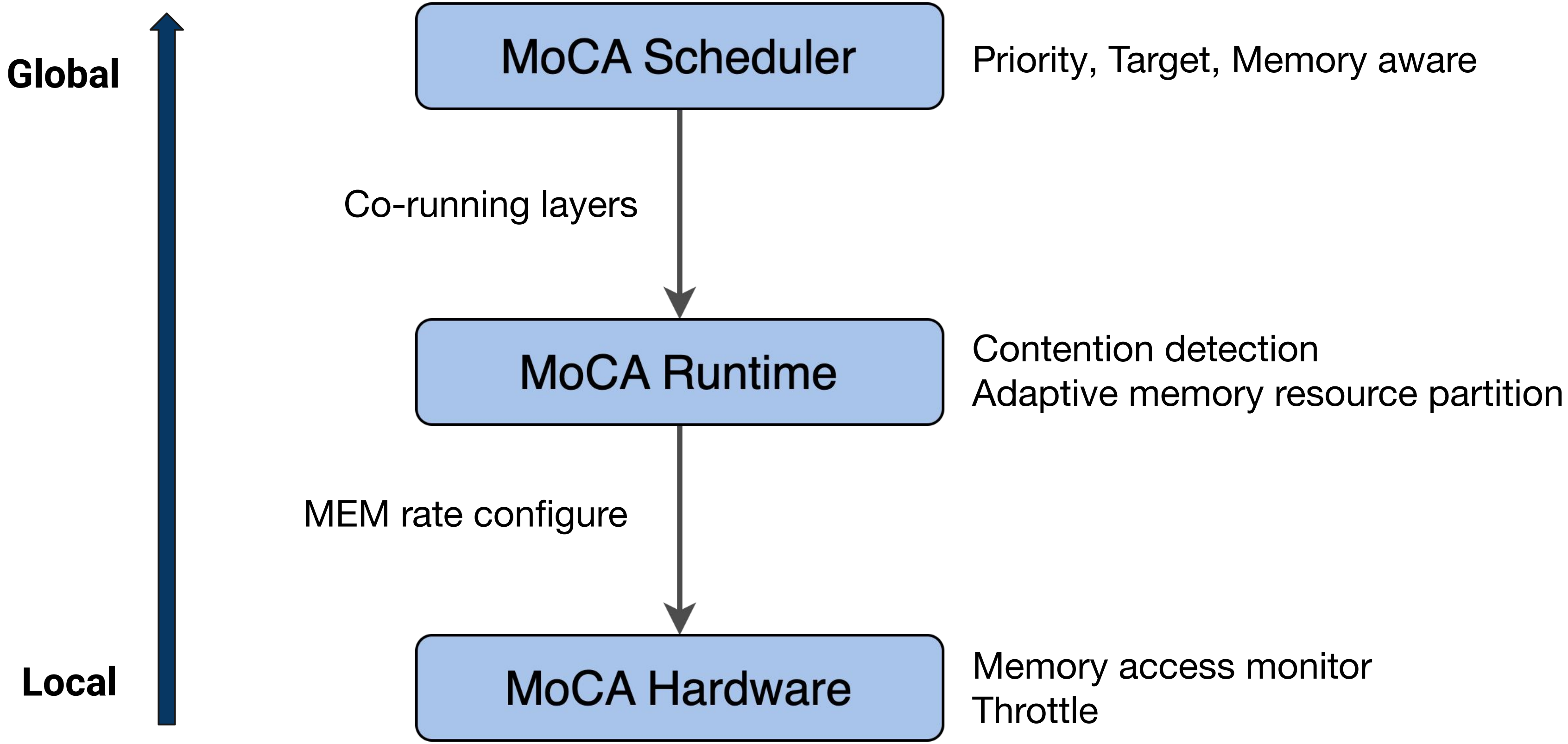
Prior Works in Multi-tenancy



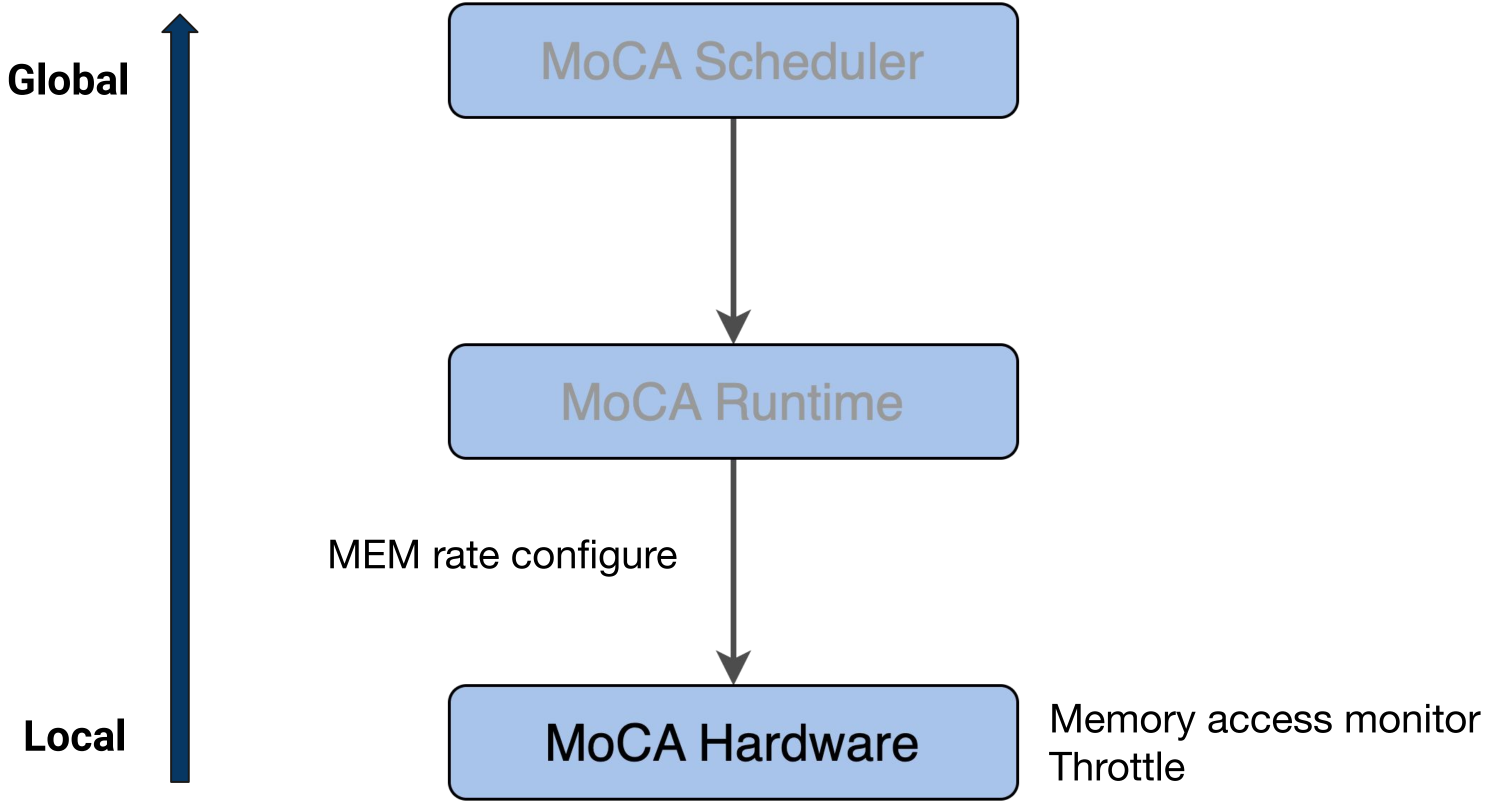
Dynamically partitioning **memory** resources

Index

MoCA's Full-Stack Implementation

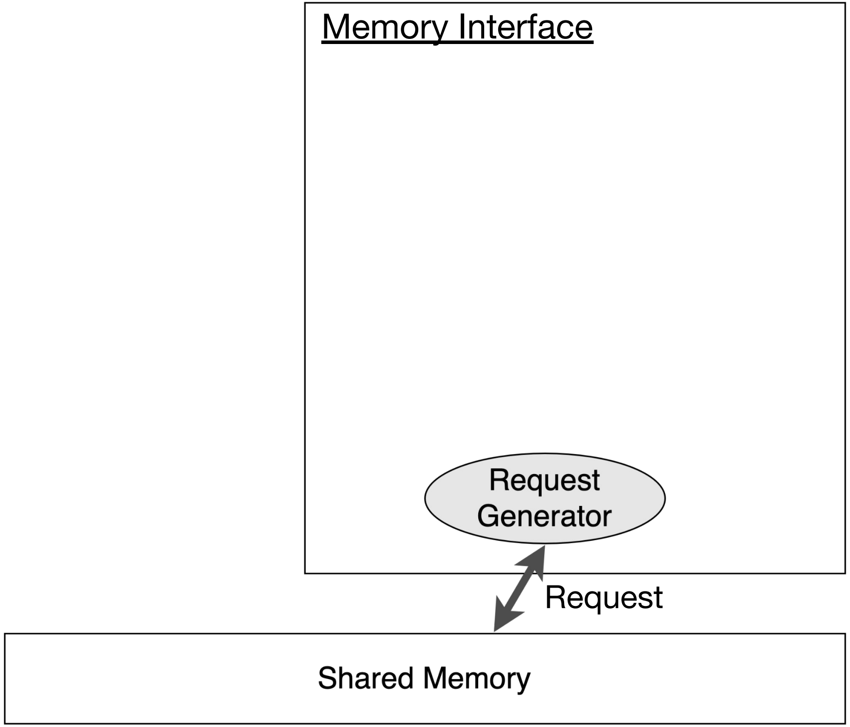


MoCA's Full-Stack Implementation



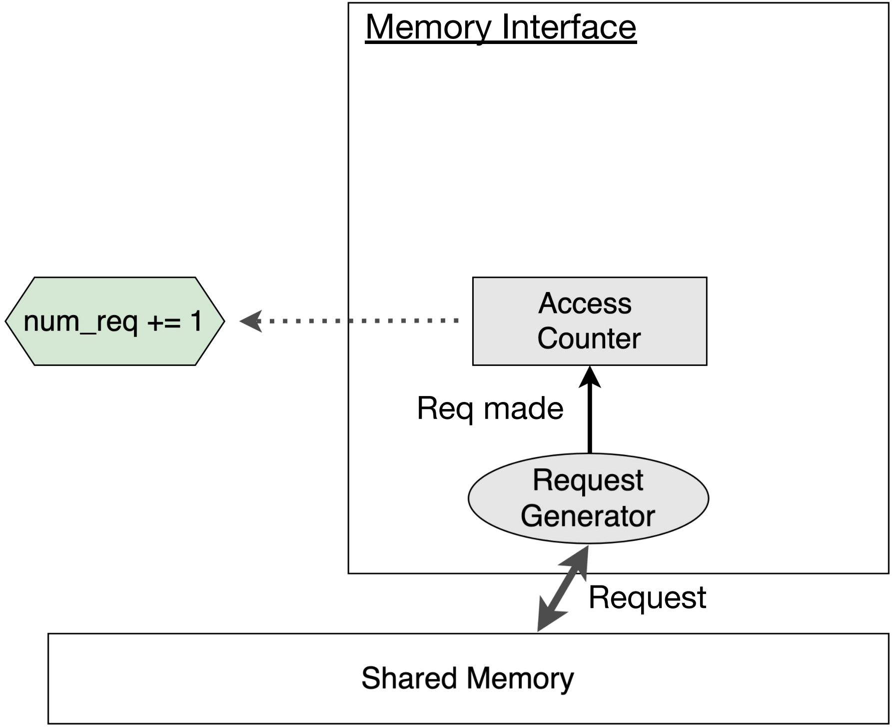
MoCA Hardware

- Inputs from Runtime: *window, Threshold*
- Implemented inside Memory Interface
  - Self-contained module



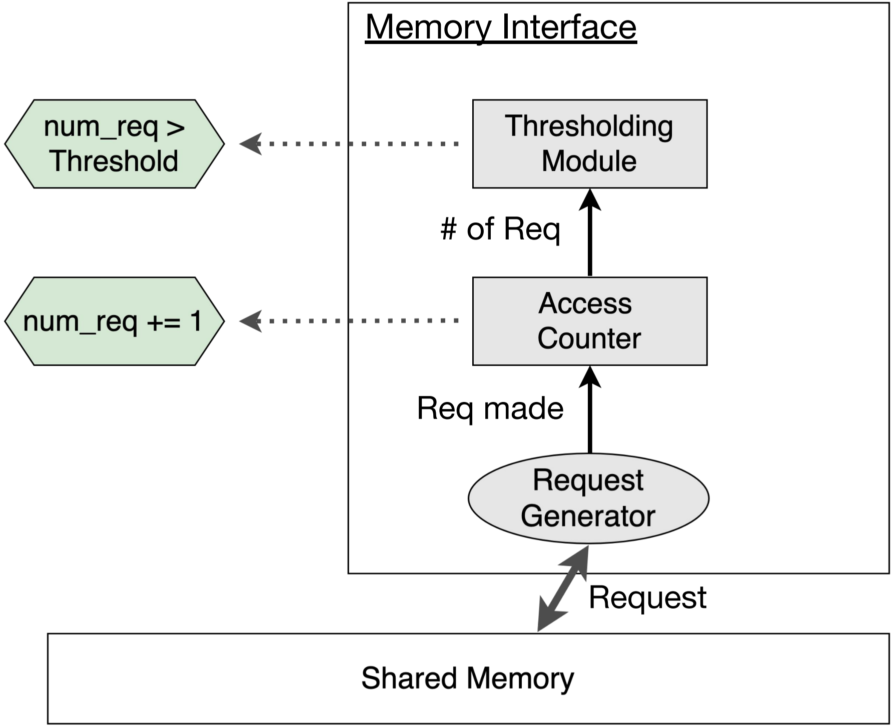
MoCA Hardware

- Inputs from Runtime: *window, Threshold*
- Implemented inside Memory Interface
  - Self-contained module



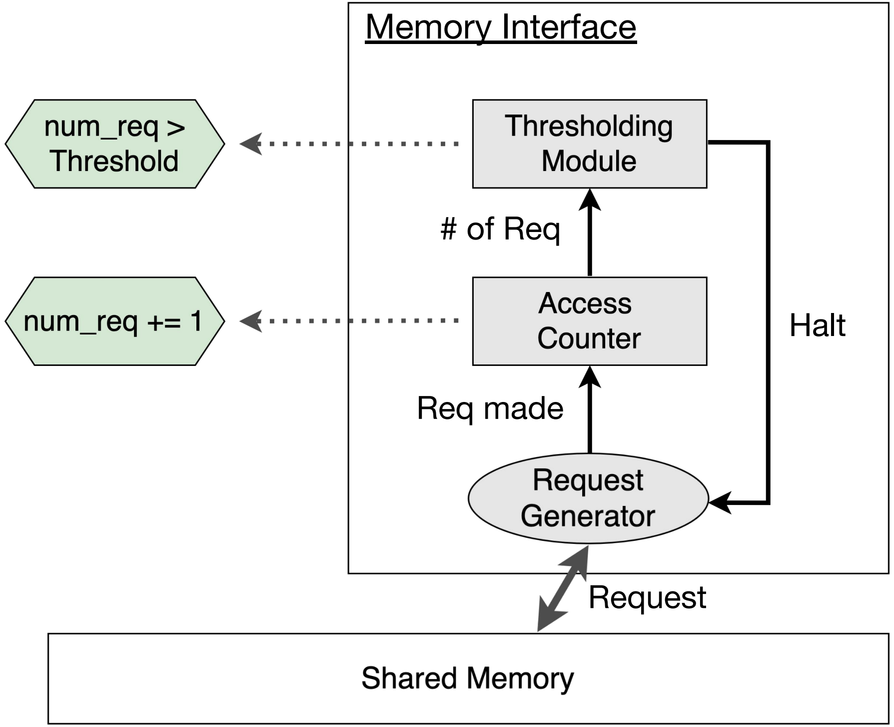
MoCA Hardware

- Inputs from Runtime: *window, Threshold*
- Implemented inside Memory Interface
  - Self-contained module
- Controls Mem req rate using 2 params
  - Monitoring time "*window*"
  - *# request "Threshold"* per time *window*

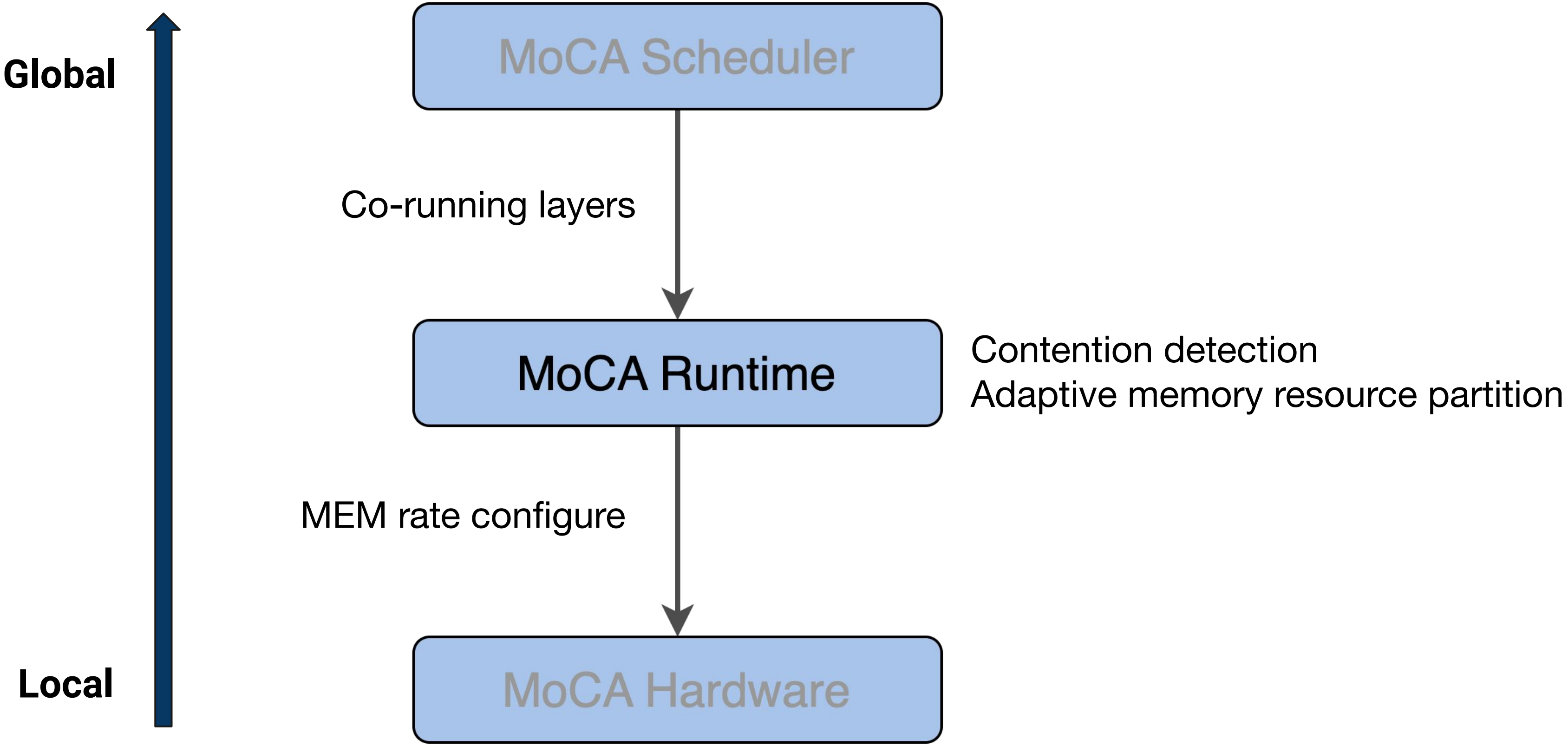


MoCA Hardware

- Inputs from Runtime: *window, Threshold*
- Implemented inside Memory Interface
  - Self-contained module
- Controls Mem req rate using 2 params
  - Monitoring time "*window*"
  - *# request "Threshold"* per time *window*
- Generates ld/st to meet configured rate
  - Halt if it goes over

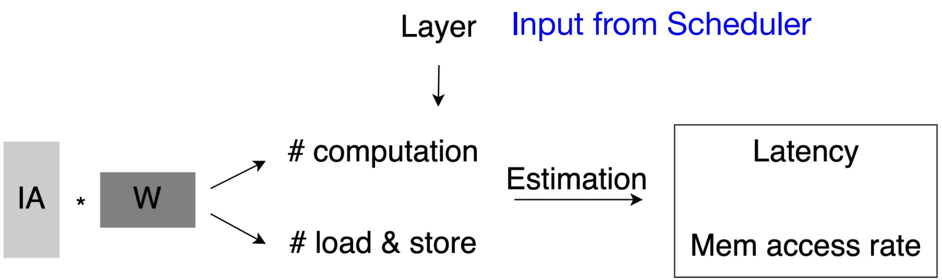


MoCA's Full-Stack Implementation

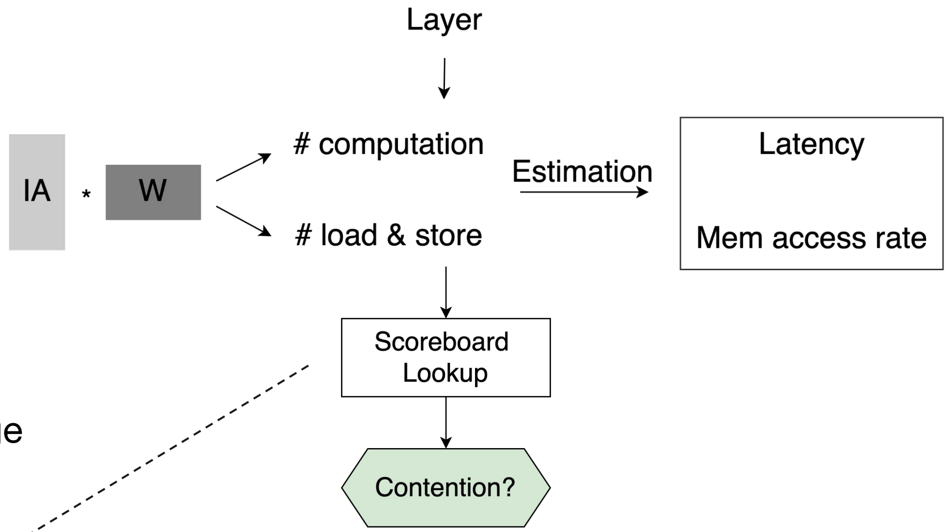


MoCA Runtime

- Leverage DNN regularity
- Calculates latency estimate of a layer
  - Using # of computation, # of ld/st
- Calculates required Mem access rate



MoCA Runtime

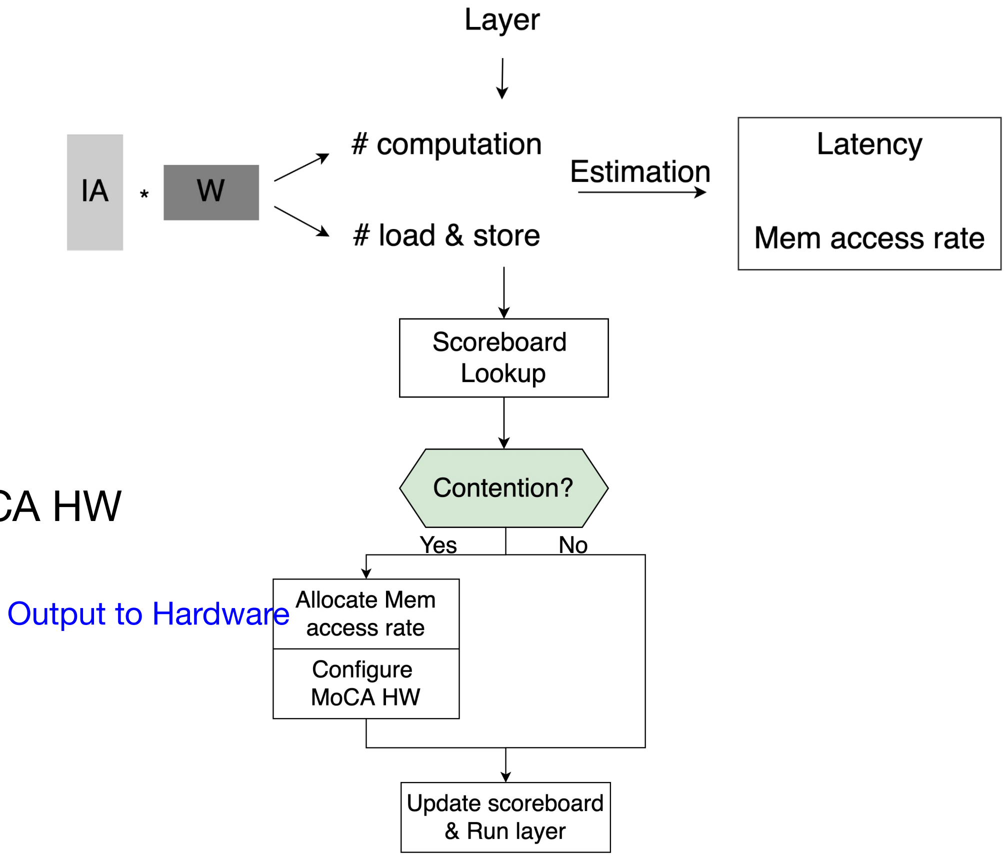


- Runtime contention detection
  - Sum up all required Mem access rate
  - Detect contention when bandwidth usage exceeds available bandwidth

Global Scoreboard			
Task 1	**Task 2**	**Task 3**	...
BW_1	BW_2	BW_3	...
Score_1	Score_2	Score_3	...

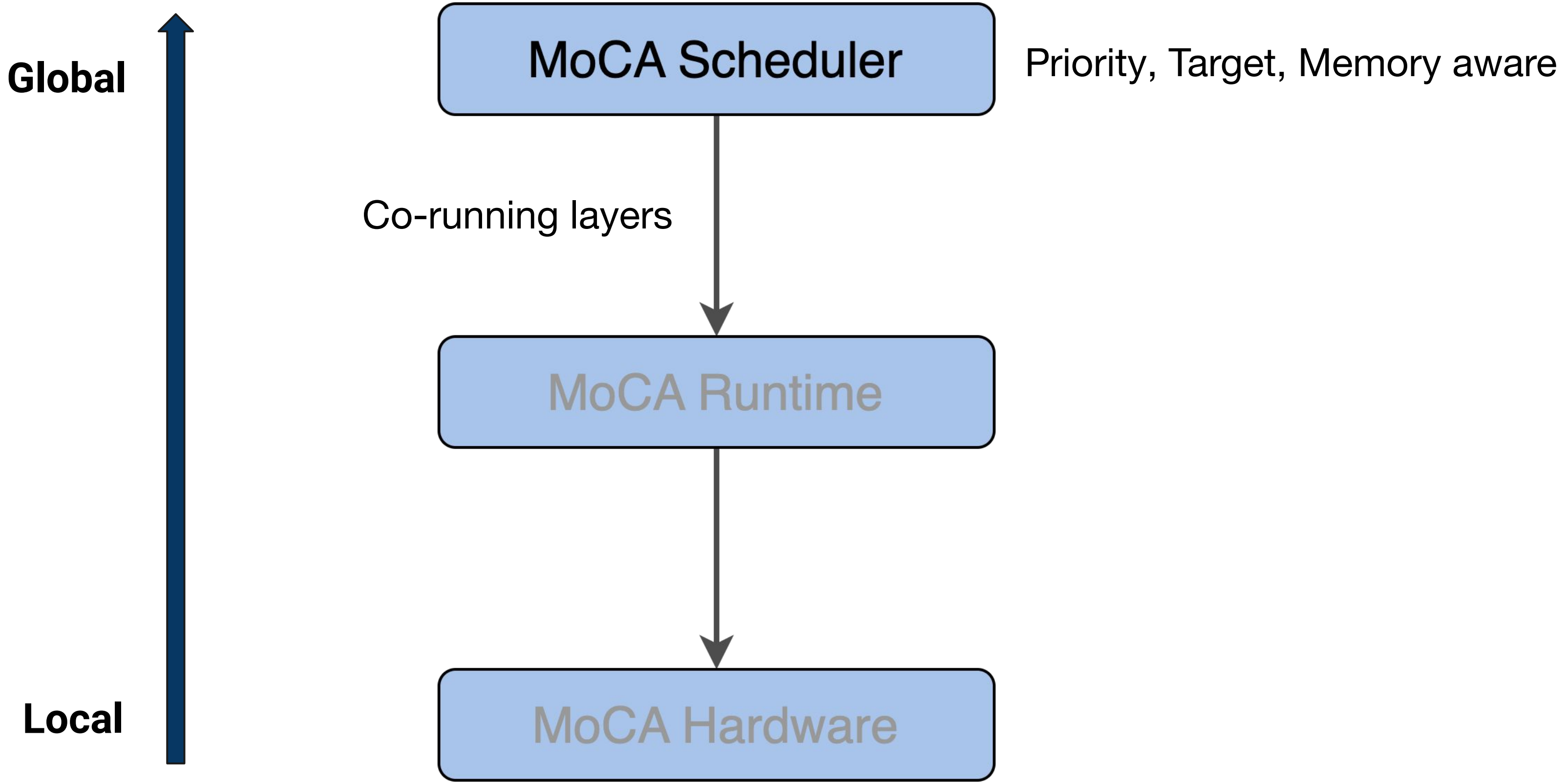
compare(System_BW, Σ BW_i)

MoCA Runtime



- If contention detected, configure MoCA HW
  - Monitoring time “*window*”
  - *“# Req”* per time *window*
- Dynamic memory partition using dynamic priority score
  - User-given priority + Target

MoCA’s Full-Stack Implementation

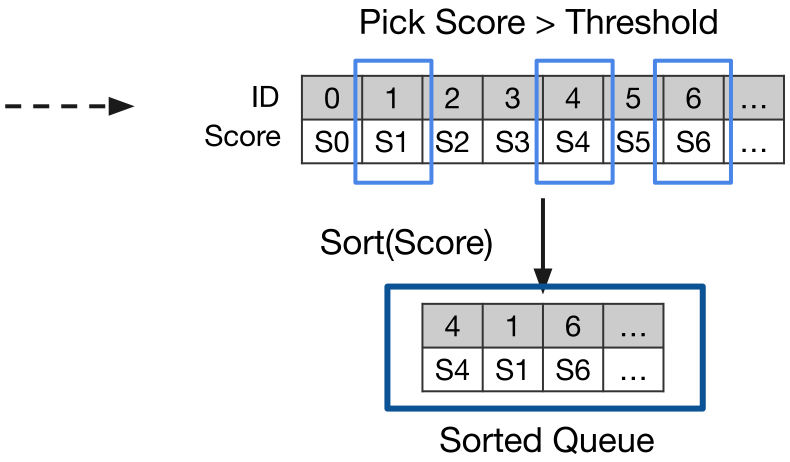


MoCA Scheduler

Task Table

ID	User_given_priority	Dispatched_time	Target
1	Priority 1	Time 1	Target 1
2	Priority 2	Time 2	Target 2
3	Priority 3	Time 3	Target 3
⋮			

Compute Score (priority, slack)

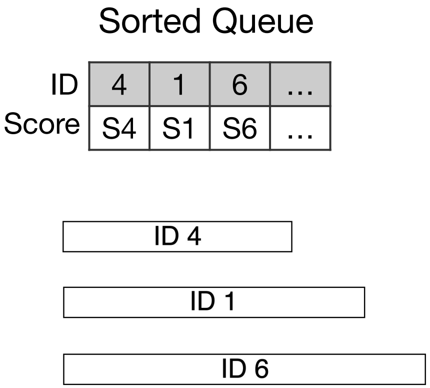


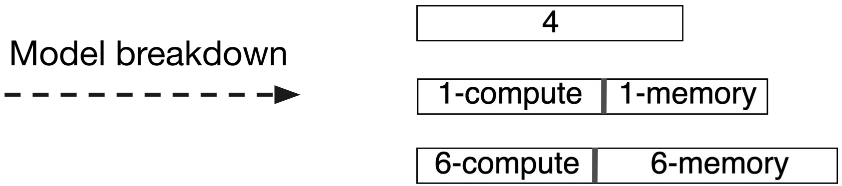
$\text{slow\_down}_i = \text{waiting\_time}_i / \text{estimated\_time}_i$

$\text{dynamic score}_i = \text{priority}_i + \text{slow\_down}_i$

- Priority, target aware dynamic scoring
- Lightweight, low overhead

MoCA Scheduler





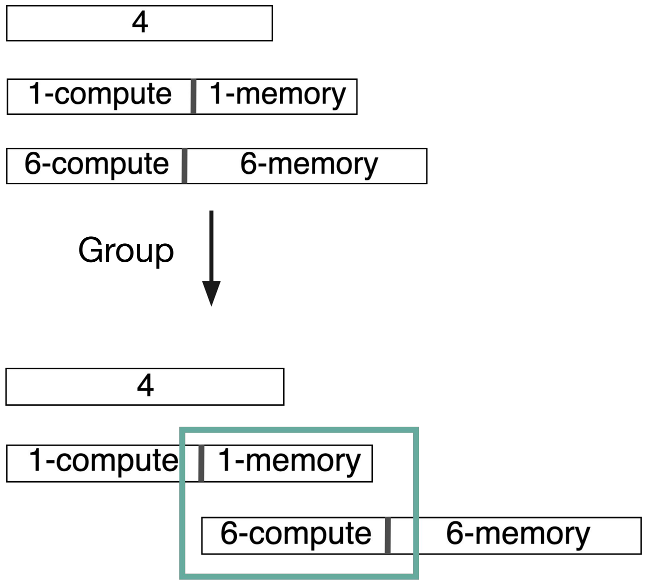
Breakdown models into compute/memory intensive parts

(Use compute-to-memory ratio)

MoCA Scheduler

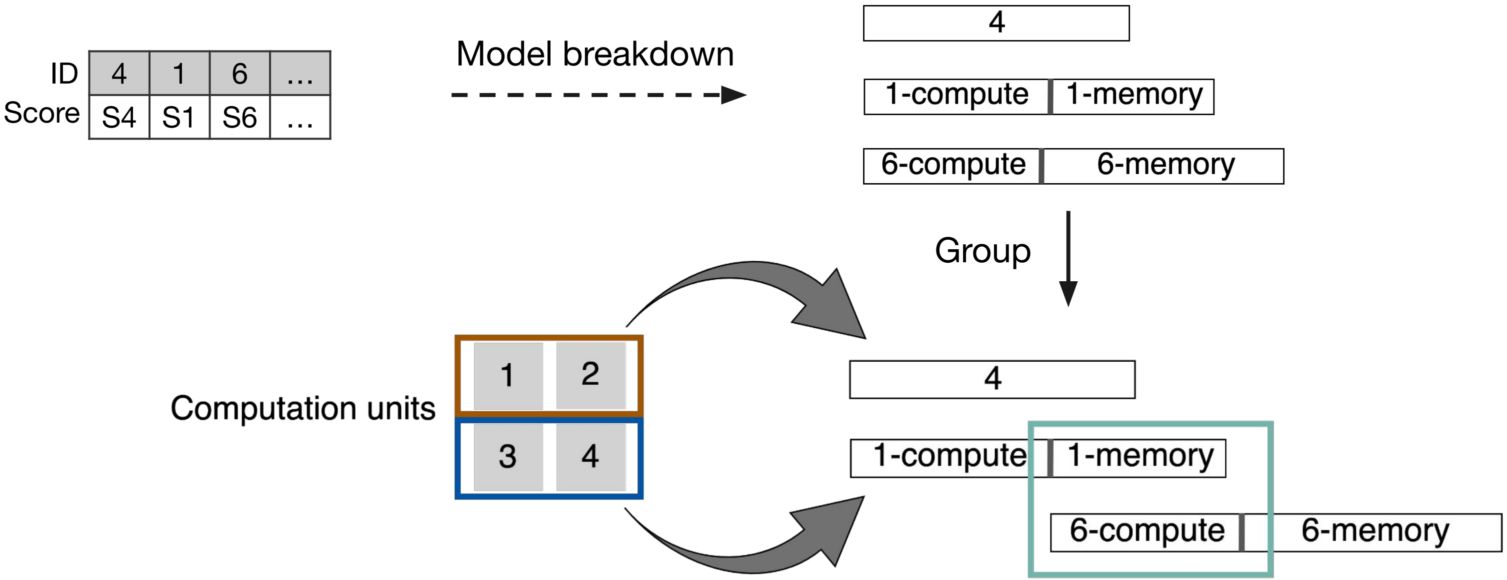
ID	4	1	6	...
Score	S4	S1	S6	...

Model breakdown



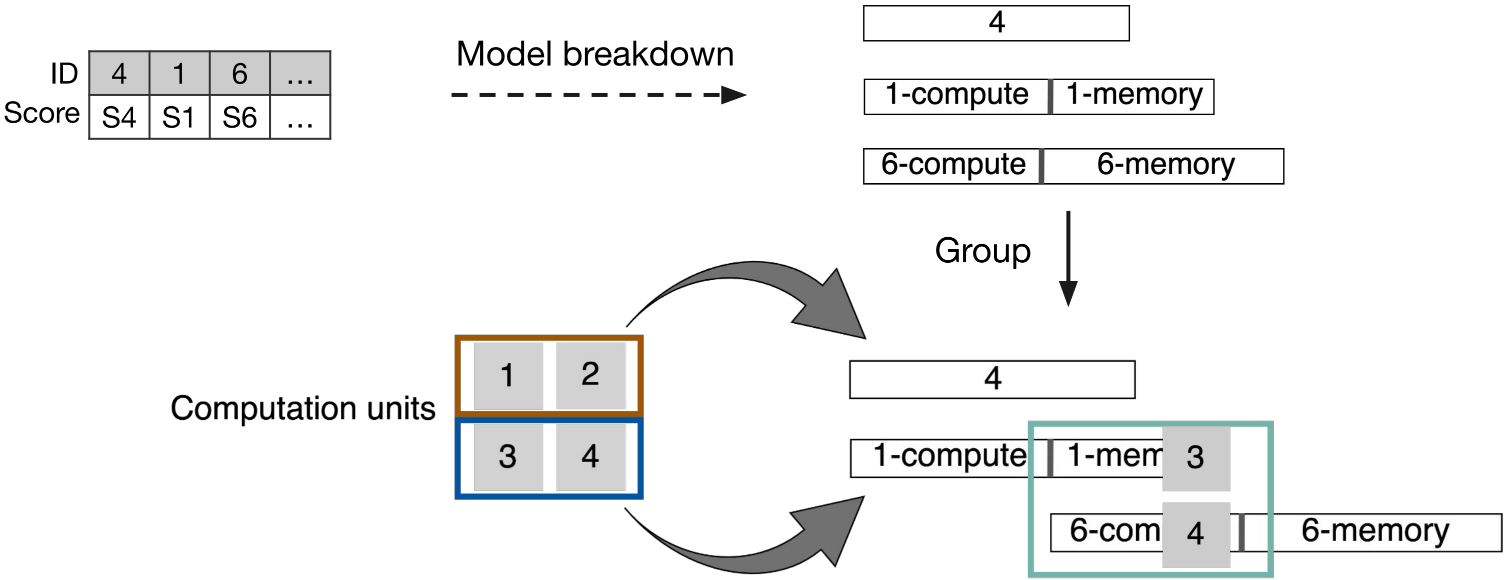
- Group compute intensive and memory intensive part
    - Better resource utilization

MoCA Scheduler



- Allocate computation resources
- Decides workload to run concurrently

MoCA Scheduler



- Decides workload to run concurrently
    - Memory-demanding & Compute-demanding tasks co-scheduled

Index

MoCA Evaluation

- Implementation details
  - Hardware: Chisel RTL language, Gemmini
  - Software: C++, Linux pthread
    - Runs on top of full Linux stack
  - Simulator: FireSim

Parameter	Value
Systolic array dimension (per tile)	16x16
Scratchpad size (per tile)	128KiB
Accumulator size (per tile)	64KiB
# of accelerator tiles	8
Shared L2 size	2MB
Shared L2 banks	8
DRAM bandwidth	16GB/s
Frequency	1GHz

Chipyard SoC configuration

MoCA Evaluation

- Multi-tenant DNN accelerator baselines
  - **PREMA**: time-multiplexing
  - **Static Partitioning**: no repartitioning resource during runtime
  - **Planaria**: dynamically repartition of compute resources
- Benchmarks: 7 different DNN inference models
  - Grouped by model size, 3 sets
- QoS targets
  - 3 different latency targets
    - QoS-H: 1.2x
    - QoS-M: 1x
    - QoS-L: 0.8x

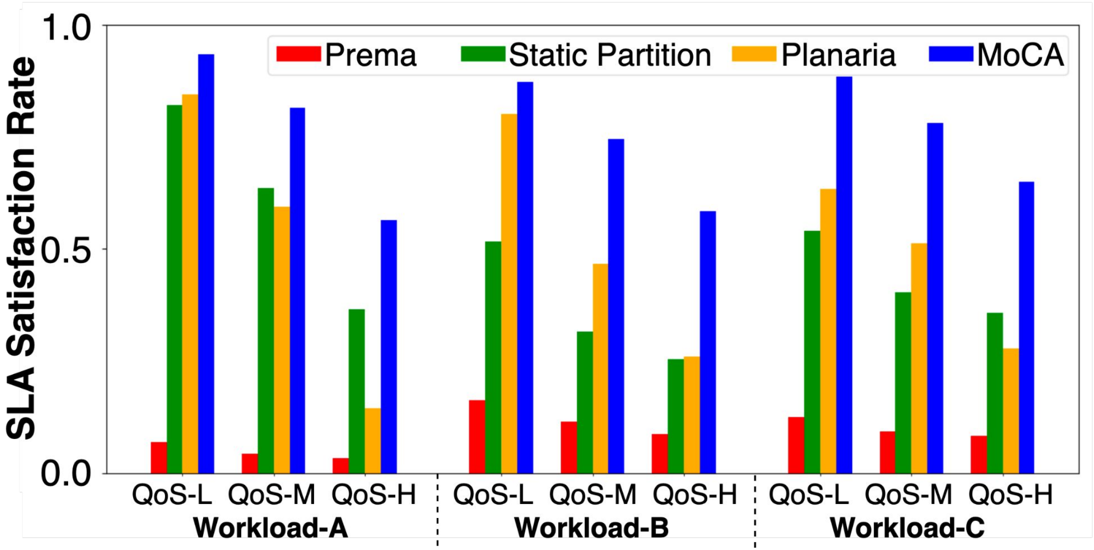
Workload	Model Size	DNN Models
Workload set-A	Light	SqueezeNet, Yolo-LITE, KWS
Workload set-B	Heavy	GoogLeNet, AlexNet, ResNet50, YoloV2
Workload set-C	Mixed	All

Index

1. Motivation
2. MoCA System
3. Methodology
4. Evaluation results

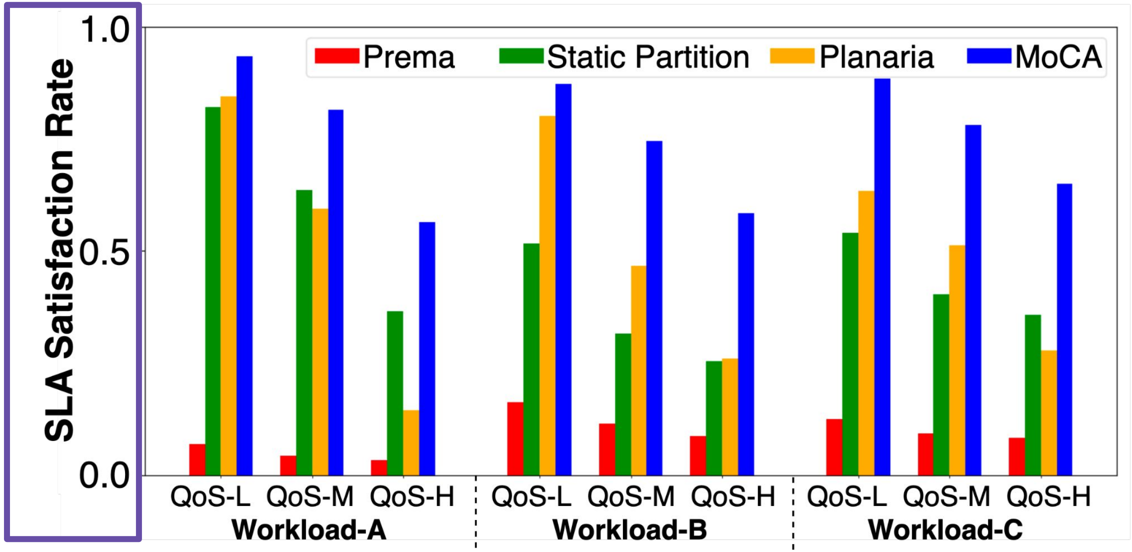
SLA Satisfaction Rate Improvement

- SLA (Service Level Agreement) satisfaction
  - Whether the request meets QoS target

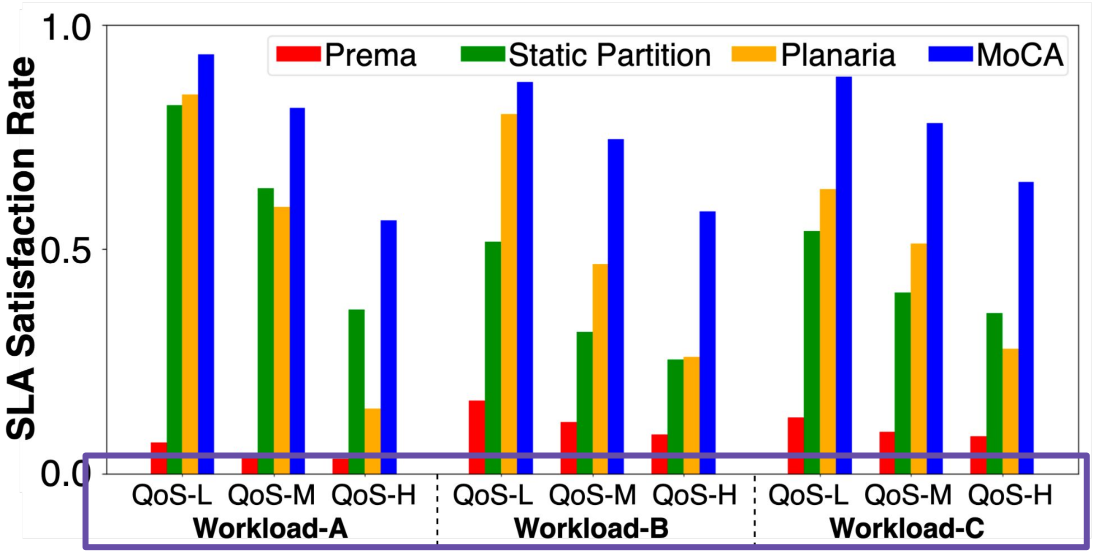


SLA Satisfaction Rate Improvement

- SLA satisfaction rate
  - Absolute value
  - Range 0 (all fail) ~ 1 (all met QoS)



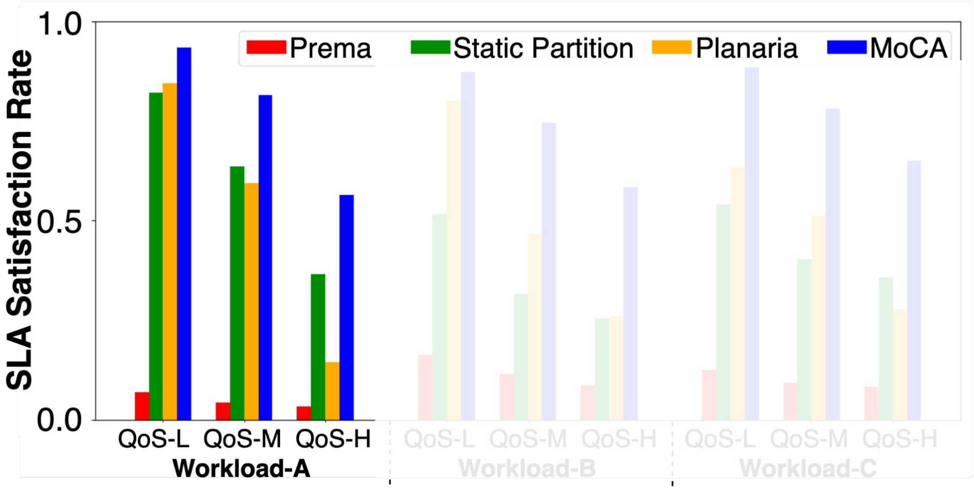
SLA Satisfaction Rate Improvement



- 2-level x-axis
  - Each workload set subdivided into QoS target level

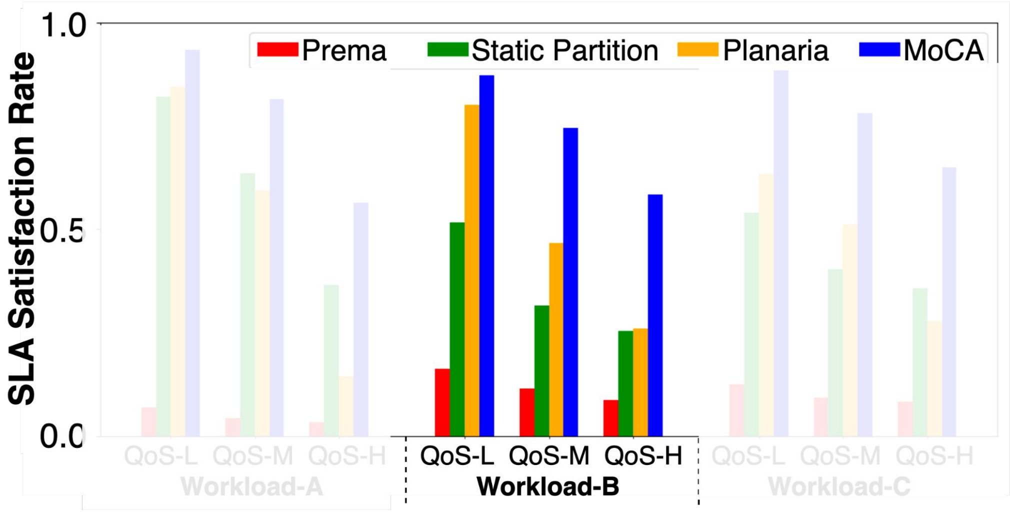
SLA Satisfaction Rate Improvement

- Workload-A: Light models
  - Prema: poor due to low scalability of light models
  - Planaria: poor due to pronounced thread migration overhead
  - MoCA’s advantage more pronounced for QoS-H



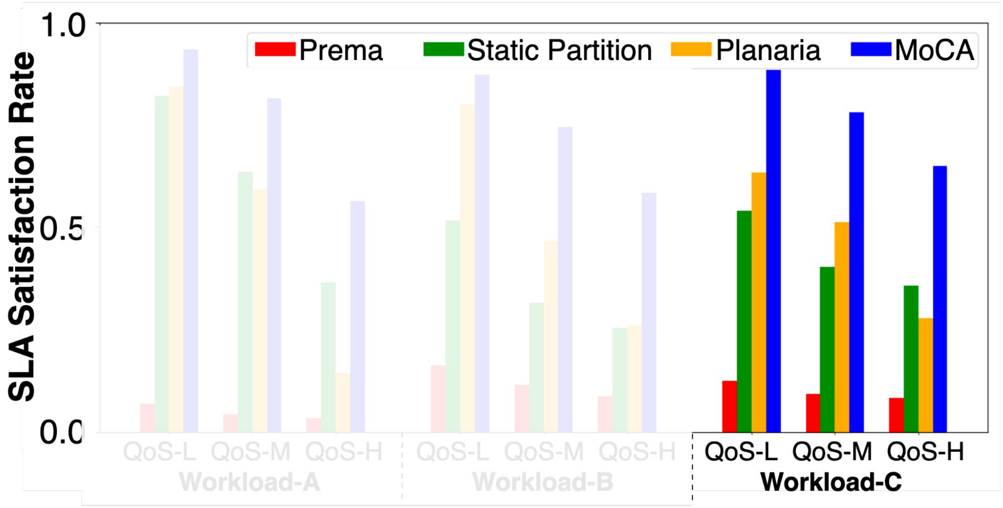
SLA Satisfaction Rate Improvement

- Workload-B: Heavy models
  - MoCA's advantage over Planaria more pronounced for QoS-H
    - Less thread migration overhead



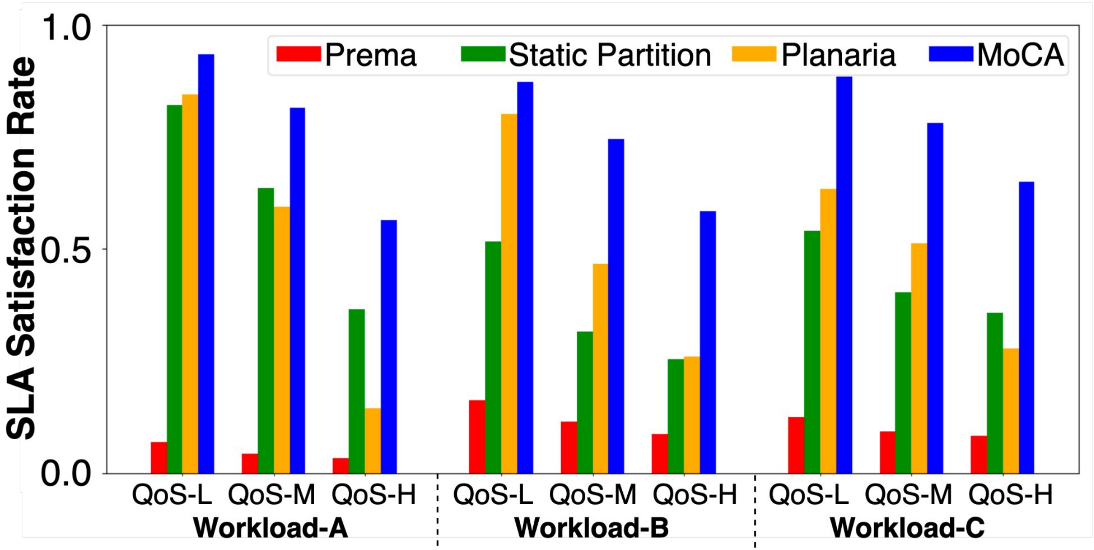
SLA Satisfaction Rate Improvement

- Workload-C: All models
  - Baselines: in between workload-A & -B
  - MoCA: co-schedule memory-intensive & light model with mixed workload set



SLA Satisfaction Rate Improvement

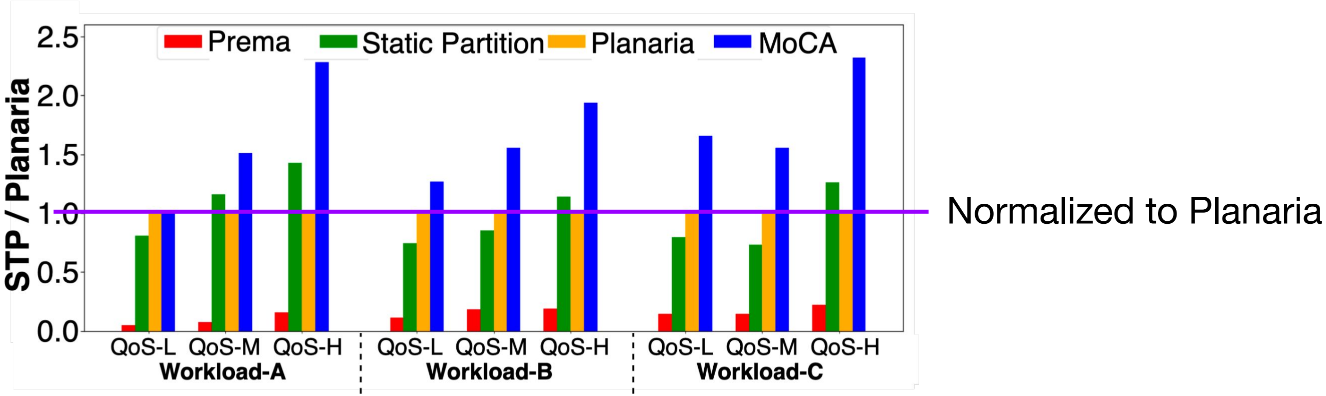
- MoCA improves SLA satisfaction rate:
  - Shows effectiveness of ability to modulate shared memory contention
  - Shows good adaptiveness without thread migration overhead



  - To Prema: 8.7x (geomean), 18.1x (max)
  - To Static Partition: 1.8x (geomean), 2.4x (max)
  - To Planaria: 1.8x (geomean), 3.9x (max)

Throughput Comparison

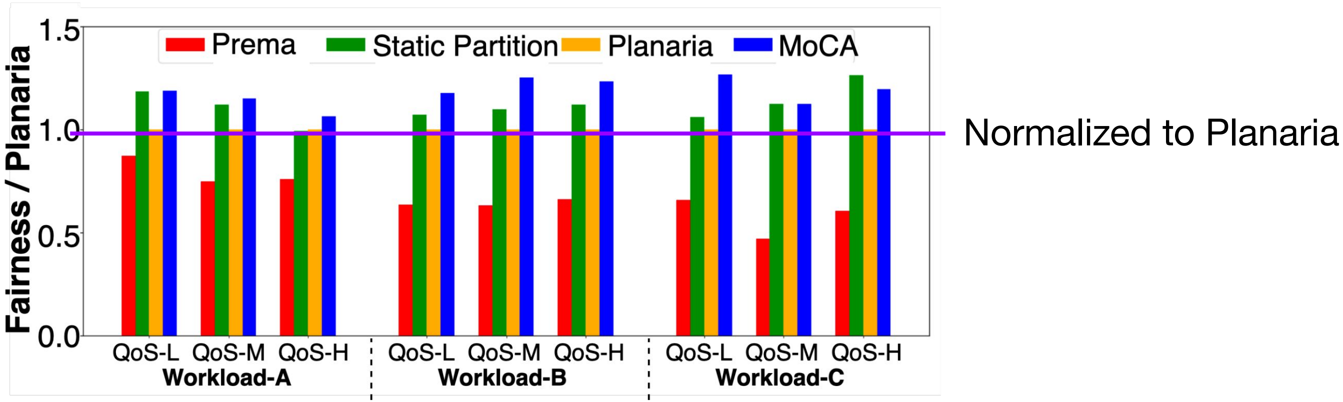
- MoCA constantly shows better throughput than baselines
- Workload-C (mixed): shows highest improvement
  - Better compute/memory utilization
  - More co-location of memory and compute intensive layers



- To Prema: 12.5x (geomean), 20.5x (max)
- To Static Partition: 1.7x (geomean), 2.1x (max)
- To Planaria: 1.7x (geomean), 2.3x (max)

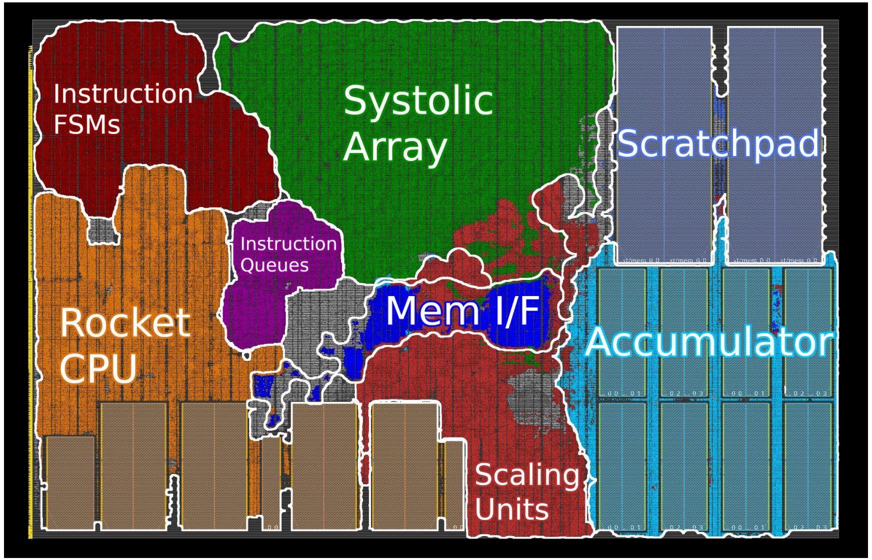
Fairness Comparison

- Fairness metric:
  - Measures the degree to which all programs have equal progress
  - Evaluate priority aware scoring
- Fairness improvement
  - Co-runners do not unequally starve



Physical Design & Area Analysis

- Synthesize, Place & Route using GF 12nm
  - Synthesis: Cadence Genus
  - Place-and-route: Cadence Innovus



Layout of an accelerator tile with MoCA

MoCA takes only small area: 0.02% out of entire

Component	Area (μm^2)	% of System Area
Rocket CPU	101K	20.5%
Scratchpad	58K	11.7%
Accumulator	75K	15.2%
Systolic Array	78K	15.8%
Instruction Queues	14K	2.8%
Memory Interface w/o MoCA	8.6K	1.7%
MoCA hardware	0.1K	0.02%
Tile	493K	100%

Area breakdown of an accelerator tile with MoCA

Artifact Evaluation Badging

- Artifact evaluated & available
  - ORO (opened) / ROR (reviewed) / ROR-R (result reproduced)

Artifact repo: https://github.com/ucb-bar/MoCA

Acknowledgement

This research was, in part, funded by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006).

This work was also supported in part by the NSF Award CCF-1955450 and in part by SLICE Lab industrial sponsors and affiliates.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Contribution

- Develop MoCA System for multi-tenant DNN accelerator
  - Adaptively adjust contentiousness under system-level contention
- MoCA Hardware
  - Monitor memory accesses and limit the request
- MoCA Runtime
  - Runtime contention detection
  - Adaptively configure hardware based on target and priority
- MoCA Scheduler
  - Priority, target, memory contention aware scheduler for multi-tenant execution

Thanks!

Please contact seah@berkeley.edu if you have any questions 🐇