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

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

Seah Kim (seah@berkeley.edu)
Multi-tenancy for DNN

Multiple tasks share system resources

Detection
Tracking
Motion Prediction
Trajectory Planning

Autonomous Vehicle
@ Edge

Clients
Cloud Accelerator
@ Cloud
Multi-tenancy Example

Example: Pylot

Multi-tenancy Example

- Consists of multiple different modules
  - Perception, Prediction, Planning, Control
Multi-tenancy Example

- Multiple tasks exist in a module

Module consist of different models

Models with different processing rate
  -> different target deadline

Sensors

Telephoto Camera

Wide-angle Camera

LiDAR

GPS

Perception

Detections

Tracking

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

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Priorities</th>
<th>Percentage of jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>0 or 1</td>
<td>33.63%</td>
</tr>
<tr>
<td>Other</td>
<td>2 to 8</td>
<td>56.30%</td>
</tr>
<tr>
<td>Production</td>
<td>9</td>
<td>9.91%</td>
</tr>
<tr>
<td>Monitoring</td>
<td>10</td>
<td>0.13%</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>11</td>
<td>0.002%</td>
</tr>
</tbody>
</table>

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

**Prema**
[HPCA’2020]

**Planaria**
[MICRO’2020]

**This Work**

Dynamically partitioning memory resources
Index

1. Motivation
2. MoCA System
3. Methodology
4. Evaluation results
MoCA's Full-Stack Implementation

- MoCA Scheduler
  - Priority, Target, Memory aware
  - Co-running layers
- MoCA Runtime
  - Contention detection
  - Adaptive memory resource partition
  - MEM rate configure
- MoCA Hardware
  - Memory access monitor
  - Throttle
MoCA’s Full-Stack Implementation

Global

MoCA Scheduler

MoCA Runtime

MEM rate configure

Local

MoCA Hardware

Memory access monitor
Throttle
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 Scheduler
  - Co-running layers
  - MEM rate configure
  - Contention detection
  - Adaptive memory resource partition

- MoCA Runtime

- MoCA Hardware

Global

Local
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

<table>
<thead>
<tr>
<th>Global Scoreboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
</tr>
<tr>
<td>BW_1</td>
</tr>
<tr>
<td>Score_1</td>
</tr>
</tbody>
</table>

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**: Priority, Target, Memory aware
- **Co-running layers**
- **MoCA Runtime**
- **MoCA Hardware**
MoCA Scheduler

<table>
<thead>
<tr>
<th>Task Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

slow_down<sub>i</sub> = waiting_time<sub>i</sub>/estimated_time<sub>i</sub>
dynamic score<sub>i</sub> = priority<sub>i</sub> + slow_down<sub>i</sub>

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td>6</td>
<td>S6</td>
</tr>
</tbody>
</table>

Sorted Queue

Model breakdown

<table>
<thead>
<tr>
<th>ID</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Breakdown models into compute/memory intensive parts
(Use compute-to-memory ratio)

- 4
  - 1-compute
  - 1-memory
- 6
  - 6-compute
  - 6-memory
MoCA Scheduler

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

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

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

Chipyard SoC configuration
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

MoCA Evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Model Size</th>
<th>DNN Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload set-A</td>
<td>Light</td>
<td>SqueezeNet, Yolo-LITE, KWS</td>
</tr>
<tr>
<td>Workload set-B</td>
<td>Heavy</td>
<td>GoogLeNet, AlexNet, ResNet50, YoloV2</td>
</tr>
<tr>
<td>Workload set-C</td>
<td>Mixed</td>
<td>All</td>
</tr>
</tbody>
</table>
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)
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
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

Normalizing to Planaria:
- 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

![Fairness Comparison Diagram](image)
Physical Design & Area Analysis

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

MoCA takes only small area: 0.02% out of entire

<table>
<thead>
<tr>
<th>Component</th>
<th>Area ($\mu m^2$)</th>
<th>% of System Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket CPU</td>
<td>101K</td>
<td>20.5%</td>
</tr>
<tr>
<td>Scratchpad</td>
<td>58K</td>
<td>11.7%</td>
</tr>
<tr>
<td>Accumulator</td>
<td>75K</td>
<td>15.2%</td>
</tr>
<tr>
<td>Systolic Array</td>
<td>78K</td>
<td>15.8%</td>
</tr>
<tr>
<td>Instruction Queues</td>
<td>14K</td>
<td>2.8%</td>
</tr>
<tr>
<td>Memory Interface w/o MoCA</td>
<td>8.6K</td>
<td>1.7%</td>
</tr>
<tr>
<td>MoCA hardware</td>
<td>0.1K</td>
<td>0.02%</td>
</tr>
<tr>
<td>Tile</td>
<td>493K</td>
<td>100%</td>
</tr>
</tbody>
</table>
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 🐰