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

Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov,
Krste Asanović, Borivoje Nikolić, Yakun Sophia Shao
University of California, Berkeley
{seah,hngenc,vnikiforov,krste,bora,ysshao}@berkeley.edu

***Abstract*—Driven by the wide adoption of deep neural networks (DNNs) across different application domains, multi-tenancy execution, where multiple DNNs are deployed simultaneously on the same hardware, has been proposed to satisfy the latency requirements of different applications while improving the overall system utilization. However, multi-tenancy execution could lead to undesired system-level resource contention, causing quality-of-service (QoS) degradation for latency-critical applications.**

To address this challenge, we propose MoCA[1], an adaptive multi-tenancy system for DNN accelerators. Unlike existing solutions that focus on compute resource partition, MoCA dynamically manages shared memory resources of co-located applications to meet their QoS targets. Specifically, MoCA leverages the regularities in both DNN operators and accelerators to dynamically modulate memory access rates based on their latency targets and user-defined priorities so that co-located applications get the resources they demand without significantly starving their co-runners. We demonstrate that MoCA improves the satisfaction rate of the service level agreement (SLA) up to 3.9× (1.8× average), system throughput by 2.3× (1.7× average), and fairness by 1.3× (1.2× average), compared to prior work.

I. Introduction

Recent advances in deep learning have led to the broad adoption of DNN-based algorithms in tasks such as object detection [20], natural-language processing [14], AR/VR [22], [30], [52], and robotics [5], [47]. As a result, DNNs have become a core building block for a diverse set of applications running on devices ranging from edge platforms to data centers, many of which need to be executed at the same time to meet their latency requirements [36], [44].

To support the concurrent execution of these applications, *multi-tenancy* execution, where multiple applications are colocated on the same hardware, is required to improve the overall efficiency of the system. The key challenge of supporting effective multi-tenancy execution in DNN accelerators is the performance variability caused by contention for shared resources, especially for user-facing applications with strict latency requirements. Recently, new techniques have been proposed to either temporally interleave the execution of multiple DNNs [2], [9], [40] or spatially partition the compute resources of DNN accelerators [18], [30], [33] to support multi-tenancy execution in DNN accelerators.

Although previously proposed microarchitecture or scheduling techniques are effective in partitioning the compute resources of accelerators, little attention has been paid to the management of shared memory subsystems, e.g., the shared system cache and DRAM, where multiple accelerators and general-purpose cores could compete for bandwidth and capacity. The absence of system-level solutions to adaptive contention management would lead to unexpected end-to-end performance degradation.

To address this challenge, we present MoCA, a memory-centric, adaptive accelerator architecture that supports efficient multi-tenancy execution for DNN workloads. In contrast to prior work that focuses on the efficient sharing of accelerator compute resources, MoCA aims to adaptively manage the system-level shared-resource contention for co-located applications by dynamically partitioning both the compute and memory resources without incurring high overhead. In particular, MoCA exploits the regularity of DNN operators and hardware, where execution latency is highly correlated with the number of in-flight memory requests.

Specifically, MoCA consists of 1) a lightweight hardware memory access monitoring and regulation engine, 2) an intelligent runtime system that manages the memory usage of each of the co-located applications dynamically, and 3) a priority-aware scheduler that selects the workloads to execute concurrently based on its user-assigned priority, latency target, and memory resource usage. Our evaluation shows that MoCA can improve overall QoS by up to 3.9× (1.8× on average), together with an improvement in system throughput of 2.3× and fairness of 1.3×, compared to prior work that partitions compute resources only [18]. In summary, this paper makes the following contributions:

1) We develop MoCA, an adaptive contention management mechanism through co-design of hardware, runtime system, and scheduler to adaptively adjust the contentiousness of co-located DNN applications.
2) We implement the MoCA hardware in RTL that dynamically monitors and regulates the memory access rates of DNN accelerators.
3) We design the MoCA runtime system that adaptively configures MoCA hardware based on the target latency and priority of the co-located workloads.
4) We build the MoCA scheduler, a priority- and memory-contention-aware workload scheduler to select different DNN layers to run concurrently.

[1] https://github.com/ucb-bar/MoCA

5) We demonstrate the effectiveness of MoCA in FPGA-based full system simulation and synthesize and place-and-route it using an advanced 12nm process technology. Our results show that, compared to prior work, MoCA significantly improves the performance of multi-tenancy execution over a range of deployment scenarios.

II. Background and Motivation

This section discusses the need for multi-tenancy execution in today's DNN workloads, the challenges associated with multi-tenancy, along with prior work in this space.

A. Modern DNN Applications

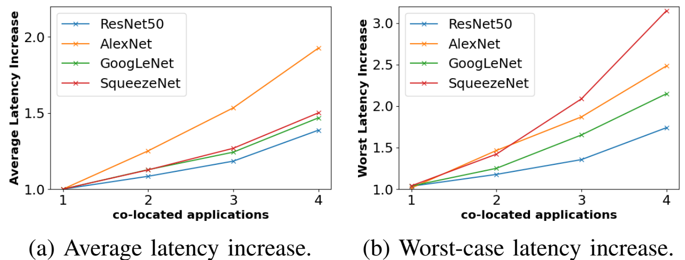
With the growing popularity of deep learning, many modern applications employ a diverse set of DNN algorithms to support different functionalities. For example, robotics [3], [5], [47] and AR/VR [22], [30], [52] applications need to detect nearby agents, track their movements, and predict the paths they will take, many of which are DNN-based and need to be performed concurrently. Furthermore, these workloads differ greatly in their latency and bandwidth requirements, ranging from real-time tasks with strict QoS deadlines, such as eye tracking on AR / VR devices [52], to offline workloads that run only when the SoC is idle, e.g., identifying people and objects in a photo album [44]. In this case, multi-tenancy execution, where multiple workloads are executed simultaneously, is a natural solution to avoid overprovisioning of hardware resources while efficiently supporting concurrent execution.

B. Challenges with Multi-Tenancy Execution

While multi-tenancy execution generally improves hardware utilization, it also often faces performance degradation due to resource contention. Even in the context of domain-specific accelerators, while each individual accelerator is generally designed in isolation, once integrated into an SoC, they typically share a number of system resources, including last-level cache (LLC), system bus, DRAM, and other I/O devices. As a result, co-located applications running on accelerators can also compete for these shared resources, leading to significant performance degradation.

To understand the performance implications of multi-tenant DNN execution, we co-locate four different DNNs [20], [23], [29], [48] on a state-of-the-art SoC with a DNN accelerator, general-purpose cores, shared system memory, and DRAM. These configurations are similar to commercially available SoCs such as NVIDIA's Xavier [46]. We randomly dispatched them at different times to capture different patterns of interference between them. We measure end-to-end execution latency using our FPGA-based RTL evaluation. Figure 1 shows the average and worst-case latency increases of multi-tenant execution. Latency is normalized to the latency of the workload running in isolation where there is no contention.

We observe at least 40% latency increase across all workloads when they are co-located with the other DNNs. In particular, AlexNet shows an almost 2× average latency increase when four applications are co-located, shown in Figure 1a, as AlexNet is quite sensitive to memory capacity as its latency is dominated by the memory-intensive fully connected layers. We also observe significant performance variations across runs. For example, the worst-case latency for SqueezeNet is more than 3× of its isolated execution, shown in Figure 1b. This is because the runtime of SqueezeNet is relatively short, making its latency quite sensitive to the behaviors of co-located workloads. The worst case happens when the entire execution of SqueezeNet is co-located with memory-intensive layers. Hence, while it is straightforward to co-locate multiple DNN workloads on the same hardware, the most important challenge is to meet the performance requirements across different co-running scenarios.



(a) Average latency increase. (b) Worst-case latency increase.

Fig. 1: Average and worst-case latency increase due to colocated DNN applications. When x=1, the workload is running in isolation with no performance degradation. When x=3, the workload is co-located with two other randomly selected workloads. We measure the end-to-end execution latency of 300 simulations with different starting time to calculate the average and worst latency across all runs.

C. Architectural Support for Multi-Tenancy

Table I summarizes hardware and software techniques developed to support multi-tenancy execution in general-purpose processors and, more recently, domain-specific accelerators. Multi-tenancy execution in traditional general-purpose cores is a well-studied topic in both software and hardware, dating back to the introduction of chip-multiprocessors. Generally, two lines of work have been proposed.

The first category focuses on *static* mechanisms, where software or hardware is configured statically to adjust the memory access rates of contentious applications. For example, QoSCompile [49] statically throttles the contentious code regions, Locality-aware complication [25] re-compiles applications to reduce multicore resource contention, and Prophet [6] leverages accurate QoS prediction to identify safe co-location opportunities. While these static approaches are lightweight and do not require runtime information, they are typically overly conservative for low-priority applications, as their memory requests are delayed regardless of whether the co-running applications are indeed sensitive to contention.

The second category targets *dynamic* mechanisms, where runtime information is collected to decide when contention management mechanisms should be triggered and how aggressive these mechanisms need to be, either in software or hardware. Most of the proposed dynamic approaches target

	General Purpose Hardware		Domain Specific Hardware		
	Software	Hardware	Temporal Partition	Spatial Partition	
				Compute Resource	Memory Resource
Static	Prophet [6], QoSCompile [49], Locality-aware Compilation [25]	SplitL2 [32], PAR-BS [39], FairQueuing [43], STFM [38], CoQoS [31]	LayerWeaver [40]	HDA [30]	CachePartition [10] MAGMA [26]
Dynamic	ReQoS [50], CAER [35], Kelp [53], Baymax [7], Quasar [13], Paragon [12], Thread-throttle [8]	FST [15], QoS in CMP [19], Utility-based Partition [42], Heracles [34] FCSP [28]	Prema [9] AI-MT [2]	Planaria [18] Veltair [33]	**MoCA**

TABLE I: Taxonomy of multi-tenancy support in general-purpose hardware and domain-specific accelerators.

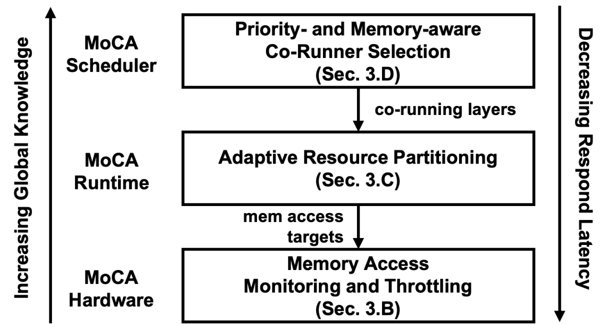


Fig. 3: Overview of the MoCA System.

general-purpose processors. While these dynamic approaches are generally more involved with additional software/hardware modules to track runtime information, they can adaptively adjust the contentious nature of an application based on the amount of memory traffic that is actually occurring in the system, leading to improved QoS and throughput.

When it comes to domain-specific accelerators, Table I further categorizes how shared resources are partitioned over time, i.e., temporal partitioning, or over space, i.e., spatial partitioning. In spatial partitioning, we also separate whether compute resources, e.g., cores or processing elements in spatial arrays, or memory resources, e.g., shared cache and DRAM bandwidth, are partitioned spatially. Previous work such as LayerWeaver [40], Prema [9], and AI-MT [2] propose time-multiplexing DNN execution, statically or dynamically, although they suffer from low hardware utilization, especially when layers cannot utilize all available resources. To improve hardware utilization, static partition techniques such as HDA [30] and MAGMA [26] propose spatial partitioning of compute or memory resources, although they cannot adapt to different dynamic scenarios due to their static nature.

Recently, dynamic spatial partition mechanisms have been proposed to further improve the co-location efficiency. Figure 2 highlights the key differences among recent efforts in this space. In particular, Planaria [18] and Veltair [33] propose dynamic allocation of compute resources for co-running workloads. Although they can adaptively allocate compute resources, e.g., processing elements for Planaria or CPU cores for Veltair, they are not able to dynamically manage memory resources, leading to low resource utilization and high thread migration overhead during compute resource repartition [33]. MoCA overcomes these challenges by emphasizing the importance of memory-centric resource management by dynamically partitioning both compute and memory resources during execution. *To the best of our knowledge,* MoCA *is the first work that adaptively partitions the memory resources to support multi-tenant execution in DNN accelerators.*

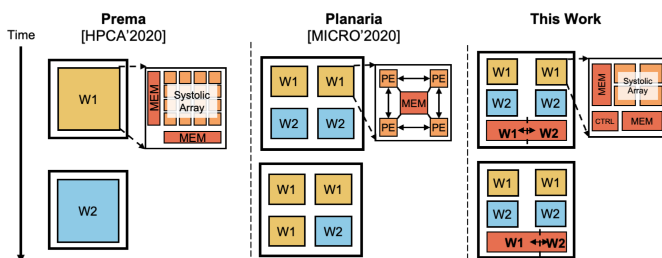


Fig. 2: Comparison of the recently proposed multi-tenancy mechanisms. Prema [9] supports temporal multi-plexing across different DNN workloads (W). Planaria [18] focuses on dynamic partition of the compute resources at the pod granularity, where each pod consists of a fixed number of PEs and scratchpads. Different from prior works, MoCA adaptively partitions *both the compute and the shared memory resources* to satisfy the resource requirements of different workloads.

III. MoCA System

We propose MoCA, a memory-centric, adaptive DNN accelerator architecture that dynamically partitions both compute and memory resources to colocated DNN workloads to improve the multi-tenant performance of DNN accelerators. In particular, MoCA dynamically manages the memory access rates of co-running workloads through hardware, runtime, and scheduler co-design to meet the QoS requirements of applications. This section first provides an overview of the MoCA system, followed by detailed discussions of MoCA's hardware, runtime system, and scheduler components.

A. MoCA *Overview*

MoCA is a full stack system composed of 1) a lightweight hardware monitoring and throttling engine to control the memory accesses of the colocated DNNs, 2) an intelligent runtime that dynamically modulates the processing rate of each workload, and 3) a priority- and memory-aware job scheduler that selects the co-running workloads, as shown in Figure 3. Starting at the bottom of the stack, the MoCA hardware monitors the memory access counts of each individual accelerator tile and throttles execution if the hardware exceeds the allocated memory access targets over a certain amount of time. The memory access target is set by the MoCA runtime system, which detects interference across co-running jobs and adaptively repartitions the available resources if needed.

Moving up the stack, the co-running jobs are selected by the MoCA scheduler, which takes the user-defined priorities and memory requirements of the network layers as input and chooses the co-running layers that meet the QoS requirements of the workloads.

Together, the hardware, runtime, and scheduler components of MoCA reduce system-level contention and deliver improved QoS for multi-tenant execution. Next, we will discuss each component of the MoCA system in more detail.

B. MoCA *Hardware*

The responsibility of the MoCA hardware is to dynamically monitor the memory access rate of each accelerator tile and throttle its execution if the target access counts have been reached. Although dynamic memory throttling has been implemented in general-purpose systems before, to the best of our knowledge, MoCA is the first work that demonstrates adaptive memory regulation in multi-tenant accelerators. Unlike early multi-tenant accelerators that focus on compute resource partition [9], [18], [33], the MoCA hardware builds on top of the decoupled access/execute nature of state-of-the-art DNN accelerators [24] and controls accelerator's memory accesses independently without changing accelerator's compute engine.

Figure 4 shows the hardware microarchitecture of the proposed MoCA DNN accelerator. In addition to the standard systolic array and buffers (to store weights (W), input activations (IA), and output activations (OA) in DNN), MoCA adds two new hardware modules between the ld/st queues and the memory request generation engine to monitor and throttle its own locally-generated memory accesses to the shared memory resource. Specifically, MoCA implements an *Access Counter to locally track its memory access counts during the monitored time window, and a Thresholding Module to prevent accelerator from further generating memory requests if the Access Counter value exceeds its threshold load during that time window, which is configured by the MoCA runtime system.*

We implement our hardware based on the Gemmini infrastructure [17], an open-source RISC-V-based DNN accelerator infrastructure that supports TPU-like decoupled access/execute execution. The systolic array and buffers are generated directly from Gemmini, and we implement the MoCA-specific access monitoring and throttle engines in Chisel RTL [1]. Specifically, the *Access Counter* monitors accelerator memory requests to check whether the current execution is overly utilizing memory than its target number of memory requests configured by MoCA runtime system.

Based on the rate at which the accelerator issues load requests, the thresholding module determines how many cycles "bubbles" should be inserted to block further memory requests. Bubbles prevent the accelerator from making additional load requests, allowing MoCA to reduce memory contention while providing each accelerator with the number of memory accesses allocated by the MoCA runtime. Whenever the access counter raises an alert that its accumulated counter exceeds threshold, the MoCA hardware begins to stall, until its status is updated by the MoCA runtime. Unlike previous work that required a

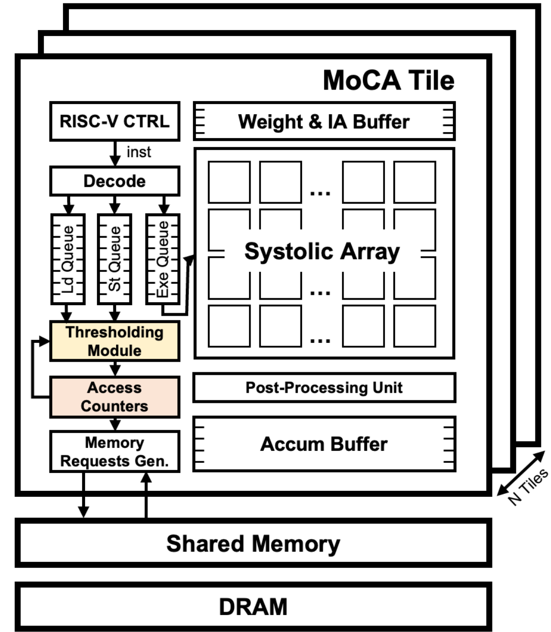


Fig. 4: The MoCA DNN accelerator that supports dynamic memory access monitoring and throttling. The MoCA hardware monitors the memory accesses that have been issued from each of the accelerator tiles and throttle their memory accesses if it exceeds the limit.

significant redesign of DNN accelerators to support flexible resource partitioning [18], the MoCA hardware engines can be implemented as lightweight finite-state machines and counters without incurring significant overhead.

C. MoCA *Runtime*

The MoCA runtime is responsible for dynamically detecting system-level interference and setting limits on the memory access rates of accelerators to resolve contention if necessary. Existing multi-tenancy solutions focus on compute resource partitioning. While easy to manage, these solutions tend to incur non-negligible repartitioning latency due to high thread migration overhead, as observed in prior work [18], [33]. Instead, MoCA's full-stack design exposes the hardware states to MoCA's runtime so that it can dynamically partition both memory and compute resources based on the demand of applications and the cost of re-partitioning. In fact, MoCA's runtime triggers the compute resource partition much less frequently to avoid its high overhead.

Specifically, MoCA's runtime consists of two components. The first part is the performance prediction of co-running layers based on available hardware resources, and the second part is the contention detection module and the hardware repartition module which activates when contention occurs. To accurately estimate the performance and memory requirements of the co-running layers, the MoCA runtime captures the data movement costs across the full memory system. If the memory bandwidth requirement of the co-running layers is greater than the available bandwidth in the system, MoCA's

Algorithm 1 Latency Estimation in MoCA Runtime

```
    ▷ % Estimate latency for COMPUTE layers %
 1: if Layer_i is COMPUTE then
 2:     Total_MAC_i ← calc_MAC_count(Layer_i)
        ▷ % Calculate compute-only time %
        ▷ % Ideal compute time with 100% PE utilization %
 3:     Compute_ideal_i ← Total_MAC_i/num_PEs
 4:
        ▷ % Calculate memory-only time %
        ▷ % Total traffic to shared L2 %
 5:     Total_MEM_i ← Total_load_i + Total_store_i
        ▷ % Subset of traffic to DRAM %
 6:     From_DRAM_i ← (Weight + Output + Bias)_size_i
        ▷ % If input can be cached, no need to reload from
    DRAM. Otherwise, reload. %
 7:     if Image_size_i > Cache_size then
            ▷ % If input activation got evicted %
 8:         From_DRAM_i += Image_size_i
 9:     end if
10:     if Per_tile_size_i > Cache_size then
            ▷ % If tile size exceeds cache size, it got evicted %
11:         From_DRAM_i+=Tiling_factor · Tile_size
12:     end if
        ▷ % Consider both L2 and DRAM transaction time %
13:     Memory_ideal_i ← From_DRAM_i/DRAM_BW + Total_MEM_i/L2_BW
14:
        ▷ % Estimate overall latency from compute & memory
    time, considering compute-to-memory ratio %
15:     Prediction_i←Max(Compute_ideal_i, Memory_ideal_i)
16:      + Min(Compute_ideal_i, Memory_ideal_i)·overlap_f
17: end if
18:
    ▷ % Estimate latency for MEM layers %
19: if Layer_i is MEM then
20:     Total_Mem_i ←InputAB_size_i+Output_size_i
21:     From_DRAM_i ← InputB_size_i + Output_size_i
22:     Prediction_i ← From_DRAM_i/DRAM_BW + Total_MEM_i/L2_BW
23: end if
```

Algorithm 2 MoCA Contention Detection and HW Update

```
    ▷ % Get latency and memory usage of the remaining layers
    using Algorithm 1 %
 1: remain_prediction ← Alg1(Layers)
 2: for Layer_i in Layers do
 3:     (From_DRAM_i,Total_MEM_i,Prediction_i)←Alg1(Layer_i)
 4:     BW_rate_i ← From_DRAM_i/Prediction_i
 5:     slack ← time_left_to_target
        ▷ % Dynamic priority score update %
 6:     priori_score←user_priority_i + remain_prediction/slack
        ▷ % Latency prediction of remaining layers of NN %
 7:     remain_prediction -= Prediction_i
 8:
        ▷ % Look up co-running applications' memory usage
    in scoreboard %
 9:     for App_j in other_Running_Apps do
            ▷ % Add current memory BW usage per workload%
10:         other_BW_rate += MEM_BW(App_j)
            ▷ % Weighted sum using dynamic priority score %
11:         weight_sum += score(App_j)·MEM_BW(App_j)
12:     end for
13:
        ▷ % Whether the current system's total memory demand
    is bigger than the maximum BW %
14:     overflow←BW_rate_i+other_BW_rate−DRAM_BW_MAX
        ▷ % Contention detected (real-time detection) %
15:     if overflow > 0 then
16:         curr_weight_sum←priori_score·BW_rate_i
            ▷ % Allocate memory w/ dynamic priority score %
17:         BW_rate_i -= overflow*weight_sum/(curr_weight_sum+weight_sum)
            ▷ % Update prediction based on allocation %
18:         Prediction_i ← BW_rate_i · From_DRAM_i
19:
            ▷ % Set HW config (dynamic memory partition) %
20:         threshold_load_i ←Total_MEM_i/Num_tile_i
21:         window_i ← Prediction_i/Num_tile_i
22:     else
            ▷ % Contention not detected, no throttling %
23:         threshold_load_i ← 0,   window_i ← 0
24:     end if
25:     UpdateScoreboard(this_App, BW_rate_i)
26:     ConfigureHW(window_i, threshold_load_i)
27:     runLayer(Layer_i)
28: end for
```

runtime declares that contention is detected. In that case, MoCA's runtime reconfigures the MoCA hardware to either limit the memory access rate or, in rare cases, repartition the compute engines, based on both user-defined priorities and target latencies of co-running applications.

Algorithm 1 illustrates the latency estimation algorithm in the MoCA runtime. Unlike compute-oriented latency estimation in prior multi-tenant solutions [9], [18], [33], MoCA considers the movement of data across the full memory system, including both shared memory (i.e., L2) and DRAM, allowing fast and accurate performance prediction based on different memory resources. Specifically, MoCA categorizes DNN layers into two types: `compute` layers that have high arithmetic intensity (such as convolutional layers or fully-connected layers), and `mem` layers that exhibit little data reuse and are composed of memory-bandwidth-bound operators (such as residual additions and max-poolings which cannot be fused with CONV layers). `overlap_f` is a parameter that represents system's ability to overlap compute and memory operations, which can be varied ($0 < \texttt{overlap_f} < 1$) by accelerator design. We provide a tuning utility that determines the optimal value of `f` for an SoC using data collected by running a few DNN layers before starting inference queries. For each layer of DNN networks, Algorithm 1 calculates the total number of multiply and accumulate operations (MAC) to compute based

on the dimensions of the layer, together with the cost of data movement between different levels of the memory hierarchy, such as DRAM and L2. Our validation shows prediction error within 10% of measured runtimes across networks and layers, consistent with other analytical models for DNN hardware such as Timeloop [41].

Algorithm 2 shows how the MoCA runtime sets parameters for the MoCA hardware engine. We first use Algorithm 1 to calculate the memory requirement and estimate the latency of the current layer without interference, which are deterministic based on the layer and hardware configuration. This value is used to calculate the workload's score. The score represents the relative importance of the workload, and is computed using both the workload's priority and whether enough time remains to finish the entire network within its target deadline. The latency prediction (remain_prediction) is updated for every layer, since the scores are updated based on the layers that have not been executed. The MoCA hardware engine is reconfigured each time the dynamic scores are updated, since relative importance and system contention would change over time due to changes in co-running workload. Dynamic priority in MoCA considers both the SLA target and user-defined static priority. In Algorithm 2, priori_score is the dynamic score used in MoCA, while slack calculates the time remaining to hit the SLA target, and user_priority is the static priority. Thus, when memory contention is detected and memory resources get partitioned, MoCA prioritizes applications with higher priorities and applications with less time left to meet their targets.

D. MoCA *scheduler*

Moving up the stack, we also need a workload scheduler to intelligently select co-running applications that can be executed concurrently in the system. In particular, the multi-tenant scheduler takes the following requirements into consideration:

1) The scheduler needs to be aware of the existence of different user-given priorities and has to react accordingly.
2) The scheduler needs be aware of both the memory and compute resource requirements of workloads.
3) The scheduler algorithm needs to be light-weight and simple enough to be computed in real-time without incurring significant overhead.

Existing multi-tenant schedulers lack the consideration of memory resources [9], [18] as well as user-defined priority and latency targets [26], [30], making them unsuitable for handle a diverse set of workloads under different deployment scenarios.

To meet these requirements, we design the MoCA scheduler, described in Algorithm 3. In particular, the MoCA scheduler considers both the user-defined priority of each task and how long the task has been dispatched when it selects co-running workloads. Specifically, MoCA uses a TaskQueue to store the dispatched workloads. Each task can be defined using either model granularity or as groups of layers if there are significant differences in the compute-to-memory ratio between layers within the model. Each of the task queue entries includes the

Algorithm 3 MoCA Scheduler for Multi-tenant Execution

```
    ▷ % During the scheduling period, calculate and update
    the score based on the user-given priority %
 1: curr = current_time()
 2: for each Task_i in TaskQueue do
 3:     Score_i ← user_given_priority_i
 4:     WaitingTime_i ← curr − DispatchedTime_i
 5:     Slowdown_i ← WaitingTime_i / EstimatedTime(Task_i)
 6:     Score_i += Slowdown_i
        ▷ % Flag the task if it is memory intensive %
 7:     if EstimatedAvg_BW_i > 0.5 × DRAM_BW then
 8:         isMemIntensiveTask[i] ← True
 9:     else
10:         isMemIntensiveTask[i] ← False
11:     end if
12: end for
13:
    ▷ % Populate ExQueue based on the updated score %
14: ExQueue ← [Task_i if Score_i > Threshold]
15: ExQueue.sort()
16:
    ▷ % Form co-running tasks based on their scores and
    memory intensiveness %
17: groupTask = []
18: while resourceAvailable do
19:     currTask = ExQueue.pop()
20:     groupTask.append(currTask)
21:     if isMemIntensiveTask(currTask) then
22:         coTask = findNonMemIntensiveTask(ExQueue)
23:         groupTask.append(coTask)
24:     end if
25: end while
26: return groupTask
```

Task ID, the dispatch time to TaskQueue, the current status of the workload, the user-given priority, and the target latency.

During each round of scheduling, the MoCA scheduler compares the scores of each task, which considers both the user-defined priority of each task and how long the task has been waiting in the queue, selects the highest ranked tasks and puts them in the execution queue. In addition, the MoCA scheduler also considers the memory resource requirements of different tasks during its scheduling. In the case of memory-bound layers that require extensive memory usage, the MoCA scheduler will co-schedule these layers with other layers later in the queue with lower memory requirements.

E. *Limitations of* MoCA

MoCA leverages the regularity of both DNN workloads and DNN hardware, where it assumes dense DNN workloads and does not exploit the sparsity of data. It also best targets spatial hardware architecture where the performance of executing dense DNN layers is largely determined by the available compute and memory resources. This is because if sparsity is considered in hardware, it can be challenging to

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

TABLE II: SoC configurations used in the evaluation.

Workload	Model size	Domain	DNN models
Workload set A	Light	Image Classification	SqueezeNet [23]
		Object Detection	Yolo-LITE [21]
		Speech Processing	KWS [51]
Workload set B	Heavy	Image Classification	GoogleNet [48], AlexNet [29], ResNet50 [20]
		Object Detection	YoloV2 [45]
Workload set C	Mixed	All	All

TABLE III: Benchmark DNNs used in evaluation and workload sets based on model size.

estimate the memory requirements of the DNN layers during runtime. However, MoCA can be augmented with an accurate performance and memory resource predictor of sparse DNNs to support sparse DNN accelerators.

IV. Methodology

This section details MoCA's implementation, our workloads and metrics used for our evaluation, as well as the baseline colocation solutions against which we compare the effectiveness of MoCA.

A. MoCA *implementation*

We implement MoCA hardware using the Chisel RTL language [1] on top of the Gemmini [17] infrastructure, a systolic-array-based DNN accelerator without multi-tenancy support. We evaluate performance on full, end-to-end runs of DNN workloads using FireSim, a cycle-accurate, FPGA-accelerated RTL simulator [27]. We also synthesize our hardware implementations on the GlobalFoundries 12nm process to evaluate the area overhead.

Table II shows the SoC configuration we use in our evaluations of MoCA, similar to the configurations in modern SoCs [46]. We configure Gemmini, TPU style systolic-array based accelerator generator, to produce homogeneous eight separate accelerator tiles on the same SoC, each of which can run a different DNN workload. Multiple accelerator tiles can also cooperate to run different layers/regions of the same DNN workload. Each accelerator tile is equipped with a 16x16 weight-stationary systolic array for matrix multiplications and convolutions, as well as a private scratchpad memory to store W, IA and OA. All tiles also share access to the shared memory subsystem, including a shared last-level cache and DRAM, which is the main source of contention with co-located workloads.

The MoCA runtime system and scheduler are implemented in C++ and run on top of a full Linux stack. MoCA uses a lightweight software look-up table for the scoreboard that is used to manage the bandwidth usage of each application and queues to keep track of the tasks dispatched. The algorithms used to calculate the hardware configuration on the MoCA runtime system and the MoCA scheduler are also implemented in software with little overhead observed.

We synthesize and place-and-route MoCA-enabled accelerator. We use Cadence Genus with GlobalFoundries 12nm process technology for synthesis. For place-and-route, we use Cadence Innovus.

B. *Workloads*

a) Benchmark DNNs: In our evaluations, we choose seven different state-of-the-art DNN inference models, including SqueezeNet [23], GoogLeNet [48], AlexNet [29], YOLOv2 [45], YOLO-lite [21], Keyword Spotting [51], and ResNet [20]. These DNN models represent a diverse set of DNN workloads, with different model sizes, DNN kernel types, computational and memory requirements, and compute-to-memory trade-offs. We grouped workloads into workload sets based on the DNN model size to capture any different behaviors across different sets. The workload classification is based on [18], which is one of our baselines. Table III shows the DNN benchmarks classified by size. Workload set-A is the group of lighter models, whereas Workload set-B groups heavier models. Workload set-C contains both, which is the mixture of set-A and set-B.

b) Multi-tenant workload sets: To generate multi-tenant scenarios, we select N different inference tasks randomly dispatched to the system [44], where N ranges from 200 to 500. We assign user-defined, static priority levels within the range 0 to 11, and follow the distribution on [11], [37], consistent with the methodology used for Prema and Planaria. MoCA considers the priority levels when it schedules and dynamically adjusts the different layers.

c) QoS targets: We set our baseline QoS based on [4] since each of our accelerator tiles is close to an edge device. In addition to the baseline QoS, we adjust our latency target to 1.2× and 0.8× QoS, which increases and decreases the latency target by 20%, to evaluate how MoCA reacts with different latency targets. QoS-H (hard) denotes 0.8× QoS, which is more challenging to achieve. QoS-L (light) is 1.2× QoS, which is lighter load. QoS-M stands for the baseline QoS target.

C. *Metrics*

We quantify the effectiveness of MoCA-enabled workload colocation using the metrics suggested in [16], including the percentage of workloads for which we satisfy the Service Level Agreement (SLA), the throughput of the colocated applications, and the fairness of the MoCA's strategy to manage shared resources. The latency of each workload is measured from the

time it is dispatched to the system till it finishes and commits, which includes both the time it waits in the task queue and its runtime.

a) SLA satisfaction rate: We set the SLA target for each workload based on the three different QoS levels we defined above in Section IV-B0c paragraph 'QoS targets' (we use QoS and SLA targets interchangeably.) In addition to the overall SLA satisfaction rate, we also measure the SLA satisfaction rate for each of the priority groups to highlight the effectiveness of MoCA.

b) Fairness: *Fairness* metric measures the degree to which all programs have equal progress. The fairness metric evaluates MoCA's priority scoring method used in the MoCA scheduler and dynamic partitioning memory bandwidth in the MoCA runtime. Here, we use C_i to denote the cycle time of the i-th workload, *single* represents only one workload running on the SoC, and *MT* represents the multi-tenant execution. Similar to other previous work on DNN multi-tenancy support [9], [18], we use a generalized version of *fairness* that measures *proportional progress* (PP) defined as follows:

$$PP_i = \frac{\frac{C_i^{single}}{C_i^{MT}}}{\frac{Priority_i}{\sum_{j=1}^{n} Priority_j}}, \qquad Fairness = min_{i,j} \frac{PP_i}{PP_j} \tag{1}$$

c) Throughput: We also analyze the total *system throughput* (STP) to evaluate the effectiveness of modulating the memory access rate of MoCA in increasing the overall utilization of hardware resources. STP is defined as the system throughput of executing n programs, which is defined by summing each program's normalized progress, which ranges from 1 to n. Increasing STP requires maximizing overall progress when co-locating multiple applications.

$$STP = \sum_{i=1}^{n} \frac{C_i^{single}}{C_i^{MT}} \tag{2}$$

D. Baseline

To evaluate the effectiveness of MoCA, we compare with three different baseline:

1) Prema [9], which supports multi-tenancy execution through a time-multiplexing DNN accelerator across different DNNs with a priority-aware scheduling algorithm;
2) Static compute partitioning, which exploits spatial co-location of multiple workloads but does not repartition the resource during runtime;
3) Planaria [18], which spatially co-locate multiple DNNs by dynamically partitioning the compute resources with a fixed compute vs. memory resource ratio;

In addition, instead of the layerwise granularity to reconfigure resources, we break down DNN networks into layer blocks, which consists of multiple layers, and reconfigure at the layer-block granularity, as recent work demonstrate layer-block granularity delivers supreme performance [33]. For fair comparison, we compare MoCA with the stated baselines on the same hardware configuration.

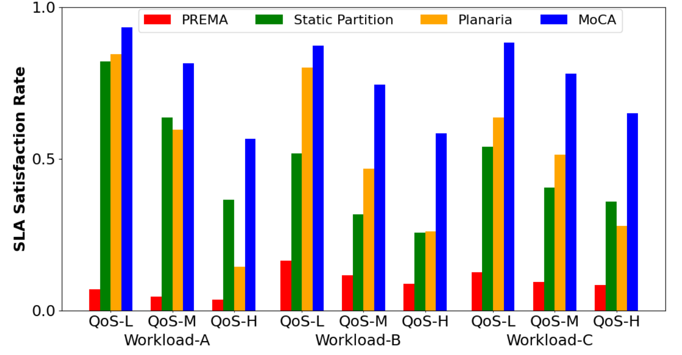


Fig. 5: MoCA's SLA satisfaction rate improvement over multi-tenancy baselines with different QoS targets (QoS-L: light latency target, QoS-M: medium latency target, QoS-H: hard latency target) and different DNN workload sizes (Workload-A: light models, Workload-B: heavy models, Workload-C: all models).

V. Evaluation

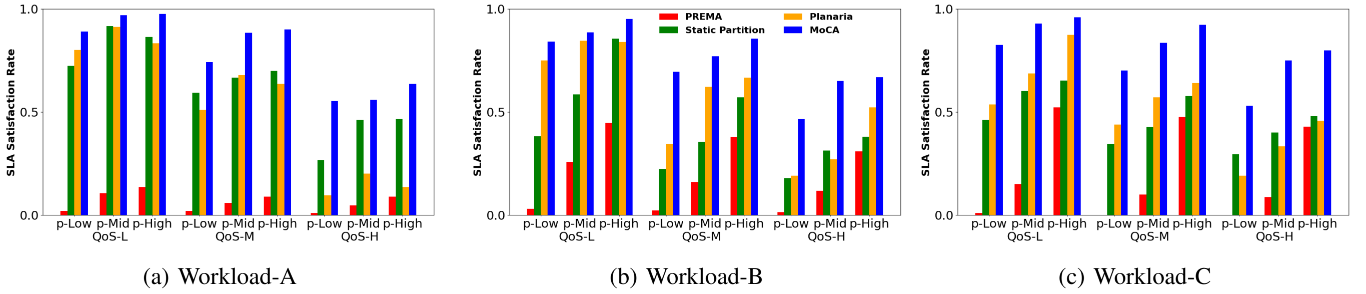
In this section, we evaluate the effectiveness of MoCA-enabled multi-tenant execution in comparison to the three baselines discussed in Section IV. In particular, we compare the effectiveness of MoCA to Prema [9] and Planaria [18], both of which are recent works improving DNN multi-tenant execution.

We demonstrate that MoCA meets the target QoS of different workloads with different priority levels, and we compare our fairness metrics to other baseline solutions. In addition to increasing SLA satisfaction rates, MoCA also improves resource utilization and system throughput, across a wide variety of workload scenarios with different DNN models and QoS requirements. Finally, in addition to the above performance improvements, we also demonstrate that implementing the MoCA's hardware components imposes only a small area overhead on DNN accelerators.

A. SLA satisfaction rate

We evaluate three different set of workloads (listed in Table III) with three different QoS targets (Hard: QoS-H, Medium: QoS-M, and Light: QoS-L), resulting in nine different runtime scenarios in total, and measure the SLA satisfaction rate for each of them in comparison to three baselines. As shown in Figure 5, MoCA consistently outperforms all our baseline multi-tenancy execution mechanisms, for all the scenarios tested. Across all scenarios, compared to Prema, MoCA achieves an 8.7× geometric mean improvement, up to a maximum of 18.1×. MoCA's SLA satisfaction rate shows a 1.8× geometric mean improvement than the static partitioning baseline, and up to 2.4× higher in the best cases. Finally, MoCA achieves up to a 3.9× higher SLA satisfaction rate than Planaria, and a 1.8× higher geometric mean.

MoCA's improvement over the baseline solutions is most pronounced for the QoS-H scenario, where the SLA target is the hardest to achieve, demonstrating the effectiveness



(a) Workload-A (b) Workload-B (c) Workload-C

Fig. 6: SLA satisfaction rate breakdown of different priority groups (p-Low: low priority, p-Mid: mid priority, p-High: high priority) and different DNN workload sizes (Workload-A: light models, Workload-B: heavy models, Workload-C: all models).

of MoCA in adaptively partitioning resources to meet the performance targets in multi-tenant scenarios. For more lenient targets, Planaria performs better than the static baseline, but its performance shows a steep degradation as the required QoS level increased. This is especially true for smaller DNN models, such as light models in Workload-A, where the number of resource reconfigurations increases, causing the cost of thread migrations in compute core repartition to be significantly more pronounced. We observe that thread migration in compute thread repartitioning takes 1M cycles on average due to thread spawning and synchronization, similar to what was observed in recent work [33], which poses significant overhead to relatively short-running smaller models. On the other hand, memory-partitioning in MoCA only takes 5-10 cycles to reconfigure the DMA's issue rate. Hence, MoCA triggers memory repartitioning more frequently than compute repartitioning to avoid its high overhead.

On the other hand, when heavier workloads are grouped together, as in Workload-B, Planaria performed better than the static baseline, but its improvement is not as great as with MoCA, especially as the QoS requirement tightens. This is because the majority of Workload-B's workloads are memory intensive, undesirably degrading the performance of co-running layers. Therefore, solutions that attempt to maintain the performance of these heavier workloads by reconfiguring the compute resources only at runtime can trigger frequent reconfigurations without much benefit, as slowdowns caused by memory contention cannot be resolved by reconfiguring only the compute resources.

Instead, MoCA dynamically detects memory contention and modulates memory access, enabling it to adapt to the requirements of large colocated DNN models more effectively than compute resource reconfiguration. In addition, reconfiguring MoCA hardware does not require expensive thread migration, as it only requires issuing new hardware configuration commands to the accelerator. Thus, MoCA achieves better performance than prior works with its intelligent memory contention management and infrequent thread migration overhead.

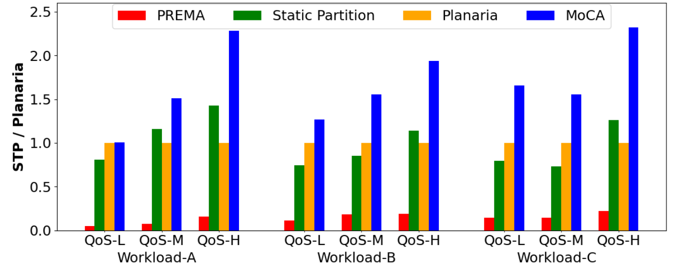


Fig. 7: STP improvement of MoCA over evaluated multi-tenancy baselines (normalized to Planaria baseline) with different QoS targets and different DNN workload sizes.

B. *Priority analysis*

We further analyze SLA satisfaction results by dividing them into different priority groups based on user-given priority scores. We classify workloads priority based on the distribution from [37], where priority scores range from 0 to 11. For visualization purposes, we grouped priorities into several categories: priority-low (p-Low) for scores 0 to 2, priority-mid (p-mid) for scores 3 to 8 and priority-high (p-high) for scores 9 to 11. Figure 6 shows the satisfaction rate grouped by different priority levels. For the p-High groups, MoCA improve the SLA satisfaction rate to 4.7× over Planaria (for Workload-A, QoS-H), 1.8× (for Workload-C, QoS-H) over the static partitioning baseline and 9.9× over Prema (for Workload-A, QoS-M). MoCA and the baselines all show a general trend to increase SLA satisfaction rates as workload priorities increased. In particular, MoCA is the only system that consistently delivers reliable performance across all workload, QoS, and priority scenarios, while other systems show performance degradation in difference cases. For example, for Workload-A, Planaria achieves worse performance for p-High workloads than p-Mid workloads, as aggressive claiming more compute resources for high-priority workloads leads to expensive thread migration costs with little performance benefits.

C. *Throughput analysis*

Figure 7 shows the system throughput improvement that MoCA achieves over the baseline multi-tenancy strategies, normalized to Planaria's STP. MoCA achieves a 12.5× geometric

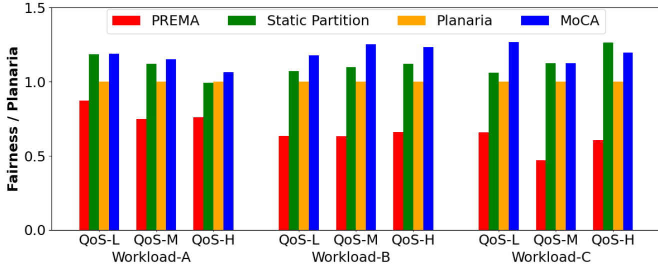


Fig. 8: Fairness improvement of MoCA over evaluated multi-tenancy baselines (normalized to Planaria baseline) with different QoS targets and different DNN workload sizes.

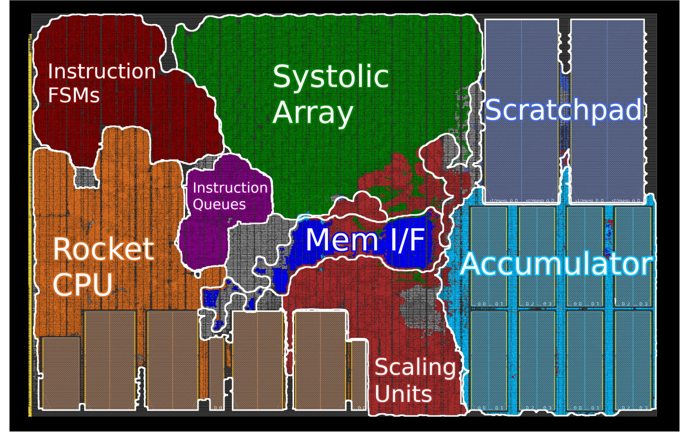


Fig. 9: Layout of an accelerator tile with MoCA.

mean improvement over Prema, and up to a 20.5× maximum improvement (for Workload-A, QoS-L and Workload-A, QoS-M). Compared to the static partitioning baseline, MoCA achieves a 1.7× higher geometric mean, with up to 2.1× improvement in certain workloads (Workload-C, QoS-M). Finally, MoCA achieves a 1.7× geometric mean improvement over Planaria, with a 2.3× maximum improvement (Workload-C, QoS-H and Workload-A, QoS-H).

Workload-A (w/ light models) shows the most improvement on MoCA compared to Planaria, with most of this improvement resulting from MoCA's less frequent threading overhead. Planaria shows poor throughput for the QoS-H scenario on lighter DNN models, as it is unable to utilize its compute resources effectively during thread migrations, whose overhead is comparable to the actual runtime of the lighter models.

For the Workload-B group (with heavy models), MoCA adaptively modulates the contention between heavy DNN models, leading to an overall improvement in STP. The MoCA runtime system dynamically detects contention during runtime, and the MoCA hardware can then resolve contention by throttling excessive memory accesses from memory-bounded layers up to a limit, calculated based on priority and slack time. With this control over the memory access rate, MoCA can facilitate the progress of co-running applications better than the baseline alternatives, increasing the overall STP.

Workload-C (with all models) shows further improvements, mainly because it included a better mix of memory- and compute-bound co-located DNN layers. Layer grouping between memory-intensive layer groups and compute-bound layer groups later in the task scheduling queue helps increase PE array utilization, resulting in higher system throughput. Although Planaria does compute resource repartitioning, its effect is limited when the performance degradation originates from memory contention, since its scheduler does not consider the memory requirements of co-running applications. In this case, multiple memory-intensive layers can be scheduled concurrently, leading to suboptimal performance.

D. Fairness analysis

We evaluate the fairness of colocation, as defined in Section IV, to show that MoCA improves the throughput at the system level without harming the overall fairness of the system. Figure 8 compares the fairness of MoCA and the three other baseline systems. Compared to Prema, MoCA shows a geometric mean improvement of 1.8×, up to a maximum 2.4× improvement (Workload-C, QoS-M). MoCA shows a 1.07× geometric mean improvement over to the static partitioning baseline, and 1.2× (Workload-C, QoS-L) improvement at most. Finally, compared to Planaria, MoCA shows a 1.2× geometric mean improvement while 1.3× (Workload-C, QoS-L) maximum improvement. In general, MoCA shows improved fairness compared to all baselines. Furthermore, the fairness benefit of MoCA was most pronounced for Workload-B, where memory intensive layers are populated as highest rate. MoCA's contention monitoring and resolving mechanism help relieve system level contention so that co-running applications of the memory bounded layers not to be unequally starved. With memory modulation, MoCA fairness outweighs static partitioning counterparts, which does not exploit contention management.

We also observe that MoCA delivers slightly lower fairness compared to the static partition system in the case for Workload-C with all the DNN networks. This is due to the fact that when there are a diverse set of layers available with different compute and memory requirements, MoCA's memory-aware scheduler tends to group short-running, compute-intensive layers together with long-running, memory-intensive layers to maximize the system-level throughput with a balanced workload combination, as demonstrated in Figure 7. At the same time, this may make some of the lower-priority, short-running models finish earlier than needed, hence, lower the fairness metric slightly.

E. Physical design and area analysis

We synthesized a MoCA-enabled DNN accelerator using Cadence Genus with GlobalFoundries' 12nm process technology. We also place-and-routed the MoCA-enabled DNN accelerator using Cadence Innovus, as shown in Figure 9. The MoCA hardware is implemented in the memory-interface of the accelerator.

As shown in Table IV, MoCA adds only a small area overhead to the DNN accelerator, increasing the size of the

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%

TABLE IV: Area breakdown of an accelerator tile with MoCA.

accelerator's memory interface by just 1.7%, and the area of the entire accelerator (including functional units) by 0.02%.

VI. Conclusion

An adaptive, memory-centric multi-tenancy accelerator architecture for DNN workloads, MoCA, has been proposed, implemented and analyzed in this paper. Differing from the existing solutions that aim to partition the compute resources of DNN hardware, MoCA focuses on dynamically managing the shared memory resources of colocated applications to ensure satisfying QoS targets while considering priority difference. MoCA leverages the regularity of DNNs and accelerators to estimate and manipulate the usage of memory resources based on applications' latency targets and user-defined priorities. Our thorough evaluation on a diverse set of DNN execution scenarios and varied target constraints demonstrates that MoCA can increase the SLA satisfaction rate up to 3.9×, with a geometric mean speedup of 1.8× for overall workloads, as well as up to 4.7× for high priority workloads over prior works. MoCA also increases the overall system throughput up to 2.3× while also improving fairness up to 1.3×. Finally, MoCA provides these benefits while incurring less than 1% area overhead of a state-of-the-art DNN accelerator.

VII. Acknowledgements

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.

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Design Automation Conference*, 2012.

[2] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[3] S. Bansal, V. Tolani, S. Gupta, J. Malik, and C. Tomlin, "Combining optimal control and learning for visual navigation in novel environments," 2019.

[4] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, 2018.

[5] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. J. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018.

[6] Q. Chen, H. Yang, M. Guo, R. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," *ACM SIGARCH Computer Architecture News*, vol. 45, 04 2017.

[7] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016.

[8] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang, "Memory latency reduction via thread throttling," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[9] Y. Choi and M. Rhu, "Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units," *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[10] H. Cook, M. Moreto, S. Bird, K. Dao, D. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," *ACM SIGARCH Computer Architecture News*, vol. 41, 06 2013.

[11] G. Da Costa, L. Grange, and I. De Courchelle, "Modeling and generating large-scale google-like workload," in *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, 2016.

[12] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013.

[13] ——, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2014.

[14] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, 2018.

[15] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems," *ACM Transactions on Computer Systems*, vol. 30, 2012.

[16] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, 2008.

[17] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Design Automation Conference*, 2021.

[18] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[19] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[21] R. Huang, J. Pedoeem, and C. Chen, "Yolo-lite: A real-time object detection algorithm optimized for non-gpu computers," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE Computer Society, dec 2018.

[22] M. Huzaifa, R. Desai, X. Jiang, J. Ravichandran, F. Sinclair, and S. V. Adve, "Exploring extended reality with ILLIXR: A new playground for architecture research," *IISWC*, 2021.

[23] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360*, 2016.

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy,

J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[25] M. Kandemir, S. P. Muralidhara, S. H. K. Narayanan, Y. Zhang, and O. Ozturk, "Optimizing shared cache behavior of chip multiprocessors," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[26] S.-C. Kao and T. Krishna, "Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[27] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[28] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. USA: IEEE Computer Society, 2004.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012.

[30] H. Kwon, L. Lai, M. Pellauer, Y.-H. Chen, T. Krishna, and V. Chandra, "Heterogeneous dataflow accelerators for multi-dnn workloads," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[31] B. Li, L. Zhao, R. R. Iyer, L.-S. Peh, M. Leddige, M. Espig, S. E. Lee, and D. Newell, "Coqos: Coordinating qos-aware shared resources in noc-based socs," *J. Parallel Distributed Comput.*, vol. 71, 2011.

[32] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the last line of defense before hitting the memory wall for cmps," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, 2004.

[33] Z. Liu, J. Leng, Z. Zhang, Q. Chen, C. Li, and M. Guo, "Veltair: Towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, 2022.

[34] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[35] J. Mars, N. Vachharajani, R. Hundt, and M. L. Souffa, "Contention aware execution: Online contention detection and response," in *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2010.

[36] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia, "Mlperf training benchmark," in *Proceedings of Machine Learning and Systems*, vol. 2, 2020.

[37] P. Minet, É. Renault, I. Khoufi, and S. Boumerdassi, "Analyzing traces from a google data center," in *2018 14th International Wireless Communications Mobile Computing Conference (IWCMC)*, 2018.

[38] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[39] ——, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *2008 International Symposium on Computer Architecture*, 2008.

[40] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[41] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[42] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006.

[43] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, 2007, pp. 245–258.

[44] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[45] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[46] F. Sijstermans, "The NVIDIA Deep Learning Accelerator," in *Hot Chips*, 2018.

[47] N. Sünderhauf, O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford *et al.*, "The limits and potentials of deep learning for robotics," *The International Journal of Robotics Research*, vol. 37, 2018.

[48] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[49] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, 2012.

[50] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: reactive static/dynamic compilation for qos in warehouse scale computers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[51] R. Tang and J. Lin, "Deep residual learning for small-footprint keyword spotting," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[52] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, "Machine Learning at Facebook: Understanding Inference at the Edge," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[53] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: Qos for accelerators in machine learning platforms," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

APPENDIX

A. Abstract

This artifact appendix section describes how to access the artifacts for each MOCA component, exercise and evaluate it as in Section V. As in Section IV, we will use FireSim FPGA-accelerated simulations to evaluate MOCA in a full-stack environment.

B. Artifact meta-information checklist

- **Runtime environment: AWS FPGA Developer AMI 1.12.1**
- **Hardware: AWS EC2 instances (c5.4xlarge, f1.2xlarge)**
- **How much disk space is required?: 200 GB (on EC2 instance).**
- **Experiments: FireSim simulations of MOCA incorporated into a RISC-V-based DNN accelerator infrastructure, running multi-tenant inference queries.**
- **Program: Chisel (RTL), C (Runtime, Scheduler), Python (Script)**
- **Metric: Target satisfaction rate (rate of each query that meets the target deadline), STP (System Throughput), and Fairness as defined in Section IV-C.**
- **Output: Parsed result from UART output of SoC, performance (SLA satisfaction rate, STP, Fairness) comparison bar plot between static partitioning baseline and MOCA for each QoS level and Workload sets. (Figure 5-8)**
- **How much time is needed to prepare the workflow?: 2 hours (scripted installation).**
- **How much time is needed to complete experiments?: 4 hours (scripted run, scripted result parsing)**
- **Publicly available: Yes.**
- **Code licenses: Several, see download.**

C. Description

1) How to access: The artifacts consist of:

1) FireSim: Top-level FPGA-Accelerated RTL Simulation Environment (https://doi.org/10.5281/zenodo.7456139)
2) Chipyard: RISC-V SoC generation environment (https://doi.org/10.5281/zenodo.7456073)
3) MOCA Hardware: MOCA implementation on Gemmini DNN accelerator (https://doi.org/10.5281/zenodo.7456052). The main implementation is on DMA.
4) MOCA Software: MOCA runtime and scheduler implementation, and tests. (https://doi.org/10.5281/zenodo.7456045). MOCA runtime and schedulers are under `imagenet` and `include`.

Users need not download the latter three repositories manually—they will be obtained automatically from Zenodo when the FireSim repository is set up.

2) Dependencies - Hardware: One AWS EC2 c5.4xlarge instance (also referred to as "manager" instance), and three f1.2xlarge instances are required (we split the workload to run on three parallel f1 instances to save runtime, which takes more than 8 hours if run on a single instance). The latter will be launched automatically by FireSim's manager. We have provided pre-built FPGA images to avoid the long latency (~8 hours) of the FPGA-built process. However, if users want to build custom FPGA images, one additional z1d.2xlarge is required.

3) Dependencies - Software: Use ssh or mosh on your local machine to remote access EC2 instances. All other requirements are automatically installed by scripts in the following sections.

D. Installation

First, follow the instructions on the FireSim website[2] to create an EC2 manager instance. You must complete up to and including "Section 1.3.1.2: Key Setup, Part 2", with the following recommendations on "Section 1.3.1":

1) When instructed to launch c5.4xlarge or z1d.2xlarge, select c5.4xlarge.
2) When entering the root EBS volume size, using 200GB is sufficient.

Once you have completed up to and including "Section 1.3.1.2 Key setup" in the FireSim docs, you should have a manager instance set up, with an IP address and key. Either do ssh or mosh to log in to the instance. From this point, all commands should be run on the manager instance.

For artifact evaluation, please clone the forked FireSim repository by running the following (else, users can clone FireSim and checkout the `MOCA` branch in the Gemmini submodule):

For artifact evaluation, begin by downloading the top-level FireSim repository from Zenodo:

```
  $ wget -O firesim-moca-ae.zip https://zenodo.org/
record/7456139/files/firesim-moca-ae.zip
  $ unzip firesim-moca-ae.zip
```

Next, run the following, which will initialize all dependencies and run the FireSim and Chipyard setup steps (RISC-V toolchain installation, matching host toolchain installation, etc.):

```
  $ cd firesim
  $ ./first-clone-setup-fast.sh
```

After the script finishes running, run the following:

```
  $ source sourceme-f1-manager.sh
```

After sourcing, complete the steps in "Section 1.3.3 Completing Setup Using the Manager". Once these steps have been completed, you are fully ready to evaluate MOCA.

E. Experiment Workflow

Now that our environment is set up, we will run MOCA artifact. First, we will begin with building the workload for MOCA. When building the workload image, users can either compile the Linux binary themselves or can use provided pre-compiled binary.

[2]https://docs.fires.im/en/1.15.1/Initial-Setup/index.html

1) Building Linux image containing workload:

1) On the manager instance, build the FireSim-compatible RISC-V Linux image using a buildroot-based Linux distribution. Follow the instruction on "Section 2.1.1" of FireSim documentation.[3]
2) Run the following to create build directory at `gemmini-rocc-tests`.

```
$ cd firesim/target_design/chipyard/generators
$ cd gemmini/software/gemmini-rocc-tests
$ ./build.sh
```

3) (Optional - If the users want to compile the binaries themselves, please skip this process) To test provided pre-compiled binaries, unzip `imagenet-binary.zip` to `build/imagenet` by running the following commands. Comment out all the tests under `test` in `imagenet/Makefile` to prevent pre-compiled binaries being overwritten.

```
$ cd build
$ rm -rf imagenet/
$ unzip ../imagenet-binariy.zip
$ cd ../
```

4) Run the following command to build the MoCA runtime and scheduler which are written in C on a full Linux environment. Running the commands will generate a `.json` file in `firesim/deploy/workloads`.

```
$ cd ../
$ ./build-gemmini-workload.sh
```

2) Running FireSim simulation:

1) Go to `firesim/deploy`, and within `config_hwdb.yaml`, paste the pre-built FPGA image entry provided in `built-hwdb-entries/`. Set this in `default_hw_config` in `config_runtime.yaml` as well.
2) For other configurations in `config_runtime.yaml`, see "Section 2.1.2. Setting up the manager configuration" of FireSim documentation.
   The following parameters must be modified:
   a) `workload_name: gemmini-tests-workload.json`.
   b) Increase the number of f1.2xlarge to boot to 3 (`f1.2xlarge: 3`), under `run_farm_hosts_to_use`.
   c) Replace `topology: three_no_net_config` under `target_config`.
   d) Set `no_net_num_nodes: 3` to launch three f1 instances running in parallel.
3) Run FireSim simulations by launching f1.2xlarge instances. Follow the instructions on "Section 2.1.3 Launching a Simulation!" of the FireSim documentation.
4) The result will be copied to a directory in `deploy/results-workload`. The generated result directory will consist of three sub-directories, each for workload-A/B/C defined on Table III, as each f1 instance run each workload type. Running the following commands on `results-workload` directory will generate figures for each workload set. Run the first command from the following command block to parse the results in the priority level group, as well as the general results as Figure 6 and Figure 5 per each QoS level and workload model set. The second command is to parse the STP and Fairness results as in Figure 7 and Figure 8. The result directory on the script (`$result_dir`) should be the directory that includes three sub-directories.

```
$ ./build_sla.sh ($result_dir)
$ ./build_stp.sh ($result_dir)
```

The test scripts will run the static partition baseline and MoCA on 250 end-to-end turns of inference queries that are dispatched randomly, with randomly assigned priority. Note that this script will not rebuild FPGA images for the system by default, since each build takes around 8 hours. We instead provide pre-built images by default on `built-hwdb-entries/`, which is used on the paper's evaluation.

The result parsing scripts that are used in build commands (`build_sla.sh` and `build_stp.sh`) are `parse_result_from_uartlog.py` and `make_fair.py`, respectively. Those Python scripts and build scripts are included in `results-workload` directory.

We provide example result plots of the pre-compiled binaries under `EXAMPLE_RESULT/` for reference.

Please make sure the running f1 instance is terminated by running `firesim terminaterunfarm`, and confirm in your AWS EC2 management console that no instances remain beside the manager.

F. Experiment customization

1) Rebuilding FPGA image: Users can change the SoC configuration by changing Gemmini DNN accelerator configuration or other SoC configuration. For example, on `Config.scala` of Gemmini `src`, users can reconfigure the internal scratchpad or accumulator size. The shared L2 size can be changed by modifying cache parameters at `RocketConfigs.scala`. For building a new FPGA bitstream, please follow the steps in "Section 3. Building Your Own Hardware Designs"[4].

2) Customizing experiment parameters: Go to `gemmini/software/gemmini-rocc-tests/include`. In the directory, header files are included for MoCA runtime, software, and parameters. Open `gemmini.h`, and change `SEED` to change the seed to generate new randomized queries. To change the number of queries, change `total_workloads`. Repeat Section E to get the result.

[3] https://docs.fires.im/en/1.15.1/Running-Simulations-Tutorial/Running-a-Single-Node-Simulation.html

[4] https://docs.fires.im/en/1.15.1/Building-a-FireSim-AFI.html