AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads

Seah Kim
University of California, Berkeley
Berkeley, CA, USA
seah@berkeley.edu

Jerry Zhao
University of California, Berkeley
Berkeley, CA, USA
jzh@berkeley.edu

Krste Asanović
University of California, Berkeley
Berkeley, CA, USA
krste@berkeley.edu

Borivoje Nikolić
University of California, Berkeley
Berkeley, CA, USA
bora@berkeley.edu

Yakun Sophia Shao
University of California, Berkeley
Berkeley, CA, USA
ysshao@berkeley.edu

ABSTRACT

With the widespread adoption of deep neural networks (DNNs) across applications, there is a growing demand for DNN deployment solutions that can seamlessly support multi-tenant execution. This involves simultaneously running multiple DNN workloads on heterogeneous architectures with domain-specific accelerators. However, existing accelerator interfaces directly bind the accelerator's physical resources to user threads, without an efficient mechanism to adaptively re-partition available resources. This leads to high programming complexities and performance overheads due to sub-optimal resource allocation, making scalable many-accelerator deployment impractical.

To address this challenge, we propose AuRORA, a novel accelerator integration methodology that enables scalable accelerator deployment for multi-tenant workloads. In particular, AuRORA supports virtualized accelerator orchestration via co-designing the hardware-software stack of accelerators to allow adaptively binding current workloads onto available accelerators. We demonstrate that AuRORA achieves 2.02× higher overall SLA satisfaction, 1.33× overall system throughput, and 1.34× overall fairness compared to existing accelerator integration solutions with less than 2.7% area overhead.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; **Distributed architectures**; **Neural networks**; • **Hardware** → **Communication hardware, interfaces and storage**; **Application-specific VLSI designs**.

KEYWORDS

Multi-tenant system, Multi-core, Accelerators, Resource Management, SoC Integration, Microarchitecture, Machine Learning

ACM Reference Format:
Seah Kim, Jerry Zhao, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3614280

1 INTRODUCTION

With the slowdown in technology scaling, architects have turned to heterogeneous multi-core many-accelerator system-on-chips (SoCs) to meet the increasing compute demands of modern workloads [25]. One particular class of applications that drives the development of many-accelerator systems is deep neural networks (DNNs). Specifically, the concurrent multi-tenant execution of DNN applications, where multiple DNN workloads are co-located on the same SoCs, has become crucial for both the cloud [44, 46, 49, 51, 59] and edge devices [22, 27, 35] to meet stringent throughput and latency service-level agreements (SLAs). Previous research has underscored the importance of spatially co-locating DNN workload executions to improve the quality of service (QoS) [21, 31, 40].

However, performance variability due to contention for shared hardware resources presents a substantial challenge for these workloads. More specifically, multi-tenant systems require a flexible and efficient mechanism to dynamically partition shared resources based on application requirements and available resources. While shared-resource management for multi-core processors has been a well-studied area in computer architecture, less attention has been paid to the accelerator interface, i.e., how accelerators interact with CPUs and the system stack.

In particular, existing accelerator integration approaches restrict options for run-time accelerator management, as workloads or threads are explicitly bound to physical accelerators [12, 31, 40] or subarrays [21, 36]. Challenges arise when the system load of the application is unknown prior to execution or when the application runs complex cascaded pipelines [27, 32, 35]. In these scenarios, kernel drivers must either explicitly preempt [12, 21] user threads, leading to high thread migration cost, or wait for user threads to complete and release their resources [31, 40], resulting in suboptimal resource partitioning and utilization. New methods have been proposed to develop virtualized interfaces aimed at reducing kernel driver overhead in accelerator deployment [14, 47]. However, these

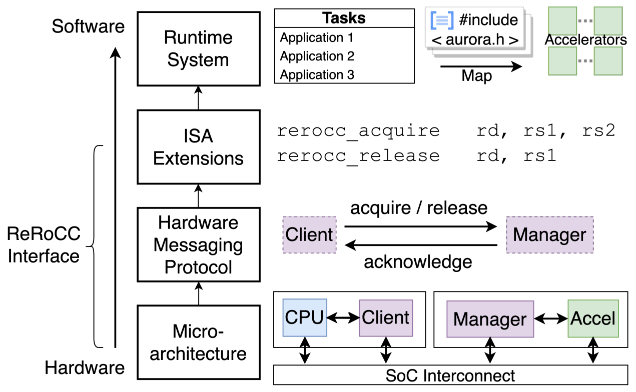


Figure 1: AuRORA is a full-stack accelerator integration methodology for scalable accelerator deployment.

approaches primarily focus on queue-based, first-come-first-serve accelerator scheduling, lacking the capability for user threads to dynamically re-partition accelerators during runtime.

To address these challenges, this work presents AuRORA[1], a full-stack methodology for integrating accelerators in a scalable manner for multi-tenant execution. AuRORA consists of ReRoCC[2] (Remote RoCC), a virtualized and disaggregated accelerator integration interface for many-accelerator integration, and a runtime system for adaptive accelerator management. Similar to virtual memory systems that provide an abstraction between user memory and physical machine resources, AuRORA provides an abstraction between the user's view of accelerators and the physical accelerator instances. AuRORA's virtualized interface allows workloads to be flexibly orchestrated to available accelerators based on their latency requirements, regardless of where accelerators are physically located. This is particularly crucial for multi-tenant execution since resources must be dynamically reallocated to meet the distinct demands of concurrent workloads. To effectively support virtualized accelerator orchestration, AuRORA delivers a full-stack solution that co-designs the hardware and software layers, as shown in Figure 1, with the goal of delivering scalable performance for heterogeneous systems with multiple accelerators. Specifically, the AuRORA stack, from bottom to top, includes:

- Low-overhead **shim microarchitecture** to interface between cores and accelerators,
- A hardware **messaging protocol** between CPU and accelerators to enable scalable and virtualized accelerator deployment on SoCs,
- **ISA extensions** to allow user threads to interact with the AuRORA hardware in a programmable fashion, and
- A lightweight **software runtime** to dynamically reallocate available resources for multi-tenant workloads.

We evaluate AuRORA in two different SoC configurations; **AuRORA-crossbar**, where all components are connected with a crossbar, and **AuRORA-NoC**, which uses a 2D mesh network-on-chip (NoC). Our evaluation demonstrates that AuRORA improves the overall SLA satisfaction rate by up to 3.99× (2.41× in geomean) for the NoC scenario and up to 3.32× (2.02× geomean) for the crossbar scenario, together with an improvement in system throughput of 2.04× (1.79 × geomean) for NoC and 1.38× (1.33× geomean) for crossbar, and fairness of 1.61× (1.41× geomean) for NoC and 1.68× (1.34× geomean) for crossbar, compared to prior work [31, 40], while incurring less than 2.7% hardware area overhead.

2 BACKGROUND AND MOTIVATION

This section discusses challenges with running multi-tenant DNN workloads and how existing approaches for accelerator integration are insufficient for addressing these challenges.

2.1 Multi-tenant DNN Execution

Multi-tenancy refers to the scenario where multiple tasks share hardware, leading to contention for system resources such as compute and memory. Shared resource partitioning is a thoroughly explored domain within computer architecture, where novel mechanisms have been proposed to manage multi-core architecture for data centers [11, 56, 57] and, more recently, on GPUs [45].

On the accelerator side, Prema [12] introduced the concept of time-multiplexing a monolithic accelerator across multiple DNN tasks. However, this approach suffers from low hardware utilization for individual DNNs. To address this limitation, spatial co-location of multiple DNN tasks has been proposed, where compute resources [21, 34, 36, 40] or memory resources [29, 31] are spatially partitioned across applications. However, all existing multi-tenant accelerators bind workloads to physical accelerators or subarrays explicitly [12, 21, 36], leading to high-performance overhead when migrating workload threads during accelerator resource reallocation. To avoid the thread migration overhead, recent works use coarser-grained scheduling to reduce the frequency of resource reallocation [31, 40]. However, such coarse-grained scheduling lacks the ability to respond promptly to dynamic changes in system load.

2.2 Physical Accelerator Integration

Table 1 provides a summary of the multi-accelerator integration strategies. We classify existing methods into two main categories: *physical integration*, where workloads are explicitly mapped onto physical accelerators, and *virtual integration*, where programmers interact solely with virtualized accelerators, with the workload-to-accelerator binding managed by a separate integration layer.

On the physical accelerator integration side, the existing space can be broadly categorized into two types: tightly-coupled and loosely-coupled. Tightly-coupled accelerators are directly implemented as part of the core datapath in a general-purpose CPU. Examples of standards for CPU-coupled accelerators include the ARM Custom Instruction interface [13], the RoCC RISC-V accelerator interface [4], and the Tensilica Instruction Extension interface [23].

Since tightly-coupled accelerators can directly access the architectural state in the host thread, software support for these accelerators can be provided in the form of low-overhead userspace-accessible custom instructions, greatly reducing software integration costs. However, tightly-coupled accelerators require expensive

[1]https://github.com/ucb-bar/AuRORA
[2]https://github.com/ucb-bar/rerocc

Table 1: Comparison of multi-accelerator integration methodologies.

	Physical Integration		Virtual Integration	
	ISA Extension	Memory Mapped	Non-Preemptive Allocation	Preemptive Allocation
Provides virtual accelerator abstraction to user threads	no	no	yes	**yes**
Software overhead for accelerator access	low	high	low to mid	**low**
Decoupling accelerators from cores	no	yes	yes	**yes**
Dynamic accelerator resource reallocation	no	yes but slow	no	**yes and fast**
Examples	[4, 13, 23]	[7, 26, 41, 43]	[14, 47]	**AuRORA**

host thread migration when adjusting accelerator affinity, as host threads must be migrated to the appropriate control core for the target accelerator [31, 40]. We measure the accelerator reallocation overhead of physically integrated accelerators when co-running four applications, ResNet50, AlexNet, GoogLeNet, and BERT-small, and observe 300-700K cycle overhead when thread migration happens. Furthermore, physical design challenges and limited instruction encoding space prohibit scaling up the number of accelerators integrated into a single general-purpose core.

The other approach is to decouple the accelerator from the core over the SoC interconnect, most commonly by binding the accelerator to memory-mapped control registers [7, 26, 43]. Attaching accelerators over memory-mapped registers is supported in all standard SoC interconnect protocols, including AMBA protocols [2], TileLink [15], Wishbone [54], and CXL [1]. This allows for scalable accelerator deployment, as many accelerators can be instantiated across a single SoC, each mapped to a unique address range of control registers. Prior work [41] proposed a novel shared-memory management for many-accelerator systems where accelerators are physically integrated with the MMIO interface. However, software support for memory-mapped accelerators is more burdensome, as privileged drivers must make the physical control registers visible to user threads and manage the allocation of accelerators to users, leading to significant performance overhead.

2.3 Virtual Accelerator Integration

The cumbersome physical accelerator integration does not scale to many-accelerator systems running multi-tenant workloads, especially when resources need to be frequently reallocated to meet the distinct demands of applications during execution. To improve scalability, recent research has proposed *virtualized* accelerator integration, which allows user threads to invoke accelerators dynamically without binding workloads to physical accelerators [14, 47]. In particular, both works have proposed ISA extensions and microarchitecture mechanisms to dynamically map user threads to accelerators. However, in addition to being closed-sourced, these efforts only allow non-preemptive resource allocation, i.e., user threads are scheduled onto accelerators in a first-come-first-served fashion using a command queue in hardware. Furthermore, in both these works, the host CPU performs address translation for the accelerator before issuing the memory request to accelerator [47], or the host core to handle TLB misses with OS handler [14]. Both cases invoke software overhead that would prohibit the CPU from performing other tasks.

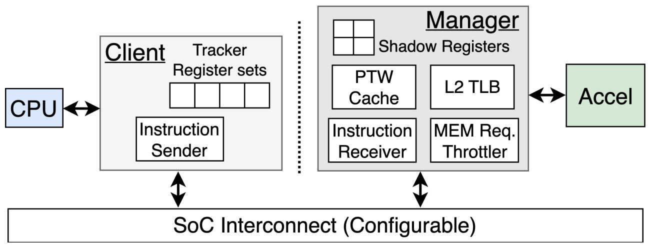


Figure 2: Overview of AuRORA microarchitecture.

Such a simple accelerator allocation approach does not allow dynamic accelerator orchestration, where accelerator resources are flexibly partitioned based on the current demands of concurrent workloads. In particular, dynamic accelerator orchestration through preemptive allocation is required for multi-tenant execution where multiple tasks share the system resource with different target requirements. *To the best of our knowledge, AuRORA is the first work that supports virtualized accelerator integration with dynamic resource allocation for multi-tenant execution.*

3 AURORA ARCHITECTURE

AuRORA is a new full-stack approach of accelerator integration for efficient multi-tenant execution on virtualized accelerators. AuRORA provides an abstraction to the software of *virtualized accelerators*, where user threads invoke virtual accelerators which are then dynamically mapped to physical accelerators by the AuRORA runtime. The following sections discuss the AuRORA microarchitecture (Section 3.1), hardware protocol (Section 3.2), ISA extensions (Section 3.3), and runtime system (Section 3.4).

3.1 AuRORA Microarchitecture

Figure 2 shows the key microarchitecture components of AuRORA `Client` and `Manager` and how they can be seamlessly integrated with existing CPU and accelerator designs.

Client. The AuRORA's `Client` shim integrates with host general-purpose cores to allow communication to and from disaggregated accelerators, while providing the architectural illusion of a tightly-coupled accelerator. Each core tracks which accelerators it has currently reserved using a hardware table in the `Client`. The `Client` is implemented as a RoCC accelerator [4], allowing it to be integrated with existing RoCC-compatible cores like Rocket [5] and BOOM [62].

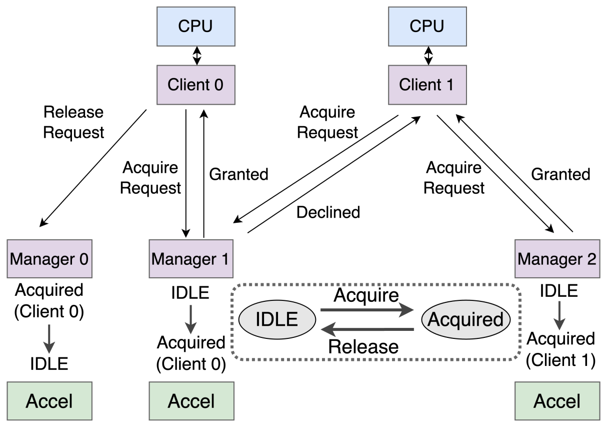


Figure 3: AuRORA's hardware protocol for how a `Client` manages accelerator integrated into `Manager` tiles.

Manager. The AuRORA's `Manager` shim wraps an existing accelerator and facilitates the virtualization and disaggregation of accelerators across the SoC interconnect. The `Manager` receives AuRORA and accelerator commands from the `Client` and forwards accelerator commands to the attached accelerators. The `Managers` also implement a shadow copy of architectural CSRs used by the accelerator MMU. These CSRs include those that describe the host thread privilege level, memory translation mode, and page table address. A page table walker (PTW), an optional PTW cache, and an L2 TLB provide an architecturally compliant memory-management-unit (MMU) to the accelerator. These modules eliminate the need for user- or supervisor-managed IOMMU, preserving the illusion of a shared MMU between the core and accelerators. To support software-managed QoS, the `Managers` also implement configurable traffic throttlers, which can be used to set bandwidth limits on accelerator memory traffic. The bandwidth limit can be set by writing to a configuration register in the `Manager`.

3.2 AuRORA Hardware Protocol

To support the integration and disaggregation of accelerators at the SoC level, AuRORA connects `Client` and `Manager` with the AuRORA hardware communication protocol. Figure 3 shows the AuRORA hardware messaging protocol between `Clients` and `Managers`. There are two states for `Manager`, *IDLE* and *ACQUIRED*. When a `Client` tries to acquire accelerators, it sends the acquire request signal to the `Manager`. If the `Manager` is in the *IDLE* state (e.g., `Client 0` to `Manager 1` in Figure 3), the *acquire* succeeds, and an acknowledgment, (i.e., the granted signal), is sent to the `Client`. The `Client` will then forward its own core's configuration registers to the acquired `Manager` to set up the MMU on the manager as a shadow of the core's. From this point, accelerator instructions issued to the `Client` will be automatically forwarded to the `Manager`.

However, when the accelerator has already been occupied by another process (e.g., `Client 1` to `Manager 1`), the acquire attempt will fail. If there are other accelerators of the same functionality in the system, the `Client` can attempt to acquire another accelerator (e.g., `Client 1` to `Manager 2`). For these cases, the software has to configure the AuRORA `Client` with a set of accelerators that share the same functionality. After the `Client` has finished using this accelerator, it sends a release message to the `Manager` (e.g., `Client 0` to `Manager 0`), returning the accelerator's `Manager` state to *IDLE*. All these transactions are non-blocking to guarantee forward progress.

Table 2: The AuRORA protocol can share the same on-chip interconnect with the memory traffic or use a separate interconnect. All listed combinations are supported in the AuRORA implementation.

		AuRORA Protocol Traffic	
		crossbar	NoC
Memory Traffic	crossbar	Supported (separate)	Supported (separate)
	NoC	Supported (separate)	Supported (separate, shared)

The AuRORA hardware protocol can be mapped onto various interconnect architectures, including crossbar and network-on-chip (NoC), as shown in Table 2. The AuRORA traffic can share the system interconnect with memory traffic or use a separate interconnect to avoid contention. In particular, AuRORA focuses on the interface between the accelerator and the CPU, which is orthogonal to the SoC interconnect standard that defines how data are transferred in SoCs. Our evaluation uses TileLink [15], an SoC interconnect standard that can provide coherent access across SoCs, with a shared global address space, since this is common in many-core/many-accelerator SoCs. AuRORA can also be implemented using other SoC interconnect standards like CXL [1], which enables a global shared memory space between chips for multi-chip integration.

3.3 AuRORA ISA Extensions

The AuRORA ISA extensions expose the virtualized and disaggregated accelerator management to software, as specified in Table 3. The `acquire` and `release` instructions allow `Client` to claim and release accelerators. When claiming an accelerator, `Client` encodes a target physical accelerator `acc_id` in the `acquire` instruction so that it can be delivered to the target `Manager`. If `acquire` succeeds, `Client` will assign a virtual accelerator index `acq_id` to this accelerator, which is then used in the rest of the runtime. The `assign` instruction maps an acquired accelerator to an available opcode on its architectural thread. This allows a single architectural thread to acquire more accelerators than the available opcode space would permit. The `memrate` instruction configures the maximum memory request rate for an accelerator for QoS management.

3.4 AuRORA Runtime

The AuRORA runtime offers mechanisms for provisioning and releasing accelerators using ISA extensions introduced earlier. Specifically, this runtime operates within userspace software, utilizing custom AuRORA instructions accessible in userspace, to adaptively partition available resources for multi-tenant execution. The runtime system is designed to be lightweight and only needs to be invoked when acquiring, configuring, or releasing an accelerator. Furthermore, it is also important to note that the AuRORA runtime maintains backward compatibility with existing RoCC-based [4] accelerator software stacks.

Table 3: AuRORA instructions.

AuRORA Pseudoinst.	Operands	Purpose
`rerocc_acquire`	`success, acc_id, acq_id`	Acquires accelerator and maps it to the local client, returns success status.
`rerocc_release`	`acq_id`	Releases an accelerator currently acquired by the local client.
`rerocc_assign`	`acq_id, opcode`	Maps a currently acquired accelerator to an available instruction opcode.
`rerocc_fence`	`acq_id`	Memory fence between core memory and an acquired accelerator.
`rerocc_memrate`	`acq_id, rate`	Sets the maximum memory request rate the accelerator can make.

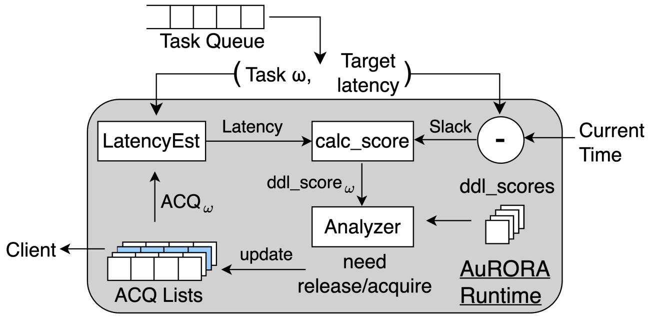


Figure 4: AuRORA runtime takes Task ω and its target latency and reconfigures the acquired accelerators for each `Client`.

To improve the performance of multi-tenant applications, the AuRORA runtime provides support for two key contention-aware partitioning: compute-resource allocation and memory-resource allocation. The compute-resource allocation dynamically partitions different numbers of accelerators for different tasks considering the NUMA effect, while the memory-resource allocation adaptively reconfigures the available memory bandwidth to different accelerators. Unlike prior works where the scheduler explicitly encodes the physical accelerator and the number of accelerators when scheduling tasks [12, 21, 31, 40], the AuRORA runtime manages virtualized accelerator resources and dynamically partitions them during runtime. As a result, a user application only needs to specify its latency target, simplifying its interaction with the AuRORA runtime.

3.4.1 *Compute-resource allocation.*

The AuRORA runtime dynamically re-partitions compute resources based on latency targets and available compute resources. Figure 4 describes how the runtime operates to allocate compute resources. The runtime receives an end-to-end DNN network (i.e., a task) ω from the task queue and is invoked before the execution of every layer. The LatencyEst module estimates the latency of each task based on its current acquired accelerators (ACQ$_\omega$). Together with the remaining Slack to its target deadline, this latency is fed into the calc_score module to calculate its dynamic deadline score (ddl_score), which indicates the likelihood of meeting the target deadline (i.e., a higher score indicates it is more likely to meet its deadline). The analyzer compares the dynamic ddl_score$_\omega$ of this task against those of other on-going tasks (ddl_scores) and decides whether task ω requires the release or acquisition of accelerators to meet their performance targets while balancing system throughput and fairness. Finally, the runtime notifies the task thread’s `Client` of the changes so that the `Client` can acquire or release accelerators based on the updated assignment from the AuRORA runtime.

Algorithm 1 AuRORA virtual accelerator allocation

```
    ▷ Functions: LatencyEst (estimates latency of task)
1:      MemoryPartition (calculates the target MEM request rate)
    ▷ Lists: Accels (accelerators in the system)
2:      ACQ (acquired virtualized accelerators)
3:      ddl_scores (dynamic deadline scores across tasks)
    ▷ Inputs: ω (task), target latency
    ▷ Outputs: Decision to release, acquire of Manager to Client
4:  Function calc_score(task, ACQ):
    ▷ %Computes dynamic score for accelerator partition%
5:      Slack ← time_left_to_target
        ▷ % Quantifies confidence in meeting the deadline %
6:      ddl_score ← Slack/LatencyEst(task, ACQ)
7:      return ddl_score
8:  for Layer_i in Layers do
9:      ddl_score_ω ← calc_score(ω, ACQ_ω)
        ▷ % Analyzer determines acquire/release %
        ▷ % num_accel: # of accelerators to release or acquire %
10:     (need_release, need_acquire, num_accel)
11:                 ← Analyze (ddl_scores, ddl_score_ω)
12:     if need_release then
13:         for iter in num_accel do
14:             release_acc ← ACQ_ω.pop()
15:             rerocc_release(release_acc)
16:             Accels(release_acc) ← IDLE
17:     if need_acquire then
18:         for iter in num_accel do
19:             idle_acc ← Accels.idle
20:             rerocc_acquire(idle_acc, acq_id_new)
21:             ACQ_ω.push(acq_id_new)
        ▷ % For memory resource optimizations %
22:     max_mem_rate ← MemoryPartition(Layer_i)
23:     rerocc_memrate(ACQ_ω, max_mem_rate)
24:     runLayer(Layer_i) && ω.popLayer(Layer_i)
```

Algorithm 1 further elaborates on this process. Upon invocation, the runtime calculates the dynamic deadline score, ddl_score, of each task on its slack. The runtime compares the ddl_score$_\omega$ of the current task with the scores of other concurrently running tasks to whether the release or acquisition of accelerators needs to happen and the number of accelerators affected. We use the latency estimation technique from [31] that considers the multi-level memory hierarchy, the number of processing elements, and per-layer compute-to-memory ratios for individual DNN execution, similar to other multi-tenant DNN execution work [12, 21].

The Analyze function in the AuRORA runtime compares the score of task ω with the scores of other tasks to decide whether ω needs to release its acquired accelerator or acquire other idle ones, based on the relative confidence in meeting the deadline target. If `release` is necessary, the runtime releases acquired accelerators, so that tasks with tighter deadlines can acquire them. If `acquire` is needed, the runtime would try to acquire idle accelerators. All of this happens in user-space code. Thus, unlike prior works [12, 21, 31, 40], AuRORA's accelerator scheduling does not require thread preemption, synchronization, and migration to reallocate the accelerator.

3.4.2 NUMA-aware compute partitioning.

Distributing accelerators and memory across an SoC's network-on-chip (NoC) interconnect inevitably causes non-uniform memory accesses (NUMA) [16, 39], which adds to system heterogeneity. Alleviating the challenges of NUMA memory systems has been well-researched in the multi-core domain [9, 17, 37, 39]. Notably, prior work has proposed scheduling by application bandwidth sensitivity as a mechanism to reduce interference in a shared multi-core or multi-accelerator system [16, 53]. Previous work has also found that thread migration overhead presents a significant challenge for such NUMA-aware thread scheduling approaches [9].

The AuRORA runtime leverages its virtual accelerator abstraction to enable simple but efficient NUMA optimization. Different workloads would face varying degrees of NUMA effect on each NoC node with NoC-based interconnect. To capture the performance slowdown caused by NUMA effects, we build an empirical performance model based on hardware measurements to capture each workload's sensitivity to NUMA. The AuRORA runtime quantifies each task's slowdown caused by the NUMA effect based on its assigned accelerators.

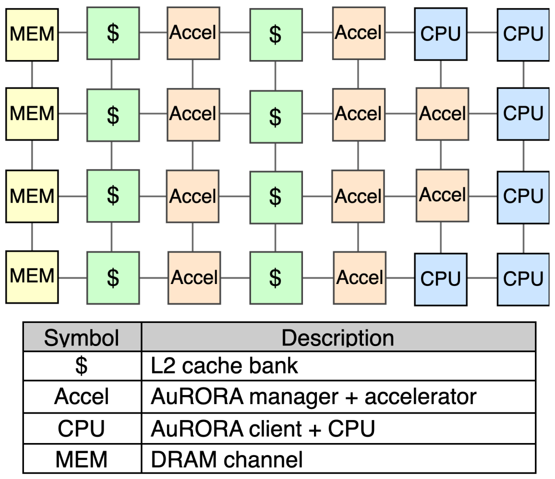
When deciding on new accelerators to acquire, the AuRORA runtime compares the relative NUMA slowdown to co-running tasks across different accelerator assignments and assigns the set of accelerators that causes the lowest relative slowdown for each task. This allows the runtime to allocate the resources to the task thread to minimize the overall system's latency degradation due to the NUMA effect. In addition, AuRORA runtime performs accelerator swapping optimization before running the layer if there are idle accelerators in the system with a lower relative NUMA slowdown for the task. This swap is implemented as an atomic series of `acquire` and `release`. When the system does not exhibit NUMA properties, for example, if the interconnects are configured as crossbar, the NUMA optimization is not enabled.

3.4.3 Memory-resource allocation.

AuRORA also supports dynamic memory re-partition, as shown in Algorithm 1 Line 22-23. It dynamically detects system-level interference and sets limits on the memory access rates of accelerators to resolve contention if necessary. AuRORA's memory re-partitioning methodology with dynamic scoring and run-time contention detection is implemented similarly to prior work [31]. Upon detection of contention over memory bandwidth, AuRORA runtime triggers `Client` to send the `memrate` instruction to the acquired `Manager` to configure the memory access rate for each instruction. Based on the configured value, `Manager` would limit the memory requests from the target accelerator.

Table 4: SoC configurations used in the evaluation.

Parameter	Value
Systolic array dimension (per tile)	16x16
Scratchpad size (per tile)	128KB
Accumulator size (per tile)	128KB
# of accelerator tiles	10
Shared L2 size	2MB
Shared L2 banks	8
DRAM bandwidth	32GB/s
Frequency	1GHz



Symbol	Description
$	L2 cache bank
Accel	AuRORA manager + accelerator
CPU	AuRORA client + CPU
MEM	DRAM channel

Figure 5: AuRORA's NoC-based interconnect setup.

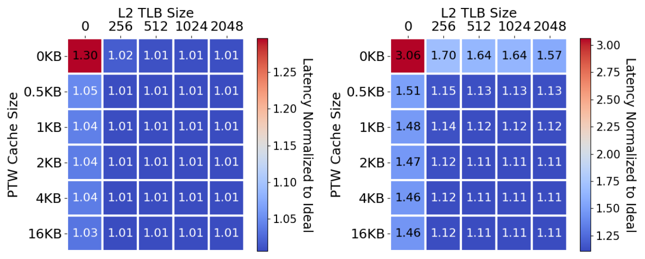
4 METHODOLOGY

This section details AuRORA's implementation, together with the workloads and metrics used for our evaluation.

4.1 AuRORA Implementation

We implement the AuRORA microarchitecture using the Chisel HDL [6] on top of the Chipyard [3] SoC framework, an open-source framework for designing and evaluating systems-on-chips. We use Gemmini [20], a systolic-array-based DNN accelerator without multi-tenancy support, as a representative DNN accelerator in our evaluation. Additionally, we implement an AuRORA protocol adapter for the Constellation [61] NoC generator, to enable evaluations on systems with a NoC-based interconnect. We evaluate AuRORA's performance of running end-to-end DNN workloads using FireSim, a cycle-exact, FPGA-accelerated RTL simulator [30].

Table 4 shows the SoC configuration we use in our evaluations of AuRORA. To demonstrate how AuRORA scales to realistic many-accelerator architectures, we evaluate AuRORA in two different SoC configurations:



(a) ResNet50 **(b) AlexNet**

Figure 6: Normalized latency sweeping Manager's L2 TLB and PTW cache size. Latency is normalized to the ideal case.

(1) **Crossbar**: All components (AuRORA Client, Manager, memory system) are connected to a crossbar. This configuration provides a uniform memory system.
(2) **NoC**: All components are integrated in a 7x4 2D mesh as illustrated in Figure 5. This configuration provides a realistic and scalable NUMA memory system for a many-core/many-accelerator SoC.

We integrate Gemmini, a TPU-style systolic-array accelerator, within a AuRORA Manager and replicate it across ten separate homogeneous Manager accelerator tiles on the same SoC. Each Gemmini accelerator is equipped with a 16x16 weight-stationary systolic array for matrix multiplications and convolutions, with private scratchpad memories to store weights and in/output activations. All the tiles also share the memory subsystem, including a shared L2 cache and DRAM.

The AuRORA runtime is implemented in C++ and operates seamlessly on top of a full Linux stack. The runtime uses a lightweight software look-up table for the scoreboard, which manages the compute allocation and memory bandwidth utilization of each application on the Clients. The runtime also implements task queues, which track generated tasks.

4.2 Microarchitecture Exploration

This section explores various configurations of AuRORA and demonstrates AuRORA's adaptability for diverse deployment scenarios.

PTW cache/TLB configuration. We sweep the Manager's L2 TLB and private PTW cache to determine the optimal TLB and PTW cache size for multi-tenant DNN execution. Figure 6 shows the effects of the Manager's L2 TLB size and PTW cache sizes on the end-to-end latency of ResNet50 and AlexNet. We generate a single DNN accelerator using Gemmini [20] with the hardware configurations described in Table 4. The latency is normalized to the bare-metal test, where no address translation happens. We notice that ResNet50's performance is saturated with a small L2 TLB and PTW cache, reaching the minimum latency with only a 256-entry L2 TLB and 0.5KB (4 ways, 2 sets) PTW cache. AlexNet, on the other hand, is dominated by fully-connected (FC) layers with frequent TLB misses, where a bigger PTW cache can take advantage of the spatial locality to reduce end-to-end latency. For both cases, a 0.5KB PTW cache and 512-entry L2 TLB are enough to minimize the end-to-end performance overhead. Thus, we use this configuration of the AuRORA Manager for further experiments.

Table 5: AuRORA end-to-end latency overhead across SoC configurations and ResNet sizes.

# accel		ResNet50 xbar	ResNet50 xbar+ NoC	ResNet50 Shared NoC	ResNet18 xbar	ResNet18 xbar+ NoC	ResNet18 Shared NoC
1	Total cycle (Normalized)	1	1.07	1.07	1	1.044	1.044
	AuRORA overhead (%)	0.38	0.36	0.38	0.49	0.48	0.48
2	Total cycle (Normalized)	1	1.112	1.132	1	1.05	1.05
	AuRORA overhead (%)	0.46	0.43	0.45	0.59	0.57	0.61
4	Total cycle (Normalized)	1	1.14	1.165	1	1.076	1.079
	AuRORA overhead (%)	0.63	0.57	0.62	0.83	0.79	0.85

SoC Configurations. To illustrate the effectiveness of AuRORA in different SoC configurations, we run experiments of ResNet50 and ResNet18 running on one, two, and four accelerators, each interconnected through three different SoC interconnect designs. Table 2 shows their performance and AuRORA overhead. In particular, we generate three different interconnect configurations:

(1) crossbar: all memory traffic and AuRORA hardware protocol traffic are connected with crossbars.
(2) crossbar+NoC: memory traffic is routed via a 4x4 2D mesh NoC, while the AuRORA traffic uses a separate crossbar.
(3) Shared NoC: both memory traffic and the AuRORA protocol traffic share the same 4x4 2D mesh NoC.

To capture the worst-case overhead from frequent AuRORA protocol traffic, accelerators are acquired and released before and after each layer (Gemm, Convolution, Residual addition). Our result clearly demonstrates that AuRORA's management overhead for accelerators is quite negligible, accounting for less than 1% of the total cycles across all the scenarios. This underscores AuRORA's flexibility and scalability for multi-accelerator systems.

4.3 Workloads

DNNs. Our evaluation uses seven different state-of-the-art DNN inference models, including SqueezeNet [28], GoogLeNet [55], AlexNet [33], YOLOv2 [52], YOLO-lite [48], Keyword Spotting [58], BERT [18] and ResNet [24]. The DNN models in the evaluation feature varied model sizes, DNN kernel types, applications, computational and memory requirements, and compute-to-memory trade-offs. To capture any distinct behaviors across various sets, we classify the workloads into workload sets based on the size of DNN models. This classification method is based on the approach presented in [31], which we use as one of our baselines. Table 6 presents the DNN benchmarks categorized by size. Workload set-A comprises the lighter models, while Workload set-B groups the heavier models. Workload set-C includes a mixture of a subset of set-A and set-B, encompassing both lighter and heavier models.

Table 6: Benchmark DNNs and workload set categorization based on model size used in the evaluation.

Workload	Model size	Domain	DNN models
Workload set-A	Light	Image Classification	SqueezeNet [28], ResNet18 [24]
		Object Detection	Yolo-LITE [48]
		Language Processing	BERT-small [18]
		Speech Processing	KWS [58]
Workload set-B	Heavy	Image Classification	GoogleNet [55], AlexNet [33], ResNet50 [24]
		Language Processing	BERT-base [18]
		Object Detection	YoloV2 [52]
Workload set-C	Mixed	All	ResNet18 [24], Yolo-LITE [48] SqueezeNet [28] AlexNet [33], ResNet50 [24], BERT-base [18]
Workload set-XR Gaming	Mixed	Hand Tracking	Hand Graph [42]
		Eye Pipeline	RITNet [10], FBNet [60]
		Depth Estimation	MiDaS [50]
		Plane Detection	PlaneRCNN [38]

To construct a multi-tenant workload from each scenario, we randomly select N number of different inference tasks, where N ranges from 200 to 300, for concurrent execution.

QoS targets. We set our baseline QoS based on prior works [8, 40], setting 25ms for AlexNet, ResNet50, 10ms for SqueezeNet and YOLO-Lite, 50ms for BERT-base and 15ms for the rest. To assess how AuRORA performs with varying latency targets, we also adjust the baseline latency target to 1.2× and 0.8× QoS, corresponding to a 20% increase and decrease in the latency target, respectively. Specifically, QoS-H (hard) denotes a 0.8× QoS latency target, which is more difficult to achieve. QoS-L (light) represents a 1.2× QoS latency target, which is a more lenient goal. QoS-M refers to the baseline QoS latency target.

Emerging applications. To demonstrate the utility of AuRORA for emerging applications, we also deploy a usage scenario for AR/VR, as suggested by XRBench [35], and create Workload set-XR using AR/VR Gaming scenarios. We construct load generation settings following the guidelines in XRBench: inference requests are injected at the target frame-per-second (FPS) processing rate with a jitter applied to each frame.

4.4 Metrics

We evaluate the efficacy of multi-tenant execution with AuRORA using the metrics proposed in [19], which are commonly used in multi-tenant evaluation [12, 21, 31]. These metrics encompass the percentage of workloads for which we meet the Service Level Agreement (SLA), the throughput of the co-located applications, and the fairness of AuRORA's resource management strategy. To determine workload latency, we measure the duration from the time it is generated until it is completed and commits, including the time it spends in the task queue and its runtime.

SLA satisfaction rate. We set the SLA target, which is the QoS latency target constraint, for each workload based on the three QoS levels defined in Section 4.3, as mentioned in the 'QoS targets' paragraph. Achieving a higher SLA satisfaction rate would mean more queries meeting the QoS latency target. We use SLA and QoS targets interchangeably in the following discussion.

Fairness. The *fairness* is a metric that measures equal progress under multi-tenant execution compared to the single task's isolated execution. Fairness metric has been used in prior multi-tenant works [12, 21, 31]. This metric assesses AuRORA's dynamic score-based virtual accelerator management, for both compute-resource partitioning and memory-resource partitioning. As shown in Equation 1, C_i represents the cycles of the i-th workload, C^{single} indicates the cycles of the workload running on the SoC with no other concurrent workloads. C^{MT} denotes multi-tenant execution cycles. We define *fairness* in terms of *normalized progress* (NP), which describes the slowdown of multi-tenant execution compared to its isolated execution without interference, suggested in [19] as follows:

$$NP_i = \frac{C_i^{single}}{C_i^{MT}}, \qquad Fairness = min_{i,j}\frac{NP_i}{NP_j} \tag{1}$$

Throughput. To evaluate the effectiveness of AuRORA in increasing overall hardware utilization, we analyze the total *system throughput* (STP). STP is defined as the system throughput of executing n programs, which sums up each program's normalized progress, ranging from 1 to n. Maximizing overall progress when co-locating multiple applications is crucial to maximizing STP.

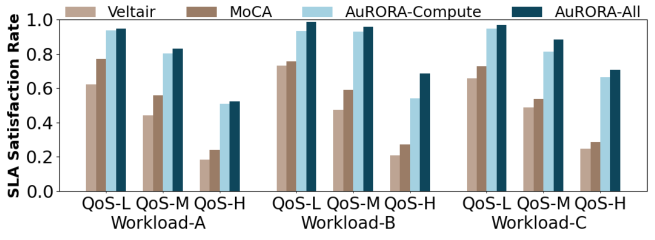
$$STP = \sum_{i=1}^{n} \frac{C_i^{Single}}{C_i^{MT}} \tag{2}$$

Real-time and QoE Score. To evaluate Workload set-XR, we use the metrics suggested by XRBench [35]: *Real-time* (RT) Score and *Quality of experience* (QoE) Score. RT Score uses a modified sigmoid function to gradually increase and decrease the score when the inference latency is shorter or longer, respectively, than the target, and we use the default value in XRBench for the parameter k. QoE score quantifies the penalty for FPS drops due to dropped frames, which is not counted in the RT score. We set the Accuracy and Energy score to 1 as AuRORA does not affect DNN accuracy and our evaluation focuses on homogeneous accelerators. The overall score is computed using the QoE, RT, Accuracy, and Energy scores as XRBench describes.

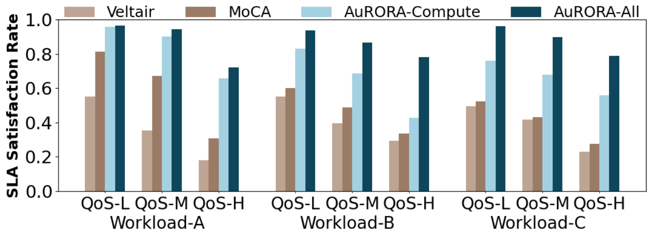
4.5 Baselines

To evaluate the effectiveness of AuRORA's virtual accelerator management and QoS optimization, we compare against two different baselines that use physical accelerator with AuRORA and measure the performance improvement. The prior works that we use as baseline are the following:

(1) Veltair [40]: dynamic compute-resource partition with coarse-grained layer-blocks to avoid rescheduling overhead;
(2) MoCA [31]: adaptive memory-resource re-partition based on system-level contention for spatially co-located DNNs.



(a) Crossbar



(b) NoC

Figure 7: AuRORA's SLA satisfaction rate improvement over evaluated multi-tenancy baselines with different QoS targets (QoS-L/M/H: light/medium/hard latency target) and DNN workload sizes (Workload-A/B/C: light/heavy/mixed models).

These baselines are the most recent works proposing system support for QoS management in multi-tenant DNN workloads and addressing the accelerator migration cost through coarse-grained scheduling. Note that Veltair is a joint adaptive compilation and scheduling work that targets different hardware platforms (CPU cluster). We take Veltair's scheduling component, which is a layer-blocking strategy and scheduler, as a physical integration baseline. Our MoCA implementation uses AuRORA's memory access rate configuration instruction to change the memory access rate, instead of modifying the accelerator's internal DMA.

For physical accelerator binding baselines, the task thread would request its target number of accelerators to meet the QoS requirement by directly pinning the accelerator. If there is a scheduling conflict, which is when there are fewer accelerators available in the system, it would attempt to pin the accelerator from the other thread that is going to finish the current layer block the earliest, and then start the execution after synchronizing and adjusting the accelerator affinity.

For AuRORA evaluation, we use two different configurations by incrementally enabling QoS optimization to show the effectiveness of each resource management feature:

(1) AuRORA-Compute, which performs dynamic compute-resource re-partitioning with virtual accelerator.
(2) AuRORA-All, which adds NUMA-aware compute partitioning for NoC deployment scenarios and dynamic memory-resource re-partitioning for both crossbar and NoC.

5 EVALUATION

In this section, we evaluate the effectiveness of AuRORA for multi-tenant workloads by comparing it to two baseline solutions. Veltair [40] and MoCA [31] are recent proposals for improving DNN multi-tenant execution by co-locating multiple DNNs while binding accelerators physically to user threads. Our evaluation demonstrates that AuRORA improves SLA satisfaction rates, STP, and fairness across a wide range of workload scenarios with different DNN models and QoS requirements with a small hardware area overhead.

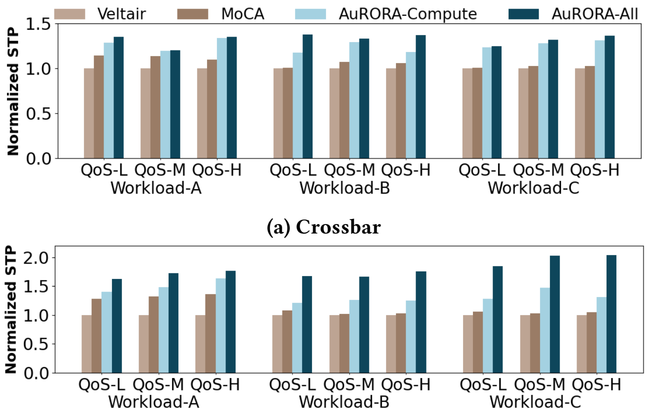
5.1 SLA Satisfaction Rate

We evaluate the effectiveness of AuRORA-enabled virtual accelerator management for multi-tenant execution using the three sets of workloads listed in Table 6, each with three QoS targets (Hard: QoS-H, Medium: QoS-M, Light: QoS-L) on two hardware platforms (crossbar and NoC), resulting in a total of 18 runtime scenarios. We measure the SLA satisfaction rate for each scenario and compare them to the baseline to demonstrate the performance improvement achieved.

5.1.1 Crossbar configuration.

AuRORA's virtual compute resource partitioning improves overall SLA satisfaction rate. Figure 7a demonstrate that AuRORA's virtual compute resource partitioning consistently outperforms the baseline methods for all scenarios. Specifically, AuRORA-Compute achieves a geomean improvement of 1.9× over Veltair (max 2.76× in Workload-A/QoS-H). Compared to MoCA, AuRORA-Compute achieves 1.6× improvement (max 2.33× in Workload-C/QoS-H). QoS-H shows the highest improvement across QoS groups, AuRORA-Compute achieving 2.68× improvement over Veltair and 2.14× over MoCA. The baseline strategies use physical accelerator binding to pin accelerators to user threads, causing significant overhead when reallocating the accelerator. To work around it, the baselines use coarser-grained resource allocation to avoid frequent scheduling conflicts and reallocation of compute resources, as the re-partitioning incurs significant overhead without a virtual accelerator abstraction. However, this restricts support for fast and adaptive system reconfiguration with increased resource contention, making it more challenging to meet QoS targets, especially for QoS-H where resource conflicts become more severe due to stingier QoS requirements. AuRORA, on the other hand, with its fast mechanism to manage and bind virtual target accelerators, supports accelerator reallocation faster in the environment with dynamically changing computation demands.

AuRORA's dynamic memory management further improves target satisfaction rate. Figure 7a shows that AuRORA-All improves SLA satisfaction by 2.02× in geomean (max 3.32× in Workload-B/QoS-H) over Veltair and 1.7× over MoCA (max 2.54× in Workload-B/QoS-H). To evaluate the impact of AuRORA's memory resource management, we further compare AuRORA-Compute and -All. AuRORA-All improves SLA satisfaction rate over AuRORA-Compute by 1.07× in geomean. Among the workload sets, Workload-B results in the most improvement over AuRORA-Compute, by 1.12×. This shows that AuRORA's memory management scheme is particularly effective in more memory-constrained environments, as heavier workloads have larger weights that would stress the memory system. Across the QoS groups, QoS-H shows the most improvement of AuRORA-All over -Compute, by 1.12×. This



(a) Crossbar

(b) NoC

Figure 8: STP improvement of AuRORA over evaluated multi-tenancy baselines (normalized to Veltair baseline) with different QoS targets and DNN workload sizes.

is because AuRORA's memory management feature can further enhance the target satisfying ability when QoS requirements become harder to meet by prioritizing memory requests of workloads with less time margin with its memory partitioning scheme.

5.1.2 *NoC configuration.*

AuRORA's NUMA-aware virtual accelerator management is efficient in NoC deployed scenarios. The impact of system-level interference varies among distributed accelerator nodes connected via NoC, primarily due to the NUMA effect. Furthermore, the extent of performance degradation differs among different DNN models, depending on the sensitivity of the workloads to a NUMA-based memory system. AuRORA's NUMA aware compute partitioning scheme captures this and optimizes through better allocation of accelerators and accelerator swapping. As Figure 7b shows, compared to baselines, AuRORA-All achieves 2.41× geomean improvement over Veltair (max 3.99× in Workload-A/QoS-H) and 1.87× over MoCA (2.85× in Workload-C/QoS-H). AuRORA-All, which enables both NUMA-aware compute resource partitioning and dynamic memory resource management, increase SLA satisfaction rate by 1.25× on geomean compared to AuRORA-Compute. Across the workload sets, AuRORA-All achieved a geomean improvement of 1.05× for Workload-A, 1.38× for Workload-B, and 1.33× for Workload-C, over AuRORA-Compute. NUMA and memory optimization achieve a higher increase in heavy or mixed sets than light ones, as the NUMA effect is more pronounced for workloads that cause more memory traffic. Thus, AuRORA-All would benefit in those scenarios by alleviating the NUMA effect with better compute partitioning and alleviating memory contention by memory partitioning.

5.2 System Throughput Analysis

We evaluate the STP of multi-tenant scenarios, as described in Section 4, to demonstrate that AuRORA improves the STP compared to the baselines.

AuRORA's virtual accelerator allocation increases overall system throughput. Figure 8a shows an improvement in system throughput in the crossbar-based system. AuRORA-Compute exhibits 1.26× geomean improvement over Veltair (max 1.34× in Workload-A/QoS-H). Compared to MoCA, AuRORA-Compute demonstrates 1.18× geomean improvement (max 1.28× Workload-C/QoS-H). Although STP improvement is overall consistent, across different scenarios, the improvement increases as the QoS requirement gets stricter, showing the highest improvement of 1.28× over Veltair in the QoS-H group. This indicates that accelerator virtualization with AuRORA can improve resource utilization across all scenarios with flexible and fast resource reallocation, especially under the increase in resource conflicts.

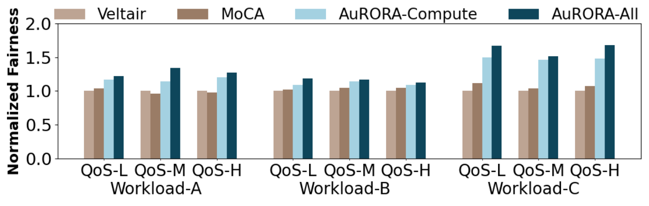
AuRORA's memory resource management improves STP. As Figure 8a shows, AuRORA-All achieves 1.33× improvement over Veltair and 1.25× over MoCA (max 1.38× and 1.37× respectively in Workload-A/QoS-L), which is 1.06× geomean STP improvement over AuRORA-Compute. Across the workload sets, AuRORA-All shows the most improvement over AuRORA-Compute in Workload-B with a 1.12× improvement. This is because memory access rate management would alleviate performance degradation due to memory interference, which becomes more prominent with heavier workloads.

AuRORA's NUMA-aware accelerator allocation improves STP. Figure 8b shows NoC deployment results. When both NUMA and memory resource optimizations are enabled, AuRORA-All improves STP 1.79× over Veltair (2.04× max in Workload-C/QoS-H) and 1.59× over MoCA (1.97× max in Workload-C/QoS-M). Compared to AuRORA-Compute, AuRORA-All achieves 1.32× STP improvement, which is greater than the crossbar scenario, which further shows the effectiveness of AuRORA-All in the system with NUMA effect in improving throughput. The impact is most prominent in Workload-C, where there was a 1.46× improvement over AuRORA-Compute across all QoS levels. The NUMA effect is more pronounced for heavier workloads and the degree of variance in the NUMA effect gets severe with workload heterogeneity. Thus, enabling NUMA optimization can alleviate this effect, leading to an improvement in the overall STP.

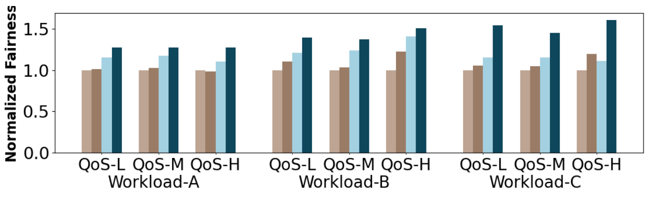
5.3 Fairness Analysis

We evaluate the overall system fairness of multi-tenant execution, as defined in Section 4, to demonstrate the effectiveness of AuRORA in improving this metric. We compare the fairness of AuRORA with the baseline strategies and normalize the results to Veltair's fairness, shown in Figure 9.

AuRORA virtual accelerator support improves fairness. Figure 9a demonstrates AuRORA-Compute increases overall fairness by 1.25× over Veltair (max 1.50× in Workload-C/QoS-L), and by 1.2× over MoCA (max 1.41× in Workload-C/QoS-M). The improvement is consistent across different QoS levels, but across workload sets, Workload-C shows the highest improvement (1.48× over Veltair, 1.38× over MoCA). Baseline physical accelerator allocation suffers latency overhead upon thread synchronization for accelerator reallocation, which impact varies across tasks, resulting in fairness degradation. This gets most pronounced for the mixed set where

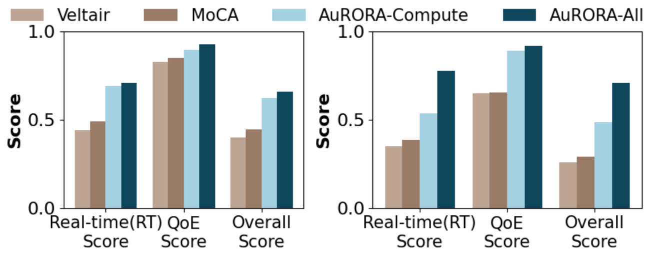


(a) Crossbar



(b) NoC

Figure 9: Fairness improvement of AuRORA over evaluated multi-tenancy baselines (normalized to Veltair baseline) with different QoS targets and DNN workload sizes.



(a) Crossbar (b) NoC

Figure 10: Real-time (RT) score, QoE score and Overall score improvement of AuRORA over evaluated multi-tenancy baselines for Workload set-XR Gaming usage scenario.

synchronizations between heavy and light models happen. AuRORA helps promote a fair system by enabling efficient acquisition and release of accelerators.

Memory resource and NUMA optimization helps further improve fairness. As shown in Figure 9b, AuRORA-All achieves 1.41× overall improvement in fairness over Veltair (max 1.61×, Workload-C/QoS-H) and 1.31× over MoCA (max 1.46×, Workload-C/QoS-L). Specifically, Workload-C shows the most improvement (1.35× comparing AuRORA-All over -Compute). Since the degree of NUMA effect differs greatly for mixed workload sets, a poorly assigned accelerator would unfairly impact some workloads, which NUMA optimization can help alleviate the issue leading to fairness improvement. For the crossbar result in Figure 9a, AuRORA-All improves fairness over Veltair by 1.34× on geomean and over MoCA by 1.3× (max 1.68× and 1.57× respectively in Workload-C/QoS-H). The memory resource management feature helps resolve shared memory system contention whose impact differs due to different compute-to-memory ratios of the workloads.

Table 7: Area breakdown of accelerator design with AuRORA.

Component	Area (μm^2)	% of Area
CPU tile	168K	100%
AuRORA `Client`	2K	1.2%
Rocket CPU	166K	98.8%
Accelerator tile	732K	100%
AuRORA `Manager`	22K	3%
Accelerator	710K	97%
Mesh	76K	10.4%
Accumulator	260K	35.5%
Scratchpad	150K	20.5%
Total (CPU tile+Accelerator tile)	900K	100%
AuRORA `Client + Manager`	24K	2.7%

5.4 Real-time and QoE Analysis

AuRORA improves meeting real-time requirements. As Figure 10 shows, AuRORA-All achieves 1.61× RT score improvement over Veltair and 1.44× over MoCA for crossbar deployment, and 2.24× over Veltair and 2.02× over MoCA for NoC deployment. Virtual accelerator allocation alone (AuRORA-Compute) shows 1.57× improvement over Veltair for the crossbar and 1.54× for the NoC scenario, which indicates the effectiveness of AuRORA's virtual compute resource management.

AuRORA improves quality of experience. AuRORA's improvement of both RT and QoE scores indicates that AuRORA is able to preserve the target FPS as well as maintain the timing requirements for the executed frames. As Figure 10 shows, AuRORA-All achieves QoE improvement of 1.12× over Veltair and 1.1 × MoCA for crossbar, and 1.41× over Veltair and MoCA for NoC deployment. Thus, AuRORA-All's Overall score improves by 1.66× and 1.49× over Veltair and MoCA for the crossbar scenario, and 2.74× and 2.45× for the NoC scenario.

5.5 Physical Design and Area Analysis

We synthesize AuRORA `Manager`-integrated Gemmini accelerator and AuRORA `Client`-integrated Rocket CPU using Cadence Genus with commercial 16nm process technology with the configuration used in the evaluation. As shown in Table 7, AuRORA incurs an overhead of 2.7% of the total area. Specifically, `Client` incurs 1.2% and `Manager` incurs 3% of CPU and accelerator tile area, respectively. `Client` overhead is minimal as it only needs enough bits to track which accelerators are assigned to the current resident thread. `Manager` also causes very low physical area overhead compared to an accelerator, as the critical architectural shadowed state is less than 100 bits of storage. The majority of its overhead is the page table walker and TLB, which is present in any accelerator that requires an IOMMU.

6 CONCLUSION

This work proposes AuRORA, a scalable accelerator integration approach that enables efficient execution of multi-tenant workloads using a virtual accelerator abstraction. Unlike existing accelerator integrations, AuRORA optimizes for dynamic contention-aware scheduling of multi-tenant tasks with minimal performance overhead with a full-stack architecture. We implement AuRORA's microarchitecture, messaging protocol, ISA, and runtime, and demonstrate its ability to improve end-to-end metrics for multi-tenant DNN workloads. Our evaluation of diverse workload sets, latency targets, and hardware deployment shows AuRORA achieves overall SLA 2.41×, STP 1.79×, and fairness 1.41× improvement compared to existing multi-tenant solution for NoC deployed scenario, and overall improvement of SLA 2.02×, STP 1.33×, and fairness 1.34× for crossbar deployed scenario, with 2.7% area overhead.

ACKNOWLEDGMENTS

This research was, in part, funded by the NSF Award CCF-2238346 and CCRI-2016662 and in part by SLICE Lab industrial sponsors and affiliates. This research was also funded in part by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006). 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] [n. d.]. Compute Express Link Interconnect. https://www.computeexpresslink.org/about-cxl

[2] AXI AMBA and ACE Protocol Specification. 2011. ARM. *Cambridge, UK* (2011).

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[4] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John R. Hauser, Adam M. Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, Jack Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moretó, Albert J. Ou, David A. Patterson, Brian C. Richards, Colin Schmidt, Stephen Twigg, Huy D. Vo, and Andrew Waterman. 2016. The Rocket Chip Generator.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[7] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Minh Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016.* ACM, 217–232. https://doi.org/10.1145/2872362.2872414

[8] Simone Bianco, Rémi Cadène, Luigi Celona, and Paolo Napoletano. 2018. Benchmark Analysis of Representative Deep Neural Network Architectures. *CoRR* abs/1810.00736 (2018). arXiv:1810.00736 http://arxiv.org/abs/1810.00736

[9] Sergey Blagodurov, Alexandra Fedorova, Sergey Zhuravlev, and Ali Kamali. 2010. A case for NUMA-aware contention management on multicore systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 557–558.

[10] Aayush K. Chaudhary, Rakshit Kothari, Manoj Acharya, Shusil Dangi, Nitinraj Nair, Reynold Bailey, Christopher Kanan, Gabriel Diaz, and Jeff B. Pelz. 2019. RITnet: Real-time Semantic Segmentation of the Eye for Gaze Tracking. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. 3698–3702. https://doi.org/10.1109/ICCVW.2019.00568

[11] Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, and Chia-Lin Yang. 2010. Memory Latency Reduction via Thread Throttling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, USA, 53–64. https://doi.org/10.1109/MICRO.2010.39

[12] Yujeong Choi and Minsoo Rhu. 2020. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), 220–233. https://doi.org/10.1109/HPCA47549.2020.00027

[13] Lauranne Choquin. 2020. *Arm Custom Instructions: Enabling Innovation and Greater Flexibility on Arm.* Technical Report. Arm Ltd.

[14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. Architecture Support for Accelerator-Rich CMPs. In *Design Automation Conference.* 843–849. https://doi.org/10.1145/2228360.2228512

[15] Henry M Cook, Andrew S Waterman, and Yunsup Lee. 2018. *Sifive tilelink specification.* Technical Report. tech. rep., SiFive Inc., 2018.

[16] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 107–118. https://doi.org/10.1109/HPCA.2013.6522311

[17] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 381–394. https://doi.org/10.1145/2451116.2451157

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). http://arxiv.org/abs/1810.04805

[19] Stijn Eyerman and Lieven Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28, 3 (2008), 42–53. https://doi.org/10.1109/MM.2008.44

[20] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *Design Automation Conference*. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[21] 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. 2020. Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 681–697. https://doi.org/10.1109/MICRO50266.2020.00062

[22] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A. Wright, Joseph E. Gonzalez, and Ion Stoica. 2021. Pylot: A Modular Platform for Exploring Latency-Accuracy Tradeoffs in Autonomous Vehicles. In *IEEE International Conference on Robotics and Automation (ICRA)*. 8806–8813. https://doi.org/10.1109/ICRA48506.2021.9561747

[23] Ricardo E Gonzalez. 2000. Xtensa: A configurable and extensible processor. *IEEE micro* 20, 2 (2000), 60–70. https://doi.org/10.1109/40.848473

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[25] Mark D. Hill and Vijay Janapa Reddi. 2019. Accelerator-level Parallelism. *CoRR* abs/1907.02064 (2019). arXiv:1907.02064 http://arxiv.org/abs/1907.02064

[26] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. 2019. Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-acceleration. In *International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942048

[27] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Jeffrey Zhang, and Sarita V. Adve. 2021. Exploring Extended Reality with ILLIXR: A new Playground for Architecture Research. arXiv:2004.04643 [cs.DC]

[28] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[29] Sheng-Chun Kao and Tushar Krishna. 2022. MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 814–830. https://doi.org/10.1109/HPCA53966.2022.00065

[30] 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. 2018. 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)*. 29–42. https://doi.org/10.1109/ISCA.2018.00014

[31] Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. MoCA: Memory-Centric, Adaptive Execution for Multi-Tenant Deep Neural Networks. In *International Symposium on High-Performance Computer Architecture (HPCA)*. 828–841. https://doi.org/10.1109/HPCA56546.2023.10071035

[32] Seah Kim, Hyoukjun Kwon, Jinook Song, Jihyuck Jo, Yu-Hsin Chen, Liangzhen Lai, and Vikas Chandra. 2023. DREAM: A Dynamic Scheduler for Dynamic Real-time Multi-model ML Workloads. arXiv:2212.03414

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[34] Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, and Vikas Chandra. 2021. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *International Symposium on High-Performance Computer Architecture (HPCA)*. 71–83. https://doi.org/10.1109/HPCA51647.2021.00016

[35] Hyoukjun Kwon, Krishnakumar Nair, Jamin Seo, Jason Yik, Debabrata Mohapatra, Dongyuan Zhan, Jinook Song, Peter Capak, Peizhao Zhang, Peter Vajda, Colby Banbury, Mark Mazumder, Liangzhen Lai, Ashish Sirasao, Tushar Krishna, Harshit Khaitan, Vikas Chandra, and Vijay Janapa Reddi. 2023. XRBench: An Extended Reality (XR) Machine Learning Benchmark Suite for the Metaverse. In *Proceedings of the 5th Machine Learning and Systems Conference (MLSys 2023) (MLSys 2023)*. Miami, FL, USA. https://proceedings.mlsys.org/paper_files/paper/2023/hash/baf570e47e7f4e314a9ffb72c4a5459c-Abstract-mlsys2023.html

[36] Jounghoo Lee, Jinwoo Choi, Jaeyeon Kim, Jinho Lee, and Youngsok Kim. 2021. Dataflow Mirroring: Architectural Support for Highly Efficient Fine-Grained Spatial Multitasking on Systolic-Array NPUs. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 247–252. https://doi.org/10.1109/DAC18074.2021.9586312

[37] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–11. https://doi.org/10.1145/1362622.1362694

[38] Chen Liu, Kihwan Kim, Jinwei Gu, Yasutaka Furukawa, and Jan Kautz. 2019. PlaneRCNN: 3D Plane Detection and Reconstruction From a Single Image. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4445–4454. https://doi.org/10.1109/CVPR.2019.00458

[39] Jun Liu, Jagadish Kotra, Wei Ding, and Mahmut Kandemir. 2015. Network footprint reduction through data access and computation placement in NoC-based manycores. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2744876

[40] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. 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 (ASPLOS '22)*. 388–401. https://doi.org/10.1145/3503222.3507752

[41] Michael Lyons, Mark Hempstead, Gu Wei, and David Brooks. 2012. The accelerator store: A shared memory framework for accelerator-based systems. *ACM Transactions on Architecture. and Code Optimization* 8 (2012), 1–22. https://doi.org/10.1145/2086696.2086727

[42] Meysam Madadi, Sergio Escalera, Xavier Baró, and Jordi Gonzàlez. 2017. End-to-end Global to Local CNN Learning for Hand Pose Recovery in Depth data. *CoRR* abs/1705.09606 (2017). arXiv:1705.09606 http://arxiv.org/abs/1705.09606

[43] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC Development with Open ESP. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20)*. https://doi.org/10.1145/3400302.3415753

[44] NVIDIA. 2018. TensorRT Inference Server User Guide.

[45] NVIDIA. 2022. Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

[46] NVIDIA Corporation. [n. d.]. *Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution.* https://github.com/triton-inference-server/server

[47] Biagio Peccerillo, Elham Cheshmikhani, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. 2023. IXIAM: ISA EXtension for Integrated Accelerator Management. *IEEE Access* 11 (2023), 33768–33791. https://doi.org/10.1109/ACCESS.2023.3264265

[48] Jonathan Pedoeem and Rachel Huang. 2018. YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers. *CoRR* abs/1811.05588 (2018). arXiv:1811.05588 http://arxiv.org/abs/1811.05588

[49] pytorch. [n. d.]. Running TorchServe. https://pytorch.org/serve/server.html

[50] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. 2022. Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-Shot Cross-Dataset Transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 3 (2022), 1623–1637. https://doi.org/10.1109/TPAMI.2020.3019967

[51] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*. 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[52] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *CoRR* abs/1612.08242 (2016). arXiv:1612.08242 http://arxiv.org/abs/1612.08242

[53] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel S. Emer, C. Thomas Gray, Stephen W. Keckler, and Brucek Khailany. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. 14–27. https://doi.org/10.1145/3352460.3358302

[54] Mohandeep Sharma and Dilip Kumar. 2012. Wishbone bus architecture-a survey and comparison. *arXiv preprint arXiv:1205.1860* (2012).

[55] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. *CoRR* abs/1409.4842 (2014). arXiv:1409.4842 http://arxiv.org/abs/1409.4842

[56] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2012. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2259016.2259018

[57] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou Soffa. 2013. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. 41, 1 (2013). https://doi.org/10.1145/2490301.2451126

[58] Raphael Tang and Jimmy Lin. 2017. Deep Residual Learning for Small-Footprint Keyword Spotting. *CoRR* abs/1710.10361 (2017). arXiv:1710.10361 http://arxiv.org/abs/1710.10361

[59] TensorFlow. [n. d.]. *Tensorflow Guide Architecture.* https://www.tensorflow.org/tfx/serving/architecture

[60] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10726–10734. https://doi.org/10.1109/CVPR.2019.01099

[61] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanović. 2022. Constellation: An Open-Source SoC-Capable NoC Generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. 1–7. https://doi.org/10.1109/NoCArc57472.2022.9911299

[62] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix section describes how to access, exercise, and evaluate the artifacts for each AuRORA component, as performed in Section 5. As described in Section 4, FireSim FPGA-accelerated simulations will be used to evaluate AuRORA in a full-stack environment.

A.2 Artifact meta-information checklist

- **Runtime environment: AWS FPGA Developer AMI 1.12.2**
- **Hardware: AWS EC2 instances (c5.4xlarge, f1.2xlarge)**
- **How much disk space is required?: 300 GB (on EC2 instance).**
- **Experiments: FireSim simulations of the SoCs in Section 4, which integrate RISC-V-based DNN accelerators and CPUs with AuRORA infrastructure, running multi-tenant inference queries.**
- **Program: Chisel (RTL), C (Runtime, Scheduler), Python (Script)**
- **Metric: Target satisfaction rate (rate for each query that meets the target deadline), STP (System Throughput), Fairness, and Real-time/QoE score metrics defined in Section 4.4.**
- **Output: Parsed result from UART output of SoC, performance comparison bar plot between baselines and AuRORA (AuRORA-All, AuRORA-Compute) for each QoS level and Workload sets. (Evaluation Figure 7-10)**
- **How much time is needed to prepare the workflow?: 2 hours (setting up FireSim manager instance and running scripted installation).**
- **How much time is needed to complete experiments?: 12 hours (workload image generation, running experiment, result parsing)**
- **Publicly available: Yes.**
- **Code licenses: Several, see download.**

A.3 Description

A.3.1 How to access. The artifacts consist of:

(1) **firesim-aurora-ae**: FireSim FPGA-Accelerated RTL Simulation Environment. (Zenodo: https://doi.org/10.5281/zenodo.8354298)
(2) **chipyard-aurora-ae**: Chipyard RISC-V SoC Generation Framework. AuRORA microarchitecture is included. (Zenodo: https://doi.org/10.5281/zenodo.8354250)
(3) **gemmini-aurora-ae**: Gemmini DNN accelerator hardware (RTL) used for the AuRORA evaluation (Zenodo: https://doi.org/10.5281/zenodo.8354236).
(4) **aurora-rocc-tests-ae**: AuRORA runtime and scheduler software, scripts and tests. (Zenodo: https://doi.org/10.5281/zenodo.8354218).

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

A.3.2 Dependencies - Hardware. One AWS EC2 c5.4xlarge instance (also referred to as "manager" instance), and two f1.2xlarge instances are required. We split the workload to run on two parallel f1 instances to reduce evaluation time, one for crossbar-based SoC and the other for NoC-based SoC. The f1.2xlarge instances will be launched automatically by the manager instance. We have provided prebuilt FPGA images to avoid the long latency (~9 hours) of the FPGA bitstream synthesis process. However, if users want to build custom FPGA images, two additional z1d.2xlarge instances are required.

A.3.3 Dependencies - Software. Use ssh or mosh on your local machine to remotely access EC2 instances. All other requirements are automatically installed by scripts in the following sections. Please use a tmux session running on the manager instance to make sure long-running jobs are not killed as our setup scripts and tests take a long time to run.

A.4 Installation

First, follow the instructions on the FireSim website[3] to create an EC2 manager instance. You must complete up to and including Section "Launching a Manager Instance: Key Setup, Part 2" (recommend selecting c5.4xlarge as evaluation use this instance).

Once you have completed up to and including Section "Configuring Required Infrastructure in Your AWS Account: 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, begin by downloading the top-level FireSim repository from Zenodo:

```
  # Enter as a single line:
  $ wget -O firesim-aurora-ae.zip https://zenodo.org/
record/8354298/files/firesim-aurora-ae.zip
  $ unzip firesim-aurora-ae.zip
```

Make sure to copy the key (firesim.pem) to `/home/centos` on the launched instance. Next, run the following, which will initialize all dependencies and run FireSim and Chipyard setup steps (RISC-V toolchain installation, matching host toolchain installation, Linux base image build, etc.):

```
  $ cd firesim-aurora-ae
  $ ./first-clone-setup.sh
```

Running the above setup script would also trigger a few prompts (about 3), so check in occasionally.

After the script finishes running, run the following:

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

Once these steps have been completed, you are fully ready to evaluate AuRORA.

A.5 Experiment workflow

Now that our environment is set up, we will run AuRORA artifact. First, we will begin with building the workload image for AuRORA.

[3]https://docs.fires.im/en/1.17.0/Getting-Started-Guides/AWS-EC2-F1-Getting-Started/Initial-Setup/index.html

A.5.1 Building Linux image containing workload. Step 1 is already contained in the `first-clone-setup.sh` setup script. Users only need to follow Step 2.

(1) (Skip - in setup script) On the manager instance, build the FireSim-compatible RISC-V Linux image using a buildroot-based Linux distribution. Follow the instructions in "Section 2.1.1 Building target software" of FireSim documentation.
(2) Run the following command to build the baselines and AuRORA runtime scheduler written in C on a full Linux environment. Running the commands will generate a workload `json` file in `deploy/workloads`.

```
$ ./build-ae-workload.sh
```

The above script will generate workload images that contain all the test scenarios used in our evaluation. It contains two workload images, one to run on the crossbar-based SoC and the other for the NoC-based SoC.

A.5.2 Running FireSim simulations. Go to `deploy/`, and run the workloads by following the steps below.

(1) For configurations in `config_runtime.yaml`, see "Section 2.1.2. Setting up the manager configuration" of FireSim documentation. Make sure to modify the following parameters ((b)-(d) should be modified already, but (a) needs to be changed manually):
  (a) "`f1.2xlarge: 2`" under `run_farm_hosts_to_use` to boot 2 f1 instances.
  (b) "`topology: eval_hw_config`" under `target_config`.
  (c) "`no_net_num_nodes: 2`" to launch two f1 instances running in parallel.
  (d) `workload_name: gemmini-tests-workload.json`.
(2) Run FireSim simulations by launching f1.2xlarge instances. Follow the instructions in "Section 2.1.3 Launching a Simulation!" of the FireSim documentation.
(3) Wait for about 10 hours for both tests to finish.
(4) The result will be copied to a directory in `deploy/results-workload`. The generated result directory will consist of two sub-directories, each for crossbar (-xbar) and NoC (-noc) defined on Section 4.1, as each f1 instance runs each evaluation SoC configuration. Check that `uartlog` has been copied to the subdirectory.

The test scripts will run the baselines and AuRORA configurations on 200 end-to-end turns of inference queries that are dispatched randomly. Note that this script will not rebuild FPGA images for the system by default, since each build takes around 8-10 hours. We instead provide pre-built images by default on `config_hwdb.yaml`, which is used in the paper's evaluation.

Please make sure the running f1 instances are terminated by running `firesim terminaterunfarm` after finishing the above experiment steps, and confirm in your AWS EC2 management console that no instances remain beside the manager.

A.6 Evaluation and expected results

After finishing running the workloads, follow the steps below to parse and view the results. Following the procedure will generate the evaluation figures (Figure 7-10), both (a) and (b). Note that non-determinism in the Linux Kernel and workload packaging processes may result in variations in the performance evaluations.

(1) Running the following commands on `results-workload` directory will generate figures for each scenario. The resulting directory on the script (`$result_dir`) should be the directory that includes two sub-directories.

```
$ ./build_sla.sh ($result_dir)
$ ./build_fair_stp.sh ($result_dir)
$ ./build_xr.sh ($result_dir)
```

(2) Run the first command to parse results in Figure 7, which is the SLA satisfaction rate of each workload set, QoS level, and SoC deployment. It will produce 6 figures named "`[SoC type]_sla_[workload set].png`".[4]
(3) The second command is to parse the STP and Fairness results as in Figure 8 and Figure 9. It will produce 6 figures named "`[SoC type]_stp_[workload set].png`" for Figure 8 and other 6 figures named "`[SoC type]_fairness_[workload set].png`" for Figure 9.
(4) The last command is to parse results of Workload set-XR in Figure 10, which evaluates RT/QoE/Overall score for Workload set-XR. It will produce 2 figures named "`[SoC type]_xr.png`".

A.7 Experiment customization

A.7.1 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. System configurations such as the shared L2 size, number of `Client` and `Manager` can be changed by modifying cache parameters at `NoCConfigs.scala`. For building a new FPGA bitstream, please follow the steps in Section "Building Your Own Hardware Designs (FireSim Amazon FPGA Images)"[5].

A.7.2 Customizing experiment parameters. Go to `gemmini/software/gemmini-rocc-tests/include`, and change test parameters in `workload_params.h`, such as `total_workloads` for total number of workloads or `SEED` to give change in randomly generated workloads. Repeat appendix A.5 and appendix A.6 to get the result.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://ctuning.org/ae/submission-20201122.html
- https://ctuning.org/ae/reviewing-20201122.html

[4]crossbar, noc for `[SoC type]` / A, B, C for `[workload set]`
[5]https://docs.fires.im/en/1.17.0/Getting-Started-Guides/AWS-EC2-F1-Getting-Started/Building-a-FireSim-AFI.html