MoCA：面向多租户深度神经网络的以内存为中心的自适应执行机制

Seah Kim、Hasan Genc、Vadim Vadimovich Nikiforov、
Krste Asanovi´c、Borivoje Nikoli´c、Yakun Sophia Shao 加州大学伯克利分校
{seah,hngenc,vnikiforov,krste,bora,ysshao}@berkeley.edu

arXiv:2305.05843v1 [cs.DC] 10 May 2023

摘要—受深度神经网络（DNNs）在不同应用领域广泛采用的推动，多租户执行应运而生：在同一硬件上同时部署多个深度神经网络，以满足不同应用程序的延迟要求，同时提高整体系统利用率。然而，多租户执行可能导致不希望出现的系统级资源争用，进而降低对延迟敏感型应用程序的服务质量（QoS）。

为应对这一挑战，我们提出 MOCA[1]，这是一种面向 DNN 加速器的自适应多租户系统。与专注于计算资源划分的现有解决方案不同，MOCA 会动态管理协同部署的应用程序所共享的内存资源，以满足其服务质量目标。具体而言，MOCA 利用深度神经网络算子和加速器中的规律，根据延迟目标和用户定义的优先级动态调节内存访问速率，使协同部署的应用程序能够获得所需的资源，同时避免其共同运行的应用程序严重缺乏资源。与以往工作相比，我们证明 MOCA 将服务级别协议（SLA）的满足率最高提升了 3.9×（平均提升 1.8×），将系统吞吐量提升了 2.3×（平均提升 1.7×），并将公平性提升了 1.3×（平均提升 1.2×）。

I. 引言

深度学习的最新进展推动了基于深度神经网络的算法在目标检测 [20], 自然语言处理 [14], 增强现实/虚拟现实 [22], [30], [52], 和机器人学 [5], [47]等任务中的广泛应用。因此，深度神经网络已成为各种应用程序的核心构建模块，这些应用程序运行在从边缘平台到数据中心的各类设备上，其中许多应用程序需要同时执行，以满足其延迟要求 [36], [44]。

为了支持这些应用程序的并发执行，需要采用多租户执行，即在同一硬件上共置多个应用程序，从而提高系统的整体效率。在深度神经网络加速器中实现有效多租户执行的关键挑战，是共享资源争用导致的性能波动，尤其是对于延迟要求严格的面向用户的应用程序。近期提出了新技术，通过时间交错执行多个深度神经网络 [2], [9], [40], 或对深度神经网络加速器的计算资源进行空间划分 [18], [30], [33], 以支持深度神经网络加速器中的多租户执行。

尽管以往提出的微体系结构或调度技术在划分加速器的计算资源方面效果显著，但共享存储器子系统（例如共享系统缓存和动态随机存取存储器）的管理却鲜少受到关注。在这些子系统中，多个加速器和通用处理器核心可能会争用带宽和容量。缺乏用于自适应争用管理的系统级解决方案，会导致不可预期的端到端性能下降。

为应对这一挑战，我们提出 MOCA，这是一种以内存为中心的自适应加速器架构，可支持深度神经网络工作负载的高效多租户执行。与以往专注于高效共享加速器计算资源的工作不同，MOCA 旨在通过动态划分计算和内存资源，自适应地管理共同部署的应用程序之间的系统级共享资源争用，同时避免产生高开销。具体而言，MOCA 利用了深度神经网络算子与硬件的规律性；执行延迟与在途内存请求的数量高度相关。

具体来说，MOCA 包含：1）轻量级硬件内存访问监控与调控引擎；2）智能运行时系统，可动态管理每个共同部署应用程序的内存使用量；3）优先级感知调度器，根据用户分配的优先级、延迟目标和内存资源使用量，选择并发执行的工作负载。评估结果表明，与仅划分计算资源的先前工作相比，MOCA 可将整体服务质量提升至多 3.9×（平均提升 1.8×），同时将系统吞吐量提升 2.3×，公平性提升 1.3×。总之，本文的贡献如下：

1）我们通过硬件、运行时系统和调度器的协同设计，开发了 MOCA 这一自适应争用管理机制，可自适应地调节共置深度神经网络应用的争用程度。2）我们使用寄存器传输级实现了 MOCA 硬件，可动态监控和调节 DNN 加速器的内存访问速率。3）我们设计了 MOCA 运行时系统，可根据共置工作负载的目标延迟和优先级，自适应地配置 MOCA 硬件。4）我们构建了 MOCA 调度器，这是一种能够感知优先级和内存争用的工作负载调度器，可选择不同的深度神经网络层并发运行。

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

5）我们通过基于现场可编程门阵列的全系统仿真展示了 MOCA 的有效性，并采用先进的 12 纳米工艺技术对其进行综合、布局和布线。结果表明，与先前的工作相比，MOCA 在一系列部署场景中显著提升了多租户执行的性能。

II. 背景与动机

本节探讨当今深度神经网络工作负载对多租户执行的需求、多租户执行所面临的挑战，以及该领域的相关先前工作。

A. 现代深度神经网络应用

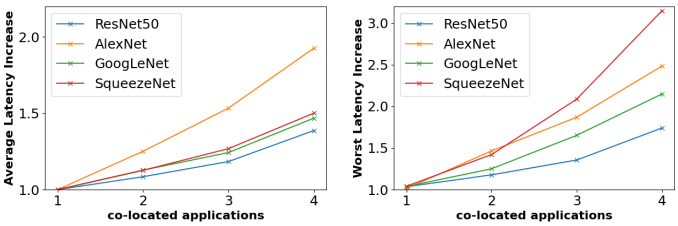
随着深度学习日益普及，许多现代应用程序会采用多种深度神经网络算法来支持不同的功能。例如，机器人学 [3], [5], [47]和增强现实/虚拟现实 [22], [30], [52] 应用程序需要检测附近的智能体、追踪其运动并预测其行进路径，其中许多任务都基于深度神经网络，且需要并发执行。此外，这些工作负载在延迟和带宽需求方面差异很大，既包括具有严格服务质量期限的实时任务（例如增强现实/虚拟现实设备上的眼动追踪 [52], ），也包括仅在片上系统空闲时运行的离线工作负载（例如识别相册中的人物和物体 [44]）。在这种情况下，多租户执行（即同时执行多个工作负载）是避免硬件资源过度配置、同时高效支持并发执行的自然解决方案。

B. 多租户执行面临的挑战

虽然多租户执行通常能提高硬件利用率，但也常因资源争用而导致性能下降。即使是在领域专用加速器的场景中，尽管每个加速器通常都是独立设计的，但集成到片上系统后，它们通常会共享多种系统资源，包括末级缓存（LLC）、系统总线、动态随机存取存储器（DRAM）以及其他输入/输出设备。因此，在加速器上共同运行的应用程序也会争用这些共享资源，从而导致显著的性能下降。

为了了解多租户深度神经网络执行对性能的影响，我们将四种不同的深度神经网络 [20], [23], [29], [48] 共同部署在一款先进的片上系统上，该系统配备了深度神经网络加速器、通用处理器核心、共享系统内存和动态随机存取存储器。这些配置与 NVIDIA 的 Xavier [46]等商用片上系统相似。我们在不同时间随机调度这些网络，以捕捉它们之间不同的干扰模式。我们使用基于现场可编程门阵列的寄存器传输级评估来测量端到端执行延迟。图 1 展示了多租户执行的平均延迟增幅和最坏情况延迟增幅。延迟以工作负载在无争用、独立运行时的延迟为基准进行归一化。

我们观察到，所有工作负载与其他深度神经网络共置时，延迟至少增加了 40%。具体而言，当四个应用程序共置时，AlexNet 的平均延迟几乎增加了 2× ，如



(a) 平均延迟增幅。 (b) 最坏情况延迟增幅。

图1：深度神经网络应用共置时的平均和最坏情况延迟增幅。当 x=1 时，工作负载独立运行，不会出现性能下降。当 x=3 时，工作负载与另外两个随机选取的工作负载共置运行。我们测量了 300 次仿真中不同启动时间下的端到端执行延迟，并据此计算所有运行的平均延迟和最坏延迟。

图1a所示。这是因为 AlexNet 对内存容量相当敏感，其延迟主要由内存密集型全连接层决定。我们还观察到，不同运行之间的性能存在显著差异。例如，SqueezeNet 的最坏情况延迟超过其单独运行时的 3× ，如图1b所示。这是因为 SqueezeNet 的运行时相对较短，因此其延迟很容易受到共置工作负载行为的影响。最坏情况出现在 SqueezeNet 的整个执行过程都与内存密集型层共置时。因此，虽然将多个深度神经网络工作负载共置在同一硬件上并不难，但最重要的挑战是在不同的共运行场景下满足性能要求。

C 语言. 多租户架构支持

表I总结了为支持通用处理器以及近年来领域专用加速器中的多租户执行而开发的软硬件技术。传统通用处理器核心中的多租户执行是软硬件领域经过深入研究的课题，其历史可追溯至片上多处理器问世之时。总体而言，已有两类研究方向。

第一类侧重于静态 机制，即通过静态配置软件或硬件来调整资源竞争型应用的内存访问速率。例如，QoSCompile [49] 会对资源竞争代码区域进行静态限流，Locality-aware complication [25] 会重新编译应用程序以减少多核资源竞争，而 Prophet [6]则利用精准的 QoS预测来识别安全的共置机会。虽然这些静态方法开销较低且无需运行时信息，但通常会对低优先级应用过于保守，因为无论并发运行的应用程序是否确实对争用敏感，它们的内存请求都会被延迟。

第二类针对动态 机制：收集运行时信息，以决定何时触发资源竞争管理机制，以及这些机制需要采取多激进的措施；这些机制可以通过软件或硬件实现。目前，大多数已提出的动态方法都针对

	通用硬件		领域专用硬件		
	软件	硬件	时间分区	空间分区	
				计算资源	内存资源
静态	Prophet [6], QoSCompile [49], 局部性感知编译 [25]	SplitL2 [32], PAR-BS [39], 公平排队 [43], STFM [38], CoQoS [31]	LayerWeaver [40]	HDA [30]	缓存分区 [10] MAGMA [26]
动态	ReQoS [50], CAER [35], Kelp [53], Baymax [7], Quasar [13], Paragon [12], 线程节流 [8]	FST [15], CMP 中的服务质量 [19], 基于效用分区 [42], Heracles [34] FCSP [28]	Prema [9] AI-MT [2]	Planaria [18] Veltair [33]	MoCA

表I：通用硬件和领域专用加速器中的多租户支持分类。

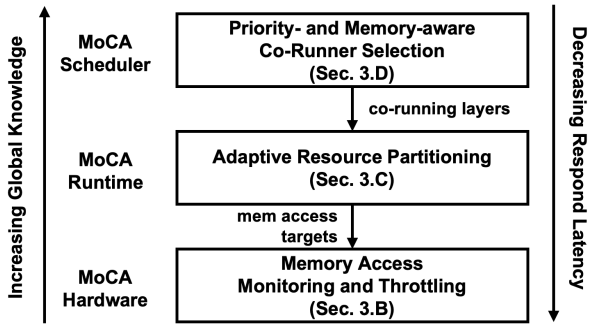


图 3：MOCA 系统概览。

通用处理器。虽然这些动态方法通常涉及更多用于跟踪运行时信息的软件／硬件模块，但它们可以根据系统中实际发生的内存流量，自适应地调整应用程序的争用程度，从而提升服务质量和吞吐量。

对于领域专用加速器，表I进一步根据共享资源是按时间划分（即时间分区），还是按空间划分（即空间分区）进行分类。在空间分区中，我们还区分了空间划分的对象是计算资源（例如空间阵列中的核心或处理单元），还是内存资源（例如共享缓存和DRAM带宽）。LayerWeaver [40]、Prema [9], 和 AI-MT [2] 等先前工作提出以静态或动态方式对深度神经网络执行进行时间复用，但它们的硬件利用率较低，尤其是在层无法利用所有可用资源时。为提高硬件利用率，HDA [30] 和 MAGMA [26] 等静态分区技术提出对计算资源或内存资源进行空间分区，但由于其静态特性，无法适应不同的动态场景。

近年来，动态空间分区机制已经被

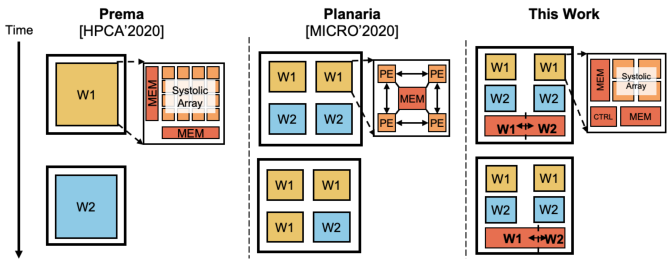


图 2：近期提出的多租户机制比较。Prema [9] 支持不同深度神经网络工作负载（W）之间的时间复用。Planaria [18] 专注于以 Pod 为粒度动态划分计算资源，其中每个 Pod 由固定数量的处理单元（PE）和暂存存储器组成。与以往的研究不同，MOCA 会自适应地划分计算资源和共享内存资源，以满足不同工作负载的资源需求。

为进一步提高共置效率而提出。图 2 突出了近期该领域研究工作的主要差异。具体而言，Planaria [18] 和 Veltair [33] 提出为并行运行工作负载动态分配计算资源。尽管它们能够自适应地分配计算资源，例如为 Planaria 分配处理单元，或为 Veltair 分配 CPU 核心，但它们无法动态管理内存资源，导致资源利用率低，并且在计算资源重新分区 [33]期间产生较高的线程迁移开销。MOCA 通过强调以内存为中心的资源管理的重要性，在执行期间动态划分计算资源和内存资源，从而克服了这些挑战。据我们所知，MOCA是首个通过自适应划分内存资源来支持深度神经网络加速器多租户执行的工作。

III. MOCA 系统

我们提出了 MOCA，这是一种以内存为中心的自适应深度神经网络加速器架构，可动态为共置的深度神经网络工作负载划分计算和内存资源，从而提升深度神经网络加速器的多租户性能。具体而言，MOCA 通过硬件、运行时和调度器的协同设计，动态管理并发运行的工作负载的内存访问速率，以满足应用程序的服务质量要求。本节首先概述 MOCA 系统，随后详细介绍 MOCA 的硬件、运行时系统和调度器组件。

A. MoCA概述

MOCA 是一个全栈系统，由以下部分组成：1）用于控制共置深度神经网络内存访问的轻量级硬件监控与节流引擎；2）能够动态调节各工作负载处理速率的智能运行时；3）用于选择并行运行工作负载的优先级与内存感知作业调度器，如图 3 所示。从系统栈底部开始，MOCA 硬件会监控每个加速器瓦片的内存访问次数；如果硬件在一定时间内超出分配的内存访问目标，就会对执行进行限流。内存访问目标由 MOCA 运行时系统设定，该系统会检测并发运行的作业之间的干扰，并在需要时自适应地重新划分可用资源。

沿系统栈向上，并发运行的作业由 MOCA 调度器选择。该调度器以用户定义的优先级和网络层的内存需求为输入，选择满足工作负载服务质量要求的并发运行层。

MOCA 的硬件、运行时和调度器组件协同作用，减少了系统级争用，并为多租户执行提供了更优的服务质量。接下来，我们将更详细地讨论 MOCA 系统的各个组件。

B. MoCA硬件

MOCA 硬件的职责是动态监控每个加速器瓦片的内存访问速率，并在达到目标访问计数时对其执行进行限流。尽管动态内存限流此前已在通用系统中实现，但据我们所知，MOCA 是首个展示如何在多租户加速器中实现自适应内存调控的研究工作。与早期专注于计算资源划分的多租户加速器不同，MOCA 硬件基于当前先进 DNN 加速器所具有的访问与执行解耦特性构建，无需改变加速器的计算引擎，即可独立控制加速器的内存访问。

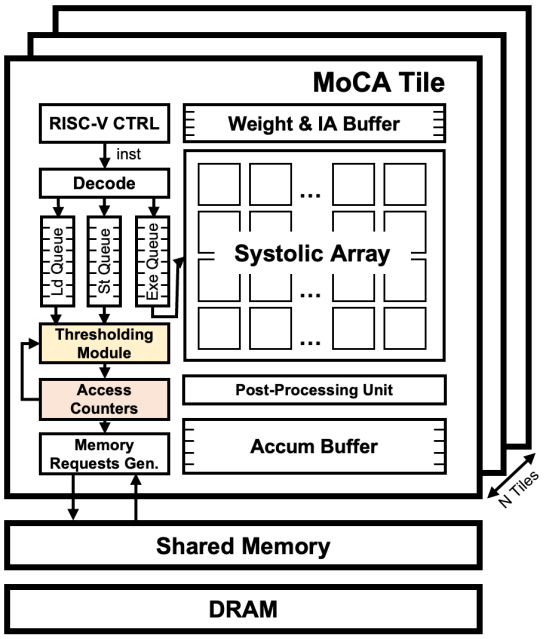


图 4：支持动态内存访问监控和限流的 MOCA 深度神经网络加速器。MOCA 硬件会监控各个加速器瓦片发出的内存访问，并在其超过限制时对其内存访问进行限流。

图 4 展示了所提出的 MOCA 深度神经网络加速器的硬件微架构。除了标准的脉动阵列和缓冲区（用于存储深度神经网络中的权重 (W)、输入激活值 (IA) 和输出激活值 (OA) ）外，MOCA 还在加载 / 存储队列与内存请求生成引擎之间增加了两个新的硬件模块，用于监控并限制其自身生成的、对共享内存资源的内存访问。具体而言，MOCA 实现了一个访问计数器，用于在监控时间窗口内跟踪其内存访问计数；还实现了一个阈值模块，当访问计数器的值在该时间窗口内超过由 *MoCA* 运行时系统配置的阈值负载时，阻止加速器继续生成内存请求。

我们的硬件基于 Gemmini 基础设施 [17]，实现。Gemmini 是一种开源的、基于 RISC-V 的深度神经网络加速器基础设施，支持类似 TPU 的访问与执行解耦。脉动阵列和缓冲区直接由 Gemmini 生成，我们则使用 Chisel RTL [1] 实现 MOCA 专用的访问监控和节流引擎。具体而言，访问计数器 会监控加速器的内存请求，以检查当前执行是否过度使用内存，即是否超过 MOCA 运行时系统配置的目标内存请求数量。

阈值模块根据加速器发出加载请求的速率，确定应插入多少个周期的“气泡”来阻止后续内存请求。气泡可阻止加速器发出更多加载请求，使 MOCA 能够减少内存争用，同时确保每个加速器获得 MOCA 运行时分配的内存访问次数。每当访问计数器发出警报，提示累积计数超过阈值时，MOCA 硬件就会开始停顿，直到 MOCA 运行时更新其状态。与以往需要进行重大重新设计才能支持灵活资源分区的 DNN 加速器不同，[18],MOCA 硬件引擎可以实现为轻量级有限状态机和计数器，而不会产生显著开销。

C. MoCA运行时

MOCA 运行时负责动态检测系统级干扰，并在必要时限制加速器的内存访问速率，以解决争用。现有的多租户解决方案侧重于计算资源分区。虽然易于管理，但如先前的工作 [18], [33] 所观察到的，由于线程迁移开销较高，这些解决方案往往会产生不可忽略的重新分区延迟。相比之下，MOCA 的全栈设计向 MOCA 运行时公开硬件状态，使其能够根据应用程序的需求和重新分区的成本，动态划分内存和计算资源。事实上，为避免较高的开销，MOCA 运行时触发计算资源划分的频率要低得多。

具体而言，MOCA 运行时由两个组件组成。第一部分是基于可用硬件资源对并发运行层进行性能预测，第二部分是争用检测模块和硬件重分区模块，后者会在发生争用时激活。为了准确估算并发运行层的性能和内存需求，MOCA 运行时会捕获整个内存系统中的数据移动开销。如果并发运行层的内存带宽需求大于系统中的可用带宽，MOCA 的

算法1 MOCA 运行时中的延迟估计 ▷ % 估计 C OMPUTE 层的延迟 % 1: 如果 $Layer_i$ 是 COMPUTE, 则 2: $Total_MAC_i \leftarrow calc_MAC_count(Layer_i)$ ▷ % 计算仅计算时间 %▷ % PE 利用率为 100% 时的理想计算时间 % 3: $Compute_ideal_i \leftarrow Total_MAC_i / num_PEs$ 4: ▷ % 计算仅内存时间 %▷ % 到共享 L2缓存 的总流量 % 5: $Total_MEM_i \leftarrow Total_load_i + Total_store_i$ ▷ % 到动态随机存取存储器的流量子集 % 6: $From_DRAM_i \leftarrow (Weight + Output + Bias)_size_i$ ▷ % 如果输入可以缓存, 则无需从动态随机存取存储器重新加载。否则，重新加载。% 7: 如果 $Image_size_i > Cache_size$，则 ▷ % 如果输入激活已被逐出 % 8: $From_DRAM_i += Image_size_i$ 9: 结束 if 10: 如果 $Per_tile_size_i > Cache_size$，则 ▷ % 如果瓦片大小超过缓存大小，则会被逐出 % 11: $From_DRAM_i += Tiling_factor \cdot Tile_size$ 12: 结束 if ▷ % 同时考虑 L2缓存 和动态随机存取存储器事务时间 % 13: $Memory_ideal_i \leftarrow \frac{From_DRAM_i}{DRAM\ BW} + \frac{Total_MEM_i}{L2\ BW}$ 14: ▷ % 根据计算内存比估计总体延迟，同时考虑计算时间和内存时间 % 15: $Prediction_i \leftarrow Max(Compute_ideal_i, Memory_ideal_i) + Min(Compute_ideal_i, Memory_ideal_i) \cdot overlap_f$ 17: 结束 if 18: ▷ % 估计 MEM 层的延迟 % 19: 如果 $Layer_i$ 是 MEM，则 20: $Total_Mem_i \leftarrow InputAB_size_i + Output_size_i$ 21: $From_DRAM_i \leftarrow InputB_size_i + Output_size_i$ 22: $Prediction_i \leftarrow \frac{From_DRAM_i}{DRAM\ BW} + \frac{Total_MEM_i}{L2\ BW}$ 23: 结束 if

算法2 MOCA争用检测与硬件更新 ▷ % 使用算法1获取剩余层的延迟和内存使用量 % 1: $remain_prediction \leftarrow Alg1(Layers)$ 2: for $Layer_i$ in Layers do 3: $(From_DRAM_i, Total_MEM_i, Prediction_i) \leftarrow Alg1(Layer_i)$ 4: $BW_rate_i \leftarrow From_DRAM_i / Prediction_i$ 5: $slack \leftarrow time_left_to_target$ ▷ % 动态优先级分数更新 % 6: $priori_score \leftarrow user_priority_i + remain_prediction\ slack$ ▷ % 神经网络剩余层的延迟预测 % 7: $remain_prediction\ -= Prediction_i$ 8: ▷ % 在记分板中查找并发运行的应用程序的内存使用量 % 9: for App^j in other_Running_Apps do ▷ % 添加每个工作负载当前的内存带宽使用量 % 10: $other_BW_rate\ += MEM_BW(App^j)$ ▷ % 使用动态优先级分数进行加权求和 % 11: $weight_sum\ += score(App^j) \cdot MEM_BW(App^j)$ 12: end for 13: ▷ % 当前系统的总内存需求是否超过最大带宽 % 14: $overflow \leftarrow BW_rate_i + other_BW_rate - DRAM_BW_MAX$ ▷ % 检测到争用（实时检测） % 15: if $overflow > 0$ then 16: $curr_weight_sum \leftarrow priori_score \cdot BW_rate_i$ ▷ % 使用动态优先级分数分配内存 % 17: $BW_rate_i\ -= overflow * weight_sum\ curr_weight_sum + weight_sum$ ▷ % 根据分配结果更新预测 % 18: $Prediction_i \leftarrow BW_rate_i \cdot From_DRAM_i$ 19: ▷ % 设置硬件配置（动态内存分区） % 20: $threshold_load_i \leftarrow Total_MEM_i / Num_tile_i$ 21: $window_i \leftarrow Prediction_i / Num_tile_i$ 22: else ▷ % 未检测到争用，不进行限流 % 23: $threshold_load_i \leftarrow 0$, $window_i \leftarrow 0$ 24: end if 25: UpdateScoreboard(this_App, BW_rate_i) 26: ConfigureHW($window_i$, $threshold_load_i$) 27: runLayer($Layer_i$) 28: end for

运行时声明检测到争用。在这种情况下，MOCA 的运行时会根据用户定义的优先级以及并发运行的应用程序的目标延迟，重新配置 MOCA 硬件，以限制内存访问速率，或在少数情况下重新划分计算引擎。

算法1展示了 MOCA 运行时中的延迟估计算法。不同于以往多租户解决方案中面向计算的延迟估计，[9], [18], [33], MOCA 会考虑数据在整个内存系统中的移动，包括共享内存（即 L2缓存）和动态随机存取存储器，从而能够根据不同的内存资源快速、准确地预测性能。具体而言，MOCA 将深度神经网络层分为两类：计算层具有较高的算术强度（例如卷积层或全连接层）；内存层的数据重用率较低，由受内存带宽限制的运算符构成（例如残差加法以及无法与卷积层融合的最大池化）。overlap_f 是一个参数，表示系统重叠计算与内存操作的能力，该能力可通过加速器设计进行调整（ < 1overlap_f $0 <$ ）。我们提供了一种调优工具，可利用在启动推理查询前运行少量深度神经网络层所收集的数据，为片上系统确定 f 的最优值。对于深度神经网络中每一层，算法1计算乘加运算（乘加运算）的总数，以便进行计算。

基于层的各个维度，以及内存层次结构不同层级之间的数据移动成本，例如动态随机存取存储器和 L2缓存。我们的验证表明，在不同网络和层上的预测误差均在实测运行时的 10% 以内，与 Timeloop [41] 等其他深度神经网络硬件分析模型的结果一致。

算法2展示了 MOCA 运行时如何为 MOCA硬件引擎设置参数。我们首先使用算法1计算内存需求，并估算当前层在没有干扰时的延迟；这些值由层和硬件配置决定，因此是确定的。此值用于计算工作负载的得分。得分表示工作负载的相对重要性，根据工作负载的优先级，以及在目标截止时间内完成整个网络所剩时间是否充足来计算。每一层都会更新延迟预测（remain_prediction），因为得分会根据尚未执行的层进行更新。每次动态得分更新时，都会重新配置 MOCA 硬件引擎，因为随着协同运行的工作负载发生变化，相对重要性和系统竞争也会随时间改变。MOCA 中的动态优先级同时考虑服务级别协议目标和用户定义的静态优先级。在算法2中，priori_score 是 MOCA 使用的动态得分，slack 用于计算达到服务级别协议目标的剩余时间，而 user_priority 是静态优先级。因此，当检测到内存争用并对内存资源进行分区时，MOCA 会优先处理优先级较高的应用程序，以及距离目标期限更近的应用程序。

D. MoCA调度器

沿着系统栈向上，我们还需要一个工作负载调度器，以智能地选择可在系统中并发执行的应用程序。具体而言，多租户调度器会考虑以下要求：

1) 调度器需要了解不同用户指定优先级的存在，并据此作出响应。2) 调度器需要了解工作负载的内存和计算资源需求。3) 调度器算法需要足够轻量、简单，能够实时计算，且不会造成显著开销。

现有的多租户调度器未考虑内存资源 [9], [18] 以及用户定义的优先级和延迟目标 [26], [30], ，因此不适合在不同部署场景下处理多样化的工作负载。

为满足这些要求，我们设计了 MOCA 调度器，如算法3所述。具体而言，MOCA 调度器在选择并行运行工作负载时，会同时考虑每个任务由用户定义的优先级，以及任务已被调度运行的时长。具体来说，MOCA 使用 TaskQueue 存储已调度的工作负载。如果模型内各层的计算内存比存在显著差异，则可按模型粒度定义每个任务，也可将其定义为若干层的组。每个任务队列条目都包含 the

算法3 MOCA 多租户执行调度器

▷ % 在调度周期内，计算并更新基于用户指定优先级的得分 %
curr =当前_时间()
对每个任务$_i$ 属于 TaskQueue **执行**
    得分$_i$ ←用户_指定_优先级$_i$
    等待时间$_i$ ← 当前 −派发时间$_i$
    减速比$_i$ ← $\frac{\text{等待时间}_i}{\text{估计时间(任务}_i\text{)}}$
    得分$_i$ +=减速比$_i$
    ▷ % 如果任务属于内存密集型，则将其标记 %
    如果 平均估计值_带宽$_i$ > 0.5 ×动态随机存取存储器带宽_ **则**
        是否为内存密集型任务[i] ← 真
    else
        是否为内存密集型任务[i] ← 假
    结束 if
结束 for
▷ % 根据更新后的得分填充 ExQueue %
ExQueue ← [任务$_i$ if 得分$_i$ > 阈值]
ExQueue.sort()
▷ % 根据得分和并发运行任务构成任务组，并内存密集度 %
任务组 = []
当 资源可用**时**
    当前任务 =ExQueue.pop()
    任务组.append(当前任务)
    如果是否为内存密集型任务(当前任务) **则**
        协同任务 =查找非内存密集型任务(ExQueue)
        任务组.追加(协同任务)
    结束 如果
结束 while 循环
返回任务组

任务 ID、任务队列的调度时间、工作负载的当前状态、用户指定优先级和目标延迟。

在每一轮调度中，MOCA 调度器都会比较各个任务的得分。得分同时考虑每个任务由用户设定的优先级，以及任务在队列中的等待时长；随后，调度器会选出排名最高的任务，并将其放入执行队列。此外，MOCA 调度器在调度时还会考虑不同任务的内存资源需求。对于需要大量内存使用的内存密集型层，MOCA 调度器会将它们与队列中排在后面、内存需求较低的其他层协同调度。

E. MOCA 的局限性

MOCA 利用了深度神经网络工作负载和深度神经网络硬件的规律性，假设深度神经网络工作负载是稠密的，并且不利用数据的稀疏性。它也最适用于空间硬件架构，因为执行稠密深度神经网络层的性能主要取决于可用的计算和内存资源。这是因为，如果硬件考虑稀疏性，可能会面临挑战，无法.....

参数	值
脉动阵列维度（每个瓦片）	16x16
暂存器大小（每个瓦片）	128KiB
累加器大小（每个瓦片）	64KiB
加速器瓦片数量	8
共享二级缓存大小	2MB
共享二级缓存存储体	8
DRAM带宽	16GB/s
频率	1GHz

表II：评估中使用的片上系统配置。

估算 DNN 层在运行时的内存需求。然而，MOCA 可以通过集成针对稀疏深度神经网络的精确性能和内存资源预测器进行扩展，以支持稀疏深度神经网络加速器。

IV. 方法

本节详细介绍 MOCA 的实现、评估中使用的工作负载和指标，以及用于与 MOCA 进行比较、以评估其有效性的基线共置方案。

A. MoCA 实现

我们使用 Chisel RTL 语言 [1] 在 Gemmini [17] 基础设施之上实现 MOCA 硬件。Gemmini 是一种不支持多租户的基于脉动阵列的深度神经网络加速器。我们使用 FireSim（一种周期精确的、基于现场可编程门阵列的加速RTL模拟器） [27]，对深度神经网络工作负载进行完整的端到端运行，以评估性能。我们还在 GlobalFoundries 12纳米工艺上综合硬件实现，以评估面积开销。

表II展示了我们在 MOCA 评估中使用的片上系统配置，该配置类似于现代片上系统中的配置 [46]。我们配置 Gemmini——一种 TPU 风格的脉动阵列加速器生成器，使其在同一片上系统上生成八个同构且彼此独立的加速器瓦片，每个瓦片都可以运行不同的深度神经网络工作负载。多个加速器瓦片也可以协同运行同一深度神经网络工作负载的不同层/区域。每个加速器瓦片都配备一个 16x16 的权重驻留脉动阵列，用于矩阵乘法和卷积，同时还配备一个私有暂存存储器，用于存储 W、IA 和 OA。所有瓦片还共享对共享存储器子系统的访问，包括共享末级缓存和动态随机存取存储器，而这正是共置工作负载发生争用的主要来源。

MOCA 运行时系统和调度器使用 C 语言++ 实现，并运行在完整的 Linux 软件栈之上。MOCA 使用轻量级软件查找表来实现记分板，用于管理每个应用程序的带宽使用情况，并通过队列跟踪已调度的任务。用于计算 MOCA 运行时系统和 MOCA 调度器上的硬件配置的算法也通过软件实现，观察到的开销很小。

我们综合并对启用 MOCA 的加速器进行布局布线。我们使用 Cadence Genus 和 GlobalFoundries 12nm 用于综合的工艺技术。对于布局布线，我们使用 Cadence Innovus。

工作负载	模型 size	领域	DNN 模型
工作负载 A 组	轻量级	图像分类	SqueezeNet [23]
		目标检测	Yolo-LITE [21]
		语音处理	关键词识别 [51]
工作负载 B组	重型	图像分类	GoogleNet [48], AlexNet [29], ResNet50 [20]
		目标检测	YoloV2 [45]
工作负载 C组	混合	All	All

表 III：评估中使用的基准深度神经网络及按模型大小划分的工作负载集合。

B. 工作负载

a) 基准深度神经网络：在评估中，我们选择了七种不同的先进深度神经网络推理模型，包括 SqueezeNet [23], GoogLeNet [48], AlexNet [29],YOLOv2 [45], YOLO-lite [21], 关键词检测 [51],和 ResNet [20]。这些深度神经网络模型代表了多种深度神经网络工作负载，其模型大小、深度神经网络内核类型、计算和内存需求以及计算与内存之间的权衡各不相同。我们根据深度神经网络模型的大小将工作负载划分为不同的工作负载集合，以捕捉各集合之间可能存在的不同特征。工作负载分类基于 [18], ，该模型是我们的基线之一。表 III 展示了按大小分类的深度神经网络基准。工作负载集合 A 包含较小的模型，而工作负载集合 B 则包含较大的模型。工作负载集合 C 同时包含两者，即集合 A 和集合 B 的混合。

b) 多租户工作负载集合：为生成多租户场景，我们从随机分派到系统中的推理任务中选取 N 个不同任务，[44], 其中 N 的范围为 200 至 500。我们为这些任务分配用户自定义的静态优先级，范围为 0 至 11，并遵循与 Prema 和 Planaria 所用方法一致的 [11], [37], 分布。MOCA 在调度时会考虑优先级，并动态调整不同的层。

c) 服务质量目标：我们根据 [4]设定基线服务质量，因为我们的每个加速器瓦片都接近边缘设备。除基线服务质量外，我们还将延迟目标调整为 1.2×和 0.8×服务质量，这会使延迟目标分别增加和减少 20%，以评估 MOCA 对不同延迟目标的响应。QoS-H（高）表示 0.8× 服务质量，实现起来更具挑战性。QoS-L（低）表示 1.2× 服务质量，负载较轻。QoS-M 表示基线服务质量目标。

C 语言。指标

我们使用 [16], 中提出的指标来量化 MOCA 实现的工作负载共置的有效性，包括满足服务级别协议（SLA）的工作负载所占百分比、共置应用程序的吞吐量，以及 MOCA 管理共享资源策略的公平性。每个工作负载的延迟从其被调度到系统中开始计算，直至其完成并提交为止，其中包括其在任务队列中的等待时间和运行时

从任务被调度到系统中，到任务完成并提交所需的时间，其中包括任务在任务队列中的等待时间和运行时。

a) SLA 满意率：我们根据第四节 IV-B0c 中“服务质量目标”一段定义的三个不同服务质量级别，为每个工作负载设定服务级别协议目标（服务质量目标和服务级别协议目标可互换使用）。除整体 SLA 满意率外，我们还测量各优先级组的 SLA 满意率，以突出 MOCA 的有效性。

b) 公平性：公平性 指标衡量所有程序进度相等的程度。公平性指标用于评估 MOCA 调度器中使用的 MOCA 优先级评分方法，以及 MOCA 运行时中的动态分区内存带宽。此处，我们用 C_i 表示第 i 个工作负载的周期时间，*single* 表示片上系统上仅运行一个工作负载，*MT* 表示多租户执行。与其他关于深度神经网络多租户支持的先前工作类似 [9],[18], ，我们采用公平性的一般化版本 *fairness* 来衡量比例进度(PP)，定义如下：

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

c) 吞吐量：我们还分析了总系统吞吐量（STP），以评估 MOCA 通过调节内存访问速率来提高硬件资源整体利用率的效果。STP 定义为执行n个程序时的系统吞吐量，即将每个程序的归一化进度相加，其取值范围为 1 到n。提高 STP 需要在多个应用程序共置运行时，最大限度地提升整体进度。

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

D. 基线

为评估 MOCA 的有效性，我们将其与三个不同的基线进行比较：

1) Prema [9], ，它通过时间复用深度神经网络加速器，在不同深度神经网络之间实现多租户执行，并采用优先级感知调度算法；2) 静态计算分区，它利用多个工作负载的空间共置，但不会在运行时重新划分资源；3) Planaria [18], ，它通过动态划分计算资源，以固定的计算资源与内存资源比例实现多个深度神经网络的空间共置；

此外，我们没有采用逐层粒度来重新配置资源，而是将深度神经网络划分为由多个层组成的层块，并以层块粒度进行重新配置；近期研究表明，层块粒度能够带来卓越的性能 [33]。为确保公平比较，我们在相同的硬件配置下将 MOCA 与上述基线进行比较。

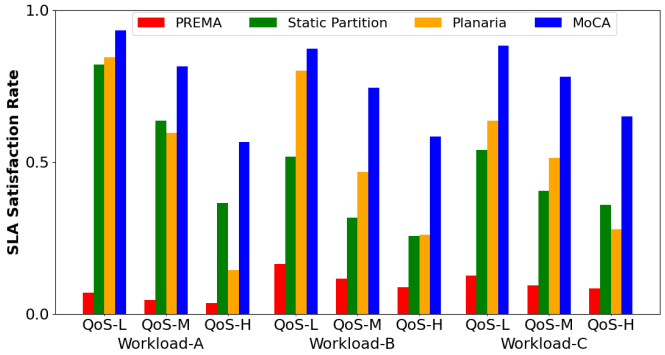


图 5：与具有不同服务质量目标的多租户基线相比，MOCA 的服务级别协议满足率提升（QoS-L：轻度延迟目标，QoS-M：中等延迟目标，QoS-H：严格延迟目标），以及不同深度神经网络工作负载规模（工作负载 A：轻型模型，工作负载 B：重型模型，工作负载 C：全部模型）。

V. 评估

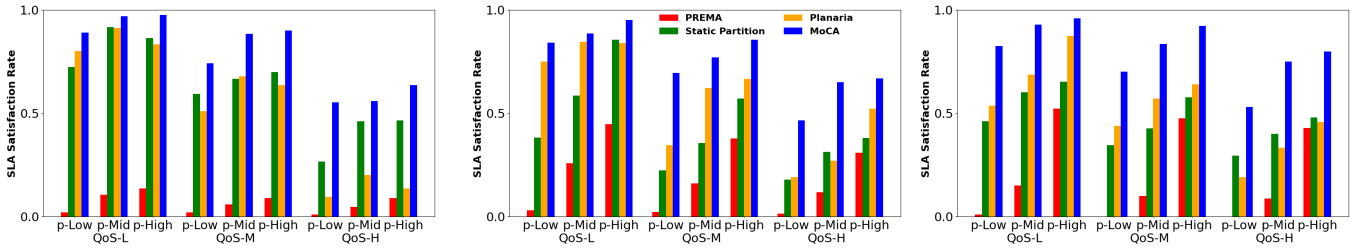
本节中，我们将 MOCA 支持的多租户执行与第四节讨论的三个基线进行比较，以评估其有效性。具体而言，我们将 MOCA 与 Prema [9]和 Planaria [18],进行比较；这两项近期研究均致力于改进深度神经网络的多租户执行。

我们证明了 MOCA 能够满足具有不同优先级的不同工作负载的目标服务质量，并将我们的公平性指标与其他基线方案进行了比较。除了提高 SLA 满意率之外，MOCA 还能够在包含不同 DNN 模型和服务质量要求的各种工作负载场景中，提高资源利用率和系统吞吐量。最后，除了上述性能提升之外，我们还证明，实现 MOCA 的硬件组件仅会给 DNN 加速器带来很小的面积开销。

A. SLA 满意率

我们评估了三组不同的工作负载（列于表 III 中），每组工作负载对应三种不同的 QoS 目标（高：QoS-H，中：QoS-M，低：QoS-L），总共产生九种不同的运行时场景，并将每种场景的 SLA 满意率与三个基线进行了比较。如图 5 所示，在所有测试场景中，MOCA 始终优于我们所有的基线多租户执行机制。在所有场景中，与 Prema 相比，MOCA 的几何平均值提升了 8.7× ，最高提升达 18.1×。MOCA 的 SLA 满意率比静态分区基线的几何平均值高出 1.8× ，在最佳情况下最高高出 2.4× 。最后，MOCA 的 SLA 满意率最高比 Planaria 高出 3.9× ，几何平均值高出 1.8× 。

与基线方案相比，MOCA 在 QoS-H 场景中的提升最为显著。在该场景下，SLA目标最难达成，这表明 MOCA 能够通过自适应地分配资源，有效满足多租户场景中的性能目标



(a) 工作负载 A　　(b) 工作负载 B　　(c) 工作负载 C

图 6：不同优先级组（p-Low：低优先级，p-Mid：中优先级，p-High：高优先级）和不同深度神经网络工作负载规模（工作负载 A：轻型模型，工作负载 B：重型模型，工作负载 C：全部模型）的 SLA 满意率细分。

。对于较宽松的目标，Planaria 的表现优于静态基线，但随着所需 QoS 级别提高，其性能会急剧下降。对于较小的 DNN 模型，这一现象尤为明显，例如工作负载 A 中的轻量级模型：随着资源重新配置次数增加，计算核心重新分区中的线程迁移开销会显著增大。我们发现，由于线程生成和同步，计算线程重新分区中的线程迁移平均需要 1M 个周期，这与近期研究 [33]，中的观察结果类似；对于运行时间相对较短的小型模型而言，这会造成显著开销。另一方面，MOCA 中的内存分区只需 5–10 个周期即可重新配置 DMA 的发出速率。因此，MOCA 触发内存重新分区的频率高于计算资源重新分区，以避免其高昂的开销。

另一方面，当较重的工作负载被组合在一起时，如工作负载 B，Planaria 的表现优于静态基线，但其提升幅度不及 MOCA，尤其是在服务质量要求变得更加严格时。这是因为工作负载 B 中的大多数工作负载都是内存密集型，因而会不利地降低并发运行层的性能。因此，试图仅在运行时重新配置计算资源来维持这些较重工作负载性能的方案，可能会频繁触发重新配置，却收效甚微，因为内存争用导致的减速比无法通过仅重新配置计算资源来解决。

相反，MOCA 会动态检测内存争用并调节内存访问，因此相比重新配置计算资源，它能更有效地适应大型共置 DNN 模型的需求。此外，重新配置 MOCA 硬件无需付出高昂的线程迁移代价，只需向加速器发出新的硬件配置命令即可。因此，凭借智能的内存争用管理以及较低的线程迁移开销，MOCA 的性能优于以往的工作。

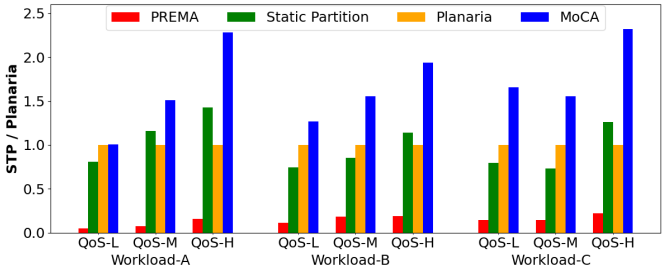


图 7：在不同服务质量目标和不同深度神经网络工作负载规模下，MOCA 相较于所评估的多租户基线方案的 STP 提升（以 Planaria 基线为标准进行归一化）。

B. 优先级分析

我们进一步分析 SLA 满意率，并根据用户给定的优先级分数将结果划分为不同的优先级组。我们根据[37], 中的分布对工作负载进行优先级分类，其中优先级分数范围为 0 到 11。为便于可视化，我们将优先级分为几类：低优先级（p-Low，分数为 0 到 2）、中优先级（p-mid，分数为 3 到 8）和高优先级（p-high，分数为 9 到 11）。图 6 展示了按不同优先级划分的满意率。对于高优先级组，MOCA 的 SLA 满意率比 Planaria 高 4.7×（工作负载 A，QoS-H），比静态分区基线高 1.8×（工作负载 C，QoS-H），比 Prema 高 9.9×（工作负载 A，QoS-M）。随着工作负载优先级提高，MOCA 和各基线的 SLA 满意率总体上都有所提升。尤其是，只有 MOCA 能够在所有工作负载、服务质量和优先级场景下始终提供可靠性能，而其他系统在某些情况下会出现性能下降。例如，对于工作负载 A，Planaria 在高优先级工作负载上的性能不如中优先级工作负载，因为它通过激进地为高优先级工作负载争取更多计算资源，导致线程迁移成本高昂，而性能收益却很小。

C. 吞吐量分析

图 7 展示了 MOCA 相较于基线多租户策略所实现的系统吞吐量提升，结果以 Planaria 的系统吞吐量为基准进行归一化。MOCA 实现了 12.5× 的几何平均值提升，

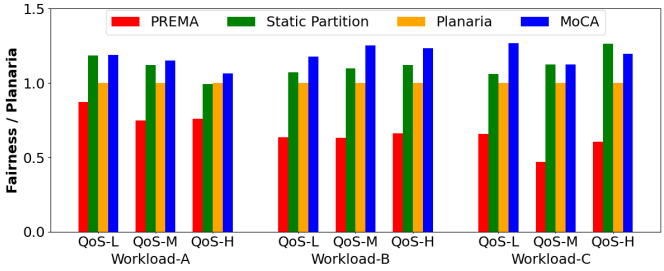


图 8：在不同服务质量目标和不同深度神经网络工作负载规模下，MOCA 相较于经评估的多租户基线方案（以 Planaria 基线为基准进行归一化）的公平性提升。

相较于 Prema，最高提升可达 20.5×（适用于工作负载 A、服务质量-QoS-L 和工作负载 A、服务质量-QoS-M）。与静态分区基线相比，MOCA 的几何平均值高出 1.7×，在某些工作负载（工作负载 C、服务质量-QoS-M）中提升最高可达 2.1×。最后，相较于 Planaria，MOCA 的几何平均值提升了 1.7×，最高提升可达 2.3×（适用于工作负载 C、服务质量-QoS-H 和工作负载 A、服务质量-QoS-H）。

工作负载 A（使用轻型模型）相较于 Planaria 在 MOCA 上的提升最为显著，其中大部分提升源于 MOCA 的线程开销出现频率较低。在较轻型 DNN 模型的 QoS-H 场景下，Planaria 的吞吐量表现不佳，因为在线程迁移期间它无法有效利用计算资源，而线程迁移的开销与轻型模型的实际运行时间相当。

对于工作负载 B 组（使用重型模型），MOCA 会自适应地调节重型 DNN 模型之间的争用，从而提升整体 STP。MOCA 运行时系统会在运行期间动态检测争用，随后 MOCA 硬件可通过限制内存受限层的过量内存访问来解决争用，限流上限则根据优先级和松弛时间计算。通过控制内存访问速率，MOCA 能比基线替代方案更好地推动并发运行的应用程序取得进展，从而提高整体 STP。

工作负载 C（包含全部模型）进一步提升了性能，这主要是因为它包含了更合理搭配的、位于同一位置的内存密集型和计算密集型深度神经网络层。将任务调度队列靠后的内存密集型层组与计算密集型层组进行层分组，有助于提高处理单元阵列的利用率，从而提升系统吞吐量。尽管 Planaria 会进行计算资源重新分区，但当性能下降源于内存争用时，其效果有限，因为它的调度器没有考虑并发运行的应用程序的内存需求。在这种情况下，多个内存密集型层可能被同时调度，导致性能不理想。

D. 公平性分析

我们评估了第四节中定义的共置公平性，以表明 MOCA 能够在不损害整体公平性的情况下，提高系统级吞吐量。

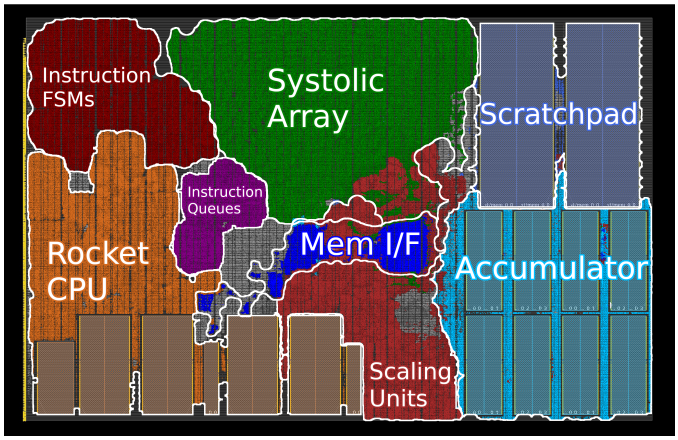


图 9：采用 MOCA 的加速器瓦片布局。

系统。图 8 比较了 MOCA 与另外三个基线系统的公平性。与 Prema 相比，MOCA 的几何平均值提升了 1.8×，最高提升 2.4×（工作负载 C、QoS-M）。与静态分区基线相比，MOCA 的几何平均值提升了 1.07×，最高提升 1.2×（工作负载 C、QoS-L）。最后，与 Planaria 相比，MOCA 的几何平均值提升了 1.2×，最高提升 1.3×（工作负载 C、QoS-L）。总体而言，与所有基线相比，MOCA 的公平性均有所提升。此外，MOCA 在工作负载 B 上的公平性优势最为显著，因为该工作负载中内存密集型层的占比最高。MOCA 的争用监控和解决机制有助于缓解系统级争用，从而避免内存受限层的并发运行应用程序受到不公平的资源限制。通过内存调节，MOCA 的公平性优于静态分区方案，而后者并未利用争用管理。

我们还观察到，在包含全部深度神经网络的工作负载 C 中，与静态分区系统相比，MOCA 的公平性略低。这是因为当可供调度的层种类繁多、计算和内存需求各不相同时，MOCA 的内存感知调度器倾向于将短时运行的计算密集型层与长时运行的内存密集型层分组，以平衡工作负载组合、最大限度地提高系统级吞吐量，如图 7 所示。与此同时，这可能会使一些优先级较低、运行时间较短的模型提前完成，从而略微降低公平性指标。

E. 物理设计与面积分析

我们使用 Cadence Genus 和 GlobalFoundries 的 12纳米工艺，综合实现了支持 MOCA 的深度神经网络加速器。我们还使用 Cadence Innovus 对该加速器进行了布局布线，如图 9 所示。MOCA 硬件部署在加速器的存储器接口中。

如表IV所示，MOCA仅为深度神经网络加速器带来少量面积开销，使

组件	面积（μ米^2）	占系统面积的百分比
Rocket CPU	101K	20.5%
暂存存储器	58K	11.7%
累加器	75K	15.2%
脉动阵列	78K	15.8%
指令队列	14K	2.8%
内存接口 不含 MOCA	8.6K	1.7%
MOCA 硬件	0.1K	0.02%
Tile	493K	100%

表IV：采用 MOCA 的加速器瓦片面积细分。

加速器的存储器接口面积仅增加1.7%，整个加速器（包括功能单元）的面积仅增加0.02%。

VI. 结论

本文提出、实现并分析了一种面向深度神经网络工作负载的自适应、以内存为中心的多租户加速器架构 MOCA。与旨在划分深度神经网络硬件计算资源的现有方案不同，MOCA 专注于动态管理协同运行的应用程序所共享的内存资源，在考虑优先级差异的同时确保满足服务质量目标。MOCA 利用深度神经网络及加速器的规律性，根据应用程序的延迟目标和用户定义的优先级，估算并调控内存资源的使用情况。我们针对多种深度神经网络执行场景和不同目标约束进行了全面评估，结果表明，与已有工作相比，MOCA 可将 SLA 满意率最高提升至 3.9×，整体工作负载的几何平均加速比达到 1.8× ，高优先级工作负载的加速比最高达到 4.7×。此外，MOCA 可将整体系统吞吐量最高提升至 2.3× ，同时将公平性最高提升至 1.3×。最后，MOCA 在带来这些收益的同时，面积开销不到最先进深度神经网络加速器的 1%。

VII. 致谢部分

本研究部分由美国政府通过 DARPA RTML 计划（合同 FA8650-20-2-7006）资助。本研究还部分得到了 NSF 奖项 CCF-1955450 的支持，以及 SLICE 实验室的企业赞助方和合作伙伴的部分支持。本文所表达的观点和结论仅代表作者，不应被解读为代表美国政府的官方政策，无论这些政策是明示还是暗示。

参考文献

[1] J. Bachrach、H. Vo、B. Richards、Y. Lee、A. Waterman、R. Avižienis、J. Wawrzynek 和 K. Asanovi´c，《Chisel：使用 Scala 嵌入式语言构建硬件》，发表于设计自动化会议，2012 年。[2] E. Baek、D. Kwon 和 J. Kim，《一种多神经网络加速架构》，发表于*2020* 年 *ACM/IEEE* 第 *47* 届年度国际计算机体系结构研讨会（*ISCA*），2020 年。[3] S. Bansal、V. Tolani、S. Gupta、J. Malik 和 C. Tomlin，《结合最优控制与学习实现新环境中的视觉导航》，2019 年。[4] S. Bianco、R. Cadene、L. Celona 和 P. Napoletano，《代表性深度神经网络架构的基准分析》，*IEEE Access*，第 6 卷，2018 年。

[5] B. Boroujerdian、H. Genc、S. Krishnan、W. Cui、A. Faust 和 V. J. Reddi，《Mavbench：微型飞行器基准测试》，载于国际微体系结构研讨会（*MICRO*）论文集，2018 年。[6] Q. Chen、H. Yang、M. Guo、R. Kannan、J. Mars 和 L. Tang，《Prophet：在不可抢占加速器上进行精确的服务质量预测，以提高仓库规模计算机的利用率》，*ACM SIGARCH*计算机体系结构新闻，第 45 卷，2017 年 4 月。[7] Q. Chen、H. Yang、J. Mars 和 L. Tang，《Baymax：面向仓库规模计算机中不可抢占加速器的服务质量感知与利用率提升》，载于第二十一届国际编程语言与操作系统体系结构支持会议论文集。ACM，2016 年。[8] H.-Y. Cheng、C.-H. Lin、J. Li 和 C.-L. Yang，《通过线程节流降低内存延迟》，载于*2010* 年第 *43* 届 *IEEE/ACM* 微体系结构年度国际研讨会，2010 年。[9] Y. Choi 和 M. Rhu，《Prema：一种面向可抢占神经处理单元的预测式多任务调度算法》，*2020 IEEE* 高性能计算机体系结构国际研讨会（*HPCA*），2020 年。[10] H. Cook、M. Moreto、S. Bird、K. Dao、D. Patterson 和 K. Asanovic，《通过缓存分区提升利用率和能效，同时保持响应能力的硬件评估》，*ACM SIGARCH*计算机体系结构新闻，第 41 卷，2013 年 6 月。[11] G. Da Costa、L. Grange 和 I. De Courchelle，《类 Google 大规模工作负载的建模与生成》，载于*2016* 年第七届国际绿色与可持续计算会议（*IGSC*），2016 年。[12] C. Delimitrou 和 C. Kozyrakis，《Paragon：面向异构数据中心的服务质量感知调度》，载于国际编程语言与操作系统体系结构支持会议（*ASPLOS*）论文集，2013 年。[13] ——，《Quasar：资源高效且具备服务质量感知能力的集群管理》，载于国际编程语言与操作系统体系结构支持会议（*ASPLOS*）论文集，2014 年。[14] J. Devlin、M. Chang、K. Lee 和 K. Toutanova，《BERT：用于语言理解的深度双向变换器预训练》，*CoRR*，2018 年。[15] E. Ebrahimi、C. J. Lee、O. Mutlu 和 Y. Patt，《通过源节流实现公平性：适用于多核内存系统的可配置高性能公平性基础机制》，*ACM*计算机系统汇刊，第 30 卷，2012 年。[16] S. Eyerman 和 L. Eeckhout，《多程序工作负载的系统级性能指标》，*IEEE Micro*，第 28 卷，第 3 期，2008 年。[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 和 Y. S. Shao，《Gemmini：通过全栈集成实现系统化深度学习体系结构评估》，载于设计自动化会议，2021 年。[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 和 H. Esmaeilzadeh，《Planaria：用于深度神经网络空间多租户加速的动态体系结构裂变》，载于*2020* 年第 *53* 届 *IEEE/ACM* 微体系结构年度国际研讨会（*MICRO*），2020 年。[19] F. Guo、Y. Solihin、L. Zhao 和 R. Iyer，《为芯片多处理器提供服务质量的框架》，载于第 *40* 届 *IEEE/ACM* 微体系结构年度国际研讨会（*MICRO 2007*），2007 年。[20] K. He、X. Zhang、S. Ren 和 J. Sun，《用于图像识别的深度残差学习》，载于*2016* 年 *IEEE*计算机视觉与模式识别会议（*CVPR*），2016 年。[21] R. Huang、J. Pedoeem 和 C. Chen，《Yolo-lite：一种针对非 GPU 计算机优化的实时目标检测算法》，载于*2018* 年 *IEEE* 大数据国际会议（*Big Data*）。IEEE计算机学会，2018 年 12 月。[22] M. Huzaifa、R. Desai、X. Jiang、J. Ravichandran、F. Sinclair 和 S. V. Adve，《借助 ILLIXR 探索扩展现实：体系结构研究的新试验平台》，*IISWC*，2021 年。[23] F. N. Iandola、S. Han、M. W. Moskewicz、K. Ashraf、W. J. Dally 和 K. Keutzer，《SqueezeNet：以少 50 倍的参数和 <0.5 MB 模型大小达到 AlexNet 级准确率》，*arXiv:1602.07360*，2016 年。[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 和 D. H. Yoon，《数据中心内张量处理单元的性能分析》，载于国际计算机体系结构研讨会（*ISCA*）论文集，2017 年。[25] M. Kandemir、S. P. Muralidhara、S. H. K. Narayanan、Y. Zhang 和 O. Ozturk，《优化芯片多处理器的共享缓存行为》，载于*2009* 年第 *42* 届 *IEEE/ACM* 微体系结构年度国际研讨会（*MICRO*），2009 年。[26] S.-C. Kao 和 T. Krishna，《MAGMA：一种将多个深度神经网络映射到多个加速器核心的优化框架》，载于*2022 IEEE* 高性能计算机体系结构国际研讨会（*HPCA*），2022 年。[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 和 K. Asanovic，《FireSim：公有云中基于现场可编程门阵列加速的周期精确横向扩展系统仿真》，载于*2018* 年 *ACM/IEEE* 第 *45* 届计算机体系结构年度国际研讨会（*ISCA*），2018 年。[28] S. Kim、D. Chandra 和 Y. Solihin，《芯片多处理器体系结构中的公平缓存共享与分区》，载于第 *13* 届并行体系结构与编译技术国际会议论文集，系列丛书 PACT '04。美国：IEEE计算机学会，2004 年。[29] A. Krizhevsky、I. Sutskever 和 G. E. Hinton，《使用深度卷积神经网络进行 ImageNet 分类》，载于第 *25* 届神经信息处理系统国际会议论文集——第 *1* 卷，系列丛书 NIPS'12。美国纽约州雷德胡克：Curran Associates Inc.，2012 年。[30] H. Kwon、L. Lai、M. Pellauer、Y.-H. Chen、T. Krishna 和 V. Chandra，《面向多深度神经网络工作负载的异构数据流加速器》，载于高性能计算机体系结构国际研讨会（*HPCA*）论文集，2021 年。[31] B. Li、L. Zhao、R. R. Iyer、L.-S. Peh、M. Leddige、M. Espig、S. E. Lee 和 D. Newell，《CoQoS：协调基于片上网络的片上系统中的服务质量感知共享资源》，并行与分布式计算杂志，第 71 卷，2011 年。[32] C. Liu、A. Sivasubramaniam 和 M. Kandemir，《为芯片多处理器抵御内存墙构筑最后一道防线》，载于第 *10* 届高性能计算机体系结构国际研讨会（*HPCA'04*），2004 年。[33] Z. Liu、J. Leng、Z. Zhang、Q. Chen、C. Li 和 M. Guo，《Veltair：通过自适应编译与调度实现高性能多租户深度学习服务》，载于第 *27* 届 *ACM* 国际编程语言与操作系统体系结构支持会议论文集，系列丛书 ASPLOS '22，2022 年。[34] D. Lo、L. Cheng、R. Govindaraju、P. Ranganathan 和 C. Kozyrakis，《Heracles：大规模提升资源效率》，载于*2015* 年 *ACM/IEEE* 第 *42* 届计算机体系结构年度国际研讨会（*ISCA*），2015 年。[35] J. Mars、N. Vachharajani、R. Hundt 和 M. L. Souffa，《争用感知执行：在线争用检测与响应》，载于国际代码生成与优化研讨会（*CGO*）论文集，2010 年。[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 和 M. Zaharia，《MLPerf 训练基准测试》，载于机器学习与系统论文集，第 2 卷，2020 年。[37] P. Minet、É. Renault、I. Khoufi 和 S. Boumerdassi，《分析来自 Google 数据中心的跟踪记录》，载于*2018* 年第 *14* 届国际无线通信与移动计算会议（*IWCMC*），2018 年。[38] O. Mutlu 和 T. Moscibroda，《面向芯片多处理器的停顿时间公平内存访问调度》，载于第 *40* 届 *IEEE/ACM* 微体系结构年度国际研讨会（*MICRO 2007*），2007 年。[39] ——，《并行度感知批处理调度：同时提升共享动态随机存取存储器系统的性能与公平性》，载于*2008* 年国际计算机体系结构研讨会，2008 年。[40] Y. H. Oh、S. Kim、Y. Jin、S. Son、J. Bae、J. Lee、Y. Park、D. U. Kim、T. J. Ham 和 J. W. Lee，《LayerWeaver：最大化资源利用率的通过逐层调度实现神经处理单元"，发表于*IEEE*国际高性能计算机体系结构研讨会（*HPCA*），2021年。[41] A. Parashar、P. Raina、Y. S. Shao、Y.-H. Chen、V. A. Ying、A. Mukkara、R. Venkatesan、B. Khailany、S. W. Keckler和J. Emer，"Timeloop：一种系统化的深度神经网络加速器评估方法"，发表于国际系统与软件性能分析研讨会（*ISPASS*）论文集，2019年。[42] M. K. Qureshi和Y. N. Patt，"基于效用的缓存分区：一种低开销、高性能的共享缓存运行时分区机制"，发表于*2006*年第*39*届*IEEE/ACM*国际微体系结构年会（*MICRO'06*），2006年。[43] N. Rafique、W.-T. Lim和M. Thottethodi，"多核处理器中DRAM带宽的有效管理"，发表于第*16*届国际并行体系结构与编译技术会议（*PACT 2007*），2007年，第245–258页。[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和Y. Zhou，"MLPerf推理基准测试"，发表于国际计算机体系结构研讨会（*ISCA*）论文集，2020年。[45] J. Redmon和A. Farhadi，"YOLO9000：更好、更快、更强"，*2017*年*IEEE*计算机视觉与模式识别会议（*CVPR*），2016年。[46] F. Sijstermans，"NVIDIA深度学习加速器"，发表于*Hot Chips*，2018年。[47] N. Sünderhauf、O. Brock、W. Scheirer、R. Hadsell、D. Fox、J. Leitner、B. Upcroft、P. Abbeel、W. Burgard、M. Milford 等人，"深度学习在机器人学中的局限与潜力"，《国际机器人研究杂志》，第37卷，2018年。[48] C. Szegedy、W. Liu、Y. Jia、P. Sermanet、S. Reed、D. Anguelov、D. Erhan、V. Vanhoucke和A. Rabinovich，"通过卷积网络实现更深层次学习"，发表于*2015*年*IEEE*计算机视觉与模式识别会议（*CVPR*），2015年。[49] L. Tang、J. Mars和M. L. Soffa，"面向友好性的编译：缓解仓库规模计算机中的服务质量争用"，发表于第十届国际代码生成与优化研讨会论文集，CGO '12系列，2012年。[50] L. Tang、J. Mars、W. Wang、T. Dey和M. L. Soffa，"ReQoS：面向仓库规模计算机服务质量的反应式静态/动态编译"，发表于国际编程语言与操作系统体系结构支持会议，2013年。[51] R. Tang和J. Lin，"面向小型关键词检测的深度残差学习"，发表于*2018*年*IEEE*国际声学、语音与信号处理会议（*ICASSP*），2018年。[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和P. Zhang，"Facebook上的机器学习：理解边缘推理"，发表于国际高性能计算机体系结构研讨会（*HPCA*）论文集，2019年。[53] H. Zhu、D. Lo、L. Cheng、R. Govindaraju、P. Ranganathan和M. Erez，"Kelp：机器学习平台中加速器的服务质量保障"，发表于国际高性能计算机体系结构研讨会（*HPCA*）论文集，2019年。

附录

A. 摘要

本工件附录部分介绍如何访问每个 MOCA 组件的工件，并按照第五节中的方法进行测试和评估。与第四节一样，我们将使用 FireSim 现场可编程门阵列加速仿真，在完整的软件栈环境中评估 MOCA。

B. 制品元信息清单

- **运行时环境：AWS FPGA 开发者 AMI 1.12.1**
- **硬件：AWS EC2 实例（c5.4xlarge、f1.2xlarge）**
- **需要多少磁盘空间？：200 GB（在 EC2 实例上）。**
- **实验：在基于 RISC-V 的深度神经网络加速器基础设施中集成 MOCA 的 FireSim 仿真，运行多租户推理查询。**
- **程序：Chisel（寄存器传输级）、C 语言（运行时、调度器）、Python（脚本）**
- **指标：目标满足率（满足目标截止时间的各项查询所占比例）、STP（系统吞吐量）以及公平性，定义见第 IV-C 节。**
- **输出：从 SoC 的 UART 输出中解析的结果，展示静态分区基线与 MOCA 在各个 QoS 级别和工作负载集合下的性能（SLA 满意率、系统吞吐量、公平性）对比柱状图。（图 5-8）**
- **准备工作流程需要多少时间？：2 小时（脚本化安装）。**
- **完成实验需要多少时间？：4小时（脚本化运行，脚本化结果解析）**
- **公开可用：是。**
- **代码许可证：有多种，详见下载内容。**

C. 描述

1) 获取方式：这些工件包括：1) FireSim：FPGA 加速 RTL 仿真环境（https://doi.org/10.5281/zenodo.7456139）；2) Chipyard：RISC-V 片上系统生成环境（https: //doi.org/10.5281/zenodo.7456073）；3) MOCA 硬件：在 Gem-mini 深度神经网络加速器上实现的 MOCA（https://doi.org/10.5281/zenodo. 7456052）。主要实现针对 DMA。4) MOCA 软件：MOCA 运行时和调度器的实现，以及测试。（https://doi.org/10.5281/zenodo. 7456045）。MOCA 运行时和调度器位于 imagenet 下，其中包括。

用户无需手动下载其余三个代码仓库——设置 FireSim 代码仓库时，系统会自动从 Zenodo 获取它们。

2）依赖项 - 硬件： 需要一台 AWS EC2 c5.4xlarge 实例（也称为“管理器”实例）和三台 f1.2xlarge 实例（我们将工作负载拆分到三台并行的 f1 实例上运行，以节省运行时；如果在单台实例上运行，则需要 8 小时以上）。后者将由 FireSim 管理器自动启动。我们提供了预构建的 FPGA 镜像，以避免 FPGA 构建流程耗时过长（ ~8 小时）。不过，如果用户想要构建自定义 FPGA 镜像，则还需要一台 z1d.2xlarge 实例。

3）依赖项 - 软件： 在本地计算机上使用 SSH 或 Mosh 远程访问 EC2 实例。其余所有要求均由后续章节中的脚本自动安装。

D. 安装

首先，按照 FireSim 网站[2] 上的说明创建一个 EC2 管理器实例。请完成“第 1.3.1.2 节：密钥设置，第 2 部分”及之前的所有步骤，并参考“第 1.3.1 节”中的以下建议：

1) 提示启动 c5.4xlarge 或 z1d.2xlarge 时，请选择 c5.4xlarge。2) 输入根 EBS 卷大小时，使用 200GB 即可。

完成 FireSim 文档中“第 1.3.1.2 节：密钥设置”及之前的所有步骤后，应该已经设置好管理器实例，并获得 IP 地址和密钥。使用 SSH 或 Mosh 登录该实例。从此处开始，所有命令都应在管理器实例上运行。

为进行制品评估，请运行以下命令克隆分叉的 FireSim 仓库（否则，用户也可以克隆 FireSim，并在 Gemmini 子模块中检出 MOCA 分支）：

对于制品评估，首先从 Zenodo 下载顶层 FireSim 仓库：

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

接下来运行以下命令，该命令将初始化全部依赖项，并执行 FireSim 和 Chipyard 的设置步骤（安装 RISC-V 工具链、安装匹配的主机工具链等）：

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

脚本运行完毕后，执行以下命令：

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

完成环境配置后，请按照“第 1.3.3 节：使用管理器完成设置”中的步骤操作。完成这些步骤后，即可开始评估 MOCA。

E. 实验流程

环境设置完成后，我们将运行 MOCA 工件。首先，我们将开始为 MOCA 构建工作负载。构建工作负载映像时，用户可以自行编译 Linux 二进制文件，也可以使用提供的预编译二进制文件。

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

1）构建包含工作负载的 *Linux* 镜像：1）在管理器实例上，使用基于 Buildroot 的 Linux 发行版构建与 FireSim 兼容的 RISC-V Linux 镜像。请遵循 FireSim 文档“第 2.1.1 节”中的说明。[3]2）运行以下命令，在 gemmini-rocc-tests 中创建构建目录。

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

3）（可选——如果用户希望自行编译二进制文件，请跳过此步骤）要测试所提供的预编译二进制文件，请运行以下命令将 imagenet-binary.zip 解压到 build/imagenet。注释掉 imagenet/Makefile 中 test 下的所有测试，以防止预编译二进制文件被覆盖。

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

4）在完整的 Linux 环境中运行以下命令，以构建使用 C 语言编写的 MOCA 运行时和调度器。运行这些命令将在 firesim/deploy/workloads 中生成一个 .json 文件。

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

2）运行 *FireSim* 模拟：

1) 前往 firesim/deploy，并在 config_hwdb.yaml 中粘贴 built-hwdb-entries/ 里提供的预构建 FPGA 镜像条目。同时，在 config_runtime.yaml 的 default_hw_config 中设置该条目。

2) 关于 config_runtime.yaml 中的其他配置，请参阅 FireSim 文档的“第 2.1.2 节：设置管理器配置”。

必须修改以下参数：

a) workload_name：gemmini-tests-workload.json。b) 在 run_farm_hosts_to_use 下，将要启动的 f1.2xlarge 数量增加到 3（f1.2xlarge：3）。c) 在 target_config 下，将 topology 替换为 three_no_net_config。d) 将 no_net_num_nodes 设置为 3，以并行启动三个 f1 实例。

3) 通过启动 f1.2xlarge 实例运行 FireSim 仿真。请按照 FireSim 文档中“第 2.1.3 节：启动仿真！”的说明操作。

4) 结果将被复制到一个目录中 deploy/results-workload。生成的结果目录将包含三个子目录，分别对应表 III 中定义的工作负载 A/B/C，因为每个 f1 实例都会运行每种工作负载类型。在 results-workload 目录中运行以下命令，将为每组工作负载生成图表。运行以下命令块中的第一条命令，解析优先级级别组中的结果，以及每个服务质量级别和工作负载模型组对应的总体结果（如图 6 和图 5 所示）。第二条命令用于解析图 7 和图 8 中的 STP 和公平性结果。脚本中的结果目录（$result_dir）应为包含三个子目录的目录。

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

测试脚本将在随机调度、随机分配优先级的 250 轮端到端推理查询上运行静态分区基线和 MOCA。请注意，默认情况下，此脚本不会为系统重新构建 FPGA 镜像，因为每次构建大约需要 8 小时。我们默认在 built-hwdb-entries/ 目录中提供预构建镜像，论文中的评估使用的就是这些镜像。

构建命令中使用的结果解析脚本（build_sla.sh 和 build_stp.sh）分别是 parse_result_from_uartlog.py 和 make_fair.py。这些 Python 脚本和构建脚本都包含在 results-workload 目录中。

我们在 EXAMPLE_RESULT/ 目录下提供了预编译二进制文件的示例结果图，供参考。

请确保运行中的 f1 实例已通过运行 firesim terminaterunfarm 终止，并在 AWS EC2 管理控制台中确认除管理器外没有其他实例。

F. 实验定制

1) 重新构建 *FPGA* 镜像：用户可以通过更改 Gemmini 深度神经网络加速器配置或其他片上系统配置来调整片上系统配置。例如，在 Gemmini 源码的 Config.scala 中，用户可以重新配置内部暂存存储器或累加器大小。修改 RocketConfigs.scala 中的缓存参数即可更改共享 L2 缓存大小。如需构建新的 FPGA 位流，请按照“第 3 节：构建自己的硬件设计”中的步骤操作[4]。

2) 自定义实验参数：进入 gemmini/software/gemmini-rocc-tests/include 目录。该目录中包含 MOCA 运行时、软件和参数的头文件。打开 gemmini.h，将 SEED 更改为新种子，以生成新的随机查询。要更改查询数量，请修改 total_workloads。重复第 E 节中的步骤以获取结果。

[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