面向机器学习推理的支持 GPU 的函数即服务

Ming Zhao亚利桑那州立大学
mingzhao@asu.edu

Kritshekhar Jha亚利桑那州立大学
kjha9@asu.edu

Sungho Hong亚利桑那州立大学
shong59@asu.edu

arXiv:2303.05601v1 [cs.DC] 9 Mar 2023

摘要——**函数即服务（FaaS）正逐渐成为一种重要的云计算服务模式，因为它能够提升各种应用的可扩展性和易用性，尤其是需要可扩展资源和复杂软件配置的机器学习（ML）推理任务。这些推理任务高度依赖 GPU 来实现高性能；然而，现有 FaaS 解决方案目前尚不支持 GPU。函数独特的事件触发和短生命周期特性，为在 FaaS 中启用 GPU 带来了新的挑战，必须考虑在 GPU 与主机内存之间传输数据（例如，机器学习模型参数和输入/输出）的开销。本文提出了一种新颖的支持 GPU 的函数即服务解决方案，使机器学习推理函数能够高效利用 GPU 加速计算。首先，该方案扩展了 OpenFaaS 等现有函数即服务框架，以支持在 FaaS 集群中的多个 GPU 上调度和执行函数。其次，该方案支持在 GPU 内存中缓存机器学习模型，以提升模型推理函数的性能，并对 GPU 内存进行全局管理，以提高缓存利用率。第三，该方案协同设计 GPU 函数调度和缓存管理，以优化机器学习推理函数的性能。具体而言，本文提出了局部性感知调度，在提高缓存命中时 GPU 内存利用率的同时，也最大限度地提高 GPU 核心的并行处理利用率。基于真实世界跟踪数据和机器学习模型进行的全面评估表明，所提出的支持 GPU 的函数即服务方案适用于机器学习推理任务；与默认的仅采用负载均衡的调度器相比，所提出的局部性感知调度器可实现 48 倍加速。**

索引词—**函数即服务、GPU 调度、缓存、机器学习推理**

I. 引言

函数即服务（FaaS）已成为一种新型云计算服务模式，使用户能够便捷地部署计算任务，并以较低成本快速扩展任务规模。然而，使用 FaaS 函数运行机器学习（ML）推理仍受到限制，因为当前的云服务提供商不支持或不直接提供可访问 GPU 资源的 FaaS 函数，而 GPU 资源对于加速计算密集型推理任务至关重要。例如，AWS Lambda [3]不会为 FaaS 提供 GPU；微软的 Azure Functions [9], FaaS 可以通过启用 GPU 的 Kubernetes 容器间接访问 GPU，但无法共享 GPU。因此，亟需在 FaaS 平台上启用 GPU，使包括机器学习推理在内的各类任务都能受益于这种服务模式。

本研究部分由美国国家科学基金会的 CNS-1955593 和 OAC-2126291 项目资助。

在 FaaS 平台上启用 GPU 对机器学习推理任务至关重要，因为这类任务计算密集，且需要低延迟才能满足服务级别协议（SLA）。生产环境中的机器学习推理应用对延迟有严格要求；例如，搜索栏中的自动建议需要在用户浏览关键词时实时返回推理结果 [9]。当输入数据可以组成大批次，且模型专为并行计算而设计时，使用 GPU 运行机器学习推理可以显著降低延迟。将一批输入数据一起处理，可使机器学习模型利用 GPU 的并行性并行处理这些数据。Transformer 等机器学习模型[27] 将循环神经网络（RNN）的顺序计算转化为独立计算，从而利用 GPU 并行化。

在 FaaS 平台中管理 GPU 资源颇具挑战，因为共享 GPU 与共享 CPU、内存等传统资源有所不同。GPU 的设计目标是通过将全部资源分配给单个 GPU 进程，最大限度地提高单个应用的吞吐性能。由于单个进程可以完全访问 GPU 资源，GPU 要求应用经过编程，避免超出可用 GPU 内存并导致内存溢出（OOM）错误。GPU 有限的共享能力也给 FaaS 平台带来了问题，因为函数需要动态共享 GPU 资源，以最大限度地提高 GPU 利用率和函数性能。

支持 GPU 的函数即服务面临的主要挑战，是在局部性与负载均衡之间取得平衡，以解决上述限制。从局部性的角度来看，支持 GPU 的函数即服务可以通过在已加载模型的同一块 GPU 上处理请求，降低函数延迟。然而，偏重局部性可能会增加请求的平均延迟，因为所有请求都会被转发到缓存了模型的 GPU，而其他 GPU 则处于空闲状态。从负载均衡的角度来看，支持 GPU 的函数即服务可以将请求均匀分配到各块 GPU，从而提高 GPU 利用率。然而，在处理工作集较大的工作负载时，负载均衡可能会增加缓存未命中，因为 GPU 上缓存的模型无法得到充分复用，并且常常会被新到达的请求从有限的内存空间中逐出。

本文介绍了一些互补组件，使现有的开源函数即服务平台能够利用 GPU

资源，并提升以 FaaS 函数形式运行的机器学习推理性能。分布式*GPU* 管理器代表 FaaS 函数处理 GPU 资源，从而将 GPU 资源管理与 FaaS 平台解耦；每个 GPU 管理器负责管理调度到某个 GPU 的请求，并估算该 GPU 上排队请求的完成时间。全局缓存管理器 将各 GPU 内存中已上传的推理模型视为缓存项；它遵循 LRU 替换策略，将局部性高的模型保留在 GPU 内存中。全局调度器 利用 GPU 管理器和缓存管理器提供的预计完成时间及 LRU 列表，将 FaaS 函数调度并分派到 GPU。

所提出的局部性感知负载均衡（*LALB*）调度器通过将函数工作负载均衡地分配到 GPU，并在处理请求时提高缓存模型的复用率，从而提升函数性能和 GPU 利用率。具体而言，调度器始终优先处理模型已缓存在空闲 GPU 上的请求，并可按乱序调度这些请求，以提高缓存命中率。如果缓存了模型的 GPU 正处于繁忙状态，调度器会利用该繁忙 GPU 的预计完成时间，判断繁忙 GPU 上的缓存命中是否比空闲 GPU 上的缓存未命中具有更短的预计完成时间。只有当繁忙 GPU 上的缓存命中无法提供更短的完成时间时，调度器才会将缓存未命中请求转发到空闲 GPU。

我们使用公有云提供商公开的真实世界跟踪数据以及生产环境中广泛使用的推理模型，对所提出的支持 GPU 的函数即服务进行了性能评估。结果表明，所提出的系统能够让模型推理函数在共享 GPU 资源上运行，并显著提升性能。具体而言，采用局部性感知负载均衡后，与基线（默认负载均衡）调度器相比，所提出的调度器将平均延迟和缓存未命中率分别降低了 80% 和 65%。采用乱序调度后，调度器进一步将平均延迟和缓存未命中率分别降低了 97% 和 81%。

本文其余部分组织如下：第二节介绍背景和相关工作；第三节描述所提出的支持 GPU 的 FaaS 系统的主要组件；第四节阐述调度策略；第五节展示性能评估；第六节讨论系统的其他方面；第七节

总结全文。

II. 背景与相关工作

A. 函数即服务

函数即服务（FaaS）正逐渐成为一种广受欢迎的云计算服务模式，可为事件驱动应用程序带来前所未有的可扩展性和成本效益。传统云服务为用户提供虚拟机或容器实例，以便部署和运行完整的应用程序；与之不同，函数即服务为用户提供平台，用于部署和运行构成应用程序的各个函数。通过让应用程序按函数粒度扩展，函数即服务使其具备更强的可扩展性——不同函数可以

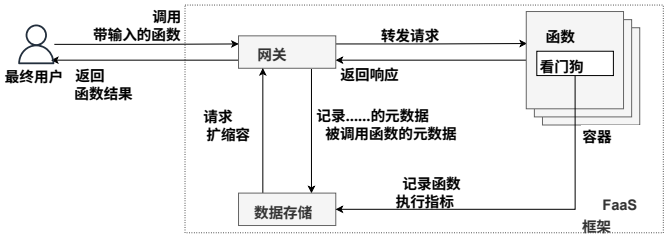


图 1：通用函数即服务架构。它包含三个主要组件：网关为用户提供部署和调用函数的接口，看门狗负责监控和执行函数，数据存储负责存储函数日志和指标。

独立扩展，且更具成本效益——云服务提供商只分配函数所需的资源，用户也只需为这些资源付费，相较于传统云服务更为经济。

图 1 展示了典型函数即服务框架的主要组件，包括网关、看门狗和数据存储。这些组件运行在 Docker Swarm [6] 和 Kubernetes [15] 等容器编排系统之上。网关是面向公众的入口，负责处理函数的创建、读取、更新和删除（CRUD）操作，并调用已注册的函数，以此与终端用户交互。看门狗在后台与容器中的函数代码一同运行，用于启动并监控容器中的函数。看门狗接收来自网关的调用请求，使用给定的输入执行函数，将函数的响应返回给网关，并将函数调用的状态和指标（例如执行延迟）存储到数据存储中。数据存储保存被调用函数的日志和指标。当函数需求动态变化时，还可以将数据存储配置为通过网关触发函数扩缩容操作。

函数即服务的终端用户无需配置任何资源或安装运行函数所需的依赖项，即可编写函数。终端用户可以使用特定 FaaS 平台提供的代码模板来部署函数即服务函数。终端用户将函数代码和模板提交给 FaaS 平台后，平台会创建一个运行中的容器，并安装模板中指定的必要资源，以构建该函数。部署后的函数会注册为 RESTful API，终端用户可以通过发送 HTTP 请求，或在其他函数或服务中实现触发器来调用该函数。

函数即服务范式的优势促使其应用于其他计算领域。例如，funX [4] 是一个基于函数的科学计算平台，可在联邦计算资源上灵活、高性能地执行函数；函数也被探索用作在异构边缘资源上进行计算的抽象 [10]。

由于现有的函数即服务系统通常不支持 GPU，函数调度主要根据 CPU 和内存来考虑其资源需求。例如，OpenFaaS 依赖 Kubernetes 等底层容器编排系统，将函数的容器排队，并调度到能够满足其资源需求的节点上。

B. GPU 计算

图形处理器（GPU）采用了不同于传统处理器的设计，配备数千个简单核心和高带宽内存架构，可为机器学习等应用提供强大的并行性。GPU 拥有多个流式多处理器（SM），其中包含计算核心、共享缓存和共享内存。GPU 是一种通过 PCI Express（PCIe）与主机通信的外部设备，应用要使用 GPU，就需要在二者之间传输数据。

GPU 有三项限制，使多个应用难以共享同一块 GPU。首先，GPU 内存容量远小于主机内存。如果待处理数据的大小超过可用的 GPU 内存容量，GPU 程序员就需要负责管理 GPU 内存中的活动工作集。其次，尽管 GPU 能够快速计算，但在传输数据（例如模型参数和输入）时会产生额外开销，需要将数据从主机内存传输到 GPU 内存。GPU 的数据传输开销源于 PCIe 接口：当前 PCIe 的最大带宽远低于 GPU 的内部内存带宽（前者约为 100 GB/s，后者约为 1 TB/s）。

为了解决上述限制，必须构建 GPU 调度器，在各块 GPU 之间平衡请求，并利用已加载到 GPU 内存中的模型来处理请求。

C. 深度学习

基于深度神经网络（DNN）的深度学习应用，是解决许多重要任务的关键方案，例如语音识别、自然语言处理、图像分类和目标检测。DNN 的架构以加权有向图表示，其中神经元被分组成多个层，每个神经元之间的连接彼此通信。

深度学习的主要任务包括训练模型和使用模型进行推理。训练过程通过执行前向传播和反向传播，迭代更新每一层的权重，使其逐步接近目标。推理则使用各层更新后的权重，通过前向传播根据输入进行预测。在图像分类中，训练会在反向传播期间更新模型权重，以缩小前向传播结果与真实标签之间的差异。推理则通过将输入数据转换为表示类别的数值结果，预测输入图像的图像类别。

用户通常使用 TensorFlow [1] 和 PyTorch [21], 等框架开发和执行机器学习任务。这些框架提供便捷的高级应用程序接口，并自动利用可用的处理器（包括 GPU）并行执行模型训练和推理。

GPU 在模型训练和推理方面都具有出色的并行性。例如，卷积神经网络中的卷积运算需要配置一个固定大小的滤波器，以生成特征表。滤波器通过遍历整个图像文件执行重复计算，而这些重复计算为 GPU 发挥并行性提供了机会。较大的输入批次大小也能让更多数据通过神经网络传播，从而使更多输入能够并行处理。

许多研究都探讨了基于 GPU 的机器学习模型训练。例如，在近期的研究中，CROSSBOW [13] 是一种新型单服务器多 GPU 系统，用于训练深度学习模型，并允许用户自由选择偏好的批量大小；AntMan [28] 则将集群调度器与深度学习框架协同设计，以便在大规模 GPU 集群上调度模型训练作业。

本文聚焦于运行机器学习推理的 FaaS 函数，因为推理任务的特性非常适合 FaaS 平台充分发挥其可扩展性。推理任务按需触发（通常由输入驱动），且运行时间相对较短，通常以秒计；相比之下，训练任务耗时要长得多（需要数小时或数天），而且往往不是由事件驱动的（训练数据通常不会频繁变化）。

TensorFlow Serving [24]、TorchServe [25], 和 NVIDIA Triton [26] 等先进的模型服务系统，使用户能够简化在单台服务器或服务器集群上部署和执行模型推理任务的流程。这类系统为用户提供接口，用于将模型部署到系统中，并利用 GPU 等资源进行推理。它们采用调度器或负载均衡器，将推理任务分配到可用资源上。然而，这些系统并非基于函数即服务范式——函数即服务能为模型服务带来独特优势，而它们也未考虑 GPU 缓存局部性；局部性对推理性能至关重要。本文的目标并非开发新的模型服务系统；相反，我们提出的基于函数的模型服务方案和感知缓存局部性的调度方法，可以由现有及未来的模型服务系统采用，以提升其易用性和效率。

D. 支持 *GPU* 的函数即服务

使用机器学习推理的 FaaS 函数需要大量计算资源并进行并行化；因此，将 GPU 引入 FaaS 对于提升推理任务的性能和可扩展性至关重要。

Kim 等人 [11] 通过让 FaaS 平台中的容器直接访问 GPU，提出了一个支持 GPU 的 FaaS 平台。尽管这项工作展示了支持 GPU 的函数所带来的优势，但这些函数使用的 NVIDIA Container Toolkit [19] 限制了多个容器共享单个 GPU。占用 GPU 的函数可能会运行非 GPU 任务，例如预处理输入图像，同时阻止其他等待中的函数使用 GPU。我们的解决方案通过让函数共享 GPU，并为函数提供经过优化的 GPU 资源管理，解决了 GPU 垄断问题。

Naranjo 等人 [18] 通过将 rCUDA（一种 GPU 虚拟化框架）[7],引入函数即服务，解决了 GPU 垄断问题。该解决方案通过拦截 FaaS 函数发出的 GPU 操作，

将其转发至 rCUDA 接口，从而阻止 FaaS 函数直接管理 GPU。它依赖 rCUDA 将函数即服务与 GPU 资源管理解耦；因此，函数即服务平台与 rCUDA 无法协调以提升性能，因为二者不共享待处理的 FaaS 请求和 GPU 利用率信息。相比之下，我们的解决方案将 GPU 资源管理集成到函数即服务框架中，并支持协同进行 GPU 函数调度和内存管理，从而提升函数性能和 GPU 利用率。

Satzke等人 [22]在 Knative [12], 之上实现了 GPU 共享功能。Knative 是一个开源框架，为在 Kubernetes 中管理 FaaS 函数提供了专门功能。该方案将单个 GPU 的内存转换为多个虚拟GPU，并提供约束策略，防止 FaaS 函数过度占用 GPU 内存。然而，该方案侧重于避免因 GPU 内存过度占用导致的内存溢出（OOM）错误，却未能解决多个函数共享同一 GPU 所增加的开销。我们的方案通过协同设计 GPU缓存管理和函数调度，既防止 GPU 内存过度占用，又解决资源共享开销。

Dakkak等人 [5] 通过实现一个守护进程来引入 GPU 缓存：该进程拦截函数的 CUDA 请求，并为其分配 GPU 内存。该方案采用统一内存和 CUDA进程间通信等 GPU内存虚拟化服务，在 GPU 内存和进程之间共享已上传的模型。该方案通过限制 GPU 内存用量来避免 GPU 内存溢出（OOM），并使用最近最少使用等基本缓存机制来降低延迟。然而，该方案没有考虑缓存局部性对 GPU 调度的影响，而且总是将请求转发给利用率最低的 GPU，导致性能不佳。相比之下，我们的方案提出了缓存局部性感知调度，以解决 GPU负载均衡与缓存局部性之间的冲突，从而显著提升性能。

虽然 Kube- Share [29] 并非专为函数即服务设计，但它通过将 GPU 视为一等资源，并允许容器对其进行调度和分时共享，改进了容器框架（Kubernetes）中的 GPU 管理；AWS 的 Kubernetes 虚拟GPU设备插件 [2]则允许容器使用 CUDA多进程服务（MPS） [17]对 GPU 进行空间共享。

上述解决方案所解决的问题与本文关注的协同 GPU 内存管理和函数调度问题互为补充。一方面，正如这些相关工作所展示的，GPU 共享使用日益增多，这进一步凸显了在并发工作负载之间有效分配有限 GPU 内存的重要性。对于模型推理任务而言尤其如此，因为模型加载延迟会对其产生显著影响。另一方面，这些相关解决方案都能从我们提出的 GPU缓存局部性感知函数调度中获益。本文其余部分将详细介绍我们解决方案的设计、实现和评估结果。

III. 设计

A. 架构

我们提出的支持 GPU 的函数即服务旨在通过优化 GPU 调度和资源管理，提升 GPU 函数的性能，尤其是模型推理函数的性能。图 2 展示了我们框架的完整架构，其中包含三个额外组件（调度器、缓存管理器、GPU 管理器），在图中以阴影框表示。这些组件基于现有的函数即服务框架（如 OpenFaaS，图中以无阴影框表示）启用并优化 GPU 函数。

这个支持 GPU 的函数即服务框架只需对网关进行少量修改。最终用户在通过网关注册函数时，可以在该函数的 Dockerfile 中加入 GPU 启用标志。网关会检查 Dockerfile 中的 GPU 启用标志，并将函数用于加载和运行模型的接口替换为自定义接口，由该接口将这些请求转发给 GPU 管理器。这一接口变更对最终用户不可见。函数中需要替换的接口很容易找到，因为用户通常会使用常见的机器学习框架进行模型推理，例如 PyTorch 中的 torch.load()、model(input)，以及 TensorFlow 中的 model.load weights()、model.predict(input)。这些应用程序接口数量有限，并且在不同版本的框架之间相对稳定。

B. 调度器

调度器决定将函数请求调度到何处（哪些节点和哪些 GPU ），以供整个函数即服务系统使用。调度器遵循特定的调度策略，该策略可在首次启动调度器组件时启用。我们将在第四节详细讨论这些策略。调度器确定需要调度某个请求后，会将函数信息与 GPU 地址打包，并转发给函数容器。函数请求包含输入数据，以及使用预训练模型进行推理的已注册函数 ID。GPU 地址包含安装了 GPU 的服务器 IP 地址，以及用于访问该服务器上 GPU 的设备名称。

图 3 展示了调度器如何根据调度策略以及 GPU 管理器和缓存管理器收集的 GPU 信息，将网关转发的请求调度到 GPU。调度器维护两种队列：系统范围的全局队列和每个 GPU 的本地队列。全局队列面向整个函数即服务系统，用于存储网关转发给调度器的所有请求。全局队列中的待处理请求按其到达时间排序。调度器根据调度策略，将全局队列中的待处理请求调度到 GPU。每个 GPU 都有一个本地队列，用于存储调度器已调度到该 GPU、但由于 GPU 当前正忙而无法立即执行的请求。

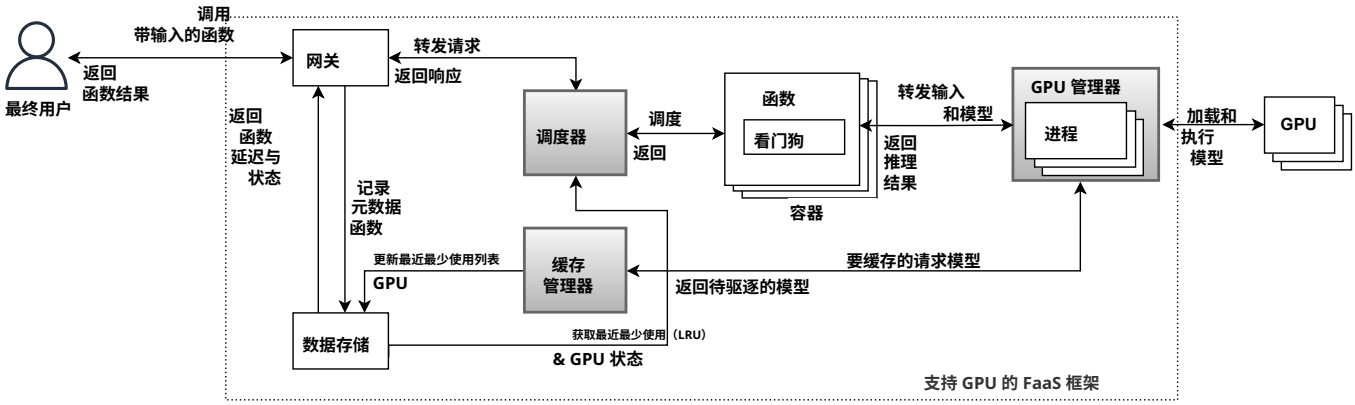


图 2：所提出的支持 GPU 的函数即服务架构。未着色的方框是现有函数即服务架构中的组件（仅做了少量修改）。着色的方框是为 GPU 管理、缓存管理和局部性感知调度而新增的组件。

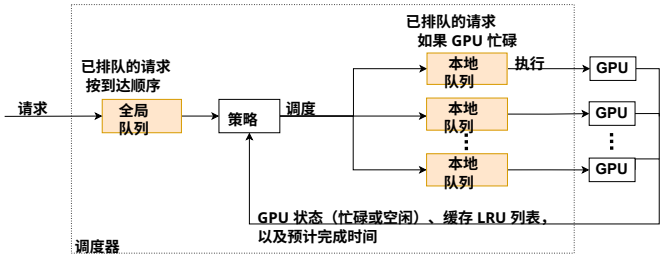


图 3：调度器架构。它维护系统范围的全局队列和每个 GPU 的本地队列，以根据给定策略和 GPU 状态调度请求。

C. GPU 管理器

GPU 管理器运行在每个 GPU 节点上，并管理该节点上运行的 GPU 进程。GPU 管理器代表函数运行 PyTorch 进程：它从定制的 PyTorch API 接收推理请求，并将结果返回给该 API。每个 GPU 进程在启动时都会上传一个推理模型；推理请求完成后，会向数据存储报告延迟。当 GPU 进程正在上传模型或处理推理请求时，GPU 管理器会向数据存储报告 GPU 状态为忙碌；GPU 进程完成任务后，则会将状态更新为空闲。GPU 管理器强制每个 GPU 一次只运行一个请求，并在 GPU 正在处理请求时将其状态设为忙碌。

GPU 管理器与缓存管理器通信，将正在运行的 GPU 进程所使用的模型作为缓存项进行维护。收到函数调用请求后，GPU 管理器会询问缓存管理器，请其判断请求的模型是否已缓存在 GPU 内存中（即缓存命中）或尚未缓存（即缓存未命中）。如果缓存未命中，GPU 管理器还会检查是否有待淘汰模型需要清除，以便为新请求所需的模型腾出空间。如果存在待淘汰模型，GPU 管理器便会终止与被淘汰模型关联的进程。随后，GPU 管理器会为新模型启动一个 GPU 进程，并将模型上传到 GPU 上运行。如果是缓存……

命中时，无需进行淘汰，并且使用所请求模型的 GPU 进程已在运行。随后，GPU 管理器会将新请求的输入转发给此现有 GPU 进程以运行推理。推理完成后，GPU 进程会返回结果，GPU 管理器再将结果返回给调度器。GPU 管理器还会根据收到的推理请求对缓存模型的 LRU 列表进行更新。

D. 缓存管理器

缓存管理器作为全局组件与调度器协同运行，并主要遵循 LRU 替换策略，管理系统中每个 GPU 内存中的模型。缓存项是由 GPU 进程上传到 GPU 内存的模型。当函数准备使用 GPU 时，会请求缓存管理器返回使用所需推理模型的 GPU 进程。如果 GPU 管理器中存在所需的 GPU 进程，则表示缓存命中，因为该 GPU 进程使用的模型已位于 GPU 内存中，函数可以使用现有的 GPU 进程并跳过模型传输。否则，表示缓存未命中，缓存管理器会请求 GPU 管理器新建一个 GPU 进程，以便将新的推理模型上传到 GPU 内存。

发生缓存未命中时，缓存管理器会收到来自 GPU 管理器的请求，其中包含 GPU 的可用内存空间以及缺失于 GPU 内存中的模型 ID。根据这些参数，缓存管理器随后确定需要从 GPU 中淘汰的待淘汰模型列表，以腾出足够空间存储新模型，并按照 GPU 的 LRU 列表选择要淘汰的模型。

E. 数据存储

数据存储保存每个推理请求的预估延迟、每个 GPU 的 LRU 列表以及每个 GPU 的状态。我们使用容器编排系统（例如 Kubernetes ）已经在使用的 Etcd [8], 来实现数据存储。

Etcd 是一种分布式键值存储，可在分布式环境中保证高度一致性。缓存管理器和 GPU 管理器会更新数据存储中的上述信息，而调度器则利用这些信息来决定将请求调度到哪些最优 GPU 上。

IV. 调度策略

FaaS 系统中的默认调度器主要考虑 CPU 和内存资源的负载均衡。在这项工作中，我们提出了一种局部性感知与负载均衡（LALB）调度器，用于应对在 GPU 上调度模型推理函数时面临的独特挑战。除了利用负载均衡功能提高 GPU 利用率外，该调度器还将上传到 GPU 的模型视为缓存项，结合局部性感知功能，以提升推理函数的性能和 GPU 的利用率。一方面，LALB 调度器的局部性感知功能会将请求转发到缓存了相应模型的 GPU，从而避免上传模型的开销，降低执行函数的延迟。另一方面，LALB 调度器也会考虑负载均衡；即使函数所需的模型缓存在繁忙的 GPU 上，只要在空闲 GPU 上执行更快，也允许发生缓存未命中。这种设计使热门模型能够自然地复制到多个 GPU 上，以满足使用这些模型进行推理的需求；随着工作负载中的热门程度随时间变化，这些模型也能够自然地被逐出。

A. 局部性感知负载均衡（*LALB*）

只有在全局队列中至少有一个请求等待，且至少有一块 GPU 空闲时，才会调用局部性感知负载均衡（LALB）调度器。算法 1 和算法 2 说明了调度器如何同时考虑 GPU 的负载均衡情况以及模型在 GPU 内存中的局部性。

首先，LALB 调度器从全局队列的队首获取请求，并检查是否有可用的空闲 GPU 能够命中缓存，即其内存中已存储所请求的模型。如果请求能够命中缓存，就将其调度到缓存了该模型的某一块空闲 GPU 上（算法 1 第 8 行）。LALB 调度器确认空闲 GPU 上没有可用的缓存命中后，会在繁忙 GPU 上查找是否可能命中缓存。

如果繁忙 GPU 上发生缓存命中，LALB 调度器会比较该请求在这块繁忙 GPU 上缓存命中时的预计完成时间，与在一块空闲 GPU 上缓存未命中时的预计完成时间。前者包括等待繁忙 GPU 完成当前请求（以及其本地队列中已排队的请求）的时间，以及新请求的推理时间。后者包括将所请求的模型上传到一块空闲 GPU 并执行推理的时间。如果在繁忙 GPU 上缓存命中的预计完成时间短于在空闲 GPU 上缓存未命中的预计完成时间，则会将该请求调度到繁忙 GPU 并移入其本地队列（算法 2 第 12 行）。该 GPU 空闲后，会始终先执行其本地队列中已有的请求，再考虑全局队列中的请求。另一方面，如果没有任何缓存了该模型的 GPU 能够比空闲 GPU 更早完成请求，LALB 调度器会将该请求调度到一块空闲 GPU 上，从而造成缓存未命中（算法 2 第 17 行）。

通过在系统中的 GPU 上对每个唯一模型进行性能分析，收集模型上传和运行推理的延迟。表 I 列出了评估中使用的模型的上传时间和推理时间。上传时间仅取决于模型大小；推理时间取决于模型和批量大小，可通过简单的回归方法进行性能分析。

B. 乱序调度

LALB 调度器还采用乱序（O3）调度策略，以进一步促进 GPU 上模型的复用。它会优先将能够在空闲 GPU 上命中缓存的等待请求调度到该空闲 GPU，即使该请求在全局队列中排在其他已排队请求之后（算法 1 第 6–10 行）。默认情况下，LALB 调度器按照请求到达的顺序调度全局队列中的请求。按序调度可确保请求之间的公平性，但可能导致性能欠佳，因为已调度的请求可能会逐出后续请求很快就需要的模型。当工作负载的工作集（不同模型的总数）较大时，这类情况的负面影响尤其严重，并可能导致抖动。乱序调度会牺牲一定程度的公平性，以换取显著提升的性能。为避免饥饿，它设定了一个限制值（默认为 25），以防等待中的请求长期得不到调度（算法 1 第 11–14 行）。如果队列中的某个请求被跳过的次数达到该限制值，调度器就会立即调度该请求，不论此举会导致缓存命中还是缓存未命中。

V. 评估

A. 方法

我们使用真实世界的机器学习推理工作负载，评估所提出的支持 GPU 的 FaaS 系统的性能。该原型基于 OpenFaaS 实现。我们采用三个指标评估所提出的支持 GPU 的 FaaS 系统：平均延迟、缓存未命中率和 GPU（SM）利用率。为展示所提出的感知缓存局部性的调度器带来的改进，我们将其性能与默认负载均衡调度器进行比较；后者仅在 GPU 空闲时调度全局队列队首的请求。

1) 工作负载：表 I 展示了工作负载中考虑的 22 种热门卷积神经网络模型。表中列出了这些模型的实际大小，以及以固定批量大小 32 运行模型推理时在 GPU 内存中的占用大小。缓存管理器使用这一峰值内存占用大小来决定缓存替换；如果超过可用内存，GPU 将发生内存溢出。

我们使用微软 Azure 函数跟踪 [23] 评估调度器的性能。选择这一跟踪，是因为它代表了由现实世界中的大型云服务提供商微软提供的 FaaS 函数工作负载

算法 1：LALB：局部性感知负载均衡 输入：空闲 GPU 列表 $idle\ GPUs$（按频率排序） 繁忙 GPU 列表 $busy\ GPUs$ 全局队列中的待处理请求列表 $requests$（按到达时间排序） GPU 的本地队列 1 对 GPU_i 中的每个 $idle\ GPUs$ {/* 优先处理已调度到 GPU_i 的本地队列中的请求 */ 2 如果 GPU_i 的本地队列不为空 {3 将本地队列队首的请求调度到 GPU_i4 继续5 }/* 查找全局队列中模型已缓存在 GPU_i 上的请求 */ 6 对 Req_s 中的每个 $requests$ {7 如果 Req_s的模型缓存在 GPU_i {中 8 将 Req_s 调度到 GPU_i9 跳出循环10 }/* 执行乱序调度限制（若禁用乱序分派，则将限制设为 0） */ 11 如果 Req_s 的访问次数高于指定限制 {12 $flag$ =LocalityLoadBalance(GPU_i,$idle\ GPUs$, $busy\ GPUs$, Req_s) 13 如果 $flag$ 为真，则跳出循环；否则继续14 } 否则 {15 增加 Req_s的访问次数16 }17 }否则 {/* 全局队列中没有模型缓存在 GPU_i 上的请求 */ 18 对 Req_s 中的每个 $requests$ {19 $flag$ = LocalityLoadBalance(GPU_i,$idle\ GPUs$, $busy\ GPUs$, Req_s) 20 如果 $flag$ 为真，则跳出循环；否则继续 21 }22 }23 }

**算法 2：函数 LocalityLoadBalance 输入：选定的空闲 GPU GPU_i空闲 GPU 列表 $idle\ GPUs$繁忙 GPU 列表 $busy\ GPUs$ 选定的请求 Req_s 输出：一个布尔值，表示 Req_s 是否被调度到 GPU_i （True），否则为 False /* 如果 Req_s 的模型未缓存在任何空闲或繁忙 GPU 上，则允许缓存未命中 */ 1 如果 Req_s的模型未缓存在任何其他 GPU 上 {2 将 Req_s 调度到 GPU_i3 返回 True4 } 否则，如果 Req_s的模型缓存在另一台空闲 GPU 上GPU_j {5 将 Req_s 调度到 GPU_j 6 返回 False7 } 否则 {/* Req_s的模型缓存在某台繁忙 GPU 上 */ 8 对于 GPU_j 中的每个 $busy\ GPUs$ {9 如果 GPU_j 缓存了 Req_s的模型 {10 估算GPU_j上请求的完成时间 11 如果完成时间小于 Req_s的模型加载时间 { 12 将 Req_s 移至 GPU_j的本地队列 13 返回 False 14 }15 }16 } 否则 {/* 允许缓存未命中 */ 17 将 Req_s 调度到 GPU_i18 返回 True 19 }20 }

模型	大小（MB）	加载时间（秒）	推理时间（秒）
squeezenet1.1	1269	2.41	1.28
resnet18	1313	2.52	1.25
resnet34	1357	2.60	1.25
squeezenet1.0	1435	2.32	1.33
AlexNet	1437	2.81	1.25
resnext50.32x4d	1555	2.64	1.29
DenseNet-121	1601	2.49	1.28
DenseNet-169	1631	2.56	1.30
DenseNet-201	1665	2.67	1.40
ResNet-50	1701	2.67	1.28
ResNet-101	1757	2.95	1.30
ResNet-152	1827	3.10	1.31
DenseNet-161	1919	2.75	1.32
Inception-v3	2157	4.42	1.63
ResNeXt-101-32x8d	2191	3.51	1.33
VGG-11	2903	3.94	1.29
Wide_ResNet-50-2_	3611	3.16	1.31
Wide_ResNet-101-2_	3831	3.91	1.32
VGG-13	3887	3.98	1.30
VGG-16	3907	4.04	1.27
vgg16.bn	3907	4.03	1.26
vgg19	3947	4.07	1.33

表 I：评估中使用的模型在 GPU 上的占用大小、加载时间和推理延迟（批量大小为 32）

Azure。该跟踪包含 14 个文件，分别记录了 14 天的函数调用。每个文件提供一列，代表每分钟；一行，代表每个唯一函数；以及一个数值，表示该唯一函数每分钟的总调用次数。我们提取跟踪的前 6 分钟数据，并将每分钟的请求数归一化为 325 个，以匹配我们用于实验的规模小得多、包含 12 个 GPU 的测试平台。

Azure 跟踪中的唯一函数总数（工作集）为 46,413，这对我们的测试平台而言过于庞大，无法处理。同时，该跟踪代表的工作负载具有高度偏斜的工作集：最热门的 15 个函数合计占每分钟总调用次数的 56%，而排名前 15 之外的函数每个所占的每分钟总调用次数均不到 0.01 %。因此，我们仅将使用最频繁的函数作为工作负载中的工作集。我们采用三种不同的工作集大小：15、25 和 35。较大的工作集包含更多唯一函数，同时将每分钟请求数的最大值保持在 325 个请求。我们将跟踪中的每个唯一函数映射到表 I 中的一个唯一模型，并确保不同大小的模型在工作负载中均匀分布。在工作负载的每一分钟内，我们随机分配不同函数的调用，同时保持每分钟总调用次数的归一化值不变。

2）数据集：对于用于推理的输入图像，我们提供了一组包含 150 个图像文件的小型数据集，其中包括 CIFAR10 [14], 改良版美国国家标准与技术研究院（MNIST） [16], 和膜翅目昆虫 [21]等标准数据集。MNIST 数据集包含 28x28 的灰度图像，分为 60,000 张训练图像和 10,000 张验证图像。CIFAR-10 包含 32x32 的 RGB图像，其中有 50,000 张训练图像和 10,000 张验证图像。膜翅目昆虫数据集包含大小从 50KB 到 2MB 不等的 RGB图像，在用于模型推理前必须进行压缩。该数据集由 245 张训练图像和 153 张测试图像组成。

3）测试平台：所有实验均在三台 GPU 服务器上进行，每台服务器配备四块 GeForce RTX 2080 GPU，并将 GPU 管理器部署为可访问 GPU 资源的 Nvidia Docker 容器。另有一台独立服务器运行用于调度器、数据存储和缓存管理器的 Docker 容器，以及 OpenFaaS 平台所需的组件。每台服务器均配备双路 Intel Xeon 八核处理器和 128GB 动态随机存取存储器，彼此通过 100Gb/s InfiniBand 网络互连。数据存储在基于网络文件系统的共享文件系统上，后端存储为一块 6TB SATA硬盘。

B. 延迟和缓存未命中率

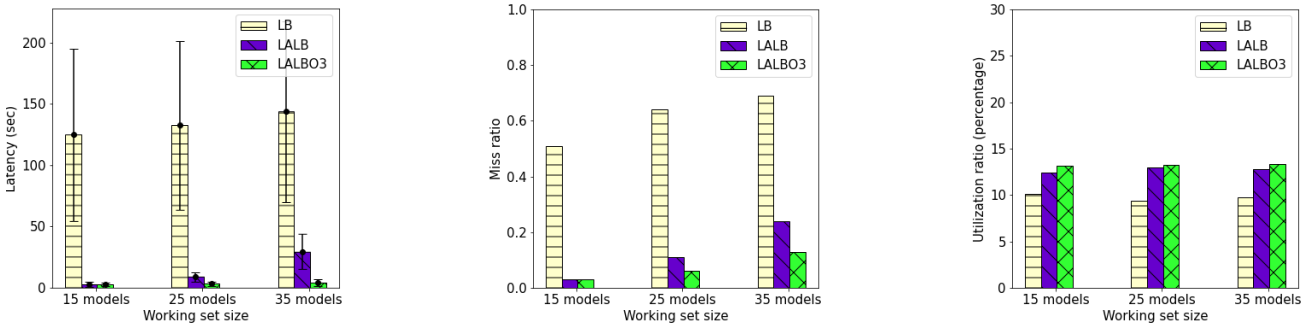
图 4a 展示了三种调度器的函数总延迟。结果表明，当工作集大小分别为 15 和 25 时，与默认负载均衡（LB）调度器相比，局部性感知负载均衡（LALB）调度器分别将平均延迟降低了 97.74% 和 93.33%。然而，随着工作集大小增加到 35，LALB 调度器的平均延迟和缓存未命中率均有所上升。结果显示，如图 4b 所示，工作集大小为 15 时，缓存未命中率降低了 94.11%；而工作集大小为 35 时，缓存未命中率降低了 65.21%。性能下降的原因在于，工作集大小进一步增大时，提升局部性会变得更具挑战性。

将乱序（O3）调度应用于 LALB 调度器后，在工作集大小为 25 和 35 时，性能进一步提升。O3 分派通过允许全局队列中的请求在能够产生缓存命中时乱序调度，提升缓存命中率。随着工作集大小增加并超出有限的 GPU 内存容量，降低缓存未命中率变得至关重要。工作集大小为 35 时，LALB 调度器将 LB 的缓存未命中率降低了 65.21%，而 LALBO3 调度器将其降低了 81.15%。

C. 利用率

图 4c 展示了使用不同调度器运行工作负载时所有 GPU 的平均流式多处理器利用率。由于每分钟请求数始终保持在 325，各调度器的流式多处理器利用率在所有三个工作集下均保持一致。流式多处理器利用率不可能达到 100%，因为 GPU 要容纳多个推理模型，而且分配过大的批量大小可能导致内存超限。

LALBO3 调度器的流式多处理器利用率最高，因为其缓存未命中率最低。流式多处理器利用率与缓存未命中率呈负相关，因为在模型上传到 GPU 内存之前，GPU 无法使用流式多处理器执行推理。发生缓存未命中时，流式多处理器利用率



(a) 平均函数延迟 (b) 缓存未命中率 (c) GPU（SM）利用率

图 4：所提出的调度器与默认负载均衡调度器在平均延迟、缓存未命中率和 GPU（SM）利用率方面的比较分析

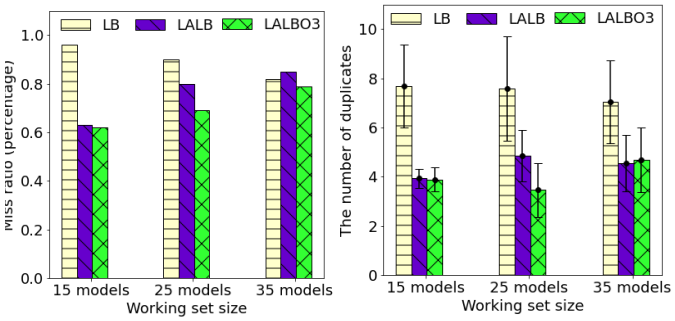


图 5：虚假未命中率 图 6：最热门模型的平均副本数

会一直保持为零，直到被替换的模型被逐出，并将新模型上传到 GPU。因此，LALBO3 调度器的流式多处理器利用率最高，因为其缓存未命中率最低。

D. 效率

本节使用虚假未命中率和重复热门模型的数量作为指标，说明各调度器的效率。系统中存在热门模型的副本，可以使这些模型的推理请求命中缓存；但副本过多也会污染缓存，降低整体缓存命中率。理想的调度器应在不降低缓存命中率的前提下，将 GPU 上重复模型的数量维持在最低水平。实验期间，我们跟踪同时缓存最热门模型的 GPU 总数，并据此统计副本的平均数量。在调度决策中，虚假未命中指这样一种缓存未命中情形：尽管请求的模型缓存在另一块 GPU 上，请求仍被转发到一块未缓存该模型的 GPU。虚假未命中率与重复模型的数量相关，因为虚假未命中决策会迫使未缓存请求模型的 GPU 存储其他 GPU 上已有的模型。

图5显示，对于工作集大小为15和25的情况，LALB 和 LALBO3 调度器都能降低虚假未命中率。默认的 LB 调度器自然具有最高的虚假未命中率（接近96%），因为它在调度请求时不考虑缓存局部性。与 LB 相比，当工作集大小为15时，LALB 和 LALBO3 分别将虚假未命中率降低了34.38%和35.41%。可用的 GPU 内存能够为较小的工作集找出重复缓存项的最佳数量，从而在不进行 O3 分派的情况下提升局部性。当工作集大小增至35时，只有 LALBO3 调度器仍能将 LB 调度器的虚假未命中率降低3.65%，因为 LALBO3 通过将等待中的请求调度到缓存所在的 GPU 上进行 O3 分派，从而进一步利用局部性。

图6展示了最热门模型副本数量的平均值。由于支持 GPU 的函数即服务使用 12 块 GPU，同一模型的副本数量最多不能超过 12。由于 LB 调度器不考虑局部性，因此可能出现重复缓存项不断相互驱逐的情况。LALB 调度器通过谨慎地选择缓存了模型的 GPU 来处理请求，从而提升局部性；缓存命中增加也减少了系统中的重复缓存项。当工作集大小为 15 时，LALB 调度器将 LB 的平均副本数量降低了 48.96%，同时将缓存未命中率降低了 94.11 %（见图 4b）。工作集大小增加会削弱 LALB 调度器维持最佳副本数量的能力；当工作集大小为 35 时，LB 的副本数量降低了 35.32%。工作集越大，在 GPU 内存容量有限的情况下，LALB 就越难维持缓存局部性。

当工作集大小为 15 时，与 LALB 相比，LALBO3 调度器并未显著减少重复项数量，因为它将 LB 调度器的平均重复项数量降低了 49.48%。由于可用 GPU 内存足以容纳大部分工作集，因此 O3 分派带来的性能提升微乎其微。当工作集大小为 35 时，LALBO3 调度器的表现优于 LALB 调度器，它将 LB 调度器的平均重复项数量降低了 33.47%。结果表明，应用 O3 分派可以进一步提升局部性性能，从而减少平均重复项数量。

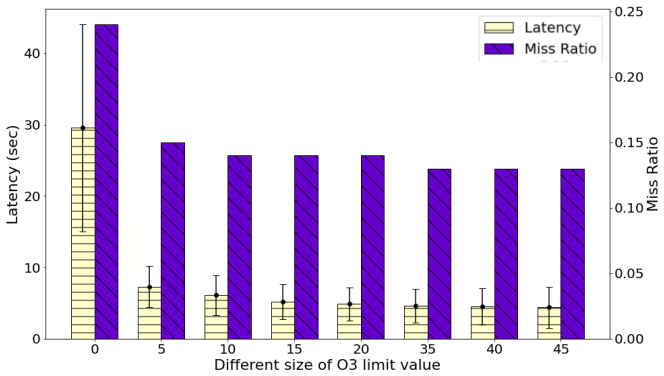


图 7：乱序（O3）调度采用不同限制值时的平均函数延迟和缓存未命中率

E. O3 敏感性测试

图 7 重点研究了 LALBO3 调度器中 O3 分派所用阈值的敏感性。该阈值决定了队列中的请求最多可以被跳过多少次，以便优先处理后续请求，因为空闲 GPU 中缓存了该请求所需的模型。我们针对工作集大小为 35 的工作负载进行实验，并将 O3 分派的指定阈值从 0 调整到 45（横轴）。阈值设为 0 时，LALBO3 退化为 LALB；而较高的阈值则可能导致请求调度时的公平性更差。我们使用平均函数延迟（左纵轴）和缓存未命中率（右纵轴）来评估不同阈值所带来的性能变化。

结果表明，随着指定的 O3 限制值增加，延迟和缓存未命中率都会降低。与 O3 限制值为 0 时相比，O3 限制值为 45 时，平均延迟和缓存未命中率分别降低了 85.1% 和 45.83%。较大的 O3 值能够提升性能，因为在调度请求时，它会优先考虑缓存命中，而非请求到达顺序。有趣的是，O3 限制值为 45 时，平均延迟方差并未增加，反而较限制值为 0 时降低了 95.93%。一方面，较大的限制值会导致某些请求不公平地等待更久；另一方面，较小的限制值会导致更多缓存未命中。结果表明，后者对延迟方差的影响占主导，因此与限制值为零（即 LALB）相比，较大的限制值显著降低了延迟方差。

VI. 讨论

开销与可扩展性。GPU 管理器采用分布式部署，每个 GPU 节点部署一个，因此能够随节点数量扩展。在每个节点上，GPU 管理器可以通过使用多个 GPU 进程来处理并发模型，从而随模型数量扩展；每个模型由一个 GPU 进程处理，以服务本地 GPU 需要提供服务的模型。

缓存管理器是全局的，并通过分别管理每个 GPU 的内存（使用独立的 LRU 列表）来确保其可扩展性。缓存管理器可以使用多个线程，并行处理来自多个 GPU 管理器的并发请求。

调度器是全局的，它运行局部性感知负载均衡（LALB）算法，为模型推理函数做出调度决策。当 GPU 空闲时，调度器会在全局队列中查找其模型已缓存在该 GPU 上的请求。为了降低这一搜索开销，调度器维护一种辅助数据结构，将排队的请求与其对应的模型关联起来——关联到同一模型的请求仍按到达顺序排列。借助这一数据结构，搜索的复杂度可限制在该 GPU 上缓存的模型数量范围内。

当请求需要调度时，调度器会检查所有缓存了该请求所需模型的 GPU——如果找到空闲 GPU，就会立即将请求调度到该 GPU；如果找到一个繁忙的 GPU，且它能够比空闲 GPU 更早地处理该请求，则将请求移至该繁忙 GPU 的本地队列。为便于进行此项搜索，缓存管理器会维护缓存了各个模型的 GPU 列表，并通过数据存储（Etcd）与调度器共享这些信息。因此，此项搜索的复杂度受限于缓存了该模型的 GPU 数量。

多租户与安全：我们的解决方案利用底层函数即服务框架来支持多租户。例如，OpenFaaS Pro 支持多个命名空间 [20]，结合其安全功能，可以对属于不同租户的函数组进行逻辑隔离。

由于我们的解决方案使来自不同租户的模型推理函数能够共享 GPU，因此我们还需要隔离 GPU 资源。恶意用户可能会尝试通过提交大量推理请求使系统过载；我们的系统可以通过限制每个租户可使用的 GPU 进程数量来应对这一问题并提供隔离。恶意用户也可能试图通过设计高局部性工作负载来操纵系统，垄断 GPU 时间和内存；我们的系统可以通过限制租户可使用的 GPU 时间份额和内存空间份额来应对这一问题并提供隔离。

GPU 异构性：我们的解决方案天然支持在模型推理函数中使用异构 GPU。只需针对系统中每种不同类型的 GPU，运行第四节 IV-A 中所述的相同性能分析流程，并在所提出的调度算法中使用经性能分析得到的模型加载和推理时间即可。

缓存替换策略：我们当前的实现采用最近最少使用策略来管理每个 GPU 上缓存模型的替换。通过将最近最少使用列表替换为其他类型的排序列表，我们的系统设计可以轻松支持其他缓存替换策略。不过，无论采用何种策略，我们提出的局部性感知调度都能通过促进缓存模型的复用、提高缓存命中率，同时维持负载均衡来提升性能。

易用性。我们的解决方案基于常用的函数即服务和机器学习框架构建，用户无需更改代码。所做的修改

对函数即服务框架和用户 Dockerfile 文件的改动很小，而且对用户完全透明。同时，我们提出的面向缓存局部性的模型推理任务负载均衡技术，以及调度器、GPU 管理器和缓存管理器的实现，也可应用于其他模型服务系统（例如 TensorFlow Serving [24], TorchServe [25]），以提升其基于 GPU 的模型推理性能。

VII. 结论

随着机器学习推理任务的应用场景不断增多，对支持 GPU 的函数即服务的需求也日益增长，因为 GPU 加速以及基于函数的部署和执行能够显著提升这类任务的性能。我们的解决方案专注于改进运行机器学习推理任务（如卷积神经网络）的 FaaS 函数，这些任务能够显著受益于 GPU 加速。然而，现有函数即服务框架对 FaaS 函数共享 GPU 资源的支持有限，而且 FaaS 函数的短生命周期使得难以克服在 GPU 上进行模型推理所产生的数据传输成本。

我们提出的解决方案可应用于不同的函数即服务框架，只需添加少量配套组件，即可引入 GPU 调度和缓存管理。我们的支持 GPU 的函数即服务能够全局管理 GPU 内存，并将上传到 GPU 中的推理模型视为缓存项，以减少数据传输开销。我们的缓存局部性感知调度器兼顾 GPU 缓存局部性和负载均衡，从而提升模型推理函数的性能。

我们使用真实世界的函数即服务跟踪数据以及生产环境中广泛使用的机器学习模型，评估了支持 GPU 的函数即服务解决方案的性能。结果表明，该方案能够显著提升模型推理性能。例如，对于工作集大小为 35 的工作负载，LALB 调度器将基线（LB）调度器的平均延迟和缓存未命中率分别降低了 79.43% 和 65.21%。此外，乱序（O3）调度可与 LALB 调度器协同工作，进一步提升局部性性能，并将 LB 调度器的平均延迟和缓存未命中率分别降低 96.93% 和 81.16%。

参考文献

[1] Mart´ın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin 等。Tensorflow：异构分布式系统上的大规模机器学习。*arXiv* 预印本 *arXiv:1603.04467*，2016 年。[2] 适用于 Kubernetes 推理工作负载的虚拟 GPU 设备插件。https://aws.amazon.com/blogs/opensource/virtual-gpu-device-plugin-for-inference-workload-in-kubernetes/。[3] Poornima Chand。使用 AWS 无服务器技术实现大规模机器学习推理。亚马逊博客，2021 年 11 月。[4] Ryan Chard、Yadu N. Babuji、Zhuozhao Li、Tyler J. Skluzacek、Anna Woodard、Ben Blaiszik、Ian T. Foster 和 Kyle Chard。funcX：面向科学的联邦式函数服务平台。*CoRR*，abs/2005 04215，2020 年。[5] Abdul Dakkak、Cheng Li、Simon Garcia de Gonzalo、Jinjun Xiong 和 Wen-mei Hwu。Trims：用于函数即服务中低延迟深度学习推理的透明隔离模型共享。载于*2019* 年 *IEEE* 第 *12* 届云计算国际会议（*CLOUD*），第 372–382 页，2019 年。[6] Swarm 模式概述。https://docs.docker.com/engine/swarm/。

[7] Jos´e Duato、Antonio J. Pe˜na、Federico Silla、Rafael Mayo 和 En- rique S. Quintana-Ort´ı。rCUDA：减少高性能集群中 GPU 加速器的数量。载于 2010 年国际高性能计算与仿真会议，第 224–231 页，2010 年。[8] Etcd：为分布式系统中最关键的数据提供分布式、可靠的键值存储。2021 年。[9] Anirudh Garg。为什么使用 Azure 函数进行机器学习推理？微软博客，2020 年 5 月。[10] Runyu Jin 、 Qirui Yang 和 Ming Zhao。函数即服务适合边缘计算吗？USENIX 协会，2020 年 6 月。[11] Jaewook Kim、Tae Joon Jun、Daeyoun Kang、Dohyeun Kim 和 Daeyoung Kim。支持 GPU 的无服务器计算框架。载于 2018 年第 26 届 Euromicro 并行、分布式与网络处理国际会议（PDP），第 533–540 页，2018 年。[12] Knative。https://github.com/knative/docs。[13] Alexandros Koliousis、Pijika Watcharapichat、Matthias Weidlich、Luo Mai、Paolo Costa 和 Peter Pietzuch。CROSSBOW：在多 GPU 服务器上以小批量大小扩展深度学习。arXiv 预印本 arXiv:1901.02244，2019 年。[14] Alex Krizhevsky、Geoffrey Hinton 等。学习微型图像的多层特征。2009 年。[15] Kubernetes。https://kubernetes.io/。[16] Yann LeCun。手写数字 MNIST 数据库。http://yann.lecun.com/exdb/mnist/。[17] 多进程服务（MPS）。https://docs.nvidia.com/deploy/mps/index.html。[18] Diana M. Naranjo、Sebasti´an Risco、Carlos de Alfonso、Alfonso P´erez、Ignacio Blanquer 和 Germ´an Molt´o。基于 GPU 虚拟化的加速无服务器计算。《并行与分布式计算杂志》，139(C)：32–42，2020 年 5 月。[19] Nvidia Docker 容器工具包。https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html。[20] OpenFaaS Pro 命名空间支持。https://docs.openfaas.com/reference/namespaces/。[21] PyTorch。https://pytorch.org/。[22] Klaus Satzke、Istemi Ekin Akkus、Ruichuan Chen、Ivica Rimac、Manuel Stein、Andre Beck、Paarijaat Aditya、Manohar Vanga 和 Volker Hilt。面向无服务器工作流的高效 GPU 共享。载于第 1 届高性能无服务器计算研讨会论文集，第 24、2020 年 17 st 页，第 – 页。[23] Mohammad Shahrad、Rodrigo Fonseca、 Í˜nigo Goiri、Gohar Chaudhry、Paul Batum、Jason Cooke、Eduardo Laureano、Colby Tresness、Mark Russinovich 和 Ricardo Bianchini。真实环境中的无服务器：大型云服务提供商的无服务器工作负载特征分析与优化。载于 2020 年 USENIX 年度技术会议（USENIX ATC 20），第 205–218 页，2020 年。[24] TensorFlow Serving。https://www.tensorflow.org/tfx/guide/serving。[25] Torchserv。https://pytorch.org/serve/。[26] Nvidia Triton 推理服务器。https://developer.nvidia.com/nvidia-triton-inference-server。[27] Tho mas Wolf、Lysandre Debut、Victor Sanh、Julien Chaumond、Clement Delangue、Anthony Moi、Pierric Cistac、Tim Rault、R´emi Louf、Morgan Funtowicz 等。Transformer：最先进的自然语言处理。载于《2020 年自然语言处理实证方法会议：系统演示》论文集，第 38 45 页，2020 年 –。[28] Wencong Xiao、Shiru Ren、Yong Li、Yang Zhang、Pengyang Hou、Zhi Li、Yihui Feng、Wei Lin 和 Yangqing Jia。AntMan：深度学习 GPU 集群的动态扩缩容。载于第 14 届 USENIX 操作系统设计与实现研讨会（OSDI 20），第 533–548 页，2020 年。[29] Ting-An Yeh、Hung-Hsin Chen 和 Jerry Chou。KubeShare：在容器云中将 GPU 作为一等共享资源进行管理的框架。载于第 29 届高性能并行与分布式计算国际研讨会（HPDC）论文集，第 173–184 页，2020 年。