arXiv:2601.11743v1 [cs.OS] 16 Jan 2026

Nixie：面向消费级 GPU 的高效、透明时间复用

Yechen Xu Yifei Wang NathanaelRen Yiran Chen Danyang Zhuo杜克大学

摘要

消费级计算机日益开始运行大型机器学习工作负载，例如大型语言模型（LLM）、文生图和交互式图像编辑。与数据中心 GPU 不同，消费级 GPU 面向单用户且不断变化的工作负载，并且每个模型的工作集通常几乎占满 GPU 内存。因此，当多个应用程序同时运行时，现有共享机制（例如 NVIDIA 统一虚拟内存）会因内存抖动和过度使用 CPU 固定内存而表现不佳。

我们设计并实现了 Nixie，这是一种可在消费级 GPU 上实现高效、透明时间复用的系统，无需对任何应用程序或驱动程序进行更改。Nixie 是一项系统服务，可协调 GPU 内存分配和内核启动行为，从而高效利用 CPU-GPU 双向带宽和 CPU 固定内存。Nixie 中的轻量级调度器还借助受 MLFQ 启发的技术，自动优先处理对延迟敏感的交互式任务，进一步提升响应能力。评估结果表明，在相同的延迟要求下，Nixie 可将实际交互式代码补全任务的延迟最多降低 3.8×，并节省高达 66.8% 的 CPU 固定内存使用量。

1 引言

现代消费级计算机越来越多地运行大型且复杂的机器学习 (ML) 工作负载。如今，消费者经常在配备消费级 GPU（如 NVIDIA 的 RTX 4090 和 5090）的台式机上运行大型语言模型 (LLM) [29, 37], 文生图扩散模型 [18], 交互式图像编辑 [35],和视频处理 [5, 32] 应用。这一趋势源于人们对隐私、交互性和降低硬件成本的追求，也得益于模型量化和优化推理运行时的快速发展 [13, 17, 42]，这些进步使本地机器学习执行变得切实可行。

尽管本地机器学习执行浪潮兴起，如今的消费级 GPU 在设计上仍然是为单个活动应用程序而打造的。与数据中心 GPU 不同，后者的工作负载通常持续时间长、采用批处理且由多个租户共享，而消费级工作负载则变化迅速、种类繁多，并且深受用户需求驱动。单个用户可能会频繁切换不同类型的模型和应用程序：查询大语言模型、生成图像、编辑照片，或运行光学字符识别、视频处理等后台批处理任务。交互式应用程序的批大小几乎总是 1，用户也期望交互式工作负载能够快速响应。这些特点构成了独特的资源管理环境，与数据中心 GPU 的环境有很大不同。

一个核心挑战是每个机器学习应用程序的工作集往往几乎占满 *GPU* 内存。现代大语言模型或扩散流水线等模型需要数十 GB 的活跃参数、激活值和临时工作区。用户往往更偏好准确度更高、规模更大的模型。因此，当两个或更多应用程序同时运行时，合计工作集很容易超出 GPU 内存。

现有的 GPU 复用机制在这些条件下表现不佳。llama-swap [21] 等简单的停止并重启方案会造成显著延迟：每次切换都需要重启应用程序并重新加载大型模型，导致响应时间达到数秒甚至更长。应用级模型交换（例如 Ollama [25]）仅限于单个应用程序，无法与其他应用程序协调。Prism [39]和 Ae- gaeon [36] 等以往系统需要手动集成，且主要针对多个小型模型共享大型 GPU 的场景，这与消费级 GPU 的情况并不相符；在消费级 GPU 上，每个模型的工作集几乎占满 GPU 内存。

NVIDIA 统一虚拟内存（UVM） [1] 提供了一种透明的 GPU 复用方式，但它依赖按需分页，并隐含地假设所有应用程序的工作集总和能够装入 GPU 内存，以维持良好的性能。当多个大型应用程序同时运行时，UVM 会出现严重抖动：页面在 GPU 和 CPU 内存之间反复迁移，导致吞吐量急剧下降，并产生大幅且难以预测的延迟峰值。TGS [34] 和 NVShare [3] 等系统缓解了 UVM 的部分抖动行为，但它们对工作负载的假设仍然严苛，无法反映

消费级 GPU 的使用模式。此外，它们继承了 UVM 的根本局限，包括过度使用 CPU 固定内存，这严重限制了大型内存密集型模型的可扩展性。

我们设计并实现了 Nixie，这是一种系统服务，可在无需修改任何应用程序或驱动程序的情况下，为消费级 GPU 提供高效、透明的时间复用。Nixie 会拦截 GPU 内存分配和内核启动行为。Nixie 的核心思想是明确控制应用程序何时占用 GPU 内存、何时让出内存，确保在任意时刻，单个应用程序的工作集都能完整驻留在 GPU 内存中。这样可以避免统一内存固有的病态抖动行为，并维持每个应用程序可预测的性能。为了实现快速上下文切换，Nixie 充分利用 GPU 与 CPU 之间的双向带宽。

Nixie 还集成了一个轻量级调度器，可自动识别延迟敏感型交互应用并赋予其更高优先级，无需用户为应用程序优先级添加任何注释。Nixie 使用类似多级反馈队列（MLFQ）的调度器。它通过跟踪应用程序的 CUDA 内核启动模式，动态调整应用优先级，并优先调度能够在分配的时间窗口内完成执行的应用程序。低优先级应用程序的时间窗口更长，以最大限度地减少上下文切换频率并摊销调度开销。

我们在消费级 NVIDIA RTX 5090 图形处理器上评估 Nixie，使用了广泛采用的应用程序（例如`llama.cpp` [13], Ollama [25],SGLang [42], ComfyUI [9]）和热门模型（例如 Qwen3 [37],Gemma3 [29],Z-Image [30],Qwen-Image [35]）。与 Ollama 等由应用程序管理的解决方案，以及基于统一虚拟内存的 GPU 复用系统 nvshare 相比，[3], Nixie 可将不同配置下的上下文切换开销降低 29.1-82.3%，并将 CPU 固定内存的使用量最多减少 66.8%。在由多应用工作负载组成的多样化工作负载（例如使用大语言模型生成图像生成提示词）上，Nixie 可实现 1.3-1.6×倍加速。Nixie 还可将交互式代码补全延迟提升 3.1-3.8×倍，同时在后台批处理工作负载上保持具有竞争力的吞吐量。

本文作出以下贡献：

- 一种全新的系统架构，可联合管理 GPU 内存使用量和内核调度，无需更改驱动程序或应用程序，即可消除 UVM 式抖动。该架构能够高效利用 CPU 固定内存以及 CPU 与 GPU 之间的双向带宽。

- 一种受多级反馈队列启发的轻量级图形处理器工作负载调度器，可自动识别并优先调度交互式应用程序，在保持整体图形处理器吞吐量的同时降低延迟。

- 一套无需修改即可支持先进图形处理器应用程序（如 llama.cpp、SGLang 和 ComfyUI）的实现，并能显著提升多种消费级 GPU 工作负载的表现。

2 动机

在此，我们阐述为何消费级 GPU 复用问题与数据中心 GPU 复用问题有着本质区别。

2.1 消费级 GPU 上的本地机器学习工作负载

近年来，模型量化和优化推理运行时的发展，使用户能够直接在个人计算机上运行大型机器学习（ML）模型。本地大语言模型推理（例如 llama.cpp、SGLang、vLLM）、文生图和交互式图像编辑流水线（例如 ComfyUI、基于扩散的工具）等应用程序日益普及。用户选择这些方式，是出于隐私、离线能力、更多微调机会以及降低硬件成本等考虑。

与传统的数据中心 GPU 工作负载不同，单个用户可能会同时运行或频繁切换多个大型模型：在共享的消费级 GPU 上查询大语言模型、生成图像、编辑照片、运行后台 AI 任务，以及使用其他应用程序。对于交互式应用程序，批大小几乎总是 1。与此同时，现代消费级 GPU（例如 RTX 4090/5090）计算能力强大，但相对于模型大小而言，GPU 内存有限。用户通常会运行能够装入内存的最大模型，以尽可能提高质量。然而，这意味着每个应用程序都可能几乎耗尽 GPU 内存。

2.2 消费级 GPU 上现有多路复用方法的局限

现有方法一：关闭一个应用程序，再运行下一个应用程序。 一种直接的解决方案是一次只运行一个应用程序。例如，社区工具 llama-swap [21]可通过终止当前应用程序并启动下一个应用程序来实现切换。然而，卸载和重新加载大型模型会造成数秒的延迟，使得交互式消费场景中的常规任务切换也变得不切实际（例如，提示大语言模型生成图像描述，并立即将其用于基于扩散模型的文生图应用程序）。

现有方法二：特定应用方法。Ollama 等应用程序 [25]通过卸载当前模型并加载新模型来切换模型。然而，这种机制与单个应用程序的

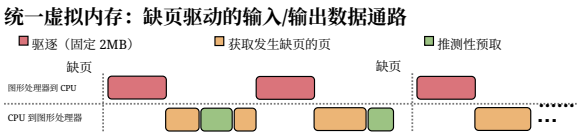


图 1：UVM 的缺页驱动迁移。对于全双工 PCI Express 通道，UVM 只能利用一半的链路带宽。

内部执行紧密耦合，无法与其他 GPU 应用协调。

另一类研究致力于使推理系统的 GPU 内存使用量具备弹性。例如，Prism [39],允许大语言模型框架动态缩减其 KV 缓存，为其他应用程序释放 GPU 内存。尽管这种方法对大语言模型有效，但它高度依赖特定领域：扩散模型、图像编辑流水线以及许多其他消费级工作负载不使用 KV 缓存，因此无法从该机制中受益。此外，这些系统要求在每个应用程序或推理框架中进行显式的软件级集成。因此，它们无法在消费级 GPU 上各种应用程序之间提供透明的系统级多路复用。

现有方法 #3：NVIDIA 统一虚拟内存（UVM）。 NVIDIA 统一虚拟内存（UVM）是一种用于处理 GPU 内存超额订阅的完全透明机制。其基本思想类似于按需分页：当 GPU 内核访问不在 GPU 内存中的页面时，硬件会向驱动程序触发缺页中断。随后，驱动程序的处理程序选择要驱逐的页面，在 GPU 和 CPU 内存之间迁移数据，并恢复内核执行。UVM 适用于：(1) 总内存占用超过 GPU 容量、但工作集能够放入 GPU 内存的单个应用程序；以及 (2) 合并后的工作集能够容纳在 GPU 内存中的多个应用程序。

尽管 UVM 完全透明的抽象颇具吸引力，但在消费级场景中，它存在四个根本性局限。首先，UVM 无法协调控制计算和内存管理，因此极易发生抖动。考虑两个应用程序 A 和 B 在一块 32 GB 的图形处理器上并发运行，并以轮询方式各生成一个 token。假设每个模型需要 24 GB 参数，那么每次前向传播都会触发至少 16 GB 数据与 GPU 内存之间的迁移。即使采用最先进的 PCIe 5.0×16 带宽（64 GB/s），这次迁移也需要大约 250 毫秒。相比之下，不涉及迁移的前向传播通常只需 20–75 毫秒。因此，仅数据传输就会导致 4-12×倍的性能下降。

其次，统一虚拟内存一次只使用 PCI Express 带宽的一个方向，尽管 PCI Express 通道是全双工的。图 1 展示了统一虚拟内存在缺页处理期间 CPU 与图形处理器之间的 I/O 路径。发生缺页时，统一虚拟内存会先将页面从图形处理器驱逐到 CPU 内存，只有驱逐完成后，才会提取所需页面，并可能再将若干预取页面从 CPU 内存传输到图形处理器内存。单次缺页可能会触发多次驱逐，随后再进行多次提取，但这些传输从不重叠：驱逐只使用从图形处理器到 CPU 的带宽，而提取和预取只使用从 CPU 到图形处理器的带宽。因此，可用 PCI Express 带宽有一半处于闲置状态。

第三，由于无法充分了解应用程序行为，UVM 在 GPU 缺页时依赖基于最近最少使用（LRU）的驱逐策略来选择页面 [4]。然而，UVM 只有在发生缺页异常时才会更新其 LRU 元数据，因此无法完整反映应用程序真实的内存访问模式。结果是，即将被访问的页面常被误认为冷页面，进而导致不必要的页面驱逐和抖动。

第四，当在消费级 GPU 上使用 UVM 时，主机内存会成为一个主要瓶颈。为了最大化 PCIe 传输性能，UVM 会在内核中分配 CPU 固定内存，用于 DMA 操作。当 UVM 作为多个 GPU 应用的后备存储时，GPU 上驻留的每个页都必须在 CPU 上对应一个固定页。这会产生大量内存开销：即使某些数据永远不会从 GPU 中被驱逐，仍会占用无法换出的、无法压缩的 CPU 固定内存。在配备有限 RAM 的消费级计算机上，这些 CPU 固定内存会直接与其他应用程序争夺资源，并显著降低整个系统的可用性。

一些 GPU 多路复用系统构建于 UVM 之上，以缓解其部分抖动问题（例如 TGS [34] 和 NVShare [3]）。例如，TGS 会为应用程序分配优先级，使低优先级应用程序仅在高优先级应用程序处于空闲状态时运行。然而，由于这些系统依赖 UVM 作为底层 GPU 内存虚拟化机制，它们继承了 UVM 的根本局限：由页错误驱动的单向 PCIe 传输，以及大量的固定内存消耗。

3 概述

我们的设计遵循三个关键目标。首先，我们旨在提供类似 UVM 的抽象，支持未经修改的 CUDA 应用程序，同时实现对计算和内存虚拟化的协调控制。其次，我们致力于重新设计虚拟内存管理，以利用涵盖 GPU 内存、CPU 固定内存、CPU 可分页内存和磁盘的分层内存系统，同时确保 CPU-GPU 数据传输充分利用全双工 PCIe 带宽。第三，与标准操作系统内核调度器类似，我们旨在优化交互式应用程序的延迟，同时不牺牲后台批处理工作负载的吞吐量。

图 2 展示了 Nixie 的架构。该系统由两个组件构成：轻量级应用程序库 Nixie Shim，以及集中式系统服务 Nixie Daemon。为实现透明性，Nixie Shim 会拦截应用程序对 CUDA 运行时 API 的调用。Nixie Daemon

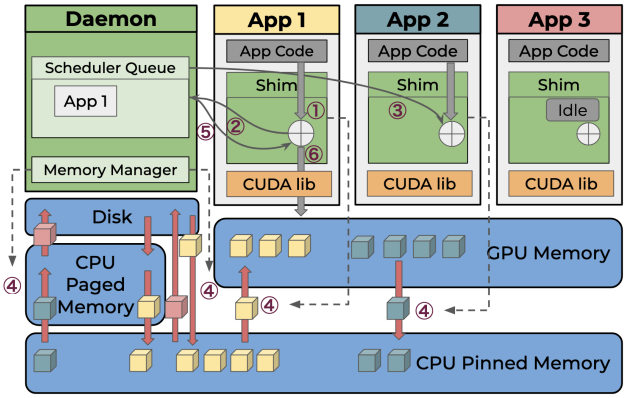


图 2：Nixie 的架构概览。绿色组件属于 Nixie。不同颜色的立方体代表来自不同应用程序的内存块。

负责全局协调并实施多路复用策略，但所有内核仍在应用程序自身的 CUDA 上下文中执行，以尽量减少内核启动的运行时开销。当 App 1 启动新内核以执行图形处理器作业时，它❶会检查 Nixie Shim 中的执行标志；❷如果 App 1 无法执行，Nixie Shim 随后会通过进程间通信将调度请求加入 Nixie Daemon 的调度器队列。当 Nixie Daemon 中的调度器决定允许 App 1 运行时，❸Nixie Daemon 会通过进程间通信暂停 App 2 的执行，并等待其正在执行的 CUDA 内核完成。接着❹Nixie Daemon 会将属于 App 1 的所有内存块移至共享固定内存，而每个 Nixie Shim 则将自身的内存块移入或移出图形处理器内存。待 App 1 的内存块准备就绪后，❺Nixie Daemon 会通知 App 1 中的 Nixie Shim 启用执行标志；随后 App ❻恢复执行。当 App 1 的所有数据准备就绪后，App 1 只需执行步骤❶和❻，无需通过进程间通信与 Nixie Daemon 通信。

Nixie Daemon 维护一个块级虚拟内存系统。Nixie 不依赖缺页驱动迁移，而是以 CUDA 分配的内存块为粒度管理内存（即 cudaMalloc 返回的单元）。每个内存块可位于以下四个位置之一：（1）图形处理器内存，（2）CPU 固定内存，（3）CPU 分页内存，或（4）磁盘。与统一虚拟内存不同，放置在图形处理器内存中的内存块无需占用 CPU 固定内存。为避免 CPU 固定内存或分页内存中出现内存碎片，我们将一个内存块拆分为多个 2MB 块，其中每个块都是不同位置之间数据迁移的单位。

Nixie Daemon 中的调度器会联合管理：（1）哪些应用程序可以启动内核；（2）Nixie 中内存块的位置。其设计遵循两项关键原则。首先，Nixie 会协调对计算和内存分配的控制。当调度器选择某个应用程序运行时，它会主动将该应用程序的内存块迁移到 GPU 内存，同时将其他应用程序的内存块换出到 CPU 内存或磁盘。

通过将调度与内存放置相结合，Nixie 确保应用程序开始执行时，同时具备 GPU 计算能力和所需的驻留于 GPU 的数据，从而避免按需分页造成的停顿并消除抖动。其次，Nixie 会自动优先处理交互式工作负载。Nixie 通过跟踪应用程序的内核启动行为来判断其是否具有交互性。借鉴 MLFQ 的思想，Nixie 会动态调整每个应用程序的优先级和时间窗口。如果应用程序在分配给它的时间窗口内未启动内核，Nixie 会将其归类为交互式应用程序，将其窗口大小减半，并提高其优先级。相反，如果应用程序完全用尽了分配给它的时间窗口，Nixie 会将窗口大小加倍，并降低其优先级。在每个决策点，调度器都会选择优先级最高的应用程序，并授予其在当前时间窗口内访问 GPU 的权限。这种方法使 Nixie 无需任何用户注释或应用程序修改，即可自动优先处理延迟敏感型应用程序，同时保持后台批处理应用程序的吞吐量。

4 支持未经修改的 CUDA 应用程序

Nixie 对 CUDA 应用程序是透明的，与统一虚拟内存类似。在统一虚拟内存中，用户需将所有 CUDA 内存分配调用从 cudaMalloc 替换为 cudaMallocManaged；通过静态重写或 LD_PRELOAD，即可轻松实现完全透明。

Nixie 使用 LD_PRELOAD 钩住 CUDA 库并拦截若干函数。除了内存分配和释放外，Nixie 还会拦截 CUDA 内核/图的启动，以控制计算行为。此外，Nixie 还会拦截任何会隐式分配内存的应用程序接口（例如 cudaStreamCreate），并报告内存使用情况（例如 cudaMemGetInfo）。应用程序有时需要准确的内存使用情况报告才能正常运行。例如，SGLang 会读取 GPU 内存，以决定分配多少 KV 缓存内存。Nixie Shim 提供此类拦截功能，并通过 UNIX 域套接字与 Nixie Daemon 通信。

为了确保应用程序无需知晓 Nixie 的存在也能正常运行，我们需要保证暂停前后运行环境完全一致，并且 Nixie 不应引入任何无效的 GPU 操作。

保持虚拟内存地址一致。 像 `cudaMalloc` 和 `cudaFree` 这样的经典方法无法保证再次调用后虚拟内存地址仍保持一致。使用先前地址的应用程序会访问已失效的内存区域。Nixie 利用 CUDA VMM API [24] 预留 GPU 虚拟地址，并在数据块迁移前后将这些地址映射到不同的物理内存分配。

迁移期间的有效内存访问。 GPU 内核的执行大多是异步的。迁移开始时，可能仍有尚未完成的内核在运行；如果这些内核访问的页面已被迁移，就可能发生无效内存访问。在迁移当前正在运行的应用程序之前，Nixie 会阻止任何新内核启动，并执行 CUDA 同步，以完成所有尚未完成的内核。这可以防止应用程序使用已迁移的数据块。

CUDA 图兼容性。 CUDA 图被广泛用于降低内核启动开销。应用程序构建 CUDA 图时，意外的 API 调用可能会中断图捕获并导致意外行为。Nixie 会拦截捕获事件的开始和结束，确保在图捕获完成之前，Nixie Shim 不会调用任何 CUDA API。

GPU 内存信息。 有些应用程序假定自己可以独占访问整个图形处理器，并使用设备范围的统计数据来决定张量放置或缓冲区分配。Nixie 根据进程自身的分配总量报告 GPU 内存使用量，避免应用程序不必要地将数据溢出到 CPU。

5 虚拟内存管理

我们首先介绍 Nixie 中内存管理的粒度，然后说明如何在分层内存中迁移内存块。

5.1 内存管理的粒度

Nixie 会拦截应用程序中的所有 GPU 内存分配，因此能够自然地了解所有内存分配。Nixie 按每次分配的粒度管理内存，并将这些单元作为整体迁移，我们将其称为块。每个块都放置在存储层次结构的特定层级中。有些应用程序（例如 llama.cpp）会将所有模型权重放入一个大型分配中。为避免块过大，Nixie 将每个块的大小限制为 128 MB。更大的分配会相应地拆分，使同一分配的不同部分能够驻留在不同的存储层次结构中。例如，一个 256 MB 的分配可以拆分为大致一半驻留在 CPU 固定内存中，其余部分则放置在 CPU 分页内存中。

由于张量（例如权重、激活值）的形状各不相同，分配的内存大小自然也会有所不同。如果 Nixie 仅在块级别管理内存，这种差异就会导致严重的内存碎片。为解决这一问题，Nixie 引入了块，以实现细粒度控制。每个块的固定大小为 2 MB，这是 VMM API 支持的最小物理分配单位。一个块由一个或多个块组成（最多 64 个）。使用大小统一的块可简化放置、迁移和记账，使 Nixie 能够高效管理内存，同时避免内存碎片。

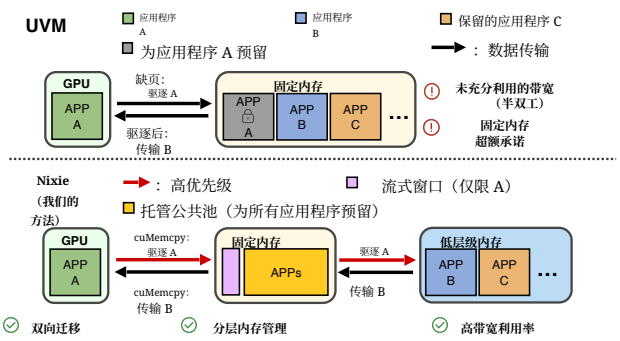


图 3：Nixie 的内存层次结构和块迁移。

5.2 分层内存

这种分层内存的一种典型实现方式，是将层 X 的全部内存固定在层 $X+1$，并将层X 视为层 $X+1$的缓存层。如果层 $X+1$ 的容量始终显著大于层 X，这种方式的效果非常好。在 CPU 缓存层次结构中，这是一个合理的假设；就将 GPU 内存视为 CPU 固定内存的缓存而言，这也是 UVM 所采用的假设。

然而，在消费级机器上，这一假设并不成立，因为 CPU 固定内存容量有限，且通常与 GPU 内存容量相当。假设有三个应用程序，每个都需要 24 GB 的 GPU 内存：UVM 将需要 72 GB 的 CPU 固定内存作为后备存储。虽然固定内存具有较高的传输吞吐量，但无法被换出，也无法由操作系统透明压缩。在运行单一工作负载的专用推理服务器上，这种情况尚可应对；但在多个应用程序共存的消费级硬件上，过度使用固定内存会严重降低整个系统的可用性。

我们不将层 X 用作层 $X+1$的缓存层，而是确保每个数据块在整个内存层次结构中恰好只有一份副本，并根据需要在各层之间迁移。图 3 展示了 Nixie 的内存层次结构。Nixie Daemon 是唯一负责发起数据迁移的组件，并掌握决定数据块在内存层次结构中位置所需的全部信息。Nixie Daemon 可以主动执行后台迁移，而无需涉及或中断应用程序。与统一虚拟内存（UVM）相比，这使 Nixie 能够使用少得多的锁页内存。

5.3 加速上下文切换

当 GPU 内存不足以容纳接下来要运行的应用程序时，就需要进行迁移，而迁移时间过长会严重影响用户体验。为了尽可能缩短迁移持续时间，我们需要充分利用链路

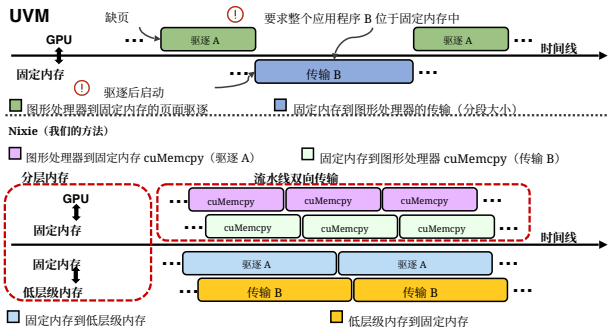


图 4：Nixie 的通信带宽使用情况：(1) GPU 内存与 CPU 固定内存之间；(2) CPU 固定内存与较低层级内存（例如 CPU 分页内存、磁盘）之间。

带宽。我们在图 4中展示了迁移过程：即将运行的应用程序 B 需要将数据迁移到 GPU 中，而当前应用程序 A 则需要被换出。

一种直接的解决方案是，分别在相邻层级之间创建多个双向传输任务，并利用背压确保有可用空间进行传输。然而，在多层级的分层场景中，背压链很长。为满足上游的传输请求，背压需要多次向下传递，而且只有相应的下游传输完成后，上游传输才能进行。这会导致较长的延迟，并降低传输链路的利用率。另一个问题是争用，尤其是在当前层级的容量小于应用程序大小时。如果传输未经过协调，B 的数据可能会完全占据这一层级。在这种情况下，我们要么必须将数据移回较低层级，从而产生不必要的流量；要么在预期 B 的数据只向上层移动时，发生更严重的死锁。

为解决这些问题，我们实现了两种方案。首先，我们不直接启动传输并仅在本地解决争用，而是通过检查所有应用程序的当前使用情况来制定全局计划。规划器会决定需要迁移哪些数据块，以及它们的源和目标，如图 5所示。Nixie Daemon 会收集所有需要迁移到图形处理器的数据块元数据，并计算图形处理器上应驱逐哪些数据块。作出驱逐决策后，规划器会逐层检查，以确定每一层能够保留多少被驱逐的数据，而无需将其进一步传输到更低层。规划器会优先处理跨越多个层级的传输。

其次，我们在 Nixie Daemon 中加入迁移编排器，以便在空间有限且块的源和目标各不相同的情况下管理迁移顺序和优先级。我们还预留了一个专用于换出应用程序 A 的小型流式窗口。通过预留该窗口，我们确保背压链下游不会被阻塞。此外，当不存在背压时，来自 A 的数据可以安全地留在此处，而不会

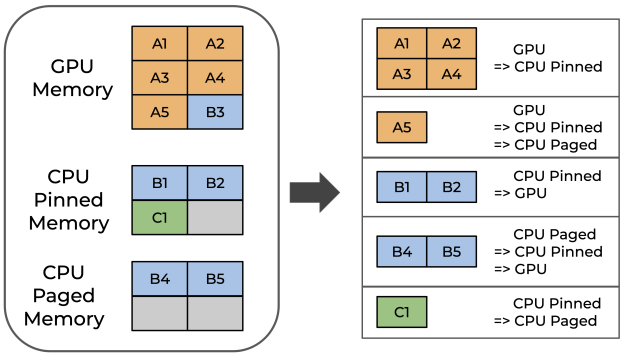


图 5：根据现有数据块的位置制定的迁移计划。

无需进一步移动，也不会影响延迟。

对于迁移，Nixie Daemon 会将传入应用程序所需的数据移至 CPU 固定内存。我们采用多线程传输以充分利用 DRAM 带宽。当传入数据在 CPU 固定内存中准备就绪后，块的 ID 及其数据块会通过 IPC 发送至 Nixie Shim。Nixie Shim 利用异步内存复制，在 2MB 块与图形处理器之间传输数据。

6 调度

与传统的 CPU 调度类似，我们的目标是：（1）交互式应用程序应获得比批处理应用程序更高的优先级；（2）不应要求用户手动指定应用程序优先级。在 CPU 上，多级反馈队列（MLFQ）是实现这两个目标的经典机制。

然而，图形处理器也带来了新的挑战。首先，没有直接的信号能够表明应用程序是交互式还是批处理式。在 CPU 上，可以通过主动让出 CPU（例如调用 yield()）或发出用于输入/输出的阻塞式系统调用等事件来推断交互性，但图形处理器工作负载中不存在这类信号。其次，由于需要在存储层次结构之间迁移大量内存块，图形处理器上下文切换的开销要高得多。

接下来，我们将介绍 Nixie 如何检测未使用图形处理器的应用程序，并据此推断应用程序的优先级。最后，我们将讨论如何利用预取，为图形处理器工作负载定制类似多级反馈队列的调度系统。

6.1 空闲检测

在传统的 CPU 调度中，调度器通常会在以下情况下被调用：（1）时间窗口到期；或（2）线程主动让出 CPU，例如调用 yield() 或发起阻塞式系统调用（如输入/输出）。这些信号使 CPU 调度器能够切换到另一个执行线程。然而，GPU 应用中不存在这两种信号。首先，Nixie 需要支持长期运行的服务（如 SGLang、Ollama），因此不存在“任务

完成”的概念。其次，GPU 应用不会提供 yield 或阻塞式系统调用的等效机制来表示空闲时段。因此，无论应用程序是否在执行有用的工作，都让其一直运行到时间窗口到期，会导致图形处理器利用率低下，并使交互式工作负载的延迟升高。

为解决这一问题，我们设计了一种机制，用于检测图形处理器应用程序何时进入空闲状态。一种直接的方法是使用 NVML 等设备 API 观察图形处理器利用率。遗憾的是，这种方法有两个缺点。首先，图形处理器级 API 的粒度较粗，更新也较慢。在我们的实验中，NVML 大约需要 600 毫秒才能报告工作负载已进入空闲状态。其次，在消费级设备上，后台任务（例如渲染或视频解码）会引入噪声，使这些信号不可靠。

我们使用基于超时的机制来检测应用程序是否处于空闲状态。CUDA API 分为两类：非阻塞操作（例如内核启动、异步调用）和阻塞操作（例如同步）。只有当距离应用程序最近一次 API 返回的时间超过固定阈值时，Nixie 才会将该应用程序标记为空闲。对于阻塞 API，Nixie Shim 会在调用执行前后拦截该调用，以确保应用程序在运行时内部阻塞时不会被错误地判定为空闲。在我们的实现中，阈值设为 100 毫秒。

6.2 优先级推断

经典的多级反馈队列会随时间降低优先级，并定期将所有优先级重置为最高级别。这种方法在 CPU 上效果良好，因为时间窗口较短。然而，在 Nixie 中，图形处理器的时间窗口要长得多。盲目重置所有优先级会抹去有用的历史记录，还可能意外抢占交互式工作负载。

我们提出了一种软优先级恢复策略。调度器会记录每个应用程序a的三个时间戳：(1) 进入空闲状态的时间i_a，(2) 上次更新优先级的时间p_a，以及 (3) 最近一次调度请求的时间q_a。理想情况下，当应用程序空闲足够长的时间时，表明其使用模式可能已经改变（例如，大语言模型现在可能会针对一个提示生成较长的回复，而针对另一个提示生成较短的回复），因此应提高其优先级，以重新估算其运行时特征。然而，如果该应用程序已在调度器队列中等待后续执行，立即提升其优先级就会抢占当前正在运行的应用程序，并缩短该优先级下所有应用程序的有效时间窗口。为解决这一问题，我们不将等待时间纳入考虑。令R 表示应用于等待时间的乘数。如果$R \geq 1$，那么只要高优先级任务持续运行，应用程序就可能在低优先级下发生饥饿。因此，我们根据处于同一优先级的应用程序数量N 动态调整R ，确保$R < 1/N$。详细的优先级推断算法如下所示

算法 1 Nixie 优先级推断

要求： 应用程序 a、待处理时间因子 R、多级反馈队列集合 $\{Q\}$ 及对应的时间分配量 $\{T\}$

$p \leftarrow a$的当前优先级
$t_a \leftarrow a$ 在当前优先级下的累计执行时间
如果 $t_a > T_p$ **则** ▷ 标准多级反馈队列降级
    将a 移至较低优先级队列
    $t_a \leftarrow 0$
elseif$a$处于空闲状态**then**
    $i_a \leftarrow$ 距上次执行的时间
    $p_a \leftarrow$ 距上次优先级变更的时间
    $q_a \leftarrow$ 自a 入队以来的时间（若无新请求则为 0）
    如果 $p_a > T_p$ **则** ▷ 防止优先级抖动
        // 补偿空闲并防止饥饿
        如果 $i_a - R \cdot q_a > T_{p-1} + t_a$ **则**
            将a 移至优先级更高的队列
        结束如果
    结束如果
结束如果

在 算法 1。

6.3 带预取的多级反馈队列

Nixie 使用 K 个队列（Q_1,$Q2$,…,Q_k），优先级从Q_1 到Q_k依次递减。调度器优先调度优先级较高的应用程序；多个应用程序优先级相同时，则以轮询方式调度，首先调度等待时间最长的应用程序。达到抢占阈值S后，如果存在另一个优先级相同的应用程序，调度器就会抢占当前应用程序。最高优先级队列的优先级降级时间配额设为$T = 8$s，抢占阈值设为$S = 4$s；之后每个优先级较低的队列都将这两个参数加倍。调度器会定期检查是否需要进行新的调度，并更新优先级。收到当前正在运行的应用程序发出的空闲通知后，调度器会执行相同的操作。

在 Nixie 的分层内存设计中，将数据从较低层级移至较高层级需要较长时间。我们利用调度器提供的信息来缩短迁移时间：当调度队列队首已有一个应用程序时，它最有可能在当前应用程序被调度离开后运行。因此，Nixie 会预取下一个应用程序所需的数据，以缩短迁移时间。

7 评估

我们使用 ~10,000 行 Rust 代码实现了 Nixie 原型。Nixie Shim 包含 ~1900 行代码，Nixie Daemon 包含 ~7400 行代码，其中有 ~700 行共享代码。接下来，我们

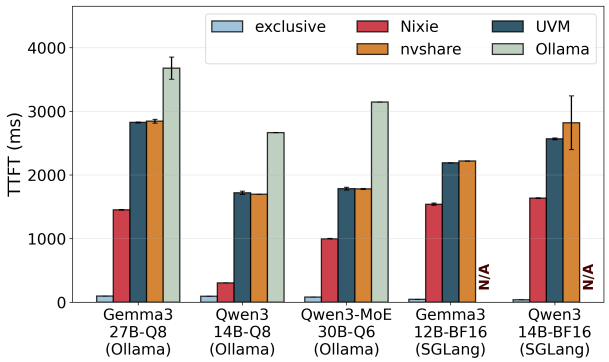


图6：上下文切换性能对比。误差条表示标准差。

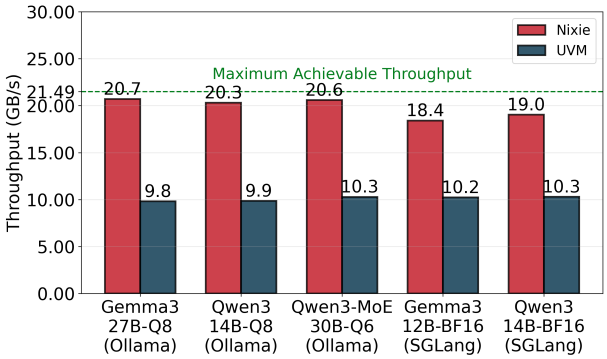


图 7：CPU 与 GPU 之间的内存复制吞吐量。

对 Nixie 的上下文切换延迟进行微基准测试，同时测试其计算开销和内存开销。然后，我们评估一组具有代表性的用例，以展示 Nixie 的实际优势。

实验设置。 我们的实验在一台台式机上进行，该机器配备 AMD Ryzen 9 9950X CPU 和 96 GB 双通道 DDR5 内存，运行频率为 3600 MT/s。系统配备两块 32GB RTX 5090 GPU，每块均通过 PCIe 5.0x8 连接。台式机运行 Debian 12，CUDA 版本为 12.9，NVIDIA 驱动程序版本为 580.95.05。我们使用 Ollama [25]（版本 0.12.11）、SGLang [42]（版本 0.5.4.post1）、`llama.cpp` [13]（版本 b7027）和 ComfyUI [9]（提交 eaf68c9）运行机器学习模型。我们从 Qwen3 [37], Gemma3 [29],Z-Image [30] 和 Qwen-Image [35] 中挑选了多种不同尺寸和量化方式的模型。例如，Qwen3-MoE 30B-Q6 表示该模型是 Qwen3-MoE 的 30B 版本，采用 6 位量化。

基线。 我们将 Nixie 与 Ollama [25]、UVM [1]、TGS [34]，和 nvshare [3] 进行比较。UVM 基线仅通过挂钩 cudaMalloc、cudaFree 和 cudaMemGetInfo 这三个应用程序接口来实现。我们既将 Ollama 作为运行于 Nixie 之上的应用程序，也将其作为基线：Ollama 可执行机器学习模型，并原生支持在模型之间切换。我们仅通过案例研究评估 TGS，并将其排除在微基准测试之外，因为 TGS 只支持恰好两个应用程序，且要求用户明确指定其中一个为高优先级、另一个为低优先级，因此与我们的微基准测试工作负载不兼容。

7.1 微基准测试

上下文切换性能。 我们通过在单个 GPU 上运行执行模型的应用程序，然后切换到同一应用程序及模型的另一个相同实例，来评估上下文切换性能。 图6 报告了以下五种设置下的首 token 时间（TTFT）： 独占模式、Nixie、统一虚拟内存、nvshare 和 Ollama。在此，独占模式 将两个应用程序置于不同的 GPU 上，因此反映了上下文切换实际上无需任何开销时的首 token 时间（TTFT）。

总体而言，与 UVM 和 nvshare 相比，Nixie 在 Ollama 案例中将首令牌生成时间缩短了 44.0%–82.3%，在 SGLang 案例中缩短了 29.7%–36.3%。Ollama 使用一种简单的基于 mmap 的策略来卸载和重新加载模型，因此在所有测试方案中性能最差。

我们强调两个有趣的观察结果。首先，对于混合专家模型（例如 Qwen3-MoE），Nixie 在首令牌生成时间方面的性能优势较小，因为预填充阶段只会激活部分专家。其次，相较于 SGLang，Nixie 为 Ollama 带来的收益更大。SGLang 会激进地分配未使用的 KV 缓存内存；Nixie 会主动将所有未使用的 KV 缓存页卸载到 CPU 内存，而 UVM 则会延迟迁移这些页面。

为了了解 Nixie 为何比 UVM 和 nvshare 更能缩短上下文切换时间，我们测量了主机内存与图形处理器内存之间的双向数据传输吞吐量。图 7展示了测量结果。作为参考，虚线标示了我们测试平台上可实现的最大双向吞吐量，该数值通过 NVIDIA 的 `nvbandwidth` [22]工具测得。

Nixie 几乎充分利用了可用的双向带宽，达到约 2× UVM 的吞吐量。Nixie 与理论最大值之间剩余的差距主要来自内存分配开销。Ollama 采用简单且分配规模较大的分配模式，而 SGLang 的分配模式则更复杂，且规模不均。因此，运行 Ollama 时，Nixie 能够实现比运行 SGLang 时更高的双向吞吐量。

在图 8中，我们分析了与不同规模的应用程序共置时对首令牌生成时间的影响。我们使用`llama.cpp` 作为主要应用程序，并使用 Gemma3 27B-Q8 作为测量首令牌生成时间的模型。随着 GPU 内存超额订阅总量减少，Nixie 的上下文切换性能有所提升；而无论 GPU 内存超额订阅程度如何，UVM 都因采用 LRU 策略而保持恒定的上下文切换开销。

有时，用户在需要同时运行其他应用程序时，可能会遇到 CPU 内存不足的情况。统一虚拟内存利用固定内存进行 DMA，以实现最佳迁移性能。我们改变 CPU 固定内存的总量

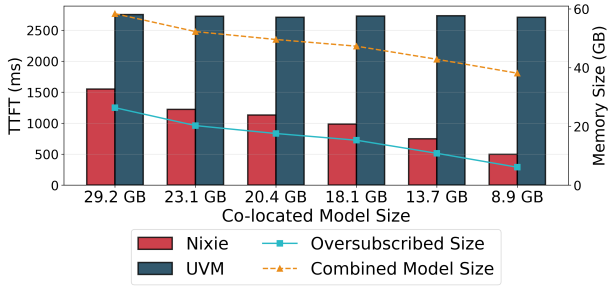
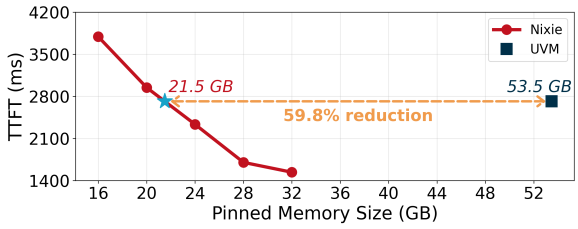
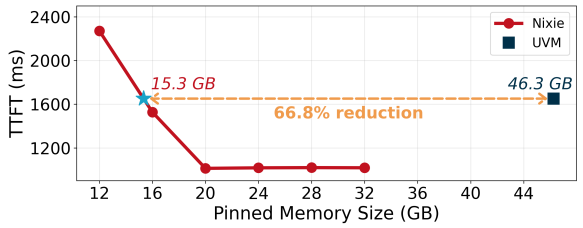


图 8：不同内存超额订阅量下的性能比较。超额订阅大小 = 组合模型大小 - GPU 内存大小。



(a) Gemma3 27B-Q8



(b) Qwen3-MoE 30B-Q6

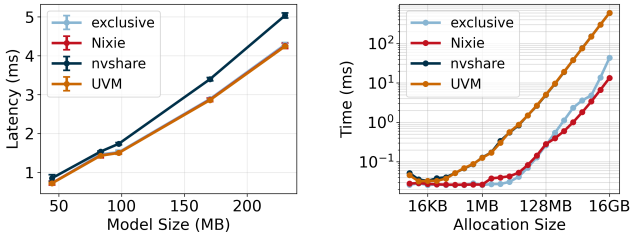
图 9：固定内存与首令牌生成时间

Nixie 可使用 16GB 至 32GB。在 图 9中，我们展示了 Nixie 仅使用 UVM 所需 CPU 固定内存的 33.2% 至 40.2%，即可达到与 UVM 相同的性能。

运行时开销。应用程序被安排在图形处理器上运行期间，Nixie 在内核启动方面应尽可能减少性能开销，即使内核本身很小也是如此。这是因为在应用程序被安排运行的时间段内，CUDA 启动调用会直接通过相应的 CUDA用户态驱动程序，不需要经由 Nixie Daemon 处理。

图 10a 展示了批量大小为 1 时，从 ResNet-18 到 ResNet-152 的各种小型残差网络模型的推理延迟。Nixie 的性能与 独占模式下的原生执行和简单的 *UVM* 拦截完全相同。*nvshare* 的开销主要来自其 *CPU* 线程同步，以及特定情况下的 *GPU*同步。

内存开销。 在图 10b中，我们测量了不同分配大小下 `cudaMalloc()` 的延迟。随后执行 `cudaMemset()` 以消除延迟初始化的影响



(a) 推理延迟　　(b) cudaMalloc 延迟

图 10：Nixie 的运行时开销。

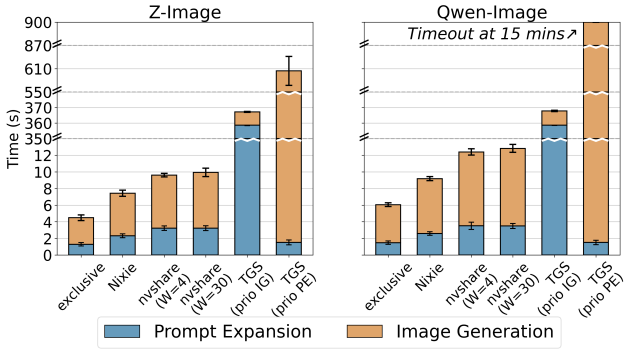


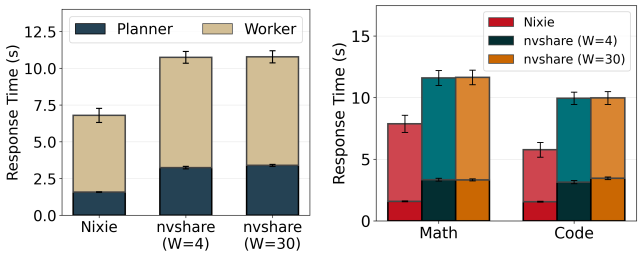
图 11：使用 Qwen3-MoE 30B-Q6 扩展文本到图像提示词，以获得更好的图像质量。误差线表示标准差。

。对于小于 2MB 的分配，Nixie 直接使用 cudaMalloc()，性能与原生实现相当。在 2MB 到 128MB 之间，Nixie 的分配时间略长于原生实现。由于 CUDA 运行时是闭源的，对于分配大小超过 128 MB 后 Nixie 为何优于原生实现，我们只能提出假设。我们推测，Nixie 使用多个连续的 128 MB 物理区域分配内存，而原生实现可能会尝试分配单个最大连续区域。UVM 和基于 UVM 的 nvshare 性能更差，因为 UVM 采用首次触碰分配策略，将物理内存分配推迟到首次访问时才进行，从而引入了更多开销。

7.2 案例研究

我们介绍四个具有代表性的用例。前两个用例属于多应用执行工作流，后两个用例则属于并发工作负载，其中多个工作负载彼此独立且不存在依赖关系。

案例 #1：提示词扩展图像生成。提示扩展 [10, 19] 是图像生成中的一种标准技术。当用户希望根据提示生成图像时，会使用大语言模型扩展提示，使图像描述更加精准，并包含更多具体细节。这一工作流程需要跨应用



(a) 总体延迟 (b) 按请求类型划分的延迟

图12：用于数学和代码任务的多智能体编排。误差线表示标准差。

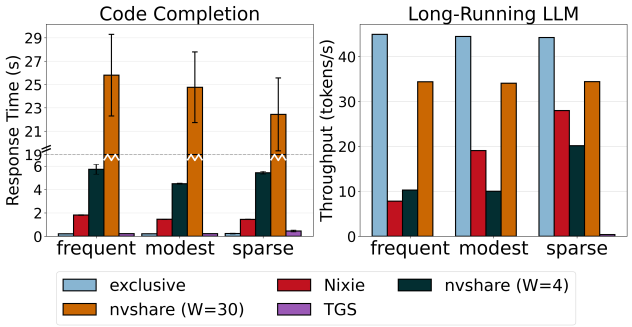


图13：长时间运行生成任务中的代码补全。误差条表示标准差。

大语言模型推理应用与图像生成应用之间的协作。

我们使用 `llama.cpp` 和 ComfyUI 对此工作流进行评估。对于提示扩展，我们使用 Qwen3-MoE 30B-Q6 [37]。对于图像生成，我们选择两个常用模型：(1) 采用 BF16 的 6B Z-Image [30]，以及 (2) 20B Qwen- Image [35]，采用 FP8。我们的提示使用 DreamBench++ 数据集 [26]。我们在两种窗口配置下将 Nixie 与 nvshare 进行比较：W = 4s，该配置与最高优先级的 Nixie 抢占阈值相匹配；以及 W = 30s，这是 nvshare 的默认设置。由于 TGS 需要显式的优先级分配，我们纳入了两个变体：一种优先处理图像生成（prio IG），另一种优先处理基于大语言模型的提示扩展（prio PE）。

图 11 报告了结果。Nixie 1.4× 并且 1.3× 比 nvshare（W = 4）更快；对于 Qwen-Image 和 Z-Image，Nixie 的性能分别达到使用两块专用图形处理器时的 65.9% 和 60.4%。值得注意的是，尽管 ComfyUI 在 Qwen-Image 生成过程中会频繁分配和释放图形处理器内存，Nixie 仍能保持更高的性能，这表明 Nixie 对数据路径上的内存分配和释放具有良好的性能稳健性。

TGS 的性能显著下降，并且在 15 分钟后仍无法完成 Qwen-Image 的图像生成。主要有两个原因：(1) TGS 假设高优先级作业必须维持恒定的吞吐量，但这并不适用于大多数面向消费者的应用程序。因此，TGS 速率限制器在这种情况下无法有效发挥作用。(2) 在内存超额订阅的情况下，TGS 强制低优先级作业通过 DMA 直接访问主机内存，而不进行迁移，实际上使计算陷入停滞。

案例 #2：多智能体系统。多智能体系统正日益成为组织复杂大语言模型应用程序的常见方式。它不再依赖单一的整体式模型，而是由规划器智能体将用户请求分解为子任务，并将每个子任务分派给专门的工作器模型。这种规划器-工作器编排模式常见于当今的编程助手、工具使用智能体以及许多近期的多大语言模型工作流中，是一种典型的多智能体工作负载 [12, 15, 33]。

我们使用开源多智能体框架 KVCOMM [38]来构建这一模型编排工作负载。在路由方面，我们使用 Qwen3 14B-Q8 来确定最佳下游模型。根据模型级基准测试 [5, 29, 31, 37]，，我们选择 Qwen3-VL 32B-Q6 作为数学专家模型，选择 Qwen3-Coder 30B-Q6 作为编程专家模型。所有模型均通过 llama.cpp 在单个 GPU 上提供服务；各配置之间唯一的区别是 GPU 共享机制。我们将 Nixie 与 nvshare 在 W=4 和 W=30下的表现进行比较。我们输入由数学请求和编程请求组成的混合流（从 GSM8K [8] 和 HumanEval-FIM [7]中随机打乱），并记录每个请求的端到端延迟，以及规划器和工作器组件各自的延迟细分。

图 12 展示了我们的结果。对于所有请求，Nixie 的速度都是 nvshare 两种配置的 1.6 倍。不同请求类型的性能提升也同样稳定：对于数学和编程任务，Nixie 的速度分别快 1.5×和 1.7× 。这些结果表明，Nixie 在多种应用程序中具有很强的灵活性，能够在单个共享GPU上实现高效的模型编排，同时不牺牲多智能体工作流的交互性。

案例3：长时间运行作业中的代码补全。基于大语言模型的代码补全是一项具有代表性的延迟敏感任务，需要即时响应。我们预计，Nixie 会自然地优先处理这类任务，而非长时间运行的作业，例如自主智能体操作或长时间的视频生成任务。为了评估 Nixie 的优先级处理行为，我们在长时间运行的智能体工作负载中执行代码补全。我们使用 Qwen3-Coder 30B-Q6 进行填充中间（FIM）补全，并使用 Gemma3 27B-Q8 模拟基于大语言模型的智能体。HumanEval-FIM [7] 用作代码补全的输入。由于代码补全请求的频率和间隔会因开发者个人情况及其工具实现而异，我们设置了三个场景来代表不同的使用模式：频繁（间隔 1 秒）、适中（间隔 3 秒）和稀疏（间隔 6 秒）。

图 13 展示了代码补全的响应时间和长时间运行的模型的吞吐量。对于频繁补全场景，Nixie 的平均响应时间为 1.8 秒；对于适中和稀疏场景，平均响应时间为 1.4 秒。

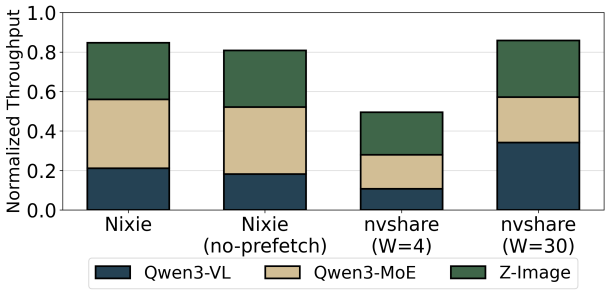


图 14：不同批处理作业

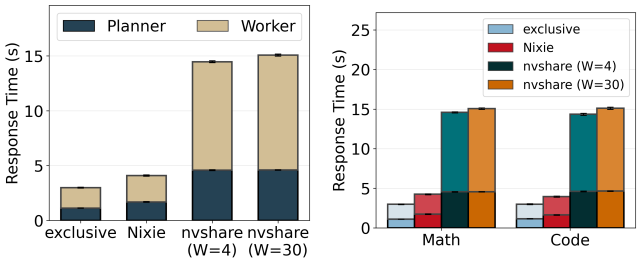
根据请求频率不同，nvshare（W=4）需要 4.5 至 5.7 秒才能完成交互式请求。与 nvshare（W=4）相比，Nixie 的速度提升了 3.1× 至3.8× 倍。当 nvshare 使用默认的 W=30 时，延迟超过 20 秒，使得大语言模型代码补全在真实世界的用例中难以实用。

就长时间运行作业吞吐量而言，在适中和稀疏场景下，Nixie 分别比 nvshare（W=4）高 90.6% 和 39.5%。在频繁场景下，Nixie 的吞吐量低 23.5%，这是意料之中的：Nixie 会优先调度交互式作业，并为代码补全分配更多 GPU 时间。TGS 在代码补全任务上取得了相近的性能，但无法让长时间运行的模型运行。实际上，TGS 退化为仅运行高优先级应用程序。

案例 #4：多个批处理任务。虽然 Nixie 面向交互式工作负载，但有些 GPU 作业负载足够重，必须长时间运行（例如，长文档处理、大批量生成）。因此，我们在多个批处理作业并发执行的设置下评估吞吐量和公平性。

我们运行三个作业：Qwen3-MoE 30B-Q6、Qwen3-VL 32B-Q6（批大小为 1 的长时间运行大语言模型请求）和 Z-image（批大小为 8）。我们还通过消融实验评估调度器中自动预取机制的有效性。我们将所提出的方案与配置为 W=4 和 W=30的 nvshare 进行比较。所有实验均持续 300 秒。为考虑不同应用程序指标的异质性，我们还单独运行每个应用程序作为参照，并报告相对于其经测量的独立运行性能的归一化吞吐量。

图 14 展示了结果。Nixie 的吞吐量与 nvshare 相当（W=30），达到理想吞吐量的 85%；即使 Nixie 采用自适应策略来优先保证响应性，而 nvshare 使用固定的较长时间片来提高吞吐量，亦是如此。在所有测试配置中，公平性都得到了有效维持。尽管在此场景中上下文切换已经不频繁，自动预取仍比不启用预取的变体额外提升了 5% 的吞吐量。相比之下，nvshare（W=4）的吞吐量仅达到 49.5%，这是频繁上下文切换所带来的开销所致。



(a) 总体延迟　(b) 按请求类型划分的延迟

图 15：RTX A5000 测试平台上的案例 #2。误差线表示标准差。

7.3 其他硬件上的性能

我们在另一台服务器上评估 Nixie。该服务器配备两颗 AMD EPYC 7352 24 核处理器、1TB 的 16 通道 DDR4 内存（速度为 3200 MT/s），以及通过 PCIe 4.0x16 连接的 8 块 RTX A5000 GPU（24GB）。该服务器运行 Ubuntu 22.04、CUDA 12.4 和 550.67 版 NVIDIA 驱动程序。

在此平台上，我们沿用**案例 #2**中的多智能体编排工作负载。鉴于 RTX A5000 的内存较小，我们使用 Qwen3 14B-Q4 作为规划器、Qwen3-VL 32B-Q4 作为数学工作器，并使用 Qwen3-Coder 30B-Q4 作为代码工作器。考虑到有更多可用图形处理器，我们在此加入独占模式变体。在此案例中，Nixie 的速度比 nvshare 快 3.4×，性能达到独占模式的 73%。基于 UVM 的方案在此实验设置下速度显著变慢。我们观察到该硬件上的 PCI Express 空闲度有所增加。我们怀疑根本原因是 NVIDIA 驱动程序较旧，以及相应较旧的 UVM 实现。

8 讨论

对其他操作系统的普适性。Nixie 利用 CUDA VMM API，该接口同时适用于 Linux 和 Windows。我们已在 Linux 上对 Nixie 进行了原型验证，并认为将 Nixie 移植到 Windows 是可行的；Windows 仍是消费者最广泛使用的操作系统。相比之下，统一虚拟内存不支持在 Windows 上进行内存超额订阅 [23]，而要在 Windows 的图形处理器驱动栈中添加这项支持，则需要投入大量工程精力。这一区别使 Nixie 成为更适合消费者平台的解决方案。此外，AMD GPU 也有类似的 VMM API [2]。

白盒方案。 Nixie 是一种完全透明的解决方案，无需了解张量语义即可运行。原则上，某些不可变数据（例如模型权重）可以直接在图形处理器上释放，而无需将其迁移回 CPU 内存，前提是应用程序保留单独的 CPU 端副本。例如，CUDA 内存复制本身会在 CPU 内存和图形处理器内存中留下重复数据。我们没有沿着这一方向继续研究，因为这将需要

了解应用程序数据结构的语义知识，从而损害透明性。然而，纳入此类语义提示是未来工作中一个有趣的方向。

与小型模型共置。 Nixie 的目标工作负载是每个应用程序都会使用大部分 GPU 内存，因此时间复用是唯一可行的方法。如果我们有一组小型模型，那么就可以进行空间复用，让不同模型同时执行，就像现有数据中心 GPU 复用工作一样 [14, 27, 41]。如何将空间复用集成到 Nixie 中，是一个很有前景的未来方向。

安全性。 Nixie Daemon 与应用程序共享相同的用户标识符，并且应用程序共享同一片 CPU 固定内存区域，用于存储迁移的 GPU 数据。恶意进程可能会访问其自身地址空间内的共享数据区域。然而，Linux 使用用户标识符来控制共享内存访问，这意味着相同用户标识符下的任何进程都可以访问 CPU 固定内存，无论这些内存是由所有进程共享，还是每个进程都有其专用的 CPU 固定内存。由于 Nixie 面向单用户环境而非多租户环境，因此这不属于我们的威胁模型。

9 相关工作

大语言模型时代之前的 GPU 复用。 在大语言模型兴起之前，GPU 复用已得到广泛研究。对于传统的计算机视觉工作负载，复用可通过在共享GPU上运行多个模型，大幅提高硬件整合程度（例如 Nexus [27],Shepherd [41], Clockwork [14]）。在这些系统中，GPU 内存会同时存储多个模型，而传入的请求会被批处理，并在一个或多个模型上执行。减少共置模型的数量或减小批大小，有助于缓解 GPU 内存压力。在这种场景下，主要挑战是最大限度地提高 GPU 计算效率。相比之下，Nixie 的目标是时间复用，而非空间复用：Nixie 不让 GPU 推理多个模型，而是让 GPU 在多个大型、内存密集型应用程序之间分时共享，这些应用程序的工作集接近 GPU 的全部容量。

大语言模型及其他大型基础模型时代的 GPU 复用。 大语言模型及其他大型基础模型从根本上重塑了 GPU 复用格局。注意力计算对内存的需求很高，尤其是在上下文较长时。即使是单个大语言模型推理，通常也会占用几乎全部可用的 GPU 内存。因此，近期研究已转向时间复用以及处理 GPU 内存超额订阅的技术。

PipeSwitch [6] 引入了一种感知模型、基于层的任务切换机制，但需要修改应用程序。此外，随着现代基础模型中迁移时间与模型执行时间之比不断增大，其效果会越来越有限。ServerlessLLM [11],Aegaeon [36], 和 Prism [39] 通过将访问频率较低的模型组件（例如，整个模型或 KV 缓存）卸载到 CPU 内存或磁盘，来应对超额订阅问题，主要是为了满足服务提供商的服务级别目标。这些系统面向数据中心 GPU 设计，其工作负载由单一服务提供商控制，并可直接集成到训练和推理技术栈中。然而，它们并不适用于消费级 GPU，因为这类 GPU 的工作负载种类繁多，应用程序不受集中管理，而且用户期待透明的解决方案。

改进统一虚拟内存。 统一虚拟内存为 GPU 内存超额订阅提供了透明的抽象，这颇具吸引力。DeepUM 等系统通过采用张量级预取来优化统一虚拟内存的性能。G 将磁盘纳入统一虚拟内存，并借助编译器管理流水线数据传输。这些解决方案需要更改硬件或 GPU 驱动程序。此外，Nixie 并非纯粹的虚拟内存系统。它不依赖缺页异常，能够同时协调计算与内存管理，从而缓解内存抖动。

GPU 调度。 已有多项研究探讨了多任务调度。XSched 提出了一个支持可抢占式 XPU 调度的框架。Autellix 将应用程序依赖关系纳入大语言模型调度。我们的调度策略相当简单，不需要任何应用程序级信息。我们的调度目标是优先处理交互式应用程序。

10 结论

消费级机器日益需要执行大型机器学习工作负载，例如大语言模型推理、文生图和交互式图像编辑。与数据中心环境不同，消费级 GPU 必须同时处理多个应用程序，而每个应用程序通常都会几乎占满 GPU 内存。我们构建了 Nixie，使消费级 GPU 能够实现高效且完全透明的时间复用，同时无需对应用程序或 GPU驱动程序进行任何修改。Nixie 将 GPU 内存分配与内核启动行为协调起来，以最大限度地减少抖动。Nixie 的调度器还通过自动识别对延迟敏感的应用程序并为其分配优先级，进一步提升了交互性。在相同的延迟约束下，Nixie 可将交互式代码补全工作负载的延迟最多降低 3.8×，同时将 CPU固定内存用量最多降低 66.8%。我们的源代码将公开提供。

参考文献

[1] CUDA 统一内存，2015年。访问时间：2025年10月。

[2] 超威半导体公司 虚拟内存管理。AMD，2025。HIP 7.1.52802 文档。

[3]Georgios Alexopoulos 和 Dimitris Mitropoulos。nvshare：不受显存容量限制的实用图形处理器共享。载于*2024* 年 *IEEE/ACM* 第 *46* 届软件工程国际会议：伴随论文集，ICSE-Companion '24，第 16–20 页，纽约，纽约州，美国，2024 年。计算机协会。

[4] Tyler Allen 和 Rong Ge。《通过深入的运行时和工作负载分析揭示 GPU 统一虚拟内存的开销》。载于*2021 IEEE* 国际并行与分布式处理研讨会（*IPDPS*），第 141–150 页。IEEE，2021年。[5] Shuai Bai、Yuxuan Cai、Ruizhe Chen、Keqin Chen、Xionghui Chen、Zesen Cheng、Lianghao Deng、Wei Ding、Chang Gao、Chunjiang Ge、Wenbin Ge、Zhifang Guo、Qidong Huang、Jie Huang、Fei Huang、Binyuan Hui、Shutong Jiang、Zhaohai Li、Mingsheng Li、Mei Li、Kaixin Li、Zicheng Lin、Junyang Lin、Xuejing Liu、Jiawei Liu、Chenglong Liu、Yang Liu、Dayiheng Liu、Shixuan Liu、Dunjie Lu、Ruilin Luo、Chenxu Lv、Rui Men、Lingchen Meng、Xuancheng Ren、Xingzhang Ren、Sibo Song、Yuchong Sun、Jun Tang、Jianhong Tu、Jian- qiang Wan、Peng Wang、Pengfei Wang、Qiuyue Wang、Yuxuan Wang、Tianbao Xie、Yiheng Xu、Haiyang Xu、Jin Xu、Zhibo Yang、Mingkun Yang、Jianxin Yang、An Yang、Bowen Yu、Fei Zhang、Hang Zhang、Xi Zhang、Bo Zheng、Humen Zhong、Jingren Zhou、Fan Zhou、Jing Zhou、Yuanzhi Zhu 和 Ke Zhu。《Qwen3-VL 技术报告》，2025年。[6] Zhihao Bai、Zhen Zhang、Yibo Zhu 和 Xin Jin。{PipeSwitch}：面向深度学习应用的快速流水线化上下文切换。载于第*14*届*USENIX*操作系统设计与实现研讨会（*OSDI 20*），第 499–514 页，2020年。[7] Mohammad Bavarian、Heewoo Jun、Nikolas Tezak、John Schulman、Christine McLeavey、Jerry Tworek 和 Mark Chen。《高效训练语言模型以补全中间内容》，2022年。[8] Karl Cobbe、Vineet Kosaraju、Mohammad Bavarian、Mark Chen、Heewoo Jun、Lukasz Kaiser、Matthias Plap- pert、Jerry Tworek、Jacob Hilton、Reiichiro Nakano、Christopher Hesse 和 John Schulman。《训练验证器以解答数学应用题》，2021年。[9] comfyanonymous。ComfyUI：功能最强大、模块化程度最高的扩散模型图形用户界面、应用程序接口和后端采用图/节点界面。https://github.com/comfyanonymous/ComfyUI，2025。

[10] Siddhartha Datta、Alexander Ku、Deepak Ramachan- dran 和 Peter Anderson。用于自适应文生图的提示扩展。载于 Lun-Wei Ku、Andre Martins 和 Vivek Srikumar（编），计算机语言学协会第 *62* 届年会论文集（第 *1* 卷：长篇论文），第 3449–3476 页，泰国曼谷，2024 年 8 月。计算机语言学协会。

[11] Yao Fu、Leyang Xue、Yeqi Huang、Andrei-Octavian Brabete、Dmitrii Ustiugov、Yuvraj Patel 和 Luo Mai。ServerlessLLM：面向大型语言模型的低延迟无服务器推理。载于 第*18*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 24*），第135–153页，加利福尼亚州圣克拉拉，2024年7月。USENIX协会。[12] Yingqiang Ge、Wenyue Hua、Kai Mei、Jianchao Ji、Jun- tao Tan、Shuyuan Xu、Zelong Li 和 Yongfeng Zhang。OpenAGI：当大语言模型遇到领域专家。载于神经信息处理系统进展（*NeurIPS*），2023年。[13]Georgi Gerganov。llama.cpp：C/C++ 大语言模型推理。https://github.com/ggerganov/llama.cpp，2025。[14] Arpan Gujarati、Reza Karimi、Safya Alzayat、Wei Hao、Antoine Kaufmann、Ymir Vigfusson 和 Jonathan Mace。像 Clockwork 一样服务深度神经网络：自底向上的性能可预测性。载于 第*14*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第443–462页。USENIX协会，2020年11月。[15] Sirui Hong、Mingchen Zhuge、Jonathan Chen、Xiawu Zheng、Yuheng Cheng、Jinlin Wang、Ceyao Zhang、Zili Wang、Steven Ka Shing Yau、Zijuan Lin、Liyang Zhou、Chenyu Ran、Lingfeng Xiao、Chenglin Wu 和 Jürgen Schmidhuber。MetaGPT：面向多智能体协作框架的元编程。载于 第十二届国际学习表征会议，2024年。[16] Jaehoon Jung、Jinpyo Kim 和 Jaejin Lee。Deepum：统一内存中的张量迁移与预取。载于第*28*届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第*2*卷，第207–221页，2023年。[17] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph Gonzalez,

Hao Zhang 和 Ion Stoica。采用分页注意力机制的大型语言模型服务高效内存管理。载于 第*29*届操作系统原理研讨会论文集，第611–626页，2023年。

[18] Black Forest Labs、Stephen Batifol、Andreas Blattmann、Frederic Boesel、Saksham Consul、Cyril Diagne、Tim Dockhorn、Jack English、Zion English、Patrick Esser、Sumith Kulal、Kyle Lacey、Yam Levi、Cheng Li、Do- minik Lorenz、Jonas Müller、Dustin Podell、Robin Rom- bach、Harry Saini、Axel Sauer 和 Luke Smith。FLUX.1 Kontext：潜在空间中用于上下文图像生成与编辑的流匹配，2025。

[19] Long Lian、Boyi Li、Adam Yala 和 Trevor Darrell。大语言模型引导的扩散：利用大型语言模型提升文本到图像扩散模型的提示理解能力，2024年。

[20] Michael Luo、Xiaoxiang Shi、Colin Cai、Tianjun Zhang、Justin Wong、Yichuan Wang、Chi Wang、Yanping Huang、Zhifeng Chen、Joseph E Gonzalez 等。Autellix：面向大语言模型智能体（作为通用程序）的高效服务引擎。*arXiv* 预印本 *arXiv:2502.13965*，2025。

[21]mostlygeek。llama-swap：llama.cpp 的模型切换，2025。用于自动模型切换的轻量级透明代理服务器。

[22] NVIDIA 公司。nvbandwidth：用于测量 NVIDIA GPU 带宽的工具。https://github.com/NVIDIA/nvbandwidth，2025。访问日期：2025-11-23。

[23] NVIDIA 公司。*Windows*、*WSL* 和 *Tegra* 上的统一内存。NVIDIA 公司，2025。CUDA 编程指南，版本 13.1。

[24] NVIDIA 公司。虚拟内存管理。NVIDIA 公司，2025。CUDA 驱动程序 API 文档，CUDA 工具包 v13.1.0。

[25] Ollama 贡献者。Ollama。https://github.com/ollama/ollama，2025。

[26] Yuang Peng、Yuxin Cui、Haomiao Tang、Zekun Qi、Run- pei Dong、Jing Bai、Chunrui Han、Zheng Ge、Xiangyu Zhang 和 Shu-Tao Xia。DreamBench++：面向个性化图像生成、与人类认知对齐的基准。在第十三届国际学习表征会议，2025。

[27] Haichen Shen、Lequn Chen、Yuchen Jin、Liangyu Zhao、Bingyu Kong、Matthai Philipose、Arvind Krishnamurthy 和 Ravi Sundaram。Nexus：用于加速基于深度神经网络的视频分析的图形处理器集群引擎。载于 第*27*届 *ACM* 操作系统原理研讨会，SOSP '19，第322–337页，美国纽约州纽约，2019年。计算机协会。

[28] Weihang Shen、Mingcong Han、Jialong Liu、Rong Chen 和 Haibo Chen。 {XSched}：适用于多种 {XPUs} 的抢占式调度。载于 第*19*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 25*），第671–692页，2025年。

[29] Gemma Team、Aishwarya Kamath、Johan Ferret、Shreya Pathak、Nino Vieillard、Ramona Merhej、Sarah Perrin、Tatiana Matejovicova、Alexandre Ramé、Morgane Rivière、Louis Rouillard、Thomas Mesnard、Geoffrey Cideron、Jean bastien Grill、Sabela Ramos、Edouard Yvinec、Michelle Casbon、Etienne Pot、Ivo Penchev、Gaël Liu、Francesco Visin、Kathleen Kenealy、Lucas Beyer、Xiaohai Zhai、Anton Tsitsulin、Robert Busa- Fekete、Alex Feng、Noveen Sachdeva、Benjamin Cole- man、Yi Gao、Basil Mustafa、Iain Barr、Emilio Parisotto、David Tian、Matan Eyal、Colin Cherry、Jan-Thorsten Pe- ter、Danila Sinopalnikov、Surya Bhupatiraju、Rishabh Agarwal、Mehran Kazemi、Dan Malkin、Ravin Ku- mar、David Vilar、Idan Brusilovsky、Jiaming Luo、An- dreas Steiner、Abe Friesen、Abhanshu Sharma、Ab- heesht Sharma、Adi Mayrav Gilady、Adrian Goedecke- meyer、Alaa Saade、Alex Feng、Alexander Kolesnikov、Alexei Bendebury、Alvin Abdagic、Amit Vadi、András György、André Susano Pinto、Anil Das、Ankur Bapna、Antoine Miech、Antoine Yang、Antonia Paterson、Ashish Shenoy、Ayan Chakrabarti、Bilal Piot、Bo Wu、Bobak Shahriari、Bryce Petrini、Charlie Chen、Charline Le Lan、Christopher A. Choquette-Choo、CJ Carey、Cor- mac Brick、Daniel Deutsch、Danielle Eisenbud、Dee Cat- tle、Derek Cheng、Dimitris Paparas、Divyashree Shiv- akumar Sreepathihalli、Doug Reid、Dustin Tran、Dustin Zelle、Eric Noland、Erwin Huizenga、Eugene Kharitonov、Frederick Liu、Gagik Amirkhanyan、Glenn Cameron、Hadi Hashemi、Hanna Klimczak-Pluci´nska、Harman Singh、Harsh Mehta、Harshal Tushar Lehri、Hussein Haz- imeh、Ian Ballantyne、Idan Szpektor、Ivan Nardini、Jean Pouget-Abadie、Jetha Chan、Joe Stanton、John Wiet- ing、Jonathan Lai、Jordi Orbay、Joseph Fernandez、Josh Newlan、Ju yeong Ji、Jyotinder Singh、Kat Black、Kathy Yu、Kevin Hui、Kiran Vodrahalli、Klaus Greff、Linhai Qiu、Marcella Valentine、Marina Coelho、Marvin Ritter、Matt Hoffman、Matthew Watson、Mayank Chaturvedi、Michael Moynihan、Min Ma、Nabila Babar、Natasha Noy、Nathan Byrd、Nick Roy、Nikola Momchev、Nilay Chauhan、Noveen Sachdeva、Oskar Bunyan、Pankil Bo- tarda、Paul Caron、Paul Kishan Rubenstein、Phil Culliton、Philipp Schmid、Pier Giuseppe Sessa、Pingmei Xu、Piotr Stanczyk、Pouya Tafti、Rakesh Shivanna、Renjie Wu、Renke Pan、Reza Rokni、Rob Willoughby、Rohith Vallu、Ryan Mullins、Sammy Jerome、Sara Smoot、Ser- tan Girgin、Shariq Iqbal、Shashir Reddy、Shruti Sheth、Siim Põder、Sijal Bhatnagar、Sindhu Raghuram Panyam、Sivan Eiger、Susan Zhang、Tianqi Liu、Trevor Yacovone、

Tyler Liechty、Uday Kalra、Utku Evci、Vedant Misra、Vincent Roseberry、Vlad Feinberg、Vlad Kolesnikov、Woohyun Han、Woosuk Kwon、Xi Chen、Yinlam Chow、Yuvein Zhu、Zichuan Wei、Zoltan Egyed、Victor Cotruta、Minh Giang、Phoebe Kirk、Anand Rao、Kat Black、Nabila Babar、Jessica Lo、Erica Moreira、Luiz Gus- tavo Martins、Omar Sanseviero、Lucas Gonzalez、Zach Gleicher、Tris Warkentin、Vahab Mirrokni、Evan Sen- ter、Eli Collins、Joelle Barral、Zoubin Ghahramani、Raia Hadsell、Yossi Matias、D. Sculley、Slav Petrov、Noah Fiedel、Noam Shazeer、Oriol Vinyals、Jeff Dean、Demis Hassabis、Koray Kavukcuoglu、Clement Farabet、Elena Buchatskaya、Jean-Baptiste Alayrac、Rohan Anil、Dmitry、Lepikhin、Sebastian Borgeaud、Olivier Bachem、Armand Joulin、Alek Andreev、Cassidy Hardin、Robert Dadashi 和 Léonard Hussenot。《Gemma 3 技术报告》，2025。

[30] Image Team、Huanqia Cai、Sihan Cao、Ruoyi Du、Peng Gao、Steven Hoi、Zhaohui Hou、Shijie Huang、Dengyang Jiang、Xin Jin、Liangchen Li、Zhen Li、Zhong-Yu Li、David Liu、Dongyang Liu、Junhan Shi、Qilong Wu、Feng Yu、Chi Zhang、Shifeng Zhang 和 Shilin Zhou。《Z-Image：基于单流扩散变换器的高效图像生成基础模型》，2025。

[31]Qwen Team。《Qwen3-Coder-30B-A3B-Instruct》。https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct，2025。

[32] Team Wan、Ang Wang、Baole Ai、Bin Wen、Chao-jie Mao、Chen-Wei Xie、Di Chen、Feiwu Yu、Haim-ing Zhao、Jianxiao Yang、Jianyuan Zeng、Jiayu Wang、Jingfeng Zhang、Jingren Zhou、Jinkai Wang、Jixuan Chen、Kai Zhu、Kang Zhao、Keyu Yan、Lianghua Huang、Mengyang Feng、Ningyi Zhang、Pandeng Li、Pingyu Wu、Ruihang Chu、Ruili Feng、Shiwei Zhang、Siyang Sun、Tao Fang、Tianxing Wang、Tianyi Gui、Tingyu Weng、Tong Shen、Wei Lin、Wei Wang、Wei Wang、Wen-meng Zhou、Wente Wang、Wenting Shen、Wenyuan Yu、Xianzhong Shi、Xiaoming Huang、Xin Xu、Yan Kou、Yangyu Lv、Yifei Li、Yijing Liu、Yiming Wang、Yingya Zhang、Yitong Huang、Yong Li、You Wu、Yu Liu、Yulin Pan、Yun Zheng、Yuntao Hong、Yupeng Shi、Yutong Feng、Zeyinzi Jiang、Zhen Han、Zhi-Fan Wu 和 Ziyu Liu。《Wan：开放且先进的大规模视频生成模型》，2025。

[33] Qian Wang、Tianyu Wang、Zhenheng Tang、Qinbin Li、Nuo Chen、Jingsheng Liang 和 Bingsheng He。《MegaAgent：一种无需预定义标准操作规程的大规模基于大语言模型的多智能体系统》。载于计算语言学协会论文集：*ACL 2025*年研究成果，第 4998–5036 页，奥地利维也纳，2025年7月。计算语言学协会。

[34] Bingyang Wu、Zili Zhang、Zhihao Bai、Xuanzhe Liu 和 Xin Jin。《面向深度学习工作负载的容器云透明图形处理器共享》。载于第 *20* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 23*），第 69–85 页，美国马萨诸塞州波士顿，2023年4月。USENIX协会。

[35] Chenfei Wu、Jiahao Li、Jingren Zhou、Junyang Lin、Kaiyuan Gao、Kun Yan、Sheng ming Yin、Shuai Bai、Xiao Xu、Yilei Chen、Yuxiang Chen、Zecheng Tang、Zekai Zhang、Zhengyi Wang、An Yang、Bowen Yu、Chen Cheng、Dayiheng Liu、Deqing Li、Hang Zhang、Hao Meng、Hu Wei、Jingyuan Ni、Kai Chen、Kuan Cao、Liang Peng、Lin Qu、Minggang Wu、Peng Wang、Shuting Yu、Tingkun Wen、Wensen Feng、Xiaoxiao Xu、Yi Wang、Yichang Zhang、Yongqiang Zhu、Yujia Wu、Yuxuan Cai 和 Zenan Liu。《Qwen-Image 技术报告》，2025。

[36] Yuxing Xiang、Xue Li、Kun Qian、Yufan Yang、Diwen Zhu、Wenyuan Yu、Ennan Zhai、Xuanzhe Liu、Xin Jin 和 Jingren Zhou。《Aegaeon：面向市场的并发大语言模型服务的有效图形处理器池化》。载于 *ACM SIGOPS* 第 *31* 届操作系统原理研讨会论文集，SOSP '25，第 1030–1045 页，纽约，纽约州，美国，2025。计算机协会。

[37] An Yang、Anfeng Li、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、Chang Gao、Chen- gen Huang、Chenxu Lv、Chujie Zheng、Dayiheng Liu、Fan Zhou、Fei Huang、Feng Hu、Hao Ge、Haoran Wei、Huan Lin、Jialong Tang、Jian Yang、Jianhong Tu、Jian- wei Zhang、Jianxin Yang、Jiaxi Yang、Jing Zhou、Jingren Zhou、Junyang Lin、Kai Dang、Keqin Bao、Kexin Yang、Le Yu、Lianghao Deng、Mei Li、Mingfeng Xue、Mingze Li、Pei Zhang、Peng Wang、Qin Zhu、Rui Men、Ruize Gao、Shixuan Liu、Shuang Luo、Tianhao Li、Tianyi Tang、Wenbiao Yin、Xingzhang Ren、Xinyu Wang、Xinyu Zhang、Xuancheng Ren、Yang Fan、Yang Su、Yichang Zhang、Yinger Zhang、Yu Wan、Yuqiong Liu、Zekun Wang、Zeyu Cui、Zhenru Zhang、Zhipeng Zhou 和 Zihan Qiu。《Qwen3 技术报告》，2025。

[38] Hancheng Ye、Zhengqi Gao、Mingyuan Ma、Qinsi Wang、Yuzhe Fu、Ming-Yu Chung、Yueqian Lin、Zhijian Liu、Jianyi Zhang、Danyang Zhuo 和 Yiran Chen。《KV-COMM：面向高效基于大语言模型的多智能体系统的在线跨上下文 KV 缓存通信》，2025。

[39] Shan Yu、Jiarong Xing、Yifan Qiao、Mingyuan Ma、Yangmin Li、Yang Wang、Shuo Yang、Zhiqiang Xie、Shiyi Cao、Ke Bao、Ion Stoica、Harry Xu 和 Ying Sheng。《Prism：释放图形处理器共享潜力，实现经济高效的多大语言模型服务》，2025。

[40] Haoyang Zhang、Yirui Zhou、Yuqi Xue、Yiqi Liu 和 Jian Huang。《G10：通过智能张量迁移实现高效统一的图形处理器内存与存储架构》。载于第*56*届*IEEE/ACM*微体系结构国际研讨会论文集，第395–410页，2023年。

[41] Hong Zhang、Yupeng Tang、Anurag Khandelwal 和 Ion Stoica。SHEPHERD：在实际环境中提供深度神经网络服务。发表于第*20*届*USENIX*网络系统设计与实现研讨会（*NSDI 23*），第787–808页，美国马萨诸塞州波士顿，2023年4月。USENIX协会。

[42]Lianmin Zheng、Liangsheng Yin、Zhiqiang Xie、Chuyue Livia Sun、Jeff Huang、Cody Hao Yu、Shiyi Cao、Christos Kozyrakis、Ion Stoica、Joseph E Gonzalez 等。Sglang：高效执行结构化语言模型程序。神经信息处理系统进展，37:62557–62583，2024年。