借助内存操作卸载，重新审视进程间通信

Misun Park†、Richi Dubey†、Yifan Yuan‡、Nam Sung Kim§、Ada Gavrilovska†佐治亚理工学院†、Meta‡、伊利诺伊大学厄巴纳-香槟分校§misun@gatech.edu、richidubey@gatech.edu、yifanyuan@meta.com、nskim@illinois.edu、ada@cc.gatech.edu

arXiv:2601.06331v1 [cs.OS] 9 Jan 2026

摘要—随着多模态和人工智能驱动的服务每个请求交换数百兆字节的数据，现有 IPC 运行时花在内存复制上的 CPU 周期占比越来越高。尽管硬件和软件机制都在探索内存卸载，当前 IPC 协议栈却缺乏统一的运行时模型，无法有效协调这些机制。本文提出 ROCKET——一套将基于硬件和软件的内存卸载集成到共享内存通信中的进程间通信运行时套件。ROCKET 系统地刻画了卸载策略与进程间通信执行之间的交互，涵盖同步、缓存可见性和并发，并引入不同的进程间通信模式，以平衡吞吐量、延迟和 CPU 效率。通过异步流水线处理、选择性缓存注入和混合协调，ROCKET 将卸载从特定于设备的功能转变为通用的系统能力。在真实工作负载上的评估表明，ROCKET 将指令数最多降低 22%，将吞吐量最多提升 2.1×，并将延迟最多降低 72%，这证明了协调式进程间通信卸载能够为现代数据密集型系统带来切实的端到端效率提升。

I. 引言

尽管计算加速取得了巨大进展，现代人工智能和数据分析流水线的瓶颈日益不在计算，而在于进程间数据移动。新兴的多模态工作负载，例如视频、图文或表格数据处理，通常每个请求都会交换数百兆字节的数据 [2]，[3], [22]。例如，批处理 256 张 RGB 图像（224×224，fp32）时，每次离线推理的数据移动量就已超过一百 MB，而多模态或视觉分析流水线会进一步增加这些传输量。单张 4K 图像就超过 30MB [8]，，多模态流水线的请求规模则更大。这些数值对应于人工智能推理流水线、大规模图查询和表格数据处理工作负载中常见的输入规模 [13], [15], [30], [32]。表 I 总结了这一趋势，展示了若干典型多模态工作负载的特征。这些工作负载如今涉及进程间 - 进程传输 50-500 MB，使内存复制成为主要的延迟和能源成本。

先前研究指出，即使在人工智能兴起之前的数据中心工作负载中，节点内数据传输（内存移动）也会消耗超过 5% 的 CPU 周期，造成了通常所说的“数据中心税”[21]，，并产生缓存污染和能耗开销等次生成本 [21]。随着多模态人工智能日益普及，数据密集型工作负载持续增长 [12]，，使得高效的进程间通信（IPC）机制以及数据中心内部的内存复制

表 I：刻画应用程序流水线进程间通信场景中代表性多模态工作负载的数据传输和内存行为的指标。

方面	MobileNetV2 XGBoost		PageRank	MilvusDB
字节数 (请求/响应)	120MB/ 800KB	25MB/ 800KB	76MB/ 320KB	1MB/ 320MB
内存复制时间 在进程间通信中（毫秒）	203	25	46	323
内存行为	大输入，低重用率	低重用率，计算受限	高重用率，内存受限	臃肿响应
配置	批量大小 200 [35]	20 万行，30 个特征	图，带有 1000 万条边 [25]	批处理搜索

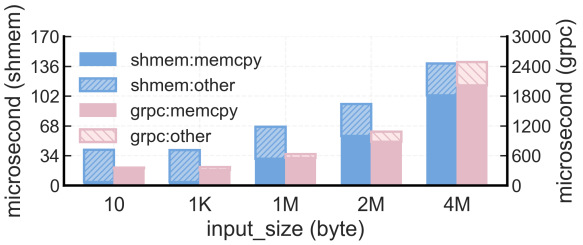


图 1：使用共享内存（shmem）和 gRPC 实现的节点内回显 RPC 的端到端延迟分解。该图量化了总延迟中归因于内存复制的部分随消息大小变化的情况。

节点发挥着关键作用，有助于实现可扩展的高性能系统架构。图 1 显示，内存复制时间在优化实现（即基于共享内存的实现，例如 Nightcore [19]）和通用 IPC 协议栈（如 gRPC）的执行时间中都占主导地位，即使数据大小适中也是如此。随着数据传输大小增加，降低这些内存数据传输成本可显著提升数据中心效率。

现有的进程间通信协议栈围绕 CPU 驱动的内存复制设计，在不耗尽核心资源和缓存层次结构的情况下，无法持续应对如此庞大的传输量。计算与通信效率之间日益扩大的差距，促使人们需要一类新的、支持卸载的进程间通信运行时。新兴的数据加速硬件引擎 [5] 为将这些传输从 CPU 卸载创造了机会，从而释放 CPU 资源以供应用任务使用，并减少内存开销造成的停顿。近期的系统级研究 [14] 表明，即使是集成在操作系统服务中的软件卸载机制，也能提升复制性能，这凸显了以下方面日益重要：

硬件和软件层的卸载感知设计。

以往关于内存卸载的研究，主要通过微基准测试和数据库工作负载考察进程内 数据移动 [4], [6], [23], ，也研究了节点间 场景下用于网络栈加速的方法 [23]。然而，将此类卸载集成到进程间通信协议栈中，会因缓存干扰和同步开销而带来微妙的权衡，现有系统难以应对这些问题。这在进程间、节点内场景中至关重要——这类场景是客户端-服务器交互 [11] 和新兴数据处理流水线 [2], [3], [22] 的核心——但仍未得到充分探索。

将内存加速器集成到进程间数据移动中，会带来新的性能隐患。大多数进程间通信系统仍依赖同步通信语义，这限制了硬件卸载所能提供的并行性。若缺乏针对这类加速器设计的方案，naïve 集成往往会破坏缓存局部性、扰乱内存访问模式，并加剧页错误和地址转换开销。这些影响非但无法加速执行，反而可能抵消卸载带来的收益，甚至降低整体性能。

本研究探讨数据移动加速器如何集成到多进程应用的共享内存 IPC 流水线中。英特尔 DSA 为研究这种集成在真实内存密集型工作负载中的表现提供了具体的观察对象。我们分析了影响先进 IPC 运行时中加速器效率的系统级因素，指出卸载何时能够提升性能、何时又会引入新的瓶颈。分析揭示了决定硬件卸载与软件契合程度的关键因素，包括缓存行为、同步成本、争用、缺页和执行模型。

基于这些发现，我们设计了 ROCKET，这是一种沿这些维度优化加速器交互的 IPC 软件栈。结果表明，性能对集成方案的选择十分敏感：ROCKET 将指令数最多降低了 22%，并将吞吐量提升了 15%。

本研究的贡献可概括如下：**(1) 系统级瓶颈分析**。我们找出了影响用户空间进程间通信中基于硬件的内存卸载的关键性能因素，包括 CPU 使用率、缓存行为和同步开销。这项分析揭示了限制加速器在实际部署中发挥效能的根本瓶颈。（§III）

（2）协议设计与实现。我们设计并实现了一套共享内存进程间通信协议——ROCKET，将内存卸载集成到用户空间流水线中。该设计体现了同步模型、执行模式和缓存行为等关键集成维度。ROCKET 通过异步流水线处理、并行性和缓存注入提升效率，同时利用 x86 上的省电指令。（第 IV 节）

(3) 实证评估。我们在配备英特尔 DSA 卸载引擎的系统上，使用真实工作负载评估 ROCKET。与 CPU 基线相比，ROCKET 的指令数最多减少 22%，吞吐量提高 15%，表现优于现有软件支持。这些提升不仅限于内存复制加速，还得益于卸载感知执行路径和注重集成的流水线结构设计。（§VI）

本研究既为内存引擎提供了实用的进程间通信设计，也系统地阐释了用户空间部署中的架构权衡，为数据密集型环境中的未来内存加速系统奠定了基础。

II. 背景：硬件辅助内存卸载

现代系统日益致力于将数据移动与 CPU 执行解耦。传统的基于 memcpy 的传输虽然简单且通用，但在带宽密集型工作负载下代价高昂，因为每次复制都会污染缓存并消耗宝贵的 CPU 周期。为缓解这些低效问题，硬件和操作系统如今将内存卸载视为一种可编程服务，使大规模传输能够在 CPU 的关键路径之外进行。从内核管理的异步复制引擎到英特尔 DSA 等集成加速器，近期的诸多进展都体现了这一转变，并促使我们进一步探索运行时系统中的软硬件集成。

A. 内存卸载的发展现状

近年来，内存卸载已成为系统设计中的重要趋势。复制操作曾被视为简单的受中央处理器限制的库调用，如今在数据密集型工作负载中却成为关键的性能瓶颈。缓存干扰、同步延迟和 CPU 周期浪费限制了可扩展性。为解决这些问题，现代系统越来越多地将复制操作委托给专用硬件或内核级服务。

在硬件层面，近期研究在内存控制器或直接内存访问引擎处执行复制 [20], [34], ，从而减少 CPU 停顿，并使传输能够与计算重叠。在内核和系统层面，复制操作被重新视为协同服务，并委托给指定线程 [14]。这些系统会跟踪复制请求之间的依赖关系，并重叠复制使用阶段，以提高吞吐量。总体而言，它们旨在将数据移动变为一种一等的、可调度的操作。然而，这些工作主要关注启用内存卸载，而非将其集成到应用级数据流中。由此产生的差距——硬件或内核机制所能实现的功能与端到端系统实际利用的功能之间的差距——正是 Rocket 发挥作用的领域。我们的目标是阐明这一缺失的软件层：一个连接支持卸载的硬件与用户空间进程间通信的运行时。

表 II 总结了这些进展及其在硬件层、内核层和系统层之间的权衡，并强调缓存管理、同步和策略灵活性仍是关键设计约束。

B. 案例研究：英特尔 *DSA*

现代内存卸载引擎的一个典型例子是 Intel DSA，它集成于 Sapphire Rapids 处理器中。DSA 在传统 DMA 的基础上扩展了用户级控制、虚拟内存支持和细粒度缓存管理。这些

表 I：各系统层中的代表性内存卸载机制。

	(MC)^2 [20]	DMX [34]	Copier [14]
层级	Mem. 控制器	跨加速器	操作系统内核
缓存污染	惰性刷新	栅栏 DMA	未处理
适用性	硬件-tied	专用配置	集成于操作系统
开销	极低	中等	中等-high

特征旨在减少缓存污染、释放 CPU 周期、实现计算与内存重叠，并提高带宽利用率。

虚拟内存支持。 与需要物理地址的传统直接内存访问不同，DSA 支持虚拟内存，简化了与用户空间应用程序的集成。它利用进程地址空间标识符（PASID）管理多个地址空间，无需显式锁定内存，因此非常适合虚拟化和多租户环境。

增强的可编程性。 DSA 的一项关键优势是可编程性更强。与依赖系统调用（例如 `ioctl`）和内核管理的描述符的传统直接内存访问引擎不同，DSA 允许通过专用 CPU 指令（例如 `ENQCMD`和 `MOVDIR64B`）直接从用户空间提交工作描述符。这样可以避免上下文切换和进入内核，将延迟从数十微秒 [10] 降低至数百个 CPU 周期（≈ 200纳秒）[16]。此外，`ENQCMD`以原子方式执行 [23]，，因此在多线程环境中无需使用锁。

C. DSA 软件栈

我们考察了使用 DSA 的两种方式：DSA 驱动程序提供的低层级接口，以及英特尔的 DSA 透明卸载（DTO）框架。前者为高级用户提供精细控制和低延迟路径；后者则通过拦截库调用优先考虑易用性，但会牺牲灵活性和性能。这些实现展示了当前 DSA 软件支持在可编程性、控制能力和性能之间的权衡。

低级编程接口。 DSA 编程模型包含三个主要步骤。首先，CPU 准备一个任务描述符，指定要卸载的内存操作，包括源地址和目标地址、传输大小以及完成标志地址。其次，通过低级入队指令将描述符提交给 DSA 设备。这一步需要直接与由`libaccel-config.h` 和 `linux/idxd.h` [16] 公开的特定于硬件的结构交互。第三，操作完成后，DSA 会设置该标志，CPU 可以通过轮询或异步方式监控

诸如 `UMWAIT` 这样的机制。虽然基于中断的完成被认为是效率最高的方法，但由于用户模式缺乏中断处理能力，因此无法使用。这种卸载模型减少了数据移动过程中 CPU 的参与，从而释放周期用于其他任务（图2）。

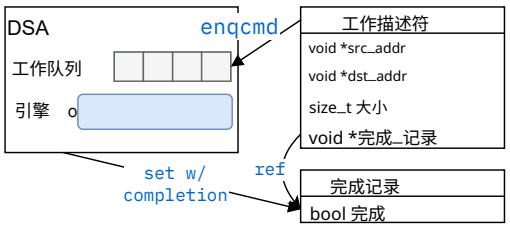


图 2：DSA 编程模型。CPU 准备任务描述符并将其提交给 DSA。DSA 执行任务并设置完成标志。随后，CPU 检查完成标志以确定任务是否已完成。

其他接口（例如存储性能开发套件（SPDK））提供了用于使用 DSA 的用户级库。然而，这些库通常只是上述核心编程模型的薄封装，并未从根本上改变任务提交或完成语义。

静态卸载。 英特尔的 DSA 透明卸载 (DTO) 框架允许应用程序在不修改源代码的情况下使用 DSA，方法是拦截诸如`memcpy()` 之类的标准库调用，并将其重定向到 DSA。这简化了采用过程，尤其适用于传统软件或闭源软件。

然而，DTO 存在两个关键局限。（1）它缺乏对卸载决策的细粒度控制：无论传输大小、局部性或重用距离如何，所有拦截到的调用都会以相同方式卸载。当使用基于 CPU 的 `memcpy()` 更合适时（例如，对于较小的延迟敏感型传输），这可能会损害性能。（2）DTO 强制采用同步执行模型，阻止 DSA 传输与 CPU 计算之间进行流水线处理或重叠，从而限制并行性。

这些限制使 DTO 易于采用，但不太适合对性能要求严苛的场景，因此促使人们探索替代性的编程模型。

III. 动机：卸载的 IPC 中的集成挑战

将内存卸载引入进程间通信会带来一些微妙的权衡，若贸然应用，可能会抵消其优势。卸载引擎减轻了 CPU 的数据移动负担，却将同步、可见性和缓存方面的责任转移给软件。这种相互作用暴露出三种反复出现的矛盾：（i）提交与完成之间的同步粒度；（ii）地址可见性和缺页处理；（iii）缓存注入和数据重用控制。要有效地将卸载机制集成到运行时系统中，并为第 IV 节提出的设计原则提供依据，理解这些矛盾至关重要。本节将以英特尔 DSA 为例，阐释并量化这些权衡。

A. 硬件层面的权衡

卸载内存操作可以释放 CPU 周期并减少缓存污染，从而有望提升整体性能。例如，卸载一次 1MB 传输可节省约 $33\mu s$，借助 DSA（≈130,000 个 CPU 周期可用于其他任务）。不过，这些收益取决于具体情境，若管理不当，也可能产生系统级开销。

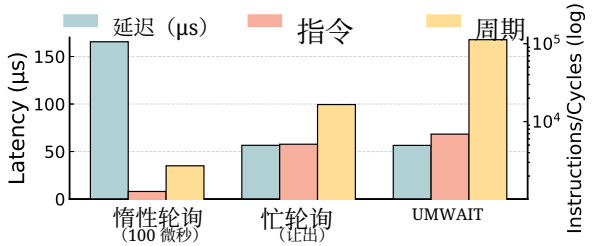


图 3：不同轮询策略在延迟和 CPU 使用率方面的比较（1MB 传输。惰性轮询：每 100μs 轮询一次；忙轮询：通过让出 CPU 进行轮询，但不休眠；UMWAIT：通过用户态中断进行轮询。）

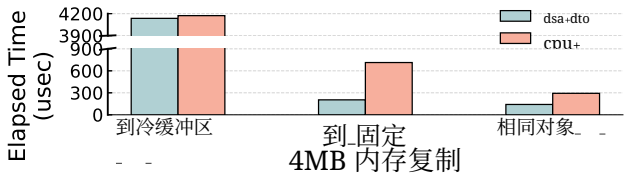


图 4：不同内存条件下 DSA 与 CPU 内存复制的性能比较。复制到固定缓冲区可将延迟降低 95%，而重复使用相同对象缓冲区可实现 97% 的降低；二者均以冷缓冲区访问为基准。

在进程间通信工作负载中，集成 DSA 会带来若干硬件层面的权衡。关键因素包括忙等待导致的同步开销、绕过 CPU 缓存造成的性能损失、缺页引发的延迟峰值，以及潜在的总线争用。接下来的章节将逐一探讨这些因素及其对系统行为的影响。

完成检查的开销。 无论 DSA 以同步还是异步方式使用，都必须通过读取不可缓存的内存映射 I/O 区域中的完成标志来检测卸载完成情况。

图 3 比较了三种轮询策略。忙等待可提供较低的延迟，但会消耗大量 CPU 周期。懒等待在延迟方面效率较低。`UMWAIT`可提供与忙等待相当的延迟，但并不提供真正的休眠或异步行为——它会将 CPU 置于浅等待状态，实际上以 25μs 的间隔进行轮询。在单线程环境中，它的主要优势是节能，而不是响应速度 [17]。

轮询会带来不可忽视的系统级开销。每次读取不可缓存的标志都会绕过 CPU 缓存并经过内存总线，从而增加争用。此外，对内存映射 I/O 区域的访问会施加严格的顺序约束，阻碍乱序执行并导致流水线停顿。这些访问还可能造成缓存和 TLB 失效，降低整体性能 [29], [37]。虽然内存卸载减少了 CPU 对数据移动的参与，但频繁轮询以实现同步可能抵消其带来的收益。这凸显了对低开销、响应迅速的同步机制的需求。在同步模式下使用 DSA 会增加同步开销并延长 CPU 空闲时间。异步模式可以降低这类开销，因此更适合高效卸载。不过，由于加速器执行的特性，完成检查无法完全避免，如何尽量降低其开销仍是一项关键挑战。

页错误的影响。 尽管 DSA 支持虚拟内存并将缺页处理延后至主机，缺页

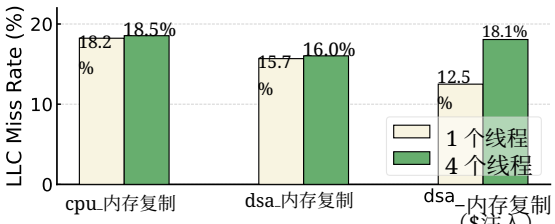


图 5：单线程和四线程执行下的末级缓存未命中率，对比 cpu_内存复制、dsa_内存复制、dsa_内存复制($inject)（微基准测试：内存复制后对目标缓冲区中的所有元素求和）。

仍会引入延迟，降低内存卸载到硬件时的性能。如图4所示，发生缺页时，DSA 相较于基于 CPU 的 内存复制 并无明显优势。相比之下，使用固定内存时，即使反复访问相同的缓冲区，DSA 的性能也显著优于 CPU。这些结果表明，频繁的缺页可能会阻碍 DSA 达到其理论峰值性能，凸显了预先映射或固定内存区域对于有效使用的重要性。

B. 缓存干扰和路径分歧：*CPU* 与 *DSA*

内存卸载改变了人们对内存层次结构中时间局部性的假设。与可受益于自动缓存保留的 CPU 内存操作不同，内存卸载默认绕过缓存，因此数据在短时间内再次使用时可能产生冷缓存效应。虽然这减少了缓存污染，却增加了近期访问的延迟。为缓解这一问题，DSA 支持显式缓存注入，可在传输期间将选定数据导入末级缓存。如图 5 所示，缓存注入能提高单线程工作负载的末级缓存命中率，但在多线程争用下可能降低性能。

为作说明，图6对比了不同的执行路径：CPU 内存复制可受益于缓存局部性和预取，而 DSA 会绕过缓存，可能导致冷缓存延迟。这些权衡凸显了应用级控制的必要性。DSA 软件支持不应采用静态的、一刀切的配置，而应提供可调接口，使开发者能够根据复用模式、访问局部性和进程间通信集成情况定制卸载方式。

按访问方向划分的执行影响。 我们总结了内存访问方向如何影响 DSA 卸载决策。

- **读入（DRAM → 缓存）：** CPU 将重复使用的数据加载到缓存中，以利用局部性。DSA 会绕过缓存，可能导致冷启动开销（图6a）。

- **写出（缓存 → DRAM）：** DSA 可避免写出数据污染缓存，这对临时数据有益。但对重复使用敏感的数据而言，绕过缓存可能导致未命中（图6b）。

虽然 DSA 有望加速内存操作，但其效果很大程度上取决于工作负载特征和系统级交互。表 III 总结了通过微基准测试发现的关键权衡，并概述了这些权衡对 ROCKET 的设计影响。这些见解为我们的执行模型提供了依据，该模型会根据传输大小、线程数和

表 I：DSA 卸载的权衡及其对系统设计的影响。每个因素都指出一项关键限制，以及 ROCKET 如何通过可配置或默认的设计决策来应对。

因素	观察到的权衡	微基准测试洞见	设计影响
数据大小	卸载并非总是有益 于小型传输	卸载 1MB 可节省 33 微秒；盈亏平衡点为 4KB 原始数据因设置开销而更高 [23]	小型传输应使用基于 CPU 的内存复制 ；并应用基于大小的阈值
缺页	DSA 支持虚拟内存 但性能会随着 缺页	缺页会抵消 DSA 加速效果； 固定内存可实现最佳性能 （图4）	复用共享内存以避免缺页故障； 强制执行预映射或锁页
缓存注入	可能改善或损害性能- 性能取决于重用时间—— ing	在单线程情况下提高命中率； 由于污染，多线程情况下性能下降 （图 5）	仅在争用程度较低时启用缓存注入 （例如单线程 同步/异步模式）
同步	频繁轮询会导致总线 争用和停顿	UMWAIT 可降低主动轮询开销，但 仅限于 25µs（图 3, [17]）	使用混合轮询（UMWAIT + 时间- 输出）；在流水线模式下推迟检查
并行性	除非在软件中显式编排，否则未能充分利用- 在软件中编排	idxd 中不支持并行执行；DTO 会阻塞 直至完成	强制采用结构化异步/流水线 执行以利用并发
软件 支持	现有工具缺乏智能 决策逻辑	idxd 提供底层控制；DTO 禁 用并行性	提供可调节的高级 API，支持 执行模式和缓存选项

复用模式有选择地应用内存卸载。第四节通过比较基于 CPU 和加速器的流水线中的数据移动，详细说明这些权衡如何影响进程间通信性能。

IV. 设计

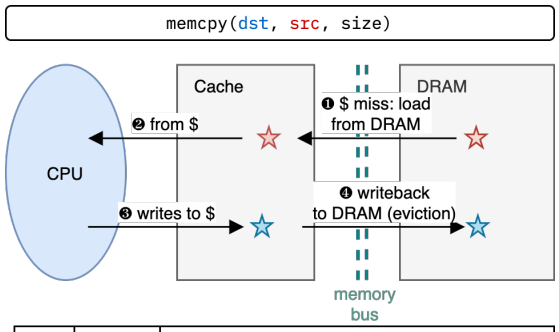
为了解决第三节中的挑战，我们提出了 ROCKET，这是一种运行时系统，将硬件辅助内存卸载集成到进程间通信套件中，用于基于共享内存的节点内通信。ROCKET 支持用户指定卸载，避免不必要的开销，并在适用时利用 DSA。它不采用固定策略，而是提供可配置的执行、同步和缓存行为模式，使进程间通信能够适应工作负载和硬件特征。这种灵活性使其能够适应动态环境。

A. 所提设计概述

整体执行流程和系统架构如图 7 所示。ROCKET 是一套基于共享内存的进程间通信套件，支持多客户端连接，并可选择性地将数据移动卸载到 DSA。服务器由消息队列、DSA引擎、请求分发器、请求处理器和查询处理器组成，可实现异步批处理，并高效地重叠 CPU 与 DSA 的工作。

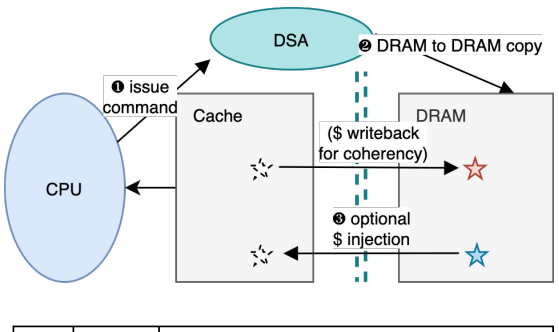
该系统设计遵循一系列硬件感知原则，这些原则源自第三节讨论的权衡：**(1) 共享内存复用。** 为减少重新映射开销并避免缺页，ROCKET 会在多次传输间复用共享内存段。为此，系统采用持久化的 消息队列和预分配的 共享内存 来保持缓冲区连续性。**(2) 灵活传输卸载。** ROCKET 提供接口，可根据数据大小和工作负载特征调整卸载决策。与静态卸载框架（例如 Intel DTO）不同，后者会静态地卸载所有传输；ROCKET 支持自适应策略，使用户能够权衡开销与延迟。

（3）内置支持 CPU-DSA 并行。 ROCKET 通过集成低



Source		Destination: Cache	Destination: DRAM (quick eviction)
	Cache	❷→❸	❷→❸→❹
	DRAM	❶→❷→❸	❶→❷→❸→❹

(a) 基于 CPU 的内存复制执行路径



Source		Destination: Cache	Destination: DRAM
	Cache	writeback→❶→❷→❸	writeback→❶→❷
	DRAM	❶→❷→❸	❶→❷

(b) 基于 DSA 的内存复制执行路径

图6：CPU 与 DSA 的内存复制执行路径对比。基于 CPU 的内存复制可自然融入缓存层次结构，而基于 DSA 的内存复制则绕过缓存，直接访问 DRAM。

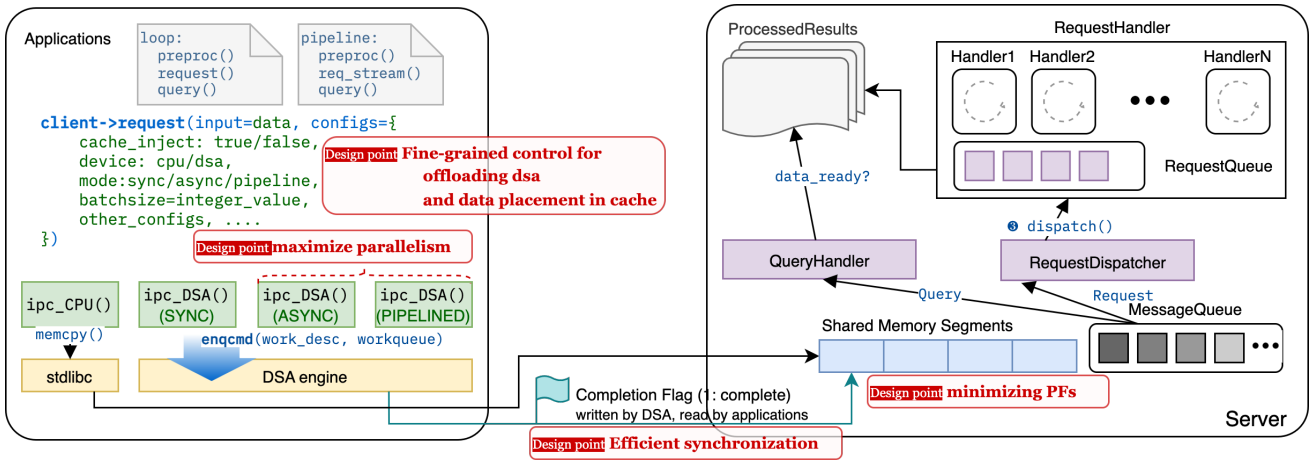


图 7：ROCKET 架构概览。其组件体现了关键设计原则，例如避免缺页、最大化并行性、可配置的选择性卸载（可启用或不启用缓存注入），以及高效同步。

层级 API 中通常留给用户处理的同步与协调机制，支持异步传输。这样，CPU 端执行便可与卸载至 DSA 的数据移动并发进行，这是 DTO 等仅支持同步的基线方案所不具备的能力。

(4) 缓存感知的数据放置。 ROCKET 将缓存注入作为一个 API 参数公开。在多客户端场景中，服务器共享执行上下文，使客户端能够根据复用可能性和系统负载选择性地启用注入。

(5) 降低同步开销。 ROCKET 通过延迟完成检查，避免在异步模式下频繁轮询。查询处理器 采用确定性时间预测，并基于UMWAIT等待，以实现低延迟协调。

这些原则共同体现在一个模块化的进程间通信栈中，该栈可适应不同的工作负载和系统级行为。

B. IPC API 规范

ROCKET 提供一组可配置的进程间通信 API，旨在适应各种工作负载，同时保持性能可移植性和易用性。该接口支持多种执行模式，并允许用户明确控制数据移动、同步和缓存注入策略。**执行模式。** ROCKET 支持三种执行模式——同步、异步和流水线式 （图 8）。每种模式在内存卸载、同步和 CPU-DSA 重叠方面各有取舍。用户可以根据工作负载特征和性能目标选择合适的模式。

- **同步模式：** 同步 模式（图 8a）以阻塞方式执行内存操作。发起线程会停顿，直至操作完成，因此适用于对延迟敏感的顺序任务。在缓存争用较低的场景中，可以启用缓存注入以减少冷缓存惩罚。

- **异步模式：** 异步 模式（图8b）将提交与完成解耦。请求会立即返回，必须显式检查是否完成。这样，CPU 可以在操作执行中时处理有用的工作，因此适用于中等并行度的工作负载，或能够隐藏数据传输延迟的场景。

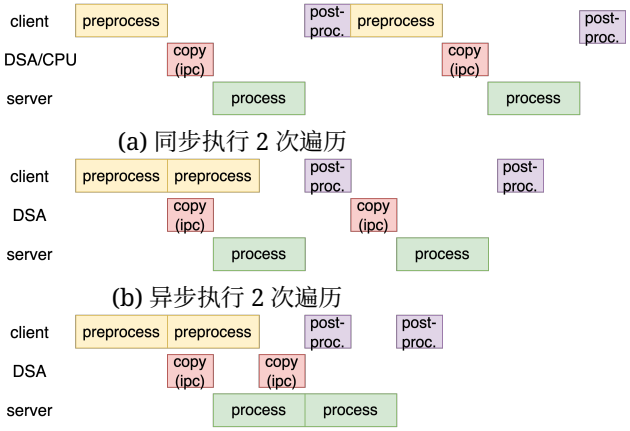


图 8：ROCKET 中的执行模式结构。各模式在同步方式和重叠策略上有所不同。

- **流水线模式：** 流水线 模式（图8c）以批次方式发起内存操作，以提高吞吐量并降低开销。它会延后对单个操作的完成检查，并在各阶段间重复使用缓冲区。当工作负载涉及大量内存流量，且预处理可以与其他操作重叠时，此模式效果良好。如果预处理占据了执行的大部分时间，其性能可能较差。

可配置参数。 所有模式都支持相同的功能接口，并提供用于配置执行行为的附加参数。为支持特定工作负载调优，API 提供了若干参数：

- **卸载控制：** 应用程序可以选择针对特定内存操作使用 DSA 或 CPU。这使得应用程序能够绕过 DSA，处理较小或无益的传输。

- **缓存注入策略：** 用户可以显式启用或禁用缓存注入。默认情况下，ROCKET 会根据经验性的性能趋势应用特定于执行模式的策略。

- **执行模式：** 应用程序可以根据其性能要求选择最合适的执行模型。

这些参数提供了细粒度控制，同时保留了合理的默认值，使用户能够按需选择最合适的内存卸载策略。

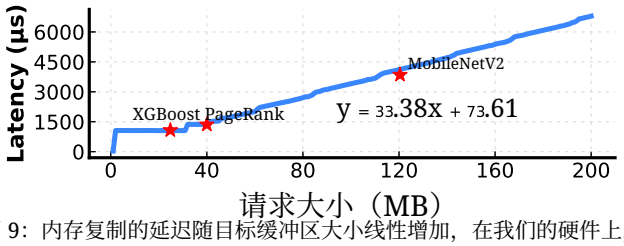


图 9：内存复制的延迟随目标缓冲区大小线性增加，在我们的硬件上，每 1MB 约为 33.4 μs。

C. 设计内部机制

共享内存区域复用。 ROCKET 使用持久化的共享内存区域来最大限度地降低缺页开销。建立连接时，服务器会分配一个固定大小的内存池，并为每个客户端分配一个专用队列对，以及发送（客户端到服务器）和接收（服务器到客户端）缓冲区。这些缓冲区只映射一次，并在整个会话期间重复使用。这样可以避免重复映射的开销，确保内存访问稳定且延迟低，从而实现高效的DSA传输。该设计的灵感来自 RDMA队列对，但针对基于DSA的复制语义进行了调整。

异步 DSA 引擎。 为实现并行并隐藏内存延迟，ROCKET 包含一个轻量级异步引擎，用于管理 DSA 命令分发和完成。它抽象了底层 DSA 原语，并为 IPC 驱动程序提供简洁的接口。请求会被路由至对应模式的处理路径；在流水线 模式下，引擎负责命令发出、完成跟踪和批处理。混合轮询策略在降低 CPU 开销与及时完成检查之间取得平衡。

完成跟踪与混合轮询策略。 频繁轮询可及时检测到完成，但会增加开销。相反，不频繁轮询可以降低开销，但可能导致响应延迟。为权衡这两种取舍，ROCKET 实现了一种混合轮询策略，将基于时间的延迟轮询与被动等待指令相结合。虽然UMWAIT可最大限度减少 CPU 参与，但其等待时间受限于 100K 个周期（≈25μs） [17]。

因此，ROCKET 实现了一种感知数据大小的延迟机制，根据请求数据大小估算预期完成时间，并据此延后轮询。设 L 为估计的延迟时间（微秒）： $L = L_{\text{fixed}} + \alpha \cdot$大小_单位_MB。 L_{fixed} 和 α 均取决于机器，但对于给定系统，在不同工作负载下保持一致（图 9）。ROCKET 还提供了一个性能分析脚本，可在初始部署期间自动推导这些参数。

在我们的实现中，我们基于实测结果使用 $L_{\text{fixed}} = 73.6\mu s$和$\alpha = 33.4\ \mu s$/MB。我们在不同的 L_{fixed} 和 α条件下重复进行了 100 次延迟测量（标准差为 $< 2\%$）。由于指令路径和内存复制引擎的传输函数均由硬件定义，复制延迟高度稳定。这些参数可能因机器而异，但在同一系统内保持一致。源代码中的辅助脚本会自动按节点重新校准这些参数，以确保可复现性。ROCKET 根据传输大小计算 L ，并休眠 $L \cdot 0.95$，以让出 CPU。随后，它使用UMWAIT开始轮询。这种混合策略减少了基于对延迟的感知预测，延后检查，直到很可能完成时再进行，从而减少不必要的轮询。轮询开始后，UMWAIT 可实现被动等待，而不会持续消耗 CPU。这种方法兼顾了延迟和效率，既避免了忙循环轮询的开销，又能及时检测到完成。它还使自适应运行时能够根据请求大小和系统负载动态调整轮询行为，在响应速度与效率之间权衡。**请求批处理支持。** 为实现高吞吐量，尤其是在流水线 模式下，ROCKET 支持应用层请求批处理。传入消息分为请求或结果查询。请求会分派给特定工作负载处理程序（例如MobileNetV2、图处理），并通过统一接口注册。处理程序异步执行，并将结果写入共享内存。通过将提交与完成解耦，ROCKET 支持延后收集结果和批处理。这提高了缓冲区复用率并减少了同步开销，使多个未完成请求能够以最少的协调共同处理。

执行栈内部机制。 如图 7 所示，ROCKET 将请求提交、执行和完成跟踪分离到不同组件中。请求分发器 从消息队列接收传入的请求，并将其分派给请求处理器，每个线程上下文都会实例化相应的处理器。处理器根据特定模式的逻辑（例如同步、异步、流水线）发出 DSA 命令。在流水线 模式下，请求会被批量处理，以最大限度地提高吞吐量并摊销开销。查询处理器 通过轮询共享内存中的结果标志来跟踪完成状态。在流水线 模式下，可显式调用该处理器。通过将完成跟踪与请求处理解耦，ROCKET 实现了可配置的同步，并支持通过用户态中断（UMWAIT）进行混合轮询。这些组件共同使 ROCKET 能够根据工作负载需求调整执行方式，同时将底层控制与面向应用程序的 API 隔离开来。

V. 实现

ROCKET 使用 C++17 实现，代码约 12,000 行。完整源代码及构建说明已在 GitHub 上公开，以支持可复现性和社区贡献。该系统基于英特尔 IDXD 驱动程序构建，并使用其最新版本与 DSA 硬件进行交互。系统在 IDXD 之上提供高级 API，在简化 DSA 使用方式的同时保留性能。使用 `accel-config` 库配置 DSA 设备。该实现利用 UMONITOR、UMWAIT 和 ENQCMD 等英特尔专用指令，以及用于高效执行的内建函数。标准功能使用 C++17、System V IPC 和 POSIX 库实现。基准工作负载中使用了 ONNX [27], XGBoost [7], BoostGL [31], MilvusDB [36], 和 OpenCV。

清单 1：用于并行执行模式的 ROCKET API

```
// 默认：同步执行（阻塞）客户端->请求（模式=“同步”，操作
=“mobilenetv2”，数据）；
```

```
// 异步：非阻塞，需要显式
    完成检查
期值 =客户端->请求(mode="异步", op="
    mobilenetv2", 数据);
future.get();  // 等待完成

// 流水线：批处理多个请求，轮询
    在批处理级别处理
// 在实际实现中,                    它被封装
    在一个专用函数中
用于(整数i = 0; i < 批次_大小;     i++) {
   作业_标识符[i] = 客户端->请求（模式="流水线", 操作
       ="MobileNetV2",    数据[i]);
}
// 正在做其他事情\dots
对于(整数i = 0; i < 批次_大小; i++) {
   // 等待所有作业完成
   结果[i] = 客户端->查询(job_ids[i]);
}
```

ROCKET API. 清单 1 展示了 ROCKET API 在三种执行模式下的使用方式。每个请求通过`mobilenetv2`指定一个操作（例如，`op`），并通过`data`提供输入。`sync`模式会阻塞直至完成，其行为类似于`memcpy`。`async`模式会返回一个期值，以支持非阻塞执行，并通过`get()`进行显式同步。`pipeline`模式会在内部将请求排队并批处理处理，同时返回 jobIDs 供后续结果查询使用。这样可以延后轮询并减少开销。服务器会提供并发元数据，帮助客户端调整缓存注入。默认情况下，同步模式会启用注入；异步模式会有条件地启用注入（单线程时启用）；流水线模式由于执行相互解耦，则会禁用注入。

配置工作量。ROCKET 提供三个关键参数：设备（`cpu`、`dsa`）、缓存_注入（开启、关闭）以及执行模式（同步、异步、流水线）。其中，模式在编译时固定，因为它决定了控制流。对于包含下游处理的端点，异步或流水线通常优于同步：前者更有利于降低延迟，后者则更有利于提高吞吐量。设备的选择取决于传输大小。实际应用往往在比微基准测试所显示的更大的传输大小下倾向于`dsa` [23]，，这是因为预处理、缓存流量以及 CPU-DSA 总线争用。缓存_注入在数据会立即重用时效果显著，但在缓存争用严重时应避免使用，因为这可能导致污染和失效。ROCKET 使用启发式默认设置——对同步/异步启用，对流水线禁用——但允许用户覆盖设置。

VI. 评估

在本节中，我们通过实验展示 ROCKET 能够在具有代表性的应用基准测试中提升端到端吞吐量并降低延迟，同时证明 ROCKET 中的设计选择能够有效应对与数据移动相关的硬件效率低下问题

A. 实验方法

测试平台。 实验平台总结见表 IV。我们使用的系统配备了 Intel Sapphire Rapids 处理器、NVIDIA A100 GPU 和一台 DSA 设备。**工作负载。** 评估使用了五项应用基准测试：MobileNetV2、XGBoost、PageRank、MilvusDB 和 Vision Transformer (ViT)，总结见表V。所选基准测试共同涵盖了数据移动压力的多个互补维度。MobileNetV2 展现了推理中典型的密集张量流传输；XGBoost 突出了细粒度特征批处理；PageRank 展现了不规则且重复利用频繁的图遍历；MilvusDB 体现了现代数据存储中的批次查询执行；ViT 则代表了现代深度学习中的 GPU密集型工作负载。它们共同涵盖了密集、稀疏和批处理访问模式——这些是数据密集型系统中进程间通信开销的主要来源，从而实现了均衡的覆盖。

表 IV：实验设置

组件	规格
CPU	英特尔(R) Xeon(R) Gold 6438Y+ 4.0GHz（Sapphire Rapids），32 个核心
GPU	NVIDIA A100（PCIe，40GB HBM2 内存，6912 个 CUDA 核心，312 TFLOPS FP16）
缓存	60MiB 末级缓存
DSA	1 个英特尔 DSA 设备，配有 1 个工作队列
RAM	704GB DDR4 4800MT/s
OS	Ubuntu 22.04.5 LTS ，内核 6.5.0-41
编译器	GNU编译器套件 11.4.0
库	GNU C库 2.35、accel-config、numactl、DTO, PyTorch 2.1.2、ONNX (1.19.1)、OpenCV, XGBoost、BoostGL、OpenVINO

表V：每个基准工作负载的流水线阶段。

基准	预处理	处理	后处理
MobileNetV2	图像解码，调整大小、归一化	CNN推理使用 ONNX 运 time	解析、格式化输出
XGBoost	构建特征向量	推理 使用预训练的提升树	解析预测输出
PageRank	构建邻接表	迭代 PageRank 计算	提取前 10 个顶点
MilvusDB w/ 图像嵌入	图像解码，调整大小，嵌入-ding 提取	近似最近邻，最相似的前三项图像	解析预测结果
Vision Transformer (ViT)	图像解码，调整大小，归一化	深度学习 基于Transformer的图像推理	解析、格式化输出

`MobileNetV2` 是一个视频分析流水线，其中每个客户端进程都会先执行基本的图像预处理，然后将数据发送到卷积神经网络（CNN）推理服务。生成的标签会经过后处理，以供远程渲染。每个请求对应一个包含 80 张预处理图像、总计 160MB 的批次。我们使用 Intel AMX 和'int8'量化模型。`XGBoost`是一个预测分析流水线，客户端从结构化表格数据中构建特征向量，并将推理卸载到服务器。每个请求包含乳腺癌数据集中的 200,000 行，总计约 25MB。`PageRank` 是一个图分析流水线，每个请求包含一个有 100 万个顶点和 1,000 万条边的图，总计 76MB。计算

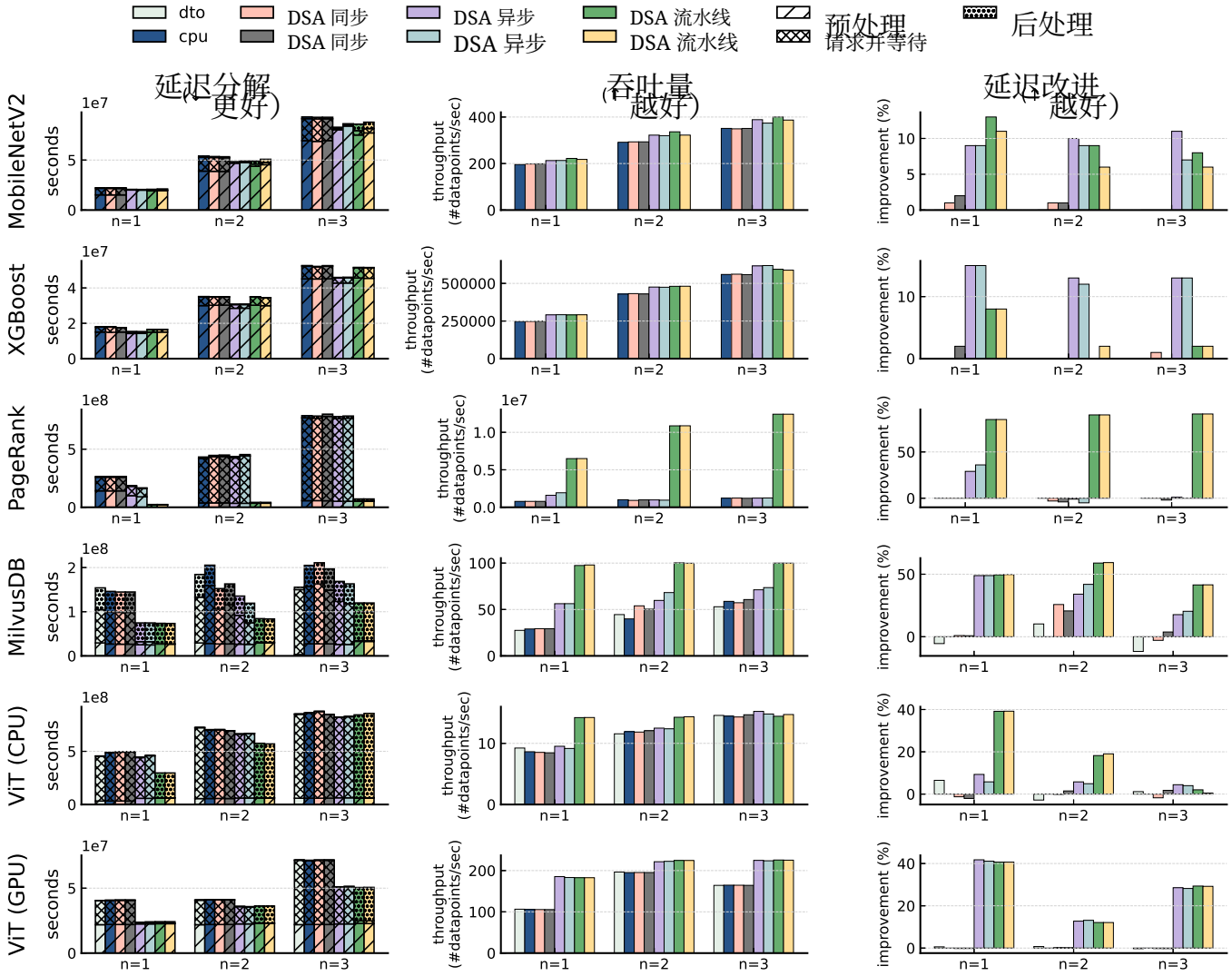


图 10：不同系统负载下，不同 ROCKET 进程间通信实现对 MobileNetV2、XGBoost、PageRank、MilvusDB、视觉 Transformer（CPU）、视觉 Transformer（图形处理器）的执行时间细分（左）、吞吐量（图像/秒，中）以及相较于基于 CPU 的基线的端到端延迟改进（右）：(a) 未过度订阅（n=1），(b) 匹配（n=2），以及 (c) 过度订阅（n=3）。

由于在多次迭代中反复访问顶点排名值，该计算具有较高的空间和时间局部性。`MilvusDB` 是一个相似性搜索流水线，每个请求都会提交 200 个通过`MobileNetV2` 提取的图像嵌入（1280 维，float32，1MB），用于在向量数据库中进行近似最近邻搜索。纳入该流水线是为了代表服务器端响应较大的应用，因为每个嵌入都会返回最相似的 3 张图像。视觉Transformer（ViT）是一个针对图像数据进行推理的深度学习分析流水线。每个客户端请求都会提交一个包含 200 张预处理图像的批次，每张图像的分辨率为 384x384 像素。纳入该模型是为了代表需要 GPU 加速的现代、高计算强度的基于 Transformer 的工作负载。表V总结了预处理、处理和后处理各阶段执行的主要操作。

性能特征。 我们的评估涵盖仅使用 CPU 和 GPU 加速两种场景。在仅使用 CPU 的情况下，计算执行主导了总运行时，而进程间通信 (IPC) 仅占延迟的约 1%。在采用图形处理器进行计算时，更快的计算阶段使进程间通信和数据移动在端到端延迟中的占比更大，从而放大了内存加速带来的收益。这一趋势与先前的研究一致 [1], [28],，该研究报告称，在数据移动成为瓶颈的 GPU 执行场景中，吞吐量提升了 5-20× 。即使在进程间通信占比较小的仅使用 CPU 场景下，ROCKET 仍通过高效处理进程间通信、降低内存总线争用、减少 CPU 周期，以及更好地重叠计算与数据传输，取得了显著提升。

B. 对延迟和吞吐量的影响

图10展示了 DSA 执行模式和缓存注入对各项基准测试端到端性能的影响。我们在三种负载条件下进行评估：(a) 未过度订阅（n=1）、(b) 匹配（n=2）和 (c) 过度订阅（n=3）。每组分别报告执行时间细分（顶部）、吞吐量

（中部），以及相对于仅 CPU 基线的延迟改进（底部）。同步 DSA 配置在 IPC 栈中与 DTO 相同，我们还为 `MilvusDB` 和视觉 Transformer 基准测试添加了 DTO 作为基线。此外，随着端到端执行时间中 IPC 通信所占比例增加，卸载的 IPC 带来的性能优势也愈发显著。与使用 CPU 相比，在 GPU 上运行视觉 Transformer 时，我们观察到使用 DSA 的优势显著增强。这表明，在多模态或多计算引擎环境中，硬件辅助 IPC 的优势可能更加突出。

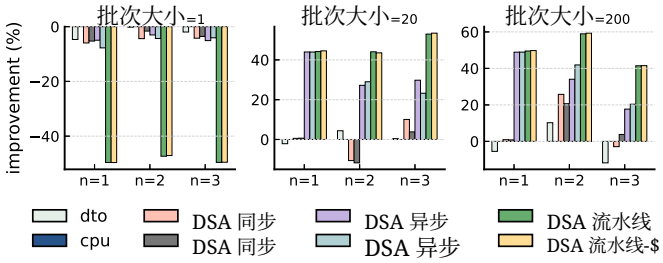


图 11：延迟改进及数据传输大小对最优模式选择的影响。（单个输入 ≈ 600KB。）即使是同一个应用，最佳性能所对应的配置参数也可能因批次大小而异，这是因为底层数据量会随之变化。

图 12 显示，异步模式下的部分延迟降低源于执行模型本身的改变，这一点可从同步_CPU到异步_CPU的性能提升看出。不过，大部分收益来自基于 DSA 的卸载，异步_DSA 的表现显著优于异步_CPU。这表明，尽管异步执行有助于减少空闲时间，但性能提升主要归功于 DSA 带来的开销降低和效率提高。

与现有软件支持的比较。在图 10 中，我们的评估涵盖了八种配置，包括DTO （供应商支持）。DTO 采用单一静态配置进行同步操作，且不支持细粒度调优 [9]。为保证公平，我们为 DTO 采用了合理的配置，启用硬件完成检查，并平衡 CPU-DSA 工作划分（AUTO_ADJUST=1, WAIT_METHOD=UMWAIT）。在我们的评估中，DTO 的吞吐量和延迟表现始终不佳，甚至逊于不使用 DSA 的基线方案。这是因为它不加区分地使用 DSA，即使卸载会适得其反，也仍然强制进行卸载，导致排队延迟增加、性能下降；而在这些情况下，基于 CPU 的复制效率更高。虽然 DTO 无需修改代码且易于采用，但它缺乏根据工作负载需求选择性使用 DSA 的灵活性。

执行模式的影响。 图10显示，异步模式（异步 和 流水线）始终表现出色，
构成同步基线，证实了通过非阻塞执行减少空闲时间这一设计目标。

同步模式在功能上与 DTO 类似，会执行阻塞式

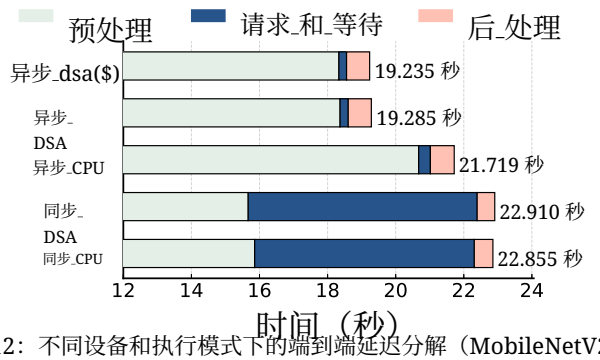


图12：不同设备和执行模式下的端到端延迟分解（MobileNetV2）。

卸载，与 CPU 内存复制相比，性能提升有限，凸显了同步使用 DSA 所产生的开销。流水线模式会将请求分批处理，并延后进行完成跟踪，从而减少轮询和同步开销。由于通常无需立即获取响应，这种延迟处理模型提升了效率。在两种异步模式中，异步模式的单次请求延迟较低，而流水线模式的吞吐量较高。这一区别在各种工作负载中始终如一。然而，在异步模式下，由于内存争用更严重，`MobileNetV2` 的预处理延迟比`XGBoost` 更长（图10），这表明当 CPU 工作负载不受内存限制时，CPU-DSA并行更有效。支持这三种模式，使 ROCKET 用户能够根据不同的延迟和吞吐量需求调整其进程间通信实现。

批量大小的影响（数据传输大小）。如图11所示，没有任何一种执行模式能在所有情况下始终优于其他模式。当数据传输大小较小时，流水线模式表现最差，但传输大小超过某个阈值后，它的效果最佳。我们还观察到，对于较小的数据传输，不使用 DSA 的纯 CPU 流水线性能最佳。即使在这些情况下，通过 DTO 采用静态 DSA 往往也会导致性能下降。

缓存注入的影响。结果表明，在低争用场景下，缓存注入能够降低延迟。例如，在单线程执行（`MobileNetV2`）中，在异步模式下启用注入，可显著降低延迟，因为缓存仍可立即重用。然而，在多线程配置、重用距离较长（可能由流水线模式导致），或数据大小足以填满末级缓存（可能发生空间/时间缓存争用）的情况下，同一机制会降低性能。多个线程同时注入会使缓存超额负载，导致缓存驱逐和命中率降低。这些结果证实，选择性缓存注入——而非简单地始终开启注入——是调优进程间通信性能的一项实用功能。

核心过度占用的影响。 在所有系统负载场景——未过度订阅、匹配和过度订阅配置——下，异步模式始终优于同步模式，其中流水线 针对吞吐量进行优化，而异步 针对延迟进行优化。值得注意的是，随着 CPU 资源变得更加受限（即线程数增加），

DSA 带来的相对延迟改进幅度会增加。这表明，在 CPU 受限的环境中，DSA 的优势最为明显，因为卸载内存操作可以为应用逻辑释放更多计算资源。在高争用场景下，减少由 CPU 引起的停顿和内存争用，相较于负载较轻的环境，能够带来更大的性能提升。

计算加速器的影响。 我们在 CPU 和图形处理器配置下对视觉 Transformer（ViT）工作负载进行了基准测试（图 10）。`ViT-cpu` 具有很高的计算强度。此外，IPC 总时间非常短（$\approx$ 0.2%）。在负载较低的流水线模式下，ROCKET 带来的收益最大，此时端到端收益源于对请求进行批处理；但在其他情况下，其影响可以忽略不计。在较高负载下（$n > 1$），资源争用更为严重，掩盖了流水线处理的收益。在 CPU 上（倒数第二行），计算受限阶段占据了运行时的主要部分。然而，图形处理器使绝对端到端吞吐量提升了10$\times$ 以上，从根本上改变了性能瓶颈。在 GPU 加速下（最后一行），计算受限阶段大幅缩短，使 IPC 和数据移动在端到端延迟中所占比例大幅增加（比 CPU 配置高约 10$\times$ ）。性能瓶颈转向图形处理器上的 IPC 和数据移动，这正是流水线模式能够提升整体系统性能的原因，即使在较高负载下（$n > 1$）也是如此。因此，ROCKET 的异步和流水线 DSA 模式可显著提升吞吐量，最高可达 40%；相对于已经比仅使用 CPU 的配置快一个数量级的基线而言，这是一项可观的提升。这种转变进一步凸显了高效数据处理的重要性。

C. 性能提升的来源

降低 CPU 周期数和总线周期数。 内存复制引发的指令开销和内存总线争用，是同步 CPU 和 DSA 执行中的关键性能瓶颈。ROCKET 通过卸载传输并使其与计算重叠来解决这些问题，尤其是在异步和流水线模式下。图 13 显示，ROCKET 将

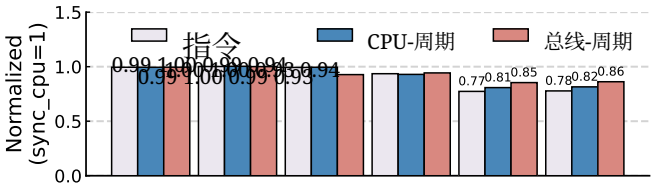


图 13：采用基于 DSA 的卸载后，归一化的指令数、CPU 周期数和总线周期数，相对于同步 CPU 基线。

与同步基线相比，指令数和 CPU 周期数最多可减少 23%。最大收益出现在流水线模式下，突发提交和延迟同步最大限度地减少空闲时间并提高硬件利用率。总线活动减少也进一步表明争用降低，且带宽利用更加高效。这些结果证实，ROCKET 能有效缓解第三节讨论的 CPU 和总线瓶颈。与 CPU 内存复制 或 DTO 等阻塞式 DSA 相比，ROCKET 的流水线策略在处理内存密集型工作负载时效率更高。

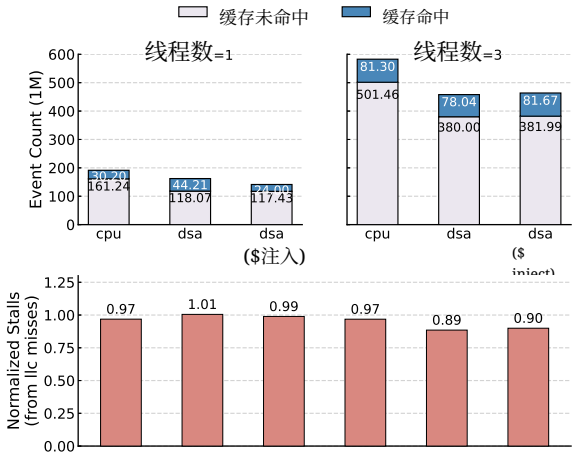


图 14：在匹配负载下，基于 DSA 卸载时的缓存命中、未命中和归一化 CPU 停顿。

对缓存效率和 CPU 停顿的影响。 涉及不可缓存内存访问的同步会降低缓存效率并引发 CPU 停顿，尤其是在多线程场景中。ROCKET 仅在可能复用时才启用缓存注入，从而平衡延迟和污染，缓解这一问题。图 14 显示，在匹配线程条件下，DSA 卸载会减少缓存命中和未命中，表明与基于 CPU 的 内存复制相比，整体缓存活动更少。这与先前研究结果 [24], 一致：缓存注入通过预取式放置减少访问量，从而降低这两项指标。然而，在较高线程数下，启用缓存注入会增加缓存访问——在三线程情形中，这是由于复用率低且存在干扰。这会导致引用抖动加剧，抵消性能收益。在两个线程且不进行注入时，流水线模式下的停顿率最低（0.89），这支持 ROCKET 的策略：在并发的延迟访问工作负载下禁用注入。

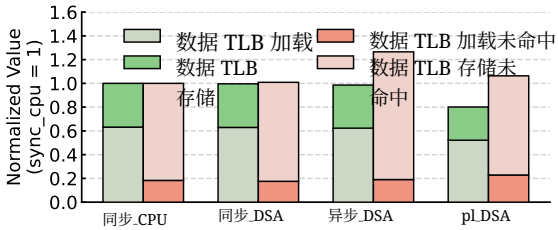


图 15：CPU 和 DSA 模式下匹配负载的归一化数据 TLB 活动。

对内存带宽和总线争用的影响。卸载可能会扰乱内存层次结构并降低 TLB 效率，尤其是在异步执行中。如图 15 所示，在异步和

流水线模式下，由于地址转换由输入输出内存管理单元处理，绕过了 CPU 中的转换后备缓冲器，因此数据 TLB 未命中率会上升。这表明，虽然卸载降低了 CPU 负载，但也限制了转换条目的重用，并削弱了局部性。ROCKET 的流水线模式通过批处理内存操作缓解了这些影响，提高了访问的可预测性。这些结果验证了 ROCKET 的设计权衡——牺牲部分 TLB 局部性以换取并行性。

ROCKET 通过解决第三节中指出的系统级低效问题来提升性能。异步模式消除了 CPU 阻塞，其中 流水线模式的吞吐量最高可提升 15%，且指令数比同步基线少 23 条。缓存注入可降低单线程情况下的延迟，但为避免缓存污染，在多线程环境下会禁用。与朴素卸载相比，批量流水线传输通过降低停顿率和总线争用，进一步缓解了内存子系统的压力。

这些结果表明，ROCKET 通过提供自适应接口，根据工作负载特征调整进程间通信行为，避免了以往方法中常见的局限，例如阻塞执行或不加区分地进行缓存注入。因此，与现有系统（如 DTO、idxd）相比，ROCKET 能更有效地利用 DSA，同时在并发、局部性变化和系统级争用情况下保持稳定的性能。

VII. 相关工作

基于 DSA 的数据移动加速。 近期研究探索了在不同领域使用英特尔数据流加速器（DSA）卸载内存密集型操作。以往工作包括对 DSA 微体系结构行为的基准测试和特征分析 [23], 通过 DTO 实现透明的应用级卸载 [9],, 以及内存去重 [18] 和分层内存系统 [26]等实际应用案例。这些系统利用 DSA 的硬件特征（例如描述符批处理和工作队列），以提升特定场景下的效率。DTO 虽然通过拦截 memcpy 调用简化了使用，但缺乏可编程性和细粒度控制。我们的工作以这些研究为基础，使用 DSA 实现进程间通信，并研究 DSA 的性能特征如何随内存条件而变化。与以往工作不同，ROCKET 支持多种执行模式，并提供可配置的延迟与卸载粒度权衡，从而能够更好地适配由多个进程构成的各类工作负载。

数据移动引擎与系统支持。以往研究探索了利用传统直接内存访问引擎（例如，英特尔 I/OAT [33]）、支持 RDMA 的网络接口卡以及 DMX [34] 和 MC^2 [20]等专用硬件来加速数据移动。这些系统利用异步卸载，减少 CPU 参与，并提高存储或加速器流水线的吞吐量。在操作系统层面，近期研究（如*Copier* [14]）将异步复制操作提升为协调式内核服务，通过在整个系统范围内管理复制与计算重叠，补充硬件卸载。相比之下，ROCKET 专注于节点内进程间通信，其内存移动的编排必须考虑进程边界、共享内存语义和响应能力。我们的设计提供细粒度控制，以满足进程间通信场景中的特定工作负载需求。尽管如此，ROCKET 的若干协调机制——例如模式切换、排队和基于成本的路径选择——也可推广到其他新兴引擎，使软件运行时系统能够更好地利用其异步能力。

VIII. 局限性与未来工作

ROCKET 面向节点内进程间通信，不支持节点间通信。尽管这限制了它在分布式环境中的直接部署，但其关键设计原则，例如异步内存编排和缓存感知传输，仍适用于其他支持内存卸载和缓存/同步控制的硬件。随着系统采用多节点 GPU、池化内存和 CXL 共享地址空间，ROCKET 所解决的问题——重叠数据移动并管理缓存效应——正变得越来越重要。然而，通过实证研究探索 ROCKET 在节点间运行时中的适用性，仍有待未来工作开展。

ROCKET 并非自动调优系统，不过提供了合理的默认设置。它通过手动接口公开关键参数（例如传输阈值、轮询预算和缓存策略）。尽管我们的实证分析有助于专家使用，ROCKET 仍缺少运行时决策引擎。我们认为，确定这类参数并了解它们与工作负载之间的相互作用，是实现有效自动调优的前提——这正是 ROCKET 的核心贡献。构建调优器（例如基于规则、统计方法或机器学习的方法）与此正交，仍属于未来工作；我们正在积极探索这一方向。

IX. 结论

硬件辅助内存卸载正成为现代系统设计的基石，它将数据移动与 CPU 执行解耦，并实现了新的并行形式。尽管此类机制有望大幅提升效率，但要充分发挥其潜力，需要软件运行时能够在进程边界间一致地管理同步、可见性和缓存行为。本文介绍了 ROCKET，这是一种将 Intel DSA 集成到用户空间进程间通信中的运行时，展示了如何在软件中系统地利用硬件卸载。ROCKET 引入了异步执行模式、复用感知缓存注入和轻量级同步，将底层卸载接口转化为可配置的进程间通信原语。对代表性工作负载的评估表明，ROCKET 可将指令数最多降低 22%，将吞吐量最多提升 2.1×，并将延迟降低 72%，证明了在运行时控制紧密配合的情况下，硬件卸载能够直接转化为端到端收益。通过在进程间通信中衔接硬件加速与软件编排，ROCKET 展示了一条切实可行的路径，使内存卸载成为一项一等系统能力。

参考文献

[1] “MLPerf 推理：数据中心基准测试套件”，MLCommons 基准测试套件，2025，访问日期：2025-07-30。

[2] A. Audibert、Y. Chen、D. Graur、A. Klimovic、J. ˇSimˇsa 和 C. A. Thekkath，《tf.data 服务：论机器学习输入数据处理的解耦》，载于《2023 年 ACM 云计算研讨会论文集》，SoCC '23 系列。美国纽约州纽约市：计算机协会，2023 年，第 358–375 页。 [在线]。网址：https://doi . 10.1145 3620678.3624666 org/ / R. Bachkaniwala、H. Lanka、[3] K. Rong 和 A. Gavrilovska，《Lotus：通过框架和硬件性能分析表征机器学习预处理流水线》，载于《2024 IEEE 国际工作负载特征分析研讨会（IISWC）》，2024 年，第 30–43 页。[4] A. Baumstark、L. Martins 和 K.-U. Sattler，《释放查询的非核心潜力：迈向无 CPU 查询处理》，载于《第 21 届数据管理新硬件国际研讨会论文集》，DaMoN '25 系列。美国纽约州纽约市：计算机协会，2025 年。[在线]。网址：https://doi.org/ 10.1145 3736227.3736243 / T. Benz、M. [5] Rogenmoser、P. Scheffler、S. Riedel、A. Ottaviano、A. Kurth、T. Hoefler 和 L. Benini，《高性能、能效出色的模块化 DMA 引擎架构》，IEEE《计算机汇刊》，第 73 卷，第 1 期，第 263–277 页，2024 年。[6] A. Berthold、C. F¨urst、A. Obersteiner、L. Schmidt、D. Habich、W. Lehner 和 H. Schirmeier，《揭秘用于内存数据处理的英特尔数据流加速器》，载于《第 2 届颠覆性存储系统研讨会论文集》，DIMES '24 系列。美国纽约州纽约市：计算机协会，2024 年，第 9–16 页。 [在线]。网址：https://doi . 10.1145 3698783.3699383 org/ / T. Chen 和 C. [7] Guestrin，《XGBoost：一种可扩展的树提升系统》，《第 22 届 ACM SIGKDD 知识发现与数据挖掘国际会议论文集（KDD）》，第 785–794 页，2016 年。[8] K. Clark、B. Vendt、K. Smith、J. Freymann、J. Kirby、P. Koppel、S. Moore、S. Phillips、D. Maffitt、M. Pringle、L. Tarbox 和 F. Prior，《癌症影像档案库（TCIA）：维护和运营公共信息存储库》，第 1045–1057 页，2013 年。[在线]。网址：https://doi . 10.1007 10278 013 9622 7 org/ /s - - - [9] I. Corporation，《 Intel DTO （数据流加速器工具）》，https:// github.com/intel/DTO，2023，访问日期：2024-02-14。[10] P. Fent、A. v. Renen、A. Kipf、V. Leis、T. Neumann 和 A. Kemper，《使用 RDMA 和共享内存实现 DBMS 的低延迟通信》，载于《2020 IEEE 第 36 届数据工程国际会议（ICDE）》，2020 年，第 1477–1488 页。[11] Y. Gan、Y. Zhang、D. Cheng、A. Shetty、P. Rathi、N. Katarki、A. Bruno、J. Hu、B. Ritchken、B. Jackson、K. Hu、M. Pancholi、Y. He、B. Clancy、C. Colen、F. Wen、C. Leung、S. Wang、L. Zaruvinsky、M. Espinosa、R. Lin、Z. Liu、J. Padilla 和 C. Delimitrou，《面向微服务的开源基准测试套件及其对云和边缘系统的软硬件影响》，载于《第二十四届编程语言与操作系统架构支持国际会议论文集》，ASPLOS '19 系列。美国纽约州纽约市：计算机协会，2019 年，第 3 18 [ ] . 10.1145 3297858.3304013 – 。在线 。[12] 3 网址：https://doi org/ / D. Gosnell，《在应用中加入实时机器学习的考量》，RTInsights，2022 年 8 月。[在线]。网址：https://www.rtinsights.com/ 3-considerations-for-adding-real-time-ml-to-applications/ [13] P. Goyal、P. Doll´ar、R. Girshick、P. Noordhuis、L. Wesolowski、A. Kyrola、A. Tulloch、Y. Jia 和 K. He，《精准的大批量小批量随机梯度下降：在 1 小时内训练 ImageNet》，2018 年。 [在线]。网址：https://arxiv . 1706.02677 org/abs/ J. He、Y. Dong、D. Du、M. Zou、[14] Z. Yu、Y. Ren、N. Jia、Y. Xia 和 H. Chen，《如何复制内存？将协同异步复制作为一流操作系统服务》，载于《ACM SIGOPS 第 31 届操作系统原理研讨会论文集》，SOSP '25 系列。美国纽约州纽约市：计算机协会，2025 年，第 1062–1081 页。 [ ] . 10.1145 3731569.3764800 在线 。网址：[15] https://doi org/ / K. He、X. Zhang、S. Ren 和 J. Sun，《用于图像识别的深度残差学习》，载于《IEEE 计算机视觉与模式识别会议论文集（CVPR）》，2016 年。[16] Intel Corporation，《英特尔数据流加速器用户指南》，2024 年，文档 ID：759709。[在线]。网址：https://www.intel.com/content/www/us/en/content-details/759709/ intel-data-streaming-accelerator-user-guide.html[17]Intel Corporation，《电源管理：用户等待指令——降低 DPDK PMD 轮询工作负载的功耗（技术指南）》，2024 年，访问日期：2025-03-30。[在线]。网址：https://www.intel.com/content/www/us/en/content-details/751859/power-management-user-wait-instructions-power-saving-for-dpdk-pmd-polling-workloads-technology-guide.html [18] H. Ji、M. Kim、S. Oh、D. Kim 和 N. S. Kim，"Para-ksm：借助数据流加速器实现并行化内存去重"，载于《2025 年 USENIX 年度技术会议论文集》（USENIX ATC）。美国马萨诸塞州波士顿：USENIX 协会，2025，即将发表。[19] Z. Jia 和 E. Witchel，"Nightcore：面向延迟敏感型交互式微服务的高效可扩展无服务器计算"，载于《第 26 届 ACM 编程语言与操作系统体系结构支持国际会议论文集》，丛书 ASPLOS '21。美国纽约：计算机协会，2021，第 152–166 页。 [在线]。获取地址：https://doi . 10.1145 3445814.3446701 org/ / A. K. [20] 2 Kamath 和 S. Peter，"(mc)：内存控制器中的惰性内存复制"，载于《2024 年 ACM/IEEE 第 51 届计算机体系结构国际研讨会》（ISCA），2024，第 1112–1128 页。[21] S. Kanev、J. P. Darago、K. Hazelwood、P. Ranganathan、T. Moseley、G.-Y. Wei 和 D. Brooks，"仓库规模计算机的性能分析"，载于《第 42 届计算机体系结构国际研讨会论文集》，丛书 ISCA '15。美国纽约：计算机协会，2015，第 158–169 页。 [在线]。获取地址：https://doi . 10.1145 2749469.2750392 org/ / M. Kuchnik、A. Klimovic、[22] J. Simsa、V. Smith 和 G. Amvrosiadis，"Plumber：诊断并消除机器学习数据流水线中的性能瓶颈"，载于《机器学习与系统会议论文集》，D. Marculescu、Y. Chi 和 C. Wu 编，第 4 卷，2022，第 33–51 页。 [在线]。获取地址：https://proceedings.mlsys.org/paper files/ paper/2022/file/d0e90e9a9310570dfa643aa3b2da6e89-Paper.pdf [23] R. Kuper、I. Jeong、Y. Yuan、R. Wang、N. Ranganathan、N. Rao、J. Hu、S. Kumar、P. Lantz 和 N. S. Kim，"现代英特尔至强可扩展处理器中数据流加速器的定量分析与指南"，载于《第 29 届 ACM 编程语言与操作系统体系结构支持国际会议论文集》，第 2 卷，丛书 ASPLOS '24。美国纽约：计算机协会，2024，第 37–54 页。 [在线]。获取地址：https://doi . 10.1145 3620665.3640401 org/ / E. A. Leon、K. B. [24] Ferreira 和 A. B. Maccabe，"使用缓存注入降低内存墙对输入/输出的影响"，载于《第 15 届 IEEE 高性能互连年度研讨会》（HOTI 2007），2007，第 143–150 页。[25] J. Leskovec 和 A. Krevl，"SNAP 数据集：斯坦福大型网络数据集集合"，https://snap.stanford.edu/data，2014 年 6 月。[26] R. Liu、T. Ma、M. Zhang、J. Huang、Y. Shan、Z. Liu、L. Xiang、Z. Lin、H. Lu、J. Rao、K. Chen 和 Y. Wu，"DSA-2LM：采用英特尔 DSA 的无 CPU 分层内存架构"，载于《2025 年 USENIX 年度技术会议论文集》（USENIX ATC）。美国马萨诸塞州波士顿：USENIX 协会，2025，即将发表。[27] ONNX 社区，"开放神经网络交换（ONNX）"，https: //onnx.ai，2019，访问日期：2025-04-10。[28] M. Park、K. Bhardwaj 和 A. Gavrilovska，"Pocket：来自边缘的机器学习服务"，载于《第十八届欧洲计算机系统会议论文集》，丛书 EuroSys '23。美国纽约：计算机协会，2023，第 46–62 页。 [在线]。获取地址：https://doi . 10.1145 3552326.3587459 org/ / V. Saravanan、K. [29] D. Pralhaddas、D. P. Kothari 和 I. Woungang，"面向多核 CPU 功耗与性能的流水线停顿减少优化算法"，《以人为中心的计算与信息科学》，第 5 卷，第 1–13 页，2015。[30] K. Shvachko、H. Kuang、S. Radia 和 R. Chansler，" Hadoop 分布式文件系统"，载于《2010 年 IEEE 第 26 届大容量存储系统与技术研讨会》（MSST）。IEEE，2010，第 1–10 页。[31] J. G. Siek 和 T. B. Community，"Boost 图形库（BGL）"，2001，访问日期：2025-04-10。 [在线]。获取地址：https://www.boost.org/doc/ libs/release/libs/graph/[32] UHD 联盟，"了解 4K：4K 超高清技术概述"，https://uhdalliance.org/，2020，访问日期：2024-04-10。[33] K. Vaidyanathan、W. Huang、L. Chai 和 D. K. Panda，"利用硬件复制引擎设计高效的异步内存操作：以 I/OAT 为例"，载于《2007 年 IEEE 国际并行与分布式处理研讨会》，2007，第 1–8 页。[34] S.-T. Wang、H. Xu、A. Mamandipoor、R. Mahapatra、B. H. Ahn、S. Ghodrati、K. Kailas、M. Alian 和 H. Esmaeilzadeh，"数据移动加速：串联跨域多加速器"，载于《2024 年 IEEE

高性能计算机体系结构国际研讨会（*HPCA*），2024，第 1043–1062 页。[35] S. Yun 和 Y. Ro，“Shvit：采用内存高效宏观设计的单头视觉 Transformer”，载于*2024*年*IEEE/CVF*计算机视觉与模式识别会议（*CVPR*），2024，第 5756–5767 页。[36] Zilliz，“Milvus：用于可扩展相似性搜索的开源向量数据库”，https://github。com/milvus-io/milvus，2020，访问日期：2025-06-21。[37] A. Zuepke、A. Bastoni、W. Chen、M. Caccamo 和 R. Mancuso，“Mempol：在核心外部监管核心内存带宽”，载于*2023*年第*29*届*IEEE*实时与嵌入式技术及应用研讨会（*RTAS*），2023，第 235–248 页。