数据中心高效架构

陈诚[1)]

[1)](华中科技大学计算机科学与技术学院, 武汉 430074)

摘 要 随着互联网的蓬勃发展，近年来数据中心的发展也十分迅猛，数据中心逐渐成为了现代社会的一种基础设施。但是随着线上业务的激增，数据中心的压力也越来越大，如何设计一种高效的架构来提高数据中心的利用率就成了一项研究热点。近年来许多学者对这个问题展开了研究。Chaojie Zhang、Alok Gautam Kumbhare 等人提出了 Flex，通过基础设施支持和分布式软件的结合实现了一个负责智能放置工作负载的离线组件和一个用于动态管理电源、可用性和性能的在线组件，极大的提升了数据中心的资源利用率，降低了数据中心的建设成本[1]。Yujeong Choi、Yunseong Kim 等人提出了 LazyBatching，通过机器学习推理的方式，设计了一个智能批处理系统，该系统可以动态调整批处理级别，以满足延迟、吞吐量和 SLA 需求，实现了灵活的批处理，降低了数据中心的平均响应时间、提高了吞吐率、满足了 SLA 需求[2]。Rohan Basu Roy、Tirthak Patel 等人提出了 SATORI，这是一种主动且同时控制多个 CMP 体系结构资源以实现多个目标的技术，通过牺牲短期的利益来换取长期的高效，有效地处理计算核心、LLC 方式、内存带宽和功率上限资源，实现了吞吐量和公平性之间的完美平衡[3]。AmirAli Abdolrashidi、Hodjat Asghari Esfeden 等人提出了 BlockMaestro，通过软硬件结合的方式，实现对程序员透明的基于任务的执行模型，通过追踪内核间的数据依赖关系来隐藏内核启动开销的影响，提高了 GPU 的利用率，加速了应用的执行[4]。

1 引言

数据中心是在一个集中的空间内实现信息的集中存储、处理、管理的一种设施，从应用层面看，包括业务系统、基于数据仓库的分析系统；从数据层面看，包括操作型数据和分析型数据以及数据与数据的集成、整合流程；从基础设施层面看，包括计算机设备、网络设备、存储设备、制冷系统和供电系统。

数据中心的雏形可以追溯到上世纪六七十年代，互联网时代的早期。而在国内，上世纪 90 年代，中国电信对外提供的数据托管和信息服务，也是我国早期数据中心的一种形式。随着日渐丰富的应用和网络的发展，现在数据中心主要为互联网内容提供商、企业、银行、政府机构等部门、单位提供大规模、高质量、安全可靠的专业化服务器托管、空间租用、网络批发带宽等业务。

当前，作为经济发展中创新性最强、增长速度最快、影响范围最广的产业，数字经济正在引领新的经济发展。近年来，数据中心发展迅猛，逐渐成为了现代社会的一种基础设施，并发展成为了一个国家的战略资源，和自然资源等占据相同地位。大数据时代的到来，使得很多公司越来越依赖数据中心。特别是在疫情的影响下，全球的线上业务激增，对数据中心的需求也成倍增长，这也导致数据中心的压力激增。

在这样的背景下，使用一种高效的数据中心架构来提高数据中心的利用率，降低数据中心的成本，成为了一项很有实际意义的研究话题。近年来，相关的问题也引发了广泛的讨论：

• 抽象的云服务提供商，如 Amazon 和 Microsoft，由于其应用的要求，必须保证其大部分工作负载的高可用性。出于这个原因，他们建立了具有高冗余的数据中心，用于电力输送和冷却。通常，冗余资源只保留在基础设施故障或维护事件期间使用，因此工作负载性能和可用性不会受到影响。不幸的是，预留的资源也会降低电力利用率，随着对云服务的需求持续增长，需要建立更多的数据中心才能应对业务的增长。充分利用每个数据中

心的基础设施(例如，电力、空间、冷却)对于减少必须建造的数据中心和节省成本至关重要。由于功率通常是瓶颈资源，大型提供商使用功率超额订阅和功率上限(通过性能调节)来安全地提高利用率，方法是在每个数据中心上部署更多的服务器[5]、[6]、[7]、[8]、[9]。然而，由于供应商必须保持高的基础设施可用性，仍然存在很大的低利用率问题。但是预留的资源也为提高数据中心利用率和降低成本提供了机会。例如，在典型的高可用数据中心中，会预留 10% ～ 50%的电力和冷却资源[10]、[11]、[12]。利用这些资源部署更多的服务器将显著增加电力使用效率，减少需要建立的新数据中心的数量，并提高云提供商的可持续性。

• 在云 ML 系统中，批处理是提高吞吐量的一项重要技术，有助于优化总成本。先验图批处理将单个 DNN 图组合成一个图，允许多个输入并行地执行。粗粒度图批处理在有效处理动态推理请求流量时变得不是最优的，从而使得数据中心剩下了很多的性能。

• 多核体系结构使数据中心能够越来越多地将多个作业放在一起，以提高资源利用率和降低运营成本。但是，简单的将多个作业放在一起可能只会略微增加系统吞吐量。更糟糕的是，当多个用户使用同一物理内核的时候，一些用户可能会观察到更大比例的性能下降。

• 当今的图形处理单元(GPU)是一种计算能力强大、耗用大量能量的并行设备，利用其单指令、多线程(SIMT)范式[13]、[14]、[15]、[16]、[17]能够同时处理数千个线程的应用程序。随着现代 GPU 工作负载规模和复杂度的增长，对 GPU 计算能力的需求也在不断增加。新出现的工作负载也给 GPU 带来了巨大的负担。通过在一个执行过程中启动数百个内核的应用程序，内核启动的开销会变得很大[18]，[19]，[20]。这些内核通常还有着重要的数据依赖关系[21]、[22]、[23]。例如，CNN 中的层生成的数据将在下一层中使用。并且需要使用同步来处理内核之间表现出来的数据依赖关系，这也就会导致 GPU 利用率不足。已经有人提出基于任务的执行模型来解决这些问题，方法是将应用程序移植到专有的基于任务的编程模型上，以便指定任务和任务依赖关系，但这需要程序员付出大量的努力。

2 原理和优势

由此可见提高数据中心的利用率、降低数据中心的成本，对于数据中心的发展来说至关重要。针对以上的问题，学术界进行了深入的研究：

2.1 Flex：零预留功率的高可用性数据中心

文章提出了“零预留功率”的数据中心概念以及如何管理这些数据中心的 Flex 系统，确保在“零预留”的工作负载下，应用仍然能够获得所需的性能和可用性。Flex 利用软件冗余工作负载的存在，可以容忍较低的基础设施可用性，同时对那些需要高基础设施高可用性的工作负载造成最小的性能降低。更详细地说，文章展示了大型云提供商可以在高可用的数据中心分配所有预留资源，用于部署额外的服务器，同时最小化维护事件的影响。

Flex 主要包括：

(1)一个新的离线工作负载放置策略，在确保故障或维护事件期间的安全的同时，减少了电力滞留;

(2)一个分布式系统，监测故障并在检测到故障时根据工作负载的要求快速降低电力消耗。

评估显示，Flex 产生不到 5%的滞留电量，并将部署服务器的数量增加了 33%，这意味着每个数据中心站点可以节省数亿美元的建设成本。最后，文章还总结了在微软数据中心将 Flex 引入生产的经验教训。

2.2 LazyBatching：云机器学习推理的 SLA 感知批处理系统

文章提出了一种基于 LazyBatching 的批处理系统，该系统考虑了单个图节点的粒度，而不是整个图的粒度，从而实现灵活的批处理。文章证明了 LazyBatching 可以智能地确定高效批处理的节点集。LazyBatching 在平均响应时间、吞吐量和 SLA 满意度方面，分别比图批处理实现了平均 15×、1.5×和 5.5×的改进。

2.3 SATORI：以牺牲短期利益换取长期利益的高效、公平的资源分配

SATORI 是一种划分多核架构资源的新策略，以同时实现两个相互冲突的目标:提高系统吞吐量和实现协同工作之间的公平性。SATORI 是唯一一种主动且同时控制多个 CMP 体系结构资源以实现多个目标的技术。SATORI 可以有效地处理计算核

心、LLC 方式、内存带宽和功率上限资源，以实现吞吐量和公平性之间的最佳平衡------这也补充了最近发表的基于经济学、协同过滤和机器学习[24]，[25]的协同位置下的能量效率方法。

2.4 BlockMaestro：在 GPU 系统中启用程序员透明的基于任务的执行模型

BlockMaestro 是一种软硬件结合的解决方案，可以提升 GPU 利用率，给应用加速。BlockMaestro 结合了命令队列重新排序、内核启动时静态分析和运行时硬件支持，以动态识别和解析内核之间的线程块级数据依赖关系。通过在内核启动时对内存访问模式的静态分析，BlockMaestro 可以提取内核线程间的块级数据依赖关系。BlockMaestro 还引入了内核预启动，以减少多个依赖内核的内核启动开销。内核预启动的正确性是通过在运行时通过硬件支持动态解析线程块级数据依赖来实现的。BlockMaestro 在依赖数据的基准测试中提高了 51.76%(最高达 2.92x)的平均速度，并且需要最小的硬件开销。

Flex 通过“零预留”极大的提升了数据中心的资源利用率，降低了数据中心的建设成本；LazyBatching 通过机器学习推理的方式，实现了灵活的批处理，降低了数据中心的平均响应时间、提高了吞吐率；SATORI 划分多核架构资源，牺牲短期的利益来换取长期的高效，实现了吞吐量和公平性之间的完美平衡；BlockMaestro 通过软硬件结合的方式，实现对程序员透明的基于任务的执行模型，提高了 GPU 的利用率，加速了应用的执行。

3 研究进展

3.1 Flex：零预留功率的高可用性数据中心

Flex 的目标是在确保安全(即防止级联故障)和保证工作负载的性能和可用性 SLA 的同时，尽可能多地使用预留的电源。分布式冗余电源层次结构与混合工作负载需求是 Flex 的关键推动因素。

在传统的、具有 4N/3 分布式冗余的非 flex 数据中心中，只有 75%的电力分配给 IT 设备，其余的则预留。图 1(左)说明了这样一个数据中心;75%的水平线表示故障转移预算，即可以安全分配和使用的已配置电力的总量。因为保留功率，在故障切换事件(A)期间，剩余功率设备(如 ups)的总功率负载仍低于其总容量(B)。该场景不需要纠正操作，但以预留功率为代价。而在 Flex 数据中心中，我们使用所有的功能来部署 IT 设备。正如我们在图 1(右)中所示，这可能会使电力使用峰值超出故障转移预算(A)。在正常(非故障)运行期间，所有电力设备的总容量足以承受这种高负载。然而，当故障(或计划维护)事件与高负载(B)同时发生时，剩余设备上的总负载超过其容量。如果透支时间过长，会导致级联故障，初始设备故障导致另一台设备关闭，再导致另一台设备关闭。

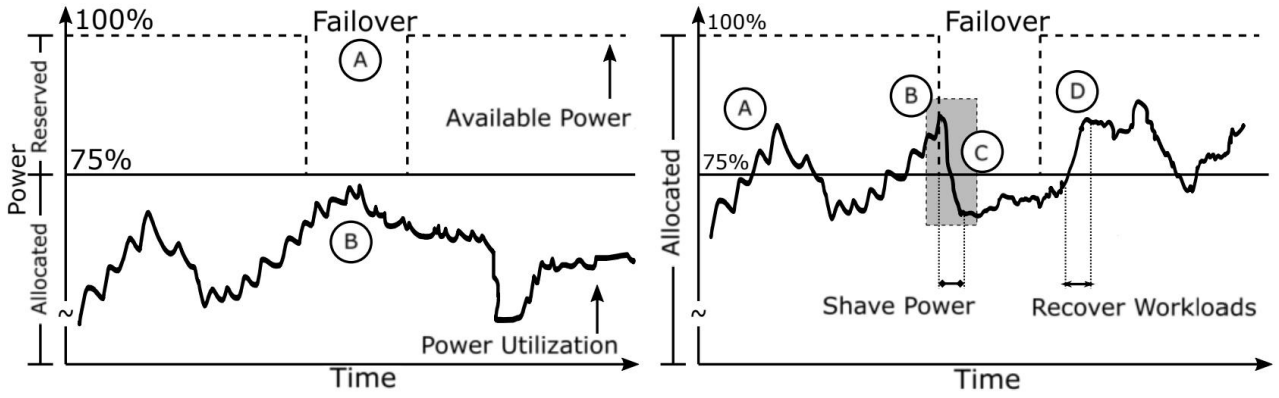


图1. 传统数据中心预留功率(左)和Flex数据中心零预留功率(右)的功率配置

为了防止级联故障，我们必须在剩余设备的过载容限(C)范围内，迅速削减超过其容量的电力使用。我们建议通过关闭软件冗余工作负载和限制非冗余但有能力的工作负载的功率来实现这一目标。一旦故障得到缓解或维护完成，电源帽就可以被解除，服务器就可以重新开启(D)，以恢复满负荷运行。IT 设备的部署应该确保足够的软件冗余和有能力的工作负载可用，即使在最坏的情况下(即 100%的电力利用率)，也要将电力消耗降至故障转移预算以下。无法产生这样的工作负载多样性可能会导致权力搁浅。设备遥测技术和 Flex 系统控制器成为关键任务，必须具有高度可用性，以确保不会错过过载情况，并在容许时间内(基于过载脱扣曲线)采取行动。强制功率上限或关闭服务器可能会对工作负载产生重大影响，无论是在性能和/或可用性方面。虽然确保安全至关重要，但最小化工作负载影响和仅在所需的最小服务器子集上采取行动也很重要，以便将能力安全地降低到故障转移预算以下。

Flex 用三个组件来解决这些问题，如图 2 所示。首先，一个离线组件(称为 Flex-Offline，标记为 A)，它可以满足不同工作负载的短期容量需求，并优化服务器的位置(章节 IV-B)。第二，高可靠的电力遥测管道(B)，提供电力设备的实时监测。第三，高度可用的 Flex 控制(C)在故障转移期间做出并强制决定关闭或限制已部署服务器的一个子集。我们称这组控制器为 Flex-Online。Flex-Online 依赖于 ups 及其电池的临时过载能力;基础设施的其他部分(例如，交换机、电缆、断路器)通常具有更高的容忍度。例如，图 3 显示了我们的 ups 的电池寿命开始和结束的公差曲线，作为负载的函数。在最坏的情况下，故障转移负载为 133%，UPS 提供 10 秒的容错时间，然后在 100%负载下提供额外的 3.5 分钟。这种额外的骑行允许发电机启动并接收来自电池的负载。因此，我们对 Flex-Online 的端到端延迟实施了 10 秒的限制，包括故障转移检测、遥测采集和控制器动作，以将功率降低到额定(100%)UPS 容量以下。

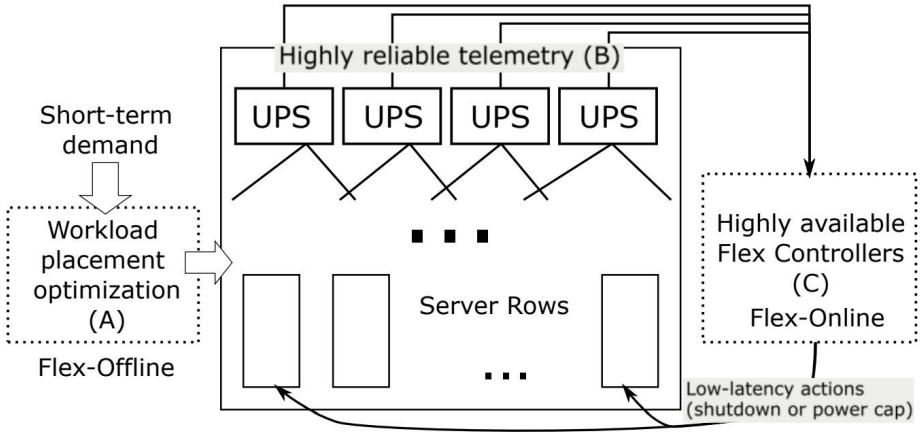


图2. Flex概览

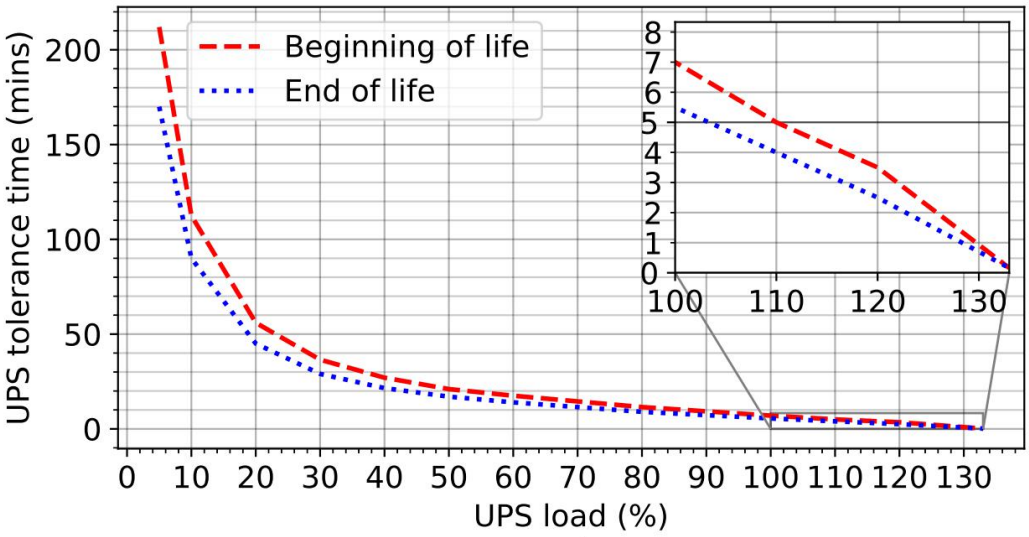


图3. UPS公差作为负载的函数

3.2 LazyBatching：云机器学习推理的 SLA 感知批处理系统

文章提出了一种名为 LazyBatching 的智能批处理系统，该系统可以动态调整批处理级别，以平衡延迟、吞吐量和 SLA(服务水平协议)满意度。传统图批处理的一个关键限制是：当正在进行的一批请求尚未完成其执行时，它无法为新到达的请求提供服务。LazyBatching 不是让一个批处理的输入不间断地执行，直到整个图完成，而是节点级维护调度粒度，并允许不同的(批处理的)输入交错执行。实际上，LazyBatching 调度器可以抢占和暂停当前正在进行的批处理，直到新到达的输入被优先调度以赶上被抢占批处理的进度。这种灵活的节点级调度使得被抢占和抢占的请求可以在任何给定的层进行批处理，这大大提高了批处理的效率。尽管“惰性”批处理必须等待新接收的输入赶上其进度，但是上述的“惰性”批处理输入的有效性还是取决于预先抢占的批处理输入是否仍然能够满足 SLA 目标。文章提出的 LazyBatching 的关键创新是开发了一种 SLA 感知的调度算法，该算法利用 ML 推理领域的特定属性来智能地决定何时以及哪些输入需要延迟批处理。具体地说，LazyBatching 确定当前正在进行的请求的剩余 SLA 松弛时间，并利用该信息动态地判断是抢占还是继续这个正在进行的请求，以满足 SLA 目标，同时最大化系统吞吐量。由于 LazyBatching 可以灵活地适应重载或轻负荷推理请求流量条件下的批处理，它将最终用户从搜索最优的批处理超参数(例如，批处理时间窗和最大批处理大小)中解脱出来，而不像传统的静态图批处理那样。

LazyBatching 系统有助于提高吞吐量，同时仍然满足云 ML 推理的 SLA 目标。以下是文章的主要贡献:

• 文章开发了一个 SLA 感知的冗余预测模型，该模型利用 ML 推理的领域特异性特性来预测冗余估计的 DNN 推理时间。

• 提出了 LazyBatching，一种用于云推理的低成本和实用的批处理系统。与之前的工作不同，此解决方案不局限于特定类型的 DNN 层，可以灵活地适应部署环境，而无需手动调整批量参数。

• 与图批处理相比，LazyBatching 在延迟、吞吐量和 SLA 满意度方面分别提供了平均 15 倍、1.5 倍和 5.5 倍的改进。

3.3 SATORI：以牺牲短期利益换取长期利益的高效、公平的资源分配

当多个工作负载同时位于 CMP 机器上并共享多个体系结构资源时，SATORI 是第一个同时主动控制系统吞吐量和公平性目标的解决方案。SATORI 提出了新的解决方案，以弥补现有方法中存在的优化缺陷。SATORI 开发了一种基于贝叶斯优化(BO)理论的新方法来探索多资源分区配置空间。特别是，SATORI 基于 BO 的方法能够设计一种实际可行的技术，可以在一个真实的系统上以在线的方式工作。使用 BO 背后的关键点是建立简单和刚好足够精确的模型，以找到接近最佳的解决方案，而不需要离线分析、检测、基于离线深度学习或强化的训练或构建复杂的性能模型。对于不同的离线训练数据集[26]，[27]，离线分析、检测等可能会产生高开销，并且可能无法移植到新的应用环境。

通过对模型的轻微误差进行容忍，SATORI 可以在线实现接近最优配置。SATORI 能够快速高效地导航大配置空间，找到最优的资源分区配置。它同时对多个资源进行联合探索——消除了现有方法的局限性，即为每个共享资源维护一个有限的状态机或每次探索一个资源维度。

在一个目标和另一个目标之间进行权衡是有益的，可以在长期内获得更高的回报。SATORI 在短期内优先考虑一个目标，然后在长期的基础上在两个目标之间取得平衡。不幸的是，目标的动态重新排序会导致正在优化的目标函数随着时间的推移而改变，因此，应用 BO 变得具有挑战性。为了解决这一挑战，SATORI 实现了一个新的增强功能，动态、单独地监控系统级吞吐量和工作级公平指标，针对多个目标以构建一个新的整体组合 BO 目标函数。这个目标函数成功地改变了目标的优先级，同时通过边界保持了传统 BO 的预期行为动态重新确定目标优先次序的幅度和时机。SATORI 的固有设计使它适合于附加的并发优化目标。SATORI 评估证明了它在一系列场景和并行工作负载中的有效性。例如，在吞吐量方面，SATORI 比最近的工作负载共定位技术(包括 dCAT、CoPart 和 PARTIES)分别高出 19%、17%和 14%，同时实现更高的公平性，分别高出 25%、17%和 14%。SATORI 在离线、实际上不可行的 oracle 方案的 8%以内执行。SATORI 可以与最近基于经济学、资源复用和机器学习的有针对性的方法相补充和结合[24]，[25]。

3.4 BlockMaestro：在 GPU 系统中启用程序员透明的基于任务的执行模型

3.4.1 概述

为了解决 GPU 内核启动开销和内核间的数据依赖关系所导致的 GPU 利用率低的问题，已经有人提出了许多基于任务的执行模型和运行时[21]、[22]、[28]、[29]。这些框架要求程序员将应用程序分解为任务，并通过专有的编程模型(如 AMD ATMI[5]、CUDA Graphs[30]、OpenMP tasks[31]等)表达如图 4 所示的任务依赖关系，然后由运行时强制执行。基于任务的执行的主要好处是：通过共同启动一组内核作为一个整体[28]、[30]，或者通过启动一个持久内核来处理进入其工作队列的任务，可以显著减少内核启动开销；而依赖的任务只要满足数据依赖关系就可以开始执行。然而，为了获得这些好处，现有的 GPU 应用程序必须重构成这些专有的基于任务的编程模型。

为了减少程序员的干预，文章提出了 BlockMaestro，它使用现有的 SIMT 编程模型支持基于任务的执行，并避免了大量的代码修改。BlockMaestro 背后的关键观点是，内核预启动和细粒度内核间数据依赖解析实现了基于任务的执行模型。通过预先启动内核，掩盖内核启动的开销。为了加强正确性，在解决线程块级别的数据依赖关系之前，不会执行预先启动的依赖内核的线程块(TB)。BlockMaestro 概览如图 5。

实现基于任务的执行模型有 3 个问题需要解决：

- 如何识别和提取数据之间的依赖关系？
- 如何减少内核启动开销？
- 如何保证内核预启动的正确性？

3.4.2 识别和提取数据之间的依赖关系

3.4.2.1 内核-内核数据依赖

内核之间的数据依赖关系出现在全局内存中的数据中。由于采用了 SIMT 编程模型，内核的输入和输出都得到了良好的定义。内核使用的全局内存中的每个区域都是通过 API 调用来分配的，比如 CUDA 中的 cudaMalloc 或 HIP 中的 hipMalloc。在内核启动时的即时编译阶段，可以通过对内核的 PTX 或 SASS 代码进行静态分析来识别加载和存储地址。

如果内核要读取或写入已分配的全局内存区域，则必须将内存分配的基指针传递给内核启动 API。对于内存 API，以类似的方式传递基指针。写是主机到设备的操作，读是设备到主机的操作。因此，内核和 API 调用之间的数据依赖关系可以在命令队列中以一种相当直接的方式识别。

3.4.2.2 线程块间数据依赖

BlockMaestro 需要通过强制执行线程块级别的数据依赖来避免依赖停滞的线程块。通过在线程块的粒度上强制内核间数据依赖，可以重叠依赖内

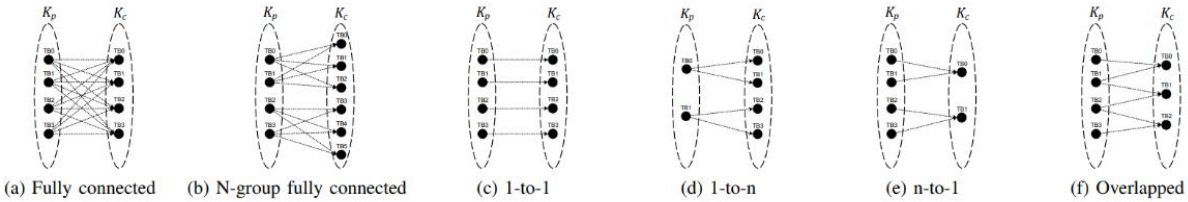


图4. 常见内核TB之间的依赖模式

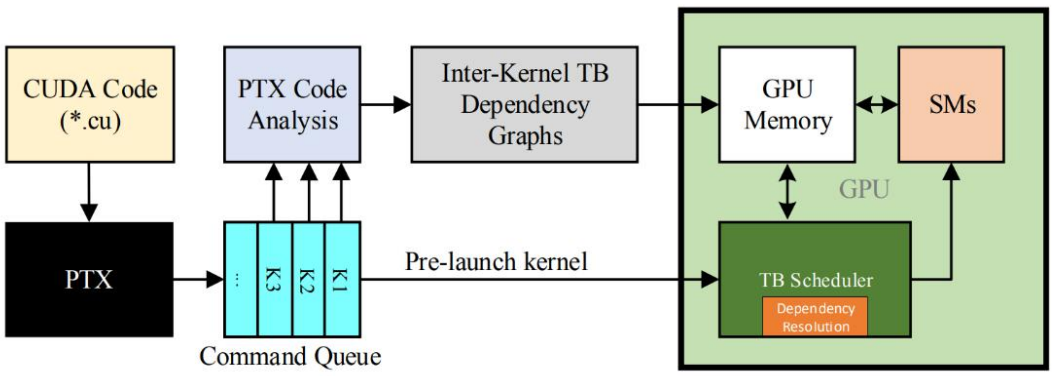
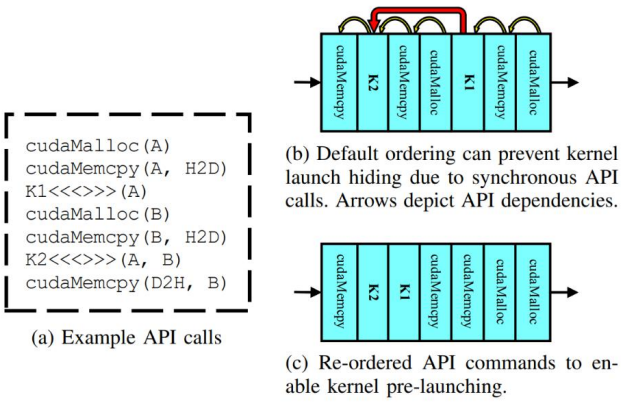


图5. BlockMaestro概览

核的线程块的执行。BlockMaestro 执行即时编译器静态分析（在内核启动时）来识别 read-after-write (RAW)全局内存中的依赖项。这些 RAW 依赖关系在运行时由线程块调度程序强制执行。在线程块粒度上识别 RAW 依赖关系的关键是识别每个线程块所触及的数组索引。

3.4.3 内核预启动

内核预启动是通过让下一个内核在前一个内核完成之前启动来实现的。为了实现这一点，需要允许多个内核同时从一个命令队列中执行，同时防止某些 CUDA API 调用阻塞命令队列或阻塞下一个 CUDA API 的发布。在图 6a 中，展示了 CUDA API 调用的一个示例。当主机执行代码并到达 CUDA API 调用时，它将调用连同必要的数据发送到命令队列(默认 CUDA Stream1)。常用的 CUDA API 调用包括分配内存、向 GPU 全局内存和主机内存传输数据、启动内核和设备同步。默认情况下，内核调用是异步(非阻塞)的。然而，CUDA 内存 API 调用是同步的(也就是说，host 在函数返回之前会被阻塞)，这可能会限制内核的预启动。如图 6 所示，当 K1 被启动并执行时，我们不能预先启动 K2，因为 cudaMalloc 和 cudaMemcpy 命令必须先完成。最大化内核启动隐藏的一个潜在解决方案是，识别命令队列中 API 之间的真正数据依赖关系，并重新排序命令，以最大化内核启动隐藏。这是通过移动内核启动位置的方式实现的。图 6c 显示了这样一个顺序，它仍然满足 API 调用之间的数据依赖关系。如果可以为内核分配内存，就可以启动内核。否则，它们将不得不等待资源变得可用。



(a) Example API calls

(b) Default ordering can prevent kernel launch hiding due to synchronous API calls. Arrows depict API dependencies.

(c) Re-ordered API commands to enable kernel pre-launching.

图6. 命令队列中API排序对内核启动隐藏的影响

图 7 展示了 BlockMaestro 的操作，图中包括了四个内核的启动：K1 到 K4，以及它们相应的线程块(标记为 0-2 的 K1，3-6 的 K2，7-8 的 K3，K4 缺省)。垂直黄条表示每个内核的启动开销。图 7a 显示了基准的 GPU 执行时间线，其中内核执行被序列化。在基准场景中，由于内核启动开销和依赖延迟的线程块导致 GPU 利用率不足，因此存在效率低下的问题。例如，即使 K2:5 和 K2:6 已经完成，K3:8 也要等到 K2 全部完成后才能开始。

图 7b 说明了内核的预启动，来隐藏内核启动的开销。在内核 K1 启动后，BlockMaestro 将预发布内核 K2。为了加强正确性并解决 K1 和 K2 之间的数据依赖关系，线程块调度程序保守地阻塞 K2 的执行，直到 K1 的所有块都完成。虽然消除了内核启动的开销，但依赖性暂停和资源利用不足仍然存在。

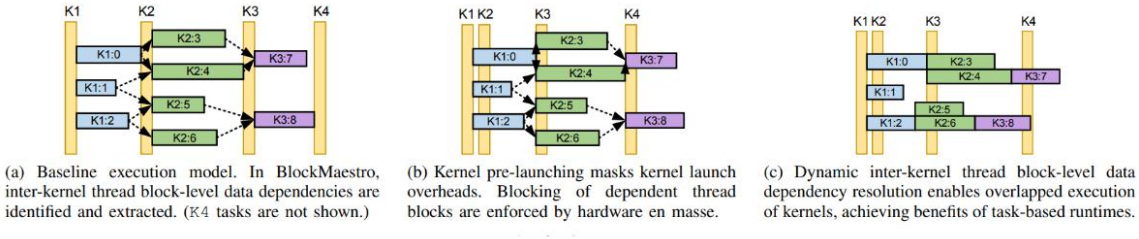
为了缓解这些问题，基于任务的运行时允许程序员通过动态创建任务并指定它们的依赖关系来表达任务执行。这允许块在满足依赖关系时开始执行可以避免使用持久内核的内核启动开销。在基于任务的运行时中，一旦 K2:5 和 K2:6 完成，K3:8 将能够立即执行。为了达到同样的目标，BlockMaestro 引入了内核预启动和内核间数据依赖解析来消除内核启动的开销，并使依赖内核的线程块能够重叠执行。

为了充分实现基于任务的执行的好处，在 PTX 到 SASS 发生即时编译时，进一步识别在内核启动时存在于依赖内核(图中有箭头注释)之间的线程块级数据依赖。图 7c 说明了内核间数据依赖关系解析，它利用依赖内核对之间的线程块级数据依赖信息。这些数据依赖关系在运行时由线程块调度程序强制执行，并动态解析。这使得任何准备好的线程块都可以开始执行，而不管它们在哪个内核中运行。

3.4.4 强制内核间依赖

内核间依赖是通过使用一个表示两部分依赖图的依赖列表来实现的。在 BlockMaestro 中，通过在硬件使用依赖列表缓冲区和父计数器缓冲区来加强内核间的依赖。

图 8 描述了 BlockMaestro 的 TB 调度器的硬件设置。当设备从主机接收到内核时，依赖项列表和初始父计数器是存储在全局内存中。因此，对于每一个(预)启动的内核，GPU 需要在全局内存中跟踪



(a) Baseline execution model. In BlockMaestro, inter-kernel thread block-level data dependencies are identified and extracted. (K4 tasks are not shown.)

(b) Kernel pre-launching masks kernel launch overheads. Blocking of dependent thread blocks are enforced by hardware en masse.

(c) Dynamic inter-kernel thread block-level data dependency resolution enables overlapped execution of kernels, achieving benefits of task-based runtimes.

图7. BlockMaestro内核预启动与依赖关系解析

依赖列表的基址和父计数器的基址。为了最小化全局内存访问量，在线程块调度器中包含了依赖列表缓冲区和父计数器缓冲区。依赖列表缓冲区对正在执行的线程块的依赖项进行跟踪，父计数器缓冲区对未执行的子线程块的依赖项进行跟踪。

当线程块被调度执行时，依赖项列表中的线程块条目被缓冲到依赖项列表缓冲区中。然后读取条目来识别子线程块。属性中不存在的项在父计数器缓冲区中，分配一个条目并获取子线程块的父计数器值。因为这个依赖项列表和父计数器条目中的信息在线程块完成执行之前是不需要的，所以这个缓冲过程不在关键路径上。当 TB 完成时，使用依赖项列表缓冲区标识每个子 TB ID，并将其索引到父计数器缓冲区中，以减少父计数器计数。当父核计数为 0 时，相应的子核就可以执行了。当父 TB 完成时，我们在依赖项列表缓冲区中释放一个条目，当子 TB 被选择执行时，我们在父计数器缓冲区中释放一个条目。

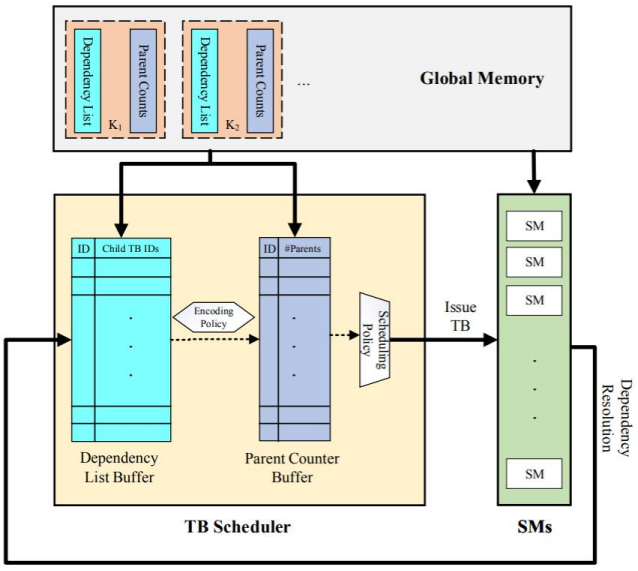


图8. BlockMaestro硬件设计

3.4.5 总结

文章的工作贡献如下：

• 建议内核预启动，以掩盖依赖内核的内核启动开销。此外，还引入了命令队列重新排序，以增加内核预启动的机会。

• 利用编译器的支持来提取现有 GPU 应用程序的内核间数据依赖，而不需要程序员的干预。由 SIMT 编程模型提供的定义良好的 GPU 应用程序结构允许以二分依赖图的形式提取数据依赖关系。

• 提出解决方案来解决内核间线程块之间的细粒度数据依赖。这确保了预启动内核的正确性，并使相关线程块在数据依赖项得到满足后立即开始执行。

4 总结与展望

（1）Flex 提出“零预留”，通过基础设施支持和分布式软件的结合实现了一个负责智能放置工作负载的离线组件和一个用于动态管理电源、可用性和性能的在线组件，极大的提升了数据中心的资源利用率，降低了数据中心的建设成本；

（2）LazyBatching 通过机器学习推理的方式，设计了一个智能批处理系统，该系统可以动态调整批处理级别，以满足延迟、吞吐量和 SLA 需求，实现了灵活的批处理，降低了数据中心的平均响应时间、提高了吞吐率、满足了 SLA 需求；

（3）SATORI 划分多核架构资源，是一种主动且同时控制多个 CMP 体系结构资源以实现多个目标的技术，通过牺牲短期的利益来换取长期的高效，有效地处理计算核心、LLC 方式、内存带宽和功率上限资源，实现了吞吐量和公平性之间的完美平衡；

（4）BlockMaestro 通过软硬件结合的方式，实现对程序员透明的基于任务的执行模型，通过追踪内核间的数据依赖关系来隐藏内核启动开销的影响，提高了 GPU 的利用率，加速了应用的执行。

这些方法从不同的角度出发，提高了数据中心的利用率，降低了数据中心的成本，提升了数据中心的性能。当然除了数据中心的利用率、建设成本、响应速度需要提升，还有很多问题需要考虑，比如说降低数据中心的能效比也是一个很重要的问题。虽然近年来数据中心蓬勃发展，但也存在着很多的问题，数据中心的发展还有很长的路可以走。

参 考 文 献

[1]Zhang C, Kumbhare A G, Manousakis I, et al. Flex: High-availability datacenters with zero reserved power[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021: 319-332.

[2]Choi Y, Kim Y, Rhu M. Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference[C]//2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2021: 493-506.

[3]Roy R B, Patel T, Tiwari D. Satori: efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021: 292-305.

[4]Abdolrashidi A A, Esfeden H A, Jahanshahi A, et al. Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021: 333-346.

[5]X. Fan, W.-D. Weber, and L. A. Barroso,“Power provisioning for a warehouse-sized computer,” in Proceedings of the 34th International Symposium on Computer Architecture, 2007.

[6]C.-H. Hsu, Q. Deng, J. Mars, and L. Tang,“Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters,” in Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems, 2018.

[7]S. Li, X. Wang, X. Zhang, V. Kontorinis, S. Kodakara, D. Lo,and P. Ranganathan,“Thunderbolt: Throughput-optimized, quality-of-service-aware power capping at scale,” in Proceedings of the 14th Symposium on Operating Systems Design and Implementation, 2020.

[8] Y. Li, C. R. Lefurgy, K. Rajamani, M. S. Allen-Ware, G. J. Silva,D. D. Heimsoth, S. Ghose, and O. Mutlu,“A Scalable Priority-Aware Approach to Managing Data Center Server Power,” in Proceedings of the International Symposium on High Performance Computer Architecture,2019.

[9] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza,and Y. J. Song,“Dynamo: Facebook’s data center-wide power management system,” in Proceedings of the 43rd International Symposium on Computer Architecture, 2016.

[10] K. McCarthy and V. Avelar,“Comparing ups system design configurations,” APC white paper 75, Schneider electric Data center science center, 2016.

[11] W. P. Turner IV, J. PE, P. Seader, and K. Brill,“Tier classification define site infrastructure performance,” Uptime Institute, vol. 17, 2006.

[12] M. Wiboonrat,“An optimal data center availability and investment trade-offs,” in Proceedings of the 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008.

[13] M. Abdel-Majeed, D. Wong, and M. Annavaram,“Warped gates: Gating aware scheduling and power gating for gpgpus,” in 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),2013, pp. 111–122.

[14] M. Abdel-Majeed, D. Wong, J. Kuang, and M. Annavaram,“Origami:Folding warps for energy efficient gpus,” in Proceedings of the 2016 International Conference on Supercomputing, ser. ICS’16. New York,NY, USA: Association for Computing Machinery, 2016. [Online].Available: https://doi.org/10.1145/2925426.2926281

[15] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. AbuGhazaleh,“Corf: Coalescing operand registerfile for gpus,” in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM,2019, pp. 701–714.

[16] H. A. Esfeden, A. Abdolrashidi, S. Rahman, D. Wong, and N. AbuGhazaleh,“Bow: Breathing operand windows to exploit bypassing in gpus,” in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020, pp. 996–1008.

[17] H. Jeon, H. A. Esfeden, N. B. Abu-Ghazaleh, D. Wong, and S. Elango,“Locality-aware gpu registerfile,” IEEE Computer Architecture Letters,vol. 18, no. 2, pp. 153–156, 2019.

[18] G. Chen and X. Shen,“Free launch: optimizing gpu dynamic kernel launches through thread reuse,” in Proceedings of the 48th International Symposium on Microarchitecture, 2015, pp. 407–419.

[19] I. El Hajj, J. G ómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m.Hwu,“Klap: Kernel launch aggregation and promotion for optimizing dynamic parallelism,” in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2016, pp. 1–12.

[20] T. H. Hetherington, M. Lubeznov, D. Shah, and T. M. Aamodt,“Edge:Event-driven gpu execution,” in 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE,2019, pp. 337–353.

[21] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong,“Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus,” in 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 50), 2017, pp.600–611

[22] M. E. Belviranli, S. Lee, J. S. Vetter, and L. N. Bhuyan,“Juggler:a dependence-aware task-based execution framework for gpus,” in Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2018, pp. 54–67.

[23] A. E. Helal, A. M. Aji, M. L. Chu, B. M. Beckmann, and W.-c.Feng,“Adaptive task aggregation for high-performance sparse solvers on gpus,” in 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2019, pp. 324–336.

[24] Neeraj Kulkarni, Gonzalo Gonzalez-Pumariega, Amulya Khurana, Christine A Shoemaker, Christina Delimitrou, and David H Albonesi. Cuttlesys:Data-driven resource management for interactive services on reconfigurable multicores. In MICRO. IEEE, 2020.

[25] Iyswarya Narayanan, Adithya Kumar, and Anand Sivasubramaniam.Pocolo: Power optimized colocation in power constrained environments.In 2020 IISWC, pages 1–12. IEEE, 2020.

[26] Ewa Deelman, Christopher Carothers, Anirban Mandal, Brian Tierney,Jeffrey S Vetter, Ilya Baldin, Claris Castillo, Gideon Juve, Dariusz Kr ´ol, Vickie Lynch, et al. PANORAMA: An Approach to Performance Modeling and Diagnosis of Extreme-Scale Workflows. The International Journal of High Performance Computing Applications, 31(1):4–18, 2017.

[27] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M Shipman, and Chita R Das. D-Factor: A Quantitative Model of Application Slow-Down in Multi-Resource Shared Systems. ACM SIGMETRICS Performance Evaluation Review, 40(1):271–282, 2012.

[28] AMD,“Atmi (asynchronous task and memory interface),” https://github.com/RadeonOpenCompute/atmi, 2016, accessed: 2020-06-11.

[29] S. Chatterjee, M. Grossman, A. Sbˆırlea, and V. Sarkar,“Dynamictask parallelism with a gpu work-stealing runtime system,” in International Workshop on Languages and Compilers for Parallel Computing.Springer, 2011, pp. 203–217.

[30] NVIDIA,“Getting started with cuda graphs,” https://devblogs.nvidia.com/cuda-graphs/, 2018, accessed: 2019-11-17.

[31] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli,X. Teruel, P. Unnikrishnan, and G. Zhang,“The design of openmp tasks,”IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 3,pp. 404–418, 2009.