Computer Science and Application 计算机科学与应用, 2023, 13(5), 953-963
Published Online May 2023 in Hans. https://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2023.135093

Hans汉斯

基于组件的应用中的动态GPU资源管理

罗　婧，魏　雄

武汉纺织大学，计算机与人工智能学院，湖北　武汉

收稿日期：2023年4月6日；录用日期：2023年5月8日；发布日期：2023年5月17日

摘　要

在边缘和云环境中，使用图形处理单元(GPU)作为高速并行计算设备可以提高计算密集型应用程序的性能。随着要处理的数据量和复杂性的增加，多个相互依赖的组件序列在GPU上共存并共享GPU资源。由于缺乏用于动态GPU资源分配的低开销和在线技术会导致GPU使用不平衡并影响整体性能，提出了高效的GPU内存和资源管理器。管理器通过使用共享内存和动态分配部分共享GPU资源来提高整体系统性能。评估结果表明，与默认GPU并发多任务处理相比，动态资源分配方法能够将具有各种并发组件数的应用程序的平均性能提高29.81%。同时，使用共享内存可使性能提高2倍。

关键词

GPU，资源管理，空间多任务处理，优化

Dynamic GPU Resource Management in Component-Based Applications

Jing Luo, Xiong Wei

School of Computer and Artificial Intelligence, Wuhan Textile University, Wuhan Hubei

Received: Apr. 6th, 2023; accepted: May 8th, 2023; published: May 17th, 2023

Abstract

In edge and cloud environments, using graphics processing units (GPUs) as high-speed parallel computing devices can improve the performance of computing intensive applications. As the amount and complexity of data to be processed increases, multiple interdependent component sequences coexist on the GPU and share GPU resources. Due to the lack of low overhead and online technology for dynamic GPU resource allocation, which can lead to uneven GPU usage and affect overall performance, an efficient GPU memory and resource manager is proposed. The manager

文章引用: 罗婧, 魏雄. 基于组件的应用中的动态 GPU 资源管理[J]. 计算机科学与应用, 2023, 13(5): 953-963.
DOI: 10.12677/csa.2023.135093

improves overall system performance by using shared memory and dynamically allocating partial shared GPU resources. The evaluation results indicate that the dynamic resource allocation method can improve the average performance of applications with various concurrent component counts by 29.81% compared to the default GPU concurrent multitasking processing. At the same time, using shared memory can increase performance by 2×.

Keywords

GPU, Resource Management, Spatial Multitasking, Optimization

1. 引言

具有通用计算支持的图形处理单元(GPU)正在成为边缘和云部署的重要组成部分。GPU 的大规模并行计算能力允许加速数据密集型应用程序。例如，与传统 CPU [1]相比，在自动驾驶应用中的路径规划等应用中使用 GPU 可以加速实时计算。

随着 GPU 更积极地用于基于组件的应用程序，这类应用程序的整体性能取决于数据传输开销和序列中每个组件的性能。管理组件对共享 GPU 资源的竞争性使用面临着各种挑战。为了提升 GPU 的整体性能，本文提出并评估了一个解决方案：在基于组件的应用程序中，组件之间使用共享的 GPU 内存，从而消除了频繁和不必要的数据拷贝的需要。因此，它甚至可以适用于实时系统，不需要任何离线基准测试或测试案例。同时，将组件限制在其各自的最佳资源下，并将高比例的资源分配给其他资源密集型的组件，可以使整个基于组件的应用程序的性能得到改善，并可以防止内部排队和输入损失。

2. 基于组件的应用程序

如今的云应用程序通常被分解为组件或微服务，以获得更好的可扩展性和维护性。对于优化的部署，通常会使用位置感知来安排组件。最先进的 GPU 支持多任务处理[2]，使多个组件能够使用相同的 GPU，并且与为组件使用单独的 GPU 相比，有可能显著提高性能和成本。

尽管基于组件的应用程序中的组件与功能无关，但它们可能会遇到不同类型的依赖关系[3]。本文使用具有线性序列依赖关系的基于组件的程序，其中每个组件必须等待前一个组件的执行，然后才能开始自己的处理。同时，不同的输入可以并且应该由组件并行处理。例如，如图 1 所示，基于实时组件的对象检测应用程序由具有线性依赖性的组件链组成。首先解码组件从实时摄像机流接收帧，并且增强组件必须等待结果才能开始处理。类似地，对象识别必须等待给定帧的增强。然而，不同的帧可以由组件并行处理。

GPU 通常被用作后备加速器，以加速应用程序的某些部分，同时也由 CPU 进行管理。GPU 上运行的组件被称为内核。在典型的设置中，应用程序输入在 CPU 上接收，并通过 PCIe 总线传输到 GPU。执行后，结果被带回 CPU，并传输到下一个处理步骤。在基于组件的应用程序中，使用单个 GPU 在 CPU 和 GPU 之间多次复制数据会给处理增加大量开销，也会不必要地加载 PCIe 总线。最先进的 GPU 框架支持进程间内存共享的构建块，从而能够避免不必要的数据拷贝。

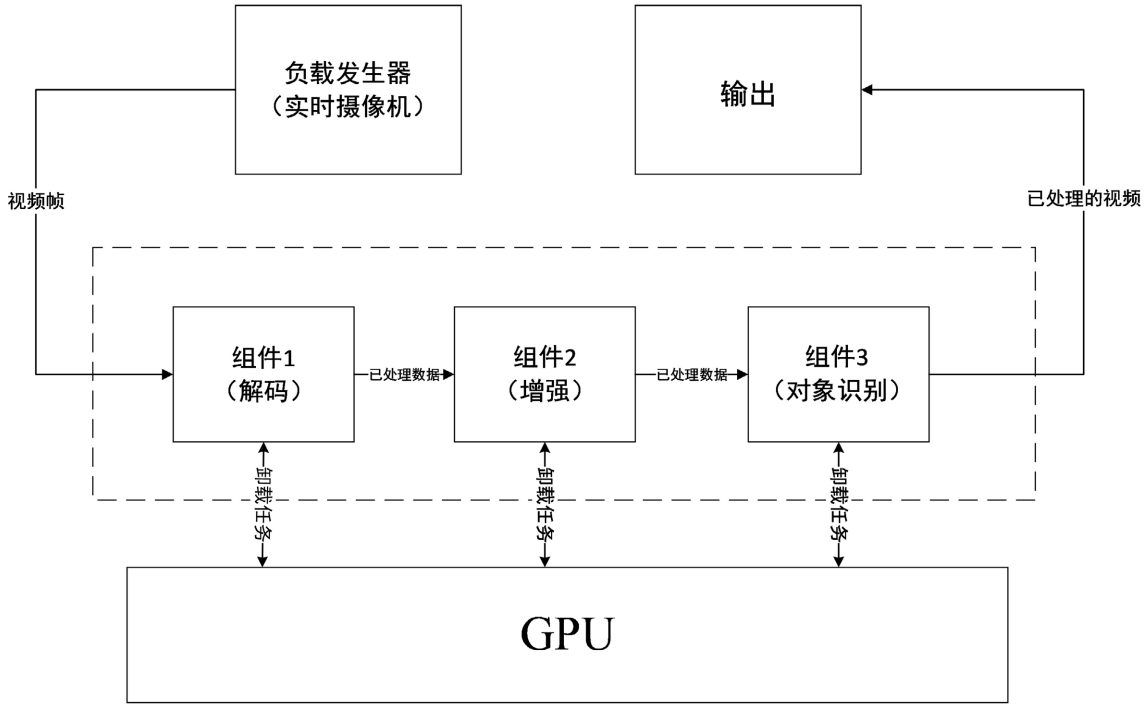

Figure 1. Components running concurrently on the GPU
图 1. GPU 上并发运行的组件

虽然单个 GPU 可以提高数据密集型应用程序的性能，但要达到最大容量，就需要适当的资源共享配置。主要挑战之一是管理可能需要不同资源的组件对共享 GPU 资源的竞争性使用。向一个组件分配比所需的资源更多的资源可能会导致资源的浪费，同时也可能增加其他组件的延迟时间。

为了有效地管理并发组件之间的 GPU 资源，时态多任务技术允许组件通过在它们之间拆分 GPU 时间来顺序访问 GPU 资源。如果另一个组件需要 GPU 资源，它必须等待正在运行的组件完成或通过抢占获取资源[4] [5] [6]除了抢占开销外，这种技术可能会限制 GPU 利用率，从而限制整体性能，因为只允许一个组件占用 GPU 资源。在具有线性依赖关系的基于组件的应用程序的特定环境中，抢占组件会导致延迟在组件链中传播。当前的 GPU (如 NVIDIA、AMD、Intel)在多个应用程序(更具体地说是 GPU 上下文)同时执行时，默认情况下实现时态多任务。

另一种方法是空间多任务处理，它在并发组件之间分配 GPU 资源[7]。这种技术允许组件在 GPU 上同时执行，方法是将处理器的一部分分配给每个组件。通过对资源进行分区来同时占用 GPU，可以提高基于组件的应用程序的性能。NVIDIA 通过其多进程服务(MPS)为不同组件实现空间多任务和流式多处理器(SM)共享[8]。

虽然空间多任务是实现高性能的正确方法，但为并发组件选择处理器部分的策略是具有挑战性的。现有的方法主要是基于保证每个应用程序的服务质量(QoS)或尊重的优先级的内核[9]。然而，在基于组件的应用程序中，所有的组件具有相同的优先级，必须考虑整个系统的服务质量，而不是单个组件。此外，现有的方法分区处理器的基础上离线基准测试的每个单独的内核[10]。然而，由于基于组件的应用程序中的组件的相互依赖性，一个动态的和在线的分区预计会产生更好的结果。

3. 研究方法

为了解决共享 GPU 资源的竞争使用问题，本文描述了一种在应用程序组件之间分配和共享 GPU 资

源的方法。该方法由三部分组成：内存管理器、在线性能优化器和 GPU 资源管理器。图 2 示出了该体系结构。

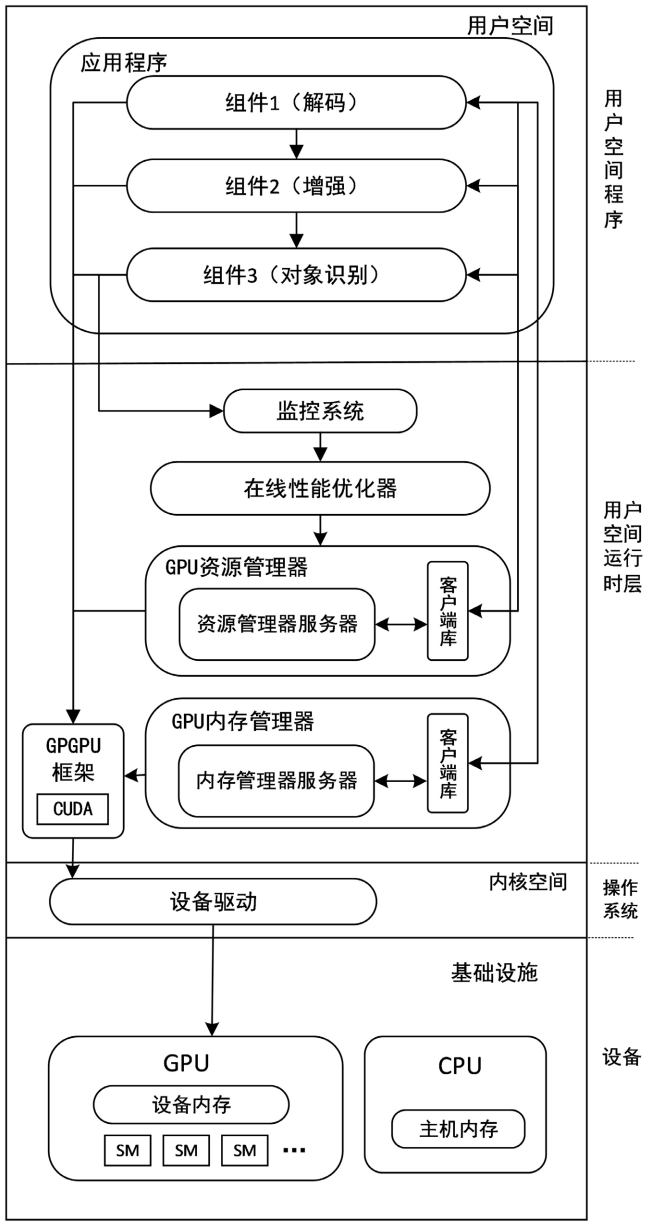

Figure 2. System architecture
图 2. 系统架构

如图 2 所示，在最高层有在 CPU 上执行的应用程序组件，并使用 GPU 作为加速器。该系统是在用户空间运行时层，并使用 NVIDIA CUDA，一种由 NVIDIA 推出的通用并行计算架构作为通用图形处理器(GPGPU)框架。CUDA 应用程序编程接口(CUDA API)允许对 GPU 编程、管理计算资源(SMs)和 GPU 内存。这些接口使用 GPU 驱动程序在 GPU 上运行应用程序。本文还依靠 NVIDIA MPS 在同一 GPU 上高效地启用多个应用程序。虽然这里依赖于 NVIDIA，本文提出的系统也可以实现在其他的通用图形处理器框架。

3.1. 在线性能优化器

在线性能优化器旨在为组件和给定的输入流找到合适的资源共享配置。优化器机制通过分配与工作负载和资源需求相关的最佳资源组合来最大化总体性能。

如图 2 所示，优化器依赖于监控系统。本文专门为基于组件的应用程序的整体吞吐量进行优化。从应用程序的最后一个组件收集周期性吞吐量信息，例如周期可以是 100 ms。在实践中，实际吞吐量由组件的复杂性决定，这可能会限制最低可能的周期。基于监控输入，在线性能优化器定期做出分配决策，以找到最佳的 GPU 资源分配，从而获得最高的总体性能。在线性能优化器的结果用作向 GPU 资源管理器中的组件动态分配资源的输入。

对在 GPU 上执行的并发组件的资源分配应该与它们的工作量要求和对整个基于组件的应用的贡献成正比。通常情况下，每个组件都需要特定部分的 SMs 来拥有峰值性能，分配超过该部分的 SMs 并不能进一步提高其处理特定输入的性能。另外，限制某些组件的资源，即使在其最佳状态下，也可能使资源密集型组件获得更多资源，从而提高整体性能。这也可以防止内部排队的问题，如果一个资源密集度较低的组件处理的输入比其进行的组件多，就会发生内部排队的问题。

在默认情况下，空间多任务为所有并发的组件分配相同的资源，结果可能无法满足组件的要求，特别是高密集型组件(应用链的瓶颈)。在这方面，所提出的机制的目标是为所有组件找到最佳分配，通过最小化非密集型组件的资源分配和使高密集型组件使用大部分的资源，使链的总体性能最大化。工作负载感知动态资源分配包括两个主要步骤：1) 找到明显的瓶颈部件，并为其分配最大的资源；2) 确定其余部件的 SM 分配比例，使整体性能最大化。

3.1.1. 指定瓶颈组件

链中的组件由两个参数表征：内核执行时间(即使用资源的时间)；所需的 GPU 线程数。对于 GPU，后者通常在两个级别上表示，一个块中的线程数和一个资源网格中的块数。基于执行时间或 GPU 线程做出决策需要详细的分析方法，而且通常影响所用线程数量的输入大小可能事先不知道[11]。

因此，为了找到瓶颈构件，本文提出了一种基于各个构件对整体性能的影响进行决策的方法。链的总体性能始终取决于高工作负载组件。减少分配给这些组件的资源会显著降低性能，利用这个特性找到瓶颈组件。

基于 NVIDIA 多进程服务(MPS)文档[8]，每个组件的最佳 SM 百分比为$100\%/(0.5\mathrm{N})$，其中 N 是在 GPU 上同时运行的组件数量。然而分析表明，这种方法不会产生高性能，特别是在组件中有不同工作负载的情况下。第一步，在推出的时候，该机制将相等的 SMs 百分比 R_{eql} 分配给所有组件，如公式(1)所示。

$$R_{eql} = \left(R_{SM}/N\right) \tag{1}$$

其中，R_{SM} 表示所有可用资源的 200%，以便当有空闲资源时，给 GPU 调度器更多的自由来重叠组件之间的执行[8]；N 代表链中的组件数。

下一步，该算法反复检查每个特定组件对整体性能的影响。每个组件的分配资源都受到很大限制。在这种情况下，所有组件都可能降低整体性能。但是瓶颈组件的影响明显高于其他组件。如果资源有限，则该组件被标记为瓶颈组件。整体性能降低了预定义的阈值。在线优化器对接收到的性能报告进行比较，以在这方面做出决定。

3.1.2. 确定 SM 分配百分比

在确定瓶颈后，还需求出所有组件的 SM 分配。与其他组件相比，瓶颈组件需要更多的资源，因此该方法授予它们对所有 SM 的访问权。这一点很重要，因为在实践中，组件可能不会在 GPU 上并发执行，

限制它们的 SM 使用直接影响应用的性能。

对于其余的组件，被认为是 100%的资源 R 被平均分配，如公式(2)和(3)所示。R_C 表示每个非瓶颈组件的分配资源。

$$R = \sum_{N_C} (R_C) \tag{2}$$

$$R_C = (100/N_C) \tag{3}$$

在公式(3)中，N_C 代表非瓶颈组件的数量。

将有限的资源分配给非瓶颈组件，可以防止它们在并发运行的情况下限制瓶颈组件的资源。在这一点上，有一个合理的组件的 SM 分配比例。然而，通过对非瓶颈组件的分配进行微调，可以提高整体性能。提出算法 1，以改善每个非瓶颈组件的分配资源(R_C)，以实现更好的性能。

算法 1：确定 SM 百分比的算法

输入 L_C: {非瓶颈组件列表}
输入 N_C: 非瓶颈组件的数量
输入 $L_{SM}[N_C]$: {每个组件的预定义 SM 部分的列表}
输入 N_{SM}: 预定的 SM 部分的数量
输入 performance: 当前性能
输出 C_A: 分配组件 SM 部分
    for i <- 1,..., N_C do
        C_{min} <-从 L_{SM}[i]中查找具有最小 SM 的 SM 部分
        C_{max} <-从 L_{SM}[i]中查找具有最大 SM 的 SM 部分
        C_{RC} <-从 L_{SM}[i]中查找具有 R_C 数值 SM 的 SM 部分
        C_A <- C_{max}
     newPerformance <-从在线性能优化器获得 C_A 的性能
      if(newPerformance - performance) > D_{thr} then
            从 L_{SM}[i]中删除小于 C_{RC} 的 SM 部分
            C_{max} <- C_A; C_A <- mid(C_{RC}, C_A); C_{min} <- C_{RC};
      else(newPerformance - performance) <= D_{thr} then
            从 L_{SM}[i]中删除大于 C_{RC} 的 SM 部分
            C_A <- mid(C_{RC},C_{min}); C_{max} <- C_{RC}; C_{min} <- C_{min};
       while L_{SM} [i] != empty do
            newPerformance <-从在线性能优化器获得 C_A 的性能
            if(newPerformance - performance) > D_{thr} then
                从 L_{SM}[i]中删除小于 C_{RC} 的 SM 部分
                C_{min} <- C_A; C_A <- mid(C_A, C_{max}); C_{max} <- C_{max};
            else (newPerformance - performance) <= D_{thr} then
                从 L_{SM}[i]中删除大于 C_{RC} 的 SM 部分
                C_{max} <- C_A; C_A <- mid(C_{min}, C_A); C_{min} <- C_{min}
       Return C_A

该算法根据非瓶颈组件的数量重复进行。对于每个非瓶颈组件，定义一组 SM 资源的不同部分。这些部分的范围在 5%到 100%之间。N_{SM} 代表每个组件预先定义的 SM 部分的数量，尝试所有 SM 部分的组合是相当耗时的。由于有 N_C 个非瓶颈组件和 N_{SM} 个 SM 部分，所以有 $N_{SM}^{N_C}$ 组合。该算法定义了最大变量 C_{max} 和最小变量 C_{min} 以及 C_{RC}，它们分别是具有最大 SM 的部分，具有最小 SM 的部分，以及具有 R_C 数值的 SM 的部分。之后，该算法将分配给组件的 SM 部分从 C_{RC} 改为 C_{max}。并使用优化器报告将新的性能与以前的性能进行比较。定义了一个阈值 D_{thr} 来比较新的性能。

如果新的 SM 部分对性能的改善超过了 D_{thr}，算法就会在 C_{RC} 和 C_{max} 之间的范围内寻找合适的 SM 部分。该算法删除了所有 SM 百分比小于 R_C 的部分。否则，它发现分配更多的 SM 不会影响性能。在这种

情况下，该算法试图找到该组件是否甚至需要更少的 SM。它设定 C_{RC} 和 C_{min} 之间的范围，并在这个范围内寻找合适的 SM 部分。每次它都将 SM 部分改为 C_{mid}，以寻找其对整体性能的影响。C_{min} 代表在每个选定范围内具有中间 SM 百分比的 SM 部分。该算法将继续进行，直到所有的 SM 部分选项被删除，并且为该组件设置了适当的 SM 部分。该算法在为所有非瓶颈组件找到适当的 SM 部分后完成。在最坏的情况下，算法的性能是 $O(N_C \times \log N_{SM})$，比原来的 $O(N_{SM}^{N_C})$小得多。

值得一提的是，所描述的方法在某些条件下可能会被定期重新执行，以确保一致的优化性能。例如，如果在设置了确定的最佳配置后，发现性能突然发生变化。

3.2. GPU 资源管理器

为了将 GPU 资源的特定部分分配给组件，设计了 GPU 资源管理器。它从在线性能优化器接收 SM 分配配置，并执行组件的资源限制。为了加强这些限制，会使用特定数量的 SMs。

创建各种 GPU 上下文。由于创建上下文非常耗时，要为每个应用程序预定义了一个具有不同 SM 配置的上下文池。GPU 资源管理器可能会根据在线性能优化器的结果更改组件的上下文以在执行期间增加或限制其计算资源。使用这种技术，实现了重新分配的 SMs 动态的组件的目标。而且，如果未来 NVIDIA 的多进程服务工具将允许对应用程序进行动态 SM 分配，那么解决方案将变得更容易实现，而无需上下文。

3.3. GPU 内存管理器

GPU 内存管理器被设计成一个专门的中央系统，以减少传输数据的延迟。该系统由两个主要部分组成：内存管理器服务器和客户端库。

内存管理器服务器使用共享的 GPU 内存，以消除 CPU 和 GPU 之间频繁的数据复制，减少组件之间的数据传递。进程间内存句柄作为分配内存的指针，负责在组件链中共享内存。作为第一步，内存管理服务器从第一个组件接收由所需数据大小组成的分配请求。第一个组件可能会保留比自己所需更多的内存，以便为其他组件保留额外的内存。通过使用 CUDA API，内存管理服务器分配特定大小的内存和与之相关的 GPU 内存要求(句柄)，并将句柄输出给组件。该组件将数据复制到所分配的 GPU 内存中，在完成对数据的 GPU 执行后，该组件将句柄传递给下一个组件。一个句柄对象代表已经被转移到 GPU 内存的数据。其他组件可以使用该句柄访问该内存。在完成所有组件的执行后，处理后的数据或结果，作为链的最终输出，被最后一个组件带回 CPU。最后，最后一个组件负责请求内存管理服务器释放分配的内存和 GPU 上的句柄。

拥有一个独立的 GPU 内存管理器的关键因素是，即使在个组件发生故障的情况下，保留的共享内存仍然可以在其他组件中使用。此外，管理器可能会定期迭代正在使用的句柄，并释放已分配太长时间的句柄，假定关联的句柄由于组件故障而丢失。

GPU 内存管理器允许仅在组件之间传输句柄，而不是将所有数据发送到下一个组件。句柄的固定大小(CUDA 的情况下为 64 字节)使得组件之间传输数据的延迟与数据大小无关。这种方法的另一个优点是系统中 PCIe 总线的使用率显著降低。

4. 系统实现

为了评估本文提出的系统并测量其性能，实现了一个系统，包括 GPU 内存管理器，在线性能优化器 GPU 资源管理器以及与它们通信的相关库。还实现了一个基于组件的应用程序，包括在 GPU 上并发执行的组件链。在本节中介绍了实现的一些细节。

GPU 内存管理器和 GPU 资源管理器的接口是使用基于原生的 HTTP 协议的面向资源的接口设计方

法(HTTP REST)实现的。

4.1. 内存管理器服务器和客户端库

为了在组件之间共享 GPU 内存，进程间通信(IPC)被用作已分配内存的句柄对象。本方法只是在使用用户数据报协议(UDP)发送的消息中交换处理对象。客户端可以通过接口分配和释放内存。

虽然分配句柄的过程是一个轻量级的操作，但是从 GPU 分配内存是非常耗时的。在这方面，设计了两个内存池：一个非活动池和一个活动池。从第一个组件接收到分配请求后，存储管理使用所需的数据大小分配一组句柄，并将它们存储在非活动池中。对于每个分配请求，从非活动池中检索一个空闲句柄，而无需从 GPU 分配的开销。当前正在使用的句柄存储在活动池中。在有一个空的非活动池的情况下，内存管理从 GPU 分配一组新的内存。根据分配请求的速率改变活动池容量和非活动池容量。在接收释放内存请求的情况下，句柄从活动池移动到非活动池。最后，如果有一个全容量的非活动池，它将被移除。此外，在第一个组件使用的库中实现了句柄池，以减少与服务器通信的开销。这个池通过存储一组来自内存管理器的句柄来消除开销，这些句柄可用于新的输入。

4.2. 资源管理器服务器和客户端库

GPU 资源管理器实现了两个主要接口，包括 SM 配置接口和 SM 建议接口。数据存储用于存储在线性能优化器的性能信息和相关 SM 配置。在线性能优化器使用 SM 配置接口报告配置决策。为了分配特定的 SM 部分，组件使用具有特定 SM 配置的上下文。为每个组件创建一个具有不同 SM 部分的预定义上下文池，以消除在更改 SM 部分时初始化上下文的开销。组件使用 SM 建议接口向资源管理器服务器请求适当的 SM 百分比，从而在预定义上下文之间切换。

5. 实验

在本节中，评估了本文提出的方法，并分两部分给出了实验结果。在第一部分中，评估了旨在减少数据传输开销的 GPU 内存管理器。在第二部分中，从吞吐量比较的角度给出了动态 GPU 资源分配的分析结果。

实验使用具有 108 个 SM 的 NVIDIA A100 GPU 进行。在一系列相互依赖的组件上进行评估，这些组件将输入矩阵相乘。在输入数据大小和内核复杂度的情况下，这些组件被设计为可配置组件。因此，系统可以在具有不同内核特性的各种工作负载下进行评估。配置细节如表 1 所示。一个内核函数的全部线程块构成一个网格(grid)，而线程块的个数就记为网格大小(grid size)。每个线程块中含有同样数目的线程，该数目称为线程块大小(block size)。由于 A100 GPU 由 108 个 SM 组成，因此每个线程块具有 32 × 32 个线程、线程网格(grid)大小超过 10 × 10 的内核能够利用所有资源。因此，内核特性被定义为四类，包括小型、中型、大型和超大计算密集型内核。

Table 1. Compute configuration
表 1. 计算配置

线程网格大小(块)	线程块中线程数(个)	输入矩阵	内核类型	描述
4 × 4	32 × 32	120 × 120	S	小型计算密集型
8 × 8	32 × 32	240 × 240	M	中等计算密集型
12 × 12	32 × 32	360 × 360	L	大型计算密集型
16 × 16	32 × 32	480 × 480	XL	超大计算密集型

为了评估使用共享 GPU 内存的影响，将使用 GPU 内存管理器的链(称为内存共享链)的吞吐量与基准方法进行了比较。在基准方法中，数据以常规 UDP 消息的形式传输。使用链中不同数量组件的小型计算配置的执行结果如图 3 所示。吞吐量的比较表明，当有两个以上组件时，使用 GPU 内存管理器会使这些测试用例的吞吐量提高两倍以上。考虑到链的第一个和最后一个组件负责与内存管理器服务器通信并将数据复制到 GPU 或从 GPU 复制数据，在具有两个组件的链中，开销抵消了使用共享 GPU 内存的优势。在具有两个以上组件的链中，这种开销可以忽略不计。

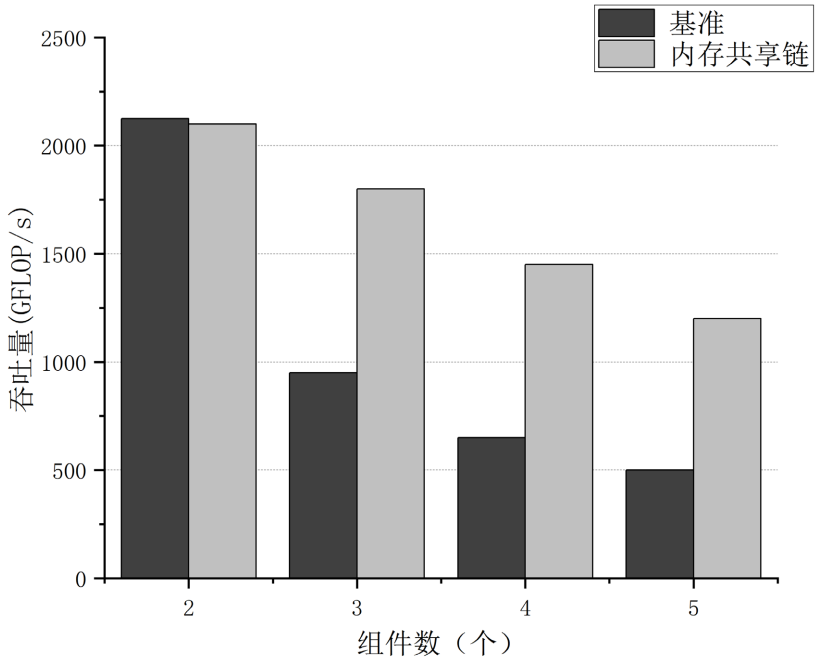

Figure 3. Chain throughput comparison of different sizes of chain
图 3. 不同规模链条的吞吐量对比

为了评估动态资源分配算法(DRA)，将算法的结果与作为 GPU 默认空间共享资源的多进程服务(MPS)算法的结果进行了比较。在这两种情况下，GPU 内存管理器都用于内存共享。设置了不同的链集，由五个不同配置的部件组成。表 2 显示了每组组件的规格。

Table 2. Components configuration
表 2. 组件配置

链集	计算负载级别	计算配置组合
SC-1	小	L, S, S, S, S
SC-2	小	S, S, S, XL, S
MC-1	中等	S, L, S, XL, S
MC-2	中等	S, L, M, S, S
MC-3	中等	M, S, XL, S, S
LC-1	大	M, S, XL, S, L
LC-2	大	M, L, XL, S, S
XLC-1	特大	S, L, M, M, L
XLC-2	特大	S, L, M, M, XL

评估寻找瓶颈和分配最佳 SM 部分的能力，以便通过在链中拥有不同的计算密集型组件级别来优化吞吐量。还设置了具有不同工作负载级别的不同组件组合。在这方面，所有定义的集合被分为四大类：小、中、大和超大的计算密集型链。将结果与默认的 CUDA MPS 方法进行比较，其中所有组件都可以访问完整的 GPU，如图 4 所示，与 MPS 方法相比，该方法的吞吐量提高了 29.81%。

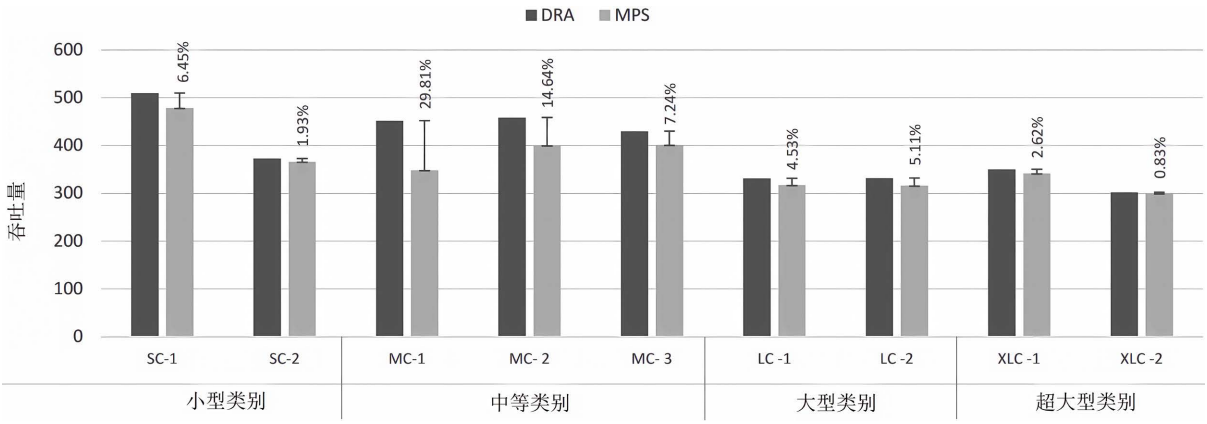

Figure 4. Chain throughput comparison over MPS for different workload combinations of components in the chain of 5 components
图 4. 5 个组件链中组件的不同工作负载组合在 MPS 上的链吞吐量比较

考虑到 GPU 中的可用 SMs，计算密集程度超过 S 的组件有可能成为链的瓶颈。瓶颈的复杂程度和链中现有瓶颈的数量直接影响到链的资源需求。增加这些项目中的每一项都会增加链的计算密集程度和组件之间的资源争夺。该算法提高了该类别的吞吐量达 6.45%。在中等类别中，组件之间的争夺随着资源需求的增加而增加。所提出的方法导致了最好的改进，在这个类别中为 29.81%。在大型和超大型类别中，链上有两个以上的瓶颈时，吞吐量的改善是最小的，分别达到 5.11%和 2.62%。由于计算，组件的资源需求超过了可用资源。因此，无法通过将更多的资源转移到瓶颈上来显著提高吞吐量。

6. 结论

本文提出了一种 GPU 内存共享解决方案和 GPU 资源管理器，以优化基于组件的线性依赖应用的吞吐量。同时，演示了在单个 GPU 上为基于组件的应用程序使用共享内存提供了显著的性能好处。GPU 资源管理器方法根据组件的工作负载动态地在并发组件之间分配 GPU 计算资源。实验结果表明，与 MPS 算法相比，该算法的应用吞吐量提高了 29.81%。相比其他方法都得到了很大的提升，但仍有不足之处，比如所提出的方法是基于一个单一的 GPU 与给定数量的 SMs，然而基于组件的应用程序可能会在一个或多个服务器中的多个 GPU 上部署和执行，以加速工作负载。在这种情况下，它不会改变应用的逻辑，同时监视整个吞吐量。单个 GPU 在线性能优化器也能够处理此类部署，因为它可以访问与每个 GPU 关联的 GPU 资源管理器。在未来的工作中，将会进一步加快基于组件的应用程序主要在多个 GPU 上的执行速度，并解决使用多个 GPU 优化性能可能面临的挑战。

参考文献

[1] Capodieci, N., Cavicchioli, R., Bertogna, M., *et al.* (2018) Deadline-Based Scheduling for GPU with Preemption Support. 2018 *IEEE Real-Time Systems Symposium* (*RTSS*), Nashville, TN, USA, 11-14 December 2018. https://doi.org/10.1109/RTSS.2018.00021

[2] Adriaens, J.T., Compton, K., Kim, N.S., *et al.* (2012) The Case for GPGPU Spatial Multitasking. *IEEE International Symposium on High Performance Computer Architecture*, New Orleans, LA, USA, 25-29 February 2012.

https://doi.org/10.1109/HPCA.2012.6168946

[3] Nardin, I., Righi, R., Lopes, T., *et al.* (2021) On Revisiting Energy and Performance in Microservices Applications: A Cloud Elasticity-Driven Approach. *Parallel Computing*, **108**, Article ID: 102858. https://doi.org/10.1016/j.parco.2021.102858

[4] Chen, L., Zigerelli, A., Yang, J., *et al.* (2018) A Dynamic and Proactive GPU Preemption Mechanism Using Checkpointing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **39**, 75-87. https://doi.org/10.1109/TCAD.2017.2705154

[5] Garg, S., Kothapalli, K. and Purini, S. (2018) Share-a-GPU: Providing Simple and Effective Time-Sharing on GPUs. 2018 *IEEE* 25*th International Conference on High Performance Computing* (HiPC), Bengaluru, India, 17-20 December 2018. https://doi.org/10.1109/HiPC.2018.00041

[6] Zhao, C., Gao, W., Nie, F., *et al.* (2022) A Survey of GPU Multitasking Methods Supported by Hardware Architecture. *IEEE Transactions on Parallel and Distributed Systems*, **33**, 1451-1463. https://doi.org/10.1109/TPDS.2021.3115630

[7] Liang, Y., Huynh, H.P., Rupnow, K., *et al.* (2015) Efficient GPU Spatial-Temporal Multitasking. *IEEE Transactions on Parallel and Distributed Systems*, **26**, 748-760. https://doi.org/10.1109/TPDS.2014.2313342

[8] NVIDIA (2022) Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[9] Aguilera, P., Lee, J., Farmahini-Farahani, A., *et al.* (2014) Process Variation-Aware Workload Partitioning Algorithms for GPUs Supporting Spatial-Multitasking. 2014 *Design*, *Automation & Test in Europe Conference & Exhibition* (*DATE*), Dresden, Germany, 24-28 March 2014, 726-731. https://doi.org/10.7873/DATE.2014.189

[10] Aguilera, P., Morrow, K. and Kim, N.S. (2014) Fair Share: Allocation of GPU Resources for Both Performance and Fairness. 2014 *IEEE* 32*nd International Conference on Computer Design* (*ICCD*), Seoul, South Korea, 19-22 October 2014. https://doi.org/10.1109/ICCD.2014.6974717

[11] Zhang W, Chen Q, Zheng N, *et al.* (2021) Towards QoS-Awareness and Improved Utilization of Spatial Multitasking GPUs. *IEEE Transactions on Computers*, **71**, 866-879. https://doi.org/10.1109/TC.2021.3064352