doi:10.11887/j.cn.202205004 http://journal.nudt.edu.cn

面向计算流体力学的图形处理器资源管理*

翁 跃,张献伟,张 曦,卢宇彤
(中山大学 计算机学院,广东 广州 510006)

摘 要:针对求解计算流体力学过程中图形处理器资源利用率低的问题,提出面向计算流体力学的图形处理器资源优化管理方案。基于计算流体力学的算法特性和同时运行任务的执行特点,设计合理的调度方案。通过动态改变不同任务的启动规模和启动时间,在减少资源竞争的同时提高图形处理器资源的有效使用。实验结果表明:本文提出的资源管理方案相比基线方法在不同任务规模下的平均加速比达到1.64,对图形处理器的硬件资源使用也有了显著的提升。

关键词:计算流体力学;图形处理器;资源管理;调度
中图分类号:TN95 **文献标志码**:A **开放科学(资源服务)标识码(OSID)**:
文章编号:1001-2486(2022)05-035-10

Graphics processing unit resource management for computational fluid dynamics

WENG Yue, *ZHANG Xianwei*, *ZHANG Xi*, *LU Yutong*
(School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China)

Abstract: Aiming at the problem of low resource utilization of GPU (graphics processing unit) in the process of solving CFD (computational fluid dynamics), a CFD-oriented GPU resource optimization management scheme was proposed. Based on the characterization of the CFD and tasks running concurrently, a reasonable scheduling scheme was designed. By dynamically changing the startup scale and time of different tasks, our method was able to reduce resource competition while improving the effective use of GPU resources. The experimental results show that compared with the baseline method, the average speedup ratio of our proposed resource management scheme reaches 1.64x speedup under different task scales, and the use of GPU hardware resources has also been significantly improved.

Keywords: computational fluid dynamics; graphics processing unit; resource management; scheduling

计算流体力学(computational fluid dynamics, CFD)是流体力学的一个分支,它利用数值分析和数据结构来分析和解决与流体流动有关的问题,可以广泛应用于各种工程实践,如空气动力学和航空航天分析,自然科学和环境工程,工业系统设计和分析等。借助计算机的计算能力,CFD可以模拟流体的自由流动以及流体与边界面的相互作用,实现对物理过程的仿真。如今已经有一些优秀的数值模拟软件用于求解CFD问题,例如拥有多种求解器可以解决可压缩、不可压缩流体模拟OpenFOAM[1]、应用在航空航天领域的FUN3D[2]、适合多物理场耦合的Fludity[3]、结构和非结构网格耦合的计算框架NNW-PHengLEI(风雷)[4]等。

随着任务规模的不断扩大,求解问题愈加复杂,传统的中央处理器(central processing unit, CPU)处理器已经难以提供足够的算力支持,这给CFD的应用和发展带来了严峻挑战。近些年来,随着通用图形处理器(general purpose graphics processing unit, GPGPU)的快速发展,其高并发、低功耗的特性受到高性能计算领域的青睐,被应用到CFD和其他诸多科学应用。如何充分利用图形处理器(graphics processing unit, GPU)的计算能力,提高CFD的计算性能和并行效率,是当前推动CFD发展的重要课题之一。

GPU的设计遵循单指令多线程(single instruction multiple threads, SIMT)的模式,即在同一时刻大量线程并行执行同一条指令。GPU高

* **收稿日期**:2022-01-13
基金项目:国家自然科学基金资助项目(62102465);国家重点研发计划资助项目(2016YFB0200902);国家数值风洞工程资助项目(NNW2019ZT6-B18);广东省引进创新创业团队资助项目(2016ZT06D211)
作者简介:翁跃(1997—),男,广东潮州人,硕士研究生,E-mail:wengy8@mail2.sysu.edu.cn;
张献伟(通信作者),男,副教授,博士,硕士生导师,E-mail:zhangxw79@mail.sysu.edu.cn

并行优势的发挥依赖于计算所需数据在合理时延范围内的快速返回。然而，非结构网格(unstructured grid)的流体问题求解面临严重的访存瓶颈。原因在于网格的不规则数据存储需要通过几何拓扑结构间接索引数据，这将导致频繁的不规则内存访问和缓存替换，显著增加数据的访问延迟，极大地限制了 GPU 的指令执行，造成计算流水资源的浪费[5]。

针对以上问题，一种可行的解决方案是同时运行多个任务实现 GPU 共享使用和资源的差异化使用，从而提高整体利用率。每个任务程序都会具有各自的特性，不同的任务组合将会带来不同的性能影响。当互补型的任务(如计算密集型和访存密集型任务)共同执行时，由于需求的主要硬件资源不同，任务间的竞争被弱化，可以在保证任务的执行效率的同时，使资源使用更加均衡。反之，相同特性的任务一起运行将会给资源竞争带来负面影响，很容易造成执行效率的不升反降[6-7]。因此，设计一个合理的混合任务调度机制对于 GPU 资源使用显得尤为重要。本文将基于 CFD 对 GPU 进行资源管理设计，通过搭配多任务共享 GPU 来提升整体的资源利用率。对于每个任务，将借助调优工具(profiling tool)进行性能分析，得到程序的执行特性，并结合收集的 GPU 资源使用情况来动态调度任务的执行时间和核函数大小。

1 研究背景

1.1 计算流体力学

计算流体力学是流体力学中一种重要的技术，通过对纳维－斯托克斯方程的求解，来预测流体流动时的状态和变化，用于科学问题的研究或者机械设计的工程实现。在过去的 30 多年里，CFD 被广泛地应用到多个领域，例如：在生物工程中，模拟心脏跳动、血液通过动脉和静脉流动、细胞流体等[8]；在机械工程中，模拟热量传递过程，涡轮机械仿真，叶轮机械设计，预测机翼的空气动力学等[9-11]；在食品制造行业中，模拟灭菌和制冷的过程等[12]。

图 1 展示了 CFD 在运输机模型外流场气动模拟中的应用。随着应用领域的拓展以及应用规模的增大，CFD 对计算资源的需求也在不断提升。因此，许多 CFD 模拟[13]必须在集群或者超级计算机上的高性能计算(high performance computing，HPC)系统中执行。随着计算技术的发展，GPU 由于其低能耗和高并发的特性，在

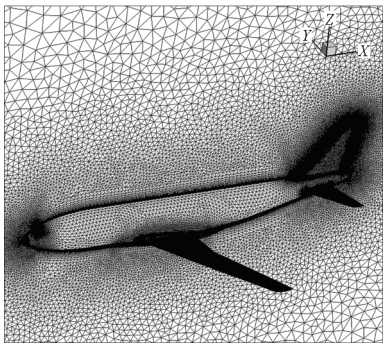

图 1 运输机模型外流场气动模拟计算网格

Fig. 1 Aerodynamic simulation grid of transport plane model

HPC 系统中发挥了越来越重要的作用，许多 CFD 的应用也开始在 GPU 上进行移植加速和优化设计[14]。

1.2 GPU 架构

图 2 展示了基于 NVIDIA 设计的通用图形处理器架构。一个 GPU 的组成包括多个流式处理器(streaming multiprocessors，SMs)、共享二级缓存(L2 cache)、全局内存(global memory)等，其中最核心的部分是 SMs。作为最基础的计算单元，每个 SM 上有多个计算核心，如 INT32、FP32、FP64 和 tensor 用于满足不同的计算精度需求。在内存上，每个 SM 包含 L0 和 L1 缓存，分别支持指令和数据的缓存。共享内存(shared memory)和 L1 共享同一个内存区域，用于减少合作线程之间的对全局内存的访问次数。其他厂商的 GPU，如 AMD 等，也采用相似的 GPU 设计，只不过用了不同的描述术语。编程人员利用并行编程框架，如 CUDA[15]、OpenCL[16]、HIP[17] 等编写并行程序。核函数(kernel function)包含大量的线程，并且可以被进一步划分为 blocks 和 grids，它们将通过设备的驱动和运行在 GPU 上执行。每个核函数的线程将被划分为多个 warp，warp 是 GPU 任务调度的基本单元。以 CUDA 为例，每个 warp 包含 32 个线程(threads)，这 32 个线程遵循 SIMT 的

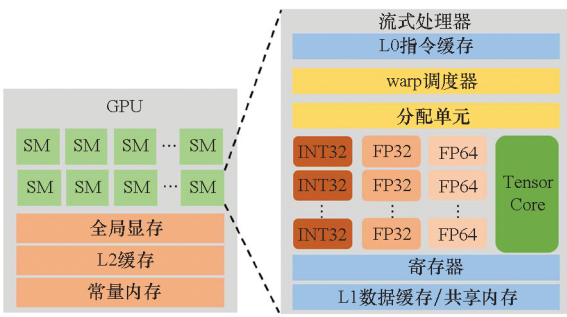

图 2 GPGPU 抽象架构

Fig. 2 Abstract architecture of GPGPU

执行模式,通过 SM 上的 warp 调度器被分发到相应的计算单元(CUDA core)上执行计算。

1.3 GPU 程序性能分析

为了更好地剖析 GPU 程序的性能瓶颈,研究人员设计了多种 GPU 性能分析工具,从实现上来讲可以分为:

基于硬件层面的实现,如 nvprof [18] 和 rocprof [19],基于硬件计数器来记录组件中发生的指定操作,如缓存请求和运行周期等。通过将原始计数器的结果转换为高级的度量(如转化为采样时间内的内核执行效率、内存利用率等),用户能够清晰地度量和理解低级的硬件活动,进而分析核函数的执行情况,调试和优化程序。在本文中,将采用 nvprof 来剖析程序进而获得执行特点。

在软件层面的实现上,二进制插桩框架(binary instrumentation frameworks)(如 GT-Pin[20] 和 NVBit[21])被广泛地应用。二进制插桩是指在二进制程序动态执行时注入插桩代码,从而实现程序分析。由于插入了额外的分析代码,这种方式有可能会改变原程序的运行逻辑,并且带来负面的影响[22]。除此之外,还可以通过改变编译行为来实现程序分析,例如 DARSIE[23] 和 CUDAAdvisor[24]。这些工具在编译期间添加更多辅助指令,以监视指令的有效性和内存访问。分析工具提供的信息有助于全面了解程序的特点,进一步了解程序运行过程中各种 GPU 硬件资源的使用情况,可用于指导混合调度的设计。

2 相关工作

2.1 非结构网格求解

近年来,由于非结构网格对于复杂外形的模拟具有更好的适应性,逐渐成为工程应用上的主流建模选择,也衍生出了不同精度的求解方式。基于二阶精度的有限体积方法 [25-27] 具有较好的鲁棒性和可靠性,可以应用于对精度要求较低的应用场景。但低阶的格式存在数值耗散与色散问题,难以解决如湍流、非线性作用等复杂场景的问题,对此需要采用高阶的格式[28]。基于非结构高阶的计算方法也有着迅速的发展,主要包括:kexact 有限体积方法[29]、间断 Galerkin 方法[30]、谱体积方法[31]、谱差分方法 [32] 以及混合多种求解方法的重构修正(correction procedure via reconstruction, CPR)方法[33]。本文中使用的是有限体积方法[34]。有限体积方法可以利用有限元素构建不规则网格,并将守恒态的微分方程离散化。同时,有限体积方法的计算速度和有限差分法接近,但相比有限元素法更加快速,可以极大减少计算量[35]。非结构网格有限体积方法在航空航天、海洋环境等许多科学计算领域得到广泛的应用,形成了一些优秀的数值模拟软件,例如 OpenFOAM[1]、FUN3D[2]、Fludity[3]、NNW-PHengLEI(风雷)[4]等。

2.2 GPU 资源管理

GPU 的计算核心是流多处理器(streaming multiprocessor, SM),围绕 SM 资源的分配问题,Adriaens 等[36] 介绍了三种管理方式——协作式多任务执行(cooperative multitasking)方式、抢占式多任务执行(preemptive multitasking)方式和空间多任务执行(spatial multitasking)方式。协作式执行方式允许任务在一段时间内独占 GPU,由于一个任务往往无法充分利用所有的 GPU 资源,所以存在闲置资源。对于抢占式执行方式,多个任务以时间片的形式轮流使用 GPU,这可以缓解任务的过长等待时间,但也会带来多次上下文切换的开销[37-38]。空间多任务执行方式允许多个任务同时分享 GPU 资源,每个任务都只会占用一部分 SM [39],但对于局部而言,单个 SM 的使用率仍存在提升空间[40]。并且,对于 SM 的控制和 warp 放置往往需要深入驱动层面的修改或者硬件的辅助支持,所以往往通过模拟器来实现方法和设计。而本文提出的方法可以直接应用在实际的 GPU 上面,不需要额外的硬件支持。也有学者把核函数的执行作为切入点,Pai 等[41] 提出了不同核函数静态融合的方式来提高 GPU 资源利用率,还有学者致力于寻找最优的核函数执行对[6-7],以使得两个核函数能以较低的性能损失来换取 GPU 资源利用率的提高。这种方法考虑了不同核函数之间的差异和共性,能够更全面地利用不同类型的资源。但是缺点在于需要进行大量的代码修改和核函数的调整,不利于实际的应用。本文的方法只需要很少的代码微调,更加方便可行。

NVIDIA 提供了多种技术以实现多个任务共同执行,包括流(stream)[42]、Hyper-Q [43]、多进程服务[44](multi-process service, MPS)和多实例 GPU[45](multi-instance GPU, MIG)。借助流技术,开发人员可以将核函数执行和数据复制放到不同的流序列中,实现异步执行。Fermi 架构之后,Hyper-Q 作为一种硬件支持,可以实现多个 CPU 进程或线程共享 GPU。MPS 进一步发挥 Hyper-Q

的性能,支持协同多进程应用,尤其是 MPI 类型的任务。在最新的 Ampere 架构中, MIG 允许 GPU 资源被划分为 7 个 GPU 实例,可以用于运行不同的应用,每个实例之间互不影响。这些方法可以实现 GPU 资源共享或划分,但是在提高资源利用率方面仍然存在一定的局限性。例如在 MIG 中,每个实例内部依旧需要关心资源利用率的问题。本文提出的方法和以上的技术是正交的,可以很好地与这些技术结合,进一步发挥 GPU 的性能。并且不需要额外的硬件支持或者驱动修改,只需要微调部分代码便可以支持本文的设计,并带来实际的性能提升。

3 研究动机和方法设计

3.1 研究动机

3.1.1 访存延迟导致指令执行低效

在移植和优化风雷软件(NNW-PHengLEI)[4]过程中发现移植后的 GPU 风雷程序在求解非结构网格问题时并无法充分利用硬件资源,其中的热点函数需要被更细致地分析。

表 1 列出了风雷程序中计算时间占比最大的前 8 个核函数名全称及其缩写,这 8 个核函数合计执行时间约占任务整体时间的 72%。图 3 展示了这些核函数每个时钟周期完成的指令条数(instructions per cycle,ipc)。*ipc* 表示每个采样周期指令的吞吐数目,可以用于衡量程序对于处理器性能的利用率,其理论峰值是 4.0 [18]。然而这些热点函数的平均 *ipc* 值只有 0.21,最大值也仅有 0.43,这表明风雷程序无法充分利用 GPU 的计算能力。

表 1 风雷 GPU 程序前 8 个热点核函数名全称及其缩写

Tab. 1 Full names and abbreviations of the top 8 hot kernel functions of the PHengLEI GPU program

缩写	核函数名全称
K0	GPUCompGradientGGNodeNewInteriorFaceCal
K1	GPUInteriorFaceNCountQNodeTNodeCal
K2	GPUCompVisfluxTEST
K3	GPUReConstructFaceValueLoop2
K4	GPUInteriorLimitCalculateAtomic
K5	GPULoadFluxInterior
K6	GPUGetQlQr
K7	GPUMinMaxCompareInteriorAtomic

图 4 展示了影响这些热点函数的原因。可以发现最重要的两个原因是 *stall_memory_dependency* 和 *stall_memory_throttle*。*stall_memory_dependency* 表示因为所需资源没有可用或没有充分利用,或者由于给定类型的请求太多而导致内存操作无法执行,从而导致的停顿百分比。而 *stall_memory_throttle* 是由于内存节流而发生指令延迟的百分比。这两项延迟原因是程序延迟的最主要原因,由此可以推断风雷程序的瓶颈在于访存等待。进一步深入分析发现,该访存问题是由非结构网格数据存放特点导致的。在非结构网格中,相邻数据结构(如点、面、体)之间的数据存放是非连续的,需要根据拓扑结构间接索引数据,这导致了缓存的失效,需要从全局内存中读取数据,这将带来极大的延迟,导致 GPU 计算资源的闲置和浪费。这种访存延迟型的任务在科学计算中并不少见[46-47]。对此,可以尝试让计算型任务和这类访存延迟型任务共享 GPU 资源,在访存型任务等待数据读取返回的同时,计算型任务可以使用计算单元,这将一定程度上提高 GPU 资源的整体利用率,并且保障任务的执行性能。

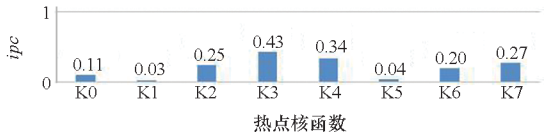

图 3 风雷 GPU 程序前 8 个热点核函数每个时钟周期完成的指令条数

Fig. 3 Instructions per cycle of the first 8 hot kernel functions of the PHengLEI GPU program

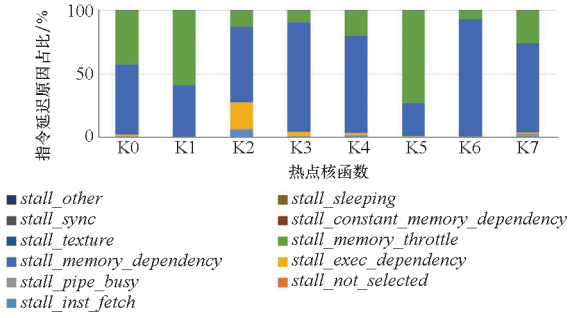

图 4 风雷 GPU 程序前 8 个热点核函数的指令延迟原因

Fig. 4 Stall reason of the first 8 hot kernel functions of the PHengLEI GPU program

3.1.2 拆分函数可以带来性能提升

在以往的经验中,研究人员会尝试将多个小的核函数合并为一个大的核函数,以提高函数的执行效率[48],但这种方法并不适用于所有情况。尝试对 K6 进行拆分,这个函数主要负责数据的索引和读取。将原先函数中的大循环拆分为 5 个小循环并串行执行,在拆分之后,总体的平均执

行时间减少了 5%,其他指标的分析如图 5 所示。其结果进行了正则化处理,将拆分前的各项指标和结果设为 1。从图 5 中可以看出,对 SM 的占用率 *achieved_occupancy* 也下降了 6%。这是因为核函数变得更小,能更快被执行完毕,所以采样期间的占用率下降。值得一提的是,*ipc* 和准备好可以被执行的 warp 数目 *eligble_warps_per_cycle* 出现了较大的提升,相比拆分前分别提高了 57% 和 85%。这是由于核函数规模变小,数据访问的瓶颈现象有所缓和,*stall_memory_dependency* 下降了 2%,*stall_memory_throttle* 也下降了 10%。一旦小批量数据读取完成便可以马上被执行,计算性能被进一步得到利用。同理,拆分后的核函数所需的数据依赖量更小,更容易成为可以被执行的 warp。这将可以缓解大核函数的访存瓶颈,使得 GPU 的资源被更好地利用。这启发研究人员可以在某种硬件资源竞争情况严重(如频繁访存)的时候,适当减小核函数的规模大小,让出部分资源(如对 SM 的占用),在减轻资源竞争的同时,维持甚至提升核函数的执行效率,也让其他任务有更多的资源可以被使用。因此,可以尝试在资源不足时,动态缩小核函数的规模,让出部分资源,可以被其他准备好的任务使用,从而提升 GPU 的整体资源利用率。

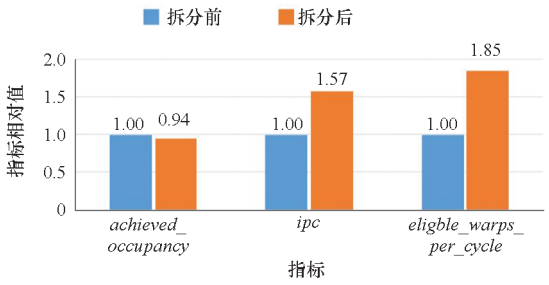

图 5　K6 拆分前后的性能指标对比

Fig. 5　Comparison of performance metrics before and after K6 separation

3.2　GPU 资源管理设计

3.2.1　应用性能分析

风雷程序具有庞大的任务结构,为了更好地进行方法设计,本文不失一般性地选择了来自 Rodinia 的 CFD Solver[47]。这个任务同样是面向非结构网格的求解,具有相同的任务属性且程序结构更加简短。在未来工作中将对风雷软件进一步解耦,拆分为更小的 kernel,适合多个 stream 并行。本文结合 CUDA Toolkit [15],一共选取了 8 个不同的任务。表 2 展示了这些任务的基本信息,详细的介绍将在 4.1 节中给出。

表 2　任务介绍

Tab. 2　Tasks introduction

任务全称	缩写	核函数数目	输入规模	任务类型
Computational Fluid Dynamics	CFD	4	fvcorr. domn. 097 K	访存密集型
HotSpot3D	HS3D	1	power_512 × 8	访存密集型
Vector Add	VA	1	204. 8 K	计算密集型
Matrix Multiplication	MM	1	(1. 28 K, 1. 28 K) × (1. 28 K, 2. 56 K)	计算密集型
HotSpot	HS	1	power_1024	计算密集型
Breadth-First Search	BFS	2	graph2M	延迟型
K-Means Clustering	KM	1	kdd_cup	延迟型
PathFinder	PF	1	10 K × 10 K	延迟型

基于 nvprof[18] 对这些程序进行分析,在图 6 中展示了这些任务单独运行时的 4 个重要的指标,包括衡量指令执行效率的 *ipc*,衡量 warp 在 SM 上的执行情况的 *achieved_occupancy* 和 *sm_efficiency*,以及衡量内存资源使用情况的 *dram_utilization*。不同指标的量纲将被统一映射到 0 ~ 10 之间。nvprof 以核函数为基本单位给出每个核函数的指标,为了对任务有一个整体的衡量,对于有多个核函数的任务,把每个核函数的执行时间占比作为权重,对同个任务内的不同核函数指标进行加权求和。从图 6 中可以发现,不同任务之间的性能存在较大的差异。根据不同任务的执行特征,将任务分为 3 类,分别为计算密集型、访存密集型和延迟型。CFD 和 HS3D 属于访存密集型,因为这两类任务的 *dram_utilization* 指标很高,而 *ipc* 较低。VA、MM 和 HS 属于计算密集型,这

类任务的特点是计算占用了程序的大部分时间，访存相对较少，且具有较高的 *ipc*。HS、BFS 和 KM 属于延迟型，这类任务对于计算和内存资源的需求都不突出，程序往往是在等待指令或者数据返回。对任务进行分类为后续的混合调度提供了指引。

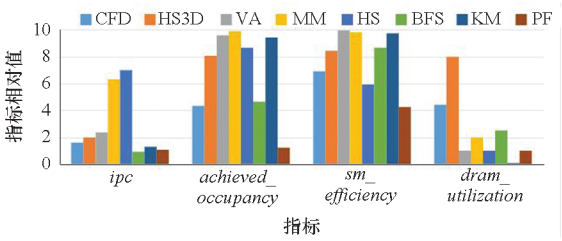

图 6 任务单独运行时的性能剖析结果

Fig. 6 Tasks profiling results when run exclusively

3.2.2 GPU 资源使用情况收集

NVIDIA 提供了一个基于 NVML [49] 的系统管理接口 nvidia-smi 用于设备的管理和监控 [50]，然而这个管理工具只能得到比较粗略的 GPU 使用率、内存利用率等，无法实现更加精细的监控。CPUTI[51] 可以实现运行过程中的程序分析，但是这会带来极大的开销。

对此进行了折中处理，利用任务单独运行时的 profiling 信息来近似估计任务启动时 GPU 资源的利用情况。例如 *dram_utilization* 被划分为 0 ~ 10 之间 10 个等级，那么可以将 GPU 总的 DRAM 可用量设定为 *gloabl_dram_utilization* = 10，当启动一个 *dram_utilization* = 3 的任务时，将更新 GPU 的资源剩余情况，得到 *global_dram_utilization* = 7。可以选择多种类型的指标来全方面地对 GPU 资源进行管理。使用了 13 个指标，包括 5 个衡量全局信息的指标（*achieved_occupancy*，*sm_efficiency*，*ipc*，*issue_slot_utilization*，*cf_fu_utilization*），2 个衡量计算单元使用情况的（*single_precision_fu_utilization*，*double_precision_fu_utilization*），以及 6 个和设备内存利用率相关的指标（*sysmem_read_utilization*，*sysmem_write_utilization*，*dram_utilization*，*ldst_fu_utilization*，*shared_utilization*，*tex_utilization*）。这些指标可以全面地描述一个任务的特性，并且对后续的调度带来一定的指导意义。对于全局资源的指标，由于单独运行和多个任务共同执行之间存在较大的差距，通过一个缩放因子 γ 进行适当的放缩以更好地指导调度。在每个任务启动之后，将消耗相应的 GPU 全局资源，在任务完成之后，这些资源将被归还。作为一种近似的收集方式，可以避免带来过多的统计开销，又可以从多个层面相对细粒度地分析当前 GPU 资源的使用情况，用于指导调度机制的设计。

3.2.3 混合任务调度

基于任务的性能分析信息和 GPU 资源使用情况收集，设计了一个混合任务的调度方式，其算法流程如图 7 所示。

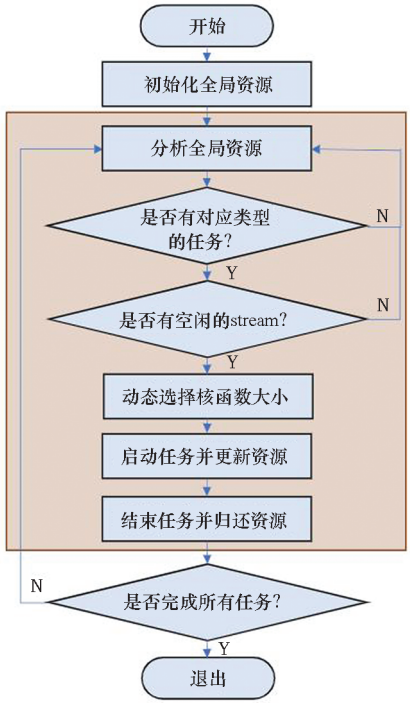

图 7 混合调度流程

Fig. 7 Flow chart of hybrid scheduling

首先会初始化全局资源，将 13 个指标所对应的全局资源全部设置为空闲。接着分析全局资源：将对计算相关的指标求均值得到 avg_{com}，同理计算与内存相关的指标 avg_{mem}。① 当 avg_{com} 和 avg_{mem} 之间的差小于一个阈值 α，那么此时可以启动任何类型的任务，优先选择延迟型任务。如果延迟型任务已经都执行完，将会依次选择计算密集型和访存密集型任务。②如果 $avg_{com} < avg_{mem}$，此时 GPU 的计算资源更紧张，将倾向于选择访存密集型任务。③如果 $avg_{com} > avg_{mem}$，此时访存紧张，会优先选择计算密集型任务。

接着会判断是否有对应类型的任务，如果没有，将会持续等待，直到检测到资源归还的信号量，那么会重新分析全局资源。

为了实现多任务共同执行，使用 CUDA 提供的流技术（stream）来实现多任务共享 GPU 资源。创建 N_{stream} 个 stream，每个 stream 每次会被一个启动的任务占有，直到任务执行结束。所以如果

有合适的任务可以执行,还需要判断是否有空闲的 stream;如果没有,将等待现有任务完成释放 stream。如果有空闲的 stream,那么可以根据当前资源剩余情况选择核函数的启动大小。这些核函数的大小是提前设定的,在任务输入数据固定的情况下,不同的核函数大小将会影响执行的时间和效率。有时候更小的核函数反而可以充分利用 SM 上的碎片资源,使得任务被更快地执行完毕。在输入不变的情况下,根据资源使用情况动态选择不同大小的核函数。接着启动任务,并消耗全局资源。在任务结束之后,归还全局资源,并判断是否还有任务未完成,重复循环迭代。

流程图中的灰色区域将会由 N_{thread} 个线程并行执行,涉及线程间的资源锁和信号量的更新。除了核函数执行部分将在 GPU 上执行,其他部分将在 CPU 上进行计算,这种设计可以充分发挥异构计算的性能,降低调度开销。

4　实验分析

4.1　实验设置

实验环境:如表 3 和表 4 所示,实验环境是基于 Ubuntu 18.04 LTS。所使用的硬件配置是一张 NVIDIA Volta 100。计算资源上共有 80 个流式处理器,每个流式处理器包括 64 个 FP32、32 个 FP64 和 8 个 Tensor 计算核心。内存资源上,显存的大小是 16 GB,并配备 6 144 KB 的 L2 级缓存,每个 SM 上包含 256 KB 的寄存器以及一块 128 KB 的 L1 和共享内存区域,其中共享内存最多可以扩展为 96 KB。使用的编程架构是 CUDA 10.1,GCC 的版本是 7.5.0。

表 3　实验硬件环境

Tab.3　Experimental hardware environment

硬件名称	配置
GPU 架构	Volta 100
显存容量	16 GB
流式处理器	80
FP32 计算单元 / 流式处理器	64
FP64 计算单元 / 流式处理器	32
Tensor 计算单元 / 流式处理器	8
L2 缓存	6 144 KB
寄存器 / 流式处理器	256 KB
L1 缓存,共享内存 / 流式处理器	128 KB

表 4　实验软件环境

Tab.4　Experimental software environment

软件名称	版本
Ubuntu	18.04 LTS
GPU driver	418.87
CUDA	10.1
GCC	7.5.0

设计实现:为了实现本文的设计,需要微调部分的任务代码。具体包括以下两个部分:①由于所有的任务都是使用默认的流来运行,所以需要更灵活地控制增加对流。将原先和内存分配相关应用程序编程接口(如 malloc(), free(), cudaMemcpy())以及核函数启动(kernel <<<gridsize, blocksize >>> ())的函数转化为支持特定流运行的 API(如 cudaMalloc(), cudaFree(), cudaMemcpyAsync(), kernel <<< gridsize, blocksize, streamId >>> ())。②将原先每个任务的入口(也即 main 函数)转换为一个可以从外部调用的入口,并且传入对应的流编号,例如 taskX.run(streamId)。这样可以更灵活地控制每个任务的启动时机以及让多个任务在不同的流中共享 GPU。

任务集选择:从 CUDA Toolkit[15] 和 Rodinia[47] 中选择了 8 个具有代表性的任务,如表 2所示。这些任务涵盖科学计算、机器学习、网络等方面,每个应用的输入规模都是固定的,采用了 Rodinia 所提供的输入数据集。对于无须读取外部输入的任务,如 Vector Add 和 Matrix Multiplication,采用随机初始化的方式对输入进行赋值。根据每个任务单独运行时的程序特性,将这些任务分为 3 类,分别是:访存密集型、计算密集型和延迟型。由于是面向 CFD 的 GPU 资源管理,所以将 CFD 作为最核心的任务,CFD 任务执行时间是整体运行时间的主要成分。

多任务共享 GPU:借助流技术和 CPU 线程来实现多任务共享 GPU,每个任务会独占一个 CPU 线程,并且独占一个流,在任务结束之后,流将被释放,被其他任务使用。一共有 N_{stream} 个流和 N_{threads} 个线程。

参数设置:对于每个任务,将重复执行 N_{r} 次。考虑到任务并行的数量过大将会增大资源竞争的可能性,所以将 N_{stream} 的数目设置为 3,即同时最多可以有 3 个任务一起共享 GPU 资源。将线程的数目 N_{threads} 设置为 6,以便当所有 stream 都被

不同任务占用时，CPU 端可以分析和选择下一个可以启动的任务，充分发挥异构的计算能力。缩放因子 $\gamma = 0.3$，阈值 $\alpha = 0.5$。缩放因子的作用在于平衡指标的范围以近似实际资源使用情况，如果不进行缩放，会导致任务之间的共同执行变得更为苛刻。缩放一些指标如 *achieved_occupancy* 和 *sm_efficiency*，这些指标在任务单独运行时具有很高的占用率，缩放这些指标能更好地让多任务共享 GPU，同时更好地近似多任务共享时这些资源的使用情况。α 被设置为 0.5，表示只有当计算资源 avg_{com} 或者访存资源 avg_{mem} 的消耗明显大于另一种资源，占据了超过一半的全局资源时，不宜再启动相同类型的任务。

基线方法：由于现有 GPU 资源管理工作大都基于模拟器进行设计和实现[37, 39-40]，无法进行公平地对比。不失一般性地，本文参考在现实中广泛使用的随机调度方式作为基线方法。使用同样的多线程和多流方式实现任务之间的并行。基线方法在选择启动任务时将采用随机选取的方式，不会考虑任务的特性和当前 GPU 的资源使用情况，模拟缺乏调度下的 GPU 共享方式。

效果指标：使用执行完所有任务的时间作为最重要的衡量指标，并将给出不同方法在运行过程中的多个指标分析。

4.2 实验结果及其分析

按照以上的实验设置，改变 N_r 的数目进行实验。当 $N_r = 2$ 时，每个任务会执行 2 次，由于一共有 8 个任务，所以任务规模是 16，同理当 N_r 是 3 和 4 时，任务规模分别是 24 和 32。这些任务可以看成是一个任务池，任务是同时到来的，但是正式被运行和处理取决于调度决策。对于同一种设定，重复执行 3 次，并取结果平均值。这 3 种设定的实验结果如图 8 所示，对比了完成批量任务所需时间的加速比。从实验结果可以发现，当任务数量分别为 16 和 24 时，本文提出的 GPU 资源管理方法可以达到 1.89 和 1.78 的加速比，可以很好地缓解资源竞争问题，提高 GPU 资源的利用率。当任务数量增大时，调度空间变得更加复杂，需要考虑的因素变多，此时加速效果有所下降，但相比基线任务，依旧有 1.24 的加速。总体来看，本文方法在不同任务规模下的平均加速比为可以达到 1.64。

图 9 给出了不同方法运行过程中的性能指标分析，将不同核函数的运行时间占比作为权重，对所有指标进行加权求和，得到最终的结果。由于

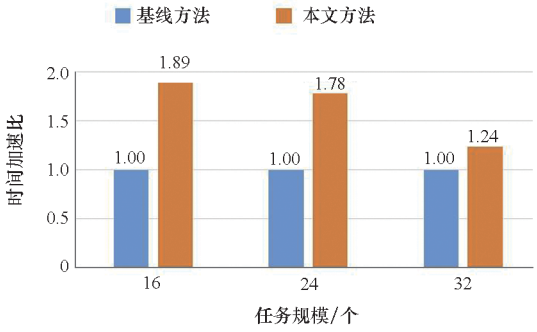

图 8 在不同任务规模下本文所提方法同基线方法在时间上的加速比

Fig. 8 Time acceleration ratio under different task scale

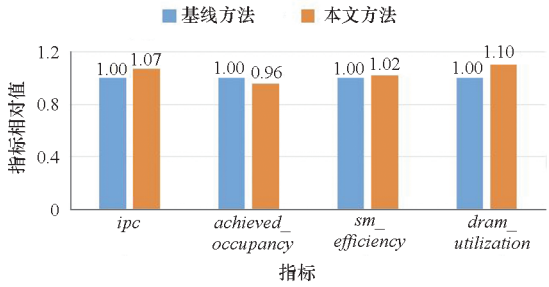

图 9 资源使用率对比

Fig. 9 Resource utilization comparison

不同的指标的量纲并不相同，所以将基线方法的值设为 1，用于正则化。不同于前面提到的 13 个用于描述任务单独使用 GPU 时的资源占用情况和指导调度的指标，在这里使用其中具有代表性的指标用于衡量多任务情况下 GPU 全局资源的使用情况。从图 9 中可以发现，本文提出的资源管理方法在 *ipc*，*sm_efficiency* 和 *dram_utilization* 指标上有了一定的提升。*ipc* 的提高表明本文方法可以更好地利用 GPU 的计算资源，*sm_efficiency* 的提升表明本文方法让更多的 warp 去充分利用 SM 的资源，至少一个 warp 使用 SM 的时间有所提高。*dram_utilization* 的提升证明本文方法可以提高内存资源的利用率，这将有利于访存型任务的执行。*achieved_occupancy* 有所下降，这是由于调度策略发现全局资源使用紧张时，会等待占用资源的释放后再启动新的任务。所以每个周期活动的 warp 数目有所下降，这也使得 GPU 资源竞争的情况有所缓解，间接带来计算效率的提升。

5 结论

GPU 的运用提高了 CFD 问题的求解速度，极大地推动了 CFD 的研究和发展，而在求解过程中如何对 GPU 资源有更高效的管理也是一个富有挑战性的问题。本文提出了一种面向 CFD 的

GPU 资源管理方法,通过分析搭配不同任务的程序特性,设计合理的 GPU 资源收集方法和调度策略,在提高硬件利用率的同时,加速了程序的执行效率。

在未来工作中,更加全面的 GPU 资源收集方法将会被考虑,包括对程序执行过程中的硬件需求有更细致的考虑。还将设计纠错机制,对近似的信息做进一步的更正。对于程序特性信息的收集,还会考虑更多的指标,包括延迟原因的加入。同时将增大任务数量和规模,设计更具弹性的核函数规模调整方法,可以根据实时的资源使用情况进行动态调整,提高方法的鲁棒性和可拓展性。

参考文献(References)

[1] WELLER H G, TABOR G, JASAK H, et al. A tensorial approach to computational continuum mechanics using object-oriented techniques [J]. Computers in Physics, 1998, 12(6): 620 – 631.

[2] BIEDRON R T, CARLSON J R, DERLAGA J M, et al. FUN3D Manual: 13.7[M]. Hampton: National Aeronautics and Space Administration, Langley Research Center, 2020.

[3] ZHENG J, ZHU J, WANG Z, et al. Towards a new multiscale air quality transport model using the fully unstructured anisotropic adaptive mesh technology of Fluidity (version 4.1.9) [J]. Geoscientific Model Development, 2015, 8(10): 3421 – 3440.

[4] 赵钟, 张来平, 何磊, 等. 适用于任意网格的大规模并行 CFD 计算框架 PHengLEI[J]. 计算机学报, 2019, 42(11): 2368 – 2383.
ZHAO Z, ZHANG L P, HE L, et al. PHengLEI: a large scale parallel CFD framework for arbitrary grids[J]. Chinese Journal of Computers, 2019, 42(11): 2368 – 2383. (in Chinese)

[5] WENG Y, ZHANG X, GUO X H, et al. Effects of mesh loop modes on performance of unstructured finite volume GPU simulations[J]. Advances in Aerodynamics, 2021, 3: 21.

[6] JIAO Q, LU M, HUYNH H P, et al. Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS [C]//Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2015.

[7] LIANG Y, HUYNH H P, RUPNOW K, et al. Efficient GPU spatial-temporal multitasking [J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 26(3): 748 – 760.

[8] RAMAN R K, DEWANG Y, RAGHUWANSHI J. A review on applications of computational fluid dynamics [J]. International Journal of LNCT, 2018, 2(6): 137 – 143.

[9] BORRELL R, DOSIMONT D, GARCIA-GASULLA M, et al. Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: application to airplane aerodynamics [J]. Future Generation Computer Systems, 2020, 107: 31 – 48.

[10] MARTINS J R R A. Perspectives on aerodynamic design optimization [C]//Proceedings of AIAA SciTech Forum, 2020.

[11] SYNYLO K, KRUPKO A, ZAPOROZHETS O, et al. CFD simulation of exhaust gases jet from aircraft engine [J]. Energy, 2020, 213: 118610.

[12] KAUSHAL P, SHARMA H K. Concept of computational fluid dynamics (CFD) and its applications in food processing equipment design [J]. Journal of Food Processing and Technology, 2012, 3(1): 1000138.

[13] LIU X C, ZHONG Z M, XU K. A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms[J]. Future Generation Computer Systems, 2016, 56: 759 – 765.

[14] SLOTNICK J, KHODADOUST A, ALONSO J, et al. CFD vision 2030 study: a path to revolutionary computational aerosciences[R]. Hampton: National Aeronautics and Space Administration, Langley Research Center, 2014.

[15] NVIDIA. CUDA[CP/OL]. [2021 – 11 – 20]. https://developer.nvidia.com/cudatoolkit Accessed 6 21.

[16] KHRONOS. Opencl[CP/OL]. [2021 – 11 – 20]. https://www.khronos.org/opencl/ Accessed 6 21.

[17] AMD. Hip [CP/OL]. [2021 – 11 – 21]. https://rocmdocs.amd.com/en/latest/Current_Release_Notes/ROCm-Learning-Center.html Accessed 6 21.

[18] NVIDIA. Nvprof[CP/OL]. [2021 – 11 – 21]. https://docs.nvidia.com/cuda/profiler-users-guide/index.html Accessed 6 21.

[19] AMD. Rocprofiler[CP/OL]. [2021 – 11 – 25]. https://github.com/ROCmDeveloper-Tools/rocprofiler Accessed 6 21.

[20] KAMBADUR M, HONG S, CABRAL J, et al. Fast computational GPU design with GT-pin[C]//Proceedings of IEEE International Symposium on Workload Characterization, 2015: 76 – 86.

[21] VILLA O, STEPHENSON M, NELLANS D, et al. NVBit: a dynamic binary instrumentation framework for NVIDIA GPUs[C]// Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019: 372 – 383.

[22] ZHANG X W, SHCHERBAKOV E. DELTA: validate GPU memory profiling with microbenchmarks[C]//Proceedings of the International Symposium on Memory Systems, 2020: 97 – 104.

[23] YEH T T, GREEN R N, ROGERS T G. Dimensionality-aware redundant SIMT instruction elimination [C]// Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020: 1327 – 1340.

[24] SHEN D, SONG S L, LI A, et al. CUDAAdvisor: LLVM-based runtime profiling for modern GPUs[C]//Proceedings of the International Symposium on Code Generation and Optimization, 2018: 214 – 227.

[25] GASKELL P H, LAU A K C. Curvature-compensated convective transport: SMART, a new boundedness-preserving transport algorithm[J]. International Journal for Numerical Methods in Fluids, 1988, 8(6): 617 – 641.

[26] VAN LEER B. Towards the ultimate conservative difference scheme Ⅲ. Upstream-centered finite-difference schemes for ideal compressible flow [J]. Journal of Computational Physics, 1977, 23(3): 263 – 275.

[27] VAN ALBADA G D, VAN LEER B, ROBERTS W W, JR. A comparative study of computational methods in cosmic gas dynamics[M]//Proceedings of Upwind and High-resolution Schemes. Berlin: Springer, 1997: 95 – 103.

[28] 罗磊, 高振勋, 蒋崇文. CFD 技术发展及其在航空领域中的应用进展[J]. 航空制造技术, 2016, 59(20): 77 – 81.
LUO L, GAO Z X, JIANG C W. CFD development and application in aviation [J]. Aeronautical Manufacturing Technology, 2016, 59(20): 77 – 81. (in Chinese)

[29] BARTH T J, FREDERICKSON P O. Higher order solution of the Euler equations on unstructured grids using quadratic reconstruction [C]//Proceedings of the 28th Aerospace Sciences Meeting, 1990.

[30] COCKBURN B, SHU C W. The Rung-Kutta discontinuous Galerkin method for conservation laws V: multidimensional systems [J]. Journal of Computational Physics, 1998, 141(2): 199 – 224.

[31] SUN Y Z, WANG Z J, LIU Y. Spectral (finite) volume method for conservation laws on unstructured grids VI: extension to viscous flow [J]. Journal of Computational Physics, 2006, 215(1): 41 – 58.

[32] SUN Y Z, WANG Z J, LIU Y. High-order multidomain spectral difference method for the Navier-Stokes equations[C]// Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit, 2006.

[33] WANG Z J, GAO H, HAGA T. A unifying discontinuous formulation for hybrid meshes[M]//Proceedings of Adaptive High-order Methods in Computational Fluid dynamics. [S.l.]: World Scientific, 2011: 423 – 453.

[34] MOUKALLED F, MANGANI L, DARWISH M. The finite volume method in computational fluid dynamics[M]. Berlin: Springer, 2016.

[35] 彭思显. 应用二维浅水流方程之有限体积数值模式模拟溃坝水流[J]. 坡地防灾学报, 2011, 10(2): 25 – 36.
PENG S X. Dam-break flow simulation using 2D finite volume numerical model of shallow water equations[J]. Journal of Slopeland Hazard Prevention, 2011, 10(2): 25 – 36. (in Chinese)

[36] ADRIAENS J T, COMPTON K, KIM N S, et al. The case for GPGPU spatial multitasking [C]//Proceedings of IEEE International Symposium on High-Performance Computer Architecture, 2012.

[37] PARK J J K, PARK Y, MAHLKE S. Chimera: collaborative preemption for multitasking on a shared GPU [J]. ACM SIGARCH Computer Architecture News, 2015, 43 (1): 593 – 606.

[38] TANASIC I, GELADO I, CABEZAS J, et al. Enabling preemptive multiprogramming on GPUs[J]. ACM SIGARCH Computer Architecture News, 2014, 42(3): 193 – 204.

[39] WANG Z N, YANG J, MELHEM R, et al. Simultaneous multikernel GPU: multi-tasking throughput processors via fine-grained sharing [C]//Proceedings of International Symposium on High Performance Computer Architecture (HPCA), 2016.

[40] LEE M, SONG S, MOON J, et al. Improving GPGPU resource utilization through alternative thread block scheduling[C]//Proceedings of International Symposium on High Performance Computer Architecture (HPCA), 2014.

[41] PAI S, THAZHUTHAVEETIL M J, GOVINDARAJAN R. Improving GPGPU concurrency with elastic kernels[J]. ACM SIGARCH Computer Architecture News, 2013, 41 (1): 407 – 418.

[42] NVIDIA. Stream[CP/OL]. [2021 – 12 – 04]. https://docs.nvidia.com/cuda/ cuda-runtime-api/group_CUDART_STREAM.html Accessed 6 21.

[43] NVIDIA. Hyper-q[CP/OL]. [2021 – 12 – 04]. https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf Accessed 6 21.

[44] NVIDIA. MPS[CP/OL]. [2021 – 12 – 04]. https://docs.nvidia.com/deploy/mps/ index.html Accessed 6 21.

[45] NVIDIA. MIG[CP/OL]. [2021 – 12 – 06]. https://docs.nvidia.com/datacenter/ tesla/mig-user-guide/index.html Accessed 6 21.

[46] STRATTON J A, RODRIGUES C I, SUNG I-J, et al. Parboil: a revised benchmark suite for scientific and commercial throughput computing [R]. Champaign: University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing, 2012.

[47] CHE S, BOYER M, MENG J Y, et al. Rodinia: a benchmark suite for heterogeneous computing [C]// Proceedings of IEEE International Symposium on Workload Characterization (IISWC), 2009.

[48] WEN Y, O′BOYLE M F P. Merge or separate? Multi-job scheduling for OpenCL kernels on CPU/GPU platforms[C]// Proceedings of the Workshop About General Purpose Processing Using GPUs, 2017.

[49] NVIDIA. NVML[CP/OL]. [2021 – 12 – 06]. https://developer.nvidia.com/nvidia-management-library-nvml Accessed 6 21.

[50] NVIDIA. Nvidia-smi[CP/OL]. [2021 – 12 – 06]. https://developer.nvidia.com/nvidia-system-management-interface Accessed 6 21.

[51] NVIDIA. Cputi[CP/OL]. [2021 – 12 – 06]. https://docs.nvidia.com/cuda/ cupti/index.html Accessed 6 21.