基于线程池的 GPU 任务并行计算模式研究

李 涛[1),2)] 董前琨[1)] 张 帅[1)] 孔令晏[1)] 康 宏[1)] 杨愚鲁[1)]

[1)](南开大学计算机与控制工程学院 天津 300071)

[2)](中国科学院计算技术研究所计算机体系结构国家重点实验室 北京 100109)

摘 要 GPU 已经成为具有高并发高内存带宽的通用协处理器,但是 GPU 与 CPU 在体系结构和编程模型上存在很大差异,导致 CPU-GPU 异构计算系统的编程复杂度提高,即使采用统一计算设备架构(CUDA)提供的 kernel 并发技术和多流技术也较难充分控制和利用 GPU 上的计算资源,难以有效地处理不规则的并行应用问题.为从体系结构角度探索 GPU 硬件支持的页锁定内存和统一虚拟地址空间等特征,该文提出了 CPU 辅助任务调度管理下的基于线程池技术的 GPU 任务并行计算模型 CAGTP,实现了 CPU-GPU 异构计算系统上的共享内存式程序设计.提出并设计了 CPU 端的任务队列、计算线程块级任务调度器、任务槽和 GPU 端的任务复用 kernel 函数等机制,实现了 CPU 与 GPU 间的高效细粒度任务交互,避免了原生 CUDA 程序中多次启停 kernel 函数的开销,有效地支持了 GPU 上的细粒度不规则并行任务计算,而且利用模型 API 接口函数能够降低 CPU-GPU 异构计算系统的编程难度.实验结果表明,CAGTP 模型中任务调度的开销是 kernel 函数调用的 5%,有效提升了通用矩阵乘、乔列斯基分解和 K 均值、T 近邻等典型线性代数和机器学习算法的计算性能;CAGTP 模型易于扩展使用多块 GPU,且在性能差异较大的多个 GPU 之间达到负载均衡,能够高效求解混合任务和具有不规则并行性的应用问题.

关键词 异构计算系统;统一计算设备架构;线程池;任务并行;任务复用函数

中图法分类号 TP393 **DOI 号** 10.11897/SP.J.1016.2018.02175

GPU Task Parallel Computing Paradigm Based on Thread Pool Model

LI Tao[1),2)] DONG Qian-Kun[1)] ZHANG Shuai[1)] KONG Ling-Yan[1)] KANG Hong[1)] YANG Yu-Lu[1)]

[1)](*College of Computer and Control Engineering, Nankai University, Tianjin* 300071)

[2)](*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100109)

Abstract GPUs have become general-purpose co-processors with high concurrency and memory bandwidth. However, due to the huge difference of architecture between GPU and CPU, programming on CPU-GPU heterogeneous systems has been proved difficult and time-consuming. CUDA (Compute Unified Device Architecture) is a general purpose parallel computing platform and programming model introduced by NVIDIA. It enables thousands of threads on NVIDIA's GPU to be developed for high performance computing, and provides convenience to leverage the parallel computing ability of GPUs to some extent. Nevertheless, it is still a problem to fully utilize GPU computational resources and reasonably schedule computational tasks running on GPUs, which has become the main bottleneck to take advantage of the parallel computing ability of GPUs and apply them to accelerate practical applications, such as matrix operations from linear algebra area and machine learning algorithms. This paper proposes CAGTP (CPU-assisted GPU thread

收稿日期:2017-02-06;在线出版日期:2017-12-28.本课题得到国家自然科学基金(61872200)、天津市自然科学基金(16JCYBJC15200,17JCQNJC00300)、计算机体系结构国家重点实验室开放课题(CARCH201504)、天津市大数据与云计算科技重大专项(15ZXDSGX00020)、高等学校博士学科点专项科研基金(20130031120029)资助.**李 涛**,男,1977 年生,博士,副教授,中国计算机学会(CCF)高级会员,主要研究方向为并行与分布式处理、机器学习. E-mail: litao@nankai.edu.cn. **董前琨**,男,1990 年生,硕士,中国计算机学会(CCF)学生会员,主要研究方向为并行与分布式处理、GPGPU 计算. **张 帅**,男,1986 年生,博士,主要研究方向为并行分布式计算、GPGPU 计算. **孔令晏**,男,1992 年生,硕士,主要研究方向为并行与分布式处理、机器学习. **康 宏**(通信作者),男,1973 年生,博士,讲师,主要研究方向为大数据处理、数据库、机器学习. E-mail: kanghong@nankai.edu.cn. **杨愚鲁**,男,1961 年生,博士,教授,主要研究领域为互联网络、并行机体系结构、分布式计算.

pool), a thread pool based GPU task parallel model with the assistance of CPU in task scheduling. First, we take advantage of page-locked memory and unified virtual address space, which have been supported in new generation of GPU architectures and new versions of CUDA, to improve the communication efficiency between CPU and GPU in CAGTP. Then, we design the I/O task queues, block-level task scheduler and task slots on CPUs in CAGTP, allowing users to dynamically schedule tasks that are to be calculated on GPUs. Besides, the task-multiplexed kernel is designed on GPU, which is the core of CAGTP and can achieve the dynamic scheduling of thread blocks on GPUs. Based on these mechanism, CAGTP allows efficient scheduling of fine-grained tasks, and effectively avoids the cost of launching kernels multiple times in native CUDA programs. Moreover, CAGTP supports the calculation of irregular fine-grained parallel tasks on GPUs. Last of all, we provide several Application Programming Interfaces on CAGTP model, which can effectively reduce the complexity and time-consumed of programming on CPU-GPU heterogeneous systems. To test both the performance and extendibility of CAGTP by kinds of applications with different task sizes, we pick out GEMM (General Matrix-Matrix Multiplications) and Cholesky decomposition from liner algebra area, TNN (T Nearest Neighbor) algorithm and *K*-means algorithm from machine learning area, and the mixed application including SPMV (Sparse Matrix Vector Multiplication) and Black Scholes algorithm in the experiments. The results show that CAGTP solutions can dramatically improve the efficiency of task interaction in CPU-GPU heterogeneous systems, with its time spent for task scheduling only accounts for 5% of native CUDA kernel functions' launching time. At the same time, CAGTP can achieve obviously higher performance than traditional CUDA paradigms, especially for irregular task-parallel applications such as Cholesky decomposition and other mixed applications. Last but not least, CAGTP also shows good extendibility when applied to multiple GPUs, it can achieve load balancing even for distinct NVIDIA's GPUs.

Keywords heterogeneous computing system; CUDA; thread pool; task parallelism; task-multiplexed kernel

1 引 言

GPU 已成为具有高并行性和高存储带宽的典型众核处理器,利用 GPU 进行高性能计算能够实现远超多核 CPU 的计算性能,它的研究及应用已成为高性能计算领域的重要分支[1-2]. 将 GPU 应用于科学计算产生了通用 GPU 计算(General Purpose GPU computing,GPGPU)的概念,并在计算密集型和数据密集型等应用上获得了明显性能加速[3]. 统一计算设备架构 CUDA(Compute Unified Device Architecture)能够让程序员使用熟悉的 C/C++语言在 NVIDIA GPU 上进行并行程序设计. 在以规则应用如通用矩阵乘为代表的线性代数领域,使用 CUDA 可以达到高度优化的 CPU 程序性能的数倍[4]. 表 1 列举了 NVIDIA 每一代架构中主流 GPU 的特征及其计算性能.

表 1 典型 GPU 技术规格

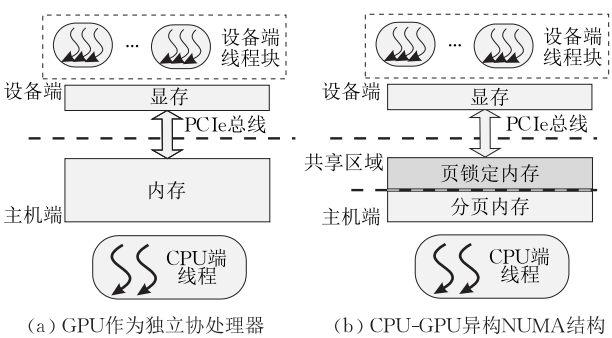
GPU	核心数目	架构	单精度浮点性能/Tflops	双精度浮点性能/Tflops
GTX 260	192	Tesla	0.71	*
GTX 480	480	Fermi	2.02	0.67
GTX 780	2304	Kepler	3.49	0.18
Tesla K40	2880	Kepler	4.29	1.43
GTX Titan X	3072	Maxwell	6.10	0.20
Tesla P100	3584	Pascal	10.60	5.30

尽管目前已有很多研究利用 GPU 的高并行性对具体应用实现了性能加速,但更为成功的应用仍然局限于并行性质较好的规则应用上[5-6]. 具有不规则并行性或需要细粒度 CPU-GPU 交互的应用较难在 GPU 上取得明显的性能提升,而且很难充分利用 GPU 的全部计算资源. 例如,Parboil2 基准测试的结果表明,在 Fermi 架构 GPU 上平均仅利用到硬件资源的 20%~70%[7]. 其主要原因在于原生 CUDA 编程模型更加侧重于可用性及其在不同 GPU

上的扩展性，并未挖掘不同 GPU 的结构特征及其计算效能[7-8]. 如何提高 GPU 硬件资源的利用率进而提升 GPU 通用计算的性能非常关键.

CUDA 编程模型提供了 GPU 上的数据并行抽象机制，但是并不支持有效的任务并行. 尽管 GPU 和 CPU 在体系结构方面存在诸多差异，然而数据并行与任务并行相结合还是 CPU-GPU 异构系统中更好的并行计算模式[9]. 目前已有工作在 CUDA 的基础上设计 CPU-GPU 任务并行，一种是采用并发内核执行 CKE(Concurrent Kernel Execution)技术[10-13]，通过为每一个任务启动一个 CUDA kernel，然后并发执行这些独立的 kernel 来达到任务并行的目的；另一种是持久化 kernel 技术，通过将分解出来的计算任务动态地调度到 GPU 线程块上实现[14-17]，虽然这种方式能够消除 CKE 中启停 kernel 函数的开销[14-15]，但是 CPU 和 GPU 之间的任务交互仍然是明显的性能瓶颈. 在这种有限的 CPU-GPU 交互以及 GPU 控制情况下，利用 CPU-GPU 异构系统很难实现高效的细粒度任务交互及数据传输，特别是针对不规则并行计算任务，很难发挥 CPU-GPU 异构系统的计算性能.

Fermi 架构的 GPU 开始支持页锁定内存和统一虚拟地址空间 UVA(Unified Virtual Address Space)[4]. 页锁定内存是进程虚拟地址空间中的一段，包含这一段的内存页面常驻物理内存，不会被操作系统交换到虚拟内存中. 在支持 UVA 的设备上，页锁定内存可以被映射到设备端的地址空间中，并且能够被设备端的 GPU 线程直接访问. 基于此，由 GPU 作为独立的协处理器，CPU-GPU 异构系统变成了主机端与设备端构成的非均一内存访问 NUMA(Non Uniform Memory Access Architecture)结构，如图 1 所示. 此时，CPU 与多个 GPU 可看作共享存储上紧耦合的非对称多处理器系统，为实现细粒度任务交互与同步提供了结构基础.



(a) GPU作为独立协处理器　　(b) CPU-GPU异构NUMA结构

图 1　CPU-GPU 异构体系结构

在多核和多 CPU 计算系统中，利用线程池等编程技术能够实现高效的任务并行，通过共享的内存访问区域可进行高效的同步与线程间交互，从而支持更细粒度的并发和不规则的并行模式[18]. 但在 CPU-GPU 异构系统中，由于 CPU 和 GPU 的体系结构和编程模式的差异，并没有充分挖掘其共享内存特性，导致 GPU 资源利用率低、任务交互开销大等问题. 基于 CUDA 编程模型在 CPU-GPU 异构系统上进行共享内存任务并行设计，仍然存在诸多困难和挑战. 首先，CUDA 提供的原始 CPU-GPU 交互功能有限，很难有效减小 CPU 和 GPU 间的任务交互代价；其次，在进行任务调度时需要使用线程安全的任务队列，很难有效减小 GPU 端众多线程进行任务队列操作的开销；再有，CPU 将任务调度到 GPU 上以后，无法实时跟踪 GPU 上的任务状态，很难有效处理 CPU 和 GPU 不同任务之间的依赖关系[19].

因此，本文基于页锁定内存和 UVA 等 GPU 结构特征，将 CPU-GPU 异构系统视为 NUMA 结构，提出了 CPU 辅助任务调度管理下的基于线程池技术的 GPU 任务并行计算模型 CAGTP(CPU-Assisted GPU Thread Pool)，并设计了适于 CPU-GPU 异构系统的有效编程接口. 通过在 CPU 和 GPU 共享的内存区域内开辟任务槽和任务复用 kernel 等机制，实现了计算线程块级动态任务调度策略，使得 CPU 和 GPU 间的任务交互开销显著低于持久化 kernel 和 CKE 等技术. 应用线性代数和机器学习等典型算法的实验表明，基于 CAGTP 模型能够有效地提升 CPU-GPU 异构系统的计算性能和 GPU 资源利用率. 本文主要贡献如下：

(1) 将 CPU-GPU 异构系统视为 NUMA 结构，提出 GPU 任务并行计算模型 CAGTP，通过任务复用 kernel 等技术优化 kernel 函数重启动开销以及提高 GPU 资源利用率.

(2) 设计线程块级的 GPU 任务动态调度机制，利用任务槽技术使 CPU 端的调度线程能够及时追踪 GPU 上任务的执行状态，提高了 CPU-GPU 间的任务交互能力.

(3) 设计基于共享内存的 GPU 任务并行编程接口，实现对于细粒度不规则任务的高效并行计算.

(4) 易于将 CAGTP 模型扩展到多 GPU 上，并且能够在性能差异较大的多个 GPU 上实现负载均衡.

本文第 2 节简要介绍 CPU-GPU 异构体系、CUDA 编程模型以及线程池技术；第 3 节详细描述 CAGTP 模型，着重讨论基于该模型的异构并行编程模式；第 4 节通过对线性代数和机器学习等典型

算法的求解验证 CAGTP 模型的性能优势;第 5 节简要总结已有 GPU 任务并行工作.

2 相关技术

本节简要介绍 GPU 体系结构、CUDA 编程模型以及线程池技术,这是将 CPU-GPU 异构系统视为 NUMA 结构并提出 CAGTP 模型的基础.

2.1 CUDA 编程模型

CUDA 使用的是结构化线程模型,GPU 线程是通过启动 kernel 函数创建的,一个 kernel 函数包含多个线程块,图 2 展示了 CUDA 的结构化线程模型.CUDA 运行时会映射并以轮询方式将这些不同尺寸的线程块调度到流多处理器(SM)上,而且多个线程块能够被映射到同一 SM 上并发执行.但在运行过程中,一个线程块不能被切分或者迁移到多个 SM 上[4].通常,一个程序会包含多个线程块,在 kernel 函数启动时仅会激活部分线程块,其余的线程块则在运行时通过硬件调度来激活.

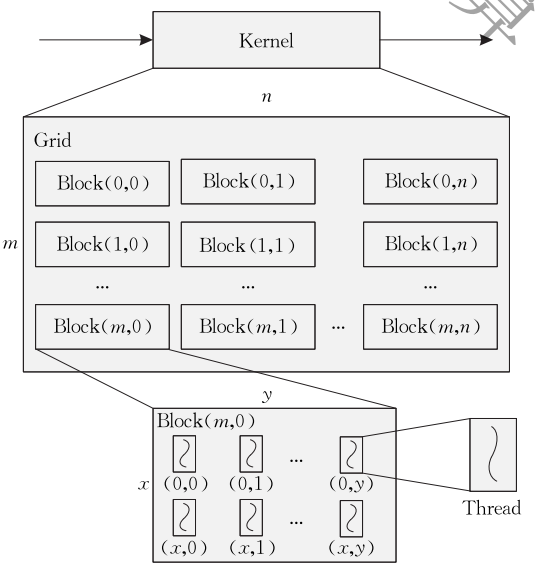


图 2 CUDA 结构化线程模型

在 CUDA 编程模型中,CPU 和 GPU 通过 PCIe 总线进行连接,GPU 通常作为独立的协处理器使用,此时 CPU 与 GPU 是松耦合连接,CPU 和 GPU 之间的协作类似于消息传递机制.在程序执行前通过特定接口将数据拷贝到 GPU 端的设备内存中;然后通过调用 kernel 函数创建大量 GPU 线程进行计算,在计算过程中与主机端程序几乎没有交互;在计算完成后,主机端程序再通过特定接口将结果数据拷贝到主机端内存.因此,CPU 和 GPU 之间的数据传输和交互开销较大,对细粒度不规则应用较难取得很好的并行加速,在一定程度上限制了 GPU 的应用范围[20-21].

2.2 NUMA 结构

从体系结构上看,主机端和 GPU 端形成了一个 NORA(No Remote Access)结构,在这种结构上的 CUDA 编程模型实际上是消息传递式的,如图 1(a)所示.主机端与 GPU 端具有各自独立的地址空间,相当于消息传递模型中的独立节点,两者通过消息传递完成数据交互和同步.

GPU 对页锁定内存和 UVA 等特征的支持使得主机端与 GPU 端不再是严格的 NORA 结构.主机端线程通过前端总线和北桥直接访问页锁定内存,访存带宽为 10 GB/s~20 GB/s,而 GPU 端线程访问页锁定内存需要通过 PCIe 总线,在实验用机器上访存带宽约为 6.3 GB/s.因此,主机端和 GPU 端实际上形成带有共享内存访问区域的异构 NUMA 体系结构,如图 1(b)所示.

虽然主机端和 GPU 端的线程在访问页锁定内存的带宽上有所差异,但是在这种体系结构上,原有的按需创建 GPU 线程、消息传递式的 CUDA 编程模型已经不再是最优选择.基于异构 NUMA 体系结构的共享内存特性设计新的共享内存编程模式不仅可行,而且对扩展以 GPU 为代表的协加速器的应用范围,提升 CPU-GPU 异构系统的计算性能非常重要.

2.3 线程池技术

线程池技术是一种共享内存平台上常用的并行程序设计技术,可以避免反复创建和销毁线程的开销,从而支持细粒度的任务并发.线程池主要包括四个部分:线程管理器、工作线程、任务接口和输入输出任务队列,如图 3 所示.通过由前期特定的任务在处理器上创建线程,当新任务到达时,处理器便利用已经创建的线程执行任务,当处理完所有任务后,由

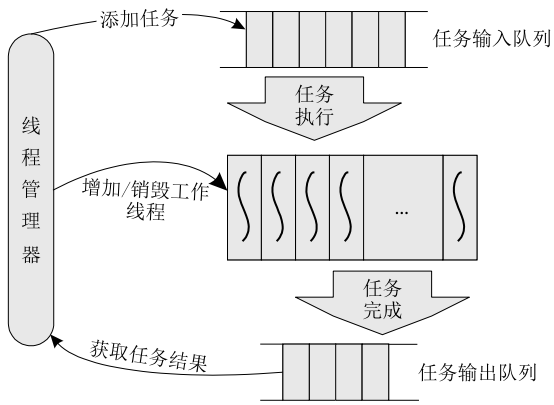


图 3 线程池构成

另一个特定任务销毁线程.这种静态创建线程的机制避免了 Fork-Join 模型中重复创建和销毁线程的开销,计算任务一般是通过线程安全的数据结构分发给多个工作线程并保证负载均衡,从而有效利用硬件资源提高处理效率[22].

典型的线程池技术包括线程构建模块 TBB[23]和 Cilk[24].TBB 是以最大限度利用多核处理器性能为目标的多功能并行编程框架,涵盖了可扩展内存分配、线程安全的并发容器和原子操作抽象等工具.在执行时使用了基于工作窃取的并行结构,针对一些复杂任务如图形渲染、视频压缩等,很容易构建任务流水线来获得优异性能.针对细粒度任务,英特尔为 C++编译器提供了 Cilk 语言扩展技术,以便使用嵌套并行进一步提高程序并行性.它对每一个线程使用一个双端队列来跟踪任务的执行,同时在任务执行时将双端队列作为栈使用,可以消除一部分异步操作.

在 CPU 上创建和结束线程的开销是创建或销毁任务的 18～100 倍,而且通过任务进行同步的开销也远低于同步多个线程的开销,因此线程池技术能够更好地支持细粒度的任务并发[22].同时,由于不同任务可以包含不同类型、不同量级的计算操作,因此应用线程池技术能够更好地支持复杂不规则的并行计算模式.

基于上述讨论,将 CPU-GPU 异构系统视为共享内存 NUMA 体系结构,基于线程池技术设计共享内存结构上的并行计算模式,在 GPU 端创建静态的工作线程,利用页锁定内存进行任务交互和同步,有助于打破 CUDA 编程模型的消息传递模式限制,优化 CPU 和 GPU 之间的任务交互与同步开销,实现对不规则、细粒度任务的并行计算及保证其负载均衡,进而提升 CPU-GPU 异构系统的计算性能并拓展其应用范围.

3　CAGTP 模型

本节详细描述 CAGTP 模型,包括模型的组织结构、实现方式以及编程接口等,最后简要介绍该模型的特点.

3.1　模型结构

CAGTP 模型的设计理念是基于线程池技术在 GPU 端创建静态的 GPU 常驻线程,借助分配在页锁定内存中的共享内存区域,由 CPU 将计算任务调度到 GPU 端的工作线程.CPU 和 GPU 之间通过共享内存实现任务交互与线程同步,以有效支持不规则细粒度任务的并发执行.由于 kernel 函数的启动开销为 4 μs～7 μs,本文约定计算时间在微秒数量级的计算任务为细粒度任务.CAGTP 模型结构如图 4 所示,包括一组运行在 GPU 端的任务复用 kernel 函数、一组分配在页锁定内存中共享区域内的不会被替换出去的任务槽、一个 CPU 端的调度线程以及一组由调度线程管理的任务输入输出队列.

不同于 CPU 面向延迟优化的设计理念,GPU 主要是面向计算吞吐量优化,并不擅长执行判断和分支等指令.因此在 CAGTP 模型中,需要借助 CPU 端的调度线程来完成任务调度以及任务间依赖关系

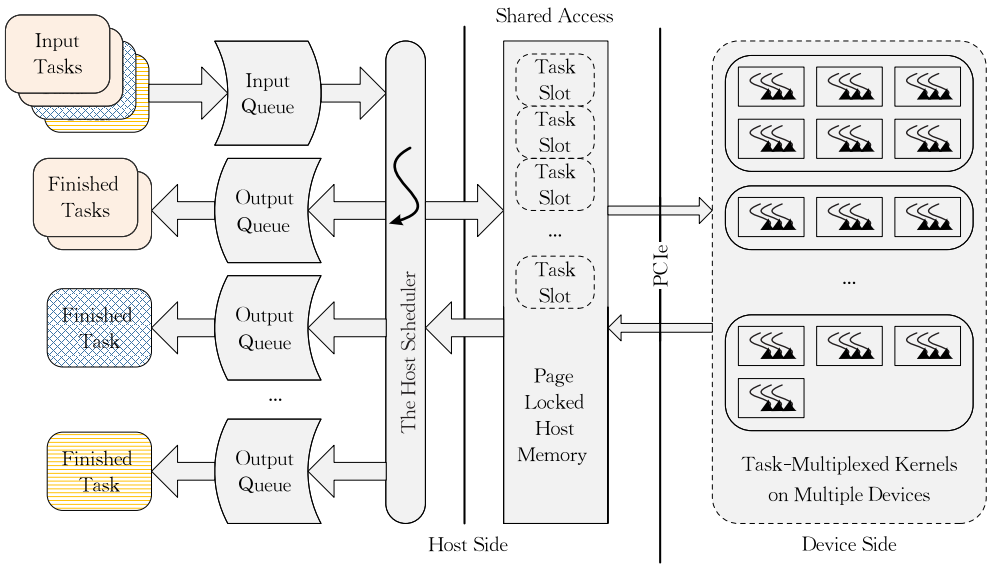


图 4　CAGTP 模型结构

处理,正因为这样便将该模型取名为 CPU 辅助的 GPU 线程池模型. 对于 CPU-GPU 异构系统整体而言,CAGTP 便于将 GPU 端设计集中于如何最大化任务的吞吐率和硬件资源利用率,而不会产生因为在 GPU 端进行复杂的判断和控制操作导致计算效率的下降的问题.

在 GPU 体系结构不支持页锁定内存等特性时,CPU 和 GPU 之间的交互以及 kernel 函数调用等都具有较高的开销,传统的消息传递模式并不能很好地支持细粒度、不规则的并行性应用. 因此,CAGTP 模型的设计目标是挖掘 CPU-GPU 异构 NUMA 结构特征,利用页锁定内存和 UVA 等共享内存支持,优化 CPU 和 GPU 之间的细粒度任务交互与并发执行性能,提高 GPU 硬件资源利用率,有效实现对具有复杂依赖关系任务的并行计算.

3.2 模型实现

CAGTP 模型的实现主要由主机端的输入输出队列、任务调度器、任务槽和 GPU 端的任务复用 kernel 函数四部分组成.

3.2.1 输入输出队列

CAGTP 模型包含一个任务输入队列和多个任务输出队列,这组任务队列是该模型的用户接口. 所有任务输入和输出队列都是线程安全的,即它们可以被多个主机端线程同时访问. 其中,输入队列可以是普通队列,也可以是带有优先级的队列. 用户将需要计算的任务压入任务输入队列,对每一个任务指定一个任务输出队列索引,进而在指定的任务输出队列中将计算完成的任务返回给用户. 任务输出队列的数目由用户指定,使用多个任务输出队列的目的在于方便地处理任务间的依赖关系.

3.2.2 任务调度器

任务调度线程、任务槽和任务复用 kernel 函数是 CAGTP 模型的核心部分,共同实现了计算线程块级别的动态任务调度. 如图 5 所示,调度线程是一个持续运行的 CPU 线程,负责将任务输入队列中的任务输入序列映射到 GPU 端的不同计算线程块上,由 GPU 端运行的任务复用 kernel 函数不断地接收和执行调度来的计算任务. 具体而言,调度线程主要做两个具体操作,一是将任务输入队列中的任务分配给空闲的任务槽,二是从任务槽中回收 GPU 端执行完成的任务. 最后,当需要结束任务复用 kernel 函数时,调度线程会在每个任务槽中放置一个带有特殊标记的任务,该任务将会结束 GPU 端的所有线程.

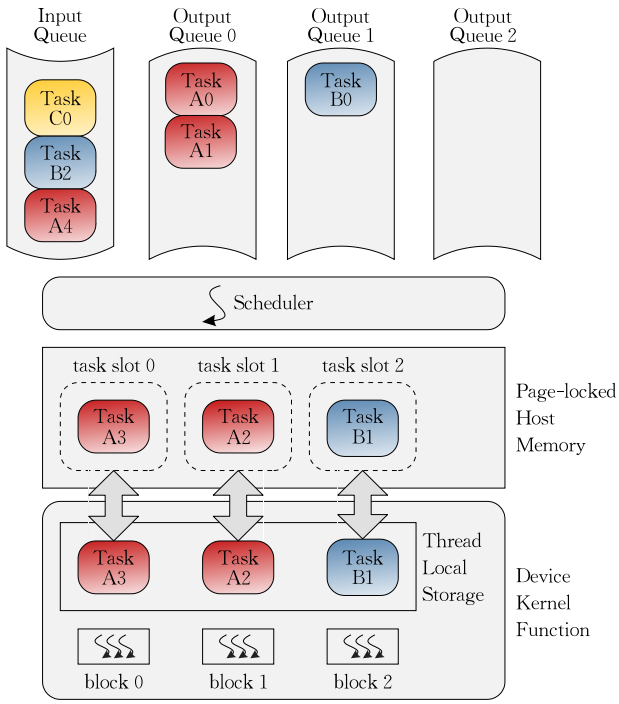


图 5 计算块级任务调度示意图

实现 CAGTP 模型的关键在于 CPU 和 GPU 之间的计算任务传递,这里是通过分配在页锁定内存中的一组任务槽来完成的. GPU 端的每个计算线程块都有一个专有的任务槽,当需要将某个计算任务调度给 GPU 端的计算线程块时,CPU 端调度线程将该计算任务拷贝到相应的任务槽中,由 GPU 端的计算线程块从任务槽中取出任务并予以执行. 当任务执行完成后,GPU 端的计算线程块修改相应任务槽的状态,以通知 CPU 端调度线程该任务已执行完毕,并由其将任务收回后压入指定的任务输出队列中.

页锁定内存能够被 CPU 和 GPU 两端的线程共享访问,该模型中 CPU 和 GPU 使用内存 load/store 指令直接访问页锁定内存中的共享区域,因此在 CPU 和 GPU 之间传递任务的开销很低. 传递一个任务的时间开销大约为 50 ns,其中包括计算任务被拷贝到任务槽以及从任务槽将任务拷贝到 GPU 端的两次拷贝,约为一次普通 kernel 函数调用时间开销的 1%,因此该模型能够有效支持 CPU 和 GPU 之间的细粒度任务交互.

3.2.3 任务槽

任务槽是在 CAGTP 模型中支持 CPU 和 GPU 之间进行高效任务调度的关键,是 CPU 端的调度线程和 GPU 端的计算线程块进行交互的主要载体. 在 CAGTP 模型的构造阶段,系统即在 CPU 端

的页锁定内存中分配任务槽. 在 Fermi 或者更新结构的 GPU 上，GPU 端的线程能够使用内存 load/store 指令直接访问页锁定内存，大大降低了在 CUDA 编程模型中使用 cudaMemcpy*()等数据接口进行 CPU 和 GPU 任务交互所产生的开销.

为了支持 CAGTP 模型中的计算线程块级的任务调度，任务槽被设计为有状态的，其状态转换关系如图 6 所示. 任务槽的状态被 CPU 初始化为 IDLE(空闲状态)；CPU 端的调度线程将任务压入任务槽后，将其状态变为 READY(就绪状态)；GPU 端的计算线程块发现与其对应的任务槽状态变为 READY 后，便从任务槽中取出计算任务予以执行，任务完成后计算线程块中的某个 GPU 线程将任务槽的状态置为 FINISHED(完成状态)；CPU 端的调度线程从状态为 FINISHED 的任务槽中回收 GPU 端执行完成的任务，然后将其压入指定的输出队列，并设置任务槽的状态为 IDLE；当所有的任务计算完毕，需要结束任务复用 kernel 函数的执行时，由 CPU 端的调度线程将任务槽的状态设置为 EXIT(退出状态)；GPU 端的计算线程块查询到对应的任务槽状态为 EXIT 时就结束执行. 在从 IDLE 到 READY 的状态转换中，CPU 端的调度线程必须在任务被拷贝到任务槽之后，才能进行相应的状态转换；同理，GPU 端也必须在任务计算结束，并对任务槽中的任务数据进行更新(如有)后，才能将任务槽的状态由 READY 变为 FINISHED.

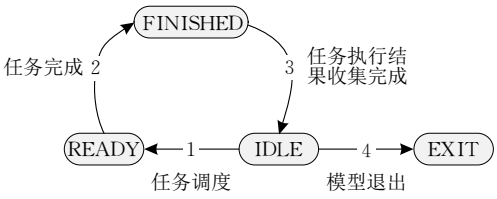


图 6　任务槽状态转换图

为了在保证任务调度效率的情况下简化 GPU 端的功能实现，在 CAGTP 模型构造阶段分配的任务槽的数目等于系统中所有 GPU 设备上计算线程块的总数目. 因此，GPU 端的每个计算线程块有自己专有的任务槽，并从中获取来自主机端的计算任务. 一个任务槽不会被 GPU 端的多个计算线程块共享访问，因此计算线程块之间不会因为获取任务而产生竞争，而且任务结束时也不会产生任务槽的访问冲突.

3.2.4　任务复用 kernel 函数

在 CAGTP 模型中，任务复用 kernel 函数在系统启动后被静态地启动，然后在每个 GPU 端的流多处理器上产生一个或多个计算线程块，并在任务调度期间一直驻留在 GPU 中并保持运行状态. 每个计算线程块从与其对应的任务槽中获取计算任务并执行，完成一个计算任务后尝试从任务槽中继续获取新的计算任务，直到接收到一个带有特殊标记的任务时退出.

如图 7 所示，使用任务复用 kernel 函数能够带来两方面的性能提升：一是避免了连续多个 kernel 函数之间的结束-停顿-启动开销，在 CAGTP 模型中一个任务调度开销约为 200 ns，而启动一个 kernel 函数则需要 4 μs～7 μs，因此可以有效地消除启动-停止 kernel 函数所带来的时间开销. 二是避免了每个 kernel 函数结束前一段时间内不能充分利用 GPU 硬件资源的情况，即 kernel 函数结束前的性能碎片，从而提高技术资源利用率.

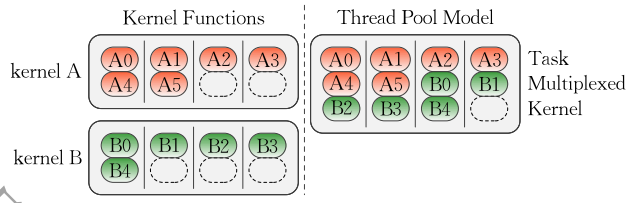


图 7　任务复用 kernel 函数

任务复用 kernel 函数是 CAGTP 模型中的核心，它打破了 CUDA 编程模型中 kernel 函数的边界，在 GPU 端的线程块级别重新组织计算任务，有力地支持了计算线程块级的任务调度. 任务复用 kernel 函数的复用有两层含义：首先，任务复用 kernel 函数中不同计算线程块执行的任务可以是不同的. 通过不同性质任务的并发执行，例如计算密集型任务和存储密集型任务，能够提高应用对 GPU 硬件资源的利用率. 其次，一个计算线程块多次执行的任务也是不同的，而且任务粒度比 kernel 函数的更细，这就可以将多个应用中原本需要多个 kernel 函数调用才能完成的计算融合在一起，消除多次 kernel 函数的调用开销.

在 CAGTP 模型中，在一个 GPU 设备上同一时刻只有唯一一个任务复用 kernel 函数在执行，并发任务执行消除了 CUDA 程序设计中使用多个流重叠计算的必要性，而且在该模型中重叠计算与数据传输比 CUDA 编程模型更为自然和简单.

3.3　编程接口

CAGTP 模型的设计目标是实现 CPU-GPU 异构 NUMA 系统上的不规则细粒度任务并行，因此

该模型提供了追踪 GPU 任务状态的方法.而在 CUDA 编程模型及其它相关工作中,一旦计算任务被调度到 GPU 端,CPU 便无法实时获取任务状态,只有等 GPU 端的全部 kernel 函数结束以后,CPU 才能获知 GPU 端的全部计算任务已经完成.

基于 CAGTP 模型的计算线程块级任务调度机制能够支持大量任务的动态创建与计算,单个 GPU 设备可被视为多个活动的计算线程块,从而给程序员提供了一个简单易用的计算模式.从程序员角度来看,CAGTP 模型就是一个任务输入队列和多个任务输出队列,将满足依赖关系的计算任务压入任务输入队列,然后在某个任务输出队列上等待结果.为实现对特定应用的计算,需要将数据划分为公共数据和任务特定数据,将计算表达成任务并压入任务输入队列,以及在计算过程中根据完成的任务判断哪些任务的依赖关系被满足.

在 CAGTP 模型上实现的主要 API 接口如下:

```
class cagtp_t {
  // start the kernels and the host scheduler.
  void run (void);
  void set_no_more_task_flag (void);
  // wait for the kernel and the host scheduler to exit.
  void synchronize (void);
  void push_task (task_t const &task, int output_index=0);
  // pop a task from an output queue, synchronous call.
  void pop_finish (task_t &task, int output_index=0);
  // try to pop a task from an output queue, asynchronous call.
  bool try_pop_finish (task_t &task, int output_index=0);
  // query how many tasks are exepcted from an output queue.
  size_t get_unfinished_count (int output_index=0);
};
```

在模型运行过程中,通过调用 push_task()接口可以在任何时刻将计算任务压入任务输入队列中,并为其指定一个输出队列索引.程序员可以调用非阻塞的 try_pop_finish()接口检查某个任务输出队列中是否有完成的任务,或调用 pop_finish()接口在某个任务输出队列上等待任务完成.程序员也可以在任何时候通过调用 get_unfinished_count()接口查询某个输出队列上还有多少计算任务未完成或在队列中等待被弹出.

在 CAGTP 模型中,假定被压入任务输入队列中的任务都能够被并发执行,即这些任务之间是没有依赖的.CAGTP 模型本身并不处理任务间的依赖关系,而是通过多个任务输出队列将完成的任务返回给用户进程,用于进行任务之间的依赖性判断.程序员可以使用该模型提供的 API 在上层处理任务间的依赖关系,下面给出了判断任务间依赖关系的示例.假定需要处理三个任务 a、b 和 c,且任务 c 依赖于任务 a.首先,任务 a 和 b 被压入到任务输入队列中,并且指定不同的任务输出队列索引;然后通过调用 pop_finish()接口在任务输出队列 0 上等待任务 a 完成;接下来,当任务 a 从任务输出队列 0 中被取出后,任务 c 就可以被压入任务输入队列中执行,从而保证了任务 c 与任务 a 之间的依赖关系.由于任务 b 被指定了一个不同的任务输出队列索引,所以任务 b 的执行能够与任务 a 和 c 的执行并发进行.

```
task_t a, b, c, tmp;
cagtp.push_task (a,0);
cagtp.push_task (b,1);
cagtp.pop_finish (tmp,0);
cagtp.push_task (c,0);
cagtp.pop_finish (tmp,1); //got task b
cagtp.pop_finish (tmp,0); //got task c
```

与 CUDA 编程模型相比,CAGTP 模型主要有以下优点:

第一,使程序员的关注点从 GPU 整体变为多个活动的计算线程块,将 GPU 程序设计的思维模式从如何聚合出能够占用整个 GPU 的 kernel 函数变为如何将应用问题表达为计算任务.任务的粒度比 kernel 函数小很多,利用 CAGTP 模型能够很好地实现细粒度任务的并行执行.随着 GPU 硬件的发展,编写大 kernel 函数利用整个 GPU 的硬件资源会更加困难,而基于 CAGTP 模型,只要同一时刻有多于计算线程块的任务,就能通过足够多任务的并发执行充分利用 GPU 的硬件资源.

第二,通过将数据传输表达为带有依赖关系的任务,便于程序员实现数据传输与任务计算两种操作的重叠.基于 CAGTP 模型的编程消除了使用多个 CUDA 流处理计算任务间依赖的必要性,即使使用 CUDA 流技术进行计算和传输的重叠,也明显降低了程序员对 GPU 硬件的细节考虑.

第三,支持细粒度 CPU-GPU 协作式程序设计,这是该模型与使用持久化 kernel 或者 CKE 等编程模式的最大不同.通过任务输入输出队列和任务槽等技术实现了 CPU 和 GPU 之间高效的不规则细粒度任务交互,并且根据任务的计算性质来决定该任务在 CPU 还是 GPU 上执行,易于实现 CPU 和 GPU 协同计算.

第四,使程序员不用过多考虑 GPU 底层硬件

的细节，而且方便实现一个 GPU 设备被多个 CPU 端进程共享使用. 基于 CAGTP 模型，程序员不用直接操作 GPU 线程，而只需要将任务压入任务输入队列，由模型进行 GPU 硬件线程管理和负载均衡等工作，可以有效地提高 CPU-GPU 异构系统上的编程效率.

4　实验及结果分析

本节主要分析和验证 CAGTP 模型的应用性能及其可扩展性. 实验在一台同时支持 4 块显卡的工作站上进行，主要硬件配置如下：

CPU：Intel Ivy Bridge E5-2620V2 6 核

主存：16 GB 1600 MHz DDR3

总线：PCIe x16

GPU0：NVIDIA GTX480

GPU1：NVIDIA Tesla K40M

GPU2：NVIDIA GTX780

操作系统：Ubuntu 14.04

CUDA 版本：CUDA Toolkit 6.5

4.1　微基准测试

微基准测试是根据待测系统的特性专门编写的用于测量其某一方面性能指标的程序，本节从任务结构体大小、计算线程块数量以及 PCIe 带宽消耗等方面对模型进行分析.

4.1.1　任务结构大小

对每一种任务结构体大小测试 100 000 个任务调度的总时间，并计算平均每个任务的调度开销. 由于任务中没有包含任何计算指令，因此平均每个任务的调度开销主要来自于 CAGTP 模型的运行时调度. GTX 480 和 Tesla K40 分别有 16 和 15 个流多处理器，在每个流多处理器上启动四个计算线程块，实验结果如图 8 所示.

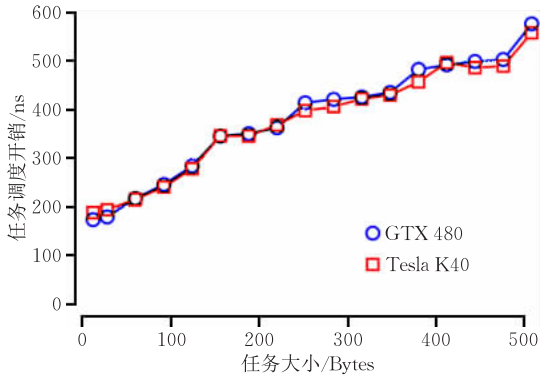


图 8　任务结构体大小对调度开销的影响

实验结果表明，任务调度开销随任务结构体大小增加而增加. 典型的 16 字节～32 字节的任务平均开销约为 200 ns，而一次 kernel 函数调用开销约为 4 μs，是 CAGTP 任务调度开销的 20 倍. 在两种 GPU 上的结果一致表明了调度开销主要是由 CAGTP 调度过程中线程安全队列的出入队、通过 PCIe 总线将任务传递到 GPU 端等操作引起的，GPU 端的计算能力对开销影响不大. 进一步分析表明，大约 160 ns 耗费在操作输入输出队列上，调度线程计算、拷贝任务到任务槽、传递任务到 GPU 端等花费 40 ns 左右. 在 CAGTP 模型中完成一个任务要经过四次出/入队列，对于线程安全的任务队列，出入队列实际上是一个串行化的过程，这一部分开销不能被多个任务均摊.

4.1.2　计算线程块数量

任务结构体大小固定为 24 字节，测试使用不同数量的计算线程块完成 100 000 个任务的总时间，并计算出平均每个任务的调度开销. 同样，任务中不包含任何计算指令，因此每个任务的平均调度开销反映了计算线程块的数量对模型运行时调度开销的影响情况. GTX 480 和 Tesla K40 的每个流多处理器上最多驻留 8 和 16 个活动线程块，测试结果如图 9 所示.

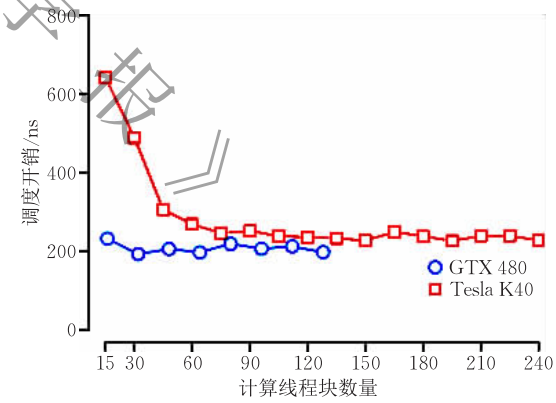


图 9　计算线程块的数量对平均调度开销的影响

对于 GTX 480 而言，任务的平均调度开销几乎不受计算线程块数量的影响，在每个流多处理器上启动一个计算线程块就可以较好地隐藏任务的调度延迟. 而在 Tesla K40 上，每个流多处理器上至少启动 3 个计算线程块才能够较好地隐藏任务的调度延迟，这意味着在 Kepler 架构的 GPU 上运行 CAGTP 模型时，如果所使用的计算线程块数量较少，有可能由于不能较好地隐藏任务的调度延迟而导致应用整体性能达不到最优.

4.1.3 PCIe 带宽

CAGTP 模型中的任务复用 kernel 函数以忙等方式从任务槽中拷贝任务，因此即使没有向输入队列中压入任务，运行时任务调度操作也会占用一部分 PCIe 带宽. 任务结构体大小固定为 24 字节，启动 CAGTP 模型但不调度任务，记录不同数量的计算线程块在一段固定时间内的任务复用 kernel 函数访问任务槽的总次数，然后计算 PCIe 带宽消耗，结果如图 10 所示.

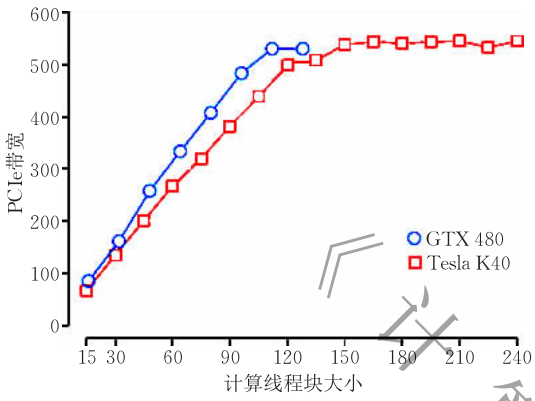


图 10 PCIe 带宽消耗

运行测试 PC 机的 PCIe×16 总线带宽大约为 6.3GB/s，任务复用 kernel 函数的 PCIe 带宽消耗随着计算线程块数量的增加而持续增加，在 GTX 480 上使用超过 112 个计算线程块或在 Tesla K40 上使用超过 120 个计算线程块时，PCIe 带宽消耗达到最大值(约 9%)，此时任务复用 kernel 函数访问任务槽的频率最高都达到 1.4 亿次/秒.

因此，如果任务需要通过 PCIe 总线访问 CPU 端的内存，那么就有可能与 CAGTP 模型中的任务复用 kernel 函数竞争 PCIe 带宽，同时降低应用和线程池调度的性能. 而且在 Tesla K40 上使用过多的计算线程块会增加单任务延迟. 在实际应用中，使用的计算线程块越多，任务就应该具有越高的计算通信比，尽量避免与 CAGTP 模型竞争 PCIe 带宽资源.

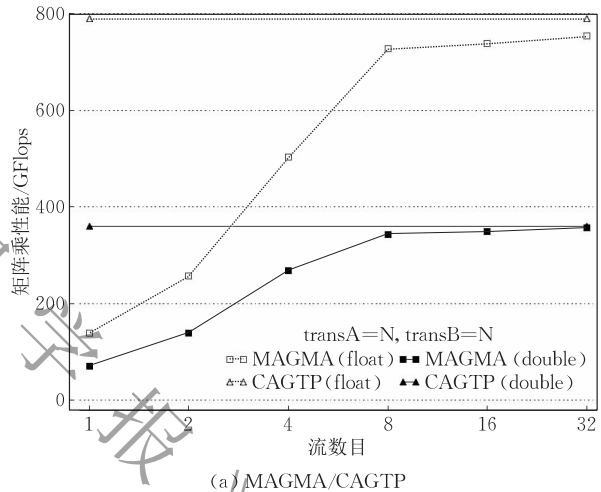
4.2 线性代数应用

4.2.1 批量通用矩阵乘

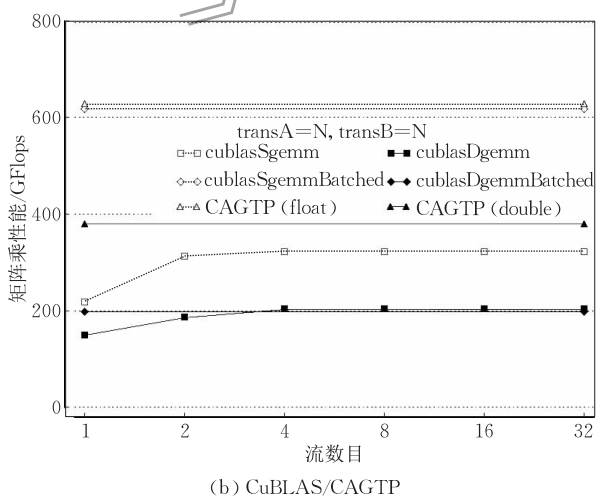
通用矩阵乘 GEMM(General Matrix-Matrix Multiplications)是线性代数中最基础的操作，在实际应用中往往也是最耗时的操作，其时间复杂度为 $O(n^3)$[25]. 批量通用矩阵乘(Batched GEMM)通过同时计算多个小规模 GEMM，旨在验证 CAGTP 模型的细粒度任务并行相对于普通 kernel 函数调用方式具有更好的性能.

本文实验中使用 CUDA 编程模型支持的流数据传输，使用 CuBLAS 和 MAGMA 提供的接口函数完成多个 GEMM 的计算. 此外，在 CuBLAS 函数库中专门针对 batched GEMM 提供了程序接口 cublas〈t〉gemmBatched()，可以直接实现批量矩阵乘. 在 CAGTP 模型中，本文采用了 GEMM 标准接口中的 kernel 函数，同时计算 128 个 GEMM，其中每个任务的矩阵规模为 64×64.

首先，实验测试使用 1～32 个 CUDA 流时并发调用 cuBLAS 单次 GEMM 接口的性能. cuBLAS 中的批量矩阵乘接口 cublas[DS]gemmBatched()以及 CAGTP 模型都只用一个 CUDA 流. 如图 11(a)所示，cublas[DS]gemm()接口在使用 4 个 CUDA 流时达到最佳性能，但其最佳性能仍然不如 CAGTP 模型.



(a) MAGMA/CAGTP



(b) CuBLAS/CAGTP

图 11 批量矩阵乘性能

CuBLAS 的两个函数在计算小规模矩阵乘时会调用特殊优化的 kernel 函数，为在 GPU 的流多处理器间实现负载均衡，这些 kernel 函数将结果矩阵

划分为小尺寸子矩阵，因而这些 kernel 函数在计算小规模 GEMM 时能充分利用 GPU 的流多处理器. 但由于每个线程块上的计算量非常小，这些 kernel 函数无法达到单次 GEMM 接口计算大规模矩阵乘时的性能. CAGTP 模型避免了 kernel 函数的多次启停开销，其性能高于 cublas〈t〉gemm(). CuBLAS 中的批量矩阵乘接口 cublas〈t〉gemmBatched() 是为批量小规模矩阵乘设计的，也使用较小尺寸的子矩阵划分. 对单精度和双精度的情形，其性能都低于 CAGTP 模型.

其次，使用 MAGMA 代码方式进行 GEMM 计算实验，如图 11(b)所示. 相比于 MAGMA，基于 CAGTP 模型的批量 GEMM 实现能够获得稳定且更高的性能. 而基于 MAGMA 的批量 GEMM 实现则需要启动更多的 CUDA 流，并且达到 8 个 kernel 函数的并发执行后才能获得接近于 CAGTP 模型的批量 GEMM 实现性能.

基于 CAGTP 模型的批量 GEMM 实现能够获得更好的计算性能，主要得益于在 CAGTP 模型上实现的任务调度机制，能够有效地避免多次 kernel 函数调用的启停开销，并且只需要使用 1 个 CUDA 流，不需要对 kernel 函数进行复杂的优化操作.

4.2.2　Cholesky 分解

Cholesky 分解是线性代数中一种重要的矩阵分解算法，广泛应用于科学和工程计算中，其目标是将一个矩阵 $\boldsymbol{A}$ 分解为一个下三角矩阵 $\boldsymbol{L}$ 与其转置矩阵 $\boldsymbol{L}^{\mathrm{T}}$ 的乘积，即 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$[26]. 在使用分块策略计算 Cholesky 分解过程中，多个任务之间存在依赖关系，本文以 Cholesky 分解为例来验证基于 CAGTP 模型能够有效地处理不规则并行计算.

在 MAGMA 的 dpotrf_gpu() 接口函数中，用到了一系列 BLAS 和 LAPACK 接口，主要包括 SYRK(对称 k 秩更新)、GEMM、TRSM(求解三角方程组)和 TRMM(三角矩阵乘)等接口. 本文将 TRSM 分解为矩阵求逆 TRTRI 和三角矩阵乘 TRMM 两个部分，使用 kernel 函数 dgemm_kernel_fermi() 实现 GPU 端的 SYRK、GEMM 和 TRMM 操作，进一步基于 CAGTP 模型实现了 Cholesky 分解. 针对每一个子矩阵，将 SYRK、GEMM 和 TRMM 操作都作为一组计算任务，CAGTP 模型维护一个依赖数组处理任务之间的依赖关系.

双精度正定矩阵的 Cholesky 分解结果如图 12 所示，其中 Kernel 表示直接使用 kernel 函数 dgemm_kernel_fermi() 的实现，Kernel Opt 表示使用 CuBLAS 库中的 cublasDsyrk() 和 cublasDtrmm() 两个函数来实现 SYRK 和 TRMM 两个操作，在实现过程中不存在冗余计算，CAGTP 表示基于 CAGTP 模型的 Cholesky 分解实现. CAGTP 实现取得较其它两种实现更好性能的原因有两方面. 第一，由于能够通过压入任务控制参与计算的部分，所有位于对角线下的 64×64 大小的子矩阵在 CAGTP 实现中是完全不被计算的，因此 CAGTP 实现节约了一部分冗余的计算. Kernel 实现与 kernel 优化实现的性能差异能够近似地反映由于节约计算所带来的性能提升. 第二，SYRK、GEMM 和 TRMM 三个步骤的计算任务在 CAGTP 实现中能够几乎连续地被执行，因此减少了 GPU 硬件的空闲时间，提升了对 GPU 硬件的利用率.

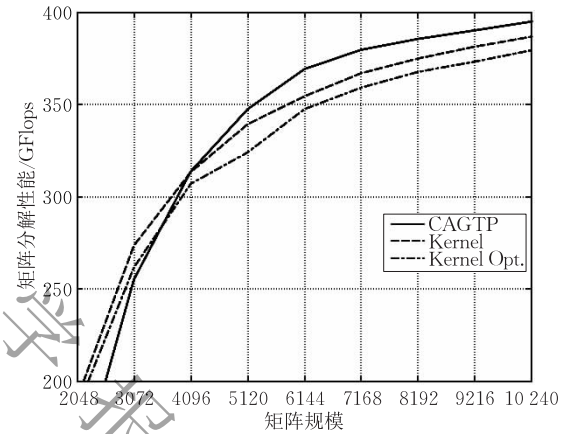


图 12　Cholesky 分解性能

4.3　机器学习应用

4.3.1　TNN 分类

T 近邻(T Nearest-Neighbor，TNN)算法是一种简单有效的监督分类算法，其主要思想是通过计算不同样本之间的距离来进行分类[27].

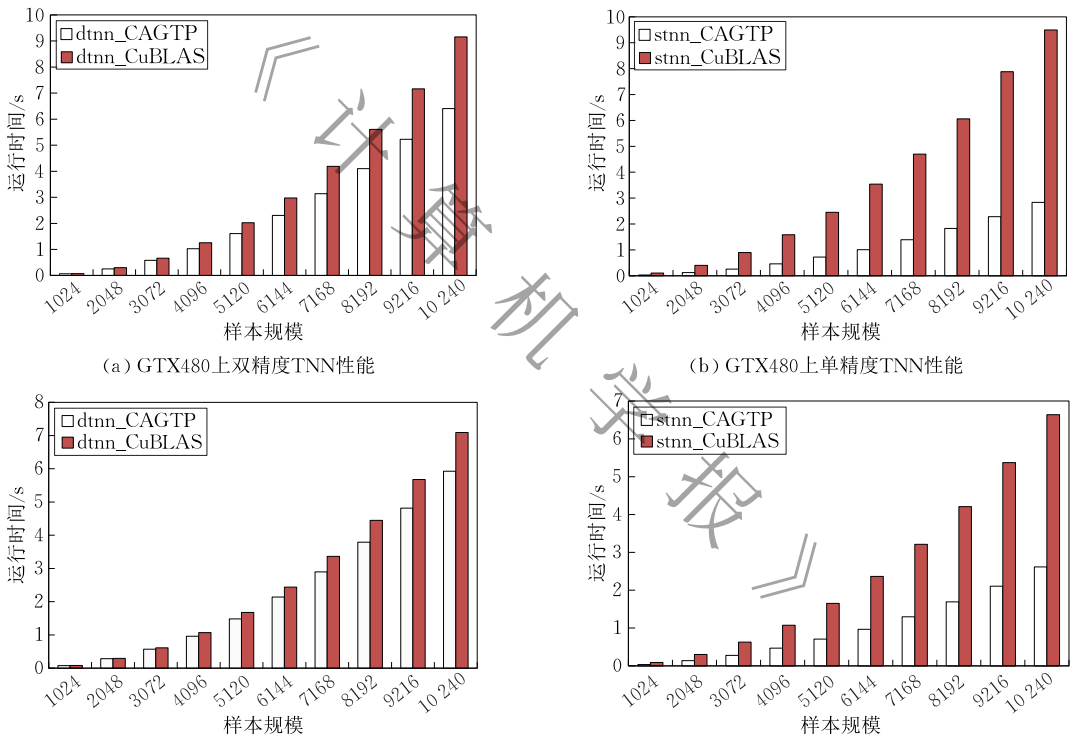
假设训练样本集为 $\boldsymbol{\chi}=\{(\boldsymbol{x}_i,y_i)\mid \boldsymbol{x}_i\in\mathbb{R}^d\}_{i=1}^n$，其中 $\boldsymbol{x}_i$ 表示第 i 个样本的 d 个特征维度上的特征值组成的向量，y_i 表示第 i 个样本的分类标签，n 为样本个数. 在进行 TNN 分类时，对于一个没有标签的新样本，通过计算其在特征空间中与其它样本的欧氏距离，然后选取样本集中特征与其最相似(距离最邻近)的 T 个分类标签，就可以知道每个样本与其所属分类的对应关系. 在 TNN 分类算法中，计算新样本与原有样本集中的样本点的相似度是最耗时的，也就是计算如下欧氏距离的操作，

$$dist(\boldsymbol{x}_i,\boldsymbol{x}_r)=\|\boldsymbol{x}_i-\boldsymbol{x}_j\|^2=\|\boldsymbol{x}_i\|^2-2\boldsymbol{x}_i\cdot\boldsymbol{x}_j+\|\boldsymbol{x}_j\|^2.$$

基于 CAGTP 模型实现 TNN 分类算法时，本文将欧氏距离的计算分解为两个步骤，可以直接基于现有的高性能 CuBLAS 库来实现矩阵乘操作. 首先，通过将样本集的特征矩阵乘以自身的转置得到 $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$，此步骤的矩阵乘是最耗时的操作；然后，利用样本特征的二范数计算样本间的欧氏距离.

TNN 算法实现主要包括计算欧式距离和寻找最近距离的排序操作，在 CAGTP 模型中的实现分为两步：首先在 GPU 端计算样本集与自身转置的乘积，然后 CPU 使用所得矩阵中的元素求出两两标本点间的相似性距离并插入到 n 个最大堆中，得出稀疏相似矩阵 $\boldsymbol{S}$.

实验结果如图 13 所示，其中数据集是随机生成的，所有时间数据是运行 512 次取平均的结果. 无论单精度还是双精度，基于 CAGTP 模型的 TNN 算法实现都具有更高的性能，主要原因是借助 CPU 和 GPU 共享的页锁定内存，针对细粒度的多个计算任务，能够达到很好的 CPU 计算（构建最大堆）和 GPU 计算（计算欧式距离）以及数据传输等方面的重叠执行，将时间缩短至 GPU 计算所需要的总时间与 CPU 端处理所需要的总时间两者中的最大值. 对于使用 CuBLAS 实现的 TNN 算法，由于计算过程中 CPU 和 GPU 数据传输与各自的计算不能做到 CAGTP 模型中的重叠，CPU 端的最大堆排序和数据传输大大影响了整体运行时间.



(a) GTX480上双精度TNN性能　(b) GTX480上单精度TNN性能

(c) Tesla K40上双精度TNN性能　(d) Tesla K40上单精度TNN性能

图 13　CAGTP 和 CUBLAS 实现的 TNN 性能

4.3.2　K-means 聚类

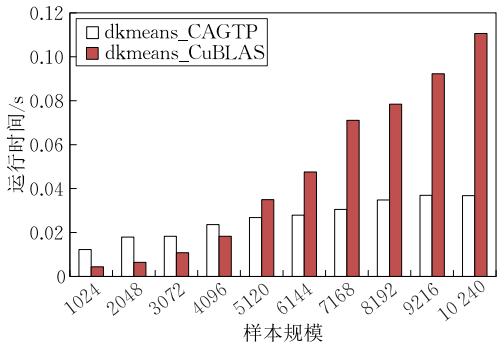
K 均值（K-means）算法是无监督学习的聚类算法中最基础的一种[28-29]，与 TNN 分类算法一样，通常与其他算法结合使用来达到更好的聚类效果. 聚类的目标是为了将相似的样本数据归为一簇，K 均值聚类算法就是将样本数据分为 K 个不同的簇，其中每个簇的中心是用簇内所有样本的均值计算而来的.

在使用 K 均值算法进行聚类时，首先直接在样本数据集中选择 K 个样本作为初始质心（一般是随机选择），然后通过重复地将样本点分配到最近的簇中，并计算簇中所有样本的均值来更新簇心，直到发现簇分配结果没有变化时表示聚类结束，返回当前的簇心. 每一步寻找新的簇心时，需要计算每个样本点到簇心的欧氏距离，这是 K-means 算法求解过程中最耗时的操作.

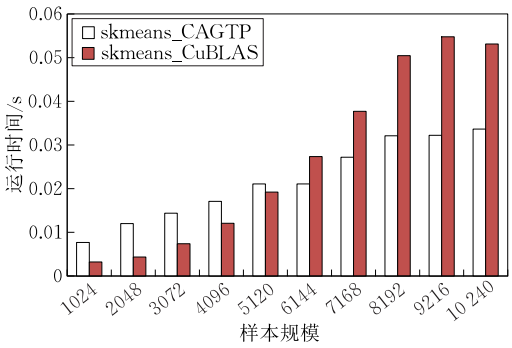
基于 CAGTP 模型实现 K 均值聚类算法的过程与 TNN 分类算法类似，以矩阵乘的方式求解 $\boldsymbol{x}_i \cdot c_j$，将其分解为在 GPU 上执行的多个细粒度计算任务. CPU 端的线程可以通过读取页锁定内存中的输出队列，获取在 GPU 上执行完成的任务结果，

进而计算出样本点到每个簇心的距离.

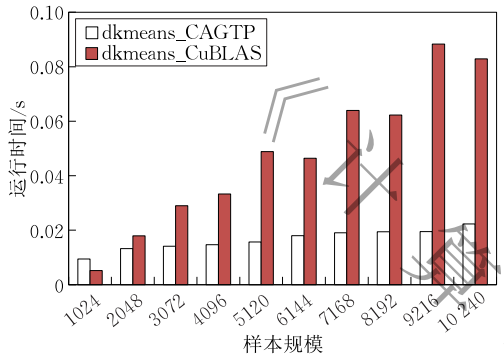
实验结果如图 14 所示，其中基于 CuBLAS 的实现采用了 cublas⟨t⟩gemm()接口函数. 数据集是随机生成的，样本规模从 1024 逐步增大到 10 240，其中每个样本的特征维度均为 1024，并且都是将数据集聚类为 128 类，即 $K=128$.



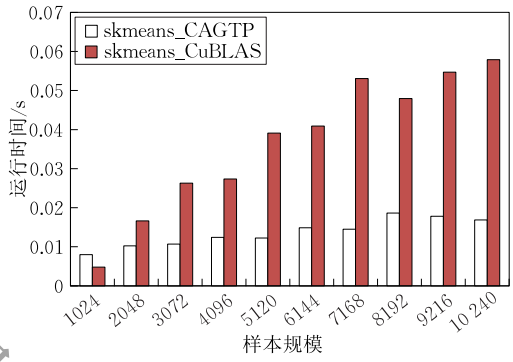
(a) GTX 480上双精度K-means性能



(b) GTX 480上单精度K-means性能



(c) Tesla K40上双精度K-means性能



(d) Tesla K40上双精度K-means性能

图 14　CAGTP 和 CUBLAS 实现的 K-means 性能

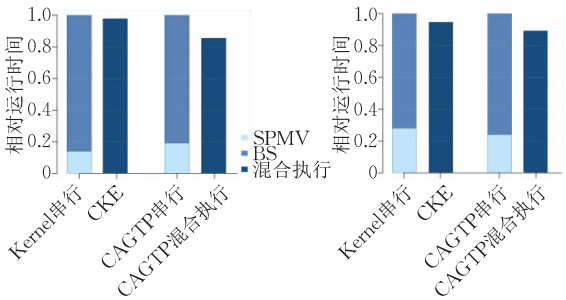
无论是采用单精度还是双精度，在数据集规模比较小时，由于任务数及其计算量相对较小，任务间的同步和调度开销以及 CAGTP 模型的初始化开销所占比例较大，因此基于 CuBLAS 的矩阵乘来计算欧氏距离的执行时间更短. 但当数据集达到一定规模时，尤其是对于双精度而言，随着任务数及其计算量逐步增加，基于 CAGTP 模型实现的 K-means 算法能够充分利用 GPU 硬件资源以及 CPU 和 GPU 间的任务交互能力，进而达到更高的计算性能.

4.4　混合任务应用

稀疏矩阵向量乘(SPMV)和 Black Scholes 算法是两种具有不同资源需求的应用. SPMV 是线性代数中稀疏矩阵运算的常用操作之一，本文使用 CSR 格式存储矩阵数据，在计算过程中一次访存后做一个浮点乘法和一个浮点加法操作，因此整个计算过程需要大量的访存操作[30]. Black Scholes 算法主要用于金融领域中求解著名的期权定价公式布莱克-斯科尔斯方程，该方程是一个二阶偏微分方程，求解过程中有较多的超越函数运算(如指数预算、对数运算等)以及浮点数除法等操作[31].

SPMV 和 Black Scholes 算法两者对硬件资源需求不同，前者对存储带宽要求比较高，GPU 的高访存带宽能够有效地执行 SPMV 操作；后者属于计算密集型操作，数据量及其传输要求不高，但有大量计算操作需要由处理器执行. 利用 CAGTP 模型可以容易地混合不同性质的计算任务予以并行执行，充分发挥 CPU-GPU 异构系统上以任务并行方式求解应用问题的性能.

本文在CPU-GPU异构系统上对SPMV和Black-Scholes算法进行四种不同的实现：(1) 采用 CUDA 编程模型实现 SPMV 和 BlackScholes 算法，称为 Kernel 串行实现；(2) 采用 CKE 技术，将求解 SPMV 和 BlackScholes 算法的两个 kernel 函数轮流启动在已经创建好的 CUDA 流上，使其并发执行；(3) 在 CAGTP 模型上使用串行和并行两种方式执行 SPMV 和 Black Scholes 算法的计算任务. 在 GTX480 和 Tesla K40M GPU 上的执行结果如图 15 所示，其中运行时间数据是 512 次执行求平均计算而来.



(a) GTX 480 (b) Tesla K40

图 15 混合任务性能

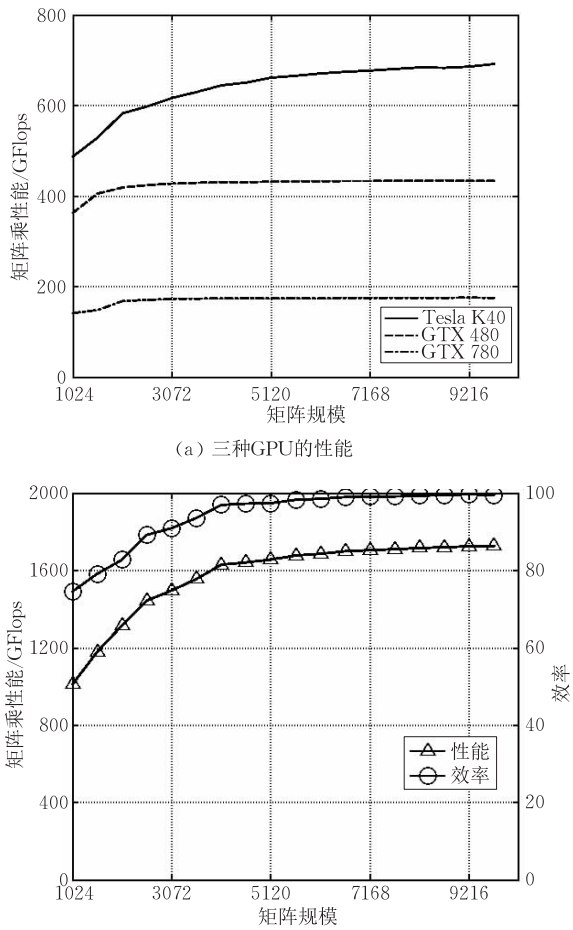
可以看出,基于 CAGTP 模型的混合任务并行执行比 CKE 获得了更好的性能,节省了 10%左右的运行时间. 同时,串行执行方式与 kernel 串行方式的运行时间一样,表明了 CAGTP 模型引入的额外开销可以忽略不计. 使用 CKE 技术时,只有数据传输与 kernel 函数执行操作重叠,同一时刻在一个 GPU 上只能执行一个 kernel 函数,无法充分发挥 GPU 的性能. 而基于 CAGTP 模型则能够实现混合任务的并发执行,尤其是应用对系统的资源需求不同时,能够提高 CPU-GPU 异构系统的访存带宽、GPU 计算核心等多种资源的利用率,进而提升系统的整体计算性能.

4.5 多 GPU 支持

在单 CPU 和多 GPU 构成的 CPU-GPU 异构系统中,CAGTP 模型将多个 GPU 设备视为一个统一的流多处理器. 由于 GPU 型号的不同,比如 GTX 480 和 Tesla K40M 等,不同 GPU 上的流多处理器会具有不同的特性和性能. 在进行 GPU 端的任务调度时,由 CPU 端的调度进程负责不同流多处理器之间的负载动态均衡,尤其是对于性能差异的多个 GPU 设备而言,负载均衡处理尤其重要.

本文实验使用 GEMM 对三个型号 GPU 的性能测试结果如图 16(a)所示,由于 GTX 780 对双精度运算只有有限支持,其双精度计算性能只有单精度的 1/24. 而 Tesla K40 的双精度性能达到 GTX 480 的 1.5 倍以上[4]. 由此可见,三种 GPU 的性能差异很大,通过使用较大规模的 GEMM 问题求解,能够很好地验证 CAGTP 模型的扩展性及其任务调度的动态负载均衡能力.

首先定义多 GPU 效率为使用 4 个 GPU 同时完成特定规模的 GEMM 所达到的性能与使用 4 个 GPU 单独完成同样规模的 GEMM 的各自性能之和的比值. 在上述 4 路 GPU 工作站上求解 GEMM 的性能及其效率结果如图 16(b)所示. 对于矩阵规模大于 4096 的 GEMM,其效率达到 95%以上,而



(a) 三种GPU的性能

(b) 多GPU性能

图 16 模型的多 GPU 支持

且当矩阵规模大于 6144 时效率则达到了 99%左右. 由于 GTX 780 的性能约占 4 个 GPU 性能之和的 10%,达到 90%以上的效率说明在 CAGTP 模型中,任务调度机制能够有效地利用 GTX 780 的计算性能,具有很好的负载均衡能力. 但是当矩阵规模小于 3072 时,由于分解得到的任务数量比较少,无法较为均衡地调度给所有计算线程块,因此在小规模 GEMM 求解时,导致各 GPU 负载差异较大,致使系统效率偏低.

实验表明,在具有多个 GPU 设备的 CPU-GPU 异构系统环境中,CAGTP 模型具有良好的扩展性以及负载均衡能力. 并且在应用问题规模比较大,能够分解出足够多计算任务的情况下,系统的计算效率能够达到 100%,进而说明基于 CAGTP 模型能够充分利用 GPU 等硬件资源,有效地提高系统的整体计算性能.

5 相关工作

CUDA 编程模型提供了 GPU 上的数据并行机

制，在 CPU-GPU 异构系统上实现了类消息传递的编程模式，但不能有效地支持任务并行. 在 CUDA 编程模型基础上，目前已有较多工作研究 CPU-GPU 异构系统上的共享内存编程模式，主要包括三类：CKE 技术、持久化 kernel 和 kernel 融合技术，实现了 GPU 上或者包含 CPU-GPU 交互的任务并行编程模式.

(1) CKE 技术

该技术主要基于原 CUDA 编程模型实现对多个 kernel 函数的并发执行，也可实现对不同资源需求的 kernel 函数的组合，达到充分利用资源提升性能的目标. 文献[10-13]提出并实现了基于 CKE 技术的多 kernel 优化方法，能够提高 GPU 的资源利用率以及应用程序的整体性能. Pai 等人[7]提出了弹性 kernel 概念，基本实现对 GPU 上的 kernel 函数所需资源的细粒度控制，能够增加并发 kernel 函数的数量，并降低单个 kernel 函数的等待时间，提高了系统的整体吞吐量. 利用弹性 kernel 将多个 kernel 函数合并并进行 kernel 并发执行时，当这些 kernel 具有不同的资源需求时，该机制能够显著提升应用的整体求解性能[11]. Wende 等人[12]提出一种 kernel 函数重排序方法，根据各个独立的 kernel 函数的资源需求对其调用顺序进行调整，能够提升 kernel 函数并发执行的性能. Wang 等人提出的上下文汇集模型可以并发执行来自不同 GPU 程序的多个 kernel 函数，并能获得比 CUDA 上下文切换方式更好的性能[13]. Zhong 等人[32]实现了对 kernel 函数进行动态分片的系统 Kernelet，将 kernel 函数分片后可以根据资源需求进行调用，更好地提升 GPU 资源利用率和 kernel 并发执行的性能. 在 CUDA 编程模型的 CKE 技术中，kernel 函数的执行完全由 GPU 硬件调度来决定[10-13]，而 CAGTP 模型则允许程序员对任务调度及执行过程进行更好的控制，通过这种显式的任务执行过程控制能够更好地优化任务调度及资源利用，进而提升系统的整体性能.

StarPU 是一个基于 CKE 技术实现的 CPU-GPU 异构系统上的并行任务调度系统[33]，通过将任务表达为有向无环图(Directed Acyclic Graph, DAG)的形式，可以处理具有复杂任务依赖关系的应用. 对于 DAG 表达的不规则并行应用，对每个任务独立调用 kernel 函数并利用 CUDA 模型的 kernel 并发执行技术，由于受到上下文切换、kernel 函数启停等开销的影响，性能仍然受到很大影响.

(2) 持久化 kernel 技术

持久化 kernel 技术是将任务动态地调度到一个始终处于运行状态的 kernel 函数上执行，进而实现 GPU 端的任务并行化[14-17]. Chen 等人[14]设计的运行时系统通过异步拷贝将任务调度到驻留 GPU 上的 kernel 函数中，尽管以批次方式能够降低其调度开销，但任务的平均调度开销仍达到 1 μs，接近于启动一个 kernel 函数的开销，而且 CPU 和 GPU 之间的交互进一步制约了持久化 kernel 函数的执行效率. Tzeng 等人[15]在单 GPU 上使用持久化 kernel 函数实现了多任务调度的并行执行. Chatterjee 等人[16]通过引入“finish-async”方式的 API，能够对不规则的应用取得良好性能. 他们虽然都在 GPU 上实现了具有任务依赖关系的应用求解，但该系统并不支持 CPU 和 GPU 协同计算. Krieder 等人[17]设计的 GEMTC 框架将任务队列存储在 GPU 的全局内存中，但其平均任务调度时间仍为 63 μs. Sun 等人[34]设计了一种非终止式 kernel 函数执行模型，对 GPU 上的 Linux 等操作系统的 kernel 函数进行了并行加速. Peters 等人[35]使用驻留 kernel 函数处理多个用户对同一物理 GPU 设备的多个任务请求，提高了 GPU 设备的利用率.

不同于 CUDA 编程模型的 Fork-Join 模式，持久化 kernel 技术通过 CPU 上的调度线程实现任务调度，绕开了 CUDA 编程模型中硬件调度的限制[36]，在一定程度上提高了资源利用率及计算性能. 但是，持久化 kernel 技术对于 CPU-GPU 协同计算的支持仍需改进，并且 GPU 上的任务调度开销仍然较大.

(3) Kernel 合并/融合技术

在 CUDA 编程模型中，通过对不同资源需求的 kernel 函数进行合并或融合，形成一个大 kernel 函数，等效于在 GPU 上实现多个 kernel 函数的并发执行[37-38]. 由于不同的 kernel 函数对 GPU 资源需求的互补性，在一定程度上能够提高计算吞吐率. Calhoun 等人研究了多个 kernel 函数合并时可能存在的问题，例如访存、CPU-GPU 任务交互等冲突，单个 kernel 函数合并后很难达到合并前的性能，使得 kernel 合并/融合技术的实际应用存在较大困难[19]. 而且由于 GPU 硬件的快速发展，单个 GPU 上的处理核心及存储资源都大大增加，利用目前的 CUDA 编程模型，很难通过 kernel 合并/融合技术形成一个足够大的 kernel 函数来充分利用 GPU 的各种资源.

Gelado 等人在 CPU-GPU 异构系统上设计了非对称分布共享内存编程模型(ADSM)[39]，它允许 CPU 和 GPU 之间通过统一地址空间进行数据管

理,并且非对称式共享内存设计便于更好地保持数据一致性. Cabezas 等人提出一种共享存储下多 GPU 节点自动化 Kernel 并行框架(AMGE)[40],从体系结构角度进行多 GPU 节点间的 NUMA 架构分析,使用 GPU 间的远程访问特性,通过线程块调度策略使数据能够在各 GPU 间进行有效分解和访问,达到很好的性能加速效果.但两者并没有详细分析任务级并行特性,较难充分发挥 CPU-GPU 异构系统性能,而且 AMGE 只有在数据访问消耗低于 5%时才达到最高性能,应用较为单一.

CAGTP 模型是在视 CPU-GPU 异构系统为共享存储结构的基础上提出的,其优点在于能够通过页锁定内存共享区域进行高效的同步与任务交互.由 CPU 端的调度线程实现线程块级别的任务调度,消除了 kernel 函数启动和停止的开销,而一个任务的调度开销仅为 200 ns 左右,大大降低了系统的任务调度开销.而且,调度线程通过任务槽以及输入输出队列能够得知 GPU 端任务的执行状态,进而实现对任务依赖关系的处理,达到对细粒度不规则任务并行模式的有效支持.

6 总 结

利用 CUDA 难以充分发挥 CPU-GPU 异构系统中的 GPU 硬件资源及其性能,本文将 CPU-GPU 异构系统视为异构 NUMA 结构,提出 CPU 辅助任务调度的基于线程池技术的 GPU 任务并行编程模型.在 CPU 端,基于 CPU 和 GPU 共享的页锁定内存区域,实现了任务槽和输入输出任务队列,进而由一个调度线程实现了 GPU 端的线程块级任务调度及其相应的接口函数.这种实现能够有效地降低 CPU 和 GPU 之间的任务调度开销,同时便于获知 GPU 端任务执行的状态,这种交互能力的提高更利于处理不规则并行应用.在 GPU 端,实现了在应用生命周期中持续运行的任务复用 kernel 函数机制,在一定程度上隐藏了底层 GPU 硬件特征,同时避免了使用多个 kernel 函数的启停开销以及所带来的 GPU 资源利用不充分等问题,这种细粒度任务并行方式提高了 GPU 端的任务调度效率及其负载均衡.通过经典线性代数和机器学习等应用实验表明,基于 CAGTP 模型能够更好地支持细粒度不规则以及混合任务等的并发执行,对提高 CPU-GPU 异构系统的资源利用率及计算性能具有重要作用.

参 考 文 献

[1] Wang Y, Davidson A, Pan Y, et al. Gunrock: A high-performance graph processing library on the GPU//Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, USA, 2016: 11

[2] Sparsh M, Jeffrey S V. A survey of CPU-GPU heterogeneous computing techniques. ACM Computing Surveys, 2015, 47(4): 69

[3] Alawneh S, Howell C, Richard M. Fast quadratic discriminant analysis using GPGPU for sea ice forecasting//Proceedings of the High Performance Computing and Communications (HPCC). New York, USA, 2015: 1585-1590

[4] Nvidia C. CUDA C Programming Guide 6.0, Nvidia Design Guide. Santa Clara, USA: Nvidia Corporation, 2014

[5] Li T, Wang D, Zhang S, Yang Y. Parallel rank coherence in networks for inferring disease phenotype and gene set associations. Advanced Computer Architecture (ACA 2014). Berlin, Germany: Springer, 2014: 163-176

[6] Li T, Liu X, Dong Q, et al. HPSVM: Heterogeneous parallel SVM with factorization based IPM algorithm on CPU-GPU cluster//Proceedings of the 2016 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2016). Heraklion, Greece, 2016: 74-81

[7] Pai S, Thazhuthaveetil M J, Govindarajan R. Improving GPGPU concurrency with elastic kernels. ACM SIGPLAN Notices, 2013, 48(4): 407-418

[8] Kothapalli K, Banerjee D S, Narayanan P J, et al. CPU and/or GPU: Revisiting the GPU vs. CPU myth. arXiv preprint arXiv:1303.2171, 2013

[9] Chakrabarti S, Demmel J, Yelick K. Modeling the benefits of mixed data and task parallelism//Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95). Santa Barbara, USA, 1995: 74-83

[10] Sarkar S, Mitra S, Srinivasan A. Reuse and refactoring of GPU kernels to design complex applications//Proceedings of the IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA 2012). Leganes, Spain, 2012: 134-141

[11] Li T, Narayana V K, El-Ghazawi T. A static task scheduling framework for independent tasks accelerated using a shared graphics processing unit//Proceedings of the 17th International Conference on Parallel and Distributed Systems (ICPADS). Tainan, China, 2011: 88-95

[12] Wende F, Cordes F, Steinke T. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering//Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing (SAAHPC 2012). Chicago, USA, 2012: 74-83

[13] Jiao Q, Lu M, Huynh H P, et al. Improving GPGPU energy-efficiency through concurrent kernel execution and

DVFS//Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. San Francisco, USA, 2015: 1-11

[14] Chen L, Villa O, Krishnamoorthy S, et al. Dynamic load balancing on single-and multi-GPU systems//Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2010). Atlanta, USA, 2010: 1-12

[15] Tzeng S, Lloyd B, Owens J D. A GPU task-parallel model with dependency resolution. Computer, 2012, 45(8): 34-41

[16] Chatterjee S, Grossman M, Sbîrlea A S, et al. Dynamic task parallelism with a GPU work-stealing runtime system// Proceedings of the Languages and Compilers for Parallel Computing (LCPC'13). Fort Collins, USA, 2013: 203-217

[17] Krieder S, Raicu I. GeMTC: GPU Enabled Many-Task Computing [Ph. D. dissertation]. Department of Computer Science, Illinois Institute of Technology, Chicago, USA, 2013

[18] Majumdar S, Jain I, Gawade A. Parallel quick sort using thread pool pattern. International Journal of Computer Applications, 2016, 136(7): 36-41

[19] Gupta K, Stuart J A, Owens J D. A study of persistent threads style GPU programming for GPGPU workloads// Proceedings of the Innovative Parallel Computing (InPar) 2012. San Jose, USA, 2012: 1-14

[20] Calhoun J, Jiang H. Preemption of a CUDA kernel function// Proceedings of the 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD 2012). Kyoto, Japan, 2012: 247-252

[21] Adriaens J T, Compton K, Kim N S, et al. The case for GPGPU spatial multitasking//Proceedings of the IEEE 18th International Symposium on High Performance Computer Architecture (HPCA 2012). New Orleans, USA, 2012: 1-12

[22] Robison A D. Intel Threading Building Blocks (TBB), Encyclopedia of Parallel Computing. New York, USA: Springer, 2011: 955-964

[23] Saule E, Çatalyürek Ü V. An early evaluation of the scalability of graph algorithms on the Intel MIC architecture// Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW 2012). Shanghai, China, 2012: 1629-1639

[24] Robison A D. Composable parallel patterns with Intel Cilk Plus. Computing in Science & Engineering, 2013, 15(2): 66-71, 87

[25] Jhurani C, Mullowney P. A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices. Journal of Parallel and Distributed Computing, 2015, 75(1): 133-140

[26] Chen Y, Davis T A, Hager W W, et al. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. ACM Transactions on Mathematical Software, 2008, 35(3): 22

[27] Chen Y, Luo T, Liu S, et al. Dadiannao: A machine-learning supercomputer//Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. Cambridge, UK, 2014: 609-622

[28] Huang X, Xiong L, Wang J, et al. Parallel weighting *K*-means clustering algorithm based on graphics processing unit. Journal of Information & Computational Science, 2015, 12(18): 7031-7040

[29] Zhang Shuai, Li Tao, Jiao Xiao-Fan, et al. Parallel TNN spectral clustering algorithm in CPU-GPU heterogeneous computing environment. Journal of Computer Research and Development, 2015, 52(11): 2555-2567(in Chinese)
(张帅, 李涛, 焦晓帆等. CPU-GPU 异构计算环境下的并行 T 近邻谱聚类算法. 计算机研究与发展, 2015, 52(11): 2555-2567)

[30] Zhang S, Li T, Jiao X, et al. HLanc: Heterogeneous parallel implementation of the implicitly restarted Lanczos method// Proceedings of the 3rd International Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications. Piscataway, USA, 2014: 403-410

[31] Chesney M, Scott L. Pricing European currency options: A comparison of the modified black-scholes model and a random variance model. Journal of Financial and Quantitative Analysis, 1989, 24(3): 267-284

[32] Zhong J, He B. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(6): 1522-1532

[33] Augonnet C, Thibault S, Namyst R, et al. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 2011, 23(2): 187-198

[34] Sun W, Ricci R. Augmenting operating systems with the GPU. arXiv preprint arXiv:1305.3345, 2013

[35] Peters H, Köper M, Luttenberger N. Efficiently using a CUDA-enabled GPU as shared resource//Proceedings of the IEEE 10th International Conference on Computer and Information Technology (CIT 2010). Bradford, UK, 2010: 1122-1127

[36] Chen T, Du Z, Sun N, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. ACM SIGPLAN Notices, 2014, 49(4): 269-284

[37] Gregg C, Dorn J, Hazelwood K M, et al. Fine-grained resource sharing for concurrent GPGPU kernels//Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism. Berkeley, USA, 2012: 389-398

[38] Wahib M, Maruyama N. Scalable kernel fusion for memory-bound GPU applications//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, Louisiana, 2014: 191-202

[39] Gelado I, Stone J E, Cabezas J, et al. An asymmetric distributed shared memory model for heterogeneous parallel systems. ACM SIGARCH Computer Architecture News, 2010, 38(1): 347-358

[40] Cabezas J, Vilanova L, Gelado I, et al. Automatic parallelization of kernels in shared-memory multi-GPU nodes//Proceedings of the 29th ACM on International Conference on Supercomputing. California, USA, 2015: 3-13

LI Tao, born in 1977, Ph. D., associate professor. His research interests include parallel and distributed processing, machine learning.

DONG Qian-Kun, born in 1990, M. S. His research interests include parallel and distributed processing, GPGPU computing.

ZHANG Shuai, born in 1986, Ph. D. His research interests include parallel and distributed processing, GPGPU computing.

KONG Ling-Yan, born in 1992, M. S. His research interests include parallel and distributed processing, machine learning.

KANG Hong, born in 1973, Ph. D., lecturer. His research interests include big data processing, database technology and machine learning.

YANG Yu-Lu, born in 1961, Ph. D., professor. His research interests include interconnection network, parallel computer architecture, and distributed computing.

Background

GPUs (Graphics Processing Units) are widely used as high performance computing devices for general purpose computing now. They offer more data and thread parallelism along with higher memory bandwidth, as compared to CPUs. A wide range of applications are well suited to this form of parallelism and achieve large speedups on a GPU. In GPU programming, programmers explicitly express parallelism based on application characteristic, and organize them in a hierarchy of threads, thread blocks and kernels. They divide the workloads to be performed among thread blocks at the programming model level, and not for optimal utilization of hardware resources. Especially with more irregular workloads are implemented on the GPU, limitations of the current GPU programming model become apparent.

Since the scheduling is controlled by GPU hardware, it is difficult to implement task parallelism on GPU. Take NVIDIA's GPU for example, different tasks are usually carried out using CUDA kernels. Fermi architecture supports concurrent kernel executions (CKE), however, it uses cooperative kernel scheduling. As long as one kernel has sufficient thread blocks to occupy all the Streaming Multiprocessors (SMs), it can take the entire GPU in spite of the seriously low resource utilization. Concurrent execution of two such kernels will mostly be degraded to sequential kernel execution. Current programming models do not support running kernels concurrently with fine-grained and dynamic control over resource allocation. Although kernels placed on different CUDA streams are potential candidates for being executed concurrently, there is no guarantee to be executed in the order they are invoked in the host program, so the data dependencies across kernels. Furthermore, even Fermi architecture does not allow multiple different applications to access GPU resource simultaneously. Indeed, most current computer systems offer a degree of heterogeneity and use multicore CPUs in the computation, as they evolving and offering powerful computational ability. But the heterogeneity of multicore CPU and GPU, both at the architectural level and programming level at the same time, raises the programming difficulties. The performance is strongly influenced by the dependence that exists between parallel code and heterogeneous architecture. Specifically, scheduling tasks to processors often requires considerable programmer effort on the CPU-GPU heterogeneous system.

In this paper, we present a CPU-assisted GPU thread pool (CAGTP) model based on the Fermi architecture supported page-locked host memory, in which computing blocks are used to execute fine-grained multi-tasks in parallel on multi-GPU systems. The computing blocks are active throughout the execution of the kernel, which is similar to the persistent kernel. However, the CAGTP model has better support to the CPU-GPU hybrid computing, especially for executing highly irregular applications. Comparing with CKE, concurrent task executions supported by the CAGTP model has higher task parallelism, better computation and data transfer overlap, and less task launch overheads. Furthermore, multi-tasks' launch overheads can be overlapped with each other. In order to dynamically combine task and data parallelism for executing applications, the computing block level task scheduler is designed and a simple programming interface is further implemented for creating dynamic tasks and handling load balancing.