计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20148354
53(6):1249-1262,2016

基于 CUPTI 接口的典型 GPU 程序负载特征分析

郑 祯 翟季冬 李 焱 陈文光
(清华大学计算机科学与技术系 北京 100084)
(z-zheng14@mails.tsinghua.edu.cn)

Workload Analysis for Typical GPU Programs Using CUPTI Interface

Zheng Zhen, Zhai Jidong, Li Yan, and Chen Wenguang
(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084)

Abstract GPU-based high performance computers have become an important trend in the area of high performance computing. However, developing efficient parallel programs on current GPU devices is very complex because of the complex memory hierarchy and thread hierarchy. To address this problem, we summarize five kinds of key metrics that reflect the performance of programs according to the hardware and software architecture. Then we design and implement a performance analysis tool based on underlying CUPTI interfaces provided by NVIDIA, which can collect key metrics automatically without modifying the source code. The tool can analyze the performance behaviors of GPU programs effectively with very little impact on the execution of programs. Finally, we analyze 17 programs in Rodinia benchmark, which is a famous benchmark for GPU programs, and a real application using our tool. By analyzing the value of key metrics, we find the performance bottlenecks of each program and map the bottlenecks back to source code. These analysis results can be used to guide the optimization of CUDA programs and GPU architecture. Result shows that most bottlenecks come from inefficient memory access, and include unreasonable global memory and shared memory access pattern, and low concurrency for these programs. We summarize the common reasons for typical performance bottlenecks and give some high-level suggestions for developing efficient GPU programs.

Key words graphics processing unit (GPU); workload analysis; Rodinia; performance counter; performance metric

摘 要 基于图形处理器(graphics processing unit, GPU)加速设备的高性能计算机已经成为目前高性能计算领域的一个重要发展趋势.然而,在当前的 GPU 设备上开发高效的并行程序仍然是一件非常复杂的事情.针对这一问题,1)总结了影响 GPU 程序性能的 5 类关键性能指标;2)采用 NVIDIA 公司提供的 CUPTI 底层接口,设计并实现了一套 GPU 程序性能分析工具集,该工具集可以有效地分析 GPU 程序的性能行为;3)采用该工具集对著名的 GPU 评测程序集 Rodinia 中的 17 个程序和一个真实应用程序进行了负载特征分析.总结出常见性能瓶颈的典型原因,并给出一些开发高效 GPU 程序的建议.

关键词 图形处理器;负载特征分析;Rodinia;硬件计数器;性能指标

中图法分类号 TP338.4

收稿日期:2014-12-10;**修回日期**:2015-08-11

基金项目:国家自然科学基金项目(61103021);国家"八六三"高技术研究发展计划基金项目(2012AA010901)
This work was supported by the National Natural Science Foundation of China (61103021) and the National High Technology Research and Development Program of China (863 Program) (2012AA010901).

近年来,基于加速设备的超级计算机已经成为高性能计算领域一个重要发展趋势.在2014年6月最新发布的超级计算机TOP500排名中,有48个计算机系统使用了GPU加速器.其中,在前15位的超级计算机中有5个计算机使用了GPU加速器.目前,NVIDIA公司的GPU是最主流的GPU加速器,在上述TOP500中有48个使用GPU加速器的超级计算机中,有46个使用的是NVIDIA架构的GPU,其余2个使用了AMD的GPU.

尽管基于加速设备的超级计算机变得越来越流行,但是,在这些系统上编写高效率的并行程序仍然是一件非常复杂的事情.我们以NVIDIA的GPU设备为例,NVIDIA公司推出的通用编程框架CUDA(compute unified device architecture)是目前应用最为广泛的编程模型,但编写高效的CUDA程序非常困难.一方面,GPU设备的存储器层次结构较为复杂,除了片外的全局存储器(global memory)、常数存储器(constant memory)和纹理存储器(texture memory)之外,GPU还向应用开发人员开放了对片内的共享存储器(shared memory)的访问接口,这些存储器都需要程序员在程序中显示地进行管理;另一方面,CUDA编程模型本身也提供多层次的程序组织方式,具有网格(grid)、线程块(thread block)、warp等,不同组织形式下有不同的线程间通信机制和存储器共享与访问机制.这些复杂的编程机制导致GPU程序的编写及优化比较困难.

针对GPU程序性能的瓶颈,目前国内外研究人员提出了很多相关的优化技术.Zhang等人[1]提出了一种软件的方法在运行时优化GPU程序中控制和访存的不规则问题,但是目前该工作只支持简单的访存行为优化,尚不支持较复杂的程序.Yang等人[2]提出一个基于指导编译语句的方式优化GPU程序中的嵌套并行.Margiolas等人[3]开发了一个可移植的框架优化OpenCL程序中CPU和GPU之间通信传输开销高的问题.Xiang等人[4]发现在GPU程序中不同warp间会存在显著的负载不均衡现象,由于目前GPU的warp调度方式会导致GPU资源的显著浪费,针对上述问题他们提出一种新的资源管理方式,可以有效提高GPU程序性能并降低系统的整体能耗.尽管已有大量的研究工作试图自动优化GPU程序的各种性能行为,但是,缺少对典型GPU程序的负载特征的详细分析和总结.

为了有效分析当前典型GPU程序的负载特征,本文首先归纳出GPU程序若干重要性能指标,其中包括计算访存比、实际浮点性能、带宽利用率、GPU占用率、共享存储器访问效率、全局存储器访问效率等.并且,为了有效获取GPU程序的上述指标,我们采用NVIDIA公司提供的底层CUPTI接口(CUDA profiling tools interface)自主开发了一套性能分析工具,该工具可以在不修改用户源码的情况下精确获取GPU程序上述性能指标.本文采用该工具集对著名的GPU评测程序集Rodinia[5]中17个程序和全球正压大气浅水波模式应用程序[6]进行了性能分析,统计了各个程序上述的性能指标,并对部分重要性能指标表现的性能瓶颈进行了程序优化.本文的分析结果对CUDA程序优化及未来GPU体系结构的设计都具有指导意义.

1 相关工作

目前已有一些工具可以进行CUDA程序性能分析,但都存在一些不足.PAPI(performance application programming interface)[7]是应用广泛的程序性能分析工具,提供了针对CUDA程序的性能分析接口.使用PAPI接口在待测试程序代码中进行插装,就可以得到CUDA程序的部分硬件计数器值.但如前所述,使用PAPI进行性能测试时需要对源代码进行修改,这增加了性能测试工作的难度和繁琐程度;另外,PAPI性能分析接口每次只能够读取少量的硬件计数器值,为得到大量硬件计数器值,需要多次修改插桩代码并多次运行.TAU(tuning and analysis utilities)[8]提供了功能更为强大的CUDA程序性能分析功能.该工具可以追踪CUDA程序各种库函数操作的执行时间和具体执行信息(如复制函数复制的字节数等),并能够在不修改待测试程序源码的情况下收集程序运行时的硬件计数器指标值.但在CUDA程序的1次运行中,TAU只能够收集少量的硬件计数器指标值,为了收集较多的指标值,需要手动地多次执行整个程序,使用较为繁琐;另外,TAU只能收集单纯的硬件计数器值,不能收集或计算综合性能指标,而独立的硬件计数器值难以直接反映程序的性能特性,这使得其性能分析结果难以引导开发者进行性能优化.Visual Profiler[9]是NVIDIA官方提供的CUDA性能分析工具.该工具提供可视化的操作界面,能够收集GPU上所有的硬件计数器值,且能够收集并计算一些综合性能指标.但该工具对部分重要性能指标默认不做收集,而需要开发者进行指标的选择,初级

CUDA 程序开发者难以选择出重要的性能指标并通过查看众多性能指标而快速确定程序性能瓶颈.

目前已有一些 CUDA 程序优化方法的研究. Yang 等人[10]的工作中通过对 10 个开源 CUDA 程序的分析,总结了若干种导致程序性能低下的程序模式,并通过实验证明对这些程序模式的优化能够显著提高程序性能. Stratton 等人[11]的工作中描述了对 CUDA 程序优化有关键作用的方法,包括数据存储方式的转换、合并访问、负载均衡化、控制并行粒度等方法,这些优化模式应用在 CUDA 程序上有普遍的性能提高. Yang 等人[12]的工作中提出了一个源到源的 CUDA 程序自动优化工具,其中采用的优化方法包括向量化,全局内存合并访问,线程合并以及数据预取等方法. 在 Li 等人[13]以及 Chen 等人[14]的工作中,研究了变量存储位置转换的问题. Li 等人[13]的工作主要涉及到全局内存和共享内存变量转换为寄存器变量,以及共享内存变量转换为全局内存变量这几个方面;Chen 等人[14]的工作则侧重于将全局内存变量转变为其他类型变量,结果显示通过内存变量位置的合理转换,程序性能能够得到很大的提升. Wahib 等人[15]的工作中,提出了通过合并核函数的方式进行数据重用的方法,被合并的核函数需要有共同的参数,此时这些参数的访问能够被重用,实验显示该方法对部分程序有很好的优化效果. 李建江等人[16]提出了一种将单 GPU 程序转化为多 GPU 程序的方法,通过该转化能够使大型程序运行在多 GPU 上,取得更好的加速效果. 这些针对 CUDA 程序的研究都取得了一定的效果,但这些研究都侧重于以具体的方法对某一类程序进行优化,并没有对普通程序进行性能瓶颈分析和定位.

对常用 CUDA 程序做负载分析的相关工作较少. Che 等人[5]的工作中,对基准测试程序套件 Rodinia 进行了多样性分析,说明了 Rodinia 中程序能够较为广泛地覆盖并行程序的特性,并分析了影响 CUDA 程序性能的一些问题,但该工作中并没有研究反映 CUDA 程序性能特性的性能指标,也没有针对 CUDA 重要性能特性对 Rodinia 做负载特征分析;同时,该工作中只是对一套基准测试程序套件进行分析,没有对真实应用程序进行性能分析.

目前已有一些针对 CUDA 程序的部分特性进行性能建模的工作. 在 Williams 等人[17]提出的 Roofline 模型中,分析了实测浮点性能与理论浮点性能峰值、带宽峰值及操作密度的函数关系. 该模型中,操作密度是指单位字节的内存数据传输之下的操作数量,为操作数量与访存数量的比值,对于给定的操作密度,通过该模型可以确定程序是计算受限还是访存受限. 本论文中借鉴了该工作的方法,确立了计算访存行为的性能指标.

2　CUDA 程序性能指标

为了有效地分析 CUDA 程序的负载特征,明确程序的性能瓶颈,本文提出 5 类关键性能指标描述 CUDA 程序的性能行为:

1) 计算访存行为(计算访存比、实测浮点性能和带宽利用率)

计算访存比对于确定程序的类型以及检测程序性能受限因素有重要作用:①对基准测试程序进行计算访存比测试可以统计出通用程序的计算访存特征;②对于单个程序,综合计算访存比、GPU 带宽与带宽利用率以及理论浮点性能峰值与实测浮点性能可以看出程序是访存受限还是计算受限[17]. 科学计算中,大部分计算指令为浮点型指令,而通常影响程序性能的访存方式主要是全局存储器访问,因此,统计浮点计算指令与全局存储器访存比更有意义.

我们定义浮点计算与全局访存比为浮点指令的条数与全局存储器访问字节数的比率,记为 $R_{\text{f-g}}$,记浮点指令条数为 I_{f},核函数执行时间为 D_{k},单位时间内全局存储器的读与写的字节数分别为 T_{gld} 与 T_{gst},则有:

$$R_{\text{f-g}}=\frac{I_{\text{f}}}{(T_{\text{gld}}+T_{\text{gst}})\times D_{\text{k}}}. \tag{1}$$

由于我们收集的 T_{gld} 与 T_{gst} 为运行时的指标,最终得到的全局存储器访问字节数也是实际值,而非理论值在作分析时,理论的存储器访问量与实际值可能会有偏差.

GPU 实测浮点性能 $P_{\text{f-achieved}}$ 是指 GPU 每秒实际执行的浮点运算次数,浮点计算单元利用率 U_{f} 是指实测浮点性能与理论峰值 $P_{\text{f-peak}}$ 的比率. 这 2 个指标能够反映出程序的浮点计算量及浮点计算单元的利用情况,其计算方法为

$$P_{\text{f-achieved}}=\frac{I_{\text{f}}}{D_{\text{k}}}. \tag{2}$$

$$U_{\text{f}}=\frac{P_{\text{f-achieved}}}{P_{\text{f-peak}}}. \tag{3}$$

GPU 带宽利用率 U_{dram} 可以通过指标 T_{dramread},$T_{\text{dramwrite}}$ 以及 GPU 的带宽 B 求得,其中 T_{dramread} 和 $T_{\text{dramwrite}}$ 分别表示单位时间内 GPU 设备上内存读写

字节数.U_{dram}能够综合地反映程序对存储器访问的效率,其计算方法为

$$U_{dram}=\frac{T_{dramread}+T_{dramwrite}}{B}. \tag{4}$$

2) GPU 占用率

GPU 占用率(occupancy)是指实际运行的 warp 占最大理论 warp 数量的比值,包括理论 GPU 占用率和实际 GPU 占用率.影响理论 GPU 占用率的指标有 3 个:每个线程块中线程的数量、每个线程占用的寄存器数量和每个线程块分配的共享存储大小.实际 GPU 占用率由 CUPTI 中定义的 *achieved_occupancy* 来表示,该值受到每个网格中线程块数量及实际运行环境的影响,需要在程序运行时通过 CUPTI 相关函数获取.

GPU 占用率反映了程序实际开启的并行计算线程数量占理论最大数量的比例,一般来说,比值越大表示程序对 GPU 资源的利用率越高,并行度越好.一方面,GPU 占用率受开发者配置的并行线程数量的影响,开发者配置的线程数过少时 GPU 占用率会较低;另一方面,GPU 占用率受单位线程占用的存储器资源数量的影响,当每个线程中局部变量较多从而占用较多寄存器时,或者每个线程块占用的共享内存数量较多时,会因为存储器资源不足而无法发起太多线程,导致 GPU 占用率较低.GPU 占用率可以反映程序在这 2 方面的特性.

对于访存受限的 CUDA 程序,当其并行度较高时,大量的线程并行执行通常能够有效地掩盖数据传输带来的时延,从而提高程序性能.因此 GPU 占用率是反映访存受限的 CUDA 程序性能的重要指标.但对于计算密集型程序,单纯提高 GPU 占用率并不能很有效地提高程序性能.

3) 共享存储器访问效率

反映共享存储器访问效率的指标是 *shared_efficiency*,表示对共享存储器数据访问的请求数据量与实际数据量的比率,其值越大表示共享存储器访问效率越高.

在共享内存的最佳访问状态下,每个 half-warp 中不同线程访问到不同的 bank 中(或所有线程访问同一地址),此时各个线程能够同时访问共享内存,*shared_efficiency*=1.0;当存在 bank 冲突时,不同线程访问到同一个 bank 时会使得各线程只能依次访问,此时 *shared_efficiency* 降低.因此,*shared_efficiency* 能够反映出共享存储器使用中的 bank 冲突(bank conflict),当该指标值较低时,往往表示存在 bank 冲突.

4) 全局存储器访问效率

反映全局存储器访问效率的指标是 *gld_efficiency* 和 *gst_efficiency*.*gld_efficiency* 的含义是对全局存储器数据读取的请求数据量与实际数据量的比率,其值越大表示读取数值的效率越高,该比率可能小于 1.0,可能等于 1.0,也可能大于 1.0.*gst_efficiency* 表示对全局存储器数据写入的请求数据量与实际数据量的比率,与 *gld_efficiency* 的意义类似,值越大表示效率越高.

全局内存的最佳读取方式是多个线程读取同一地址,此时多个线程只需一次全局内存读取,*gld_efficiency*>1.0;当同一个 half-warp 中的线程对全局内存的读取地址连续且满足一定的地址对齐规则时(不同 GPU 架构下对齐要求不同),该 half-warp 中各线程对全局内存的读取能够合并为同一条读取指令,此时 *gld_efficiency*=1.0;当 half-warp 中不同线程对全局内存变量的访问不连续或不满足地址对齐规则时,不同线程需多次读取全局内存,而 CUDA 中对全局内存读取时是以 32 B,64 B 或 128 B 为单位的,单次读取单个变量时会读入大量不需要的内存,*gld_efficiency*<1.0.对于全局内存的写入和 *gst_efficiency* 有类似的结论.

gld_efficiency 和 *gst_efficiency* 可以有效地检测出非合并访问(uncoalesced access),对于测试 CUDA 程序全局存储器访问效率很有效.

5) 线程束执行效率

线程束执行效率主要由线程分歧的情况来反映.对于 CUDA 核函数,当分支条件控制指令数不大于某个阈值时,编译器会使用带有谓词的指令替换分支指令,各个线程会根据控制条件将各条替换过的分支指令与设置为 true 或 false 的谓词相关联,此时每一条分支指令都会在任何一个线程中发射,但最终只有谓词为 true 的指令才会被实际执行;当分支数大于阈值时,编译器不会进行分支谓词替换,此时同一 half-warp 中执行不同分支的线程无法并行执行,会出现一部分线程等待其他线程的情况.

能够反映核函数执行过程中线程分歧的指标是 *warp_execution_efficiency* 和 *warp_nonpred_execution_efficiency*.*warp_execution_efficiency* 是指在流处理器中各个 warp 中实际运行的线程数量占最大理论线程数量的比值的平均值,该值为 1.0 时表示不考虑分支谓词替换分支指令的情况下

没有线程分歧,否则一定有线程分歧. *warp_nonpred_execution_efficiency* 是指在一个流处理器上各个 warp 中执行非分支预测指令的线程数量占最大理论线程数量的比值的平均值,该值为 1.0 时表示没有使用分支谓词替换分支指令,否则存在分支预测.综合 2 个指标可以全面地了解核函数中的线程分歧情况.

通过上述指标的分析,可以清楚地了解 CUDA 程序的计算特征、访存特征以及 GPU 特有的性能行为,例如,线程分歧和共享内存使用情况.下面介绍本文如何通过 GPU 的 CUPTI 接口获取上述性能指标.

3　性能工具的实现

本文设计并实现的性能分析工具能够在待测试程序运行时收集第 2 节讨论的重要性能指标,使用该工具时无需修改待测试程序源码.为实现该工具,我们对 CUDA 运行时库函数进行了插桩,重写了部分关键的库函数,并通过设置 LD_PRELOAD 环境变量使得 CUDA 程序在调用库函数时链接并执行我们重新实现的版本,进而在执行时运行采集性能指标的程序段.对库函数插桩前后核函数执行流程如图 1 所示:

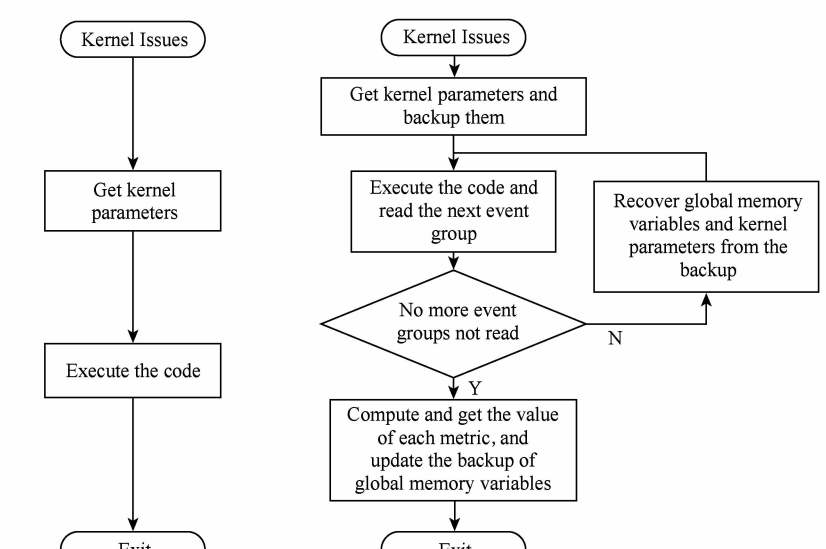


Fig. 1　The processes of the kernel before and after instrumentation.

图 1　对库函数插桩前后核函数执行流程

我们在插入的代码中使用 CUPTI 接口[18]来收集 CUDA 程序的性能指标,CUPTI 可以收集的指标主要包括程序运行信息、事件和 CUPTI 定义的 Metric 三类.其中,事件指 GPU 的硬件计数器值,而 Metric 则是由若干个事件值计算出的综合性指标.在第 3 节中描述的重要性能指标大部分可以直接由 Metric 来表示,另外一部分可以通过其他 Metric 求出,因此我们的目标是收集需要的事件值,进而得到重要的 Metric 值.

为得到需要的重要 Metric 值而必须首先得到较多的事件值,而由于硬件计数器数量限制,每次核函数运行时可以收集到的事件值数量有限,因此需要多次执行核函数,分多次收集事件值.为此,我们通过对库函数插桩而改变了核函数的发射过程,使得原先一次执行的核函数被多次发射,从而在每次执行时求得一部分硬件计数器值,最终求出所有需要的性能指标值.通过这样的方式,核函数被多次执行,而整个程序则仍然只需要执行一次,且整个过程对使用者来说是透明的.核函数发射的控制函数主要是 cudaConfigureCall, cudaSetupArgument, cudaLaunch,前两者将核函数配置参数与运行参数压入栈中,后者发射核函数,编译时源程序中核函数调用语句会被这 3 个函数所替代.本文实现的性能分析工具重新实现了这 3 个函数,在前两者中备份了函数参数,在新的 cudaLaunch 中多次调用原函数 cudaLaunch,从而实现核函数的多次执行,每次调用后收集若干事件值,在收集到所有所需事件值后得到所需的各个 Metric 值.为了保证多次调用核函数后程序的

正确性,需要在每次调用 cudaLaunch 前恢复部分变量值为进入核函数时的初始值,并重新读取备份的函数参数.

通过修改库函数将原先核函数的每次执行变为多次执行后,如不恢复变量值为进入核函数时的初始值,则可能导致程序的错误,因为核函数执行过程中可能修改部分全局内存变量的值,此时原先只执行一次的核函数被修改为多次执行后会导致部分全局内存变量被累计修改多次,从而导致错误.为了避免这样的错误,需要提前备份可能被修改的变量的值,并在每次重复调用 cudaLaunch 前将上次执行时被改变的变量恢复为正确的初始值,从而使核函数的多次执行是在同样的初始条件之下进行,在最后一次重复调用 cudaLaunch 之后更新变量备份值.需要备份的变量为作用域大于核函数作用域且在核函数中可以被改变的变量,满足此条件的变量只有全局存储器变量,为了保证所有可能被改变的全局存储器变量都被备份,我们备份了程序中所有的全局存储器变量,并通过对相应库函数进行插桩而实现了备份值跟随原变量值的同步更新.为了实现变量的备份与同步更新,性能工具实现时维护了一个链表,链表中每个节点对应于一个全局内存变量,节点保存了被备份变量的内存地址及备份值.我们重新实现了 cudaMalloc 等库函数,每当通过这些函数新创建一个全局内存变量时在链表中增加 1 个节点,从而保证备份所有的全局内存变量,同时,我们重新实现了 cudaMemcpy 等函数,每当通过这些函数改变内存变量值时,根据内存变量地址找到对应的备份节点,同步更新节点的备份值,从而实现备份值的同步更新.在每次重复调用 cudaLaunch 前恢复全部全局内存变量值的方法为:遍历备份链表,将各节点指向的内存地址的内存值恢复为节点存储的备份值;最后一次重复调用 cudaLaunch 后,则以类似方法将链表中各个节点的备份值更新为对应变量的当前值.

最后,我们对该工具的有效性与正确性进行了验证.该工具收集的部分指标与 Visual Profiler 一致,对于该部分指标,我们将该工具与 Visual Profiler 的输出结果进行了对比,结果显示两者没有明显差异;对于在 Visual Profiler 中没有的指标,其中间结果都可以由 Visual Profiler 进行收集,我们验证了这些指标的中间结果的正确性,最后通过对 CUDA 官方示例中部分程序的源码及运行结果进行分析而验证了这些指标的有效性与正确性.

4 CUDA 程序负载特征分析

本文对著名的 CUDA 评测程序 Rodinia 中 17 个程序和全球正压大气浅水波方程求解应用程序采用上述性能指标进行分析.本文的实验平台为 Dell Precision T5600 服务器,CPU 型号为 Intel Xeon E5-2620,内存大小为 64 GB,操作系统为 Red Hat Enterprise Linux 6.4 Server,GPU 型号为 NVIDIA Tesla K20c,该 GPU 的 CUDA 计算核心数量为 2 496 个,共享内存大小为 4 800 MB.实验中使用的 CUDA 版本为 CUDA 6.0.

Rodinia 中程序能够较为广泛地覆盖并行程序的特性,其 CUDA 版本覆盖了 CUDA 程序的多方面特性,因此其成为了对 CUDA 程序进行测试分析的重要程序集.本文分析的 Rodinia 中 17 个程序如表 1 所示:

Table 1 Programs in Rodinia

表 1 Rodinia 中程序简介

Programs (Abbreviation)	Dwarves	Domains
b+tree(B+T)	Graph Traversal	Search
backprop(BP)	Unstructured Grid	Pattern Recognition
cfd(CFD)	Unstructured Grid	Fluid Dynamics
Gaussian(GS)	Dense Linear Algebra	Linear Algebra
hotspot(HS)	Structured Grid	Physics Simulation
kmeans(KM)	Dense Linear Algebra	Data Mining
lavaMD(LV)	N-Body	Molecular Dynamics
lud(LUD)	Dense Linear Algebra	Linear Algebra
mummergpu(MM)	Graph Traversal	Bioinformatics
myocyte(MC)	Structured Grid	Biological Simulation
nn(NN)	Dense Linear Algebra	Data Mining
nw(NW)	Dynamic Programming	Bioinformatics
particlefilter(PT)	Structured Grid	Medical Imaging
pathfinder(PF)	Dynamic Programming	Grid Traversal
srad_v1(SR1)	Structured Grid	Image Processing
srad_v2(SR2)	Structured Grid	Image Processing
streamcluster(ST)	Dense Linear Algebra	Data Mining

4.1 Rodinia 负载分析

1) 计算访存分析结果

Rodinia 中 17 个程序的浮点计算与全局内存访问比如图 2 所示,可以看出,37 个核函数中有 9 个函数该指标值大于 1,其中有 4 个函数该指标值大

于 2,有 2 个函数该指标值大于 3,分别是 LV 程序的核函数(7.97)和 KM 的函数 kmeansPoint(86.25).部分程序的浮点计算访存比为 0,该类程序中没有浮点计算.

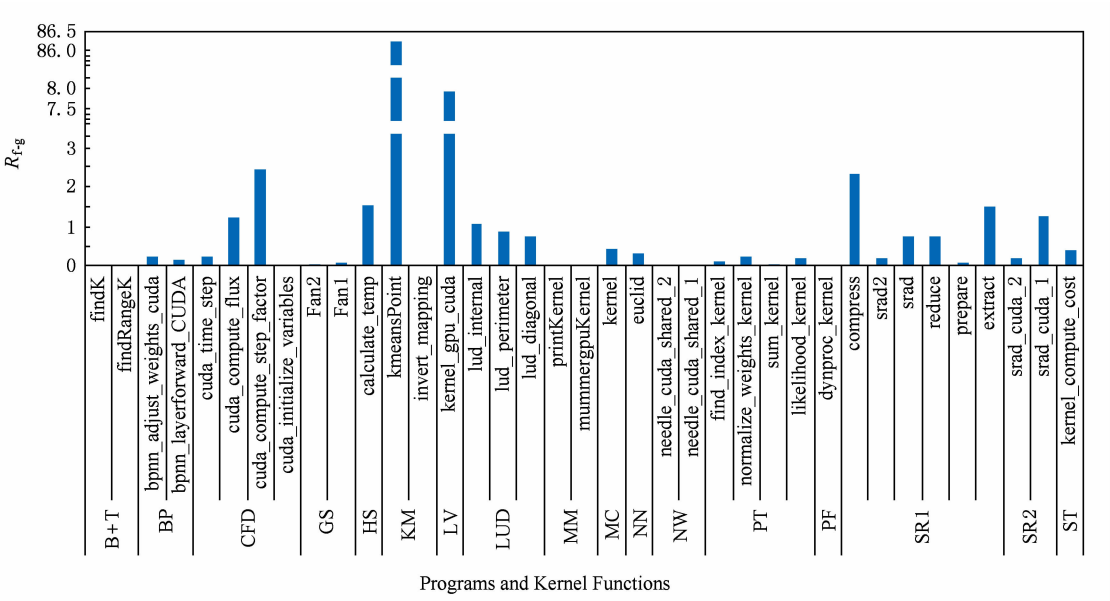


Fig. 2　The $R_{f\text{-}g}$ values of the 17 programs in Rodinia.

图 2　Rodinia 中 17 个程序浮点计算与全局内存访问比

本文中分析的应用程序为求解全球正压大气浅水波方程的程序,该方程为大气模拟过程中最基础、最重要的方程.该程序支持多种混合架构,我们选取其单 GPU 版本进行了性能分析.

17 个程序的浮点计算单元利用率如图 3 所示,可以看出,这 17 个程序的浮点计算单元利用率普遍很低,37 个核函数中有 36 个的该指标值小于 0.04,且其中 28 个核函数该值小于 0.01;LV 程序的核函数该指标值最高,但也只达到 0.177.这说明 17 个程序对浮点计算单元的利用率普遍较低.

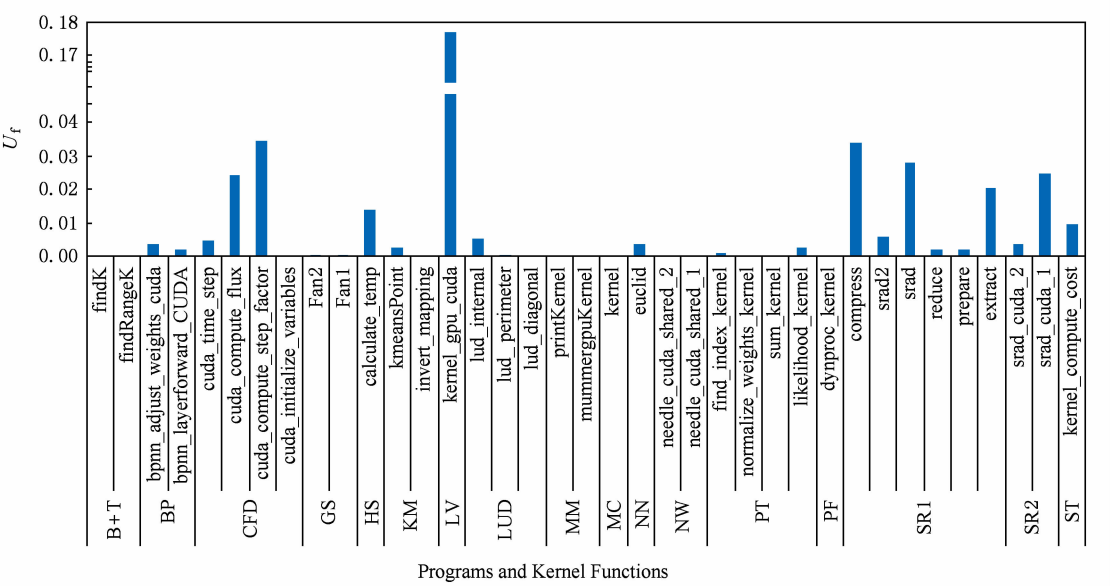


Fig. 3　The U_f values of the 17 programs in Rodinia.

图 3　Rodinia 中 17 个程序的浮点计算单元利用率

17 个程序的带宽利用率 U_{dram} 指标值如图 4 所示,可以看到,在 37 个核函数中有 28 个核函数的带宽利用率小于 0.5;CFD 的函数 cuda_time_step 的该值最高,为 0.7267.程序带宽利用率整体不够高的原因可能有 2 个:①程序本身的特性,部分程序由于算法本身的特性及程序规模等原因,使得其即使

访存方式达到最优也无法太多地利用带宽;②程序本身访存方式设计不够合理,导致带宽利用率较低.

另外,与浮点计算单元利用率相比,程序的带宽利用率整体高于前者.

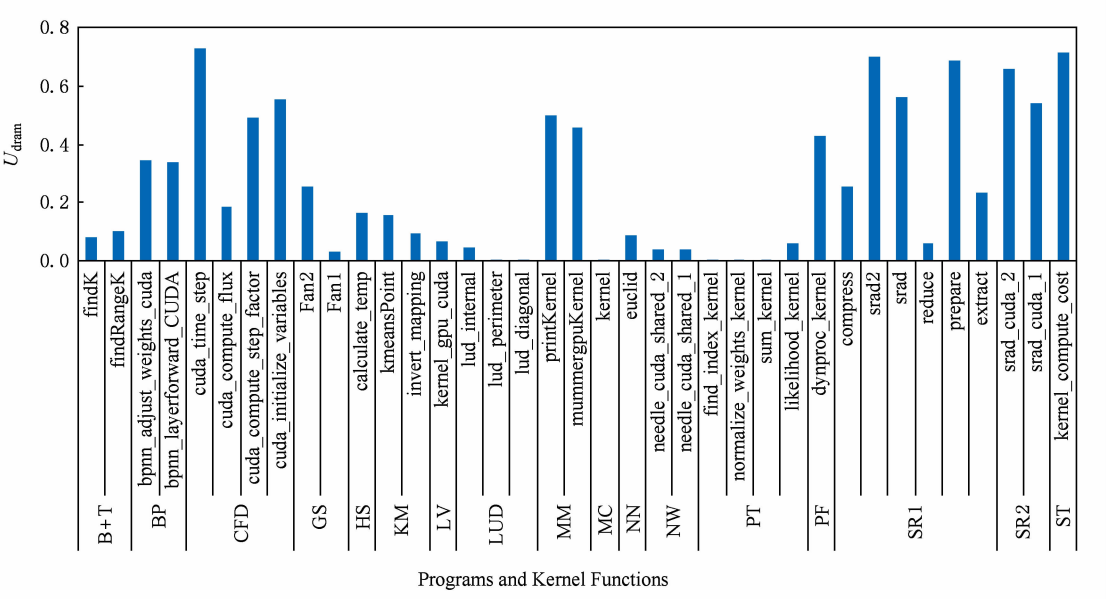


Fig. 4 The U_{dram} values of the 17 programs in Rodinia.

图 4 Rodinia 中 17 个程序的带宽利用率指标值

从以上分析可知,Rodinia 中程序的浮点计算单元利用率普遍较低,程序带宽利用率普遍优于浮点计算单元利用率.同时,多数程序的浮点计算与全局内存访问比较低,说明程序的全局内存访问量相对浮点运算而言更多,这与带宽利用率相对略高而浮点计算单元利用率低的趋势是一致的.由此可见,程序的带宽利用率与浮点计算单元利用率一定程度上受程序本身计算访存数量特征的影响.

2) occupancy 分析结果

Rodinia 中 17 个程序的 occupancy 如图 5 所示,在 37 个核函数中有 22 个达到了 0.7,其他 15 个核函数的 occupancy 值较小,其中 GS 程序的函数 Fan2、LUD 程序的函数 lud_perimeter 与 lud_diagonal、MC 程序的核函数 kernel 值最低,都为 0.2,其理论 occupancy 值也较低,为 0.25.

对于 GS 程序的函数 Fan2,在输入为 1 024×

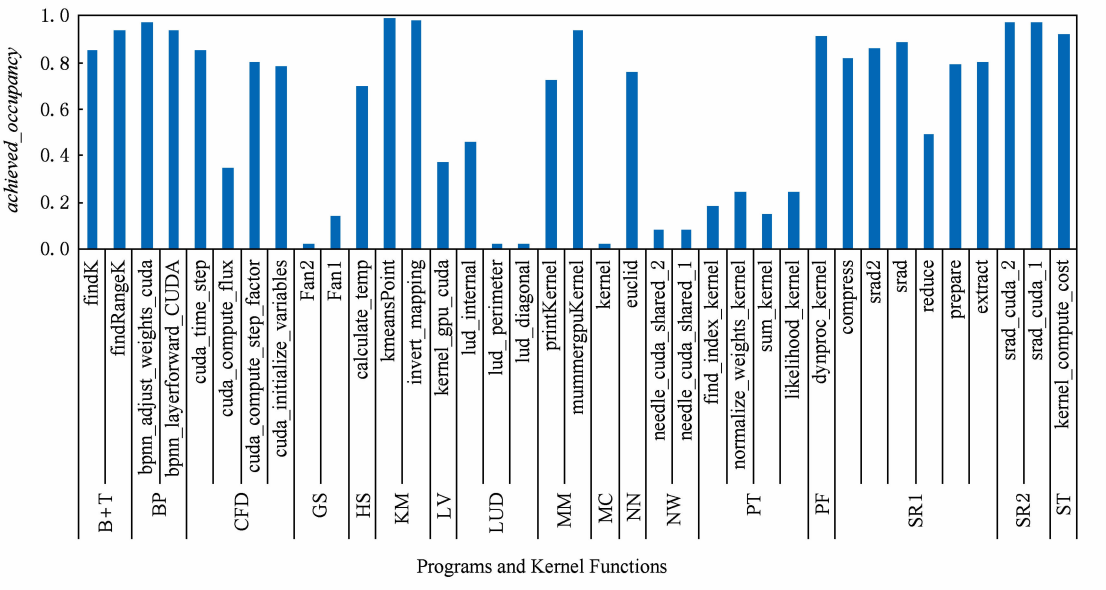


Fig. 5 The *achieved_occupancy* values of the 17 programs in Rodinia.

图 5 Rodinia 中 17 个程序各核函数的实际 occupancy 值

1 024 的矩阵的情况下,程序网格的维数为(256,256,1),线程块维数为(4,4,1),由此可见,线程块维数较小最可能是导致 occupancy 值较低的原因. 修改代码,将线程块维数设置为(16,16,1),测试结果显示实际 occupancy 值提升为 0.86.

修改程序前后测试信息如表 2 所示,可以看出增大线程块维数进而增大 occupancy 值能够有效地减少核函数执行时间(减小了 0.56). 同时,在采用此方法增大 occupancy 之后,程序的全局存储器访问效率有所降低(*gld_efficiency* 与 *gst_efficiency* 值减小),这说明在增大 occupancy 之后,程序中同时运行的 warp 更多,程序并行性更高,高并行性使得访存时延被大量的计算指令的执行有效地掩盖了.

Table 2　Optimization Result of Fan2 Function in GS Program

表 2　对 GS 程序的函数 Fan2 优化结果

Program	Grid Dimension (*X*,*Y*,*Z*)	Block Dimension (*X*,*Y*,*Z*)	Theory Occupancy	Achieved Occupancy	Kernel Execution Time/s
The Original Program	(256,256,1)	(4,4,1)	0.25	0.22	0.307 773
The Optimized Program	(64,64,1)	(16,16,1)	1.00	0.86	0.166 333

3) shared_efficiency 分析结果

Rodinia 中 17 个程序的 *shared_efficiency* 值如图 6 所示,在 37 个核函数中有 14 个使用了共享存储器,在这 14 个函数中有 8 个程序的 *shared_efficiency* 指标值达到了 1.0,2 个程序的 *shared_efficiency* 指标值达到了 0.8,其他 4 个函数的 *shared_efficiency* 指标值较小,其中 LV 程序的函数 kernel_gpu_cuda 有最低的利用率,为 0.4211.

图 7 显示了 LV 程序的函数 kernel_gpu_cuda 在使用共享内存时存在 bank 冲突. 该函数中定义了 2 个共享内存数组 *rA_shared* 和 *rB_shared*,数组元素都是结构体,大小为 32 B. 在计算能力为 3.5 的 GPU 中,共享存储器被分为 32 个 bank,该程序运行时 bank 的存储单元大小为 4 B,则每访问连续的 32 个 4 B 后会回到原先访问的 bank 中. 由此可知,该程序中同一个 warp 中对 2 个共享内存数组的第 n 个元素的访问与第 $n+4$ 个元素的访问会产生 bank 冲突. 对于数组 *rA_shared*,warp 中不同的线程之间会产生大量的 bank 冲突;对于数组 *rB_shared*,由在循环中访问的过程中,half-warp 的不同线程可能访问到相同的 bank 中,从而造成 bank conflict 冲突.

4) 全局存储器访问效率分析结果

Rodinia 中 17 个程序的 *gld_efficiency* 与 *gst_efficiency* 值如图 8 所示. 对于 *gld_efficiency*,MM 程序的函数 mummergpuKernel 有最低值 0.032,

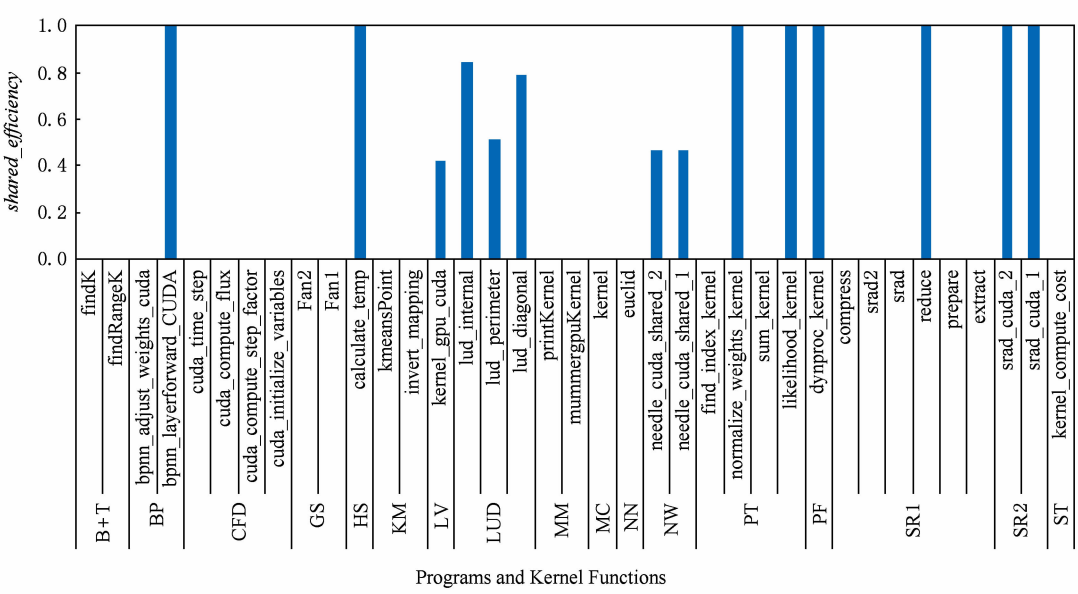


Fig. 6　The *shared_efficiency* values of the 17 programs in Rodinia (0 means no use of shared memory).

图 6　Rodinia 中 17 个程序各核函数的 *shared_efficiency* 值(值为 0 表示没有使用共享内存)

PT 程序的函数 find_index_kernel 有最高值 7.3438;对于 *gst_efficiency*,MM 程序的函数 printKernel 有最低值 0.0938,CFD 的函数 cuda_time_step 及其他程序的部分函数有最高值 1.0.

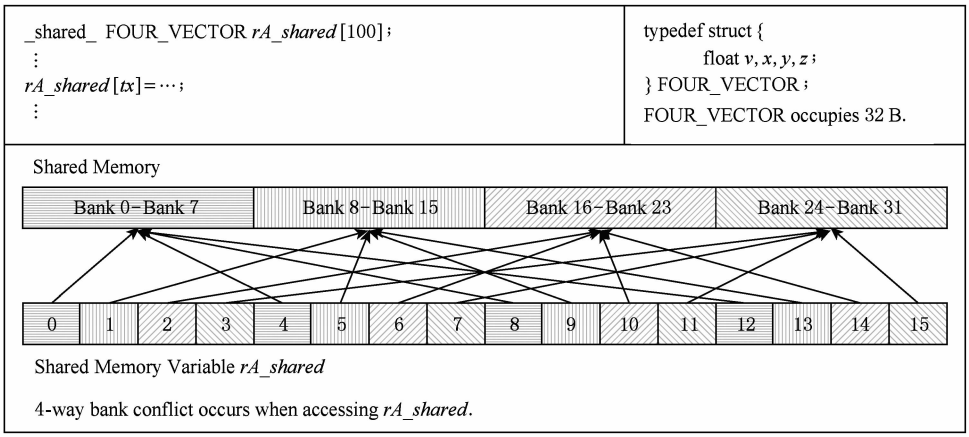


Fig. 7 The bank conflict for function kernel_gpu_cuda in program LV accessing *rA_shared*.

图 7 LV 程序函数 kernel_gpu_cuda 访问变量 *rA_shared* 时的 bank 冲突

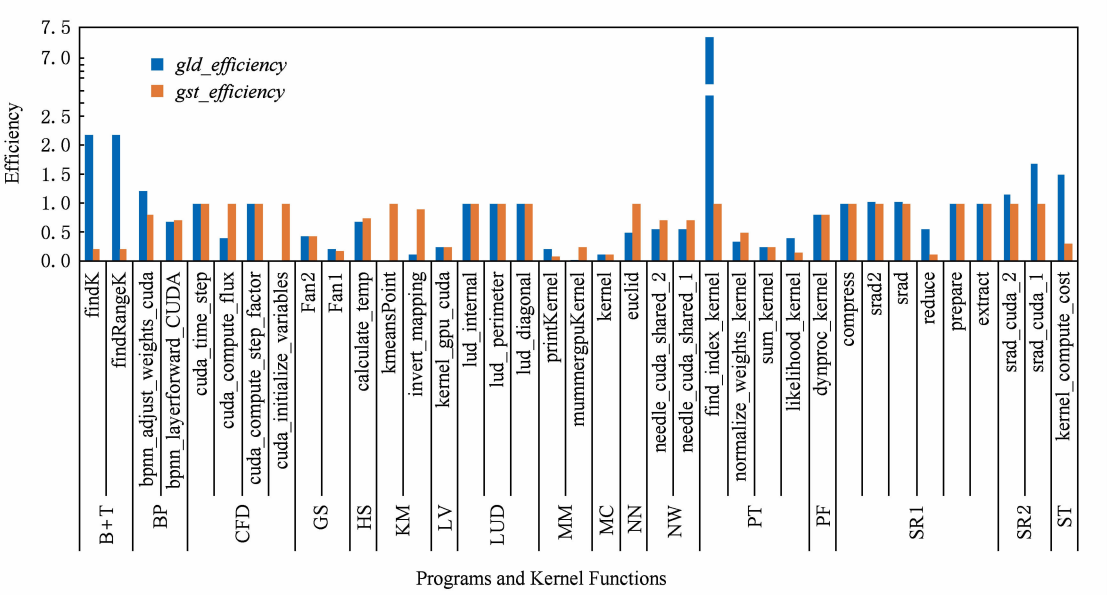


Fig. 8 The *gld_efficiency* and *gst_efficiency* values of the 17 programs in Rodinia.

图 8 Rodinia 中 17 个程序各核函数的全局存储器访问效率

如图 9 所示,MM 程序的函数 mummergpuKernel 中对全局内存的访问存在非合并访问的现象.在同一个线程中,对全局内存数组多次以比较随机的索引值进行访问,warp 中的不同线程对该数组的访问的位置也很难连续,因此无法合并访问.最终该函数的 *gld_efficiency*=0.032.

对于 PT 程序的函数 find_index_kernel,其对全局内存块 *arrayX* 和 *arrayY* 的访问一般是连续多次访问同一个地址,这种情况下 warp 在执行访存指令时可以只进行一次访存,访存效率较高,*gld_efficiency*=7.3438.

图 10 为 MM 程序的函数 printKernel 中部分代码结构,该函数中存在大量对结构体类型的全局内存数组元素的成员的访问,访问的地址难以满足合并访问所要求的对齐规则,且 warp 中各个线程访问的地址可能不连续,使得出现大量非连续访问,*gst_efficiency* 值较低,*gst_efficiency*=0.0938.

在 CFD 的 cuda_time_step 等函数及其他程序的部分函数中,对同一 warp 中对全局内存的访问是连续的并且是对齐的,因此访问效率很高,*gst_efficiency*=1.0.

5) warp 执行效率分析结果

Rodinia 中 17 个程序的 warp 执行效率值如图 11 所示,图 11 中包括 2 项指标:*warp_execution_*

efficiency 与 *warp_nonpred_execution_efficiency*. 可以看到,在 37 个核函数中,大部分函数的 warp 执行效率指标值达到或接近 0.8,只有少数函数的指标值较低,同时可以看到大部分核函数中,*warp_execution_efficiency* 值都会略大于 *warp_nonpred_execution_efficiency* 值,说明在大部分核函数中都存在分支指令的谓词替换.

MC 程序中函数 kernel 的 warp 执行效率值最

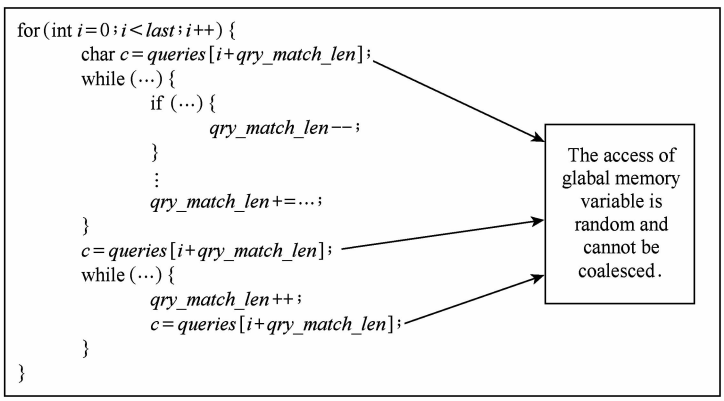


Fig. 9　The analysis of the performance bottleneck of function mummergpuKernel in program MM.

图 9　MM 程序的函数 mummergpuKernel 性能瓶颈分析

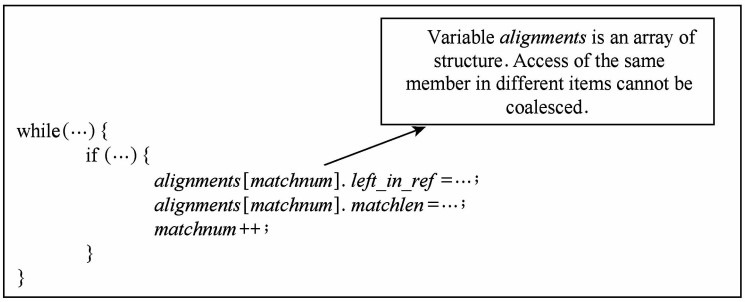


Fig. 10　The analysis of the performance bottleneck of function printKernel in program MM.

图 10　MM 程序的函数 printKernel 性能瓶颈分析

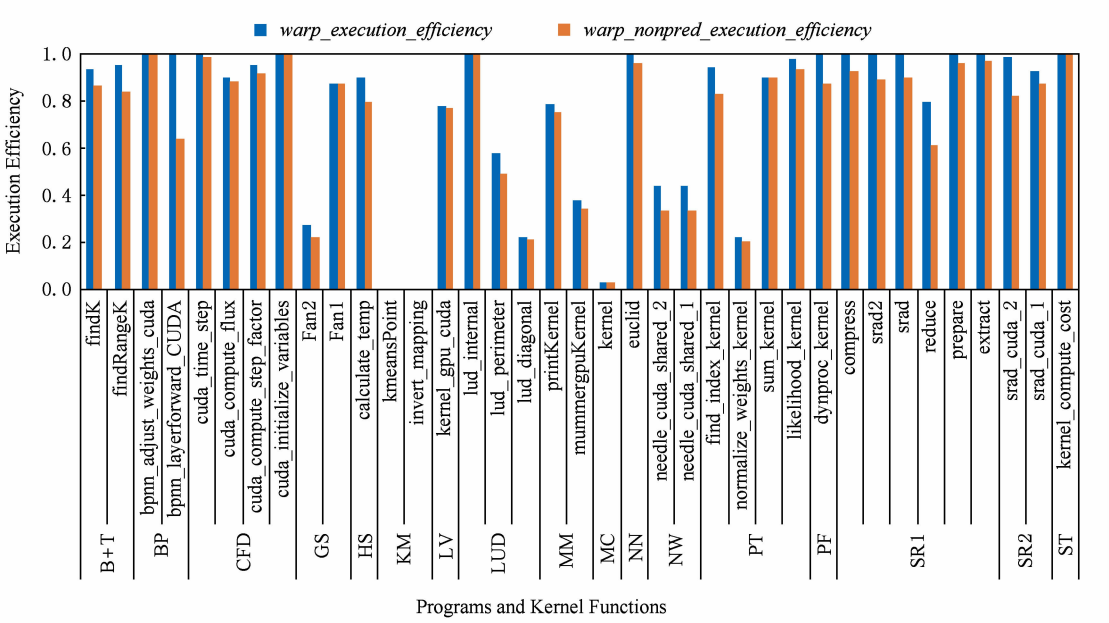


Fig. 11　The warp execution efficiency of the 17 programs in Rodinia.

图 11　Rodinia 中 17 个程序各核函数的 warp 执行效率

低，*warp_execution_efficiency*=0.0344，*warp_nonpred_execution_efficienc*=0.0328. 如图 12 所示，MC 程序的 kernel 运行时只使用了 2 个 warp，而每个 warp 中只有一个线程在执行串行程序，其他 31 个线程空闲，从而使线程分歧比较严重，线程利用率非常低.

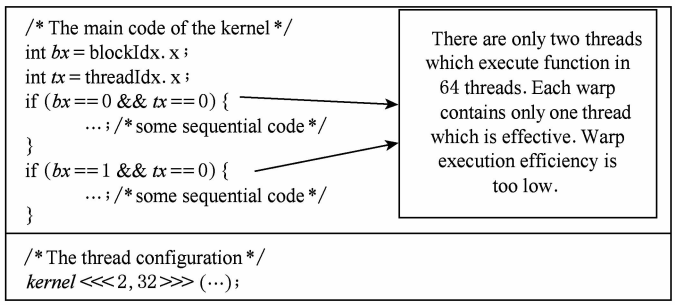


Fig. 12 The analysis of the performance bottleneck of function kernel in program MC.

图 12 MC 程序的函数 kernel 性能瓶颈分析

LUD 程序的函数 lud_internal 不存在分支指令，warp 中所有线程执行路径一致，因此不存在线程分歧，warp 执行效率最高，*warp_execution_efficiency* 和 *warp_nonpred_execution_efficiency* 值都为 1.0.

4.2 应用程序分析

求解全球正压大气浅水波方程的程序分为使用共享内存的版本和不使用共享内存的版本，我们对使用共享内存的版本做了分析，分析结果如表 3 所示：

Table 3 The Analysis Result of The Cubed-sphere Shallow-water Model Program

表 3 全球正压大气浅水波程序性能分析结果

Class	Metric	Value
Calculation and Memory Access	Ratio of Float Point Calculation and Global Memory Access	7.40
	Achieved Float Point Performance	0.05
	Bandwidth Utilization	0.70
Occupancy	Theory	0.50
	Achieved	0.48
Shared Memory Access Efficiency	Shared Memory Access Efficiency	0.54
Global Memory Access Efficiency	Read	0.93
	Write	1.00
Warp Execution Efficiency	Non-Prediction	0.91
	Prediction	0.90

该程序浮点计算与全局内存访问比较高，分析代码可知，程序计算指令比较多且大多数访存是针对的共享存储器，从而有比较高的浮点计算与全局内存访问比值. 程序的实测浮点性能与浮点计算单元利用率比较低，而带宽利用率较高，因此，优化时需要重点针对计算指令进行.

程序的 GPU 占用率指标值较低，分析代码可知，程序中使用了大量的共享内存（每个线程使用 40.5 KB 共享内存），且大量的局部变量占用了大量的寄存器（每个线程使用 63 个寄存器），从而使程序无法同时执行太多的线程，导致并行度不够高. 优化时，可以将局部变量和共享内存中的常量存储于常数存储器中或使用宏定义来代替，以减少对共享内存和寄存器的使用.

程序的共享内存访问效率比较低，存在 bank 冲突. 优化时需要改变共享内存数组的组织与访问形式，消除 bank 冲突. 程序的全局内存访问效率很高，说明很好地利用了合并访问的机制；程序线程束执行效率很高，不存在大量的影响性能的条件分支.

总体来说，程序应该从计算指令吞吐率、GPU 占用率、共享内存数据结构设计和访问方式 3 个方面来进行优化.

5　CUDA 程序性能建议

根据对 Rodinia 和浅水波应用程序的负载分析,本文总结出 CUDA 程序的性能建议:

1) GPU 占用率(occupancy)是反映程序性能的重要指标,能够反映出程序对 GPU 并行计算单元的利用情况,该值较低表示程序对 GPU 并行计算单元的利用率较低.影响程序 occupancy 的指标包括线程块中线程数量、线程占用的寄存器数量和线程块分配的共享存储大小,其中,线程数量反映开发者分配的最大并行线程量,寄存器数量和共享存储大小反映单位线程占用的存储器资源大小.在开发 CUDA 程序时,一方面要尽量减少核函数中局部变量的数量,以减少对寄存器的占用,同时控制共享内存的占用,通过减少单位线程占用的资源量来增大可以同时发起的线程数量;另一方面,在寄存器和共享内存占用量足够的情况下,按照程序输入规模尽量多地开启并行执行的线程.

2) 共享存储器访问效率(*shared_efficiency*)反映的是程序中共享内存的使用效率,能够反映程序的 bank 冲突,该值低于 1.0 时表示存在 bank 冲突.不同的 GPU 架构之下,共享内存会被分为 16 或 32 个 bank,bank 的存储单位大小为 4 B 或 8 B.使用共享存储器时,需要根据变量所占存储空间大小进行控制,尽量使 half-warp 中的不同线程不访问同一个 bank(除非是访问同一个变量).计算能力不小于 2.x 的 GPU 的 bank 数量为 32,这是目前主流的 GPU 结构.对于这其中 bank 大小为 4 B 的 GPU,共享内存数组元素存储大小不大于 8 B 时,在每个线程只访问一个元素的情况下各线程连续访问时恰好不会发生 bank 冲突;大于 8 B 时可能发生 bank 冲突,典型地,当数组元素为 8B 的 2 倍及以上的整数倍时会发生 bank 冲突,由此可知数组元素为基本变量类型时一般不会发生 bank 冲突,数组元素为存储大小较大的结构体时容易发生 bank 冲突,bank 大小为 8 B 的 GPU 在以上情况下发生 bank 冲突的数组元素存储大小为以上对应大小的倍数.开发者使用共享内存时需要注意以上规律.

3) 全局存储器访问效率(*gld_efficiency* 和 *gst_efficiency*)反映程序中全局内存的访问效率,对应值较小时表示全局内存访问模式或访问时对齐方式没有达到最优.全局内存的合并访问机制大大提高了其访问效率,合并访问对于访问方式有一定的要求,不同的 GPU 架构之下其要求不尽相同,但都是针对访问地址的连续和对齐 2 方面.全局内存访问的典型不合理模式包括:依次连续访问元素类型为结构体的数组中结构体元素的同一个成员,进而导致访问地址不连续;条件控制语句中根据变化的条件访问全局内存,导致访问地址的离散化.全局内存访问相对较慢,可以使用其他类型存储器来代替全局存储器以加速数据访问速度,对于常量可以使用常数存储器存储,对于需要多次访问的大量数据则可以存储于共享内存中;对于由于数据量过大等原因而无法存储于其他存储器的全局内存变量,对其访问时尽量使访问地址连续,尽量不要使用结构体数组的数据结构形式,该数据结构形式下很容易使内存访问地址离散化,也要尽量避免在条件控制语句中出现的离散访问;同时,对于不同的 GPU 架构,需要注意对全局内存数据访问的对齐方式,以充分利用合并访问特性进行访存加速.

4) 线程束执行效率(*warp_execution_efficiency* 与 *warp_nonpred_execution_efficiency*)反映了程序的线程分歧情况,前者没有考虑分支预测,后者反映分支预测的情况,综合 2 个指标可以全面了解线程分歧情况.线程分歧会导致大量线程等待或执行无效指令,对程序性能影响很大,程序开发者应尽量减少控制语句的使用,使用一些方法将控制语句转化为非控制语句;对于无法避免的控制语句,尽量使其分支以 half-warp 来划分,使得同一个 half-warp 中执行同一个分支,从而避免线程分歧.

6　结　　论

本文中,我们分析总结了 CUDA 程序的重要性能指标,基于 CUPTI 接口开发了能够获取这些性能指标的性能分析工具,使用该工具对 Rodinia 中的程序进行了负载分析,并对主要性能指标显示的部分程序的性能瓶颈在源代码中进行了原因分析,之后对全球正压大气浅水波模式应用程序进行了性能分析,讨论了其性能特性.对 Rodinia 的负载分析结果较全面地反映了常用 CUDA 程序在各方面的负载特征,该结果可以为未来 CUDA 程序优化分析及 GPU 系统结构优化分析提供数据资料.通过对重要性能指标反映有性能瓶颈的部分程序进行分析,我们为开发者展示了常见性能瓶颈的典型原因,并提出了一些开发高效程序的建议.

参 考 文 献

[1] Zhang E Z, Jiang Y, Guo Z, et al. On-the-fly elimination of dynamic irregularities for GPU computing [J]. ACM SIGPLAN Notices, 2011, 46(1): 369-380

[2] Yang Y, Zhou H. CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications [J]. Journal of Computer Science and Technology, 2015, 30(1): 93-106

[3] Margiolas C, O'Boyle M F P. Portable and transparent host-device communication optimization for GPGPU environments [C] //Proc of Annual IEEE/ACM Int Symp on Code Generation and Optimization. New York: ACM, 2014: 55-65

[4] Xiang P, Yang Y, Zhou H. Warp-level divergence in GPUs: Characterization, impact, and mitigation [C] //Proc of the 20th Int Symp on High Performance Computer Architecture (HPCA). Los Alamitos, CA: IEEE Computer Society, 2014: 284-295

[5] Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing [C] //Proc of IEEE Int Symp on Workload Characterization. Piscataway, NJ: IEEE, 2009: 44-54

[6] Yang C, Xue W, Fu H, et al. A peta-scalable CPU-GPU algorithm for global atmospheric simulations [J]. ACM SIGPLAN Notices, 2013, 48(8): 1-11

[7] Browne S, Dongarra J, Garner N, et al. A portable programming interface for performance evaluation on modern processors [J]. International Journal of High Performance Computing Applications, 2000, 14(3): 189-204

[8] Shende S S, Malony A D. The TAU parallel performance system [J]. International Journal of High Performance Computing Applications, 2006, 20(2): 287-311

[9] NVIDIA Corporation. NVIDIA visual profiler [CP/OL]. 2014 [2014-07-15]. https://developer.nvidia.com/nvidia-visual-profiler

[10] Yang Y, Xiang P, Mantor M, et al. Fixing performance bugs: An empirical study of open-source GPGPU programs [C] //Proc of the 41st Int Conf on Parallel Processing (ICPP). Piscataway, NJ: IEEE, 2012: 329-339

[11] Stratton J A, Anssari N, Rodrigues C, et al. Optimization and architecture effects on GPU computing workload performance [C] //Proc of Innovative Parallel Computing. Piscataway, NJ: IEEE, 2012: 1-10

[12] Yang Y, Xiang P, Kong J, et al. A unified optimizing compiler framework for different GPGPU architectures [J]. ACM Trans on Architecture and Code Optimization, 2012, 9(2): 970-983

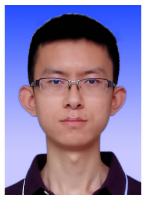
[13] Li C, Yang Y, Lin Z, et al. Automatic data placement into GPU on-chip memory resources [C] //Proc of IEEE/ACM Int Symp on Code Generation and Optimization (CGO). Piscataway, NJ: IEEE, 2015: 23-33

[14] Chen G, Wu B, Li D, et al. PORPLE: An extensible optimizer for portable data placement on GPU [C] //Proc of the 47th Annual IEEE/ACM Int Symp on Microarchitecture (MICRO). Piscataway, NJ: IEEE, 2014: 88-100

[15] Wahib M, Maruyama N. Scalable kernel fusion for memory-bound GPU applications [C] //Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2014: 191-202

[16] Li Jianjiang, Li Xinggang, Lu Chuan, et al. A template technology for transplanting from single-GPU programs to multi-GPU programs [J]. Journal of Computer Research and Development, 2010, 47(12): 2185-2191 (in Chinese)
(李建江, 李兴钢, 路川, 等. 一种单 GPU 程序向多 GPU 移植的模板化技术[J]. 计算机研究与发展, 2010, 47(12): 2185-2191)

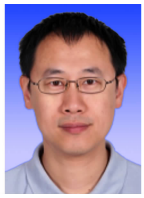
[17] Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for multicore architectures [J]. Communications of the ACM, 2009, 52(4): 65-76



Zheng Zhen, born in 1991. PhD candidate. His main research interests include high performance computing.

Zhai Jidong, born in 1981. PhD and assistant professor. His main research interests include high performance computing and performance evaluation.

Li Yan, born in 1985. PhD. His main research interests include high performance computing and parallel computing.



Chen Wenguang, born in 1972. PhD and professor. His main research interests include high performance computing and compiler optimization.