第41卷 第1期 2014年1月 | 计 算 机 科 学 Computer Science | Vol. 41 No. 1 Jan 2014

基于GPU的并行计算性能分析模型

王卓薇　程良伦　赵武清

（广东工业大学计算机学院　广州 510006）

摘　要　针对GPU并行计算领域缺少精确的性能分析模型和有针对性的性能优化方法，提出一种基于GPU的并行计算性能定量分析模型，其通过对指令流水线、共享存储器访存、全局存储器访存的性能建模，来定量分析并行程序，帮助程序员找到程序运行瓶颈，进行有效的性能优化。实验部分通过3个具有代表性的实际应用（稠密矩阵乘法、三对角线性方程组求解、稀疏矩阵矢量乘法）的性能分析证明了该模型的实用性，并有效地实现了算法的优化。

关键词　GPU，性能定量分析模型，指令流水线，共享存储器访存，全局存储器访存

中图法分类号　TP311　　　**文献标识码**　A

Parallel Computation Performance Analysis Model Based on GPU

WANG Zhuo-wei　CHENG Liang-lun　ZHAO Wu-qing

(Faculty of Computer, Guangdong University of Technology, Guangzhou 510006, China)

Abstract　In order to solve the problem of lacking accurate performance analysis model in parallel computation field based on GPU, we proposed a quantitative performance model which can simulate the performance of three major components of GPU including instruction pipeline, shared memory access time, and global memory access time. It is designed to build a performance model that helps programmer find the performance bottlenecks and improve the system's performance efficiently. To demonstrate the usefulness of the model and to optimize the algorithms performance, we analyzed three representative real-world programs: dense matrix multiplication, tridiagonal systems solver, and sparse matrix vector multiplication.

Keywords　GPU, Quantitative performance model, Instruction pipeline, Shared memory access time, Global memory access time

1　引言

目前基于GPU通用计算的研究，大量工作都专注于应用的开发，很少专注于支持性能剖析和分析的工具。商业程序分析工具如ATI Stream Profiler[1]、NVIDIA Parallel Nsight[2]等都建立在GPU的计算功能模拟器[3,4]之上，只提供程序运行时间的统计，却不涉及针对程序性能的统计分析。因此在并行程序执行过程中，有关确定程序性能瓶颈以及估算程序潜在性能优势的相关工作都依赖于程序员手中的笔和纸。除了商业程序分析工具之外，同时也出现了一些有关自动调优运行机制的研究[5-9]。自动调优运行机制对于性能优化工作来说是一个强大并有效的工具，但存在着两个不可忽略的问题。首先，这些研究几乎没有提供GPU程序执行过程的精确分析。其次，自动调优机制需要程序员在并行化过程中，以参数化形式编写程序，以便在性能分析时能够适应不同参数（并行粒度、灵活的内存访问模式、循环展开的程度、应用程序特定参数等等）的调整方案。

针对上述问题，本文专注于并行程序在GPU平台上运行时性能瓶颈体现的3个重要组成部分（指令流水线、共享存储器访存、全局存储器访存）开发和设计基于微基准程序的性能分析模型，以定量分析的模式确定程序的性能瓶颈，找出导致该性能瓶颈的原因，以便能有效预测程序在经过优化及并行架构改进之后带来的潜在的性能优势。通过稠密矩阵乘法、三对角线性方程组求解、稀疏矩阵矢量乘法典型应用的性能分析，详细说明该模型如何指导优化并行应用，并提出GPU体系架构的改进策略，包括硬件资源分配、bank conflicts、线程块选取、访存事务粒度等。

本文第2节介绍GPU环境下并行计算性能建模与分析方法；第3节、第4节、第5节分别提出了指令流水线、共享存储器访存、全局存储器访存的性能定量分析模型；第6节以分别代表指令流水线、共享存储器访存、全局存储器访存性能瓶颈的稠密矩阵乘法、三对角线性方程组求解、稀疏矩阵矢量乘法为例给出实验性能测评与分析，证明该模型的有用性并指导性能优化；最后总结全文。

2　性能定量分析模型

要建立基于GPU的性能定量分析模型，需要提出定量分析程序性能瓶颈的基准，这些基准被称为微基准程序。如

到稿日期：2013-03-26　返修日期：2013-06-24　本文受广州市科技项目（2012Y2-0031），博士后基金（2013M531825），国家自然科学基金（U1201251）资助。

王卓薇（1985－），女，博士，讲师，主要研究方向为高性能计算，E-mail：wangzhuowei0710@163.com；程良伦（1964－），男，博士，教授，主要研究方向为物联网、信息物理融合系统；赵武清（1986－），男，博士，主要研究方向为云计算。

何设计与开发微基准程序是本节需要重点解决的问题。在CUDA应用程序的开发过程中，按照开发流程的先后顺序，程序的编写与优化需要解决任务串行并行划分、数据执行步骤确定、程序平稳正确运行、显存访问优化、指令流划分、资源均衡、主机与显存之间通信优化等7个问题。因此可以将影响并行程序性能的瓶颈分为3类：指令流水线、共享存储器访存、全局存储器访存。本文根据微基准程序分别对这3类影响GPU性能的主要功能部件进行建模，预估每个功能部件的执行时间，根据拥有最长执行时间的功能部件，就可以确定该程序的性能瓶颈。在本文中，假设具有性能瓶颈的功能部件运行时间基本上覆盖了其他模块的运行时间，这主要是由于以下两个原因：

(1)GPU允许同步指令集、共享存储器以及全局存储器的操作。

(2)GPU的设计哲学就是通过零代价的细粒度线程切换，也就是保持多个active warp的上下文，来隐藏延时。

通过栅栏同步将程序分成多个执行阶段，由于GPU的栅栏同步在线程块中本地执行，因此当一个SM中只有一个线程块时，可以通过栅栏同步将所有执行阶段序列化，并确定每个执行阶段的性能瓶颈。如果一个SM中有多个线程块，由于每个执行阶段在时间上会有重合，只能确定在整个程序执行过程中对系统性能产生影响最大的性能瓶颈。

3 性能分析模型

3.1 硬件平台

本文实验硬件平台CPU为Intel(R)Core(TM)920，两块GPU，NVIDIA Geforce9600用作显示，NVIDIA GTX 285用作计算。

3.2 软件平台

性能模型的数据流图如图1所示。程序通过CUBIN生成器生成基于GPU原始代码的微基准程序benchmarks，由于本地机器代码指令集是不公开记录的，因此我们使用cubin的反编译工具Decuda。通过Decuda建立一个工具来修改原始的二进制指令，将修改的指令合并在二进制代码序列的后面，最后将修改的代码嵌入到执行文件。这个工具可以避免编译的干扰，例如消除无作用的代码(dead code)。

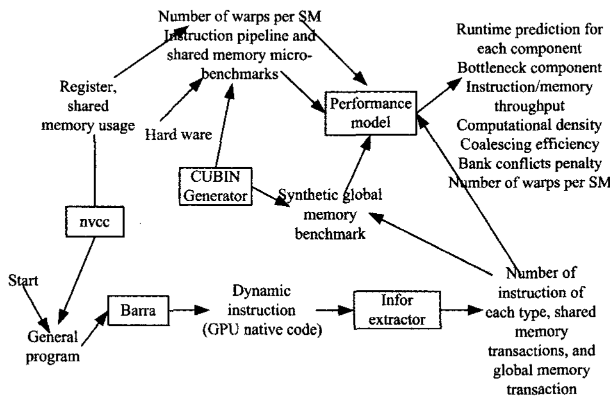


图1 基于GPU的并行计算定量性能分析模型数据流图

使用功能模拟器Barra[10]生成程序的动态执行信息，该信息包含了每条指令执行的次数。然后通过Info extractor提取从功能模拟器Barra产生的动态指令，生成性能模型3个功能模块(不同类型指令个数、共享存储器访存事务次数、全局存储器访存事务次数)的输入。

4 指令流水线

GPU通过交错执行多个并行线程来达到隐藏指令流水线和访存延迟的目的。如果指令流水线是完全饱和的，则表明此时系统运行达到了性能峰值。指令流水线的性能建模难度在于指令流水线不饱和，对于GPU应该如何建模。

在GPU的指令加载中，任务分发是由计算分发单元(Compute Scheduler Units,CSU)来完成的，而真正进行分发的单位则是协作线程阵列CTA(Collaborative Thread Array)。共享存储器能够被同一线程块中的线程共享，因此一个SM应该分配属于CTA中的线程。目前普遍使用的硬件中最多只能由16个warp(相当于512个线程)组成一个CTA。

计算分发单元通过轮询算法将每个CTA均匀地分发到每个SM中，同时一个SM中应该尽可能多地分配CTA。由于SM是以warp为单位执行CTA，因此一个warp中所有线程必须属于同一CTA。换言之，一个SM上可以同时存在多个warp上下文，但在同一时刻只能存在一个warp在系统上执行。天然的细粒度线程并行切换隐藏访存延时是GPU发挥高性能计算的关键因素，要实现高吞吐量，就应该提高SM中计算资源的利用率，保证每个SM上能有足够多的active warp。因此指令流水线的不饱和跟没有足够的active warp有很大的关系。

不同指令类型需要不同的计算资源，这取决于这个指令有多少个功能单元。因此本文根据指令运行在一个SM上所需的不同的功能单元数量来对所有指令进行分类，如表1所列，一个SM中有10个乘法器功能单元(其中8个来自于浮点运算单元，2个来自特殊功能单元)。指令吞吐量的理论峰值计算如式(1)所示，其中numberFunctionUnits代表功能单元的数量，frequency代表GPU的核心频率，SM代表流多处理器(Stream Multiprocessor)的数量，而WarpSize则代表warp中thread的数量(一般一个warp中有32个thread)。

$$\frac{numberFunctionUnits \times frequency \times SM}{WarpSize} \quad (1)$$

表1 指令类型

指令类型	功能单元数量	指令名称
Type 1	10	mul
Type 2	8	Mov,add,mad
Type 3	2	Sin,cos,log,rep
Type 4	1	double precision floating point

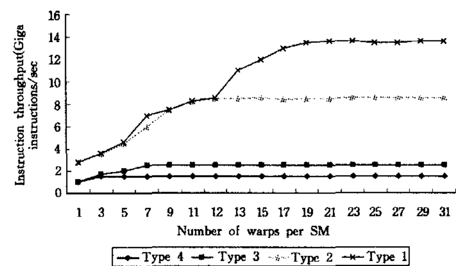


图2 不同指令类型对应的指令吞吐量

例如，指令MAD的峰值吞吐量为$\frac{8 \cdot 1.48GHz \cdot 30}{32}=$ 11.1Giga instuctions/s。一个MAD需要进行2次浮点运算，

因此理论峰值浮点性能为 11.1 • warpSize • 2＝355.2 • 2＝710.4GFLOPS。

通过微基准测试程序来反复运行各类型的指令，根据改变 active warp 数量来测量不同类型指令的吞吐量，如图 2 所示，可以看到越多的功能单元就需要提供越多的 active warp 去隐藏指令流水线的延迟。还可以发现，在现代 GPU 体系架构中，一个 SM 至少有 6 个 active warp 才能隐藏延迟。

5 共享存储器

计算能力 1.0/1.1/1.2/1.3 硬件中，每个 SM 的共享存储器大小为 16kB，被组织为 16 个 bank(按 0—15 编号)，共享存储器的访存带宽的理论峰值如式(2)所示，公式中 SM 表示流多处理器个数，SP 表示流处理器个数(一个 SM 中有 8 个 SP)。

$$SP \cdot SM \cdot frequency \cdot 4\text{B} = 8 \cdot 30 \cdot 1.48\text{GHz} \cdot 4\text{B} = 1420\text{GB/S} \quad (2)$$

同样根据改变 active warp 数量来测试共享存储器的访存吞吐量，如图 3 所示。比较图 2 与图 3，可以发现共享存储器比指令流水线拥有更长的存储流水线，因此需要更多的 active warps 来隐藏共享存储器的访存延迟。由于存储器位于 GPU 片上，因此访存速度比 local/global memory 快很多。通常情况下，共享存储器的延迟几乎只有 local/global memory 的 1/100，访问速度与寄存器相当。共享存储器与指令流水线相比，需要更多的 active warps 来隐藏访存延迟，这是由于数据在访问共享存储器时，出现了 bank conflicts，影响了共享存储器的访存事务的次数。

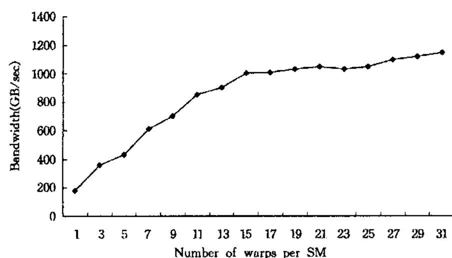


图 3 不同的 active warps 对应的共享存储器访存带宽

在共享存储器性能建模中，编写了一个自动获取共享存储器访存带宽的程序，该程序综合考虑了由于 bank conflict 导致共享存储器访存延迟的问题，根据应用程序在共享存储器访存中 bank conflict 的程度，来确定共享存储器访存事务的次数。

6 全局存储器

由于全局存储器在各个 SM 之间进行共享，因此在对全局存储器访存带宽的建模中，不能像指令流水线和共享存储器那样使用 active warps 来对全局存储器进行建模。在 GPU 现代架构中，一个 grid 中的任意线程都能在全局存储器中的任意位置进行读写，因此全局存储器的访存带宽主要受以下 3 个因素影响：

(1)block 数量；

(2)每个 block 中的 thread 数量；

(3)每个 thread 的全局存储器访存事务次数。

为了能够有效利用全局存储器的高带宽，需要通过上述 3 个因素进行合理调节。全局存储器的理论带宽峰值如式(3)所示。

$$\frac{MemoryFrequency \cdot busWidth}{8\text{bits/byte}} = \frac{2.484\text{GHz} \cdot 512\text{bits}}{8\text{bit/byte}} = 160\text{GB/S} \quad (3)$$

本文根据上述 3 个因素来对全局存储器的带宽进行建模，如图 4 所示。30 个 SM 被分成 10 组，每组有 3 个 SM 共享一个单独的存储流水线。从图中可以看到锯齿形状以 10 为单位进行改变。测量的带宽更接近理论峰值，所对应的 block 的数量正好为 10 的倍数。从图中还可以看出随着 block 数量的增长，图像的抖动变得越来越平滑。当没有足够的全局存储器访问次数来隐藏访存延迟时，全局存储器的带宽几乎和 block 的数量成线性的关系，但是此时全局存储器的访存带宽远远低于理论带宽峰值。

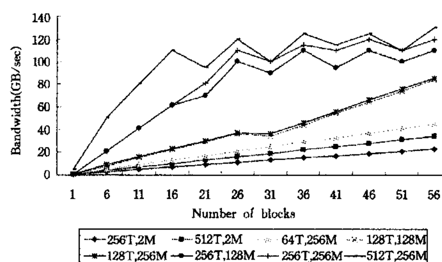


图 4 不同线程块、线程以及每个线程访存次数对应的全局存储器带宽

由于全局存储器的访存比较复杂，因此很难像对指令流水线和共享存储器一样来对全局存储器进行简单的建模。影响 GPU 应用程序性能最明显的因素就是全局存储器是否满足合并内存访问条件。为了能够很好地解决这个问题，应当遵循合并内存访问协议。在全局存储器建模中，综合考虑全局存储器的合并内存访问，根据应用程序在实际运行过程中是否满足合并内存访问的条件，确定共享存储器访存事务的次数。

7 案例研究

7.1 稠密矩阵乘法

在稠密矩阵乘法中，对矩阵进行了棋盘划分，使用共享存储器来实现矩阵乘法。在这个实例中，每个 block 负责数组 C 中一个方块 Csub 的计算，其中每个线程负责计算 Csub 的一个元素，如图 5 所示。Csub 是两个巨型矩阵 A 与 B 相乘。A 的 sub-matrix 与 Csub 有相同的行索引，B 的 sub-matrix 与 Csub 有相同的列索引。为了适应设备资源，两个巨型矩阵被划分为大小为 block_size 的正方形矩阵，Csub 就是这些小正方形矩阵乘积的和。矩阵乘积计算过程为首先从全局存储器加载两个相应的正方形矩阵到共享存储器，一个 thread 负责加载一个元素；接下来每个 thread 负责计算乘积中的一个元素；thread 累计每个乘积的结果到一个寄存器中，一旦结束就写回全局存储器。

sub_matrix 的大小对应程序设定的 block_size，所以 sub_matrix 的大小可以根据 block_size 的设定而进行改变。本文研究的工作是在 Volkov 和 Demmel[11] 提出稠密矩阵乘算法的基础上进行性能分析，该算法也是利用分块策略来计算矩阵乘法，设定 sub_matrix 尺寸为(16×16)，但是并没有说明

该尺寸优于其他尺寸(8×8,32×32)的原因;另外在执行过程中,该算法的实际性能只达到了 GPU 性能峰值的 58%,也没有说明性能没有达到理想峰值的原因。接下来本文将通过性能定量分析模型来对上述两个问题进行解释说明。

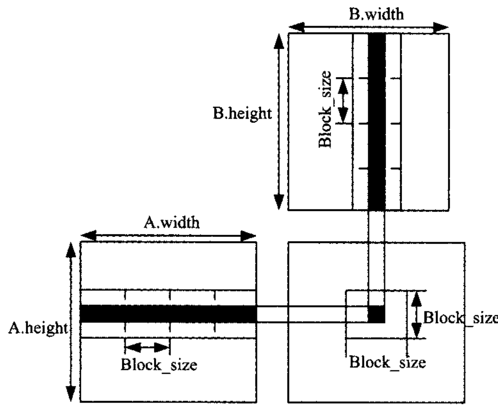


图 5 稠密矩阵乘法

首先测试模拟 sub_matrix 尺寸分别为 8×8、16×16、32×32 时稠密矩阵乘算法的程序性能。理想情况下,sub_matrix 尺寸越大,其程序性能越好,这主要有以下两个原因:

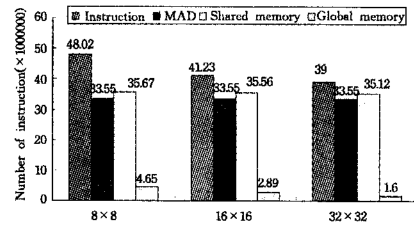
(1)尺寸越大的 sub_matrix 越能有效减少存储器加载次数。图 6(a)显示了 sub_matrix 尺寸由 8×8 增加到 16×16 时,全局存储器访存事务次数减少了 45%,而 sub_matrix 尺寸由 16×16 增加到 32×32 时,全局存储器访存事务次数则相应减少了 40%。

(2)尺寸越大的 sub_matrix 越增加了矩阵计算的密度。图 6(a)同时也显示了当 MAD 指令数保持常数($\frac{matrixSize^3}{warpSize}$)时,sub_matrix 尺寸越大,系统所有动态指令数就越少。

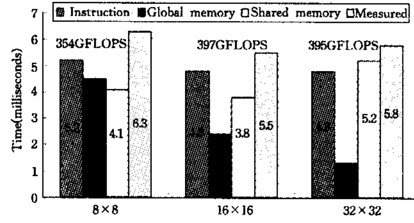
但在实际执行过程中,发现 sub_matrix 尺寸为 16×16 时,程序性能最优,图 6(b)显示 sub_matrix 分别为 3 个尺寸所对应的指令流水线、共享存储器访存、全局存储器访存的执行时间,以及程序性能。从图中可以看出,当 sub_matrix 的大小为 8×8 和 16×16 时,系统性能瓶颈为指令流水线;为 32×32 时,系统性能瓶颈由指令流水线转移到共享存储器访存。尽管 32×32 拥有与 16×16 相近的共享存储器访存指令数(见图 6(a)),但是在共享存储器访存上花费的时间开销较大(见图 6(b)),这主要是由于不同的 sub_matrix 所使用的硬件资源也不相同,如表 2 所列。32×32 所使用的寄存器和共享存储器的数量明显要高于 8×8 和 16×16,并且 active block 的数量由 8 个降到 3 个。这将导致没有足够的并行计算来隐藏指令流水线和共享存储器访存延迟,最终导致性能整体下降。

图 6(b)显示了使用性能分析模型模拟的程序执行时间要比实际测试执行时间低 14%,这是由于本文没有考虑栅栏同步对程序性能产生的影响,block 中线程的栅栏同步能够保证所有数据存储到共享存储器之后才进行计算,所以实际并行运行的 warps 的数量要比总的 warps 少,导致在实际运行过程中指令和共享存储器的吞吐量也相应较低。总结该算法性能只达到 GPU 性能峰值 58%的主要原因有两点:

(1)指令流水线不饱和,指令吞吐量只达到性能峰值的 83%;



(a) 一个 warp 的指令数,MAD,共享存储器访存次数,全局存储器访存次数



(b)实际测试性能与性能模型性能

图 6 sub_matrix 分别为 8×8,16×16,32×32 的性能对比(矩阵为 1024×1024)

(2)尽管所有指令中,参与计算的 MAD 指令占到 80%左右,但依然有 20%的指令(程序控制指令、寻址指令、存储操作数指令等)并未参与计算。

根据以上分析结果,可以进一步提出以下优化策略:

(1)当 sub_matrix=8×8/16×16 时,active warps 最大值为 32,active block 最大值为 8。如果在不改变其他条件的情况下,将 active block 增加到 16,将会有更多并行 warps 来达到更高指令和存储器访存吞吐量。

(2)当 sub_matrix=32×32 时,如果增加 SM 中寄存器和共享存储器的数量,一个 SM 将有更多的 warps 参与计算,因此可以获得更高的程序性能。

表 2 sub_matix 分别为 8×8、16×16、32×32 时,寄存器、共享存储器的使用情况以及 active blocks 与 active warps 的数量

sub_matrix (size)	寄存器 (个)	共享存储器 (Byte)	Blocks (寄存器数量限制) (个)	Blocks (共享存储器数量限制) (个)	Active blocks (个)	Active warps (个)
8×8	16	348	16	47	8	16
16×16	30	1088	8	15	8	16
32×32	58	4284	3	3	3	6

7.2 三对角线性方程组求解

三对角线性方程组求解是众多科学与工程计算中最重要也是最基本的问题之一[12,13]。三对角线性方程组的表达式为

$$AX=Y$$

其中,$A\in R_{n\times n}$为非奇异矩阵,当$|i-j|>1$时 $a_{ij}=0$,$X=(X_1\ X_2\cdots\ X_n)^{\mathrm{T}}$,$Y=(Y_1\ Y_2\cdots\ Y_n)^{\mathrm{T}}$。

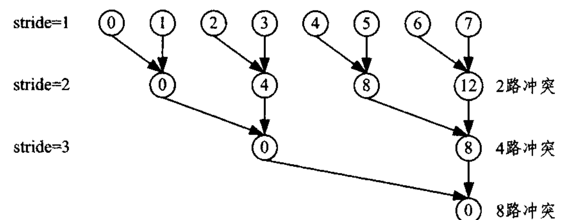


图 7 向前约化通信模式

循环规约(Cylic Reduction,CR)是求解三对角线性方程

组的典型方法，基本思想是每次迭代将偶数编号方程中的奇变量消去，只剩下偶变量，问题转变成求解仅由偶变量组成规模减半的新三对角方程组。求解该新方程组得到所有的偶变量后，再回代求解所有奇变量，因此向前约化和向后回代是循环规约的两个基本步骤。由于这两个步骤具有相似的通信模式，因此本文以向前约化为例进行详细说明，如图 7 所示，对于 n 个输入元素，向前规约需要 $\log 2^n$ 执行步。所有输入数据首先加载到共享存储器，然后在片上进行计算。从向前约化的过程中可以看出：

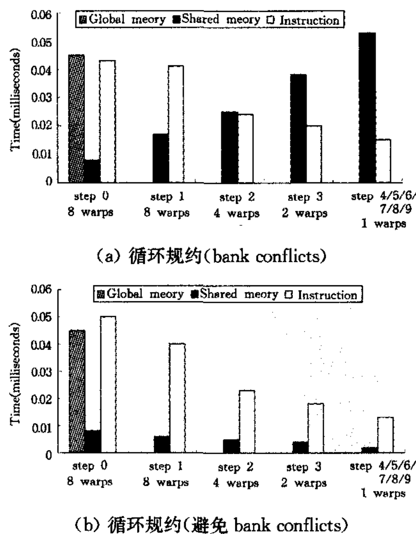
(1)第一次循环(stride=1)中，只有 index=0,2,4,6 的线程在做有效计算，即这些线程分别与其后跨度为 1 的元素加和操作。

(2)第二次循环(stride=2)中，只有 index=0,4,8,12 的线程在做有效计算，即这些线程分别与其后跨度为 2 的元素加和操作。

(3)第三次循环(stride=4)中，只有 index=0,8 的线程在做有效计算，即这些线程分别与其跨度为 4 的元素加和操作。

(4)依次类推。

向前约化过程中随着 stride 不断增加，会造成越来越剧烈的 bank conflict。例如 stride=1 时，0－15 号 half-warp 线程形成 2 路冲突，stride=2 时，half-warp 线程形成 4 路冲突，stride=4 时，half-warp 线程形成 8 路冲突。假设本例中，有 2 个一维 block，每个 block 有 512 个线程，输入数据元素数量为 1024 个，每个线程对应处理一个数据，向前约化共需 $\log 2^{(512)}=9$ 个执行步。在第 1 次循环中，只有 256 个线程分别与其后跨度为 1 的元素进行计算；在第 2 次循环中，就只有 128 个线程分别与其后跨度为 2 的元素加和操作；依次类推，直到最后一次循环，只有一个线程参与计算，即与其后跨度为 256 的元素加和操作。GPU 最小的执行单元为 warp(32 个线程)，在第 4－9 次循环中，由于程序算法的性能特点相同，因此将 4－9 次循环性能瓶颈作相同分析。

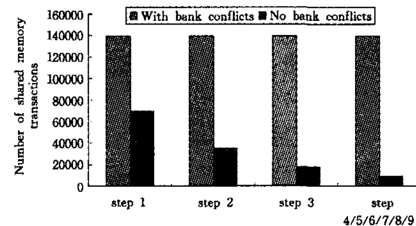


(a) 循环规约(bank conflicts)

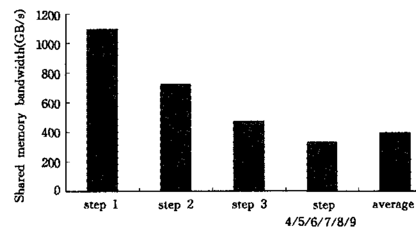
(b) 循环规约(避免 bank conflicts)

图 8 循环规约算法分别在 bank conflicts 和避免 bank conflicts 下的性能瓶颈分析(block 大小为 512，step 0 表示将数据从全局存储器加载到共享存储器，计算过程则从 step 1 开始)

图 8 显示了循环规约分别在 bank conflicts 和避免了 bank conflicts 的情况下的性能瓶颈分析。由于共享存储器数量的限制，一个 SM 中只分配一个 block，所有 thread 都来自于同一个 block，在每次循环计算步骤中，为了保证数据的正确性，需要进行栅栏同步，因此数据从全局存储器加载到共享存储器的过程，以及后面在共享存储器中的计算都以串行执行。如图 8(a)所示，循环规约过程的性能瓶颈分别为：第 0 次循环，全局存储器访存延迟；第 1 次循环，指令吞吐量；后面所有次循环，共享存储器访存延迟。这是因为每次循环的工作量随着循环规约次数的增加而减少，然而由于越来越剧烈的 bank conflict，每次循环的共享存储器访存事务次数没有减少反而基本保持常数不变，如图 9(a)所示。而共享存储器的访存带宽却随着 active warps 数量的减少而减少，如图 9(b)所示。



(a) 循环规约每次循环时共享存储器访存事务数



(b) 循环规约中不同 active warps 下共享存储器访存带宽

图 9 循环规约算法的共享存储器访存带宽及访存事务次数

图 8(b)显示通过填充技术避免 bank conflicts，其性能瓶颈由共享存储器访存延迟转移到指令流水线，程序性能得到大幅提升。在第 1 次循环计算中，由于填充技术导致更复杂的寻址计算，因此其性能瓶颈为指令吞吐量；在随后的循环计算中，由于避免了 bank conflicts，其性能瓶颈也由共享存储器访存转移到指令吞吐量。

性能分析模型测试性能和实际执行性能对比如图 10 所示，误差基本在 7.5%范围内，因此本文提出的定量性能分析模型的正确性得到了有效验证。在循环规约执行过程中，共享存储器访存消耗时间最长；在避免 bank conflicts 循环规约执行过程中，指令执行时间所占比重最大，而共享存储器访存时间所占比重则较小。在现代 GPU 体系架构中，通用存储器的体系设置目前只能用于解决具有简单、重复计算特点的问题。随着 GPU 通用计算应用范围越来越广泛，所需解决的问题也越来越复杂，本文提出性能分析模型的目的就是不仅能对性能瓶颈进行定量预测分析，而且还能有效指明 GPU 体系架构的改进方向。根据三对角线性方程组求解的性能分析，可以得出两条改善性能的方法：

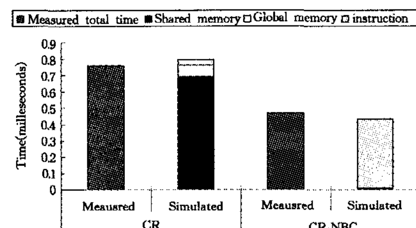


图 10 循环规约在 bank conflicts (CR)和循环规约避免 bank conflicts(CR-NBC)的预测与实际性能对比

(1)更改共享存储器中 bank 数量,将其设置为素数,可以有效避免 bank conflicts。

(2)释放一个 block 中未使用的硬件资源,也就是使用较小尺寸的 block,使得一个 SM 中能够拥有更多的 active block 进行并行计算,达到增加 warps 并行度并减少指令和共享存储器吞吐量的目的。

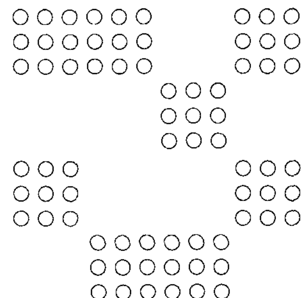
7.3 稀疏矩阵矢量乘法

稀疏矩阵矢量乘算法(SpMV)是科学与工程计算中的典型应用,但是在 GPU 平台上进行并行计算的过程中,由于稀疏矩阵对应的行元素与向量对应相乘的元素都将被读入 GPU 全局存储器,这意味着每次浮点运算需要更多的存储器访存开销,因此该算法的全局存储器访存开销是影响系统性能的关键因素。

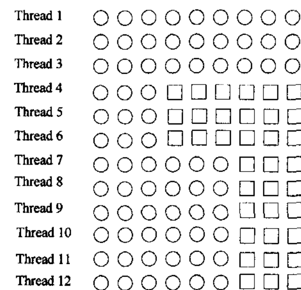
稀疏矩阵具有多种数据存储格式(COO、CSR、CSC、ELLPACK),Bell 等[14]根据不同的数据存储格式对稀疏矩阵矢量乘算法性能展开研究,发现 ELLPACK 存储格式(简称 ELL)满足合并内存访问条件,其性能要优于其他存储格式。ELL 构成方式与 CSR 非常类似,将非零元素移动到矩阵的左侧,而零元素则被移动到矩阵的右侧,整理好矩阵后,根据矩阵中每行最多非零元素个数确定 ELL 存储格式中最大行长 Max,位于 Max 之后的所有零元素则被丢弃。图 11 显示了 12×12 稀疏矩阵的 ELL 存储格式。ELL 由两个数组构成,分别是 A 和 Col_Idx,稀疏矩阵按列宽进行列扫描,非零元素依次存储于数组 A 中,而数组 Col_Idx 则用来存储数组 A 中非零元素在该矩阵对应位置的列坐标。在 GPU 平台进行计算时,为了相邻线程能够访问相邻存储地址,满足合并内存访问条件,由一个线程负责稀疏矩阵一行数据的计算,按列宽进行扫描,加载数据 A、Col_Idex 以及向量元素到全局存储器。

Choie 等在文献[15]中对 ELL 存储格式进行改进,提出分块 ELL 存储格式(BELL)。BELL 相较于 ELL 的优势在于将稀疏矩阵按块进行存储,不再需要存储所有非零元素的列坐标,而只需存储每个块矩阵靠左第一个元素的列坐标。对于图 11(a)所示的 12×12 稀疏矩阵,可以分成 8 个 3×3 块矩阵,每个块矩阵全局存储器只需存储 9 个非零元素、1 个列坐标以及 3 个向量,因此 BELL 格式将列坐标和向量元素加载次数分别减少到原来的 1/9 和 1/3。由于每个线程负责稀疏矩阵相邻 3 行元素的计算工作(如图 11(c)所示),线程的访问地址是不连续的,并且非对齐,此时全局存储器访问开始不满足合并内存访问条件,导致系统性能下降。为了满足合并内存访问,需要对行元素的执行顺序进行错序调整,使得相邻线程能够访问连续存储地址(如图 11(d)所示),从而达到提升系统性能的目的。

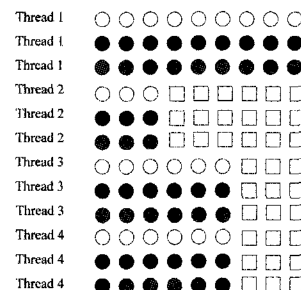
同理,向量对全局内存的访问是否满足合并访问条件也是影响程序性能的关键因素,取决于稀疏矩阵的稀疏程度。一般相邻的两行非零元素拥有相似存储地址,如果这两行元素经过错序调整后相隔越远,则对应相乘的矢量元素共享一个单独内存访存事务的机会就越少。因此也需要对矢量元素的加载顺序进行错序调整。图 12 显示了矢量交错排序对全局内存访存事务次数的影响程度。为了使理解简单一些,假设 half-warps 大小为 2 个而不是 16 个 thread,全局内存访存事务大小为 8byte 而不是 32byte。图 12(a)显示了每个线程访问矢量的次数取决于稀疏矩阵,第一组两个线程(thread 1,thread 2)和第二组两个线程(thread 3,thread 4)没有共享的内存访问事务,例如,thread1 访问第一个矢量地址与 thread 2 访问第一个矢量地址相隔 6 个连续访存地址空间,导致不能进行一次 8byte 合并内存访问。图 12(b)则显示交错排序之后,thread 3 和 thread 4 共享 6 个访存事务,可以进行合并内存访问,同时 thead1 访问第一个矢量地址与 thread2 访问第一个矢量地址距离也由 6 个地址空间减少到 2 个。



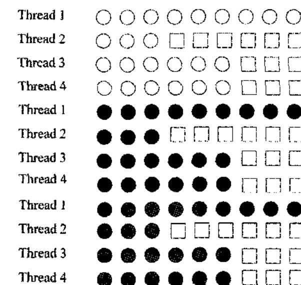
(a) 12×12 稀疏矩阵



(b) ELL 存储格式(一个 thread 负责稀疏矩阵一行元素计算)



(c) BELL 存储格式(一个 thread 负责稀疏矩阵 3 行元素计算)



(d)交错 BELL 存储格式

图 11 12×12 稀疏矩阵的 ELL 存储格式

按照 3×3 矩阵进行分块的稀疏矩阵中,假设线程访存事务粒度为 4byte,理想的情况下,如果所有的全局内存访问满足合并内存访问条件,则 ELL 格式处理一个块矩阵需要 4+4+4=12byte 数据传输。然而在 CUDA 编程模型中,默认线程访存事务大小为 32byte,并且线程对全局内存中矢量的访问不满足合并内存访问条件,这将导致平均每个矢量内存访

问事务需要 6.67byte。如果将 CUDA 编程模型中线程访存事务大小减少为 16byte,则平均每个矢量内存访问事务大小也相应减少为 4.55byte。图 13 显示了 ELL、BELL+IM、BELL+IMIV 3 种不同存储格式下稀疏矩阵、列坐标、矢量平均访存事务所需的字节数。从图中可以看出矢量平均内存访存事务随着线程访存事务字节数的减少而减少。对于 BELL 格式,每个块矩阵中 9 个非零元素共享一个列坐标,每个列坐标内存访问次数减少到 1/9,因此列坐标全局内存访存事务所需字节数相较于 ELL 显著减少,同时又由于矢量进行错序调整后满足合并内存访问条件,因此矢量平均内存访存事务所需字节数相较于没有经过错序调整的也得到大幅减少。

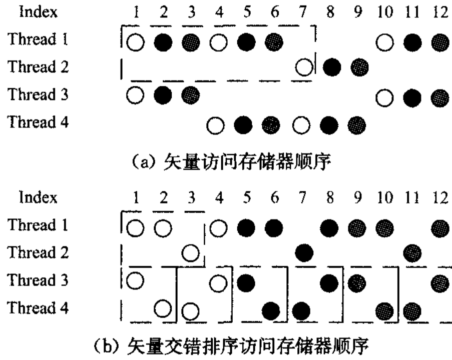
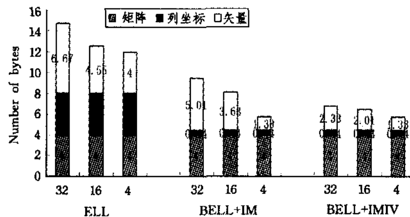


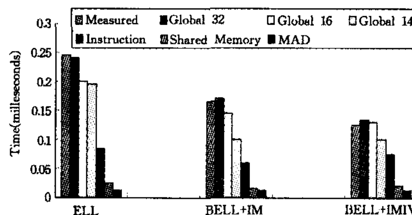
图 12　矢量交错排序对全局内存访存事务次数影响程度



BELL+IM:BELL 存储格式,稀疏矩阵经过错序调整;BELL+IMIV:BELL 存储格式,稀疏矩阵经过错序调整,矢量经过错序调整

图 13　ELL、BELL+IM、BELL+IMIV 3 种不同存储格式下稀疏矩阵、列坐标、矢量全局存储器平均访存事务所需字节数

图 14 显示了稀疏矩阵矢量乘算法实际测试性能与性能分析模型模拟性能的对比,误差保持在 5%范围内。从图中可以看出稀疏矩阵矢量乘算法的性能瓶颈是全局存储器访存延迟,当 CUDA 内存访存事务大小为 16byte 时,程序性能得到提升。如果全局存储器访存延迟得到隐藏,此时系统性能瓶颈则由全局存储器访存延迟转移到指令流水线。但是此时程序性能仍然要远远低于 GPU 的浮点性能峰值,这是由于该算法的计算密集度低,导致所有指令中只有 1/10 左右的指令用于计算。根据稀疏矩阵矢量乘算法的性能分析,可以得出两个改善性能的方法。



Global 32:全局内存访存事务大小为 32byte;Global 16:全局内存访存事务大小为 16byte;Global 4:全局内存访存事务大小为 4byte

图 14　12×12 稀疏矩阵矢量乘算法实际测试性能与性能分析模型模拟性能的对比

(1)将稀疏矩阵与对应相乘矢量进行错序调整,使之满足合并内存访问条件;

(2)更改全局共享存储访存事务大小,将 32byte 调整为 4byte,列坐标与矢量对全局存储器平均访存所需字节数减少,减少了访存延迟。

结束语　本文针对 GPU 并行计算领域缺少精确的性能分析模型和有针对性的性能优化方法,提出一种基于 GPU 的并行计算性能定量分析模型。该模型分别对指令流水线、共享存储器访存、全局存储器访存进行性能建模,对并行程序进行定量分析,帮助程序员找到程序运行瓶颈,进行有效性能优化,并指明未来 GPU 体系架构的发展方向。在指令流水线建模中,首先根据功能单元数量来划分指令类型,然后测试不同 active warps 并行数量下不同指令类型的指令吞吐量。在共享存储器访存建模中,同样根据不同 active warps 并行数量来测试共享存储器的访存带宽,并考虑 bank conflicts 对程序性能造成的影响因素。在全局存储器访存建模中,根据 block 数量、每个 block 中 thread 数量以及每个 thread 对全局存储器访存事务次数来测试全局存储器的访存带宽,并考虑合并内存访问对性能造成的影响因素。最后,实验部分通过稠密矩阵乘法、三对角线性方程组求解、稀疏矩阵矢量乘对该性能分析模型进行定量分析,找出程序性能瓶颈,从而实现了较好的性能优化。

参考文献

[1] Profiler A S. ATI Stream Profiler[OL]. http://developer.amd.com

[2] Nsight N P. NVIDIA Parallel Nsight[OL]. http://developer.nvidia.com

[3] Collange S, et al. Barra: A Parallel Functional Simulator for GPGPU[C]//IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS). 2010

[4] Diamos G F, et al. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems[C]//19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010. Vienna, Austria: Institute of Electrical and Electronics Engineers Inc, 2010

[5] Ryoo S, et al. Program optimization carving for GPU computing [J]. Journal of Parallel and Distributed Computing, 2008, 68 (10): 1389-1401

[6] Liu Y, Zhang E Z, Shen X. A Cross-Input Adaptive Framework for GPU Program Optimizations[C]//23rd IEEE International Parallel and Distributed Processing Symposium, IPDPS 2009. Rome, Italy: IEEE Computer Society, 2009

[7] Meng J, Skadron K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs[C]//23rd International Conference on Supercomputing, ICS'09. Yorktown Heights, NY, United states: Association for Computing Machinery, 2009

[8] Choi J W, Singh A, Vuduc R W. Model-driven autotuning of sparse matrix-vector multiply on GPUs[C]//2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10. Bangalore, India: Association for Computing Machinery, 2010

[9] Baskaran M M, et al. A compiler framework for optimization of affine loop nests for GPGPUs[C]//22nd ACM International

Conference on Supercomputing, ICS'08. Island of Kos, Greece: Association for Computing Machinery, 2008
[10] Collange S, et al. Barra: A Parallel Functional Simulator for GPGPU. in Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)[C]//2010 IEEE International Symposium on. 2010
[11] Volkov V, Demmel J W. Benchmarking GPUs to tune dense linear algebra[C]// 2008 SC-International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008. Austin, TX, United states: IEEE Computer Society, 2008
[12] Zhang Y, Cohen J, Owens J D. Fast tridiagonal solvers on the GPU[C]//2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10. Bangalore, India: Association for Computing Machinery, 2010
[13] Goddeke D, Strzodka R. Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid [J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 23(1): 22-32
[14] Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors[C]// SC'09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing. Nov. 2009, 18: 1-11
[15] Choi J W, Singh A, Vuduc R W. Model driven autotuning of sparse matrix-vector multiply on GPUs[C]//Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010). ACM, Jan. 2010: 115-126

(上接第 15 页)
[52] Barnes J. High Integrity Software: the SPARK approach to Safety and Security [M]. London, UK: Addison-Wesley, 2006: 3-53
[53] Kaiser C, Pradat-Peyre J, Évangelista S, et al. Comparing Java, C# and Ada Monitors queuing policies: a case study and its Ada refinement [J]. ACM SIGAda Ada Letters, 2006, 26(2): 23-37
[54] Klein J. Use of Ada in Lockheed Martin for air traffic management and beyond [C]// Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada. Albuquerque, New Mexico, USA, November 2006
[55] Carlisle M. Automatic OO parser generation using visitors for Ada 2005 [C]//Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada. Albuquerque, New Mexico, USA, November 2006
[56] Dewar R. Ada 2005 & high integrity systems [C]//Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada. Albuquerque, New Mexico, USA, November 2006
[57] WG9. ISO/IEC 8652: Ada Reference Manual(Ed. 3)[S]. 2007
[58] Burns A, Wellings A. Concurrent and Real-Time Programming in Ada 2005 [M]. Cambridge, London, UK: Cambridge University Press, 2007: 451-453
[59] Sward R. Using Ada in a Service-Oriented Architecture [C]// Proceedings of the 2007 Annual ACM SIGAda International Conference on Ada. Fairfax, Virginia, USA, November 2007
[60] Barnes J. Ada 2005 Rationale: The Language-The Standard Libraries [M]. Berlin, Germany: Springer, 2008: 31-237
[61] Tokar J. 30 years after steelman, does DoD still have a software crisis? [C]//Proceedings of the 2008 Annual ACM SIGAda International Conference on Ada. Portland, OR, USA, October 2008
[62] Brosgol B. From strawman to Ada 2005: a socio-technical retrospective [C]// Proceedings of the 2008 Annual ACM SIGAda International Conference on Ada. Portland, OR, USA, October 2008
[63] Martínez P, Drake J, Pacheco P, et al. An Ada 2005 Technology for Distributed and Real-Time Component-Based Applications [C]// Reliable Software Technologies-Ada-Europe 2008, 13th Ada-Europe International Conference on Reliable Software Technologies. Venice, Italy, June 2008
[64] Sebesta R. Concepts of Programming Languages[M]. 9th International Edition. Hong Kong: Pearson Education, 2009
[65] Schonberg S. Ada 2012 Intrim Report [C]//Proceedings of the 2010 Annual ACM SIGAda International Conference on Ada. Fairfax, Virginia, USA, October 2010
[66] Sward R. The Rise, Fall and Persistence of Ada [C]//SIGAda'10 Proceeding of the ACM SIGAda Annual International Conference on SIGAda. USA, October 2010: 71-74
[67] Rosen J. Developing a Profile for Using Object-Oriented Ada in High-Integrity Systems [J]. ACM SIGAda Letters, Fairfax, Virginia, USA, 2010, 31(1): 9-10
[68] Moore B. Parallelism Generics for Ada 2005 and Beyond [C]// SIGAda'10 Proceeding of the ACM SIGAda Annual International Conference on SIGAda. USA, October 2010: 41-52
[69] Barnes J. A Brief Introduction to Ada 2012 [R]. Edinburgh, UK: The GNAT Pro Company, 2011
[70] Saez S, Terrasa S, Crespo A. A Real-Time Framework for Multiprocessor Platforms Using Ada 2012 [C]// Reliable Software Technologies-Ada-Europe 2011-16th Ada-Europe International Conference on Reliable Software Technologies. Edinburgh, UK, June 2011
[71] Chapman R, Jennings T. OOT, DO-178C and SPARK [C]//Reliable Software Technologies-Ada-Europe 2011-16th Ada-Europe International Conference on Reliable Software Technologies. Edinburgh, UK, June 2011
[72] Rosen J-P. Object Orientation in Critical Systems: Yes, in Moderation-Position Paper for the DO178C and Object-Orientation for Critical Systems Panel [C]// Reliable Software Technologies-Ada-Europe 2011-16th Ada-Europe International Conference on Reliable Software Technologies. Edinburgh, UK, June 2011
[73] Tokar J, Jones F, Black P, et al. Software vulnerabilities precluded by spark [C]// Proceedings of the 2011 Annual ACM SIGAda International Conference on Ada. Denver, Colorado, USA, November 2011
[74] WG9. ISO/IEC 8652: 2012 (E). Ada Reference Manual[M]. December 2012
[75] Schonberg E, Pucci V. Implementation of a simple dimensionality checking system in Ada 2012[C]// Proceedings of the 2012 ACM conference on High integrity language technology. New York, NY, USA, December 2012
[76] Ruiz J, Comar C, Moy Y. Source Code as the Key Artifact in Requirement-Based Development: The Case of Ada 2012[C]//17th Ada-Europe International Conference on Reliable Software Technologies. Stockholm, Sweden, June 2012
[77] Tempelmeier T. Teaching Concepts of Programming Languages with Ada[C]// 17th Ada-Europe International Conference on Reliable Software Technologies. Stockholm, Sweden, June 2012