GPU矩阵乘法的性能定量分析模型

尹孟嘉[1,2] 许先斌[1] 熊曾刚[2] 张 涛[2]

(武汉大学计算机学院 武汉 430072)[1] (湖北工程学院计算机与信息科学学院 孝感 432000)[2]

摘 要 性能评价和优化是设计高效率并行程序必不可少的重要工作,存储系统的性能高低直接影响到处理器的整体性能。利用GPGPU-Sim对GPU的存储层次结构进行了模拟,找出了SM数量与存储控制器数量之间最佳配置关系。矩阵乘法是科学计算领域中的基本组成部分,是一种具有计算和访存密集特点的典型应用,其性能是GPU高性能计算的一个重要指标。性能模型作为并行系统性能评价的新的技术解决方案,具有许多其它性能评价方法无法比拟的优势。建立了一个性能模型,模型通过对指令流水线、共享存储器访存、全局存储器访存进行定量分析,找到了程序运行瓶颈,提高了执行速度。实验证明,该模型具有实用性,并有效地实现了矩阵乘法的优化。

关键词 GPU,GPGPU-Sim,矩阵乘法,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 TP312 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.12.003

Quantitative Performance Analysis Model of Matrix Multiplication Based on GPU

YIN Meng-jia[1,2] XU Xian-bin[1] XIONG Zeng-gang[2] ZHANG Tao[2]

(School of Computer, Wuhan University, Wuhan 430072, China)[1]

(School of Computer and Information Science, Hubei Engineering University, Xiaogan 432000, China)[2]

Abstract Performance evaluation and optimization are indispensable work when designing efficient parallel program, and the performance of storage system directly affects the performance of the processor. We used GPGPU-Sim to simulate the storage hierarchy of GPU, and found out optimal quantity allocation relationship between SM and storage controller in GPU. Matrix multiplication is an essential part in the field of scientific computing, as a representative application with both computation and memory access intensiveness, and its performance is an important indicator of GPU high-performance computing. Performance model is a new technology solution as parallel systems performance evaluation, which has many advantages. In order to improve the performance of matrix multiplication, this paper proposed a quantitative performance model based on GPU. The model quantitatively analyzes instruction pipeline, shared memory access and global memory access, establishes the performance model, finds the performance bottlenecks and improves the execution speed. The experiment proves the model has practicability, and effectively realizes the optimization of the matrix multiplication algorithm.

Keywords GPU, GPGPU-Sim, Matrix multiplication, Quantitative performance analysis model, Instruction pipeline, Shared memory access, Global memory access

1 引言

"高性能计算技术"研究可分为4大分支:计算机系统的性能测评、体系结构、系统软件与语言、并行算法,主要研究大规模计算机系统的性能分析预测衡量指标、相应的基准测试分析预测方法,进一步评估计算机系统的性能,为计算机系统设计提供依据。基准程序测试可以分为测量计算机系统的总体性能的宏基准测试程序和测量计算机系统某一个特定方面性能的微基准测试程序。常见的方法有:(1)Linpack:以测试系统浮点峰值运算能力为主要目的[2];(2)SPEC:由复合了CPU、MPI、OpenMP、Web测试程序和功耗测试程序等的多组程序构成[3];(3)HPCC方法:综合考虑了计算、访存、通信与输入输出性能指标,用于高效能计算机系统的性能测试,试图取代传统的Linpack Benchmark[4]。

在高性能计算中,往往90%以上的时间都消耗在矩阵乘法上,因而它的性能优化一直是研究的热点。所以急需建立一种性能模型,用来反映各个参数对矩阵乘法性能的影响,分析计算和访存瓶颈,提高矩阵乘法性能。Yuan Nan基于cannon并行矩阵乘法算法(Cannon 1969)提出了一种多核处理器上的矩阵乘法性能模型[5]。Gunnels J A提出了一种分层

到稿日期:2015-01-29 返修日期:2015-03-15 本文受国家自然科学基金(61370092),湖北省自然科学基金(2013CFC005),湖北省中青年创新团队(T201410)资助。

尹孟嘉(1979-),女,博士,副教授,CCF会员,主要研究方向为高性能计算、体系结构,E-mail:hbyinmj@163.com;许先斌(1954-),男,博士,教授,博士生导师,主要研究方向为高性能计算、体系结构;熊曾刚(1974-),男,博士,教授,硕士生导师,主要研究方向为体系结构、计算机网络;张 涛(1980-),男,硕士,实验师,主要研究方向为体系结构、高性能计算。

计算的方法来实现矩阵乘法，该方法降低了在存储层次间搬运数据的平均开销[6]。Long G 提出一种多核结构上的矩阵乘法性能模型，在模型中给出了片上可放置的最大处理器核数目以及理论峰值性能[7]。矩阵乘法的算法非常简单，只需要三重循环，但是性能却与理论值相差甚远。CUDA 程序主要借助其并行运算能力，故应尽量减少存储器访问和通信所带来的性能瓶颈。改进的策略有 3 个：使硬件持续保持忙碌状态；增大内存吞吐率；增大指令吞吐率[8]。

功能模拟器用于预测和验证新的体系结构是否满足功能要求，它模拟目标体系结构的功能，关注数据的正确性。Baghsorkhi 等[9]提出了一个预测基于 GPU 的通用应用程序性能的分析模型。Hong 和 Kim 在 2009 年国际体系结构年会(ISCA'09)上提出了一种 GPU 性能解析模型[10]。为了评估计算机系统与时间有关的特性，研究者引入了性能模拟器，它准确地模拟每个操作所要花费的时钟周期数和数据地址信息，却并不需要模拟计算机系统中的数据[11]。目前学术界共有 4 种 GPU 模拟器，分别为 Atlia 和 Qsilver(针对传统图形流水线的)、Barra(针对 NVIDIA G80 系列芯片结构)、GPGPU-Sim(面向通用计算领域的)。

传统的性能模型只对程序进行统计分析，但并不进行性能分析。它们首先基于 GPU 的抽象架构建立一个分析模型，然后通过微基准测试程序来证明该模型。根据测试程序的执行时间以及算法的复杂度来计算该程序的持续计算时间和内存带宽，通过对实际性能的对比，就可以判断该程序的性能瓶颈属于计算密集型还是存储器限制型。而本文先设计一个微基准测试，然后观察这个测试结果，最后再基于这个结果提出一个性能预测模型，基于指令流水线、共享存储器访存、全局存储器访存进行分析。这种方法可以兼顾到架构和编程过程中影响性能的关键因素。前面的研究主要关注于对程序执行时间的预测，而本文主要关注于确立一个能够定量分析程序性能瓶颈的方法，从而实现优化。

提出并实现一种新的更加高效的算法是提升程序处理性能的重要方法。设计运行于 CPU 上的算法，往往通过复杂的控制结构利用各运算数与中间结果的关系来调整运算顺序、剪除冗余计算来达到减少运算量的目的。但这是 GPU 所不能承受的。GPU 的运算单元较 CPU 更多，有限运算量的增减对于整体性能的影响并不显著，与之相对的是 GPU 线程间通信的能力较弱，使得中间结果的共享成为运算中的瓶颈。此外，GPU 与片外存储空间的通信会造成较大延迟，故对运行于 GPU 上的程序进行优化的一个重要切入点就是在完成计算任务的前提下减少对片外存储空间的访问频率，或说是发挥每次调入数据的最大利用率，进而提升程序的计算访存比[12]。本文利用 GPGPU-Sim 对 GPU 的存储层次结构进行了模拟，找出了 SM 数量与存储控制器数量之间的最佳配置关系，提升了程序性能。

本文第 2 节介绍了矩阵乘法的并行化；第 3 节详细介绍了如何建立性能模型；第 4 节介绍性能模拟器；第 5 节结合实例对性能模型进行验证；最后是本文的工作总结。

2 矩阵乘法的并行化

矩阵乘法是一种计算量大、耗时的运算。用 CPU 进行矩阵乘法时，运算效率低下，即使用 CPU 提高单个核心性能都会遇到瓶颈。本文将运用 GPU 在 CUDA 架构上实现并优化矩阵乘法。

2.1 矩阵乘法的串行实现

矩阵乘法在算法实现上比较简单，只需进行三重循环。串行算法的实现过程是：A 矩阵中的一行和 B 矩阵中的一列对应元素进行相乘，再将相乘结果相加得到矩阵 C 中的对应元素。如果矩阵 A 的大小为 $m*k$，矩阵 B 的大小为 $k*n$，那么矩阵 C 的大小则是 $m*n$，算法如下：

```
for (i=0; i<N; i++)
for (j=0; j<N; j++)
{float ctemp=0;
  for (k=0; k<N; k++)
    Ctemp+=A[i*N+k]*B[k*N+j];
  C[i*N+j]=Ctemp;}
```

2.2 矩阵乘法的并行实现

2.2.1 矩阵乘法的基本实现

假定矩阵乘法为 $C=AB$，矩阵 A 的大小为 $m\times k$，矩阵 B 的大小为 $k\times n$，矩阵 C 的大小为 $m\times n$，那么总的计算量是 $2mnk$。

在 GPU 上，带状划分是矩阵乘法的基本实现过程，矩阵 A 的一行和矩阵 B 的一列由每个线程分别读入，再将各个对应的元素相乘，相乘的结果最后求和就是矩阵 C 中对应位置的元素，整个过程不使用共享存储器，如图 1 所示。

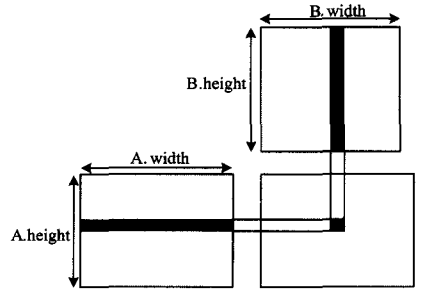

图 1 矩阵相乘中 C 中元素的计算

由图可知，在整个 kernel 中，要想完成对矩阵 C 的计算，全局存储器需要对矩阵 A 进行 $B.width$ 次读取，对矩阵 B 进行 $A.height$ 次读取。每个线程都负责 C 中一个位置的元素值的计算，for()循环完成 A 中第 X 行与 B 中第 X 列对应元素的乘加运算，假设数组在每个维度上的尺寸都是 BLOCK_SIZE 的整数倍。整个计算过程如下所示：

```
{float Cvalue=0;
  Int row=blockIdx. y*blockDim. y+threadIdx. y;
  Int col=blockIdx. x*blockDim. x+threadIdx. x;
  for(int e=0;e<A. width;++e)
  Cvalue+=A. elements[row*  A. width+e]*B. elements[e*
  B. width+col];
  C. elements[row*C. width+col]=Cvalue;}
```

在这个算法中，由于对内存的重复读取，因此内核的运算速度不尽人意，计算量是 $2mnk$ flop，而全局内存的访问量为 $2mnk$ B。若矩阵维数为 1024×1024，则矩阵相乘的计算量就有 2Gflop，随着矩阵维数的增大，运算量就变得相当大，这样就浪费了大量的时间在内存的读取上。

2.2.2 矩阵乘法的分块实现

在矩阵乘法中，矩阵 A 和矩阵 B 是只读的，至少要写入一次到 C 中。矩阵 C 中的不同部分需要用到不同的 block 来

计算，为了保证不同的 block 的独立性，各个 block 计算的部分是不重叠的。

在矩阵乘法中，对矩阵进行分块划分，使用共享存储器进行线程间的通信可以获得更好的性能[13]。矩阵 C 中一个方块 *Csub* 的计算由每个 *block* 负责，*Csub* 的一个元素由每个线程负责计算。从图 2 可以看到，*Csub* 是两个矩阵的乘：矩阵 C 中一个方块 *Csub* 的行索引与矩阵 A 的 *sub-matrix* 相同，其列索引与矩阵 B 的 *sub-matrix* 相同。这样矩阵 A、矩阵 B 被分别划分为大小为 *block_size* 的正方形矩阵，这些小正方形矩阵的乘积的和由 *Csub* 计算。实现过程如下：首先两个正方形矩阵从全局存储器加载到共享存储器，每个乘积中的一个元素由 thread 负责计算，这样乘积的结果由 thread 累积到一个寄存器中，一旦结束就写回全局存储器[14]。

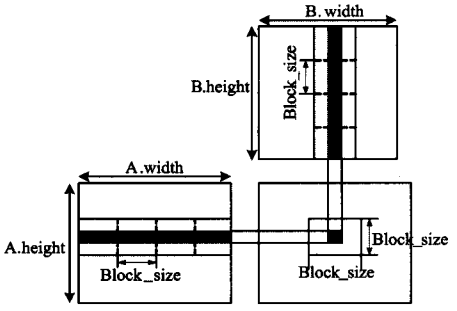

图 2　矩阵相乘中 C 中元素的计算

这样的分块划分使矩阵 A 被读 $B.width/block\_size$ 次，矩阵 B 被读 $A.height/block\_size$ 次，从而节约了大量全局存储器带宽，也利用了高速的共享存储器。*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。将 A 的一行 title 和 B 的一列 title 进行计算得到 block(0,0)，实现过程如下所示。

```
For(int a=Wa * BOLOCK_SIZE * by, b=BOLOCK_SIZE * bx; a <=
Wa * BOLOCK_SIZE * by+wA-1; a+= BOLOCK_SIZE, b+=
BOLOCK_SIZE)
{_shared_float As[BOLOCK_SIZE][BOLOCK_SIZE];
 _shared_float Bs[BOLOCK_SIZE][BOLOCK_SIZE];
AS(ty,tx)=A[a+wA * ty+tx];
BS(ty,tx)=B[B+wB * ty+tx];
_syncthreads();
For(int k=0;k<BOLOCK_SIZE;++k)
  Csub+=AS(ty,k) * BS(k,tx);
_syncthreads();
Int c=wB * BOLOCK_SIZE * by+BOLOCK_SIZE * bx;
C[c+wB * ty+tx]=Csub;}
```

这个代码实现中不存在 bank conflict 问题，因为在进行两个子块相乘再加的 for()循环中，每个 half-warp 访问的是 AS[]一行的元素，分别分布在 16 个 bank 中，对 BS[]的访问也分布在 16 个不同的 bank 中，不存在 bank conflict 问题，但是因为数组大小必须是 *block_size* 的整数倍，每个 title 会被重复取出多次，所以需要改变 *sub_matrix* 的大小。

3　性能模型建立

传统的性能模型中，程序员需要根据测试程序的执行时间以及算法的复杂度来计算该程序的持续计算时间和内存带宽。执行完后，程序员通过对比实际性能和 GPU 的峰值性能，就可以判断该程序的性能瓶颈属于计算密集型(Compute-bound)还是存储器限制型(Memory-bound)。如果该程序属于存储器限制型，可以进行的优化策略包括降低内存访存事务的次数、减少数据在 CPU 与 GPU 之间的传输次数等[15]；如果该程序属于计算密集型，则能够进行的优化策略与存储器限制型正好相反。

一个程序具有较低的计算密度，它很可能属于指令吞吐量瓶颈，而不是计算密集型瓶颈。传统的性能模型基本上不考虑片上共享内存以及 bank conflicts 对程序性能的影响，只能对性能的分析做出粗略的估计，在很多的情况下都无法确定程序的性能瓶颈。本文建立了一个性能模型，模型通过对指令流水线、共享存储器访存、全局存储器访存进行定量分析，找到性能瓶颈，进行优化。

3.1　指令流水线性能建模

指令流水线建模是根据功能单元的数量来划分指令的类型，其有效吞吐量取决于内存延迟和带宽。提高指令流吞吐量的方法一般有[16]：(1)更多地使用高吞吐量的指令；(2)优化各种存储器的访问过程，在提高访存带宽的同时降低访存延迟；(3)在 SM 中触发更多的活动线程，通过合理配比访存与计算处理，在两次访存之间调用更多的计算操作来覆盖访存延迟。

如果对处理结果的精度要求较低，那么通过使用吞吐量较低的指令，就能够有效地提升 GPU 的运算效率。CUDA 数学函数库包含了绝大多数常用的数学运算函数，所以在指令选取时应该尽可能使用执行效率较高的指令，或者选那些快速版本的函数，即在相同功能的前提下用高效指令取代那些效率较低的指令。

CUDA 程序中的控制指令(包括常见的循环、条件等控制流指令)可能严重影响程序的性能，因为这些指令可能会引起同一个 warp 内的线程跳转到不同的分支执行。为了避免在 warp 内发生分支，程序员可以使用制导语句 # pragma unroll 对特定的循环进行展开；执行控制流指令时，分支条件只与线程的 ID 有关时，可以通过修改控制条件；编译器也可以通过展开循环以及谓词执行 if、switch 语句。

访存指令包括任何读写、共享本地、全局存储器的指令。指令本身只需要 4 个时钟周期，但由于访问存储器的延迟可能非常大，因此就应该在访存期间执行足够多的算术指令来隐藏全局存储器的访存延迟，或者减少访问显存。由于寄存器文件、共享存储器中 bank 的宽度以及对显存合并访问所能达到最大带宽均为 32bit，因此将数据尺寸对齐为 32bit 可以有效地提高访存性能，即当数据尺寸较小时，将几个较小的数据合并成 32bit 的数据块；相应地，当数据尺寸较大时，则将其拆分成每线程 32bit 再进行访问。

3.2　共享存储器建模

共享存储器(Shared Memory)作为片内的高速存储器，同一 block 中所有线程都可以对其进行访问，是实现线程间通信延迟最小的方法，其访问速度与寄存器的访问速度相当[13]。由于数据在访存共享存储器时，出现了 bank conflicts，影响了访存事务的次数，因此与指令流水线相比，共享存储器需要更多的 active warps 来隐藏访存延迟。对于一个给定的普通程序，首先要根据共享存储器中的块冲突(bank conflicts)来确定该程序对其访问过程中产生访存的次数，接着依

据不同 active warp 的并行数量来评估共享存储器的访存带宽。

每个 bank 的宽度以 32bit 固定划分，相邻的 bank 中对应存放相邻的 32bit 数据，在每个时钟周期里每个 bank 可以提供 32bit 的带宽。每个 warp 大小都是 32 个线程，而一个 SM 中的共享存储器被划分为 16 个 bank。在线程对共享存储器的访问中，一个 warp 访问请求被划分为两个 half-warp 访问请求来完成，每一个 half-warp 正好对应 16 个 bank 的访问。不同存储器模块可以互不干扰地独立完成，因此可以同时完成对位于 n 个 bank 上的 n 个地址的访问，这时并行访问的有效带宽是一个 bank 时的 n 倍。当 half-warp 请求访问的多个地址位于同一个 bank 中，这时就出现了 bank conflict。由于存储器不能同时响应多个访问请求，因此当多个访问请求集中在一个 bank 中时，这些请求就只能串行完成，这就是所谓的“bank conflict”。为了解决“bank conflict”，将这一组请求划分为几次独立请求，这些独立请求不存在 conflict，有效带宽也会因此降低几倍，而不存在 conflict 的请求个数正是降低的倍数。但如果在同一个 half-warp 中的所有线程访存请求集中在同一地址时，就可以通过只产生一次广播来响应所有线程的请求。

3.3 全局存储器性能建模

全局存储器是 GPU 内最主要的存储设备，它没有 Cache，所以访问延迟非常大，很可能成为程序的性能瓶颈。制约全局存储器的访存带宽的主要因素有 3 个：block 的数量、每个 block 中 thread 的数量、每个 thread 的全局存储器访问事务次数[14]。全局存储器能否满足合并内存访问条件是影响 GPU 性能最明显的因素，所以应当遵循合并内存访问协议。共享存储器访存事务的次数可以根据是否满足合并内存访问的条件来优化确定。

如果 half-warp 内的线程访问连续地址，却没有对齐段，在设备 1.0/1.1 上，对一个元素的访问都会导致一次传输；在 1.2 及以上的设备上，则要看 half-warp 请求的所有地址是否都位于一个 128Byte 段内。当处于同一段的数据被所有线程同时访问时，只需要进行一次 128Byte 的合并访问，如图 3 所示。

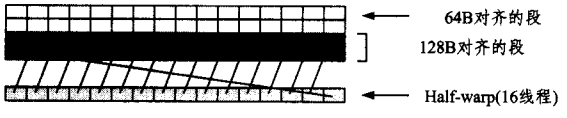

图 3 128Byte 段内的非对齐访问

如果 half-warp 访存的地址连续，但横跨两个 128Byte 段，此时会产生两次传输，如图 4 所示。

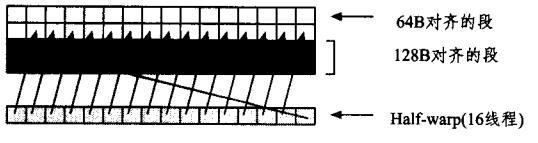

图 4 两个 128Byte 段内的非对齐访问

4 性能模拟器

性能模拟器主要用于对计算机系统的与时间有关的特性进行评估，模拟器要准确模拟出每个操作花费的时钟周期数和数据地址信息，而不需要对计算机系统中的数据进行模拟[12,17]。

GPGPU-Sim 是由 UBC 大学的 Tor M. Aamod 等人开发的针对统一架构 GPGPU 的体系结构模拟器[12]。它是基于 Simple scalar 实现的，与 Barra 相比功能更全面，代码也比较容易阅读。GPGPU-Sim 所模拟的 GPU 由 3 部分结构组成，如图 5 所示，包括：流多处理器、存储器系统以及它们之间的互联网络，这些部分与 GPU 硬件结构相对应。虚拟的互联网络将每个包含有一个流水线结构的流多处理器 SM 同存储器子系统连接起来。最新版本的 GPGPU-Sim 模拟器可以通过多个 SM 之间共同使用一个到互联网络的接口来实现。

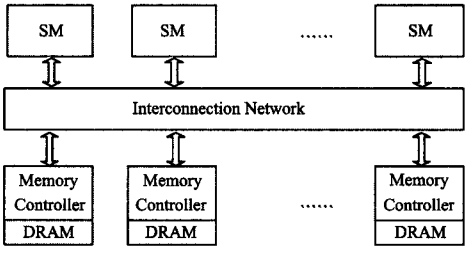

图 5 GPGPU-Sim 模拟的基本 GPU 体系结构

CUDA 程序可以通过替换 CUDA SDK 中的 common.mk 文件在 GPGPU-Sim 上实现直接编译得到可执行文件，不需要添加任何更改，GPGPU-Sim 模拟流程如图 6 所示。

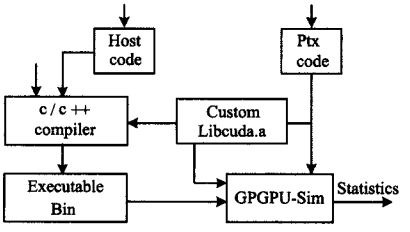

图 6 GPGPU-Sim 模拟流程

由图 6 可以看到主机-设备交互部分的代码的编译连接由 C/C++编译器完成，而包含的 kernel 函数的代码则由 NVCC 编译形成 ptx 代码。libcuda.a 是 GPGPU-Sim 的一个自定义库文件，其包含的函数的第一条语句均为模拟器定义的一个宏 GPGPUSIM_INIT，它的主要功能是完成模拟器、GPU 硬件配置的初始化及统计相关的选项等，这些都实现了 CUDA Runtime 函数库中大部分函数的重定义。GPGPU-Sim 中这个自定义的库文件可以在执行模拟的过程中调用，同时它也作为 C/C++编译器连接的对象在编译 CUDA 中的非 kernel 代码时出现。在模拟开始前，GPGPU-Sim 首先将调用并行部分 kernel 函数的地址记录下来，再启动执行程序，当前指令地址值和事先记录的地址列表中的项目匹配时，就执行并行部分，结束模拟后统计信息。

5 实验结果及分析

CUDA 程序在一台 Linux 服务器上测试，CPU 为 Intel Core 2.4GB，Linux 操作系统版本为 Red Hat 5.3，编译工具为 NVCC 和 CUDA TOOLKITS，模拟器为 GPGPU-Sim，2 块显卡：NVIDIA Geforce 9600 用来显示，NVIDIA GTX 285 用来计算。

首先利用 GPGPU-Sim 对 GPU 的存储层次结构进行了模拟，找出 SM 数量与存储控制器数量之间最佳配置关系。本实验使用矩阵乘法来进行测试。保持存储控制器的数目不变，SM 的数目逐渐增加，程序的性能也会发生变化，其变化趋势如图 7 所示。

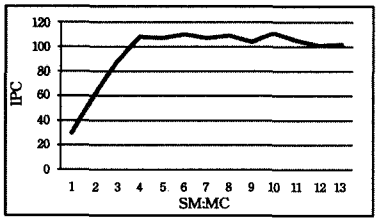

图 7　矩阵乘法性能在不同配置下的变化趋势

从图 7 可以看出，在一定范围内，维持存储控制器数目不变时，随着 SM 数目的增加，计算速度也越来越快，程序的性能也随之提高。当 SM 的数目达到一定的值时，即使再增加 SM 的数目，程序的性能也无法提升，这是因为达到存储控制器带宽所能支持的上限后，存储器无法迅速有效地为片上计算资源提供源操作数。因此，系统中 *SM* 和存储控制器的数目应当维持一定的比例值才能达到性能最优，由图 7 可知，该值应维持在 4 左右，此时二者的频率相同，并且存储控制器使用数目最少。

在稠密矩阵乘法中，本文基于 Volkov 和 Demmel[18] 提出的算法，利用了分块策略来计算矩阵乘法。其中 *sub_matrix* 的大小由程序的 *block_size* 的尺寸来确定，所以通过性能模型来确定稠密矩阵乘算法在不同的 *sub_matrix* 下的性能瓶颈。

sub_matrix 的尺寸分别为 8×8、16×16、32×32，首先测试在不同尺寸下矩阵乘法的程序性能。

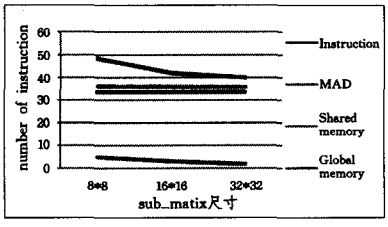

图 8　*sub_matrix* 不同尺寸时的程序性能

从图 8 可以看到，当 *sub_matrix* 的尺寸由 8×8 增加到 16×16 以及由 16×16 增加到 32×32 时，全局存储器访存事务的次数都呈下降趋势。同时也显示了当 MAD 指令数保持为常数（$\frac{matrixSize^3}{warpSize}$）时，只用尺寸较大的 *sub_matrix*，系统的所有动态指令数也相应地减少了。*sub_matrix* 的尺寸越大，越能有效地减少冗余存储器的加载次数并增加矩阵计算的密度，程序性能越好。

sub_matrix 的大小为 8×8 和 16×16 时，指令流水线为系统性能的瓶颈；*sub_matrix* 的大小为 32×32 时，系统性能瓶颈由指令流水线转移到共享存储器的访存。在实际的执行过程中，*sub_matrix* 的尺寸为 16×16 时，其系统的性能是最好的，如图 9 所示。在图 8 中，*sub_matrix* 的尺寸为 32×32 和 16×16 时有相近的共享存储器访存指令数；在图 9 中，*sub_matrix* 的尺寸为 32×32 时在共享存储器访存上花费的时间比 16×16 时要多一些。这是因为不同的 *sub_martrix* 所使用的硬件资源，32×32 使用的寄存器和共享存储器的数量比 8×8 和 16×16 要多一些；active block 的数量下降，性能下降，并行计算不足以隐藏指令流水线和共享存储器访存的延迟。

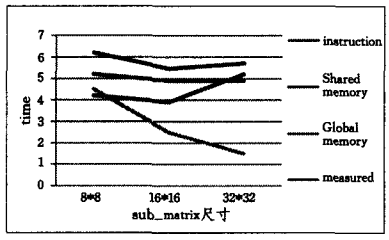

图 9　矩阵大小为 1024×1024，*sub_matrix* 大小分别为 8×8、16×16、32×32 时的性能对比

当 *sub_matrix* 的尺寸划分为 8×8 和 16×16 时，*active warps* 最大值为 32，*active block* 最大值为 8。增加 *active block* 的大小到 16 时，当其他条件不发生改变，则使得更多的并行 warps 中的指令和存储器访存吞吐量达到更高。当 *sub_matrix* 的尺寸为 32×32 时，由于 SM 中寄存器和共享存储器的数量的提升，一个 SM 将会有更多的 warps 参与计算。如果 A、B 的维度为 n，那么计算矩阵乘法需要的操作数是 $2*n*n*n$，如果利用 GFlops 来衡量矩阵乘法的执行速度，矩阵乘法的执行时间为 t，进而得出矩阵乘法的浮点运算速度为 $(2*n*n*n)/t$。

图 10 为 CPU 和 GPU 上，尺寸不同的矩阵乘法的执行速度的对比，并对本文的代码的性能与 CUBLAS 代码进行了比较，CPU 执行时调用 Intel MKL 中的 BLAS 库，GPU 上的实现采用了简单的并行方法以及优化策略以后的方法。从图中可以看出，并行后，随着矩阵规模的增大，多线程的并行性就更加显著，矩阵乘法的性能有了一定提升，且加速比与矩阵规模成正比。但是当输入规模大于 2048 时，GPU 优化算法的速度达到了峰值，并基本保持不变。这是因为当输入数据不多且没有超过 2048 时，速度比较慢，从显存到 GPU 核心读取数据的时间没有被覆盖；而当计算量比较大且输入超过 2048 时，可以掩盖大部分的访问时间。

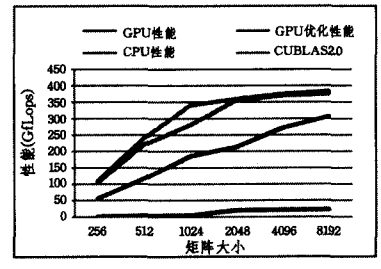

图 10　CPU\GPU 性能对比

在这里，对计算循环进行了重排，因此只有一个输入矩阵的 *sub_matrix* 需要加载到 shared memory，而不是所有输入矩阵的 *sub_matrix* 都需要加载。由于对 B 矩阵的访问不满足共享内存的对齐访问要求，因此从效率的角度考虑，应降低对 B 矩阵的访问频率。实现的过程中可以通过将 A 子矩阵 y 轴方向上的大小变为原来的 4 倍，这样 B 子矩阵从片外全局内存取到共享内存中的数据可供原来划分下的 4 个 A 子矩阵使用，单个线程完成矩阵 C 的 4 个元素值的计算。矩阵 A 访问次数不变，而矩阵 B 的访问次数为原来的 1/4，从而提高了执行效率。

结束语　GPU 的体系结构为高性能计算提供了良好的可编程性，为了获得 GPU 高性能程序设计的一般方法，本文

（下转第 22 页）

rence. 2000:349-356

[6] Henretty T, Veras R, Franchetti F, et al. A stencil compiler for short-vector simd architectures[C]// Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. ACM, 2013:13-24

[7] Kong M, Veras R, Stock K, et al. When polyhedral transformations meet SIMD code generation[J]. ACM SIGPLAN Notices, 2013, 48(6):127-138

[8] Bondhugula U, Gunluk O, Dash S, et al. A model for fusion and code motion in an automatic parallelizing compiler[C]// Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010:343-352

[9] Rosen I, Nuzman D, Zaks A. Loop-aware SLP in GCC[C]// GCC summit. 2007:131-142

[10] Nuzman D, Rosen I, Zaks A. Auto-vectorization of interleaved data for SIMD[J]// ACM SIGPLAN Notices, 2006, 41(6):132-143

[11] 何颂颂,顾乃杰,任开新. 一种面向数据密集型应用的并行程序执行模型[J]. 小型微型计算机系统,2013,34(7):1457-1461
He Song-song, Gu Nai-jie, Ren Kai-xin. Parallel Program Execution Model for Data-intensive Applications[J]. Journal of Chinese Computer Systems, 2013, 34(7):1457-1461

[12] Open64. Overview of the open64 Compiler Infrastructure[EB/OL]. http://open64.sourceforge.net, 2006

(上接第 17 页)

探索了 GPU 程序性能优化技术,对在 GPU 上进行高性能程序设计的经验进行了总结。本文利用 GPGPU-Sim 对 GPU 的存储层次结构进行了模拟,找出了 SM 数量与存储控制器数量之间的最佳配置关系,发现要使系统达到最佳性能,当二者的工作频率相同时,这一比例值为 4 时可以实现。建立了一个性能模型,通过对指令流水线、共享存储器访存、全局存储器访存的定量分析,找出了性能瓶颈,提高了执行速度。指令流水线中,指令类型是根据功能单元数量来划分的,进而对每种类型的存储器进行优化,提高指令吞吐量。对于共享存储器,主要考虑 bank conflicts 对程序性能可能产生的影响,测试其在不同 active warps 并行数量下访存的带宽。而在全局存储器访存的建模过程中,主要考虑合并内存访问对性能造成的影响,通过分析 thread 数量、block 数量、每个 thread 对全局存储器访存事务次数来设置访存带宽。最后,实验部分比较了在 CPU 和 GPU 上不同维度的矩阵乘法执行速度,并对本文的代码的性能和 CUBLAS 代码进行了比较,证明了该模型的实用性,并有效地实现了矩阵乘法的优化。

参 考 文 献

[1] Liu Jie, Chi Li-hua, Jiang Jie, et al. Performance evaluation methodology for massively parallel computer systems[J]. Computer Engineering&Science, 2013, 35(3):25-30

[2] Dongarra J J, Luszczek P, Petitet A. The LINPACK benchmark: Past, present, and future [J]. Concurrency and Computation: Practice and Experience, 2003, 15(9):803-820

[3] SPEC benchmarks [OL]. http://www.spec.org/benchmarks.html

[4] Luszczek P, Dongarra J, Koester D, et al. Introduction the HPC challenge benchmark suite[OL]. http://icl.cs.utk.edu/hpcc/pubs March

[5] Yuan Nan, Zhou Yong-bin, Tan Guang-ming, et al. High Performance Matrix Multiplication on Many Cores[OL]. http://asg.ict.ac.cn/tgm/europar09.pdf

[6] Gunnels J A, Henry G M, Van De Geijn R A. A family of high-performance matrix multiplication algorithms: Lecture Notes in Computer Science, 2001[C]// Proceedings of the International Conference on Computational Science(ICCS'01). Springer-Verlag, 2001:51-60

[7] Long Guo-ping, Fan Dong-rui, Zhang Jun-chao, et al. A performance model of dense matrix operations on many-core architectures: Lecture Notes in Computer Science, 2008[C]// Euro-Par 2008-Parallel Processing. Las Palmas de Gran Canaria, Spain: Springer Berlin Heidelberg, 2008:120-129

[8] Liang Juan-juan. Design and implementation based on GPU BLAS library [D]. Hefei: University of Science and Technology of China, 2010

[9] Baghsorkhi S S, Delahaye M, Patel S J, et al. An adaptive performance modeling tool for GPU architectures[C]// Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010). ACM, 2010:105-114

[10] Hong S, Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness[C]// Proceedings of the 36th International Symposium on Computer Architecture(ISCA 2009). 2009:152-163

[11] Yu Zhi-bin, Jin Hai, Zou Nan-hai. Computer architecture software-based simulation[J]. Journal of Software, 2008, 19(4):1051-1068

[12] Cai Jing. Analysis Key Technologies of GPGPU Architecture and Research, Extend on Simulator[D]. Changsha: National University of Defense Technology, 2009

[13] Zhang Shu, Chu Yan-li. GPU High-performance computing [M]. Beijing: China Water & Power Press, 2009

[14] Wang Zhuo-wei, Cheng Liang-lun, Zhao Wu-qing. Parallel Computation Performance Analysis Model Based on GPU[J]. Computer Science, 2014, 41(1):31-38

[15] Wang Zhuo-wei. Research on Performance Optimization for Numerical Computation based on GPU [D]. Wuhan: Wuhan University, 2012

[16] Cheng Si-yuan. Research on Performance Evaluation and Optimization for CPU-GPU Heterogeneous System [D]. Changsha: National University of Defense Technology, 2011

[17] Wai Lun-fung. Dynamic Warp Formation: Exploiting Thread Scheduling for Efficient MIMD Control Flow on SIMD Graphics Hardware [D]. University of British Columbia, 2008

[18] Volkov V, Demmel J W. Benchmarking GPUs to tune dense linear algebra, 2008[C]// 2008 SC International Conference for High Performance Computing, Networking, Storage and Analysis(SC 2008). United States: IEEE Computer Society, 2008

[19] 邹航,王华秋,黄勇. 基于 GPU 加速的彩虹表分析 MD5 哈希密码[J]. 重庆理工大学学报(自然科学版),2013,27(7):61-66
Zou Hang, Wang Hua-qiu, Huang Yong. GPU Accelerated Rainbow Tables Analysis of MD5 Has Password[J]. Journal of Chongqing University of Technology(Natural Science), 2013, 27(7):61-66