ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, Supplement, December 2010, pp.251−262

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于延迟隐藏因子的 GPU 计算模型*

袁 良[1,2,3], 张云泉[1,2+], 龙国平[1], 王 可[1], 张先轶[1,2]

[1](中国科学院 软件研究所 并行软件与计算科学实验室,北京 100190)

[2](中国科学院 计算机科学国家重点实验室,北京 100190)

[3](中国科学院 研究生院,北京 100049)

A GPU Computational Model Based on Latency Hidden Factor

YUAN Liang[1,2,3], ZHANG Yun-Quan[1,2+], LONG Guo-Ping[1], WANG Ke[1], ZHANG Xian-Yi[1,2]

[1](Laboratory of Parallel Software and Computational Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](State Key Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[3](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: zyq@mail.rdcps.ac.cn

Yuan L, Zhang YQ, Long GP, Wang K, Zhang XY. A GPU computational model based on latency hidden factor. ***Journal of Software*, 2010,21(Suppl.):251−262.** http://www.jos.org.cn/1000-9825/10026.htm

Abstract: The general purpose GPU computing technology has been successfully used to accelerator many important applications. Though researches have designed many programming models and performance models, the amount of effort required to optimize the performance of applications on GPUs is still very high. In this paper, we propose a GPU computational model based on the ability that how much an algorithm could hide the latency. The experimental results show that our model could predict three matrix multiplication algorithms well.

Key words: general purpose GPU computing; computational model; performance model; latency hidden factor; GPU optimization

摘 要: 近年来在生物计算,科学计算等领域成功地应用了 GPU 加速计算并获得了较高加速比.然而在 GPU 上编程和调优过程非常繁琐,为此,研究人员提出了许多提高编程效率的编程模型和编译器,以及指导程序优化的计算模型,在一定程度上简化了GPU上的算法设计和优化,但是已有工作都存在一些不足.针对GPU低延迟高带宽的特性,提出了基于延迟隐藏因子的 GPU 计算模型,模型提取算法隐藏延迟的能力,以指导算法优化.利用 3 种矩阵乘算法进行实测与模型预测,实验结果表明,在简化模型的情况下,平均误差率为 0.19.

关键词: GPU 通用计算;计算模型;性能模型;延迟隐藏因子;GPU 性能优化

近年来,GPU 通用计算技术发展迅速,从硬件角度,GPU 提供了较之 CPU 更高的浮点计算能力和带宽,例如 nVidia 最新发布的 Fermi 架构 GPU[1]双精度浮点计算峰值为 515 Gflop/s,单精度计算峰值为 1.03 Tflop/s,带宽为 144 GB/s,而 Intel i7-975 处理器[2]峰值为 55.36 Gflop/s,QPI 带宽 25.6 GB/s,四核达到 100GB/s,但由于 CPU 核

* Supported by the National High-Tech Research and Development Plan of China under Grant Nos.2006AA01A125, 2009AA01A129, 2009AA01A134 (国家高技术研究发展计划(863)); 核高基资助项目(2009ZX01036-001-002); 中国科学院知识创新工程重大项目课题(KGCX1-YW-13); 财政部国家重大科研装备研制项目(ZDYZ2008-2)

Received 2010-06-15; Accepted 2010-12-10

之间的内存是 NUMA 架构,因此 CPU 内存带宽不能简单与 GPU 对比.由此可见,GPU 在数据并行方面的计算能力更强,越来越多的应用[3]也开始使用 GPU 作为 CPU 的协处理器加速计算,但是由于 GPU 属于众核架构,在 GPU 上编程调优是比较困难的工作.

因此从软件角度,厂商提供了的不同编程模型,例如 AMD 的 Brook++[4],nVidia 的 CUDA[5]以及最新跨平台的 OpenCL[6],即使存在这些编程模型,在 GPU 上编写和优化代码仍然是非常繁琐的任务,许多研究工作开发了不同编程模型及编译器,降低编程难度.由于本文设计的计算模型基于 CUDA,因此本文只介绍 CUDA 相关研究工作.文献[7]设计了 OpenMP 到 CUDA 的源到源编译器,并在 OpenMP 端和 CUDA 端运用不同优化方法改进程序性能.文献[8]提出了计算模型和数据模型两种类似 OpenMP 的编译制导语句,降低了 GPU 编程难度.CuPP[9]集成 CUDA 到 C++框架中,消除了 CUDA 中一个主机线程只能管理一个设备的限制,并提供了传址调用接口等功能简化了 GPU 编程.CUDA-Lite[10]提供一个源到源翻译器,输出优化的 CUDA 代码,编程时只需管理 global 内存,加入注释标明在 GPU 上运行的数据结构.

但是,虽然这些编程模型以及固定的优化方法可以简化编程难度,但是在 GPU 优化程序仍然是非常困难而繁琐工作,例如,文献[8]没有利用数组私有化,循环交换和循环分块等优化技术,需要手工优化;一些应用优化参数的搜索空间巨大,搜索最优值非常耗时.为此,研究人员设计了不同的计算模型,用以指导和简化 GPU 优化.文献[11,12]是面向早期的 GPU 性能和计算模型.文献[13]是 nVidia 提供的基于 Excel 表的工具,能够计算 CUDA 线程的占用率,即实际活动 warp 数目与 SM 上能够容纳最大 warp 数目比值,CUDA 编译器通过最小化线程使用的寄存器数目来增加活动线程块数目,以此来提高占用率.文献[14]提出了两个性能标准,效率=1/(每个线程执行的指令数×线程个数),效率越高越好.利用率=(每个线程执行的指令数/同步或访存个数)×((WTB−1)/2+(BSM−1)WTB),第 1 项表示在一次同步或者访存操作之前执行的指令个数,第 2 项表示一个 SM 中独立的 warp 个数,WTB 是线程块内的 warp 数,BSM 是一个 SM 中的 block 数.文献[15]为优化 SpMV 算法提出了性能模型,通过测量不同参数配置的离线测试程序获得实际运行时间,得到线程启动时间,和每次迭代的延迟隐藏参数.文献[16]对算法在多 GPU 上对 3 种 GPU 架构建立可扩展性模型,对 GPU 执行时间,PCI-E 传输时间,RAM 和 DISK 时间建模,分析预测多 GPU 执行时间.文献[17]文章定义两个并行度,MWP(memory warp parallelism),在一个 warp 执行一条访存指令到该 warp 可以执行下一条指令期间,一个 SM 可以访存的 warp 个数,这段时间用 Mem_cycles 表示.定义 CWP(computation warp parallelism),在 Mem_cycles 内可以执行的 warp 数加 1.根据带宽是否为瓶颈,用峰值带宽和程序访存模式计算 MWP.当 CWP 大于 MWP 时,访存是程序的主要开销,反之,当 MWP 大于 CWP 时,计算是程序的主要开销.文献[18]文章对 block 和 warp 调度器根据轮询(round-robin)模式进行模拟来计算运行时间,只有遇到数据不在寄存器或者同步等阻塞操作时才进行线程切换,此时根据计算是否能隐藏数据访问延迟,更新运行时间.以此模型对不同循环展开因子计算预测时间,找出最优值.文献[19]类似文献[18],面向 iterative stencil loops(ISL)问题,此类问题通常通过建立镜像区,增加冗余计算,来优化通信和同步,文献[19]对 GPU 建立模型,分析不同镜像区大小的性能,计算最优大小.文献[20]改进了文献[14,17]没有考虑 warp 内控制流,共享存储 bank 冲突和 SIMD 流水线延迟的缺陷,对 CUDA 线程模型 3 个层次进行分析并提出相应指标计算方法评估算法在每一层中特性:在线程级提出 ILP,表示在寄存器一级共享数据以及隐藏延迟;在 warp 级开发 DLP,保证全局内存合并访问,减少共享内存 bank 冲突;在线程块一级开发线程间数据共享.

这些模型虽然能对某些算法较准确预测性能,但仍存在一些不足之处.文献[17]指出更高占用率[13]并不代表更高性能,且两个标准只有在内存带宽不是瓶颈的时候才有用[14].有些模型[15]不是通过建立模型计算 kernel 时间.文献[19]面向特定问题,其模型具有局限性.文献[17]没有考虑存储层次,假设单一的带宽和延迟.我们的模型中共享内存是重要的因素,延迟和带宽参数都不同.在计算 MWP 时将带宽受限和延迟受限两种情况分开,而对大量线程而言,第 1 个线程限制为延迟,而后面的线程,由于轮询调度模式,带宽更为重要,因此本文统一考虑带宽和延迟,文献[18]计算时间的算法是一个简单的模拟器,对循环展开计算更为方便,比模型更为精确,但是只是对循环展开因子做判断,模型没有给出指导改进算法瓶颈.只是面向循环展开因子,没有考虑带宽影响.模型

没有具体参数,只是最终预测时间的比较,而本文首次提出延迟隐藏因子概念,为优化程序提供了改进思路.文献[20]更关注 SIMD 计算延迟的隐藏,而本文更关注访存隐藏.将内存等待分为带宽限制和延迟限制两种,而本文模型统一考虑带宽和延迟.此外,除了文献[18]以外,大部分已有模型都通过平均化来获得模型参数,例如,文献[17]中 MWP-1 个 warp 可以隐藏 CWP=Mem_cycles/Comp_cycles+1,而本文模型基于 PTX 指令集,精确对程序建模,不在程序层次进行平均化求参数.由于 GPU 的设计理念是大量并行多线程隐藏延迟,硬件架构提供高带宽而不是低延迟,因此,本文提出延迟隐藏因子,用来描述程序能够隐藏延迟的能力,并基于延迟隐藏因子设计 GPU 计算模型.

本文第 1 节简要介绍 GPU 计算.第 2 节提出基于延迟隐藏因子的 GPU 计算模型,包含连续计算和线程块同步时两种延迟隐藏因子.第 3 节用矩阵乘算法代码验证模型正确性.第 4 节是对全文的总结,并给出下一步工作.

1 GPU 通用计算

由于本文计算模型基于 nVidia GPU,本节简要介绍 nVidia 硬件架构和 CUDA 编程模型.nVidia GPU 由可扩展多线程流处理器(stream multiprocessor,简称 SM)阵列构成,每个 SM 由若干计算内核,共享存储和寄存器堆构成.GPU 执行结构为单指令多线程(SIMT)模式,SIMT 与 SIMD 的不同之处在于后者指定了数据宽度,而 SIMT 中的每一个线程可以执行不同的代码路径.SM 将线程块中每一个线程映射到一个计算内核,每一个标量线程独立的在自己的指令空间和寄存器状态运行.

GPU 的设计理念是大量并行多线程隐藏延迟,所以 GPU 硬件设计目标是高带宽和线程调度切换低开销而不是低延迟,GPU 带宽比 CPU 高,并且在一个 SM 上配置了大量寄存器,使得 GPU 可以通过线程间调度隐藏延迟,因此 GPU 优化主要优化访存,合理利用带宽,使得 warp 间访存对齐以及利用共享内存降低带宽需求.GPU 的线程切换低开销,较高访存延迟和较小共享内存和寄存器空间都限制了 GPU 上应用以细粒度并行为主.

1.1 CUDA简介

nVidia 提供 CUDA 编程模型,对 GPU 硬件进行抽象.CUDA 包括 3 个基本概念:线程组织层次,共享内存和同步.CUDA 线程模型又分为 3 个层次:最高层是 grid,grid 内线程执行相同 kernel 函数,可组织成一维或者二维形式;中间层是线程块,每个线程块内线程可以组织成三维,同一线程块内线程可以通过 SM 上共享内存共享数据,对数据进行原子操作,并通过__syncthreads()原语进行同步,同一 grid 中不同线程块中的线程不能互相通信和同步;最后,硬件将线程分为 warp 执行以提高访存性能,并以 warp 为单位来进行创建,管理,调度,执行,一个 warp 内线程共享和执行相同的指令.

CUDA 线程执行的 kernel 是 C 函数,在调用时按照指定 grid 维数和线程块维数并行执行.CUDA 线程可以通过内置变量 gridDim,blockIdx,blockDim 和 threadIdx 获得线程 ID.Kernel 启动时内置变量和函数参数存放在共享内存中.

每个线程有私有本地内存,每个线程块有共享内存,所有线程可以访问全局内存.共享内存和全局内存都有一些规则,当满足这些规则时,能够没有冲突的合并进行访问,提高访存带宽利用率.

2 GPU 计算模型

2.1 GPU模型的简化

本文模型只考虑 nVidia GPU,模型基于 PTX 指令集,不采用 decuda,而是通过在编译选项中加入-ptx 生成的 ptx 文件指令流和 NVCC 显示的资源利用信息来计算模型参数.虽然 NVCC 生成的 ptx 文件有大量冗余信息,例如在寄存器一层没有重用等,因此 ptx 文件通常需要与循环展开因子相当数目的寄存器,但是我们的模型只通过 ptx 文件进行指令流分析,而资源使用情况则根据编译输出信息获得.由于函数参数存放在共享内存中,我们忽略基于参数计算的地址等信息的时间.

本文的模型主要面向规则算法,具有规则访问模式,这使得模型可以不考虑 warp 内指令流分支,不考虑共

享内存访问冲突,不考虑全局内存合并模式访问,简单假设对共享内存和全局内存的所有访问都满足最优访问模式要求,简化了模型的设计,尽管如此,这种假设可以容易的通过扩展访存延迟和带宽数值进行修正.

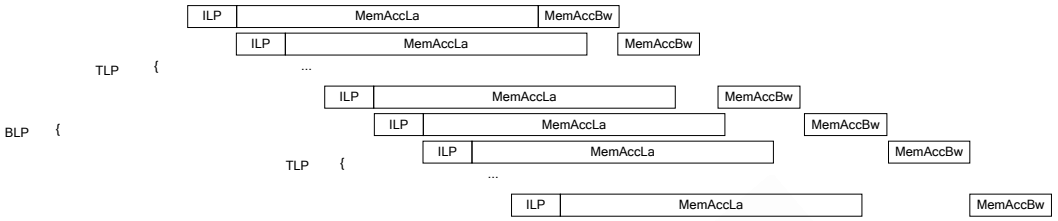

Fig.1 ILP, TLP and BLP on GPU

图 1 GPU 的 ILP,TLP 和 BLP

2.2 GPU计算模型

GPU 上运行一个 warp,无论是需要的数据仍在传送中,或者是需要进行同步操作,SM 都需要进行 warp 切换和调度,只要执行 warp 遇到需要等待数据指令,GPU 就进行切换,文献[23,24]都用循环展开数据相关指令验证了这一点.我们用线程运行时的这些切换点将线程切分为若干基本块,即一个 SM 以线程基本块为执行单位,到达切换点时,SM 执行下一个 warp,假设线程有 NBB(number of basic blocks)个基本块.根据文献[18],线程的调度是轮询模式,由于所有 SM 共享全局内存带宽,不能严格做到平均分配带宽,因此 warp 间不能严格同步,使得计算同一线程块内 warp 间和线程块间隐藏因子较为复杂,通过文献[18]中假设,我们可以认为同一 SM 上所有 warp 为基本块一级的并发运行,且在切换时,warp 指令流最多只差一个基本块,这样使得我们的模型计算延迟隐藏因子较为简单.

由于一个 warp 内线程同步运行,我们将 GPU 上运行的一个 warp 视为 CPU 中一个线程,即本文所指 GPU 线程 ILP(instruction-level parallelism,指令级并行)为一个 GPU 上 warp 内部指令级并行性,而 GPU 线程 TLP(thread-level parallelism,线程级并行性)指一个 SM 上属于同一线程块的活动 warp 间并行性,由于一个 SM 可以有多个线程块,我们将线程块间并行称为 BLP(thread block-level parallelism,线程块级并行性).

模型整体思想类似于 RAM(h)[25]模型,从数据角度观察 kernel,模型区别计算与访存,分别计算两者时间开销,并为访存延迟建模,而计算时间用简单的硬件吞吐量累加计算,GPU 模型与 RAM(h)区别在于:(1) 加入延迟掩藏因子参数,用以描述 GPU 开销较小的 warp 切换这一特性;(2) RAM(h)而不考虑计算开销,这在不进行 ILP 优化时是适用的,然后由于 GPU 寄存器较大特性,开发 ILP 是获取并行性和隐藏延迟的重要方法,因此我们的模型考虑计算开销.

由于资源和硬件限制,程序在 GPU 上的运算需要几次迭代[15,17,18],每次迭代都是相同的代码不同的数据,为此,通过每次每个 SM 执行的线程块数量 BLP 和总共需要执行的线程块数量 NB(number of blocks),可以计算出迭代次数 $RepNum=\frac{NB}{BLP\times NSM}$,其中 NSM(number of SMs per GPU)为 GPU 上 SM 数目.

对一次迭代计算时间 *TimeOneRep*,从一个 thread 角度计算 ILP 和运行时间与延迟[17],而从 TLP 和 BLP 角度考虑线程间的切换以及线程块内同步.计算延迟均为等待访存数据的延迟引起,因此计算延迟已包含在访存延迟之内,计算能隐藏访存延迟,因此,计算访存延迟因子 *LatencyHidden* 乘以访存延迟,加上计算耗时,即为一次迭代最终时间,而整个程序用时 *TimeTotal* 为迭代次数乘以 *TimeOneRep*.

$$TimeOneRep=\sum_{i=0}^{NBB}(SMComputeTime_i+MemAccessLatency_i\times LatencyHidden_i),$$

$$TimeTotal=TimeOneRep\times RepNum.$$

模型将在 SM 上执行的指令耗时记为 *SMComputeTime*,包括计算指令,访存发射指令和同步指令,均看为计算指令,用指令的吞吐量来计算在一个基本块在 SM 上的运行时间 *SMComputeTime*,*Throughputs* 为计算指令吞吐量的函数,输入为指令,输出为指令吞吐量,因此得到第 i 个基本块的计算公式:

$$SMComputeTime_i = \sum_j Throughput(ComputeInstruction_j) + \sum_k Throughput(MemAcdinstruction_k).$$

一般情况下,例如访存指令(*MemAccInstruction*),吞吐量等于 warp 内线程个数除以每个 SM 的核数量,即 $Throughput = \frac{ThreadsPerWarp}{CoresPerSM}$,在 CUDA 1.x 上为 4 cycle/warp,在 CUDA 2.0 上为 2 cycle/warp.对于几种特殊的指令,例如需要访问共享存储的 MAD 和 ADD 计算指令(compute instruction)的吞吐量为 6 cycle/warp.

Table 1 Hardware limitations in CUDA compute capability

表 1 CUDA 计算能力的硬件限制

Compute capability	1.0	1.1	1.2	1.3	2.0
x- or *y*- dimension of a grid	64K	64K	64K	64K	64K
x- or *y*- dimension of a block	512	512	512	512	1 024
z- dimension of a block	64	64	64	64	64
Threads/Warp	32	32	32	32	32
Threads/Thread block	512	512	512	512	1 024
Warps/Multiprocessor	24	24	32	32	48
Threads/Multiprocessor	768	768	1 024	1 024	1 536
Thread blocks/Multiprocessor	8	8	8	8	8
Shared memory/Multiprocessor	16K	16K	16K	16K	48K
Register file size	8K	8K	16K	16K	32K
Register allocation unit size	256	256	512	512	64
Shared memory bank	16	16	16	16	32
Local memory/thread	16KB	16KB	16KB	16KB	512KB
Allocation granularity	Block	Block	Block	Block	Warp
Shared memory allocation unit size	512	512	512	512	128

2.3 连续计算时延迟隐藏因子

延迟隐藏因子 *LatencyHidden* 的计算包括 3 个方面,warp 内线程隐藏因子,线程块 warp 间隐藏因子和线程块间隐藏因子:

- warp 内线程隐藏因子 *ILP*,*ILP* 代表一个 warp 连续计算用时,包括线程内两个相邻 warp 调度点间的计算指令和访存指令发射耗时,对于每一个基本块 *i*,都计算对应的 ILP_i.
- 同一线程块内 warp 间隐藏因子考虑同步的影响和 *TLP*,由于本文一个 warp 视为一个线程,*TLP* 的限制即为 warp 个数限制.*TLP*=min{GPU 硬件限制 warp 数目,kernel 资源限制 warp 数目},GPU 硬件限制包括 GPU 单个 SM 上 warp 和 threads 数目限制,kernel 资源限制包括寄存器和共享内存限制单个 SM 上 warp 数目限制.线程块同步的硬件在计算隐藏因子时用到.表 1 列出了 CUDA 不同设备计算能力的资源限制.
- 线程块间隐藏因子考虑 *BLP*,代表一个 SM 上同时并发执行的 block 数目,类似于 *TLP*,*BLP*=min{GPU 硬件限制 block 数目,kernel 资源限制 block 数目},GPU 硬件限制包括 GPU 单个 SM 上 thread block,warp 和 threads 数目限制,kernel 资源限制包括寄存器和共享内存限制单个 SM 上 warp 数目限制.

由于 GPU 线程切换开销较小,*ILP* 与 *TLP*,*BLP* 的关系更为紧密,在不考虑同步的情况下,*ILP*,*TLP* 和 *BLP* 可以同等对待,正是如此,文献[20]中的计算才将 *ILP*×*TLP*×*BLP* 作为 kernel 内计算延迟隐藏的分母,但是没有区分 *TLP* 与 *BLP*,而这在计算下一节中的同步时延迟隐藏因子时要区分.由于一个线程块可以进行同步操作,因此,模型针对是否存在同步操作分为两种延迟隐藏因子,下面分别予以讨论.没有同步操作的连续基本块 i 的延迟隐藏因子分为四种情况讨论.首先给出几个定义,$MemAccLa_i$ 表示访问数据所在存储层次的延迟,$MemAccBw_i$ 为一个 warp 运行第 i 个基本块时由于带宽所引致的延迟,即为一个 warp 内数据总量除以一个 SM 分到的带宽,$MemAccBw_i = \frac{DataSize_i}{BandwidthPerSM}$,$DataSize_i$ 代表第 i 个基本块访存数据量,类似文献[17],本文假设带宽在 SM 之间平分. $BandwidthPerSM = \frac{Bandwidth}{NSM}$.

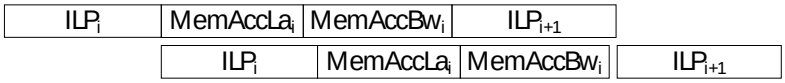

Fig.2 Case 1 A: ILP_i, $ILP_{i+1} \geqslant MemAccBw_i$, $ILP_{i+1} \geqslant ILP_i$

图 2 第 1 种情况 A: ILP_i, $ILP_{i+1} \geqslant MemAccBw_i$, $ILP_{i+1} \geqslant ILP_i$

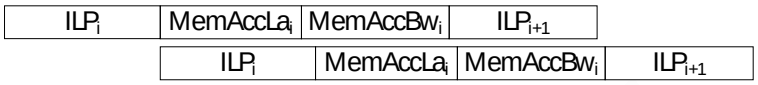

Fig.3 Case 1 B: $ILP_{i+1} \geqslant MemAccBw_i$, $ILP_{i+1} \delta ILP_i$

图 3 第 1 种情况 B: $ILP_{i+1} \geqslant MemAccBw_i$, $ILP_{i+1} \delta ILP_i$

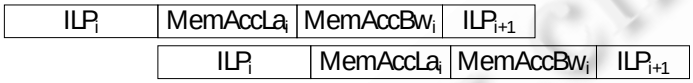

Fig.4 Case 2 A: $ILP_{i+1} \delta MemAccBw_i$, $ILP_i \geqslant MemAccBw_i$

图 4 第 2 种情况 A: $ILP_{i+1} \delta MemAccBw_i$, $ILP_i \geqslant MemAccBw_i$

Fig.5 Case 3 A: ILP_i, $ILP_{i+1} \delta MemAccBw_i$

图 5 第 3 种情况 A: ILP_i, $ILP_{i+1} \delta MemAccBw_i$

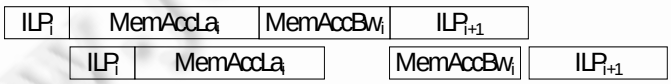

Fig.6 Case 4 A: $ILP_{i+1} \geqslant MemAccBw_i$, $ILP_i \delta MemAccBw_i$

图 6 第 4 种情况 A: $ILP_{i+1} \geqslant MemAccBw_i$, $ILP_i \delta MemAccBw_i$

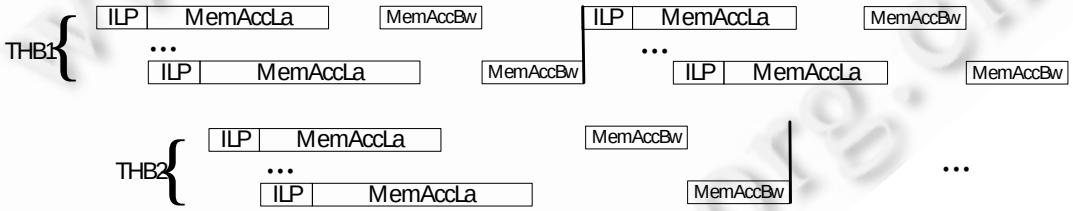

Fig.7 Case 5: Synchronization

图 7 第 5 种情况:同步时的情况

2.3.1 情况 1

如果 ILP_i 与 ILP_{i+1} 均大于 $MemAccBw_i$,则每个 $warpMemAccLa_i$ 与 $MemAccBw_i$ 两段时间没有间隔,$MemAccBw_i$ 与 ILP_{i+1} 两段时间间隔位 $\mathrm{Max}\{0,(ILP_{i+1}-ILP_i)\}$,图 2 和 3 分别是 $ILP_{i+1} \geqslant ILP_i$ 和 $ILP_{i+1} \delta ILP_i$ 的情况,但是无论是出现哪种情况,此段间隔是由于第 i+1 个基本块的前 j–1 个 warp 的计算延迟引起的,已经计算在 *SMComputeTime* 里,在这里应该忽略,因此 SM 上第 j 个 warp 中第 i 个基本块的延迟隐藏因子为

$$LatencyHiddenCon1_{i,j} = \mathrm{Max}\left\{0,1-\frac{ILP_i \times (TLP \times BLP - j) + (j-1) \times (ILP_{i+1} - \mathrm{Max}\{0,(ILP_{i+1} - ILP_i)\})}{MemAccLa_i + MemAccBw_i}\right\}$$

$$= \mathrm{Max}\left\{0,1-\frac{ILP_i \times (TLP \times BLP - j) + (j-1) \times \mathrm{Min}\{ILP_{i+1},\ ILP_i\}}{MemAccLa_i + MemAccBw_i}\right\}.$$

2.3.2 情况 2

如果 ILP_{i+1} 小于 $MemAccBw_i$,ILP_i 大于 $MemAccBw_i$,则每个 warp 的 $MemAccLa_i$ 与 $MemAccBw_i$ 两段时间没有间隔,$MemAccBw_i$ 与 ILP_{i+1} 两段时间也没有间隔,此时 ILP_{i+1}δILP$_i$,属于情况 1 的特例,如图 4 所示,因此 SM 上第 j 个 warp 中第 i 个基本块的延迟隐藏因子为

$$LatencyHiddenCon2_{i,j}=\mathrm{Max}\left\{0,1-\frac{ILP_i\times(TLP\times BLP-j)+(j-1)\times ILP_{i+1}}{MemAccLa_i+MemAccBw_i}\right\}.$$

2.3.3 情况 3

如果 ILP_i 与 ILP_{i+1} 均小于 $MemAccBw_i$,如图 5 所示,则每个 warp 的 $MemAccLa$ 与 $MemAccBw$ 两段时间间隔($MemAccBw_i$–ILP_i),SM 上第 j 个 warp 中第 i 个基本块的延迟隐藏因子为

$$LatencyHiddenCon3_{i,j}=\mathrm{Max}\left\{0,1-\frac{ILP_i\times(TLP\times BLP-j)+(j-1)\times ILP_{i+1}}{MemAccLa_i+MemAccBw_i+(j-1)\times(MemAccBw_i-ILP_i)}\right\}.$$

2.3.4 情况 4

如果 ILP_{i+1} 大于 $MemAccBw_i$,ILP_i 小于 $MemAccBw_i$,则每个 warp 的 $MemAccLa$ 与 $MemAccBw$ 两段时间间隔($MemAccBw_i$–ILP_i),$MemAccBw_i$ 与 ILP_{i+1} 两段时间间隔位(ILP_{i+1}–$MemAccBw_i$),如图 6 所示,同情况 1,因为这段时间已经计算在 $SMComputeTime$ 里,不能隐藏延迟,因此 SM 上第 j 个 warp 中第 i 个基本块的延迟隐藏因子为

$$\begin{aligned}LatencyHiddenCon4_{i,j}&=\mathrm{Max}\left\{0,1-\frac{ILP_i\times(TLP\times BLP-j)+(j-1)\times(ILP_{i+1}-(ILP_{i+1}-MemAccBw_i))}{MemAccLa_i+MemAccBw_i+(j-1)\times(MemAccBw_i-ILP_i)}\right\}\\&=\mathrm{Max}\left\{0,1-\frac{ILP_i\times(TLP\times BLP-j)+(j-1)\times MemAccBw_i}{MemAccLa_i+MemAccBw_i+(j-1)\times(MemAccBw_i-ILP_i)}\right\}.\end{aligned}$$

2.3.5 合并 4 种情况

通过分析上述 4 种情况延迟隐藏因子的计算公式,利用 Max 和 Min 操作,可以通过下式合并 4 种情况,即在连续计算时的隐藏因子计算公式统一为

$$LatencyHiddenCon_{i,j}=\mathrm{Max}\left\{0,1-\frac{ILP_i\times(TLP\times BLP-j)+(j-1)\times\mathrm{Min}\{ILP_{i+1},\mathrm{Max}\{ILP_i,\ MemAccBw_i\}\}}{MemAccLa_i+MemAccBw_i+(j-1)\times\mathrm{Max}\{0,MemAccBw_i-ILP_i\}}\right\}.$$

2.4 线程块同步时延迟隐藏因子

因为线程块内线程可以共享数据,在这一层次利用共享内存,所以本文以一个线程块来考虑同步问题,由图 7 所示,线程块内同步操作带来两方面影响:线程块 1 要在所有线程块内 warp 数据到达之后进行一次同步,因此线程块内所有线程的访存结束时间一致,所以延迟隐藏因子分母包含 Max{0,$MemAccBw_i$–ILP_i}项的乘数要向上取整到线程块内线程数目(NT)的倍数;同一线程块内的 warp 不能在相邻基本块间隐藏访存延迟,因此延迟隐藏因子分子的包含 ILP_{i+1} 项的乘数要向下取整到线程块内线程数目的倍数.此时延迟隐藏因子为

$$LatencyHiddenSyn_{i,j}=\mathrm{Max}\left\{0,1-\frac{ILP_i\times(TLP\times BLP-j)+\left\lfloor\frac{(j-1)}{NT}\right\rfloor\times\mathrm{Min}\{ILP_{i+1},\mathrm{Max}\{ILP_i,MemAccBw_i\}\}}{MemAccLa_i+MemAccBw_i+\left(\left\lceil\frac{(j-1)}{NT}\right\rceil-(j-1)\%NT\right)\times\mathrm{Max}\{0,MemAccBw_i-ILP_i\}}\right\}.$$

2.5 模型计算公式

实际上,$LatencyHiddenCon$ 代表连续计算时由 ILP,TLP 和 BLP 共同隐藏延迟的因子,$LatencyHiddenSyn$ 表示由于线程块内同步,BLP 隐藏延迟的因子.

由于 CUDA 1.x 上 GPU 以线程块为调度粒度,所以在 CUDA 1.x 上假设每个线程在最后语句有一条隐式同步语句,在计算最后一个基本块延迟隐藏因子时使用 $LatencyHiddenSyn$.而由于 CUDA 2.0 上 GPU 以 warp 为调度粒度,所以同步粒度为 warp 内线程个数,由于目前 CUDA GPU 均为 32 threads/warp,我们利用 $LatencyHiddenCon$ 计算 CUDA 2.0 上最后一个基本块的延迟隐藏因子.而 ILP,$MemAccBw$ 和 $MemAccLa$ 的下标

计算为模加,即 $ILP_{NB+1}=ILP_1$,最终计算公式如下:

$$TimeOneRep=\sum_{i=0}^{NBB}(SMComputeTime_i+MemAccessLatency_i\times LatencyHidden_i)$$

$$=\sum_{i=0}^{NBB}\left(SMComputeTime_i+\sum_{j}^{TNT}MemAccessLatency_{i,j}\times LatencyHidden_{i,j}\right).$$

3 实验结果

本文实验平台为 nVidia Tesla C1060,包含 30 个 SM,240 个核,频率为 1.30 GHz.本文使用类似文献[17]中 micro-benchmark 以及相关文献测得模型所需要的 GPU 数据,测得数据如表 2 所示.由于矩阵大小不同执行时间差距很大,我们根据矩阵规模统一将计算时间转化为计算峰值.图 9 给出了本节用到的缩率语.

本节使用 3 个矩阵乘算法来验证模型预测正确性.Naïve 算法每个线程计算一个矩阵 C 元素,线程间没有合作,每次均从全局内存取矩阵 *A* 和 *B* 数据,算法如图 10 所示,我们取线程块大小为 8×8 和 16×16 两种类型对不同矩阵规模进行测试和性能预测.CUBLAS 1.1[22]算法一个线程块内线程协作加载矩阵 *A* 和 *B* 子块数据到共享内存,在线程块一级重用数据,算法如图 12 所示,同样,我们取线程块大小为 8×8 和 16×16 分别测试.文献[23]矩阵乘算法每个线程计算多个矩阵 *C* 元素,并只在共享内存中重用矩阵 *B* 的子块,而从全局内存协作加载所需要的矩阵 *A* 数据,算法如图 13 所示.我们选取矩阵规模为 2 的整数幂次加上小的临边,例如我们取 2^i+32 矩阵大小,这可以避免 warp 内访存操作落入同一内存 bank,如何根据访存位置对模型进行优化,以及根据 ptx 自动进行分析,将是下一步工作.

由于篇幅限制,我们只给出 Naïve 矩阵乘算法的模型流程,如图 11 所示,其余两个算法过程类似 Naïve 和 CUBLAS 的对应流程.根据上节所述模型相关公式,开发了对应的延迟隐藏因子求解器.通过 NVCC 编译器获得算法资源使用情况,基于此利用 Occupancy[13]计算线程,线程块和 warp 数目,进一步得到 *TLP* 和 *BLP*.ptx 优化器对 NVCC 输出的 ptx 代码进行优化,去掉冗余部分,输出优化的 ptx 文件传给 ptx 分析器,针对指令流,提出模型所需要的流程,并计算相应的 *ILP*,如图 8 所示,然后相关结果统一交给延迟隐藏因子求解器,输出模型预测时间,其结果直接表示在结果中.模型测试用例以及相关中间计算结果在表 3 中给出.

Table 2 The specifications of Tesla C1060

表 2 Tesla C1060 相关参数

Throughput(cycle/warp) Latency(cycle)	CUDA 1.3 Tesla C1060		
	Th.	La.	Bw
Access global memory	4	550	102
Access shared memory	4	36	50
MAD using REG	4	24	N.A.
MAD using REG & SM	6		
MUL using REG	4	24	
MUL using REG & SM	6		
ADD using REG	4	24	
ADD using REG & SM	6		

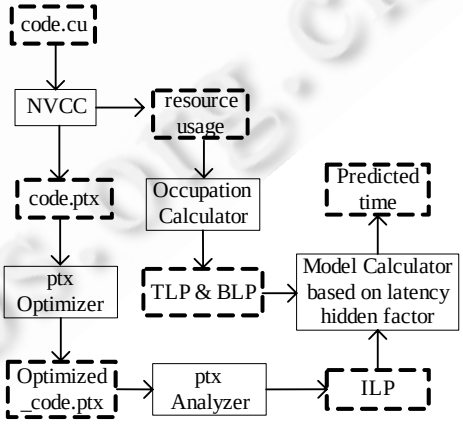

Fig.8 Model computing process

图 8 计算模型处理流程

图 14~图 16 分别给出了 3 种算法在不同配置下的实测运行峰值和模型预测峰值,可以看到,本文提出的模型,在性能趋势上较好的符合了实际情况,平均误差率为 0.19,虽然误差率较大,但是这也误差来源没有使用 decuda 获得精确资源使用情况和指令流,对 warp 切换时机没有精确模拟,并且没有区分合并模式访存与非合并模式访存区别,而是用同一的带宽和延迟计算,而这些模型所的简化对于同一算法来说是相同的,因此模型的预

测曲线与实际测量曲线在趋势上一致.模型在预测 Naïve 矩阵乘时,由于分块大小为 16 时,half-warp 在访问矩阵 *A* 和 *C* 时均为合并模式,所以我们预测的结果和实测结果较为接近,而在分块大小为 8 时,由于 half-warp 需要访问 *A* 和 *C* 的两行,不能进行合并模式访存,因此实测性能低于预测结果.

Table 3 The characteristics of benchmarks

表 3 使用的测试用例

MMM	Naïve	CUBLAS 1.1	Volkov's[23]
Size of C's block, sto ed	1× , reg	32× 2, reg	64× 6, reg
Size of A's block, sto ed	1× , reg	32×3 smem	64 ., reg
Size of B's block, sto ed	1× , reg	32×3 smem	16×1 , smem
Threads/Thread block (ThB)	8×8, 16×16	8×8, 16×16	16×4
reg/Thread block (ThB) (4 Bytes)	10, 10	13, 13	30
smem/Thread (Bytes)	48, 48	560, 2 096	1 156
Constant (Bytes)	8, 8	4, 4	80
Occupation (thread blocks)	8, 4	8, 4	2
Size of matrix (WA	64…4 96 (+32)	256…8 92 (+16)	64…8 92 (+32)
Number of threads (T	64…4 96 (+32)	256…8 92 (+16)	4…5 2 (+2)
Number of basic blocks (NBB)	WA+2	3 (WA/BS)+1	

CI=Computing Instruction

GA=Global memory Access

SA=Shared memory Access

BS=Block Size

WA=Width of matrix *A*

Fig.9 Summary of instructions

图 9 模型指令缩写总结

Compute some values

Loop WA times

Load element of *A* to register

Load element of *B* to register

Update *C*;

Write *C* to global memory

Fig.10 Naïve MMM pseudocode

图 10 简单矩阵乘法伪码

$ILP_1=CI_n+GA_2$

$DateSize_1=Double_3$

$ILP_2=\ldots=ILP_{WA+1}=MAD_1+GA_2$

$DateSize_2=\ldots=DateSize_{WA+1}=Double_2$

$ILP_{WA+2}=GA_1$

$DateSize_{WA+2}=Double_1$

Fig.11 Naïve MMM model process

图 11 简单矩阵乘模型处理过程

Compute some values

Loop (WA/BS) times

Load *A*s to shared memory;

Load *B*s to shared memory;

syncthreads;

Loop *B*s times

Update *C*s;

syncthreads;

Write *C*s to global memory

Fig.12 CUBLAS 1.1 pseudocode

图 12 CUBLAS 1.1 矩阵乘伪码

Compute some values

Loop WA/16 times

Loop 4 times

Load Bs to shared memory;

syncthreads;

Loop 16 times

Load element of *A* to register

Loop 16 times

Load *B*s to register

Load *B*s to register

Update *C*s;

syncthreads;

Loop 16 times

Write *C*s to global memory

Fig.13 Volkov's pseudocode

图 13 Volkov 矩阵乘算法

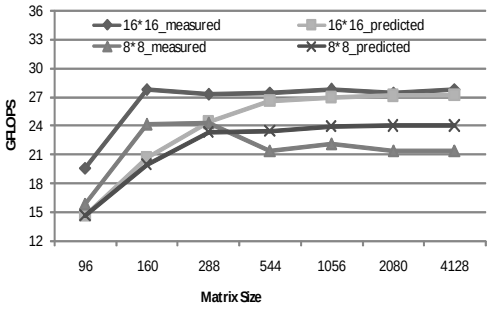

Fig.14 Results of the Naïve algorithm

图 14 简单矩阵乘算法结果

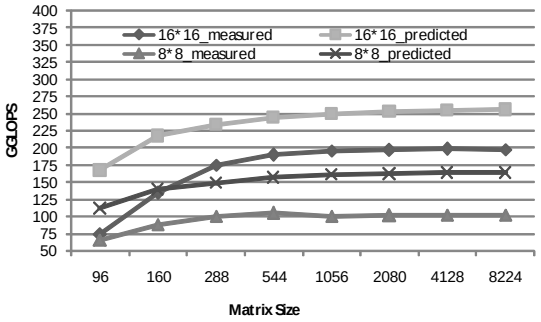

Fig.15 Results of the CUBLAS 1.1 algorithm

图 15 CUBLAS 矩阵乘算法结果

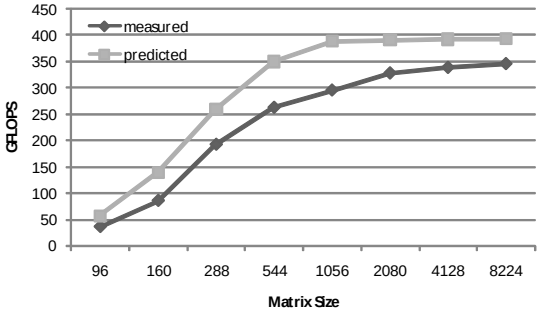

Fig.16 Results of the Volkov's algorithm

图 16 Volkov 矩阵乘算法结果

尽管存在较大误差,但计算模型的本质意义不在于精确预测,而在于提供指导算法优化的指标和方法,从实验结果我们可以看出,本文提出的模型在算法运行时间趋势上与实测一致.

4 结论和下一步工作

GPU 提供了更高浮点计算能力及带宽,但是利用 GPU 编程和优化比较繁琐和困难,因此编程模型和计算模型一直是 GPU 计算的研究热点.基于 GPU 延迟隐藏能力,本文提出面向 CUDA 程序的延迟隐藏因子.根据是否描述线程块内同步,分为连续计算延迟隐藏因子以及同步时延迟隐藏因子两种,用以描述算法隐藏延迟能力.基于提出的因子,本文设计了新的 GPU 计算模型.实验结果表明新的模型能够较好预测算法性能.

本文模型针对每个线程的每个基本块计算延迟隐藏因子,在精确性和简单性之间进行了平衡选取和设计,由于精确的模型目的为理解硬件处理过程,而模糊的模型的目的在于指导算法的改进,所以,在下一步工作中,我们将分别从精确性和模糊性两个方面对本文模型进行改进:将对一个 kernel 统一计算一个延迟隐藏因子,描述算法整体隐藏延迟能力,为算法设计提供更具指导性的性能指标;在 CUDA 2.0,例如 Fermi 上验证模型将是下一步工作;开发自动版本的 ptx 优化器和分析器;使用 decuda 分析,加入访存分析,进一步改进模型分析结果.

References:

[1] Next Generation CUDA Architecture. http://www.nvidia.com/object/fermi_architecture.html

[2] Support for the Intel_Core_i7 Processor Extreme Edition. http://www.intel.com/p/en_US/support/highlights/processors/corei7ee

[3] CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html

[4] Brook+. http://developer.amd.com/gpu/ATIStreamSDK/pages/Documentation.aspx

[5] CUDA 3.0 Downloads. http://developer.nvidia.com/object/cuda_3_0_downloads.html

[6] OpenCL. http://www.khronos.org/opencl/

[7] Lee S, Min S-J, Eigenmann R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. 2009.

[8] Han TD, Abdelrahman TS. hiCUDA: A high-level directive-based language for GPU programming. In: Proc. of the 2nd Workshop on General Purpose Processing on Graphics Processing Units. Washington, 2009. 52−61.

[9] Breitbart J. Cupp—a framework for easy cuda integration. In: IPDPS 2009: Proc. of the 2009 IEEE Int'l Symp. on Parallel & Distributed Processing. Washington, 2009. 1−8.

[10] Ueng SZ, Lathara M, Baghsorkhi SS, Hwu WW. Cuda-Lite: Reducing GPU programming complexity. In: LCPC 2008. LNCS 5335, 2008. 1−15.

[11] Govindaraju NK, Larsen S, Gray J, Manocha D. A memory model for scientific algorithms on graphics processors. In: Proc. of the 2006 ACM/IEEE Conf. on Supercomputing. 2006.

[12] Liu WG, Wittig WM, Schmidt B. Performance predictions for general-purpose computation on GPUS. In: ICPP 2007: Proc. of the 2007 Int'l Conf. on Parallel Processing. Washington, 2007. 50.

[13] CUDA Occupancy Calculator. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[14] Ryoo S, Rodrigues CI, Stone SS, Baghsorkhi SS, Ueng SZ, Stratton JA, Mei W. Program optimization space pruning for a multithreaded GPU. In: CGO 2008: Proc. of the 6th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. New York: ACM, 2008. 195−204.

[15] Choi JW, Singh A, Vuduc RW. Model-Driven autotuning of sparse matrix-vector multiply on GPUs. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. 2010.

[16] Schaa D, Kaeli D. Exploring the multiple-GPU design space. In: IPDPS 2009: Proc. of the 2009 IEEE Int'l Symp. on Parallel & Distributed Processing. Washington: IEEE Computer Society, 2009. 1−12.

[17] Hong S, Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: Proc. of the 36th Annual Int'l Symp. on Computer Architecture. 2009.

[18] Murthy CS, Ravishankar M, Baskaran MM, Sadayappan P. Optimal loop unrolling for GPGPU programs. In: 2010 IEEE Int'l Symp. on Parallel & Distributed Processing (IPDPS). IEEE, 2010. 1−11.

[19] Meng JY, Skadron K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUS. In: ICS 2009: Proc. of the 23rd Int'l Conf. on Supercomputing. New York: ACM, 2009. 256−265.

[20] Baghsorkhi SS, Delahaye M, Patel SJ, Gropp WD, Hwu WW. An adaptive performance modeling tool for GPU architectures. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. 2010.

[21] Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu WW. Optimization principles and application performance evaluation of a multithreaded GPU using Cuda. In: PPoPP 2008: Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New York: ACM, 2008.

[22] CUBLAS 1.1. http://developer.nvidia.com/object/cuda_2_1_downloads.html

[23] Volkov V, Demmel JW. Benchmarking GPUs to tune dense linear algebra. In: Proc. of the 2008 ACM/IEEE Conf. on Supercomputing. 2008.

[24] Cui X, Chen YF, Mei H. Improving performance of matrix multiplication and FFT on GPU. In: ICPADS 2009: Proc. of the 2009 15th Int'l Conf. on Parallel and Distributed Systems. Washington: IEEE Computer Society. 2009. 42−48.

[25] Zhang YQ. DRAM(*h*): A parallel computation model for high performance numerical computing. Chinese Journal of Computers, 2003,26(12):1660−1670 (in Chinese with English abstract).

附中文参考文献:

[25] 张云泉.面向高性能数值计算的并行计算模型 DRAM(*h*).计算机学报,2003,26(12):1660−1670.

袁良(1984－),男,河北保定人,博士生,主要研究领域为并行算法,并行编程模型.

张云泉(1973－),男,博士,研究员,博士生导师,主要研究领域为高性能计算及并行数值软件,并行计算模型,并行数据库,海量数据并行处理.

龙国平(1982－),男,博士,助理研究员,主要研究领域为计算机体系结构.

王可(1981－),博士,副研究员,主要研究领域为并行计算.

张先轶(1983－),男,助理研究员,主要研究领域为 GPU 等异构加速部件的性能优化技术,并行数值软件.