2014 年 5 月 第 5 期
Vol. 35 No. 5 2014

图广度优先搜索算法面向图形处理器的优化方法研究

刘 谷[1],安 虹[1,2],李小强[1],吴石磊[1]

[1](中国科学技术大学 计算机科学与技术学院,合肥 230027)
[2](中国科学院 计算机系统结构重点实验室,北京 100080)
E-mail: gliu@mail.ustc.edu.cn

摘 要: 近年来,图形处理器(GPU)以其丰富的计算资源和低廉的成本逐渐在高性能计算领域取得一席之地,对于具有规则访存特性的并行程序具有明显的加速作用.但是以图广度优先搜索(BFS)算法为代表的某些不规则应用,在图形处理器上性能表现平平.为了解决不规则程序在图形处理器上的性能瓶颈问题必须分析其行为特征,面向特定体系结构提出有针对性的程序优化方法.本文通过分析图广度优先搜索算法的在 GPU 上的并行性模式,访存特性以及工作负载,提出了基于并行性剖析与反馈的计算资源重配置方法,动态队列的层次优化方法,以及线程级负载平衡方法.实验表明以上优化方法能够显著提高图广度优先搜索算法为代表的不规则程序在 GPU 上的性能.

关 键 词: 图形处理器;图广度优先算法;不规则程序;并行性剖析;优化方法
中图分类号: TP303 **文献标识码:** A **文 章 编 号:** 1000-1220(2014)05-1074-06

Study on Optimizing Techniques of Breadth-first Search Algorithm on Graphic Processing Unit

LIU Gu[1], AN Hong[1,2], LI Xiao-qian[1], WU Shi-lei[1]
[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](Key Laboratory of Computer System and Architecture, Chinese Academy of Science, Beijing 100080, China)

Abstract: Recently, graphic processing unit (GPU) has played an important role in high performance computing for its abundant computing resource and relative low cost. It has a tremendous advantage in accelerating parallel programs with memory-access regularity. However, some irregular applications such as bread-first search (BFS) can only achieve moderate performances on GPU for architectural restriction. Therefore irregular program needs to be analyzed and optimized specifically on GPU. In this paper, we analyzed the parallelism pattern, memory-access features and work-load distribution of BFS on GPU, and accordingly present some optimization techniques, including the re-configuration method based on parallelism pattern profiling, the hierarchical queue management, and the balanced thread-level workload distribution. Experiment shows that these optimization techniques can remarkably improve the BFS and other irregular program on GPU.

Key words: graphic processing; breadth-first search algorithm; irregular program; parallelism profiling; optimizing techniques

1 引 言

近年来,图形处理器(GPU)在高性能通用计算领域逐渐受到人们的重视.GPU 不仅具有丰富的计算资源和较低廉的成本,同时具有完整易用的编程开发环境,例如 Nvidia 公司提供的统一计算设备架构(CUDA).许多通用领域的应用程序已经在 GPU 平台上进行移植并取得极高的加速比,其主要原因在于 GPU 芯片采用超并发多线程的 SPMD 体系结构,使得大量的硬件线程能够并发执行,一方面开发了程序中的数据级并行性,另一方面能够掩盖部分访存开销.GPU 的结构特性尤其适合用于加速具有规则访存特性的程序,例如稠密矩阵乘法,FFT 等.

但是,近年来研究人员也注意到某些不规则应用在 GPU 上并未获得同样的性能提升,有些程序甚至比 CPU 处理器上的串行程序更慢[1].这些程序包括稀疏矩阵乘法,无结构网格计算,有限自动机以及图算法[2]等,它们在科学计算领域中同样占据重要的地位.在 GPU 上加速困难的程序通常具有某些共同的行为特征,例如基于索引或指针的数据表示形式,运行时需维护队列或堆栈等动态变化的工作集,迭代算法的串行语义明显且控制流复杂,程序可用的数据并行性模式随输入数据改变等.总的来说,这些程序在 GPU 上的性能瓶颈问题是其不规则访存特性造成的[3].这些特性限制了不规则程序在 GPU 上的性能提升,同时也对 GPU 的结构发展提出了新的问题.

图算法中的广度优先搜索(BFS)就是一类典型的不规则程序,它被广泛地应用于集成电路设计,人工智能,计算机辅助设计以及社会网络系统等领域,具有较强的性能需求和研究意义,在很多高性能计算平台上有并行实现版本[6,10-12].但是已发表的论文显示,GPU 对 BFS 的性能提升效果并不显著,针对 GPU 结构的程序优化手段是主要的应付策略.例如,

收稿日期: 2013-02-19 基金项目: 国家"八六三"高技术研究发展计划重大项目子课题项目(2005CB321601, 2006AA01A102-5-2)资助;国家"八六三"高技术研究发展计划项目(2009AA01Z106)资助. 作者简介: 刘 谷,男,1983 年生,博士研究生,研究方向为微处理器体系结构;安 虹,女,1963 年生,博士,教授,研究方向为计算机体系结构、微处理器体系结构;李小强,男,1986 年生,博士研究生,研究方向为微处理器体系结构;吴石磊,男,1987 年生,硕士研究生,研究方向为微处理器体系结构.

Harish[5,9] 提出 BFS 算法基于线程任务划分的并行实现,较早的提出了在 GPU 上开发 BFS 内在并行性的基本方法. Deng[4] 在其文章中提出使用 GPU 上的稀疏矩阵算法库来实现 BFS 算法迭代,但是引入的冗余计算使得时间复杂度超过了串行算法,难以推广到较大规模的图. Hong[13] 提出的虚拟 warp 划分方法从软件上提供了较好的负载平衡能力,但是降低了 GPU 计算资源的利用率,同样不适合于大规模图的计算. Luo[7] 从经典串行 BFS 算法出发,提出了时间复杂度相同的并行实现,使用两个队列交替存放中间节点解决了部分随机访存问题,但无法解决不规则图带来的负载平衡问题.

本文分析了 BFS 在 GPU 上造成性能瓶颈问题的几个主要因素,包括并行性模式,访存模式以及负载平衡问题,有针对性的提出了三种优化方法: 基于并行性剖析与反馈的计算资源重配置方法,动态队列的层次优化方法,以及线程级负载平衡方法. 经过实验评测,证实了我们提出的方法能够有效地解决 BFS 在 GPU 上的性能瓶颈问题,并分别与 CPU 平台上的串行程序和 GPU 上的未优化的并行版本进行了比较,获得了 2-10 倍的加速. 这些方法同样能够应用到其他不规则程序上.

本文后续章节组织如下: 第 2 章介绍相关背景知识,第 3 章分析 BFS 程序的行为特征并提出三种优化方法,第 4 章进行实验说明与性能评测,第 5 章总结全文.

2　背景知识

2.1　GPU 体系结构与 CUDA 编程模型

当前工业界主要图形处理器厂商推出的 GPU 芯片已经可以集成超过 40 亿个晶体管,峰值计算能力接近 1TFLOPs. 以 Nvidia 的 Tesla C2050 芯片为例,它主要面向通用计算领域,拥有 14 个流多处理器(Stream Multi-processor,简称 SM) ,每个 SM 中又有 32 个独立执行指令的流处理器(又称 Cuda core) . 较多的计算单元允许上千个硬件线程同时执行,其中每连续 32 个线程称为一个 warp,作为一个调度单位发射并执行. 每个 warp 被调度到同一个 SM 上,共用同一套指令发射逻辑,指令以 SIMD 的方式在 32 个流处理器上锁步执行,这样既提高了硬件设计的可重用性,又减少了控制逻辑所占芯片面积. Tesla C2050 拥有 3GB 的片外全局存储器. 当一个 warp 中的线程访问一段连续地址空间时称为对齐的访存操作,硬件可以将这些访存操作整合成一次访存事件进行处理. 这种对齐访存方式充分的利用了带宽资源,有效的提高了性能. 另外,在每个 SM 中具有一块可配置成共享存储器的片上 cache,通过它可以实现线程间的快速通信. 共享存储器的访问速度比全局存储器快 100 倍左右.

统一计算设备架构(CUDA) 是由 Nvidia 推出的一种基于 C 语言扩展的编程模型,业已获得广泛使用. CUDA 程序由运行在 CPU 上的主机代码和运行在 GPU 上的核心函数组成,其中核心函数通过主机代码进行配置和调用,在 GPU 上以多线程的方式执行. 核心函数的线程采用了 grid 和 block 两层结构,每个核心函数对应一个线程 grid,其中包含有若干个 block. 每个 block 是个包含相同数目线程的线程组,并在执行时分配到同一 SM 上执行,block 之间的执行是完全独立的,无需保证执行次序. CUDA 模型隐藏了线程的执行和调度细节,程序员需要做的只是在主机代码中配置核心函数的 grid/block 线程数目即可. 这个特点有利于提高 CUDA 程序的性能,程序员能够根据核心函数的性质和规模指定线程配置参数. 但是,在缺乏动态创建线程机制的情况下,程序员只能静态的为核心函数指定线程参数,无法实时的响应并行性的动态变化,特别是对于并行性模式复杂的不规则程序而言,必将造成程序执行效率低下.

2.2　BFS 在 GPU 上的程序行为分析

图广度优搜索算法是一类经典的图算法. 对于一个给定图 G(V, E) ,从源节点 s 出发采用广度优先策略对图中节点进行遍历,产生一棵以 s 为根包含所有可到达节点的遍历树. 在大部分应用中所关心的是每个节点在遍历树上与 s 的距离,即节点对应的层次值 level,而不关心具体的遍历过程. 在本文中我们讨论的 BFS 程序对于给定的输入图 G(V, E) 和源节点 s,输出 G 中每个节点的层次值向量 level [V] . 传统的 BFS 算法利用一个队列来保存当前待访问的节点集. 队列初始为{ s} ,每次从队列中出队一个元素,并将其未被访问到的节点加入队尾,直至队列为空,其时间复杂度为 O(V + E) . 某些 BFS 并行算法使用其他冗余数据结构来提高并行性,但是通常增加了时间复杂度. 这种做法在数据规模超过一定程度时将产生大量冗余计算从而掩盖由并行性带来的性能. 我们的 BFS 并行化算法中仍然保持了 O(V + E) 的时间复杂度不变,以适应所有规模上的数据输入.

在 [5] 中提出了 BFS 在 GPU 上的基本实现,使用了多个线程来同时处理当前层次队列中的所有节点,每个线程访问一个节点并更新下一层队列. 为了在访问队列时防止并发访问冲突通常采用锁机制来进行同步. 这种实现具有以下 3 个方面的行为特征.

2.2.1　复杂多变的并行性模式

BFS 的并行性主要来源于多个线程间同时访问队列中的元素并更新下层队列,并行度与当前层次队列的规模相关. 从 s 开始,每一层次的队列规模各不相同,而且不同类型的输入图其并行度变化趋势也各不相同. 图1显示了几种不同类型的

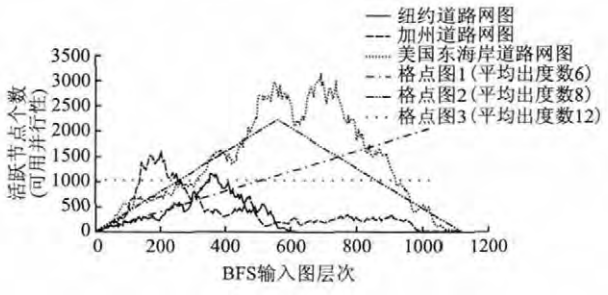

图1　不同类型输入图的并行性模式剖析

Fig. 1　Parallelism pattern profiling of different types of graphs

图的并行性模式剖析,可以看到不规则的道路网图各个层次上的节点数呈现不规则变化的趋势,而即使是规则的格点图在不同的平均出度条件下其变化趋势也是不同的. 这造成了 BFS 的并行性具有动态性,难以静态预测. 这与 GPU 的静态指定线程配置参数的特征产生了矛盾. 如果采用固定线程配置,线程数目太少会造成并行性开发不充分引起的性能损失,太多

又会降低程序执行效率浪费计算资源和功耗. 只有引入一种动态自适应的线程配置方法才能较好的解决这个问题.

2.2.2 中间队列的动态随机访存特性

BFS 使用队列保存中间数据,是各个线程都要进行更新访问的关键数据结构. 更新队列的线程以及更新位置是动态确定的,造成了队列访存的动态性与随机性,这与 GPU 的对齐访问方式形成了矛盾. 随机的队列写操作不能够自动的实现访存地址的有序化,每次写操作都会引发独立的访存事件,造成了带宽资源的浪费. 我们需要优化队列的结构,使用 GPU 对齐访存方式来更新下层队列.

2.2.3 线程负载不平衡特性

每个线程的任务规模与所访问节点的邻居节点数目有关. 对于结构不规则的输入图而言,可能会出现某些节点的邻居节点数大大超过平均出度值的情形. BFS 在处理这种图时会产生严重的负载不平衡,程序执行时间由最慢线程形成的关键路径所决定. 我们需要对不同输入图提供负载平衡检测机制,利用空闲线程资源来分摊处理不平衡的工作集.

针对上述 BFS 在 GPU 上的程序行为特征分析,我们提出的优化方法能够分别解决这些行为特征带来的性能瓶颈问题.

3 图广度优先搜素算法在 GPU 上的优化方法

针对 BFS 在 GPU 上的程序行为特征,我们提出了以下三种有针对性的优化方法,包括基于可用并行性反馈的线程配置方法,动态队列的层次化优化方法以及不规则输入图中的负载平衡优化方法. 这些优化措施能够有效的解决 BFS 算法在 GPU 平台上的性能瓶颈问题,充分发挥出 GPU 的性能优势,减轻不规则程序行为造成的性能损失.

3.1 基于可用并行性反馈的线程配置法方法

由于 BFS 算法中的无组织数据并行性[5] 具有复杂多变的动态特征,使用固定的计算资源配置可能会造成性能或资源利用率的下降. 如果分配的线程计算资源小于当前的可用并行性,那么需要经过多次迭代才能处理完一个层级的活跃节点; 如果分配的线程数超过最大的可用并行性,虽然每个层级只需一次迭代就能完成,但是对于并行性较少的层级而言会造成计算资源的浪费. 为此,我们设计了一种灵活线程配置方案 FlexBFS,能够根据可用并行性反馈信息合理的分配计算资源,从而减少迭代次数并且不会引起资源利用率的降低.

该实现的基本思想是: 在处理每个层级的活跃节点之前,依据上一层级的可用并行性反馈信息分配线程计算资源; 为此,需要在进行传播操作时用一个计数器来记录下一层级的活跃节点个数,并将此信息反馈给下一次迭代. 算法实现的伪代码如下所示.

```
FlexBFS ( G , src)
//输入图为 G 和源节点 src
{
  Workset < = { src} ; //src 加入工作集作为初始节点
  while( Workset 不为空) {
    获取 Workset 的大小 size;
    分配 size 个计算线程调用核心函数
    FlexBFS_kernel( Workset,NewWorkset) ;
    Workset < = NewWorkset; //工作集更新
  }
}
FlexBFS_kernel( Workset,NewWorkset)
//BFS 的核心函数,每个线程处理 Workset 中的一个节点
{
  parallel for i in Workset{
    若 i 有未访问过的邻居节点 j;
    将 j 加入 NewWorkset 中,更新 size 计数;
  }
}
```

为了验证 FlexBFS 的效果,我们在 GPU 平台上用 CUDA 进行了实现. 输入图采用顶点索引列表和边列表的数据结构表示,该表示能够减少输入图数据的存储空间需求. 核心函数每次处理一个层级,其中反馈的并行性信息使用一个全局变量来保存. 执行 BFS 核心函数时,每当有一个线程将未访问过的邻居节点加入新队列时,我们就对计数器加一. 为了防止不同线程把相同的邻居节点重复加入新队列,我们在判断是否访问过该节点时,使用原子操作来读取当前的访问数组 visited 值. 只有第一个访问线程会修改 visited,并加入新队列,后续的线程将会发现该节点已经被访问过,从而避免重复计数.

3.2 动态队列的层次化优化方法

BFS 算法中的动态队列访存是影响性能的一个关键因素. 作为保存中间工作集的数据结构动态队列需要进行频繁的更新操作. 由于动态队列是一个多线程共享的结构,需要使用同步机制来实现互斥访问,以保证更新操作的正确性. 通常的并行实现会使用一个锁来控制对动态队列的读写,如果某个线程需要对动态队列进行更新,那么必须先获得这个锁; 当更新结束时,再释放锁,是其他线程能够获得更新队列的机会. 使用锁机制会大大降低线程间的并发性,所有线程必须排队获取这个锁,并顺序的来更新动态队列,其性能下降至与串行程序相当.

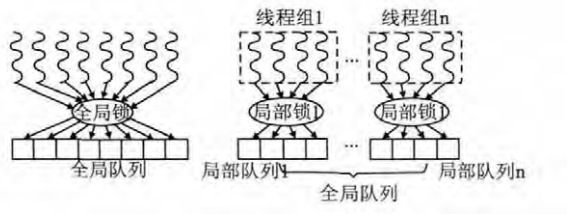

图 2 层次化的动态队列组织

Fig. 2 Hierarchical queue organization

为了解决这个问题,我们可以采用层次化的动态队列来增加更新队列操作的并发度. 我们首先将线程分成若干组,每组共享一个局部队列用来保存更新的节点,每个局部队列都有各自的互斥锁. 当某个线程需要更新局部队列时,只需要获得该局部队列的锁就能执行访存操作了. 虽然每个组内的线程仍然是顺序获取锁来更新局部队列,但不同组的线程就能够并行的执行更新操作了,因为它们更新的是不同的局部队列. 最后只需将局部队列首尾相连就得到下一层级的工作集队列了. 这里主要利用了工作集队列中所有节点都是属于同一层级的,因此相互之间的次序是可以替换的. 尽管连接局部队列有一定开销,但同时增加了动态队列访存的并行性,整体

上的性能获得了提高.

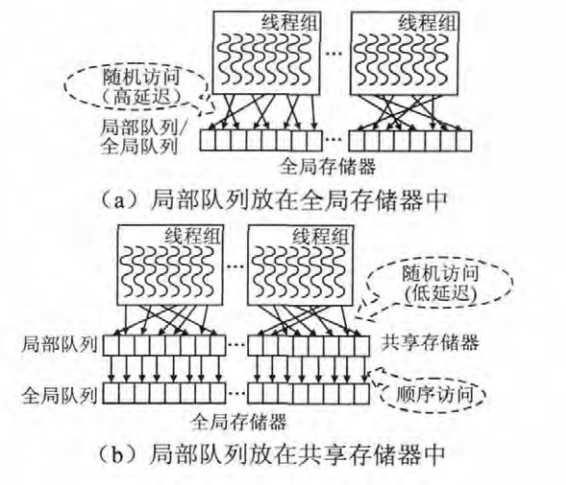

图 3　GPU 上两种局部队列的实现

Fig. 3　Two implementation of local queue on GPU

在 GPU 上实现层次化队列时可以将局部队列放在延迟较低的片上存储共享存储器中,而不是放在全局存储器中. 这是因为每个线程更新的位置在全局地址上并不一定连续,可能会引起随机分散的地址分布. GPU 对于连续地址访存模式能够利用合并访存事务而获得更高的带宽,线程组访存地址随机分布的话就不能利用这个有利特性. 如果将局部队列放在片上共享存储器中,虽然地址在线程组内仍然是随机分布的,但是因为访存延迟很小,开销并不大. 而且在局部队列拷贝到全局队列时能够充分利用顺序访存的优势,减少了整体的访存开销. 在图 3 中,我们比较了两种队列实现方法的访存事务数,从而体现了访存开销的降低.

3.3　不规则输入图中的负载平衡优化方法

在 BFS 的并行迭代算法中,为了开发活跃节点之间的粗粒度并行性,工作集中的每个活跃节点都被分配给一个线程来进行处理. 每个线程都要将未访问过的邻居节点写入下一层级的队列,因此工作量与活跃节点的邻居数目是相关的. 在处理某些不规则图时会出现个别高度数节点的情况,它们的邻居数超过平均节点度数很多,造成各个线程的负载不平衡,其它线程需要等待处理高度数节点的线程,程序执行的关键路径将由这些特殊的高度数节点所决定.

在处理这类具有高度数节点的不规则输入图时,邻接区域内的细粒度并行性已经上升为主要矛盾. 仅使用开发粗粒度并行的线程划分方式已经不再适合,需要重新考虑细粒度并行性的开发. 在 SPMD 的执行模型下,必须设计另一种核心函数来单独处理高度数节点. 为此,我们提出了基于特殊队列的负载平衡优化方法. 其伪代码如下:

```
BFS_kernel( Workset, NewWorkset, SpQueue)
{
  parallel for i in Workset{
    if i 是高出度节点
      将 i 加入 SpQueue;
    else{
      if i 有未访问邻居 j
        将 j 加入 NewWorkset;
    }
  }
  for j in SpQueue{
    parallel for k in ( j 的邻居节点集合)
      if k 未被访问
        将 k 加入 NewWorkset;
  }
}
```

关于对高度数节点的判断,通常是与图的平均度数作比较,如果超过一定阈值,例如超过平均度数 5 倍,则可以看作是一个高度数节点. 其中图的平均度数 d 可以由图的顶点数 V 和边数 E 求得,如公式(2)所示,其中 d_i 表示第 i 个节点的度数:

$$E = \frac{1}{2}\sum_{i}^{V} d_i = \frac{d}{2}V \tag{1}$$

$$d = \frac{2E}{V} \tag{2}$$

由于增加了一个专门暂存高度数节点的特殊队列,第一个并行循环中的所有线程实现了负载平衡. 在处理特殊队列中的高度数节点时,可以重新在线程中分配任务,将每个高线程节点的众多邻居分配给所有线程共同访问,从而开发了邻接区域内的细粒度并行性.

4　实验性能评测

为了评测 FlexBFS 和相关程序优化技术的性能,我们在 GPU 上进行了实现,并与其它三个版本的 BFS 实现进行了比较,它们分别是: 单核 CPU 上的串行版本和多核 CPU 平台上基于 Intel TBB 的多线程版本,以及 parboil 基准测试程序集中的 CUDA 版本 UIUC-BFS[7]. 其中串行版本采用了当前最快的 BFS 串行实现,而 UIUC-BFS 也是近年来研究工作中效率较高的版本. TBB 版本在多核 CPU 平台上的性能比传统的 OpenMP 实现要更高效,多线程的维护开销比较小.

我们的实验平台选择了 NVIDIA Tesla C2050 GPU,具有 14 个流多处理器(SM),共 448 个 CUDA 核,工作主频为 1.15 GHz,显存为 3GB. 所用的主机平台为 Intel Xeon E5506 CPU,具有 2.13GHz 主频,4MB 二级缓存和 12G 主存. 串行版本和 TBB 版本的程序在这个主机平台上执行,其中 TBB 版本采用基于串行版本进行改写,使用 4 线程测得最佳性能. 而 CUDA 版本的 FlexBFS 和 UIUC-BFS 使用 CUDA3.2 平台进行编译和执行,使用的操作系统为 Debian Linux 内核 2.6.26.

为了概括更多的输入图类型,我们选择了三大类图作为测试用例,分别是规则图,不规则图和道路网图. 其中规则图采用格点图生成,每个节点连接其周围固定个数的邻居节点,平均度数取值分别为 6,8,12,16,图的节点数规模从一百万到一千万不等; 不规则图以规则图为基础,增加了千分之一的高度数节点,其出度值取值为平均度数的 100 倍左右; 道路网图采用美国各城市地区的真实道路节点图,图的规模从 20 万到 2 千万不等,都属于稀疏图的范畴. 表 1(见下页)总结了所有测试用图的细节.

我们先对三组不同的测试用图进行了整体性能评测,下面三个图分别显示了三种不同类型的图在 4 种实现下的执行

时间比较. 图 4 反映了对规则图的处理中,两种 GPU 实现都要好于 CPU 实现,其中 FlexBFS 比串行版本获得了 3 至 5 倍的加速比,这表明 BFS 算法中的无组织数据并行性获得了较

表 1 由三种不同类型图组成的测试用例

Table 1 Test bench consists of three different types of graphs

名称	顶点数	边数	名称	顶点数	边数
规则图 1	1M	3M	不规则图 7	5M	32.44M
规则图 2	1M	4M	不规则图 8	5M	42.32M
规则图 3	1M	6M	不规则图 9	10M	34.98M
规则图 4	1M	8M	不规则图 10	10M	45.92M
规则图 5	5M	15M	BAY	321K	400K
规则图 6	5M	20M	NY	264K	366K
规则图 7	5M	30M	COL	435K	528K
规则图 8	5M	40M	FLA	1M	1.3M
规则图 9	10M	30M	CAL	1.8M	2.3M
规则图 10	10M	40M	NE	1.5M	1.9M
不规则图 1	1M	3.46M	NW	1.2M	1.4M
不规则图 2	1M	4.48M	LKS	2.7M	3.4M
不规则图 3	1M	6.34M	E	3.6M	4.38M
不规则图 4	1M	8.44M	W	6.2M	7.62M
不规则图 5	5M	17.49M	CTR	14M	17.14M
不规则图 6	5M	22.38M	USA	23M	29.16M

好的开发;而 FlexBFS 比 UIUC-BFS 性能更高的原因在于使用了快速的共享存储器来保存局部队列,降低了访存延迟. 在图5中反映了负载平衡优化在处理不规则图时所带来的好处.

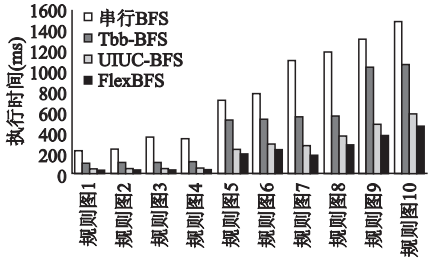

图 4 规则图上的性能比较

Fig. 4 Performance on regular gridgraphs

与串行版本相比 FlexBFS 获得了 2 到 3 倍的加速比;值得注意的是,UIUC-BFS 版本由于没有进行负载平衡优化,其性能甚至低于两种CPU版本,充分反映了在流处理器平台上对不

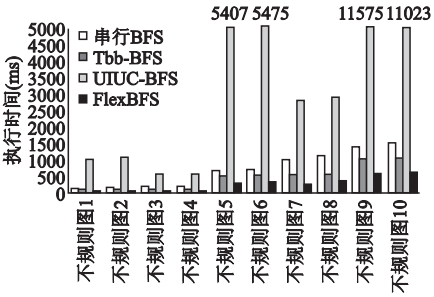

图 5 不规则图上的性能比较

Fig. 5 Performance on irregular grid graphs

规则算法进行程序优化的重要性. 在图 6 中显示了 FlexBFS 对于稀疏图的加速效果. 当图的规模较小时,所提供的无组织数据并行性很小,不足以掩盖访存延迟,因此 GPU 版本的性能提升并不明显;但是当图的规模增大时,动态队列中能够提供更多可供开发的并行性,因此 FlexBFS 能够获得接近 20 倍的加速比.

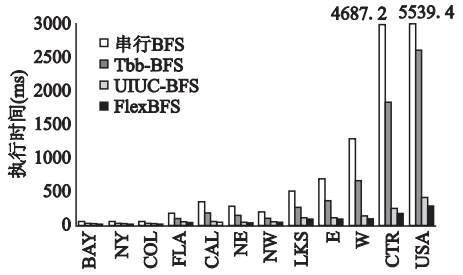

图 6 道路网图上的性能比较

Fig. 6 Performance on traffic network graphs

为了测试基于并行性反馈实现的 FlexBFS 性能,我们与计算资源分配由低到高的几种固定分配策略进行了比较,实验中分别测试了三种不同类型的输入图,见图 7. 使用固定分配策略时,持续增加线程资源并不能得到性能的进一步提升. 另外,不同类型的输入图对计算资源需求不同,程序员必须手动调整配置才能获得最佳性能. 而 FlexBFS 实现能够自适应的分配合理的线程数,而且能够使用较少的线程就获得最佳性能,比高配置的固定分配策略的资源利用率更高.

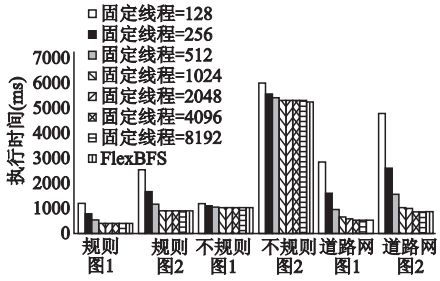

图 7 使用基于并行性反馈的灵活资源分配机制的优势

Fig. 7 Performance advantage of flexible configuration based on parallelism feedback

我们针对 BFS 算法提出的优化方法同样也适用于其它不规则程序在 GPU 上的实现. 例如,稀疏矩阵向量乘法(SpMV),单源最短路径(SSSP),最小生成树(MST)等.

5 结 论

图形处理器为高性能计算领域带来了新的发展机遇,但同时也为并行编程和系统软件带来了更大的挑战. 如何充分利用 GPU 丰富的计算资源来加速科学计算已经成为未来高性能计算领域的关键问题. 尤其是针对不规则应用的优化技术将会很大程度上影响 GPU 的应用面. 本文研究了以图广度优先搜索算法为代表的不规则程序在 GPU 上的运行时行为特征,分别从并行性模式、访存特性和负载平衡三个方面进行了分析,并提出了具有针对性的优化方法. 使用基于并行性剖析与反馈的计算资源重配置方法,能够自适应的为 BFS 程序重新分配计算资源,以满足其内在并行性模式的动态变化. 针

对 GPU 结构提出的动态队列的层次化优化方法,能够有效降低锁机制对频繁访问的共享队列造成的访问延迟,将随机访问模式限制在局部队列一级. 而线程级负载平衡方法能够充分利用所有硬件线程的计算能力,显著提高搜索不规则图的整体性能. 本文提出的优化方法同样也能应用到其它不规则程序上. 本文的后续工作包括: 实现不规则程序在 GPU 上的优化算法库; 提供更通用的 GPU 程序模板来开发具有不规则访存特性的应用,帮助应用程序员有针对性的优化程序性能; 研究控制流复杂的不规则应用在 GPU 上的优化实现.

References:

[1] Kulkarni M, Burtscher M, Inkulu R, et al. How much parallelism is there in irregular applications [C]. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP´09, 2009: 3-14.

[2] Hassaan M A, Burtscher M, Pingali K. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms [C]. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP´11, 2011: 3-12.

[3] Harish P, Narayanan P J. Accelerating large graph algorithms on the gpu using cuda [C]. In Proceedings of the 14th International Conference on High Performance Computing, HiPC´07, 2007: 197-208.

[4] Deng Y S, Wang B D, Mu S. Taming irregular eda applications on gpus [C]. In Proceedings of the 2009 International Conference on Computer-Aided Design, ICCAD´09, 2009: 539-546.

[5] Pingali K, Kulkarni M, Nguyen D, et al. Amorphous data-parallelism in irregular algorithms [R]. TR-09-05, 2009.

[6] Scarpazza D, Villa O, Petrini F. Efficient breadth-first search on the cell/be processor [J]. IEEE Transactions on Parallel and Distributed Systems, 2008, 19(10): 1381-1395.

[7] Luo L, Wong M, Hwu W M. An effective gpu implementation of breadth-first search [C]. In Design Automation Conference (DAC), 2010 47th ACM/IEEE, 2010: 52-55.

[8] Lefohn A E, Kniss J, Strzodka R, et al. Glift: generic, efficient, random access gpu data structures [J]. ACM Transactions on Graphics, 2006, 25(1): 60-99.

[9] Harish P, Vineet V, Narayanan P J. Large graph algorithms for massively multithreaded architectures [R]. IIIT/TR/2009/74, 2009.

[10] Ryoo S, Rodrigues C I, Baghsorkhi S S, et al. Optimization principles and application performance evaluation of a multithreaded gpu using cuda [C]. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP´08, 2008: 73-82.

[11] Bader D A, Madduri K. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2 [C]. In International Conference on Parallel Processing, 2006: 523-530.

[12] Yoo A, Chow E, Henderson K, et al. A scalable distributed parallel breadth-first search algoritm on bluegene/l [C]. In Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC´05, Washington, DC, USA, 2005: 25.

[13] Hong S, Kim S K, Oguntebi T, et al. Accelerating cuda graph algorithms at maximum warp [C]. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP´11, 2011: 267-276.