两种 GPU 上改进的最短路径算法

刘 欣, 王 非
(哈尔滨工业大学 深圳研究生院, 广东 深圳 518055)

摘 要: 针对图论中的最短路径问题,提出了两种在 GPU 上改进的最短路径搜索算法,即针对单源最短路径问题的基于迭代方式且采用原子锁优化的 Advanced_Atomics_SSSP 算法以及针对所有顶点间最短路径问题的采用二叉堆优化的 Heap_APSP 算法。将两种算法应用到美国公路网图和节点的度均为 6 的普通图中,通过对算法的测试表明,Advanced_Atomics_SSSP 算法的性能依赖于节点的度数,当节点的度数大于 6 时加速效果明显,当节点度数为 1 ~ 3 时加速效果不明显;而 Heap_APSP 可以达到 46 ~ 56 倍的加速比,且加速性能不受节点度的影响。

关键词: Dijkstra 算法; 单源最短路径; 所有顶点间最短路径; GPU; 原子锁; 二叉堆
中图分类号: TP301.6 **文献标志码**: A **文章编号**: 1001-3695(2014)05-1407-03
doi:10.3969/j.issn.1001-3695.2014.05.029

Two improved shortest path algorithms on GPU

LIU Xin, WANG Fei
(*Shenzhen Graduate School*, *Harbin Institude of Technology*, *Shenzhen Guangdong* 518055, *China*)

Abstract: In order to solve the shortest path problem, this paper presented two improved shortest path search algorithms on GPU: the iterative Advanced_Atomics_SSSP algorithm based on atomic lock for single source shortest path problem and Heap_APSP algorithm based on binary heap for all pair shortest paths problem. At last, it made a comprehensive comparison and analysis of these algorithms, Advanced_Atomics_SSSP is faster than CPU counterpart Dijkstra algorithm when the degree per vertex is 6. Heap_APSP algorithm is faster than CPU Floyd-Warshall algorithm and achieves 46-56x speedup and Heap_APSP algorithm's speedup has not been affected by the degree per vertex.

Key words: Dijkstra algorithm; single source shortest path; all pair shortest paths; GPU; atomic lock; binary heap

NVIDIA 发布的 CUDA 技术,为开发人员有效利用 GPU 的强大功能提供了有利条件。CUDA 被广泛应用于科学计算、生物计算、图像处理等领域,并在很多应用中取得了不同数量级的加速比,GPU 成为了目前较流行的并行开发平台。虽然在 GPU 上并行执行可以很容易达到较 CPU 加速几十甚至几百倍的性能,但设计 GPU 上的算法并非易事,开发者要想取得较高的加速比,必须深入理解算法本身以及 GPU 的架构和并行计算的机制,才可以实现较高的加速比。

最短路径问题是图论中的一个经典算法问题, 该问题的解可以应用到许多领域,如地理信息系统、路由网络、智能物流系统和生物信息系统等。该问题可分为两类:a)单源最短路径(single source shortest path,SSSP),求给定始末顶点之间的最短路径;b)所有顶点间最短路径(all pair shortest paths,APSP),求图中所有顶点对之间的全部最短路径。然而在 GPU 上求解最短路径问题并不简单。一个重要的原因就是,许多在 CPU 上编程用到的数据结构,如图(graph)、队列(gueue)、栈(stack)等,在 GPU 上表达并不方便。

文献[1]提出了 GPU 上图的表达方式,并实现了 BFS、SSSP、APSP 算法。然而它所提出的 SSSP、APSP 算法只能求出最短路径的长度,没有保存路径,而且当处理稀疏图时,文献中迭代的 SSSP 算法性能还不如在 CPU 上可以达到的性能。文献[1]提出了基于 SSSP 迭代的 APSP 算法和 Parallel-Floyd-Warshall 型的 APSP 算法。文献[2,3]是在 Parallel-Floyd-Warshall 算法的基础上进行改进,其本质是将 Floyd-Warshall 算法中比较权值操作改为 CUDA 多线程并行执行。文献[3]在基于文献[1]中迭代 SSSP 的 APSP 算法基础上,利用任务并行机制改进,仍没有实现保存具体最短路径的功能;文献[4]提出了一种分块式的 APSP 算法,可以处理大规模的图;文献[5]在文献[3]的基础上改进,利用 GPU 计算最短路径大小时并保存节点的父节点,然后在 CPU 上根据每个节点的父节点利用回溯策略记录最短路径。根据文献[1]的结论可知,使用迭代 SSSP 解决 APSP 问题,性能要优于使用 Floyd-Warshall 解决 APSP 问题。

本文的贡献有:a)对文献[1]的 SSSP 算法加以改进,不仅求出给定的源点到其他节点的最短路径的长度,并且可以保存这样的路径;b)提出了一种基于堆操作的 APSP 算法,该算法的思想是每个线程负责搜索从一个节点到其他节点的最短路径,即 SSSP,每个线程采用 heap 优化的 Dijkstra 算法。

1 GPU 上图的表达方式和二叉堆的实现方法

1.1 GPU 上图的表达方式

在 CPU 上图的表达方式可以是数组、邻接表、十字链表等。当表达稀疏图时,采用邻接表方法可以节省存储空间。在

收稿日期: 2013-07-03; **修回日期**: 2013-08-14

作者简介:刘欣(1989-),女,硕士,主要研究方向为高性能计算、基于 GPU 的 FPGA 并行布线(865761014@qq.com);王非(1967-),男,副教授,博士,主要研究方向为高性能计算、计算机辅助设计.

GPU_CUDA 编程模型中,可以很方便地对数组进行操作,但是传统 C 语言中的数据结构并非可以简单地搬到 CUDA 中来。本文采用一种改进的紧凑邻接表来表示图 $G(V,E,W)$[1],用到三个数组 $V[\,]$、$E[\,]$、$W[\,]$。($V[i+1]-V[i]$)是以节点 i 为起点的有向边数,即节点的出度,$V[i]$ 表示第 i 个节点在 $E[\,]$中的起始位置;$E[i]$ 表示第 i 条有向边的终点,从 $V[i]$ 到 $V[i+1]$ 所对应的数组 E 中的元素是节点 i 的邻接点;$W[i]$ 表示第 i 条有向边的权值。图 1 描述了一个有向图在 GPU 上的表达方式。

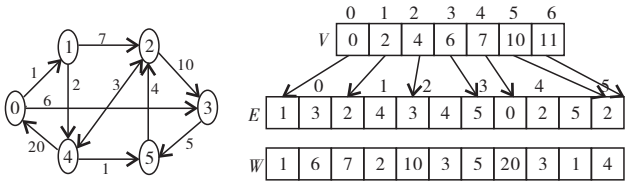

图 1 GPU 上图的表达方式

1.2 GPU 上二叉堆的实现方法

最短路径算法需要一个优先队列 Q,根据路径的长短,将节点插入 Q 或从 Q 中移除节点。当图比较大时,此项操作会花费很长时间。因此需要一种高效的方法来维护队列 Q。文献[6]讨论了 GPU 上的多种排序方法,本文借鉴其 heap 排序方法,用来维护优先队列。图中已搜索到的节点使用结构 Node 来表示,它包含以下成员:Index 表示节点的索引号;Dist 保存从源点到该节点的已搜索到的最短路径长度;Parent 保存父节点的 Index。在优先队列中,根据节点 Dist 的值,按最小二叉堆形式插入和移除节点,达到维护优先队列 Q 的目的。

插入节点的具体操作为:假设优先队列 Q 中有 $n-1$ 个节点,新节点插入到队列尾部,此时优先队列中有 n 个节点,然后根据节点的 Dist 确定第 n 个节点的确切位置。比较第 n 个节点和第($n/2$)个节点的 Dist,若第 n 个节点的 Dist 小于第($n/2$)个节点的 Dist,则交换两者,再比较第($n/2$)个节点的 Dist 和第($n/2/2$)个节点的 Dist,直至该节点的 Dist 不大于它的父节点的 Dist,或者到达优先队列的顶部。

移除节点的具体操作为:首先删除优先队列中 Dist 最小的节点,即优先队列 Q 的头节点,然后重新排列 Q,将尾部节点移至 Q 顶部。比较第一个节点与第二个节点的 Dist,若第一个节点的 Dist 大于第二个节点的 Dist,则交换两个节点,再比较第二个节点与第四个节点,直至优先队列中父节点的 Dist 不大于子节点的 Dist。假设 Q 中有 N 个节点,则插入和移除节点的时间复杂度为 $O(\log N)$[7]。

在 GPU 上,使用最小二叉堆需要有 heap memory 的支持。在 kernel 函数中,通过 malloc( ) 和 free( ) 函数分配或者释放 heap 存储空间。通过下面两个 API 函数可以获取 heap 的大小和设置 heap 的大小:

```
cudaDeviceGetLimit(size_t * size, cudaLimitMallocHeapSize)
cudaDeviceSetLimit(cudaLimitMallocHeapSize, size_t size)
```

2 可保存路径的迭代 Advanced_Atomics_SSSP 算法实现

文献[1]提出的迭代 SSSP 算法,但其功能并不完善,只能计算出从源点到其他各节点的最短路径长度,并没有保存相应的路径。同样文献[3]中的算法也没有保存最短路径所经过的节点。当搜索到更短路径时,文献[1]通过原子操作 atomicMin( ) 比较新路径与原来的路径,并保存两者中较小的路径。但由于原子操作的局限性,只能对单一变量进行修改,而路径的记录仅用一个变量表示是不完善的,这导致了文献[1]中的 Dijkstra_SSSP 算法无法保存路径。针对这个缺陷,本文对文献[1]的原子操作加以改进,引入高级原子操作,即实现锁定数据结构;构造原子锁,实现对数据结构 Node 的互斥操作,以保证数据的完整性。原子锁的构建方法如代码 1 所示。其基本思想是:分配一部分内存作为互斥体,互斥体的作用如同控制对某个资源访问的互斥信号,当某个线程从互斥体中读到 0 时,表示没有其他线程正在使用这块内存,此时该线程可以锁定这块内存并进行必要的修改。实现锁定内存位置及线程将 1 写入互斥体,这样就可以防止其他竞争的线程对这块内存锁定,其他线程需等待,直至互斥体的所有者线程将 0 写入互斥体才能尝试修改被锁定的内存。通过调用其中的 lock 和 unlock 设备函数,可以实现对数据结构 Node 的互斥操作。

在 Advanced_Atomics_SSSP 算法中,若图的节点个数为 n,则需要定义 n 个 novel_Lock 结构的原子锁。当搜索到更短路径时,调用原子锁中的 lock 函数锁定数据结构 Node 定义的数据并更改其内容,然后再调用 unlock 函数解锁。

代码 1 原子锁结构

```
struct novel_Lock {
    int * mutex;
    Lock( void ) {
        cudaMalloc( (void * * )&mutex, sizeof(int) );
        cudaMemset( mutex, 0, sizeof(int) );
    } ~Lock( void ) {
        cudaFree( mutex );
    } __device__ void lock( void ) {
        while( atomicCAS( mutex, 0, 1 ) ! = 0 );
    } __device__ void unlock( void ) {
        atomicExch( mutex, 0 );
    } };
```

采用 1.1 节介绍的方法表示非负权重的图 $G(V,E,W)$,给定源点 s,搜索 s 到其他节点的最短路径。Advanced_Atomics_SSSP 算法搜索最短路径分为两部分:a) Dijkstra_Relax,并行计算节点到其邻接点的路径,并保存最短的路径;b) Dijkstra_Update,根据 a) 中得到的结果,确认是否为最短路径,若是最短路径,则更新节点的路径。

Advanced_Atomics_SSSP 算法的伪代码如代码 2 所示。首先设置从源点 s 到其他节点的路径 Dist 为无穷大,Parent 为 No_previous,U 是数组,数组元素具有 Node 的数据类型,大小是节点个数,用来保存 Dijkstra_Relax 阶段搜索到的节点路径的长度和父节点,task 表示是否需要计算该节点到邻接点的路径,初始化为 false,源点 s 的 Dist 为 0,task 为 true。然后进入迭代阶段:a) Dijkstra_Relax,若节点的 task 为 true,则计算其到邻接点的路径长度,若出现更短路径,则对 $U[\text{nid}]$ 进行原子操作,修改 Dist 和 Parent;b) Dijkstra_Update,比较每个节点的 Dist 和对应 U 中的 Dist,若 Dist 大于对应 U 中的 Dist,则更新 Dist 和 Parent,并将节点的 task 设为 true,返回 a) 继续迭代,直至 task 中各元素都为 false。

代码 2 Advanced_Atomics_SSSP 的伪代码

```
Advanced_Atomics_SSSP (Graph G(V,E,W),s)// Initialize
U←{Index,Max,No_previous}, Dist←Max, Dist[s] ←0,
Parent←No_previous, task[s] ←true
while exist nodes to access do
    // Dijkstra_Relax
    for each vertex of V in parallel do {
        tid ←getThreadID
    if (task[tid]) then
        task [tid] ←false
        for all neighbors nid of tid do
        //nid 是节点 tid 的邻接点的索引
```

```
            if (U[tid]. Dist > Dist[tid] + W[nid] ) then
                Dev_lock[nid]. lock;
                // Dev_lock 的类型为 novel_Lock
                U[nid]. Dist ←Dist[tid] + W[nid]
                U[nid]. Parent ←tid
                Dev_lock[nid]. unlock   //解锁
                end if
                end for
    end if
    synchronization
    // Dijkstra_Update
    if Dist[tid] > U[tid]. Dist then
        Dist[tid] ← U[tid]. Dist
        Parent[tid] ← U[tid]. Parent
    end if
    }
    end while
```

在 GPU 上实现该算法时,若有 n 个节点,则需要启动 n 个线程,定义 n 个原子锁。该算法在 Dijkstra_Relax 和 Dijkstra_Update 两个过程中间需要同步,但是 CUDA 没有全局同步机制,因此需要设置两个 kernel 函数,一个完成 Dijkstra_Relax 的功能,然后通过 CPU 调用另一个 kernel 函数,完成 Dijkstra_Update 的功能。

3　利用二叉堆优化的 Heap_APSP 算法

APSP 问题用来求解给定图中所有顶点对之间的最短距离,该问题的解法不外乎分成两种类型:a)将单个 SSSP 算法包含在 for 循环体中,每次循环以一个顶点作为源点,求出源点到其他节点的最短路径,然后改变源点,进入下次循环,直到完成全部计算;b)采用 Floy - Warshall 算法,该类型算法由一个三重循环构成,算法的时间复杂度为 $O(n^3)$,n 为节点个数。

本文提出了新的算法,在 CUDA 中引入 heap 对 APSP 算法加以改进,时间复杂度降为 $O((m+n)\log n)$,其中 n 是节点数,m 是边数。称之为 Heap_APSP 算法,摆脱了文献[1]中对图中节点度的限制。文献[3]中的 SSSP 采用文献[1]中的 Dijkstra_SSSP 的方法并进行了优化,没有采用 heap; 本文则使用 heap 对 Dijkstra 算法进行优化。

Heap_APSP 的伪代码如代码 3 所示。Heap 中每个元素的类型为结构 Node。并行启动多个线程,每个线程负责将一个节点作为源点 s,搜索从 s 到其他各个节点的最短路径。线程内部使用了基于 heap 优化的 Dijkstra 算法,即在维护优先队列时,根据路径距离采用最小二叉堆排序,降低了算法的时间复杂度。

代码 3　Heap _APSP 算法的伪代码

```
Heap _APSP( Graph G(V,E,W), Dist, Parent)
Initialize;Dist ←Max, Parent ←No_previous, heaptail←1,
tid ←getThreadID;
malloc heap Memory;
s← tid;
insert s node to heap;
While heap not empty do
      Get the heap head node called current_node;
      if( current_node. Dist < Dist[ current_node. Index] )
        Dist[ current_node. Index] = current_node. Dist;
        Parent[ current_node. Index] = current_node. Parent;
        for all current_node's adjacent node with index of i
            if ( assist_array[ i]. temp_dist > Dist[ current_node. index] + W[ i] )
assist_array[ i]. temp_dist = Dist[ current_node. index] + W[ i]
if( assist_array[ i]. location = = -1) then   //节点 i 未加入到 heap
                add the node i to heap, return the node location in heap, heaptail ++
                else
replace the node i in heap, return the node location in heap
                end if
                end for
                end if
end while
free heap memory
```

在 CPU 上可以动态分配内存,随时增加 heap 的存储空间,而 GPU 上要求在 kernel 启动之前设置好 heap 的存储空间大小。因此,需要增加一个辅助数组 assist_array,数组元素的类型为结构 assist_Node。assist_Node 包含以下成员:temp_dist,保存节点的临时最短路径;location,保存节点在 heap 中的位置。

计算过程:将源点 s 的 Dist 设为 0,Parent 设为 No_previous,assist_array[0]. temp_dist 为 0,assist_array[0]. location 为 1,将源点 s 放入 heap 顶部,然后进入循环体:a)获取 heap 中 Dist 最小的元素,即 heap 的头节点,称为 current_node; b)若 current_node 的 Dist 小于 Dist[current_node. Index],就根据 current_node 的 Dist 和 Parent 更新 Dist[current_node. Index]和 Parent[current_node. Index],并计算从 current_node 到其邻接点 i 的路径,若此路径小于 assist_array[i]. temp_dist 且此节点未加入 heap 中,则将邻接点加入 heap,更新 assist_array[i]. temp_dist,并返回此节点在 heap 中的位置,若此节点已加入到 heap 中,则更新 assist_array[i]. temp_dist,修改 heap 中此节点的最短路径,并进行重新排序,返回此节点在 heap 中的位置;直至 heap 为空循环结束。这样就求出了从源点 s 到其他节点的最短路径。该过程中对 heap 插入和移除节点的操作见 1.2 节。

4　实验结果

文中所有程序的运行环境为 CPU 为 Intel Core i3 760,内存为 4 GB;GPU 为 GeForce GTX 550Ti,Graphics Memory 1GB。

表 1 是用来测试算法性能的四幅图,列出了每幅图中包含的节点和边的个数。利用二叉堆优化的 Dijkstra 算法是 CPU 上解决 SSSP 较快的方法[5]。表 2 是 CPU 串行 heap 优化的 Dijkstra 算法、文献[1]中 GPU 迭代 SSSP 算法和 GPU 并行 Advanced_Atomics_SSSP 算法的结果比较。采用美国公路网图(New York, Florida) 对 Advanced_Atomics_SSSP 算法进行测试,笔者发现本文算法可以起到加速的效果,但加速比并不理想。这两个图的共同点是节点的度比较低,平均的度是 2.5 ~ 2.7。而采用 Graph4096 和 Graph65536,对 Advanced_Atomics_SSSP 算法进行测试,则取得了 4.7 ~ 5.7 的加速比,加速效果很明显。这两个图的节点度均为 6。Advanced_Atomics_SSSP 的优势就是当图中节点度比较大时,Dijkstra_Relax 阶段并行线程中进行有效计算的线程数多,其加速性能较优。与文献[1]中的 GPU 迭代 SSSP 算法相比,其功能更加完善,虽然使用了原子锁,实现 GPU 内存的互斥,保存路径的具体节点,但算法速度并未受到很大影响。

算法 Heap_APSP 的并行线程配置如下:设置 heap 的存储空间为 512 megabytes,每个线程块中的线程数为 512,线程块为节点个数除以 512 上取整,最大不要超过 65 535,每个线程负责搜索一个节点到其他节点的最短路径,搜索完毕后立即释放 heap 存储空间。表 3 是 CPU 上 Floy-Warshall 算法、文献[1]中的基于迭代 SSSP 的 APSP 算法 APSP USING SSSP 和 GPU 上并行 Heap_APSP 算法的性能比较。与 CPU 上 Floy-Warshall 算法相比,Heap_APSP 算法可以实现 46 ~ 56 倍的加速效果,并且其加速效果与节点的度无关。与文献[1]中的 APSP USING SSSP 算法相比,实现了 1 ~ 3 倍的加速效果。　(下转第 1413 页)

出,考虑属性截止时间和空闲时间的 EDF 算法,在最坏情况下(R_{17})调度成功率也高于传统 EDF、LLF 和 PTD 算法;而采用最大空闲时间优先的 EDF 算法(R_{20})具有最高的调度成功率。原因是当任务的截止时间相同时,空闲时间越大,表示任务还需执行的时间越短,优先执行长空闲时间的任务可以在保证有一个任务执行完的基础上节省更多的时间。

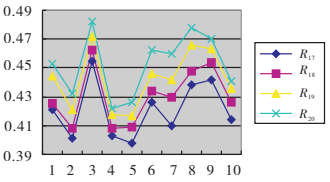

图8　PTBM调度成功率

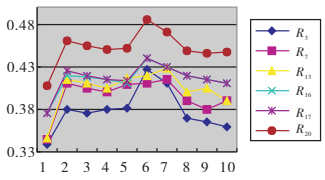

图9　EDF、LLF、PTD、PTBM调度成功率

5　结束语

本文首先讨论了算法调度成功率对于实时任务属性的敏感度,提出了不同算法对于任务属性有不同的相对敏感度。影响算法调度成功率的因素不仅包含任务属性,还包含属性的变化规律。基于敏感度的理论,分析了任务属性和变化规律对算法调度成功率的影响度,得出任务的截止时间和空闲时间是最有影响的两个因素。综合考虑这两个属性,使得任务的截止时间越小,任务的优先级越高;相同截止时间下,空闲时间越大的任务优先级越高。这是因为空闲时间越大,表示任务还需执行的时间越短,优先执行长空闲时间的任务可以在保证有一个任务执行完的基础上节省更多的时间。最后设计了 20 种不同的算法或者策略来验证仿真验证敏感度和影响度的理论,实验结果证明 PTBM 算法具有较高的调度成功率。

参考文献:

[1] ZHOU Shi, MONDRAGON R J. Accurately modeling the Internet topology[J]. Physical Review E,2004,70(6):96-108.

[2] LIU C L, LAYLAND J W. Scheduling algorithms for multiprogramming in a hard-real-time environment[J]. Journal of the ACM, 1973,20(1):46-61.

[3] LIU Yun-sheng, HE Xin-gui, TANG Chang-jie, *et al*. Special type database technology[M]. Beijing: Science Press, 2000.

[4] JENSEN E D, LOCKE C D, TODUCA H. A time-driven scheduling model for real-time operating systems[C]//Proc of IEEE Real-time Systems Symposium. Washington DC: IEEE Computer Society Press, 1985:112-122.

[5] GOOSSENS J, RICHARD P. Overview of real-time scheduling problems[C]//Proc of the 9th International Workshop on Project Management and Scheduling. 2004:13-22.

[6] 金宏,王宏安,王强,等. 一种任务优先级的综合设计方法[J]. 软件学报, 2003,14(3):376-382.

[7] 金宏,王宏安,王强,等. 改进的最小空闲时间优先调度算法[J]. 软件学报,2004,15(8):1116-1123.

[8] 李琦,巴巍. 两种改进的 EDF 软实时动态调度算法[J]. 计算机学报, 2011,34(2):943-950.

[9] 檀明,魏臻,韩江洪. 非抢占式 EDF 算法下周期性任务的最小相对截止期计算[J]. 计算机应用研究, 2012,29(2):722-724.

[10] 袁晷,檀明,周晶晶. EDF 调度算法可调度性分析方法的改进研究[J]. 计算机应用研究,2013,30(8):2429-2431.

[11] 王多强,鲁剑锋,李庆华. 实时调度中基于多特征参数的任务优先级设计方法[J]. 计算机工程与科学, 2008,30(1):73-78.

[12] BIYABANI S R, STANKOVIC J A, RAMAMRITHAM K. The Integration of deadline and criticalness in hard real-time scheduling[C]//Proc of the 9th IEEE Real-time Systems Symposium. Washington, DC: IEEE Computer Society Press, 1988:152-160.

[13] 穆阿里,吴仲光,张昭瑜,等. 一种多特征综合的实时[J]. 四川大学学报:自然科学版,2005,42(3):621-623.

(上接第 1409 页)

表 1　测试图的基本信息

Graph	Vertex	edges	Graph	Vertex	edges
Graph4096	4 096	24 576	New York	264 346	733 846
Graph65536	65 536	393 216	Florida	1 070 376	2 712 798

表 2　SSSP 算法实验结果比较

Graph	Dijkstra_ heap/ms	SSSP GPU/ms	Advanced_ Atomics_ SSSP/ms	Graph	Dijkstra_ heap/ms	SSSP GPU/ms	Advanced_ Atomics_ SSSP/ms
Graph4096	6.28	0.98	1.32	New York	157	102	118
Graph65536	21.53	4.15	3.76	Florida	1 086	1 651	1 593

表 3　APSP 算法实验结果比较

Graph	CPU_ APSP/s	APSP USING SSSP/s	Heap_ APSP/s	Graph	CPU_ APSP/s	APSP USING SSSP/s	Heap_ APSP/s
Graph4096	545	12.44	9.6	New York	23 518	1 569	511
Graph65536	13 104	743	252	Florida	65 950	2 652	1 403

5　结束语

本文针对图论中最短路径问题提出了 Advanced_Atomics_SSSP 和 Heap_APSP 算法。通过实验验证,Advanced_Atomics_SSSP 算法对每个节点的度大于 6 时,其加速效果更加明显;Heap_APSP 可以取得 40 ~ 50 倍的加速效果。根据 Heap_APSP 的特点,也可以实现有限个节点到其他部分节点的最短路径搜索。因此,笔者后续工作将把该算法应用到 FPGA 的布线算法中,期望能够缩短 FPGA 的布线时间。

参考文献:

[1] HARISH P, NARAYANAN P J. Accelerating large graph algorithms on the GPU using CUDA[C]//Proc of the 14th International Conference on High Performance Computing,2007:197-208.

[2] 张凌洁,赵英. 基于 GPU 的并行 APSP 问题的研究[J]. 电子设计工程,2012,20(17):15-18.

[3] OKUYAMA T, INO F, HAGIHARA K. A task parallel algorithm for computing the costs of all-pairs shortest paths on the CUDA-compatible GPU[C]//Proc of International Symposium on Parallel and Distributed Processing with Applications. 2008: 284-291.

[4] MATSUMOTO K, NAKASATO N, SEDUKHIN S G. Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system[C]//Proc of IEEE Interenational Conference on High Performance Computing and Communications. 2011:145-152.

[5] OKUYAMA T, INO F, HAGIHARA K. A task parallel algorithm for finding all-pairs shortest paths using the GPU[J]. International Journal of High Performance Computing and Networking,2012, 7(2):87-98.

[6] https://dev. ece. ubc. ca/projects/gpgpu-sim/browser/ispass2009-benchmarks/BFS/datalast visited[EB/OL]. (2013-04-24).

[7] JOSEPH T, Jr KIDER. GPU as a parallel machine: sorting on the GPU, CIS 700/010[EB/OL]. http://www. cis. upenn. edu/ ~ suvenkat/700/lectures/19/sorting-kider. pdf.

[8] 9th DIMACS implementation challenge-Shortest paths[EB/OL]. (2013-04-27). http://www. dis. uniroma1. it/challenge9/download. shtml, last visited.

[9] TRAN Q N. Designing efficient many-core parallel algorithms for all pair shortest-paths using CUDA[C]//Proc of the 7th International Conference on Information Technology: New Generations (ITNG). 2010:7-12.