基于GPU的全源最短路径算法

邢星星 赵国兴 骆祖莹 方 浩
(北京师范大学信息科学与技术学院 北京 100875)

摘 要 针对有向图中每对顶点之间的最短路径问题,基于CPU集群并行算法,根据GPU并行计算加速机制,提出了基于棋盘划分方式的GPU并行算法,以增加算法的并行性与数据的局部性。当有向图规模超过GPU显存限制时,进一步提出了异步并行处理的GPU最短路径算法。实验结果表明,与CPU上单核算法相比,本算法具有如下加速效果:(1)对于节点数少于10000的小规模有向图,可以实现约155倍的加速;(2)对于节点数超过10000的大规模有向图,可实现约25倍的加速。

关键词 全源最短路径,GPU,棋盘划分,Floyd算法
中图法分类号 TP311 **文献标识码** A

GPU-based Algorithm of Shortest Path

XING Xing-xing ZHAO Guo-xing LUO Zu-ying FANG Hao
(College of Information Science and Technology, Beijing Normal University, Beijing 100875, China)

Abstract As for the all-pairs shortest-path problem in the graph, based on parallel algorithm in the CPU cluster environment, depending on parallel speedup mechanism on the GPU, in order to increase the parallelism and locality, chessboard division method was chosen for task division in this parallel algorithm on the GPU. Because the graph scale is larger than the display memory, the asynchronous parallel algorithm on the GPU was presented. The experimental data proves that the algorithm has accelerating effects below compared with CPU with single core: first, for the small graph whose vertexes are less than 10000, it is about 155 times faster; second, for the large graph whose vertexes are more than 10000, it is about 25 times faster.

Keywords Shortest-path, GPU, Chessboard division, Floyd

1 引言

最短路径算法作为代表性的图论算法在航空飞行路线分配、物流选择、计算机网络路由选择等领域均有大量应用。

一般来说,最短路径问题分为单源最短路径和全源最短路径。本文关注的是所有点对之间最短路径问题,即全源最短路径问题。全源最短路径的典型串行解法包括基于矩阵乘法的动态规划方法[1]和Floyd算法[2]两种。由于这两种方法分别具有$\Theta(n^3)\cdot\log(n-1)$和$\Theta(n^3)$的计算复杂性,因此对于规模超过$N=2500$的有向图,运算时间超过1min,无法实时计算;对于规模超过$N=10000$的有向图,计算时间会超过1h。

随着GPU通用计算的发展,GPU对许多问题可以实现几十到上百倍的加速。使用GPU和CPU相比,主要有几个好处:显示芯片通常具有更大的内存带宽;显示芯片具有更大量的执行单元;和高阶CPU相比,显卡的价格较为低廉。

部分国外学者提出的GPU上求解全源最短路径问题的并行算法,包括Harish提出的基于Dijkstra单源最短路径的GPU端迭代算法[3]和Okuyama对GPU端迭代算法使用任务并行进一步改进的算法[3],以及Katz提出的一种基于Floyd算法的带状划分的GPU并行算法[5]。国内目前还缺少全源最短路径问题在GPU上的研究,只有在机群上实现的并行算法[6]。

已有算法虽然实现了一定程度上的并行,但是没有充分利用GPU硬件中细粒度的线程间通信的优势,没有提高数据的吞吐量。同时,已有算法任务划分都是采用通用的条带状划分,导致数据传输过于频繁。因为GPU显存的带宽有限,从而在和CPU传输数据时会导致延迟,降低计算效率。由于没能充分利用GPU的硬件特性来设计并行算法,已有算法的加速比相对较低,一般为3.5~15倍;并且已有算法都只适用于显存容量范围内数据规模较小的情况,实用性不足。

本文基于上述并行算法,研究实现了GPU上的全源最短路径并行算法。通过GPU上矩阵乘法的高效实现,提出了GPU端的基于矩阵乘法的动态规划算法实现;另一方面,

到稿日期:2011-04-09 返修日期:2011-07-16 本文受国家"863"高技术研究发展计划(2009AA01Z126),国家自然科学基金(60876025)资助。

邢星星(1989-),男,主要研究方向为GPU高性能计算,E-mail:xingxingxing@mail.bnu.edu.cn;**赵国兴**(1981-),男,博士,讲师,主要研究方向为理论计算机与高性能计算;**骆祖莹**(1968-),男,博士,副教授,CCF高级会员,主要研究方向为低功耗设计与物理设计、高性能计算;**方 浩**(1990-),男,主要研究方向为GPU高性能计算。

利用任务棋盘划分策略和 GPU 上异步并行技术，实现了 GPU 端的并行 Floyd 算法。根据数据矩阵规模的不同，在 GPU 上实现的算法较单核 CPU 上的算法获得了最高上百倍的加速比。在数据规模超出 GPU 显存的时候，采取了分批处理和异步并行数据处理的方法，取得了数十倍的加速比。分析说明，本文并行算法具有实现简单、扩展性强、适用于大规模计算等特点，有良好的性能和实用价值。

本文第 2 节是最短路径算法在 CPU 上实现的性能分析；第 3 节阐述最短路径算法在 GPU 上的优化实现；第 4 节是实验数据及其分析；第 5 节是数据矩阵规模过大时采用的处理算法；最后是算法进一步改进的方向。

2 最短路径的 CPU 算法实现及分析

给定有向图 $G=\langle V,E,W\rangle$，V 为顶点集合，E 为边集合，W 为各边权重。默认路径的权值都是正数。

2.1 基于矩阵乘法的最短路径串行算法

使用矩阵相乘法的核心思想是通过 n 次迭代，第 k 次迭代得到的是任意两个结点间边的条数最多为 k 时的最短路径。

设 p_{ij}^k 是从结点 v_i 到结点 v_j 的最短路径，k 表示 v_i 到 v_j 最多包含 k 条边。若 p_{ij}^k 的倒数第 2 个结点是 v_m，则得到如下递推式：

$$p_{ij}^k=\min\{p_{ij}^{k-1},p_{im}^{k-1}+w(v_m,v_j)\} \tag{1}$$

设 $P^{(k)}=(p_{ij})$，则 $P^{(k-1)}$ 即为问题的解。

如果将式(1)中“+”操作视作矩阵乘法中的“×”；将“min”视作矩阵乘法中的“求和”，则形式上就等同于矩阵乘法，即

$$P^{(k-1)}=P^{(k-1)}\times W \tag{2}$$

具体算法如下。

算法 1 基于矩阵相乘的最短路径算法

```
(1) n←rows[w]
(2) P(1)=W
(3) m←1
(4) while m<n−1
(5)       do P(2m)←P(m)×P(m)
(6)             m←2m
(7) Return P(m)
```

问题就转化为利用特殊的加法和乘法求 $P^{(n-1)}$。然后利用折半乘法，只需要进行 $\log_2^{n-1}$ 次特殊的矩阵乘法运算就可以得到所有点对的最短路径。因为 $\log_2^{n-1}$ 个矩阵乘积中每一个都需要 $\Theta(n^3)$时间，因此该算法的运行时间是 $\Theta(n^3)\cdot\log(n-1)$。若不采用折半乘法，则运行时间要慢得多，为 $\Theta(n^4)$。

2.2 串行 Floyd 算法的分析

如果从顶点 v_i 到 v_j 有路可达，则 $w[i][j]$为有限值，表示该路的长度；否则 $w[i][j]=\infty$。

把各个顶点插入图中，比较插点后的距离与原来的距离：

$$w[i][j]=\min(w[i][j],w[i][k]+w[k][j]) \tag{3}$$

根据式(3)，如果 $w[i][j]$的值变小，则将该点插入路径中。

算法 2 Floyd 串行算法

```
(1) 输入：给定的输入 n，图的邻接矩阵 W
(2) for(k=0;k<n;k++)
(3)   for(i=0;i<n;i++)
(4)     for(j=0;j<n;j++)
(5)       if(w[i][j]>w[i][k]+w[k][j])
(6)       {
(7)         w[i][j]>w[i][k]+w[k][j]
(8)       }
(9) Return W
```

对于大规模的图求解所有点对之间最短路径，Floyd 串行算法的时间复杂度为 $\Theta(n^3)$，这样要花费较长的运算时间。

对 Floyd 算法进行分析，发现上述算法过程中 k 的含义即全源最短路径是否经过第 k 个顶点 v_k。当顶点 v_k 选定的时候，进行如下步骤：

```
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
      if(w[i][j]>w[i][k]+w[k][j])
      {
        w[i][j]>w[i][k]+w[k][j]
      }
```

$W[i][j]$的更新存在一定的逻辑顺序：

1)更新 $w[k][k]$，即图中顶点 v_k 到顶点 v_k 的最短路径长度。经分析可知，$w[k][k]$不需要更新，因为必然有 $w[k][k]<w[k][k]+w[k][k]$。

2)更新 $w[i][k]$和 $w[k][j]$，$i,j=0,\cdots,n-1$，且 $i\neq k$，$j\neq k$，即图中的顶点 v_i 到顶点 v_k 和顶点 v_k 到顶点 v_j 的最短路径长度。经分析可知，同样不需要更新，因为 $w[i][k]<w[i][k]+w[k][k]$，$w[k][j]<w[k][k]+w[k][j]$。不同的 $w[i][k]$和 $w[k][j]$之间并没有固定的顺序。

3)更新 $w[i][j]$，$i,j=0,1,\cdots,n-1$，且 $i\neq k$，$j\neq k$，即图中顶点 v_i 和 v_j 之间的最短路径长度。如下式可知，$w[i][j]<w[i][k]+w[k][j]$，除顶点 v_k 外的任意两个顶点之间的最短路径长度 $w[i][j]$的更新只与 $w[i][k]$和 $w[k][j]$有关，而不同的 $w[i][j]$之间并没有固定的先后顺序。

如图 1 所示，以 $N=16$ 的矩阵为例。

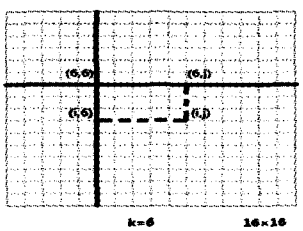

图 1 Floyd 串行算法

当 $k=6$ 时，整个 16×16 的矩阵都要更新一遍。此时更新分为 3 个阶段：

第一阶段是更新 $w[6][6]$，即图 1 中两条直线的交点。$w[6][6]$不需要更新，因为必然有 $w[6][6]<w[6][6]+w[6][6]$。

第二阶段是更新 $w[i][6]$和 $w[6][j]$，$i,j=1,2,\cdots,n$，且 $i\neq 6$，$j\neq 6$，即图 1 中两条直线上的点。同样不需要更新，因为 $w[i][6]<w[i][6]+w[6][6]$，$w[6][j]<w[6][6]+w[6][j]$。

第三阶段更新 $w[i][j]$，$i,j=1,2,\cdots,n$，且 $i\neq 6, j\neq 6$，即图中不在两条直线上的点。第三部分中，任意 $w[i][j]$ 的更新只与 $w[i][6]$ 和 $w[6][j]$ 有关。

经过上述分析可以发现第三部分的大量点的更新是整个 Floyd 算法中耗费时间最多的部分。为了实现 GPU 上的加速效果，根据 Amdahl 定律可知，为获得最大的加速比，需要对第三部分进行加速优化。

3 最短路径问题的 GPU 算法实现及分析

3.1 基于 GPU 矩阵乘法的最短路径算法

基于矩阵乘法的最短路径算法有 $\Theta(n^3)\cdot\log(n-1)$ 的计算复杂性，因此执行效率相对 Floyd 算法要低。但是考虑到矩阵乘法在 GPU 上有着高效的并行实现，因此本文将基于矩阵乘法的最短路径算法移植到 GPU 上，同样得到了相应的加速效果。

基本的矩阵乘法使用了带状划分：

$$C_{ij}=\sum_{k=1}^{n}A_{ik}\cdot B_{kj} \tag{4}$$

如式(4)所示，每个线程负责读取 A 中的一行和 B 中的一列，并计算出 C 中相应位置上的值。于是，整个 kernel 只有从全局存储器对 A 矩阵进行 B. width 次读取、对 B 矩阵进行 A. height 次读取，才能完成对矩阵 C 的计算。

在 GPU 显存中，访存共享存储器的延迟远小于全局存储器，并且使用共享存储器进行线程间通信，还可以通过更灵活的算法获得更好的性能。于是对矩阵进行了棋盘划分，即将矩阵分块使用共享存储器来实现矩阵乘法[9]，如图 2 所示。

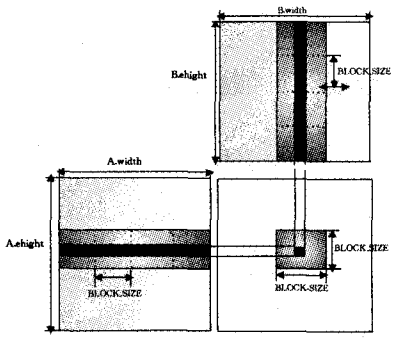

图 2 棋盘划分矩阵乘法

具体实现如下：首先从全局内存加载两个相应的正方形矩阵到共享内存，一个线程负责加载一个元素；接下来每个线程负责计算乘积中的一个元素。每个线程累积每个这样的乘积结果到一个寄存器中，一旦结束就将结果写回全局内存。这种棋盘划分方式利用了高速的共享内存，同时节约了大量的全局内存带宽，因为 A 只被读 B. width/block_size 次，B 被读了 A. height/block_size 次，从而降低了 GPU 端的通信延迟，提高了运行效率。

3.2 Floyd 算法在 GPU 上的并行实现

接下来本文从两方面进行优化。一方面，经过第 2.2 节分析知道，第三部分的每个点的更新顺序没有先后之分，因此正适合使用 GPU 的众多计算核心来并行计算。另一方面，并行 Floyd 算法通常采用域分解，即把输入的矩阵块分解成一系列子任务来处理。

国外 Katz 提出了一种基于 Floyd 算法的带状划分的 GPU 并行算法，其主要思想是将整个 Floyd 矩阵进行条带状划分，每个线程负责读取矩阵中的一行或者矩阵中的一列，然后计算出相应位置的值。根据其 3.5～15 倍的加速比可知，加速效果有限，有待进一步改进。

由于 GPU 的全局内存-共享内存工作模式与棋盘划分模式相似，因此本文采用棋盘划分来实现。这种棋盘划分方式利用了高速的共享内存，同时节约了大量全局内存带宽，因为 A 只被读 B. width/block_size 次，B 被读了 A. height/block_size 次，从而降低了 GPU 端的通信延迟，进一步提高了运行效率。

并行 Floyd 算法的思想和串行 Floyd 算法一致，其不同在于对矩阵不是以点为单位进行更新的，而是按照数据块为单位进行更新的。设数据块的长度为 $BLOCK\_SIZE$，即一个数据块内有 $BLOCKSIZE\times BLOCKSIZE$ 个点，$n=N/BLOCKSIZE$。

整个图的更新顺序不变：

1)更新 $w[k][k]$，即图中 v_k 数据块到 v_k 数据块的最短路径长度。与串行 Floyd 不同的是，$w[k][k]$需要更新，即对数据块内 $BLOCKSIZE\times BLOCKSIZE$ 个点之间进行一次 Floyd 更新。

2)并行更新 $w[i][k]$和 $w[k][j]$，$i,j=0,\cdots,n-1$，且 $i\neq k, j\neq k$，即图中的 v_i 数据块到 v_k 数据块和 v_k 数据块到 v_j 数据块之间的最短路径长度。与串行 Floyd 不同的是，它们需要更新。因为此时数据块内有很多点。$w[i][k]<w[i][k]+w[k][k]$，$w[k][k]<w[k][k]+w[k][j]$。更新 $w[i][k]$数据块和 $w[k][j]$数据块只与 $w[k][k]$有关，因而可以并行。

3)并行更新 $w[i][i]$，$i,j=0,1,\cdots,n-1$，且 $i\neq k, j\neq k$，即图中 v_i 数据块和 v_j 数据块之间的最短路径长度。$w[i][j]<w[i][k]+w[k][j]$，除 v_k 数据块外的任意两个数据块之间的最短路径长度 $w[i][j]$的更新只与 $w[i][k]$和 $w[k][j]$有关，因而可以并行。

如图 3 所示，以 $N=16\times BLOCKSIZE$ 的矩阵为例，其中数据块的大小为 $BLOCKSIZE\times BLOCKSIZE$，则可以将其视作 $N=16$ 的数据块矩阵。

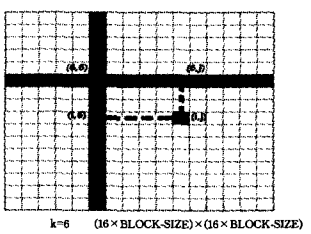

图 3 Floyd 在 GPU 并行算法实现

当 $k=6$ 时，整个 16×16 的块矩阵都要更新一遍。此时更新分为 3 个阶段：

第一阶段是更新 $w[6][6]$数据块，即图中两条直线的交点。

第二阶段是并行更新 $w[i][6]$和 $w[6][j]$数据块，$i,j=1,2,\cdots,16$，且 $i\neq 6, j\neq 6$，即图中两条直线上的数据块上的点。

第三阶段并行更新 $w[i][j]$数据块，$i,j=1,2,\cdots,16$，且

$i\neq6, j\neq6$，即图中不在两条直线上的数据块。

与串行算法不同的是，在并行计算中，由于每个数据块是由内部众多的点组成，因此在 $k=6$ 时，第一部分 $w[6][6]$ 数据块和第二部分 $w[i][6]$ 与 $w[6][j]$，$i,j=1,2,\cdots,16$，也都必须更新，因为此时小方块内有很多点。

因此在并行计算过程中，第一阶段更新时，将 $w[k][k]$ 数据块由全局内存拷贝进共享内存中，利用众多线程进行数据块内的 Floyd 并行更新。第二阶段更新时，将 $w[i][k]$ 和 $w[k][j]$，$i,j=1,2,\cdots,n, i\neq k, j\neq k$ 数据块由全局内存拷贝进共享内存中，利用众多线程分别与 $w[k][k]$ 进行数据块间的 Floyd 并行更新。第三阶段更新时，将 $w[i][j]$ 数据块由全局内存拷贝进共享内存中，利用众多线程分别和 $w[i][k]$ 和 $w[k][j]$，$i,j=1,2,\cdots,n, i\neq k, j\neq k$ 数据块进行数据块间的 Floyd 并行更新。

第一部分计算在单个流多处理器中完成，调用了部分 GPU 的并行计算能力，第二部分和第三部分均同时调用了所有 GPU 的流多处理器，因此 GPU 的并行计算能力得到了充分体现，这是算法实现高加速比的关键因素。

算法 3 Floyd 在 GPU 的并行算法

输入顶点数目 n，以及确定数据块的大小为 BLOCK_SIZE。

(1) 将数据拷贝至 GPU 显存

(2) for(k=0;k<n/BLOCK_SIZE;k ++)循环开始

a) 第一部分：并行更新 w[k][k]数据块

b) 第二部分：利用 w[k][k]并行更新 w[i][k]和 w[k][j]数据块，i,j=1,2,…,N/BLOCK_SIZE,i≠k,j≠k

c) 第三部分：利用 w[i][k]和 w[k][j]并行更新 w[i][j]数据块，i,j=1,2,…,N/BLOCK_SIZE,i≠k,j≠k

(3) 跳(2)至 k 循环结束转(4)

(4) 将数据从 GPU 显存拷贝回内存

4 实验结果和分析

在一台 PC 上进行了测试，GPU 上的程序用 CUDA 编写，显卡是 NVIDIA GeForce GTX 560 Ti，CPU 是 Intel Core 2 Q8000，并在 Window server2008rs2 系统环境下运行后得出最终实验数据。

从表 1 中可以发现，随着矩阵规模的扩大，不管是 CPU 还是 GPU 端的算法运行的时间都以 $O(n^3)$ 的趋势增长。其中，CPU 端的基于矩阵乘法算法的运行时间最长，CPU 端的 Floyd 算法运行时间次之，GPU 端的基于矩阵乘法算法的运行时间再次之，GPU 端的 Floyd 算法时间最短，效率最高。因而，最实用的 CPU 端的和 GPU 端的算法都是 Floyd 算法。GPU 上基于矩阵乘法的算法即使经过时间也长于 GPU 上的 Floyd 算法。

表 1 算法的运行时间对比表

矩阵规模 N	CPU 端矩阵乘 时间/s	CPU 端 Floyd 时间/s	GPU 端矩阵乘 时间/s	GPU 端 Floyd 时间/s
256	0.624	0.078	0.109	0.124
512	5.193	0.577	0.156	0.14
1024	46.02	4.602	0.359	0.14
2048	416.8	37.893	1.747	0.375
3072	1518	128.061	5.709	0.982
4096	3621	301.739	12.745	2.106
5120	7303	592.723	25.865	3.962

如上分析，考虑到实际意义，基于矩阵乘法的最短路径算法因为计算效率的原因不会被经常使用，因而需要对 CPU 和 GPU 端的最短路径算法进行进一步分析。可以发现，在矩阵规模比较小的时候，加速不是很明显。这是因为 GPU 显存和 CPU 内存之间的通信效率比较低。在计算量小、数据传输时间占运行时间的比重较大时，根据 Amdahl 定律可知定理，加速效果并不明显。如图 4 所示，随着矩阵规模的扩大，数据在 CPU 内存和 GPU 显存之间的传输时间所占运行时间的比例越来越小，加速效果越来越明显，最后稳定到 155 倍左右。

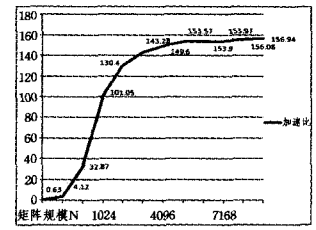

图 4 Floyd 算法 GPU 并行加速效果

5 算法的进一步改进

5.1 分批处理数据

在不同的应用中，数据矩阵规模可能不同。在单 GPU 的情况下，如果数据量过大，可能会出现内存中可以放下而显存中放不下的情况。此时为了能够在 GPU 上加速计算，需要对我们的算法进行改进。改进的核心想法就是分批处理计算，即分批次将数据从内存拷贝进入显存，待该部分数据 GPU 计算结束后，再分批从显存拷贝回内存。

单纯的分批处理会造成加速比大大下降，因为数据的局部性变差，而更多地需要进行全局性的计算，从而需要将大量数据在内存和显存之间多次拷贝。在 Floyd 的 GPU 端实现时，每个最外层循环中都要将全部的矩阵数据在内存和显存中进行一次拷贝。矩阵规模为 N 时，总共进行 N 次拷贝。而之前则只进行一次拷贝。由于 CPU 存取显卡内存时只能透过 PCI Express 接口，因此速度较慢(PCI Express x16 的理论带宽是双向各 4GB/s)，降低了执行效率，延长了执行时间。

5.2 异步并行处理数据

为了优化解决这一问题，本文采用了 GPU 端异步并行执行的方法。

此方法类似于 CPU 的流水线工作模式，即将每批数据拷贝处理过程看成一个流，程序通过流来管理并发。处于同一个流内的计算和数据拷贝是依次执行的，但是一个流内的计算可以和另一个流的数据传输同时进行。不同的流和其他的流之间可能是并行执行的，也可能是乱序执行的。

程序执行中多个流并发执行。GPU 并行调度采用的是 Round-robin 轮询调度算法[13]，每个流在分配给它的时间段执行，然后切换到下一个就绪的流。其中每个流主要分为从内存向显存拷贝数据、显存内的计算、从显存向内存拷回数据 3 大子任务。因为 GPU 独特的硬件结构，即多线程的特点，能够实现零开销的线程随时切换，可以用来隐藏延迟，则当线程相对数据来说有限、计算密度较高时，延迟就能被计算隐藏。因此每个子任务相当于被划分成了多个小子任务。这样每个流就被划分为多个小子任务，从而获得了较好的吞吐率

和重叠性。

因此通过异步执行能够使 GPU 的执行单元和存储器单元同时工作，提高资源利用率，这样可以在一定程度上减轻由于大量数据拷贝带来的计算效率的降低。

在第一阶段和第二阶段更新数据块矩阵时，计算数据量较小，不适合异步处理。而主要的数据量集中在第三阶段，并且注意到该部分最短路径计算更新不存在相互之间的先后顺序，因此可以采用异步处理，异步并行地进行计算和数据的拷贝，从而有效地隐藏数据交换带来的延迟。

同时，为了进一步提高通信效率，根据第一阶段和第二阶段数据量比较小的特点，第一阶段和第二阶段都在全局内存中操作，并不拷贝出来，只是在第三阶段时，才将大量的数据在内存和显存直接分配拷贝和处理。

表 2 给出了部分大规模数据矩阵的测试结果。最后在矩阵规模 $N=15360$，即矩阵大小为 15360×15360 时，数据量已经到达 1.5625G，这对单 PC 来说已经是很大的数据量，所以可以验证算法具有较强的扩展性和实用性。

表 2 大规模数据矩阵下的加速比

矩阵规模 N	CPU 端 Floyd	GPU 端 Floyd	加速比	GPU 端流处理 Floyd	加速比
	时间/s	时间/s		时间/s	
11264	6125.262	405.054	15.12	244.7	25.03
12288	7952.256	515.83	15.41	291.83	27.35
13312	10110.59	631.91	16.23	367.1	27.54
14336	12627.89	774.71	16.35	448.88	28.15
15360	15531.75	919.038	16.9	533.92	29.09

如表 2 所列，在使用异步并行处理之前，由于浪费了大量的通信时间，只有 15 倍左右的加速。而使用了流处理后，节省了一部分通信时间，可以达到 25 倍左右的加速。

结束语 针对全源最短路径问题的两种串行算法，本文提出一种基于 GPU 的并行算法。此并行算法实现简单，能够处理不同规模的数据并且在大规模并行计算上有着广泛的应用前景。实验结果表明，此并行算法降低了计算所有点对之间最短路径问题的时间，提高了计算效率，实现了高达 155 倍的并行加速比。

算法的不足之处在于如果数据矩阵规模过大，则算法的执行效率稍低，通信开销过大。如何克服这个缺陷，也是今后需要解决的问题。同时，将算法进一步扩展到单 PC 多显卡和 GPU 集群上，也是一项重要的研究内容。

参 考 文 献

[1] Floyd R W. Algorithm 97(SHORTEST PATH)[J]. Communications of the ACM, 1962

[2] Lawler E L. Combinatorial Optimization: Networks and Matrodis[M]// Holt, Rinehart and Winston, 1976

[3] Harish P, Narayanan P J. Accelerating large graph algorithms on the GPU using CUDA[C]// Proc. 14th Int'l Conf. High Performance Computing (HiPC'07). Dec. 2007: 197-208

[4] 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]// Proceedings of 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications. IEEE, 2008: 284-291

[5] Katz G J, Kider J T, Jr. All-pairs shortest-paths for large graphs on the GPU[C]// Proc. of the 23rd ACM

[6] 周益民，孙世新. 一种实用的所有点对之间最短路径并行算法[J]. 计算机应用，2005，25(12)：2911-2913

[7] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to Algorithms(Second Edition)[M]. The MIT Press, 2001

[8] Breshears C. The Art of Concurrency[M]. O'Reilly Media, Inc, 2001

[9] 张树，褚艳利. 高性能运算之 CUDA[M]. 北京：中国水利出版社，2009

[10] 卢凤顺，宋君强，银福康，等. CPU/GPU 协同并行计算研究综述[J]. 计算机科学，2011，38(3)：5-9

[11] 唐策善，李龙澍，黄刘生. 数据结构—用 C 语言描述[M]. 北京：高等教育出版社，2000

[12] 胡文美. 大规模并行处理器编程实战[M]. 北京：清华大学出版社，2010

[13] Tanenbaum A S, Woodhull A S. Operating Systems Design and Implementation[M]. Englewood Cliffs, NJ: Prentice-Hall International Inc, 1997

(上接第 294 页)

[9] Anderson J M, Lam M S. Global optimizations for parallelism and locality on scalable parallel machines[C]// Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation. Albuquerque, New Mexico, USA, 1993: 112-125

[10] Anderson J M, Lam M S. Data and computation transformations for multiprocessors[C]// Proceedings of the 5th ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming. Santa Barbara, California, USA, 1995: 166-178

[11] Lim A W, Lam M S. Maximizing parallelism and minimizing synchronization with affine transforms[C]// Proceedings of the Conference Record of the 24th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages. Paris, France, 1997: 201-214

[12] Lim A W, Lam M S. Maximizing parallelism and minimizing synchronization with affine partitions[J]. Parallel Computing, 1998, 24(3/4): 445-475

[13] Ulrich K. Automatic data layout with read-only replication and memory constraints[C]// Proceedings of the 10th international Workshop on Languages and Compilers for Parallel Computing. Chapel Hill, NC, USA, 1998: 419-422

[14] Olav B, Jpaul H. A liner algebra formulation for optimizing replication in data parallel programs[C]// Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing. Youktown Heights, NY, USA, 2000: 100-116