基于CUDA的并行最短路径算法

陈凯*

计算机科学与技术学院，复旦大学

摘要 本文提出了4种CUDA实现的并行最短路径算法。它们分别基于Dijkstra、Bellman-Ford、Δ-Stepping、Sparse Matrix-Vector Bellman-Ford。本文首先对于经典的Dijkstra和Bellman-Ford分别进行了并行化的改进，之后对当前性能最佳的Δ-Stepping并行算法实现了CUDA平台的改进版本，最后提出了基于Sparse Matrix-Vector的Bellman-Ford，分别实现了CSR和ELL版本的CUDA算法。在实验部分，CUDA算法与当前性能最优的Boost库中的各种最短路径算法进行了比较，证明了本文的算法在时间和空间上都有较强的竞争力，且在大数据上性能优势更明显。

关键词 最短路径，Dijkstra，Bellman-Ford, Δ-Stepping，Sparse Matrix-Vector, CSR Bellman-Ford, ELL Bellman-Ford, CUDA，GPU，并行编程

1 简介

近年来，并行计算的革命迅速发展。NVIDIA、Microsoft、Intel等一些公司都分别推出了自己的并行计算平台。在2011年，几乎所有的计算机——从上网本、笔记本、台式机、到工作站计算机都采用了多核处理器。最近智能手机也成为了多核处理器新的竞争战场。例如，NVIDIA在2010年推出针对移动平台的Tegra 2双核处理器等等。

与传统的中央处理器的数据处理流水线相比，图形处理器（GPU）的并行计算在最近几年成为了一个新概念。早期的GPU编程极为繁琐，程序只能以颜色值和纹理单元等形式作为输入数据，另外程序的写入内存方式有着严格的限制。而且GPU无法处理浮点数据。调试也极为繁琐。程序员还需学习OpenGL或者DirectX。因此，早期GPU计算并未流行。

在2006年，NVIDIA公布了第一款基于CUDA架构（Compute Unified Device Architecture）的GPU——GeForce 8800 GTX。CUDA架构为GPU计算设计了一种全新的模块，使其能在通用计算中更为方便。该架构使GPU能够解决复杂的计算问题，它包含了CUDA指令集架构（ISA）以及GPU内部的并行计算引擎。开发人员现在可以使用C/C++语言来为CUDA架构编写程序，并可以在支持CUDA的处理器上以超高性能运行。

*Email: remlostime@gmail.com

自2007年以来，以CUDA C为基础的应用程序，获得了计算速度上的极大提升。例如：医学图像、计算流体动力学、环境科学等一些领域都可以通过GPU并行计算得到性能上的提升。本文对于最短路径问题，提出了基于CUDA平台的并行算法，相对于串行算法，在速度上得到了进一步的提升。

本文余下部分的组织如下。第2章总结了过去关于最短路径的工作，同时比较了各种算法的优劣。第3章介绍了基于Dijkstra算法的CUDA版本。第4章介绍了Bellman-Ford的CUDA并行算法。在第5章中，描述了Δ-Stepping算法的CUDA改进版本。第6章提出了基于Sparse Matrix-Vector的Bellman-Ford并行算法。在最后的实验部分，比较了上述4种并行算法和Boost库中的串行和并行最短路径算法的性能。

2 相关工作

给出一个有向图$G = (V, E)$，其中$|V| = n$，$|E| = m$。让$s \in V$代表源节点。每条边$e \in E$被赋予一个非负的权重，用权重函数表示为$c : E \to \mathbb{R}$。在这里，我们定义一条路径的权重为路径中所有边的权重之和。对于单源点最短路径问题就是计算从源点s到目标节点v的权重最小的一条路径。

大多数的最短路径算法维护一个当前到s的最短路径数组d，$d(v)$代表从s到v的权重，并且算法每次都进行松弛操作（relax）更新d。算法开始时，$d(s) \leftarrow 0$, $d(v) \leftarrow \infty$。每次松弛一条边$e(v, w) \in E$，把$d(w)$设置为$d(w)$和$d(v) + c(v, w)$ 中较小的值，直到没有节点需要更新，算法结束。如果s到v不可达，则算法结束时$d(v) = \infty$。基于d的更新方式，大多数的最短路径算法可分为两种：label-setting和label-correcting。Label-setting算法（例如：Dijkstra）每次只对最短路径已经确定的节点v的相关边进行松弛，所以Label-setting只需松弛m条边。而Label-correcting算法（例如：Bellman-Ford）每次对于未确定最短路径的节点v同样会进行松弛，所以整个过程可能松弛超过m条边。

Cherkassky [32]总结了当前的各种最短路径算法。其中，最经典的一些单源点最短路径算法包括：Dijkstra [1, 14, 19]、Bellman-Ford [5, 14, 23]、以及基于启发式函数的$A^\star$ [6, 15, 7]等等。而基于这些经典算法的改进也颇多。例如：对于Dijkstra的改进包括了基于Min-Max Heap [18]、Pairing Heap [20]、Fibonacci Heap [21]的算法。一般来说这些算法提取优先队列（Q）中到s的路径最短的节点v的时间复杂度为$O(logn)$，于是算法的整体时间复杂度降为$O(nlogn + m)$。另外， [24]和 [25]分别对Bellman-Ford提出了改进算法。由于$A^\star$算法本质上是Dijkstra的泛化版本，当$h(x) = 0$(启发式函数：$h(x) \leq d(x, y) + h(y)$)时，$A^\star$就演化为Dijkstra算法了。$A^\star$的应用也很广泛，例如：游戏中的寻路算法，国际象棋的博弈树等等。$A^\star$算法的关键在于启发式函数$h(x)$的设计。但在本问题中，除了节点间的权重之外，没有更多额外的信息，启发式函数较难设计，所以$A^\star$不太适合我们的问题。

近些年，最短路径的并行算法，例如Crauser [30]、Eager [31]、Δ-Stepping [17]也被一一提出。其中，Crauser性能比较稳定，但需要维护3个优先队列，对空间要求较大，

同时对于每个节点的维护工作也较重。Eager则需要输入一个lookahead参数，用来每次寻找距离在lookahead内的节点。当lookahead太小会限制算法的并行性，太大则遍历次数会加大并且做许多额外的工作。对于lookahead的选择则是因图而异。所以Eager的算法效率依赖于lookahead 的选择，并不稳定。Δ-Stepping是并行最短路径算法中效率最高的。它根据参数Δ，在算法的第i次松弛操作时，寻找d权重在$i \times \Delta$和$(i+1) \times \Delta$之间的节点松弛。 [29]和 [27, 8]各自都对Δ-Stepping算法进行了实现。但前者是基于$9th$ DIMACS的Cray MTA-2大型机做了特别优化的代码，无法在普通的个人计算机上运行。后者是基于CUDA的并行算法实现，由于作者的算法基于网格图（Grid Graph），只适用于图像分割的最短路径应用，无法对一般的有向图进行运算。

3 CUDA Dijktra

3.1 Dijkstra算法

Dijkstra [19]的基本思想是维护一个节点队列Q，每次从Q中取出当前到源节点s路径最短的节点u，并对u的相关边进行松弛。初始时$d(s) \leftarrow 0$，$d(v) \leftarrow \infty$，其中$v \neq s$。算法至多执行$n-1$轮松弛操作就可以找出所有节点v到s的最短路径，时间复杂度$O(n^2)$。

Algorithm 1 CUDA Dijkstra

(a) Initial:
foreach $v \in V$ **do**
    $d[v] \leftarrow \infty$
end
$d[s] \leftarrow 0$
$Q \leftarrow V$
while $Q \neq \phi$ **do**
    (b) ExtractMin:
    $u \leftarrow \{v : v \in Q \wedge \forall w \in Q, d(v) \leq d(w)\}$
    if $d[u] = \infty$ **then**
        break
    remove u from Q
    (c) Relax:
    foreach $(u, v) \in E$ **do**
        if $d[u] + c(u, v) < d[v]$ **then**
            $d[v] \leftarrow d[u] + c(u, v)$
    end
end

Algorithm1给出了CUDA Dijkstra的算法框架。其中，Initial、ExtractMin、Relax都

能并行执行。Initial的时间复杂度为$O(n/t)$，ExtractMin为$O(n^2/t)$，而Relax为$O(m/t)$，其中t表示核函数中线程(thread)总数。所以整个算法的时间复杂度为$O(n/t+n^2/t+m/t) = O(n^2/t)$。

3.2 Reduction

在并行编程中，Reduction [13, 10]是许多并行操作的基石。对于ExtractMin，我们同样可以用Reduction来加速。Reduction的基本思想是将大量的数据以分治的方法处理，每次减小为原规模的一半，最后当符合结束条件时算法结束。理论上，当$t \geq n$时，ExtractMin 的时间复杂度为$O(logn)$，但由于数据间的通信和同步等问题，实际的运算要慢些。

在串行的Dijkstra算法中ExtractMin需要遍历数组一次，时间复杂度为$O(n)$。对于Dijkstra的一些$O(logn)$的改进算法 [18, 20, 21]都是基于优先队列实现ExtractMin操作，而优先队列的数据结构并不适合并行算法，所以我采用了Reduction的方法实现ExtractMin。优点在于：编程容易，对于Dijkstra的原始算法和数据结构稍作改进即可。同时维护成本低，不需要额外的空间进行优先队列的存储。提取的时间复杂度也能控制在$O(logn)$。

在 [10]中介绍了7种Reduction的方法，对Reduction渐进地进行了各种不同的优化。我选择了最好的一种，并对不同的ThreadsPerBlock(TPB)参数进行性能比较。同时采用了Thrust库 [11]中的Reduction、CPU的Loop和Heap操作进行比较。从图1可以看出，在数据规模为10^7时，256、512、1024TPB都比Loop(-O2优化) 快，最快的512TPB加速比为1.65。而Thrust的加速比更是达到6.07。未经优化的Loop耗时甚至有56790ns之多。但Reduction相对于Heap还是要慢，这是由于线程间的通信，数据传输的延迟等问题造成。可以观察到，当数据规模较小时，Loop占优势，而数据规模变大时Reduction更占优势，所以并行编程更适合大数据。

	1	10	10^2	10^3	10^4	10^5	10^6	10^7
128(TPB)	40.82	35.92	39.63	43.12	59.4	145.4	1033.26	8033.73
256(TPB)	48.56	57.37	43.96	43.05	50.49	112.43	556.43	4839.28
512(TPB)	42.64	47.32	41.2	41.55	45.97	92.04	470.487	4075.55
1024(TPB)	43.81	43.76	45.52	49.77	46.74	108.75	734.59	6475.09
Thrust	324.8	65.97	73.04	513.12	453.91	450.91	498.5	1105.1
Loop	0	0	0	0	150	630	5620	56790
Loop(-O2)	0	0	0	0	0	0	630	6710
Heap	0	0	0	0	0	0	0.124	0.1311

表 1: ExtractMin Time(ns)。数据规模从1到10^7，进行1000次ExtractMin操作取平均值。

4 CUDA Bellman-Ford

4.1 Bellman-Ford算法

Bellman-Ford算法不同于Dijkstra，它的核心思想是对图中每条边的对应节点都进行松弛，直到没有节点需要更新，属于label-correcting算法。同时可以证明至多松弛$n-1$轮算法结束，时间复杂度为$O(nm)$。

Algorithm 2 CUDA Bellman-Ford

```
(a) Initial:
foreach v ∈ V do
    d[v] ← ∞
end
d[s] ← 0
for i ← 1 to |V| − 1 do
    (b) Relax:
    foreach (u, v) ∈ E do
        if d[u] + c(u, v) < d[v] then
            d[v] ← d[u] + c(u, v)
    end
end
```

Algorithm2给出了算法的大致框架，Initial和Relax部分与Dijkstra大致相同。同时，我在算法中采用了一种对Bellman-Ford的常用改进：用队列保存每次被更新的节点。在下一次的Relax过程中提取队列的首节点，并对其相关边进行松弛，直到队列为空。一般来说小于$n-1$轮算法就能结束，所以算法的时间复杂度稍小于$O(nm)$。算法对于边的松弛的复杂度为$O(m/t)$，所以并行Bellman-Ford算法的复杂度为$O(n*(m/t))=O(nm/t)$。

4.2 循环队列

Bellman-Ford的一个经典实现是利用队列保存需要松弛的节点。这种队列的实现一般会比原算法快一些，当被松弛的节点数小于n时，图的松弛工作就已经结束了。GPU中的队列需要满足以下几点：1）快速插入、提取元素。时间复杂度控制在$O(1)$。2）GPU队列中的元素能快速复制回CPU中进行操作。

基于以上两点，我使用了循环队列，并且将此队列放入零拷贝内存中。这样既保证了GPU在$O(1)$的时间插入，同时CPU也能在$O(1)$的时间提取。循环队列的数据结构是一个长度为n的数组。其中设有头指针$head$和尾指针$tail$。$tail$所指向的是一个类似哨兵的节点，不存放数据，所以队列中最多存放$n-1$个数据。当$head=tail$时，队列为空。由于数组长度为n，所以每次更新指针p的操作为：$p \leftarrow (p+1) \bmod n$。循环队列能很好地利用以前节点废弃的空间给新节点使用。这样队列对于空间的要求较小，而且循环队列的编程复

杂度较小，对插入、删除操作均能在$O(1)$完成，很适合GPU中的队列实现。在本文中，还对GPU队列进一步进行了改进。由于并行算法线程众多，对于单个队列的操作会产生许多通信、同步等问题，势必降低效率。所以，算法将队列实现为二维数组，每个$Block$对应一个队列，进一步提升了算法效率。

4.3 Large Label Last和Small Label First优化

[26]提出了对于Bellman-Ford的Large Label Last(LLL)和Small Label First(SLF)的优化。LLL：设队首元素为1，队尾元素n，$d_{avg} = \frac{1}{n}\sum_{v=1}^{n} d(v)$。若$dist(i) > d_{avg}$则将$i$插入到队尾，查找下一元素，直到找到某个$i$使得$d(i) \leq x$，则将$i$ 取出队列进行松弛操作。SLF：设要加入的节点是j，队首元素为i，若$d(j) < d(i)$，则将j插入队首，否则插入队尾。SLF 可使速度提高15%到20%，SLF + LLL 可提高约50%。

5 CUDA Δ-Stepping

5.1 Δ-Stepping算法

Δ-Stepping [17]主要基于Dijkstra进行了并行改进。与串行Dijkstra不同的是，在第i轮算法寻找$d(v)$在$i \times \Delta$和$(i+1) \times \Delta$之间的v进行松弛。Δ-Stepping用$Bucket[i]$来保存第i轮算法需要松弛的节点，当$Bucket = \phi$时算法结束。由于松弛次数的不确定性，所以Δ-Stepping属于Label-correcting算法。Δ决定了每次$Bucket$集合的大小，所以Δ的选择也成为了算法并行度和每次松弛效率的影响因子之一，在后面我们会进一步对Δ进行讨论。

Algorithm3给出了CUDA Δ-Stepping的算法框架，算法中的Initial、Add To Request、Relax都可以并行执行。

5.2 Initial

E中的每条边都是独立的，所以对于边的初始化可以并行执行，$heavy(v)$和$light(v)$的初始化在大约$O(m/t)$的时间内完成。在本文的算法中，由于内存、显存限制以及编程便利性等原因，Initial的实现是在之后的Relax中一并实现的，在函数中即时判断$c(v,w) > \Delta$或$c(v,w) \leq \Delta$来确定对应边的类型。当图非常大时，可以省去对于$light$和$heavy$所占用的大量的内存空间，同时由于$heavy$和$light$在初始化时不可避免地产生线程间通信、同步等问题都可以一并地省去。而在$relax$中每个线程对$c(v,w)$的判断所增加的时间很小，所以算法的整体速度不会降低。

5.3 Add To Request

根据原算法，Req保存(v,x)点对，其中v代表需要被更新的节点，x是更新的距离。但由于Req的长度是不确定的，最短为0，最长为m。如果在程序的最初分配一块约$2 \times m$大小（v和x）的内存给Req，相当于整个图的大小，开销过大。另外，Req的线性结构同样对于其增长的速度是一个瓶颈，需要维护一个$size$变量来保存Req的长度，同一时间只有

Algorithm 3 CUDA Δ-Stepping

(a) Initial:
foreach $v \in V$ **do**
    $heavy(v) \leftarrow \{(v,w) \in E : c(v,w) > \Delta\}$
    $light(v) \leftarrow \{(v,w) \in E : c(v,w) \leq \Delta\}$
    $d(v) \leftarrow \infty$
end
relax$(s, 0)$
$i \leftarrow 0$
while $B \neq \phi$ **do**
    $S \leftarrow \phi$
    while $B[i] \neq \phi$ **do**
        (b1) Add To Request:
        $Req \leftarrow \{(w, d(v) + c(v,w)) : v \in B[i] \wedge (v,w) \in light(v)\}$
        $S \leftarrow S \cup B[i]$
        $B[i] \leftarrow \phi$
        (c1) Relax:
        foreach $(v,x) \in Req$ **do**
            relax(v,w)
        end
    end
    (b2) Add To Request:
    $Req \leftarrow \{(w, d(v) + c(v,w)) : v \in S \wedge (v,w) \in heavy(v)\}$
    (c2) Relax:
    foreach $(v,x) \in Req$ **do**
        relax(v,x)
    end
    $i \leftarrow i + 1$
end
Procedure relax(v,x)
    if $x < d(v)$ **then**
        $B[\lfloor d(v)/\Delta \rfloor] \leftarrow B[\lfloor d(v)/\Delta \rfloor]\ v$
        $B[\lfloor x/\Delta \rfloor] \leftarrow B[\lfloor x/\Delta \rfloor] \cup v$
        $d(v) \leftarrow x$

一个线程能进行更新，并行度很低。同时，线程间的通信、同步等问题也是时间瓶颈之一。

于是，我想到了用Req保存父节点的方法，即保存待更新节点w的前向节点v。在之后的$Relax$中进行相应的修改即可。这样做的好处有三个：第一，节省空间。相比之前最大有$2 \times m$的空间要求，算法最多只需要n的空间（即最大节点数目）。第二，降低算法复杂度。对于Req的增加只需判断v是否属于$B[i]$就可加入，其余的实现由$Relax$完成。最后，也非常重要的，我用基于Reduction的prefix sum [12]的方法计算出$ReqIndex$，得到了对应节点在Req中存放的位置，对Req的更新可以并行执行，由此解决了通信、同步的问题，提高了并行效率。

5.4 Relax

首先，对于每个线程tid根据$reqSize$找到在Req中对应的节点位置和对应的需更新节点数量。$reqSize$类似于之后提到的Compressed Sparse Row中的C_p索引数组。$reqSize[i]$记录前$i-1$个节点的链表总长。在更新时进行$c(v,w)$的判断来确定$heavy$或$light$边，以进行相应的$heavy$或$light$阶段的更新。$light$阶段的时间复杂度是$O(l/t)$，$heavy$为$O(h/t)$，其中l和h分别是$light$和$heavy$更新时边的数量，t为线程总数。

5.5 Bucket的数据结构

在算法中，另一个关键点是$B[i]$的数据结构。算法中，Bucket为集合类型。这样的结构很适用于C++ STL中的set类，但对于CUDA编程则并不合适。我们在relax时希望对Bucket进行$O(1)$时间的插入、删除工作。由于CUDA本身不提供set类型，所以设计这样一个数据结构将是非常大的挑战，而且内存、时间、通信等问题都可能成为瓶颈。基于这些问题，我对原算法进行了一些改进以适用于CUDA编程。仔细观察$B[i]$可以发现，每个节点v在同一时间只能唯一属于一个Bucket，不存在两个$B[i]$和$B[j]$同时包含v的情况，其中$i \neq j$。另外在主循环中对于$B \neq \phi$和$B[i] \neq \phi$的判断也是设计Bucket时需要考虑的问题。在我的算法中，设计了一个$BIndex(v)$的数据用来保存v所属的Bucket索引号。这样做的好处有两个：第一，节省内存。$BIndex$只需要一个n维空间的数组即可，相对于原算法的$B[i]$的集合类型所占用的空间（最大到$max(p)/\Delta$。其中，$p \in P$，P代表所有可能的路径）节省了许多。第二，操作方便。对于集合$B[i]$的各种操作的设计都将是非常复杂的，且可能无法保证在$O(1)$时间内完成。而对于$BIndex(v)$的查找、插入、删除以及之后与S的合并操作均能保证$O(1)$。

5.6 Δ的选择

Δ可以有三种选择：1)$\Delta = MEDIAN(c(e))$，2)$\Delta = \frac{1}{n}\sum_{e \in E} c(e)$，3)$\Delta = max\{c(e) : e \in E\}/max\{d : d \in Degree\}$。同时，Meyer [17]证明re-insertion的总数限制在$|P_\Delta|$，re-relaxation的总数限制在$|P_{2\Delta}|$。其中，$|P_\Delta|$代表代表路径长度最大为Δ的集合。对于任何的Δ，phases 的数量被限定在$d_c/\Delta l_{max}$，其中$d_c = max\{d(v) : d(v) < \infty\}$。所以，

$\Delta = O(1/d)$是效率和并行度平衡的一个选择。在本文中，选择了方案3，$\Delta = max\{c(e) : e \in E\}/max\{d : d \in Degree\}$，即最大权重边除以最大的节点出度的商。

6 CUDA Sparse Matrix-Vector Bellman-Ford

6.1 Sparse Matrix-Vector简介

Bell [34]提出了基于CUDA的Sparse Matrix-Vector的运算，对于稀疏矩阵的表示包括：Diagonal Format(DIA)、ELLPACK Format(ELL)、Compressed Sparse Row Format(CSR)、Coordinate Format(COO)、Hybrid Format、Packet Format(PKT)。

受 [33]的启发，稀疏矩阵的运算稍加变换同样可以用于解决最短路径的问题。对于经典的Dijkstra算法，由于优先队列的数据结构并不适合矩阵运算。而Bellman-Ford则很适合将松弛函数改写为矩阵运算：

$$d_i \leftarrow \min_{j \in N_i}(d_j + c_{ij})$$

稀疏矩阵c_{ij}表示v_j到v_i的权重。当没有d_i需要更新时，算法结束。最坏的情况需要$O(n)$次迭代，但在实际中迭代的次数要远少于$O(n)$。

从表2可以观察到E/V非常小，平均每个节点只有2.5个左右相连的节点。在 [34]中提到对于E/V很小的图，DIA的效率最佳。而DIA的大小与图中对角线的数量成正比，D_{num}的数量远远超过了内存的承受，所以在本文中并没有采用DIA的数据结构，而是采用了比较常用的CSR和ELL数据结构的Bellman-Ford算法。

6.2 CSR Bellman-Ford算法

Compressed Sparse Row(CSR)是对于稀疏矩阵最常见的表示方法。图1是有向图的邻接矩阵C，图2中C_v按行的顺序记录了C的非0元素，C_j记录了C_v相应元素的列索引。最后，$n+1$维的C_p数组记录了非0元素的数量，即$C_p[i]$ 记录前$i-1$行非0元素总数。

$$C = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 5 \\ 6 & 0 & 0 & 7 \end{bmatrix}$$

图 1: 有向图邻接矩阵C

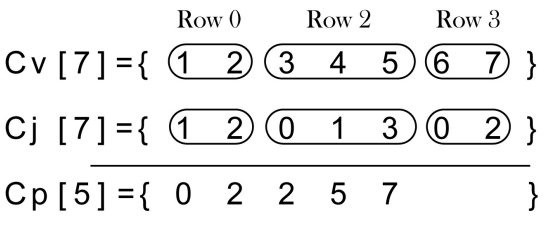

图 2: 邻接矩阵C的CSR表示

对于CSR形式的Bellman-Ford矩阵运算有两个版本：Listing1中每个线程更新一行，而Listing2 是以一组warp(32)为单位对一行进行更新。所以当$E/V < 32$时，Listing1效率更高，而$E/V \geq 32$，Listing2更好。

Listing 1: CSR Scalar Bellman-Ford Relax

```
__global__ void relax_csr_scalar(int *d_dst, int *Cv, int *Cj, int *Cp,
    int *d_src, int num_vertices)
{
    int vertice_idx = threadIdx.x + blockIdx.x * blockDim.x;
    int offset = blockDim.x * gridDim.x;

    while (vertice_idx < num_vertices)
    {
        int vertice_begin = Cp[vertice_idx];
        int vertice_end = Cp[vertice_idx+1];

        int min_dist = INT_MAX;

        for(int i = vertice_begin; i < vertice_end; i++)
            min_dist = min(min_dist, d_src[Cj[i]] + Cv[i]);

        d_dst[vertice_idx] = min_dist;

        vertice_id += offset;
    }
}
```

6.3 ELL Bellman-Ford算法

ELLPACK Format(ELL)是基于对行的非0元素的压缩进行存储的。对一个$M \times N$的矩阵，且行的非0元素最大为K时，ELL需要$M \times K$的空间。ELL对空间的需求也是较大的，比如，表2中NY的$I_{max} = 8$，所以需要的空间为$V \times I_{max} = 264346 \times 8 = 2114768$。

对于图1的ELL表示如图3和图4。C_v是每行的非0元素压缩存储，C_j是对应C_v元素在C中的列索引。

ELL的矩阵运算可以用Listing3实现，每个线程对应一行，每次线程更新一列。

7 实验

7.1 实验设置

测试数据来自2006年9th DIMACS [9]中的Distance Graph。如表2所示，图中的节点从NY的十万级别递增到USA的千万级别。从E/V可以看出平均每个节点和2.5个左右的节点相连，所以测试的地图十分的稀疏。表中的O_{min}、O_{max}、I_{min}、I_{max}代表节点的最小、最大出度以及节点的最小、最大入度。D_{num}代表非0对角线的个数。

Listing 2: CSR Vector Bellman-Ford Relax

```
__global__ void relax_csr_vector(int *d_dst, int *Cv, int *Cj, int *Cp,
    int *d_src, int num_rows)
{
    __shared__ volatile int min_dist[];

    int thread_id = threadIdx.x + blockDim.x * blockIdx.x;
    int thread_lane = threadIdx.x & (32 - 1);
    int warp_id = thread_id / 32;
    int warp_lane = threadIdx.x / 32;
    int num_warps = (blockDim.x / 32) * gridDim.x;

    int row = warp_id;

    while (row < num_rows)
    {
        int row_begin = Cp[row];
        int row_end = Cp[row+1];

        int min_dist[threadIdx.x] = INT_MAX;

        for(int i = row_begin + lane; i < row_end; i++)
            min_dist = min(min_dist, d_src[Cj[i]] + Cv[i]);

        if (lane < 16) min_dist[threadIdx.x] = min(min_dist[threadIdx.x],
            min_dist[threadIdx.x + 16]);
        if (lane < 8) min_dist[threadIdx.x] = min(min_dist[threadIdx.x],
            min_dist[threadIdx.x + 8]);
        if (lane < 4) min_dist[threadIdx.x] = min(min_dist[threadIdx.x],
            min_dist[threadIdx.x + 4]);
        if (lane < 2) min_dist[threadIdx.x] = min(min_dist[threadIdx.x],
            min_dist[threadIdx.x + 2]);
        if (lane < 1) min_dist[threadIdx.x] = min(min_dist[threadIdx.x],
            min_dist[threadIdx.x + 1]);

        if (thread_lane == 0)
            d_dst[row] = min(d_dst[row], min_dist[threadIdx.x]);

        row += num_warps;
    }
}
```

$$Cv = \begin{bmatrix} 1 & 2 & * \\ * & * & * \\ 3 & 4 & 5 \\ 6 & 7 & * \end{bmatrix}$$

图 3: 存储数据向量C_v

$$Cj = \begin{bmatrix} 1 & 2 & * \\ * & * & * \\ 0 & 1 & 3 \\ 0 & 2 & * \end{bmatrix}$$

图 4: 列索引向量C_j

Listing 3: Ell Bellman-Ford Relax

```
__global__ void relax_ell(int *d_dst, int *Cv, int *Cj, int *d_src, int
    num_rows, int num_cols_per_row)
{
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    int offset = blockDim.x * gridDim.x;

    while (row < num_vertices)
    {
        int min_dist = INT_MAX;

        for(int i = 0; i < num_cols_per_row; i++)
        {
            int col = Cj[num_rows * i + row];
            int val = Cv[num_rows * i + row];

            if (col != -1)
                min_dist = min(min_dist, val + d_src[col]);
        }

        d_dst[row] = min_dist;

        row += offset;
    }
}
```

G	V	E	E/V	O_{min}	O_{max}	I_{min}	I_{max}	D_{num}
NY	264346	733846	2.77	1	8	1	8	28792
BAY	321270	800172	2.49	1	7	1	7	42052
COL	435666	1057066	2.43	1	8	1	8	37834
FLA	1070376	2712798	2.53	1	8	1	8	69434
NW	1207945	2840208	2.35	1	9	1	9	483184
NE	1524453	3897636	2.56	1	9	1	9	870726
CAL	1890815	4657742	2.46	1	8	1	8	461930
LKS	2758119	6885658	2.50	1	8	1	8	1418978
E	3598623	8778114	2.44	1	9	1	9	1514518
W	6262104	15248146	2.43	1	9	1	9	2217466
CTR	14081816	34292496	2.44	1	9	1	9	1036394
USA	23947347	58333344	2.44	1	9	1	9	3446604

表 2: Distance Graph参数统计

测试环境如表3所示。

CPU	Intel Core i5 760
GPU	NVIDIA GeForce GTX 550 Ti
Memory	2 * 2GB DDR3-1333
Graphics Memory	1GB
OS	Windows 7 Ultimate(64bit)
CUDA	CUDA 4.0
IDE	Visual Studio 2010

表 3: 测试环境

对比算法采用了当前性能最好的Boost 1.47 [2, 3, 4]，包括串行算法:Dijkstra($O(nlogn)$)、Bellman-Ford($O(nm)$)，并行算法:Δ-Stepping($O(log^3n/loglogn)$)、Crauser($O(nlogn)$)。

7.2 性能分析

表4给出了Boost库的运行时间，可以看出Dijkstra是其中速度最快的，Δ-Stepping也很高效，速度会随CPU内核的增加而进一步提升。同时观察到，在CTR和USA这两个地图，算法的时间急速上升，这是由于这两个地图节点数量消耗内存巨大。在CPU Xeon E5430，8GB Memory，OS Windows Server 2003 R2 Enterprise x64 Edition的测试环境中，对于最大的地图USA：Bellman-Ford消耗约5.6GB内存，Dijkstra消耗约5.4GB内存，而Crauser和Δ-Stepping都超过了6.5GB，运行失败。所以在本文的测试环境中，算法分配

内存和初始化的时间会更长，并且不断与硬盘交换数据，导致算法性能急速下降。

	Dijkstra	Bellman-Ford	Crauser	Δ-Stepping
NY	**0.1**	6.708	0.749	0.141
BAY	**0.125**	8.315	0.936	0.172
COL	**0.156**	15.19	1.451	0.234
FLA	**0.359**	61.30	3.338	0.592
NW	**0.453**	95.33	4.165	0.624
NE	**0.64**	84.63	5.943	0.905
CAL	**0.734**	183.51	7.113	1.17
LKS	**1.076**	338.25	11.06	1.607
E	**1.576**	382.83	16.271	2.496
W	**2.855**	680.86	30.951	4.758
CTR	**410.14**	4905.44	4304.97	991.56
USA	**1754.54**	8762.85	4530.68	2350.99

表 4: Boost Algorithms Compute Time(secs)，计算源点V_1到其余节点的时间。算法均开启-O2优化。

CUDA Dijkstra(CuD)、CUDA Bellman-Ford(CuBF)、CUDA Δ-Stepping(CuDS)、CUDA CSR Bellman-Ford(CuCBF)和CUDA ELL Bellman-Ford(CuEBF)的运行时间如表5所示，其中CuCBF$_{scalar}$最快，同时CuDS性能也不错，如果提供更大的显存，对于大地图的处理会更好。可以看到CuD和CuBF 相对于CuDS、CuCBF、CuEBF的时间普遍慢很多。原因是测试数据E/V实在太小，每张地图的O_{max}最大也不超过10。每次，CuD和CuBF平均松弛2.5个节点，并行算法的优势被大大削弱。相对地，CuDS、CuCBF和CuEBF每次松弛更多的节点并行度更高，算法也更快。对于E/V较大的地图，CuD和CuBF的并行优势才能体现。

表6给出了CPU算法Dijkstra和Δ-Stepping与GPU算法CuDS和CuCBF$_{scalar}$的时间对比。对于小规模数据，CPU算法比较占优势，而对于千万级以上的数据，GPU并行加速就很明显了。CTR地图：CuCBF$_{scalar}$加速比$K_{Dijkstra} = \frac{T_{Dijkstra}}{T_{CuCBF_{scalar}}} = \frac{410.14}{372.57} = 1.1$，$K_{\Delta-Stepping} = 2.66$。CuDS加速比$K_{Dijkstra} = 1.02$，$K_{\Delta-Stepping} = 2.47$。USA地图：CuCBF$_{scalar}$ 加速比$K_{Dijkstra} = 2.68$，$K_{\Delta-Stepping} = 3.59$。同时，GPU版本的算法的内存消耗相对于CPU版本要少得多，对于USA地图CuDS估计消耗1.4G显存，而CuCBF$_{scalar}$只需要902MB，对于数据规模越大的数据，并行算法的优势越明显。最后，由于测试环境的显卡配置和比赛的显卡还有一定差距，所以在更高配置的显卡上应该能得到更高的加速比。

	CuD	CuBF	CuDS	CuCBF$_{scalar}$	CuCBF$_{vector}$	CuEBF
NY	125.10	596.81	4.631	**1.130**	7.405	1.220
BAY	154.18	840.64	3.830	**1.141**	7.378	1.183
COL	219.34	1648.55	11.04	**2.524**	17.31	2.568
FLA	670.71	27264.5	22.46	**9.131**	82.14	9.686
NW	739.67	时间过长	30.35	**10.50**	92.23	11.82
NE	1022.13	时间过长	16.59	**9.198**	77.66	9.719
CAL	1350.8	时间过长	38.41	**20.05**	171.28	20.37
LKS	2353.8	时间过长	72.77	**44.77**	363.51	45.18
E	3549.42	时间过长	58.50	**44.24**	351.58	46.71
W	8726.91	时间过长	140.55	**66.63**	547.63	72.22
CTR	时间过长	时间过长	400.75	**372.57**	1722.04	显存不足
USA	时间过长	时间过长	显存不足	**654.35**	4864.98	显存不足

表 5: CUDA Algorithms Compute Time(secs)，计算源点V_1到其余节点的时间。算法均开启-O2优化。

	Dijkstra	Δ-Stepping	CuDS	CuCBF$_{scalar}$
NY	**0.1**	0.141	4.631	1.130
BAY	**0.125**	0.172	3.830	1.141
COL	**0.156**	0.234	11.04	2.524
FLA	**0.359**	0.592	22.46	9.131
NW	**0.453**	0.624	30.35	10.50
NE	**0.64**	0.905	16.59	9.198
CAL	**0.734**	1.17	38.41	20.05
LKS	**1.076**	1.607	72.77	44.77
E	**1.576**	2.496	58.50	44.24
W	**2.855**	4.758	140.55	66.63
CTR	410.14	991.56	400.75	**372.57**
USA	1754.54	2350.99	显存不足	**654.35**

表 6: Boost V.S. CUDA Algorithms Compute Time(secs)

8 总结

本文提出了4种不同的CUDA并行最短路径算法，分别基于Dijkstra、Bellman-Ford、Δ-Stepping、Sparse Matrix-Vector Bellman-Ford。其中Δ-Stepping和Sparse Matrix-Vector Bellman-Ford都取得了不错的效果，且相对于Boost库中的串行和并行最短路径算法在时间和空间上都具有较强的竞争力。同时由于实验数据采用的图为稀疏图，不能很好的体现CUDA Dijkstra和CUDA Bellman-Ford算法的并行优势，这两个算法对于稠密图应该会有更好的加速比。

致谢

感谢NVIDIA公司提供的NVIDIA 2011 CUDA校园程序设计大赛平台，以及顾洁同学提供的一些插图。

参考文献

[1] *Wiki:Dijkstra.* http://en.wikipedia.org/wiki/Dijkstra\%27s_algorithm

[2] *Boost:Dijkstra.* http://www.boost.org/doc/libs/1_47_0/libs/graph/doc/dijkstra_shortest_paths.html

[3] *Boost:Parallel Shortest Path.* http://osl.iu.edu/research/pbgl/documentation/dijkstra_shortest_paths.html

[4] *Boost Bellman-Ford.* http://www.boost.org/doc/libs/1_47_0/libs/graph/doc/bellman_ford_shortest.html

[5] *Wiki:Bellman Ford.* http://en.wikipedia.org/wiki/Bellman-Ford_algorithm

[6] *Wiki:Astar.* http://en.wikipedia.org/wiki/A-star_algorithm

[7] *Amit:Astar.* http://theory.stanford.edu/~amitp/GameProgramming/

[8] *GPU Δ-Stepping Dijkstra.* http://code.google.com/p/gpuwire/downloads/detail?name=gpuwire.zip&can=2&q=

[9] *9th DIMACS Implementation Challenge - Shortest Paths.* http://www.dis.uniroma1.it/~challenge9/download.shtml

[10] *Optimizing Parallel Reduction in CUDA* http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[11] *Thrust* http://code.google.com/p/thrust/
prefix sum

[12] *Prefix Sum* http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf

[13] Jason Sanders, Edward Kandrot. *An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010. ISBN 9780131387683.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.

[15] Russell, S. J.; Norvig, P. (2003). *Artificial Intelligence: A Modern Approach.* Upper Saddle River, N.J.: Prentice Hall. pp. 97 – 104. ISBN 0-13-790395-2

[16] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011. Version 4.0.

[17] Ulrich Meyer, Peter Sanders. Δ-Stepping: A Parallel Single Source Shortest Path Algorithm. *In Proceedings of ESA'1998.* pp.393-404

[18] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Commun. ACM*, 29:996 – 1000, October 1986.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269 – 271, 1959.10.1007/BF01386390.

[20] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111 – 129, 1986. 10.1007/BF01840439.

[21] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596 – 615, July 1987.

[22] M. Garland. Sparse matrix computations on manycore gpu' s. *In Proceedings of the 45th annual Design Automation Conference*, DAC ' 08, pages 2 – 6, New York, NY, USA, 2008. ACM.

[23] Bellman, Richard (1958), "On a routing problem", *Quarterly of Applied Mathematics* 16: 87 – 90

[24] A.V.Goldberg, T.Radzik. A Heuristic Improvement of the Bellman-Ford Algorithm. *Applied Math. Let.* 6:3-6,1993.

[25] Yen, Jin Y. (1970), "An algorithm for finding shortest routes from all source nodes to a given destination in general networks", *Quarterly of Applied Mathematics* 27: 526 – 530

[26] D. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *Journal of Optimization Theory and Applications*, 88:297 – 320, 1996. 10.1007/BF02192173.

[27] D. L. Baggio. Gpu based image segmentation livewire algorithm implementation. Master' s thesis, *Technological Institute of Aeronautics*, Sao Jose dos Campos, 2007.

[28] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, Single-Source Shortest Paths with the Parallel Boost Graph Library. *9th DIMACS Implementation Challenge: The Shortest Path Problem*, November 2006.

[29] K. Madduri, D. Bader, J. Berry, J. Crobak. Parallel shortest path algorithms for solving large-scale instances. *9th DIMACS Implementation Challenge: The Shortest Path Problem*, November 2006.

[30] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, Peter Sanders. A Parallelization of Dijkstra's Shortest Path Algorithm, *In Mathematical Foundations of Computer Science*, volume 1450 of Lecture Notes in Computer Science, 722-731, 1998. Springer.

[31] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, Peter Sanders. Parallelizing Dijkstra's shortest path algorithm. Technical report, *MPI-Informatik*, 1998.

[32] B. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: *Theory and experimental evaluation. Mathematical Programming*, 73:129 – 174, 1993.

[33] M. Garland. Sparse matrix computations on manycore gpu' s. *In Proceedings of the 45th annual Design Automation Conference*, DAC 08, pages 2 – 6, New York, NY, USA, 2008. ACM.

[34] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors.*In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC ' 09, pages 18:1 – 18:11, New York, NY, USA, 2009. ACM.