计数算法主要有两种出发点，一种是从节点出发，一种是从边出发。forward[1]，compact-forward[2]属于以边为出发点的算法。Tsourakakis 提出了 eigntriangle[3]以矩阵乘法为基础的算法，邻接矩阵相乘三次，对角线求和再除以 6 得出三角形个数。部分算法精确统计三角形个数，有的则是求得近似值[4]。有的算法输入随着处理过程不断变化，以不断到达的数据流[5][6]为输入，有的算法以文件作为输入。Suri 在集群上使用 MapReduce 解决三角形计数问题[7]。这些算法的时间复杂度已经被详细的研究过，基于快速矩阵乘法的算法能够获得$O(n^{2.376})$[8]的复杂度，但空间复杂度往往使得某些算法无法在大规模的无向图上应用。存储 3M 个节点的矩阵需要数千 G 的内存，这在如今常见的设备上是难以获得的，这使得以邻接表或者 edge list 表示图形的算法更为实用。Forward 是 Schank 提出的一种$O(m^{\frac{3}{2}})$时间复杂度的算法，空间复杂度$O(m)$，Latapy 在它的基础上提出了 Compact-forward 算法，进一步降低了空间复杂度。

CUDA 是 Nvidia 推出的一种并行计算平台和编程模型[9]。CUDA 提供一种通用计算的抽象模型，与图形处理任务无关。主机发起的 Kernel 函数在 GPU 上被大量的 thread 并发执行。Thread 组织成 Block，每个 Block 在有空闲资源的时候被 GPU 上的硬件调度到多核流处理器上执行。CUDA 以一种叫做单指令多线程的方式执行 GPU 上的大量线程。每 32 个编号相邻的线程组成一个 warp，共享一个指令发射器，但各自拥有自己的寄存器，可执行自己的分支。如果某一时刻出现条件分支，同一个 warp 里的线程必须依次串行地执行所有分支的指令，不处于自身条件分支的线程不激活，需等待同 warp 其它线程执行结束，导致执行速度的延迟。SIMT 提供了线程级并行和数据并行两种抽象。

设计和实现：

串行算法 compact-forward：

Latapy 在 forward 的基础上提出的 compact-forward 算法如下：

Algorithm: compact-forward:

Input: G 的邻接表表示

1. *用一个单射函数 η() 为所有的顶点标号*
   使得对于所有的 u 和 v，若 d(u) > d(v)则有 η(u) < η(v)
2. *根据 η()对压缩的图表示排序*
3. *按照 η()升序的方式遍历每个顶点 v:*

3a. 对所有满足 η(u) > η(v)的 v 的邻居节点 u：

3aa. 设 u'是 u 的第一个邻居节点， v'是 v 的第一个邻居节点

3ab. 只要还存在 u, v 尚未被访问过的邻居节点，且满足 η (u') < η(v) and η(v') < η(v):

3aba. 若 η (u') < η (v') 那么 u' = u 的下一个邻居节点

3abb. 否则若 η (u') > η (v') 那么设置 v'为 v 的下一个邻居节点

3abc. 否则:

3abca. 三角形计数加一

3abcb. u' = u 的下一个邻居节点

3abcc. v' = v 的下一个邻居节点

该算法的时间复杂度是$O(m^{\frac{3}{2}})$，空间复杂度是$O(n+m)$，其中 n 表示顶点个数，m 表示边的个数。需要一个顶点列表，空间复杂度$O(n)$，邻接表中每条边存储两次$O(m)$，存储 η 的话也需要复杂度为 n 的空间。

分析和改进：

如果使用包含 3 个顶点的序列表示无向图中的一个三角形，那么每个三角形有 6 中表示方法，而实际上每个三角形只需计数一次就够了。如果不对顶点序列的表示方法加以限制，会产生大量冗余结果。常用的方法是给每个顶点一个唯一的编号，按照升序或者降序方式使用一个唯一的序列表示三角形，相当于按照顶点序号的大小关系把无向图转化成了有向图。这种方式能唯一的表示结果，但是不能够完全解决计数过程中重复试探的问题。以升序排列为例，{2，6，9}这个三角形的每条边可以用从小到大的顶点编号表示，三条边分别是 2->6，2->9，6->9。Edge-iterator 这类以边为出发点的算法选取一条边，如果任选一条边比如 2->9，再从 2 和 9 的邻接节点里找到公共顶点 6，可得到这个三角形。如果在公共顶点集合中编号小于 2 或者大于 9 的定点，还是会造成重复计数。Forward 和 compact-forward 算法为度数最高的顶点分配最小的编号，度数第二高的定点分配次小的编号，以此类推。下图中 min,mid,max 表示顶点编号，对于这种情况，先选择两个顶点 mid 和 max，构成边 I，然后从邻接到 mid 和 max 的交集里选择定点 min，即边 II 和 III。

这两种算法需要记录从某个顶点出发的边，也需要记录到达该顶点的边，每条边出现两次。从简化记录信息的角度来看，每条边只记录一次也能够达到相同的效果。首先选择一个顶点，从邻接到该定点的列表里选择一个点，对邻接到这两个点的列表求交集即可。以下图为例，先选择一个点，作为图中的 max，然后从 max 的邻接表里选择一个点 mid，最后从二者的邻接表里得到 min。这样每条边只记录依次，减少了三角形计数算法的空间复杂度。依照这个思路，改进 compact-forward，得到 reverse-edge forward。

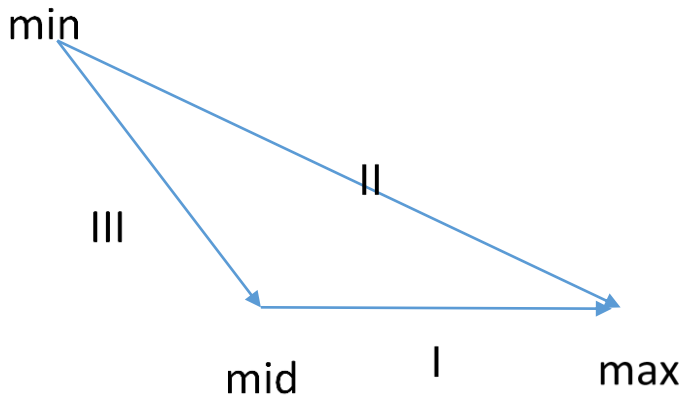

Figure 1

给度数最高的定点最低的数值编号

数值编号满足

min<=mid<=max

节点的度满足

degree(max) <= degree(mid)<=degree(min)

实现：

输入数据预处理：

SNAP 的数据文件以类似的文本格式存储。最开始的几行是注释，包括文件名，节点数目，边的数目等信息，之后每行记录一条边的信息。SNAP 文件采用边列表的方式表示无向图，即记录每条边两个节点的编号。但节点编号不都是从 0 开始的，最大的节点编号可能超过节点数目。通过预处理过程给节点重新编号，用编号数值大小表示先后关系，省去存储 η 的空间和计数过程中映射编号的时间。

预处理过程：

1．先从注释中读取顶点数，边数，再从文件读入每条边

2．遍历边列表，统计每个节点的度数。

3．按顶点度数排序，以度数从高到低的顺序，从 0 开始递增地给予每个顶点新的编号

4．再次遍历边列表，将旧的编号映射为新编号

5．对每条边用较小的节点编号作为索引，将较大的编号插入对应的邻接表里

6．对每个定点的邻接表按从小到大的顺序排序

得到的邻接表是一个 vector<vector<int> > 类型，以节点编号为第一层索引，可得到邻接到该节点的向量，且向量内元素已按照升序排列。

串行算法：

```
int reverse_edge_forward(AdjList& adjl)
{
    int ret = 0;
    for (size_t i = 0; i < adjl.adj_list.size(); i++){
        for (auto& j : adjl.adj_list[i]){
            vector<int>& node1 = adjl.adj_list[i];
            vector<int>& node2 = adjl.adj_list[j];
            ret += intersect(node1, node2);
        }
    }
    return ret;
}
```

在已排序序列上常用的求交集的方法有两种，一种是并归的方式，一种是二分查找。并归的方法从两个序列各取一个元素，比较大小。如果相等则增加三角形计数，两个序列的指针各向后移动一个元素。否则较小元素的指针后移，另一个指针不变。这种方式的时间复杂度是 O(M + N), M，N 分别是两个序列的长度。二分查找方法是依次遍历较短的序列中的每个元素，在较长的序列中搜索该元素。时间复杂度是O(mlogn)，其中 m 是较短序列的长度，n 是较长序列的长度，这种方法在两个序列长度差别较大时效果更好。由于二分查找方法需要先找出长度较短的数组，再循环进行查找。判断的过程会产生一个分支，再加上后续的二分查找循环，在 GPU 上引起的 branch divergence 更多，实际执行效果并不好，为方便和几个串行版本的实现进行比较，文中所有实现都采取并归的方法。

GPU 上的实现:

数据结构：共 3 个数组。串行版本的邻接表从小到大拼接成一个 head 数组，里面的值是条边的头节点。一个索引数组 index，以节点编号为索引，得到邻接到该节点的，编号最小的节点在 head 中的偏移，Index[x + 1] – index[x]可得邻接到 x 的节点个数。最后为了方便每个线程找到自己分配到的边的尾节点，分配一个 tail 数组。用线程编号索引 head 可得头节点，索引 tail 可得尾节点。这三个数组在预处理阶段结束时传到设备的全局存储器上去。

算法：与串行版本类似，只是不再需要两层 for 循环。图中每条边分给一个线程，每个线程的求头尾节点共同的邻接点个数，根据自身的索引，将结果写入 result 数组对应位置。

kernel 函数的 Block 和 thread 参数:Gpu 上的每个多核流处理器最多可同时驻留 2048 个线程，16 个 block，每个 block 最多 1024 个线程，同时最多驻留 64 个 warp。本文未使用片上共享内存，每个线程中使用的寄存器数目较少，不会限制处理器上驻留的线程数目。为达到处理器线程数上限，Thread 可选 64，128，256，512，本文选取 256。

最后对 result 数组进行 reduce 操作，得到总的三角形个数。

文中提到的 5 种算法的比较：

Serial CUDA 是 GPU 上运行程序的串行版本，使用同样的数据结构，只不过用串行方式执行 Kernel 函数的工作。

	Forward	Compact	Reverse-edge	Serial CUDA	CUDA
邻接表实现	Vector	Vector	Vector	Vector	数组
预处理阶段对每个节点的邻接节点列表排序	否	是	是	是	是
每条边存储次数	1	2	1	1	1
运行时内存需求	O(2n + 2m)	O(n + 2m)	O(n + m)	O(n +2 m)	O(n +3m)

Forward 需要一个邻接表存储图信息，一个邻接表存中间结果。

Compact-forward 需要一个列表表示顶点，一个列表表示和顶点相邻接的边，每条边出现两次。

Reverse-edge 需要一个列表表示顶点，一个列表表示邻接到某个点的顶点集合。

Serial CUDA 比 reverse-edge 多了一个尾节点表

CUDA 还需要一个 result 数组存储线程的计算结果。

实验：

实验环境：

硬件：

Intel i7 4710QM，8G 内存。Nvidia GTX 870m 显卡（7 个 SMX，3G 显存）。
软件：
Win8.1 操作系统，CUDA Toolkit 7 ，显卡驱动版本 347.62，Visual studio community 2013。

实验方法：

代码在 visual studio 中以 win32，release 方式编译，在规模不同的数据集上依次执行各个算法。对在主机上执行的代码，采用 GetProcessTimes 统计用户时间和内核时间，由于 GPU 上的 kernel 函数，异步内存拷贝和主机代码是异步执行的，所以使用 CUDA 提供的 event 统计执行时间。在整机处于一半负载的情况下进行实验，无 CPU 密集或 IO 密集型应用，无制图或播放视频等大量使用 GPU 的应用。得到包括从读文件到输出结果的墙上时间，完成三角形计数功能的执行时间两个结果。

实验数据：

数据来自 Stanford Network Analysis Project(https://snap.stanford.edu)

Dataset	Nodes	Edges	Triangles	File size
Amazon	334863	925872	667129	12,291KB
As-skitter	1696415	11095298	28769868	145,612KB
LiveJournal	3997962	34681189	177820130	489,799KB

实验结果：

串行，并行算法的执行时间对比

从读取数据集文件到输出结果的总时间（kernel/user space）：

Dataset	Forward	Compact	Reverse-edge	Serial CUDA	CUDA
Amazon	0.0s/0.765s	0s/0.875s	0.032s/0.719s	0.031s/0.734s	0.078s/0.688s
As-skitter	0.14s/7.89s	0.188s/9.344s	0.172s/7.891s	0.25s/7.89s	0.454s/6.703s
LiveJournal	1.609s/31.984s	0.672s/36.969s	0.672s/32.967s	0.687s/32.063s	1.485s/22.89s

先来看串行算法。计时过程包括预处理时间，各个串行算法在每个数据集上的差别不明显。Compact-forward 相比 forward 算法在预处理阶段对每个顶点向邻接表插入两次，并且对邻接表进行了排序，耗时较长。Reverse-edge 算法与 forward 类似，除了按相反的次序将节点插入邻接表之外，还对每个顶点的邻接点列表进行排序，而 reverse-edge 与 forward 的执行时间类似，可推断 compact-forward 执行缓慢主要是两次插入节点的导致的，多出来的时间主要消耗在内存分配上。Serial CUDA 的执行时间与 forward， reverse-edge 相差无几，说明 serial CUDA 使用的尾节点列表没有对整体性能产生显著影响。

再看并行算法。CUDA 代表在 GPU 上执行的算法，因为向 GPU 内存拷贝数据，发出 kernel 函数调用需要经过内核，CUDA 在内核态花费的时间长于其余 4 个串行算法。从整体执行时间上看，CUDA 明显优于其余 4 个串行算法。

三角形计数时间（kernel/user space）：

Dataset	Forward	Compact	Reverse-edge	Serial CUDA	CUDA
Amazon	0s/0.235s	0s/0.079s	0s/0.047s	0s/0.031s	0.016s
As-skitter	0s/3.594s	0s/2.328s	0s/1.641s	0s/1.36s	0.156s
LiveJournal	0.094s/15.407s	0s/11.69s	0s/10.204s	0s/8.813s	1.135s

Amazon 数据集上的 forward 算法在用户态的时间明显长于其余算法。Forward 在执行过程中会动态的构建一个邻接表，amazon 又是三个数据集中数据量最小的，因此推断在数据量小的情况下，内存分配时间显著的影响了算法的执行时间。同样由于动态分配内存的原因，forward 算法在三个数据集上的执行时间都是最长的。Reverse-edge 相比 compact-forward 将求交集的尝试次数减少了近一半，因此执行的更快。我认为 Serial CUDA 把所有的邻接表拼接成了一块连续的内存，在一定程度提高了缓存命中率，因此执行时间比 reverse-edge 更短。由于 CUDA 的 kernel 函数异步于主机代码运行，使用 CUDA 提供的 Event 计时，获取算法在 GPU 上执行的时间。可见 GPU 上的计数算法执行速度显著高于串行算法。

GPU 上的计数算法对串行算法的加速比：

Dataset	Forward	Compact	Reverse-edge	Serial CUDA
Amazon	14.69	4.94	2.93	1.93
As-skitter	23.04	14.92	10.51	8.70
LiveJournal	13.66	10.30	8.99	7.76

对于任一数据集，GPU 计数算法对 forward 有最高的加速比，compact-forward，reverse-edge, serial CUDA 这三个算法的执行速度是依次提升的，GPU 算法对于他们的加速比依次降低，对 amazon 这一小数据集，CUDA 相对于其它串行算法的加速比不是很明显。随着数据集的增加，GPU 版本能够获得 7 倍以上的加速比。

本文实验采用的 GPU 有 7 个 SMX(多核流处理器)，每个流处理器可在每个时钟周期同时选出 4 个指令流，在每个时钟周期内，每个指令流内部不相关的两条指令可同时执行。只获的了 7 倍的加速比实际上并未发挥 GPU 的最大并行计算能力。Kernel 函数里求交集部分访存基本上无法合并，branch divergence 众多，这应该是造成这一现象的主要原因。

Multikernel 的效果

ThreadDim.x = 256， blockDim.x = (边数 – 1) / 256 + 1

multikernel 建立 16 个 stream，每个 stream 分配一个 kernel

normal 是前面实验用到的版本

Dataset	multikernel	normal
Amazon	0.016s	0.016s
As-skitter	0.153s	0.156s
LiveJournal	1.132s	1.135s

计算能力 3.0 的设备[10]可以同时执行 16 个 kernel，尝试减少单个 kernel 的工作量，提高同时执行的 kernel 数量来发掘并行性，提高处理速度。Amazon 数据集上未见速度提高，在后两个数据集上，缩短时间在 1%左右。执行情况时间从 visual profiler 的结果看，尽管依

次在 16 个 stream 里分配了一个 kernel，这些 kernel 还是串行执行的。考虑到 block 数量可能对重叠区域大小有影响，后俩个数据集上 block 数又远超 1000。人为设定 block 数在 1000 以下，如 900，每个 block256 个线程的配置下，各个 kernel 之间的执行也没有完全重叠，只是上一个 kernel 执行的最后时刻与下一个 kernel 有短暂的重叠。这是因为设备无法提供能满足并行执行多个 kernel 需要的资源，无法理想地重叠 kernel 执行，多 kernel 方法在本文的实验中对整体执行时间的影响在百分之一左右。

---

1 T. Schank. Algorithmic Aspects of Triangle-Based Network Analysis. PhD thesis, Computer Science, University Karlsruhe, (2007)

2 M. Latapy, Theory and Practice of Triangle Problems in Very Large (Sparse (Power-Law)) Graphs(2006)

3 C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. ICDM, 0:608–617, 2008

4 M.Rahman, M.A.Hasan, Approximate triangle counting algorithms on Multi-cores: 2013 IEEE International Conference on Big Data

5 Z.Bar-Yossef,R.Kumar,D.Sivakumar, Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs SODA '02 Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms

6 G.Cormode,H.Jowhari:A Second Look at Counting Triangles in Graph Streams Theoretical Computer Science 2 October 2014, Pages 44–51

7 S.Suri ans S.Vassilvitskii,Counting triangles and the curse of the last reducer: Proceedings of the 20th international conference on World wide web,WWW'11,(New York, NY, USA),pp 607-614,ACM,2011

8 D.Coppersmith, S.Winograd,Matrix multiplication via arithmetic progressionsJournal of Symbolic Computation, 9(3):251-280, 1990

9 http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

10 http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities