|  |  |
| --- | --- |
| 学号 | 2017282110200 |
| 密级 |  |



**海量数据存储论文阅读笔记**

|  |
| --- |
| **FlashGraph：在商用级SSD阵列上**  **处理数十亿节点的图结构** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 计算机技术 |
| 学 生 姓 名 ： | 李辰宸 |
| 指 导 教 师 ： | 何水兵 副教授 |

二〇一七年十二月

摘 要

这是一篇的FAST 2015论文，原论文题为FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs。

线性结构、树、图，是三种最基本的数据结构。通常来说，对图的操作涉及许多随机读取和写入，而这些工作通常都在内存中执行，但是要处理一个规模巨大的图，往往需要一个计算机集群的内存容量。在本文中，原作者提出了FlashGraph引擎。FlashGraph基于SSD阵列在单台多核计算机上就能够存储数十亿节点、数千亿边的超大规模图；在性能方面，FlashGraph的SSD版本实现了80%于自身纯内存版本性能，同时，SSD版本的性能还明显优于PowerGraph——一个流行的基于纯内存的分布式图处理引擎。

本文提出的 FlashGraph 主要做三个事情，一个是利用其构造的数据尽可能紧凑来减少IO，二是尽可能采用顺序IO，三是将IO和计算并行化。为了做到这三个事情基于Linux VFS之上做了一层用户空间文件系统，主要是为了避免内存拷贝和理解IO。因为向SSD发出IO都需要进行用户态和内核态的拷贝，这样在用户态实现一个简单的页面缓存可以大大减小上下文切换带来的内存拷贝。同时因为缓存可以被应用感知，可以根据应用(Task)的上下文来提高缓存命中率。另外，FlashGraph提供的编程模型使得在每个执行体可以下方到用户空间文件系统层执行，完全利用用户构建的页面缓存来进行访问，避免用户空间的二次拷贝。

关键词：图；大容量存储；固态硬盘；IO优化；并行处理；作业调度

**ABSTRACT**

Graph analysis performs many random reads and writes, thus, these workloads are typically performed in memory. Traditionally, analyzing large graphs requires a cluster of machines so the aggregate memory exceeds the graph size. We demonstrate that a multicore server can process graphs with billions of vertices and hundreds of billions of edges, utilizing commodity SSDs with minimal performance loss. We do so by implementing a graph-processing engine on top of a user-space SSD file system designed for high IOPS and extreme parallelism.

Our semi-external memory graph engine called FlashGraph stores vertex state in memory and edge lists on SSDs. It hides latency by overlapping computation with I/O. To save I/O bandwidth, FlashGraph only accesses edge lists requested by applications from SSDs; to increase I/O throughput and reduce CPU overhead for I/O, it conservatively merges I/O requests. These designs maximize performance for applications with different I/O characteristics. FlashGraph exposes a general and flexible vertex-centric programming interface that can express a wide variety of graph algorithms and their optimizations. We demonstrate that FlashGraph in semi-external memory performs many algorithms with performance up to 80% of its in-memory implementation and significantly outperforms PowerGraph, a popular distributed in-memory graph engine.

**Key Words**: Graph; Mass storage; SSD; IO optimizing; Parallel processing; Task scheduling

目 录

[1 引言 1](#_Toc500672474)

[2 总体设计 3](#_Toc500672475)

[2.1 总体架构 3](#_Toc500672476)

[2.2 执行模型 4](#_Toc500672477)

[2.3 数据表示 5](#_Toc500672478)

[2.3.1 内存中的节点数据 5](#_Toc500672479)

[2.3.2 外存中的边表数据 5](#_Toc500672480)

[2.4 其它 6](#_Toc500672481)

[2.4.1 部分边获取 6](#_Toc500672482)

[2.4.2 边表请求合并 6](#_Toc500672483)

[2.4.3 节点程序调度 6](#_Toc500672484)

[2.4.4 分区的划分 7](#_Toc500672485)

[3 性能 8](#_Toc500672486)

[3.1 与自身的纯内存版本比较 8](#_Toc500672487)

[3.2 其它内存图引擎比较 8](#_Toc500672488)

[3.3 其它外存图引擎比较 9](#_Toc500672489)

[4 我的思考 11](#_Toc500672490)

[4.1 FlashGraph线程模型可能存在的问题和改进思路 11](#_Toc500672491)

[4.2 基于异构计算的节点程序执行模型的可行性 11](#_Toc500672492)

# 引言

大规模图的分析已经成为学术界和工业界一个重要的计算问题，例如，Facebook中的社交图有数十亿的节点，而今天整个互联网、未来整个物联网的连接图的规模还会更加令人瞠目结舌。**可见，这篇论文讨论的问题贴切的迎合了海量数据存储的热门话题——在数据爆炸增长的今天，如何利用已有的存储介质，设计高可靠、高容量、高性能的存储系统？**

正如我们在数据结构课程中了解的，图的一个简单的边遍历就包含了大量随机、不可预测的IO，由此可见，图其实不太适合利用分治的思想进行优化。尽管如此，对于上述规模的图的处理仍然只有通过划分给多台计算机共同完成，主要原因仍然是一台计算机难以胜任对整个图结构的存储。传统的图处理引擎通常包含了一下三个方面的要素：将图分割后存储在不同的计算节点的内存；算法的并行化；通过分布式共享内存或者消息机制进行通信。使用内存存储图相对于磁盘而言的确减少了访问延迟，但对不同计算机间、图的不同分块间的互相访问，又对网络性能提出了相当高的要求，而且价格不菲。

因此，近期的研究又将重心转移到了面向磁盘的图存储上，针对磁盘存储的特性做出了很多改进，例如来消除随机IO、尽可能地实现顺序访问，但是相对于纯内存系统之间性能差异依然十分明显。

既然内存速度快、容量小，外存容量大、速度次之，那么结合两者优势来实现图的存储呢？**在海量数据存储课程上，我想起了何老师为我们展示了计算机存储系统速度和容量的分层模型，如表1-1所示：**

表1.1 计算机存储设备的分层模型

|  |  |  |
| --- | --- | --- |
| 存储设备 | 速度 | 容量 |
| CPU缓存 | 很快 | 很小 |
| 内存 | 快 | 小 |
| 外存 | 慢 | 大 |

**可见，速度快的存储器的成本往往高昂、容量很小，而容量大的存储器往往速度缓慢。在设计一个存储系统时，只有充分利用到每一层存储设备的特性，才能在实现可观容量的同时提供出色的性能！这也是不少存储系统设计的核心灵魂。**

同样基于这个思路，原作者便提出了FlashGraph——使用内存存储节点、使用SSD阵列存储边的方法。内存存储节点提高了图的可扩展性，并且相对于边而言，存储节点所需要的容量也相对较小；而外部基于SSD存储的边本身也能提供客观的IO吞吐量。

原作者的FlashGraph基于SAFS文件系统构建，并以此为基础完成了一系列的技术挑战，使FlashGraph能够提供与纯内存系统相媲美的性能，包括重构IO调度、数据存放和缓存等。FlashGraph在半外存模式下能够达到和纯内存模式下相媲美的性能，并且还明显超过了PowerGraph。进一步，原作者还展示了在FlashGraph在一个34亿节点、1290亿边的图上执行广度优先搜索，整个过程只消耗了22GB的内存。

总而言之，FlashGraph在提供可观性能的同时对内存容量的依赖较小，也是一个非常具有应用价值的研究成果。

# 总体设计

正如前文所描述的，FlashGraph是一个半外存、高性能的图存储引擎，它在内存中存储和维护节点信息，在SSD阵列中存放边的信息。FlashGraph基于SAFS文件系统建立。SAFS是一个针对SSD阵列的用户空间的文件系统，可供于用于实现高吞吐量和轻量级的缓存。在性能优化方面，原作者遵循了以下的设计原则：

* 缩减IO总数。缩减IO总数是最直接的性能优化方案，FlashGraph做了以下方面的工作：压缩数据结构；提高缓存命中率；有选择地进行边的遍历。
* 尽可能使用顺序IO。尽管SSD的随机读写性能相对于磁盘十分优越，但不论如何顺序读取都会达到更高的数据带宽，并减少CPU占用率和系统内核处理。
* IO、计算并发执行。提高IO的并发度的同时异步执行计算。
* 最小化SSD的寿命损耗。每块SSD都有一定的写入次数上限，为了提高SSD的使用寿命，FlashGraph会尽量避免频繁的写入操作。

**——在小组论文研读和展示中，可以看到很多存储系统的设计都遵循了上述的一些原则。例如，我们小组展示的论文是关于近年推出的瓦片堆叠磁盘SMR的。SMR磁盘在进一步提高磁道密度的同时，其“相邻磁道覆盖写”的特性使其随机写性能大打折扣。因此，目前市面上不少商品级SMR磁盘为了实现和传统磁盘相同的接口，会保留盘面中的一块区域作为写入操作的临时缓存，从而随机写入，在一定程度上缓解SMR特性对性能的影响。**

## 总体架构

为了更好的支持FlashGraph，原作者在SAFS外封装了一层异步用户任务模式的接口， FlashGraph通过这个异步接口访问文件系统来减少性能开销。此外，在FlashGraph中（包括其它成功的图引擎也是如此），一个图操作会被分解为许多的节点程序，并以节点程序的形式被送入引擎中执行。当节点程序需要访问到SSD中数据时，引擎再通过SAFS获取所需数据。这样FlashGraph的总体架构就如图2-1所示。

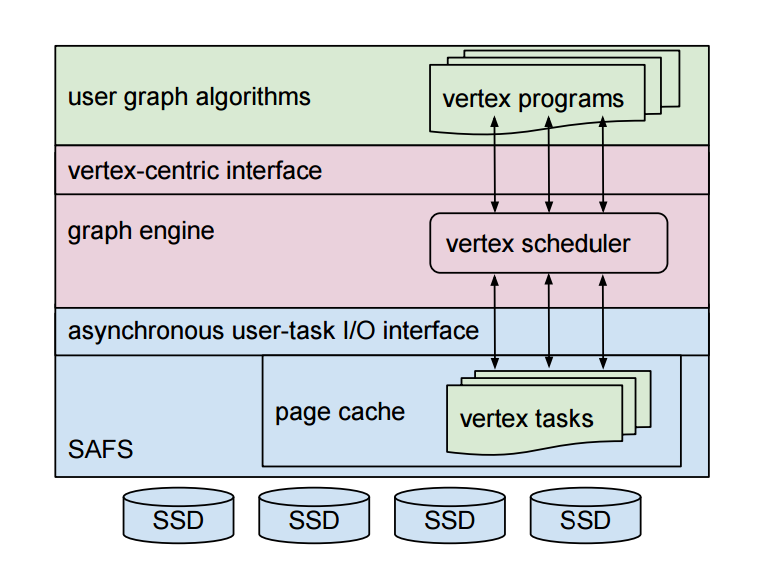


图2.1 FlashGraph总体架构

## 执行模型

对于送入引擎的节点程序，FlashGraph使用迭代的方式执行：对于每一次迭代，处理前一次迭代标记为活跃的节点；当没有活跃节点时，一次图操作执行完毕。在具体执行上，FlashGraph同样将图划分为几个相对独立的部分，并为每个部分分配一个处理节点程序的线程。每个线程都有自己的活跃节点队列，也有着自己的调度器，确保节点程序的顺利执行。

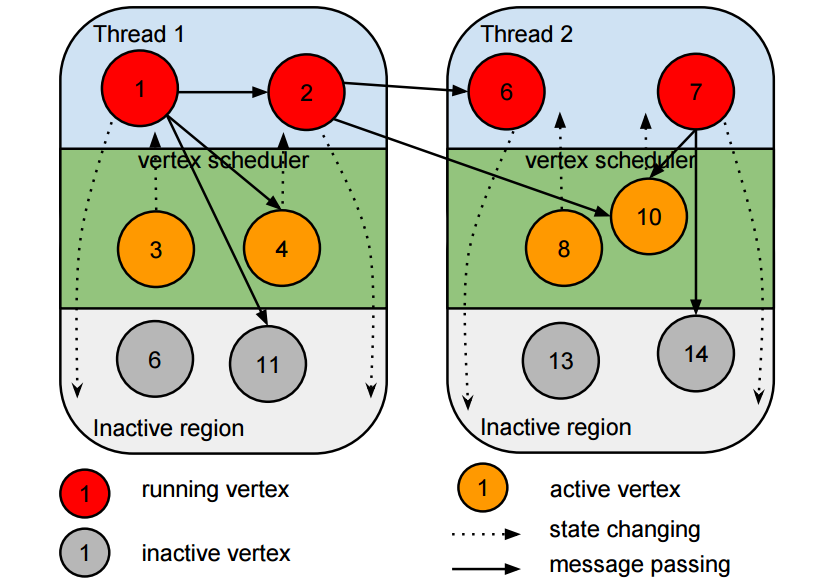


图2.2 FlashGraph的作业调度

图引擎中每个节点程序有三种状态：执行中、活跃、非活跃。一个节点程序可以被图引擎激活，也可以被别的节点程序激活；活跃的节点被调度器分配线程后进入执行中状态；当节点程序执行完成后，则进入非活跃状态。执行中节点与其它节点之间的通讯，则是基于消息传递。如图2.2所示。

## 数据表示

FlashGraph内存、外存进行存储，并在对内存和外存中的数据设计了不同的表示方式。

### 内存中的节点数据

FlashGraph中，对于内存中的每一个节点包括一下数据：

* 用于获取外存中边表位置和长度的节点序号；
* 用户定义的数据；
* 节点状态；
* 每个线程的消息队列。

为了节省存储空间，FlashGraph选择在运行时，通过节点序号计算每个边表的位置和长度，因为要直接储存这些信息，无向图每个节点都需要12个字节而有向图则需要24个，这样很难做到在内存中完全储存节点数据。

FlashGraph的具体做法是：对于度数小于255的节点，无向图使用1个字节、有向图使用2个字节来记录节点的度；对于度数大于等于255的节点，使用哈希表来储存节点的度。得到度以后，FlashGraph便可以通过度来计算边表的长度和位置（外存中的边表是按照节点序号顺序紧密存储的）。

但是，这样做一个很明显的缺点便在于，获取一个节点的边表位置和大小的计算量为O(n)、n为节点个数，对于任何一个图操作而言难以接受。因此，FlashGraph在上述两种方案中做了一个平衡，每32个边表就缓存一下信息。这样获取边表就不需要从第一个节点开始计算，其时间复杂度又降低到了O(1)，所有节点平均下来所需的存储空间也不会太高，对于无向图约为1.25个字节、对于有向图约为2.5个字节。

可见，FlashGraph采用了一种目的明确的时间换空间的设计。

### 外存中的边表数据

FlashGraph中边表的存储形式如图2.3所示。每一个节点的边表包含头、边数据和用户可定义的边属性。值得注意的是，边数据和边属性是分开存储的，这样在只需要数据不需要属性的时候就可以避免同时读取浪费带宽，这个策略在许多数据库系统中也有用到。正如在2.3.1中提到的，所有边表按照节点序号的顺序紧密排列。

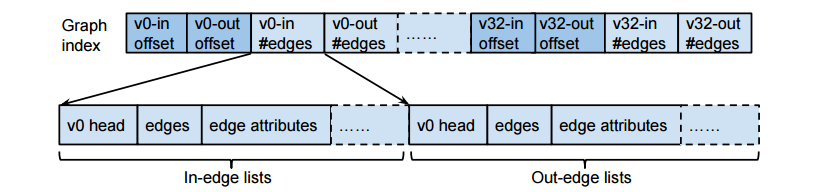


图2.3 FlashGraph边存储

此外，对于有向图，FlashGraph对入边和出边分开存储。这样设计的原因在于许多有向图操作都只需要其中某一种数据，例如广度优先遍历就只需要出边，分开存储可以有选择性地读取所需要的边数据，从而避免带宽浪费。当然，如果一个有向图操作确实需要上述的两种数据，这种设计就会导致IO数量翻倍；不过，FlashGraph通过合并IO有效地抵消了这一点。

## 其它

### 部分边获取

许多基于外存的图引擎，譬如GraphChi和X-Stream，在每次迭代都顺序读取所有的边表，这虽然简单粗暴地降低了随机IO，但是也造成了大量的带宽浪费。

各式各样的图操作虽然会产生大小不一的IO请求，但基本来说每个节点只会访问它自己的边表。FlashGraph基于SSD阵列设计，它本身相对于磁盘系统已经具备一定的随机IO能力，因此，FlashGraph可以有选择性的请求一个图操作涉及的边，从而减少不必要的IO。

### 边表请求合并

在一个图操作的每次迭代中，都会有大量的节点产生对边表的请求，FlashGraph因此认为合并边表的读取请求是可行的。每一个状态为活跃的节点的边表请求都会被预读然后合并（这也是为什么FlashGraph设置3种节点状态的原因）。

### 节点程序调度

FlashGraph基于线程对节点程序进行调度，它将图按线程分为几个相对独立的分区后，每个分区的线程处理自己分区内的节点程序。

**原文认为，这种线程模型的优点是“简单而且稳定”。实际上，我认为原作者有意的忽略的一个事实，就是这种线程模型几乎必然的会导致某些任务在线程上分配的不均匀，因为不能保证一个图操作涉及的节点均匀地分配在整个图上。因此，我在下文中提出了一种改进思路。**

### 分区的划分

FlashGraph的分区方法类似于一个二维矩阵，是将图用2维坐标的形式表示，主要是为了地分配线程。此外，通过分区，作者认为同一个分区内绝大多数节点的边表，在外存上大都能分布在同一块SSD上，这有助于线程对同一块SSD的IO请求的平衡（这点我不太明白为什么，从阵列的角度来看，或许在底层上做IO请求的平衡是更加治本的？）。

# 性能

原作者使用了3种公开的大型图数据来对FlashGraph进行性能测试，如表3.1所示，并和自身的内存版本以及其它图引擎比较。比较的算法包括有广度优先便利BFS、三角形个数TC等，还有一些囿于我个人在这方面的研究有限无法给出详细的介绍。

表3.1 计算机存储设备的分层模型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据源 | 节点 | 边 | 总大小 | 直径 |
| Twitter | 4200万 | 15亿 | 13GB | 23 |
| Subdomain | 8900万 | 20亿 | 18GB | 30 |
| Page | 34亿 | 1290亿 | 1.1TB | 650 |

## 与自身的纯内存版本比较

首先，原作者将SSD版本的FlashGraph（1GB缓存）和纯内存版本的FlashGraph在Twitter和Subdomain两个图上进行比较，得到的结果如图3.1所示。

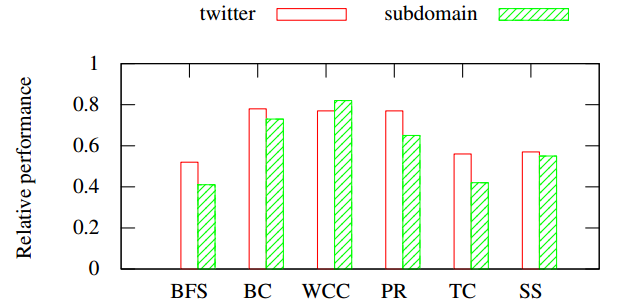


图3.1 FlashGraph边存储

图3.1横坐标为不同的测试算法，纵坐标为SSD版本和纯内存版本的性能比值。可以看到，FlashGraph与自身的纯内存版本比较时，大致实现了后者40%到80%不等的性能，考虑到内存的随机IO性能高于SSD数个数量级，这个结果已经相当优越，说明FlashGraph本身的外存系统设计效果显著。

## 其它内存图引擎比较

原作者将FlashGraph与PowerGraph和Galois两个基于内存的引擎比较，得到的结果如图3.2所示。

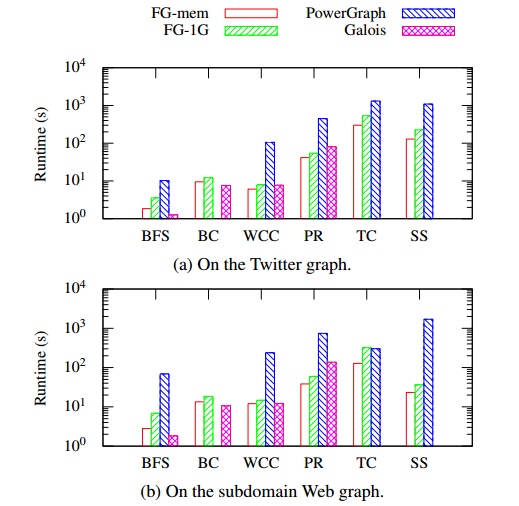


图3.2 FlashGraph边存储

图3.2横坐标为不同的测试算法，纵坐标为执行时间。FlashGraph在测试中分为纯内存实现和SSD两个版本。可以看到，在大部分测试中，FlashGraph的纯内存版本都取得几乎最佳的成绩，而SSD版本虽然稍逊，但依然表现不俗，在所有测试中均取得了由于PowerGraph的表现。

## 其它外存图引擎比较

原作者在Twitter数据源上，在相同的SSD阵列上，将FlashGraph与X-stream、GraphChi这两个基于外存的引擎进行了比较，得到的计算时长、内存占用等结果如图3.3所示。

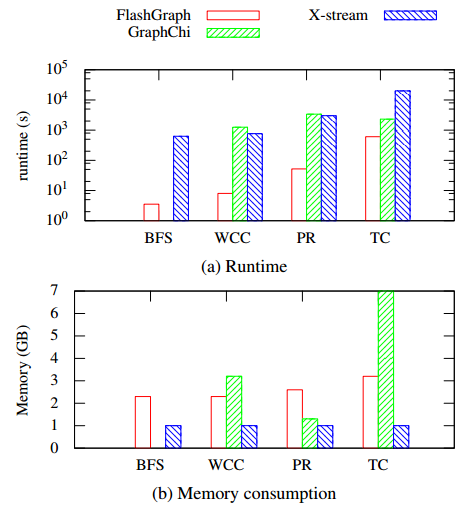


图3.3 FlashGraph边存储

可以看到，在性能上，FlashGraph相对于于X-stream和GraphChi取得了绝对的优势；在内存占用上，尽管FlashGraph在内存中存储了所有的节点信息，整体占用率相比其它基于外存的图引擎却并没有显得过于臃肿，在某些测试中相比于GraphChi还小了很多。

# 我的思考

FlashGraph是一个非常值得研究的系统，他涉及内部存储、外部存储、作业调度等计算机领域内的经典技术。它的成功表明，一个优秀的存储系统的设计必然会结合计算机学科的多个领域。

在前面的设计中，原作者提到：FlashGraph及其提到的许多其它图引擎，都是基于节点程序来完成各式各样的图操作。这个特性这引发了我在两个方面的思考。

## FlashGraph线程模型可能存在的问题和改进思路

根据原论文的描述，FlashGraph会将图划分为几个相对独立的部分，并为每个部分分配一个处理节点程序的线程、每个线程有着自己的调度器，这里其实可以看出FlashGraph其实保留了传统分布式图引擎的一些特征——除了将图进行划分外，每个划分的调度器也是独立的。这样设计的优点在于可以减轻调度的开销，也方便扩展到集群上；但也会带来另一个问题：它难以保证节点程序总是能够均匀地分配到不同的线程上。

例如，假设一个图操作恰好集中地访问同一个分区内的节点，那么按照现有的方案，将会有大量的节点程序将进入同一个分区的队列中，并只能被单个线程所处理，这样从而大大浪费了多核系统的优势。因此， FlashGraph现有的固定的线程模型，在单计算机上可能并不是一个最优的方案。

相应地，我认为一个可行的改进方案是，将调度器从各个线程中提取出来，为不同的线程统一分配节点程序。这在传统分布式图引擎上可能会带来巨大的调度开销，但是对于FlashGraph这种单机多核的图引擎，这种调度的开销是很容易通过调整每一次调度的节点程序个数来达到忽略不计的。

这种模型的实现有很多成熟的框架，一种便是基于线程池来动态调度执行节点程序，成熟的实现包括OpenMP、TBB等，它们都提供了简洁的接口，同样能够实现原作者所述的“简单而且稳定”。

## 基于异构计算的节点程序执行模型的可行性

节点程序的设定与目前计算机图形学绘制中的“顶点程序”十分类似，两者的英文原文同样是Vertex Program。不论是图引擎中的节点程序，亦或是图形学中的顶点程序，这种细粒度的并发任务非常适合于在可通用编程的图形处理器GPU上运行。

在早些年前，就有商品级GPU提供可编程的顶点程序；而近几年来更成熟的CUDA、OpenCL等规范则使CPU+GPU的异构计算更加成熟。考虑到GPU拥有比CPU多2~3个数量级的执行单元，将现有图引擎的执行单元移植到GPU上可能将带来可观的性能提升。

对于GPU而言，其最密切的内存储器是显存。而目前单张商品级显示卡的显存容量已经达到11GB，根据FlashGraph提供的数据计算，一张上述的显示卡同样可以存储多达17亿的节点。在这个假设下，边的存储同样可以放在外部的SSD阵列中，而内存则可以作为外存和显存之间的更大容量的缓存，来缓解两者之间速度的不对等，获得更大的吞吐量。

基于GPU的图引擎还有一个优点在于，即便一个显示卡的计算能力、存储容量有限，但显示卡之间是容易扩展的，并且显示卡之间的通讯基于高速的PCI-E总线，双向通讯速率高达32GB/s，且由CPU片内集成的PCI-E控制器直接控制，其不同计算节点之间的通讯代价明显低于基于网络的计算机之间的通讯代价。对于传统的分布式内存图引擎而言，原文提到，网络开销成为了其性能的拖累；如果以GPU集群替代原来的计算机集群，那么GPU间的通讯开销必然远远低于基于计算机网络的通讯开销，是一个值得尝试的研究话题。