華中科技大學

本科生毕业设计

基于多 GPU 的动态图更新和处理机制研究

院　　系	计算机科学与技术
专业班级	CS1601
姓　　名	刘本嵩
学　　号	U201614531
指导教师	张宇

2020 年 06 月 10 日

摘 要

由于图分析任务经常需要大量的计算能力，GPU 已经被业界广泛用来加速图分析任务。然而，在社交网络，欺诈检测，网络安全等领域的应用中，涉及到大规模且快速更新的动态图，在这种情况下不得不经常对图进行重构操作。这大大影响了此类应用的性能表现。工业界和学术界出现了很多基于 GPU 的大规模动态图处理技术，但它们往往在利用 GPU 的强大计算资源的同时，没能有效的利用全部 CPU 的计算能力。

以高性能的 GPMA+算法为基础，在 CPU 和 GPU 上同时并行运行动态图处理任务可以进一步提高此类应用的性能，并提高系统内的资源利用率。设计实现了一个同时利用 CPU 与 GPU 进行高效的动态图更新任务的系统，利用修改过的 GPMA 数据结构作为单个设备内的存储数据结构，同时实现了一个有效的运行时计算任务管理器，用来负责 CPU+GPU 平台上图的分区与更新的负载均衡。最终实现了 CPU 与 GPU 的多设备协同工作，验证了 CPU/GPU 的协同工作相比于单个 GPU 设备的性能提升。实验结果表明，系统能够同时充分利用平台的 CPU 与 GPU 资源，正确且高效的进行动态图更新与图分析任务。

关键词：动态图处理；大数据；分布式计算；GPU 图处理；混合图更新；

Abstract

Since graph analysis tasks often require compute-intensive operations, GPUs have been widely used to accelerate graph analysis tasks. However, many applications, such as social networking, fraud detection, and network security, involve massive and rapidly updated dynamic graphs. In these applications, the graphs have to be rebuilt again and again. This behavior greatly affects the performance of these applications. Many large-scale GPU-based dynamic graph processing technologies have appeared in industry and academia, but they usually fail to effectively utilize the computing power of CPUs while using the powerful computing resources of GPUs.

Based on the high-performance GPMA+ algorithm, running dynamic graph updating tasks on both CPU and GPU in parallel can further improve the performance of these applications and increase the resource utilization of the system. This paper implements a system that utilizes both CPU and GPU for efficient dynamic graph update tasks. It uses modified GPMA as the low-level data structure within a single device, and implements a device manager that supports high performance CPU/GPU hybrid graph processing. Finally, the collaborative work of both CPU and GPU is successfully implemented, and the performance improvement of hybrid processing compared to single devices is verified. The experimental result shows that the system can make full use of the CPU and GPU resources of the platform, and perform dynamic graph updating and graph analysis tasks correctly and efficiently.

Keywords: Dynamic Graph Processing; Big Data; Distributed Computation; GPU-based Graph Processing; CPU/GPU Hybrid Graph Updating;

目 录

1 绪 论

本章我们首先介绍了当前动态图处理系统面临的挑战和技术发展趋势，然后分析了动态图处理技术的产生及发展现状，介绍了国内外在 GPU 动态图处理领域的相关研究工作，并对本文的主要研究内容及工作意义作了具体说明。

1.1 课题背景

1.1.1 研究背景和趋势

在大数据时代，图处理系统面临着越来越大的数据量。大规模图处理任务已经成为社交网络(例如 Twitter 和 Facebook)，万维网，推荐系统等涉及高维数据的系统的重要组成部分。这些图往往是持续动态更新的，例如 Twitter 平均每天有 5 亿个推文[1]，大型 ISP 每小时每个路由器传输 10 亿个数据报[2]；同时这些大规模图数据往往还有进行实时分析的需求[3，4，5，6]，因此对大规模动态图的高效处理非常重要。

在最早期的图计算应用中，往往使用 CPU 进行动态图的更新和处理。后来，随着数据量的增大和大数据时代的到来，CPU 变得无法承受这庞大的数据量。现在，在 GPU 高性能节点上构建高效的大规模图数据的算法和系统已经日益成为研究热点，得益于 GPU 的极多的计算核心数量，它拥有比 CPU 强大得多的并行计算性能[7]。然而，使用 GPU 进行动态图处理需要在显存与主存之间进行大量数据传输，带来了 IO 上的开销。对于动态图的更新和处理，仍然没有充分的利用计算机的所有计算资源。因此，如何有效在 CPU-GPU 混合系统上，同时利用 CPU 和 GPU 的可用资源，进行动态图更新和处理是一个急需解决的问题。

1.1.2 面临的问题和挑战

同时充分利用 CPU 和 GPU 进行动态图更新。现有的大数据动态图更新系统中，大都是仅使用 GPU 设备来进行计算的。尽管算法已经针对 GPU 的众核特点而进行了充分的优化，但 CPU 到 GPU 的数据传输延迟仍然对性能造成重要影响。如果能够将部分数据交由 CPU 和主存进行处理，就可以降低任务延迟，减

少计算过程中 IO 等待时间的浪费，并因此达到更高的吞吐量。

以 GPMA+算法为例，我们在测试中发现，GPMA+的 GPU 计算核心利用率可以达到 70%左右，但 CPU 却只有一个核心在满负荷工作.如果能够同时发挥 CPU 和 GPU 的计算能力，就能够更充分的利用 CPU 靠近主存的优势，进一步提升动态图更新任务的性能。

1.2 国内外研究现状

1.2.1 基于 GPU 的图处理系统

自从 GPU 图处理问题成为研究热点以来，很多相关的研究已经陆续开展起来: Gunrock[8]是一个利用 GPU 进行高性能图处理的库。它对常用的图操作进行了抽象，提供了简洁而强大的 API，并针对 GPU 计算中的 IO，负载均衡，任务管理等问题进行了很多特定优化，获得了比 CuSha[9]，MapGraph[10]，Ligra[11]更优异的性能表现。

在此基础上，Groute[12]针对多 GPU 系统的通信优化问题提供了对应的高层抽象，并增加了分布式工作表(Distributed Worklists)，流水线处理(Pipelined Operation)等一系列针对多 GPU 的优化，大大降低了 GPU 间和 CPU 间的通信延迟带来的性能损失问题，在多 GPU 系统上取得了比 Gunrock 更好的性能表现。

关于分布式图处理的分区问题，Gurbinder Gill 讨论了包含 1D，2D 和其他策略等多种分区方式，揭示了不同分区策略对分布式系统中的通信模式的影响，论证了 CVC 这种分区方式更有利于不同运算节点间的高效交流[13]。

1.2.2 GPU 上大规模动态图的存储

实际生产环境中出现的动态图往往是极度稀疏的，因此图的存储等同于稀疏矩阵的存储。现有的 GPU 上的稀疏图处理都依赖于用排序好的元素来组成某种优化的数据格式[14,15,16,17,18,19]。 James King 提出的 DCSR[17]数据结构在 CSR[14,15]的基础上，增加了额外的 Segments 来缓存更新，在充分利用 GPU 的巨大并行计算能力的同时，还能兼容为传统 CSR 设计的图算法，在 GPU 动态图更新领域获得了很大性能提升。

但是，DCSR 是为插入设计的，而不支持删除和快速搜索。为此，新加坡国

立大学提出的 GPMA 和 GPMA+数据结构[20]，与 DCSR 采用了不同的设计思路。它采用类似于 B 树的无锁结构来管理树存储空间，并使用细粒度的 re-balance 操作取代 DCSR 的 Segments Merge 操作，在频繁的动态图更新中更充分的发挥了 GPU 的众核计算能力，并且二进制数据结构一样兼容 CSR 格式。

1.2.3 大规模动态图的实时分析

除了动态图的存储问题，对于大数据动态图中的实时分析需求，Keval Vora 注意到动态图中不同 Snapshots 的相似性，由此提出了 Fetch Amortization 和 Processing Amortization 的优化方案来减小 IO 和计算负担[21]，并在 ASPIRE 和 GraphLab 图处理系统中取得了很好的效果。后来，GraPU[22]在此基础上，针对快速更新的动态图提出了在 Buffer Node 对 Updates 进行 pre-computing，以及通过划分 Component 与 Subgraph 来平衡不同计算单元 workload 的方法，使之更加适合并行处理。

1.3 研究目的和主要内容

在本次课题研究的目的是，在动态图处理任务中同时充分利用 CPU 和 GPU 的可用资源。我们首先在 CPU 和 GPU 上分别实现 GPMA+的数据结构，然后设计并实现一个支持 CPU+GPU 混合多设备的分布式 GPMA+数据结构，并设计和实现一个用于测试的分布式图处理算法。然后设计在不同设备之间分配任务的组件，并测试多种任务分配策略的实际性能表现。

1.4 论文结构

本文的主要内容如下：

第一章我们首先介绍了当前涉及动态图处理的系统面临的挑战和技术发展趋势，然后分析了大规模动态图处理需求的产生及发展现状，介绍了国内外在相关领域的研究工作，并对本文的主要研究内容及工作意义作了具体说明。

第二章首先介绍了几种现有的用于图存储的数据结构，特别是与动态图相关的。随后介绍了 DCSR 和 GPMA+这两种现有的动态图存储技术的优劣。最后介绍了在存在图分区的图处理系统中，不同分区策略对系统性能的影响。本文将以

GPMA+作为研究的基础，设计并开发一种能够在 CPU 与 GPU 混合多设备运行 GPMA+算法的系统。

第三章我们在单 GPU 的 GPMA+算法的基础上，设计了 CPU 上的 GPMA 数据结构与 GPMA+算法，完成了多设备 GPMA 的数据结构和算法的设计工作，讨论了在多设备 GPMA 上进行 BFS 等常见图算法时的问题，以及在多设备间进行图分区的策略问题。

第四章比较详细的描述了我们的多设备动态图更新系统的具体实现，以及配套的 BFS 图算法。我们详细介绍了系统中用到的主要数据结构，单设备上的 GPMA 数据结构的实现，还实现了多设备的负载均衡相关功能，并对第三章设计的图分区策略进行了实现。最终完成了一个完整高效的具有动态图加载，更新，分析功能的系统。

第五章对我们的多设备动态图更新系统进行了详细的测试，这包括分别对 CPU 和 GPU 上单设备代码的功能测试，对真正的多 CPU 单 GPU 混合动态图更新进行了正确性测试。同时，我们测试了单设备代码在扩展为多设备时的并行度及调度开销，以及对 GPU 设备负载比例进行调整测试，这不仅确定了两套测试系统中的最佳 GPU 因数，还证明了多设备负载均衡对整个系统加速的有效性。

第六章总结了本文所做的各种工作，对整个系统的特点进行概括，并展望了动态图研究领域的前景和发展方向。

2 相关背景技术介绍

本章介绍 CPU+GPU 混合动态图处理使用到的相关技术，这包含 GPMA+数据结构和多设备的负载均衡方案。

2.1 系统需求分析

现有动态图处理系统往往是采用单一的多 CPU 或多 GPU 进行动态图处理。然而，使用 GPU 进行动态图处理的系统，通常不能有效的利用系统中的 CPU 核心。尽管算法已经针对 GPU 的众核特点而进行了充分的优化，但 CPU 到 GPU 的数据传输延迟仍然对性能造成重要影响。以 GPMA+算法为例，我们在测试中发现，GPMA+的 GPU 计算核心利用率可以达到 70%左右，但 CPU 却只有一个核心在满负荷工作。

如果能够将部分数据交由 CPU 和主存进行处理，就可以降低任务延迟，减少计算过程中 IO 等待时间的浪费，并因此达到更高的吞吐量。我们需要实现一个能够同时发挥 CPU 和 GPU 计算能力的动态图更新系统，以便更充分的利用 CPU 靠近主存的优势，进一步提升动态图更新任务的性能。

2.2 现有图存储数据结构

由于现实中的大规模图数据都是稀疏的，因此图数据的存储一般可以等同于稀疏矩阵的存储。

COO 格式是最简单的稀疏矩阵格式。它用三个数组存储整个矩阵，数组中分别保存行索引，列索引以及矩阵中所有非零条目的值。为了有效执行 SpMV 操作，必须按行对 COO 格式内的条目进行排序。

CSR / CSC 格式与 COO 相似，因为它们的数组除了值外，还完全存储三个数组中的两个，即列索引或行索引。行索引数组或列索引数组（分别对应 CSR 或 CSC）被压缩，仅存储与其他两个数组中的行/列位置相对应的偏移量。对于 CSR，行偏移量数组中的第 i 和 i + 1 项分别存储行 i 的起始和结束偏移量。由于 CSR 可以压缩列索引的特点，因此已被证明是内存访问和 SpMV 性能最好的数据

存储格式之一，并且已经被广泛使用[14,15]。CSR 的压缩数组减少了内存带宽的占用，这是加速 SpMV 操作的重要因素。

ellpack（ELL）格式使用两个数组，每个数组的大小为 m×k（其中 m 为行数，k 为固定宽度），用于存储列索引和矩阵[23,24]的值。这些数组以列优先顺序存储，以便对不同行进行更高效的并行访问。这种格式适合每行具有固定元素数的矩阵，而如果一行中存在过多的元素，ELL 的内存空间开销将会非常昂贵。混合 ellpack（HYB）格式通过结合使用 ELL 和 COO 提供了一种折衷方案。它将矩阵元素尽可能多地存储在 ELL 部分中，而元素数量大于 ELL 行宽度的元素将溢出到 COO 结构中。由于线程扭曲能够以有效的并行方式浏览连续的行，因此 ELL 和 HYB 在 SIMD 架构上已变得很流行[25]。

与此同时，研究者还开发了许多其他的稀疏矩阵格式，包括对角线（DIA），锯齿形对角线存储（JDS），块对角线（BDIA），天际线存储（SKS），平铺 COO（TCOO），块 ELL（BELL），和 sliced-ELL（SELL）[26]等。它们为一些特定类型的矩阵提供了更好的性能。

2.3 DCSR 技术介绍

DCSR(动态 CSR)[17]是一种在 CSR 基础上改进得到的数据格式，它可在保持高效的 SpMV 操作性能的同时进行快速的动态更新操作。 DCSR 的动态分配过程维护一个 row offset 数组，这个密集的数组像 CSR 一样保存了每个行的偏移量信息。列索引和矩阵元素存储在另外两个数组中，该数组在逻辑上分为多个 segment，其方式与 CSR 行偏移量对列索引和矩阵元素进行分区的方式相同。每个 segment 都是一段连续内存，用于存储行中的元素。段可能包含空隙，用来允许高速的插入。

在向某一行中插入新的元素时，先确定本行的最后一个 segment，如果空余空间足够，则会执行插入。否则就分配新的 segment 来存储新的插入。defragment 过程被偶尔执行，来减少 segments 的数量。同时，如果要运行 CSR 上的算法，defragment 过程可以将 DCSR 数据结构转换成与 CSR 几乎完全相同的底层结构，以便运行为传统 CSR 设计的图算法。

2.4 GPMA+背景技术介绍

GPMA 中的并发插入受 PMA 在 CPU 上的启发，它在 GPU 上同时并行处理一批插入。直观地，GPMA 将插入分配给线程，并使用基于锁的方法并为每个线程同时执行 PMA 算法，以确保一致性。更具体地，预先识别所有插入的叶子节点上的段，然后每个线程从下到上检查插入的片段是否仍然满足其阈值。对于每个特定的段，以互斥的方式访问它。此外，在更新位于树上同一高度的所有段之后，所有线程将同步，以避免可能的冲突，因为较低级别的段完全包含在较高级别的段中。[20]

给定一个已经存储在 GPMA 上的图，还需要为 GPMA 设计对应的图算法。特别地，让现有算法能够访问 GPMA 格式的图数据至关重要。对于 CSR 数据格式，大多数算法都通过访问 CSR 中的有序数组来定位要访问的元素。因为存储在 GPMA 上的稀疏矩阵数据也是一个有序数组，该数组与 CSR 类似地具有交错的间隙。因此，我们能够用 GPMA 的操作有效地替换现有算法中对 CSR 的操作。

由于 GPMA 具有无序的内存访问，锁开销，线程冲突，线程负载不平衡这 4 个问题，更优化的 GPMA+算法被提出。这种算法具有以下 3 个特点:

（1）更新首先按其键排序，然后分派到 GPU 线程以根据排序顺序定位其对应的叶段。

（2）将属于同一叶段的更新分组以进行处理，GPMA +以自下而上的方式逐级处理更新。

（3）在树的每一级中，利用 GPU 原语来调用所有计算资源以进行段更新。

在以上算法中，组件（1）解决了 GPMA 中未分批进行内存访问的问题，因为对更新线程进行了预先排序以实现有规律的遍历路径，优化了缓存性能。组件（2）完全避免使用锁，从而解决了原子操作和线程冲突的问题。最后，组件（3）利用 GPU 原语在所有 GPU 线程之间实现工作负载平衡。值得注意的是，GPMA 和 GPMA+的算法有区别，而底层数据结构并没有大的区别。

2.5 常用图分区技术介绍

在 1D 分区方法中，节点在主机之间分区。 如果某一个主机拥有节点 n，则

将连接到 n 的所有出边或入边分配给该主机。 在图分析领域，这种划分策略称为 Outgoing 或 Incoming Edge-Cut[27,28,29,30]。 用矩阵理论的术语来说，这相当于为主机分配行（或列）。 一维分区通常用于科学计算应用中的代码中；代码可以通过向所有主机分配大致相等数量的节点来实现计算负载平衡。 在 2D 的分区中，邻接矩阵沿两个维度都被分块，并且每个主机被分配了一些块。与一维分区不同的是，某一个节点的出边或入边都可以分布在不同的主机之间。在图分析领域，这种划分策略被称为 Vertex-Cut。二维分区包含 BVC，CVC，JVC 等常用策略。

CheckerBoard Vertex-Cuts (BVC)将邻接矩阵划分成相同大小的块，并且分配给每个主机，被分配的主机就是这个块内所有节点的 master。Cartesian Vertex -Cuts (CVC)将节点按照任意的一维分区方法分配给不同的主机，然后被分配的主机就是此节点的 master。与 BVC 不同的是，CVC 的分区可以不是等大小的。Jagged Vertex-Cuts (JVC)与 CVC 类似，然而 JVC 不是将整个矩阵分为相等大小的块，而是每一行可以被独立的进行分区，这样可以得到更平衡的负载分配。在 Gurbinder Gill 的论证和测试中发现，CVC 的分区方式可以最小化不同主机之间的数据交换，使得整个图系统的通信更有效率[13]。

2.6 本章小结

本章介绍了与 CPU+GPU 混合动态图更新系统相关的背景技术，包括传统的图存储数据结构，DCSR 动态图存储数据结构，GPMA+的数据结构及算法，和动态图分区技术。我们详细讨论了各种算法与技术的基本设计思路，特点，和各自的优势。

3 CPU+GPU 混合动态图处理系统设计

前面论述了使用 GPMA 数据结构和 GPMA+算法，完成多设备动态图处理任务的必要性、重要性和可行性。GPMA+算法和数据结构拥有更强的并行优化潜力，非常适合用作 CPU-GPU 混合动态图更新任务的底层算法之一。本章详细描述了在动态图更新任务中，关键数据结构和算法的设计，并就各功能设计时需要考虑的问题及解决方案进行了讨论。

3.1 功能需求

(1) 首先在 GPMA+算法的基础上，完成一个高性能的算法在 CPU 上的实现。以便支持接下来的 CPU-GPU 多设备 GPMA。

(2) 新实现的动态图更新系统能够高性能的完成大规模图数据的更新，包括插入和删除等操作。重点优化更新的过程，用最小的开销完成并行优化。

(3) 新实现的动态图更新系统能够充分利用系统内的多核 CPU 和 GPU 资源，要将图的数据分区在不同的设备上，并且在进行更新时充分利用每一个 CPU 核心和 GPU 核心的计算能力。

(4) 除了动态图的更新操作外，还应该在分区后的图上支持至少一种常用的图算法，用于正确性测试。程序用常规图算法的结果来检验正确性。

3.2 系统总体设计

3.2.1 单设备数据结构和图更新

在单个 GPU 或单个 CPU 上的数据结构采用已知单 GPU 上性能最好的 GPMA+。单设备上的 GPMA+数据结构概览如图 3.1 所示。

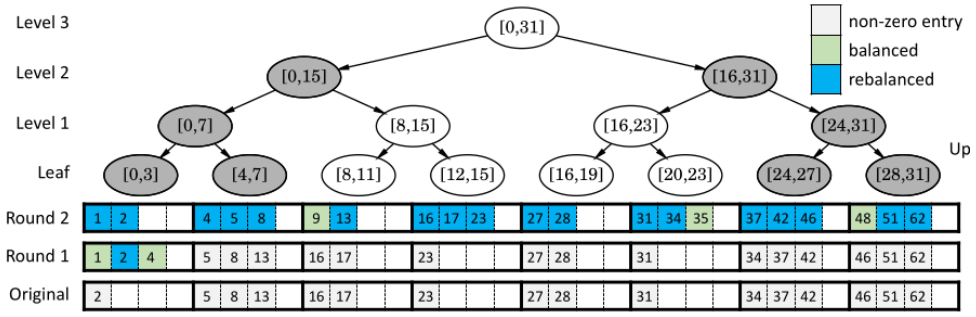


图 3.1 GPMA+数据结构概览图

GPMA+数据结构是将 PMA 数据结构针对 GPU 进行优化而来的。PMA 通过在一个有序的数组中预留空隙，来允许高速的并行插入操作。PMA 是一种自平衡的二叉树结构，对于在任意一个高度上的节点(或者说是它所对应的一个 segment)，PMA 记录了这一个 segment 的空槽使用率的上界和下界，并在必要的时候进行 re-balance 操作。正如一般的自平衡二叉树那样，这种 re-balance 操作会由叶子向根进行递归，并在必要时加倍或减半整体数组的空间。

在这种数据结构中，一般的插入或删除，很少需要递归到深层次的节点，并且不同节点的 re-balance 操作可以被并行化，可以取得较好的并行度。

Algorithm 4 GPMA+ Segment-Oriented Insertion

```
1: procedure GpmaPlusInsertion(Updates U)
2:     Sort(U)
3:     Segs S ← BinarySearchLeafSegments(U)
4:     while root segment is not reached do
5:         Indices I ← ∅
6:         Segs S* ← ∅
7:         (S*, I) ← UniqueSegments(S)
8:         parallel for s ∈ S*
9:             TryInsert+(s, I, U)
10:        if U = ∅ then
11:            return
12:        parallel for s ∈ S
13:            if s does not contain any update then
14:                remove s from S
15:            s ← parent segment of s
16:    r ← double the space of the old root segment
17:    TryInsert+(r, ∅, U)

18: function UniqueSegments(Segs S)
19:    (S*, Counts) ← RunLengthEncoding(S)
20:    Indices I ← ExclusiveScan(Counts)
21:    return (S*, I)

22: procedure TryInsert+(Seg s, Indices I, Updates U)
23:    n_s ← CountSegment(s)
24:    U_s ← CountUpdatesInSegment(s,I,U)
25:    if (n_s + |U_s|)/capacity(s) < τ then
26:        Merge(s, U_s)
27:        re-dispatch entries in s evenly
28:        remove U_s from U
```

图 3.2 GPMA+插入过程中，算法的伪代码

对 GPMA+数据结构的单设备插入过程如图 3.2 所示。Updates 是一个由 Key，Value 对构成的数组。在具体的实现中，这里的 key 是由起始终止节点的 ID 构成的，表示一条边的变量。而 value 用 1 表示插入，0 表示删除。

GPMA+的插入过程，首先在第 2 行将更新按 key 进行排序，然后在第 3 行并行查找到节点所对应的段。随后 GPMA+在 4-15 行，逐级更新 segments，完成更新操作。其中，第 7 行的 UniqueInsertion 将属于同一个 segment 的更新分类合并到 S*，并在 TryInsert 过程中检查 segment 的密度是否满足上下界要求，如果满足要求便在这一级完成这一批更新的插入或删除。

在 UniqueSegments 的实现中，使用了适合 GPU 的 RunLengthEncoding 和 ExclusiveScan。对于核心数量相对较少的 CPU，使用常规的同步原语代替这一部分操作。对于核心数量极高的 GPU，使用了 NVIDIA CUB 库来完成这两个操作。

这一步产生的 IndexSet，被传递到 TryInsert 函数中是为了在 GPU 中辅助进行同步。

最后，如果 TryInsert 过程由于 segment 密度不符合要求而失败，GPMA+会逐级向更高的根节点进行递归。如果已经递归到根节点，则会对整体存储空间进行翻倍或减半，并最终完成所有的插入操作。

GPMA+的插入与删除使用相似的逻辑。在我们的实现中，通过插入一个 value 为 0 的边来表示删除，对具体实现逻辑的修改是极小的。

3.2.2 CPU 和 GPU 的多设备图更新

在对多设备 GPMA 进行初始化时，我们为每一个设备都分配一个 GPMA 数据结构。一般来说，我们为每个 CUDA 设备(GPU)分配一个 GPMA 结构，并为每个 CPU 核心分配一个 GPMA 结构。如果系统中由 N 个 CUDA 设备，M 个 CPU 核心，则需要分配 N 个 GPU 上的实例，和 M-N 个 CPU 上的实例。随后，系统对每一个 GPMA 实例进行初始化。

在进行插入时，如图 3.3 所示，GPMA_multidev 会先将这一批插入按分配器的策略分配给每一个 GPMA 实例。随后，如果必要，将 CPU 上的 buffer 拷贝至 GPU，然后并行的启动每一个实例的 update_gpma 操作。在每一个设备的 update 操作结束后，整体的更新就被插入完成了。

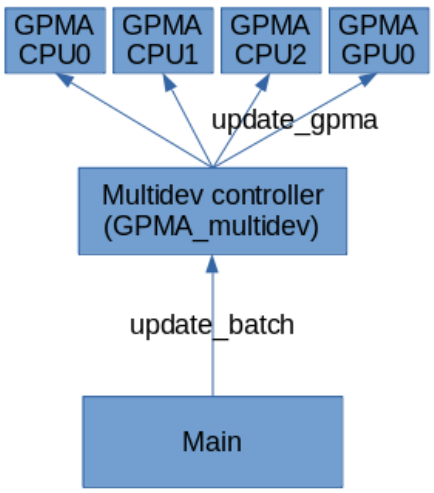


图 3.3 多设备 GPMA 插入示意图

3.2.3 单设备的图算法

由于 GPMA 的数据结构与传统 CSR 具有相似性，因此很容易将为 CSR 设计的算法迁移到 GPMA 上。如图 3.4 所示，GPMA 会像 CSR 一样记录 Row Offsets 数组，并利用 Row Offsets 来访问对应偏移量上的元素。随后，在并行 GPU 算法中剔除掉 NULL 元素(这个过程容易并行，因此在 GPU 上很快)，即可像 CSR 一样进行分析。

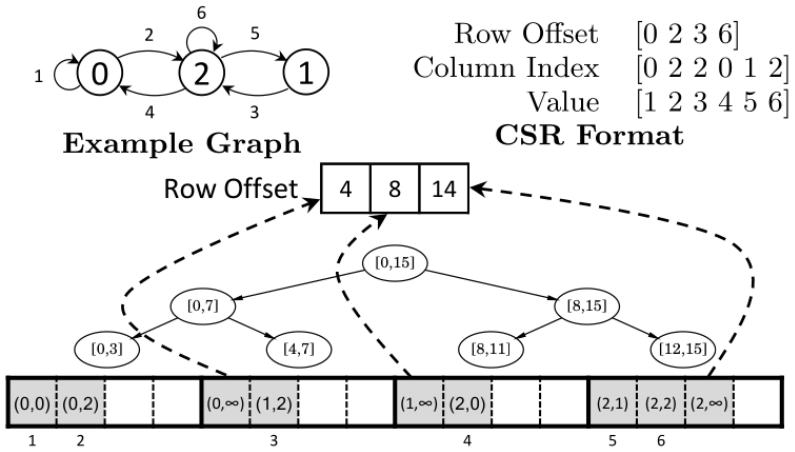


图 3.4 GPMA 与 CSR 结构示意图

GPU 上进行的 GPMA 算法分为两个部分: 在循环中，先通过 gather 操作，将 node queue 的所有节点的邻居保存在 edge queue 中，然后通过 contract 操作，将 edge queue 中的所有下一级节点进行标注，选出其中的新节点，标注上 level，并放进 node queue。随后重复这个循环，直到 edge queue 中没有任何新节点，BFS 操作结束。这整个过程如图 3.5 所示。

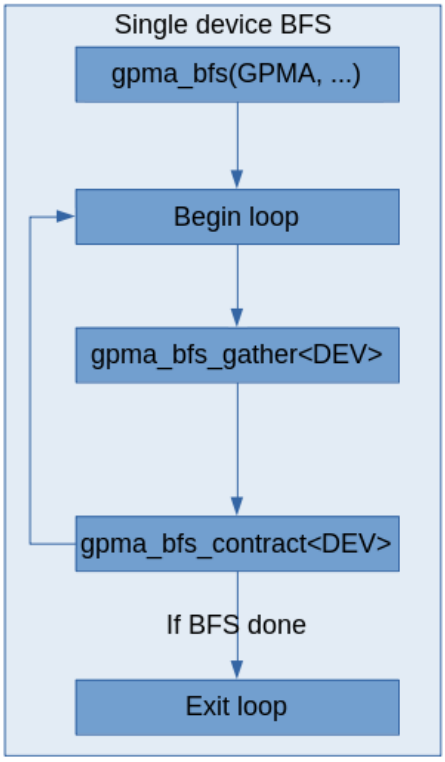


图 3.5 单设备的 BFS 过程示意图

将这 gather 和 contract 操作分开的做法，增加了算法的并行度，使算法在 GPU 上获得了更好的性能。对于 CPU 上的操作，只需用 OpenMP 将 contract 和 gather 操作并行化即可。

3.2.4 CPU 和 GPU 的多设备图算法

为了将单个设备上的 BFS 算法改为支持多个设备，我们设计了多设备上并行完成的 BFS 过程。

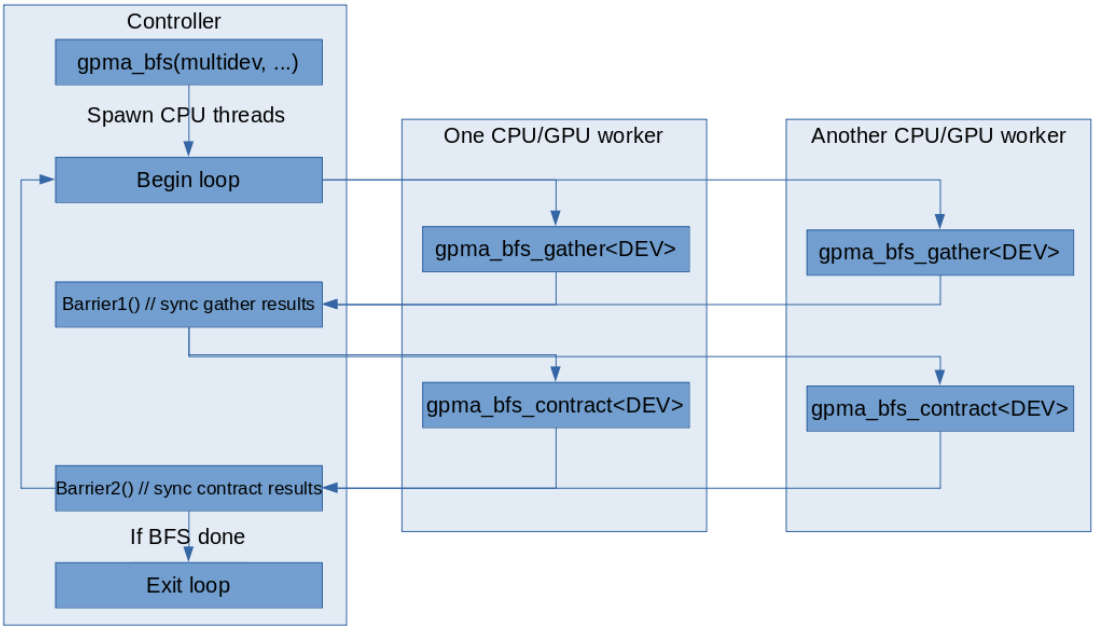


图 3.6 多设备的 BFS 过程示意图

我们在多个设备上进行的 BFS 过程如图 3.6 所示。我们首先为每一个 GPMA 实例产生一个 CPU 线程，然后每一个 CPU 线程执行一个实例的 gather 操作。在 gather 操作完成后，所有线程进行一次同步，并对每个实例产生的 edge queue 进行合并。

由于我们的计算机往往是几十个 CPU 核心，为了避免对大量的队列进行合并带来的性能开销，我们在所有的 CPU 实例共用同一个 node 队列和 edge 队列，它们用 CPU 上的原子操作进行元素的并行插入。而不同 GPU 实例一般不会共享内存空间(除非使用 cudaManagedMemory)，而机器上的 GPU 实例一般只有一到两个，所以每一个 GPU 实例都有一个独立的 node 队列和 edge 队列参与合并。

在 edge queue 合并完成后，每一个实例继续执行自己的 contract 操作。操作完成后，用相同的方法来合并 node queue 和 results 数组。随后，如果 node 队列为空，所有线程就可以全部结束，操作完成。

3.2.5 图分区策略

在本文的工作中，将节点和更新分配到设备的分配策略对性能影响非常重要。首先，GPMA 数据结构中存储的是边。我们令每一个设备拥有一些节点 ，将每一个边的起始节点所属的设备，作为负责存储这一条边的设备。同时，随着边的

数量的增加，分配器在为新的节点分配设备的同时，可以追踪目前各个设备的负载情况，并且尽可能偏好较空闲的设备。

首先，系统要确定一个系数，用来表示单个 GPU 设备与 CPU 设备的算力的比值。这个系数根据服务器上的硬件配置，运行性能测试来进行确定。然后，分配器可以根据这个系数以及负载情况，在运行时实时调整节点的分配情况。

在我们的 GPMA 数据结构，插入算法，和图算法的实现中，为了保证对边进行删除时，删除操作能被分配到含有这条边的节点上。因此，一个边被分配给一个设备后，暂时不支持迁移操作。

3.3 本章小结

本章详细描述了 CPU-GPU 混合动态图更新系统中，关键数据结构和插入，BFS，图分区等关键算法的设计，对其主要功能模块进行了详细的说明。并就各功能模块设计时需要考虑的问题及解决方案进行了讨论。

4 CPU+GPU 混合动态图处理系统实现

我们将设计一个动态图更新系统。该系统在传统的 GPU 图更新和图算法的基础上，利用类似于分布式系统的设计思路，让空闲 CPU 的算力也得到充分利用。我们实现了同时支持多个 CPU/GPU 并行计算的动态图更新系统，和配套的一个图算法。

4.1 GPMA+算法的通用实现

我们需要将 GPMA+的数据结构和算法，实现为既能在 CPU 上工作，又能在 GPU 上工作的模板类或模板方法。以便为后面的多设备 GPMA+提供基础实现。

4.1.1 CPU/GPU 的通用 GPMA 数据结构

```
template <dev_type_t DEV>
class GPMA {
  public:
    NATIVE_VEC_KEY<DEV> keys; // 存储实例中所有的 Key
    NATIVE_VEC_VALUE<DEV> values; // 存储实例中所有的 Value

    SIZE_TYPE segment_length;
    SIZE_TYPE tree_height;

    // 段的密度上限和下限，在超越限制时进行 re-balance
    static constexpr double density_lower_thres_leaf = 0.08;
    static constexpr double density_lower_thres_root = 0.42;
    static constexpr double density_upper_thres_leaf = 0.92;
    static constexpr double density_upper_thres_root = 0.84;
    thrust::host_vector<SIZE_TYPE> lower_element;
    thrust::host_vector<SIZE_TYPE> upper_element;

    // 为 CSR 格式兼容性而添加的额外成员
```

```
    SIZE_TYPE  row_num;
    NATIVE_VEC_SIZE<DEV>  row_offset;
};
```

数据结构中比较重要的字段是 keys 和 values，这里存储着图 3.1 所显示的最底层的数组。在数据结构初始化时，就会有一个 4 个元素的空树被插入到 keys/values 中。row_offset 成员也会在每一个更新的过程中被维护，这个数组就相当于 CSR 的 row offset 数组，在 BFS 算法运行过程中发挥重要作用。segment_length 和 tree_height 记录了树的总高度，和 segment 的长度，它们会在 update_gpma 过程中被维护。同时，结构中还包含了在 re-balance 中会用到的结构和阈值常量。

4.1.2 GPMA+插入过程

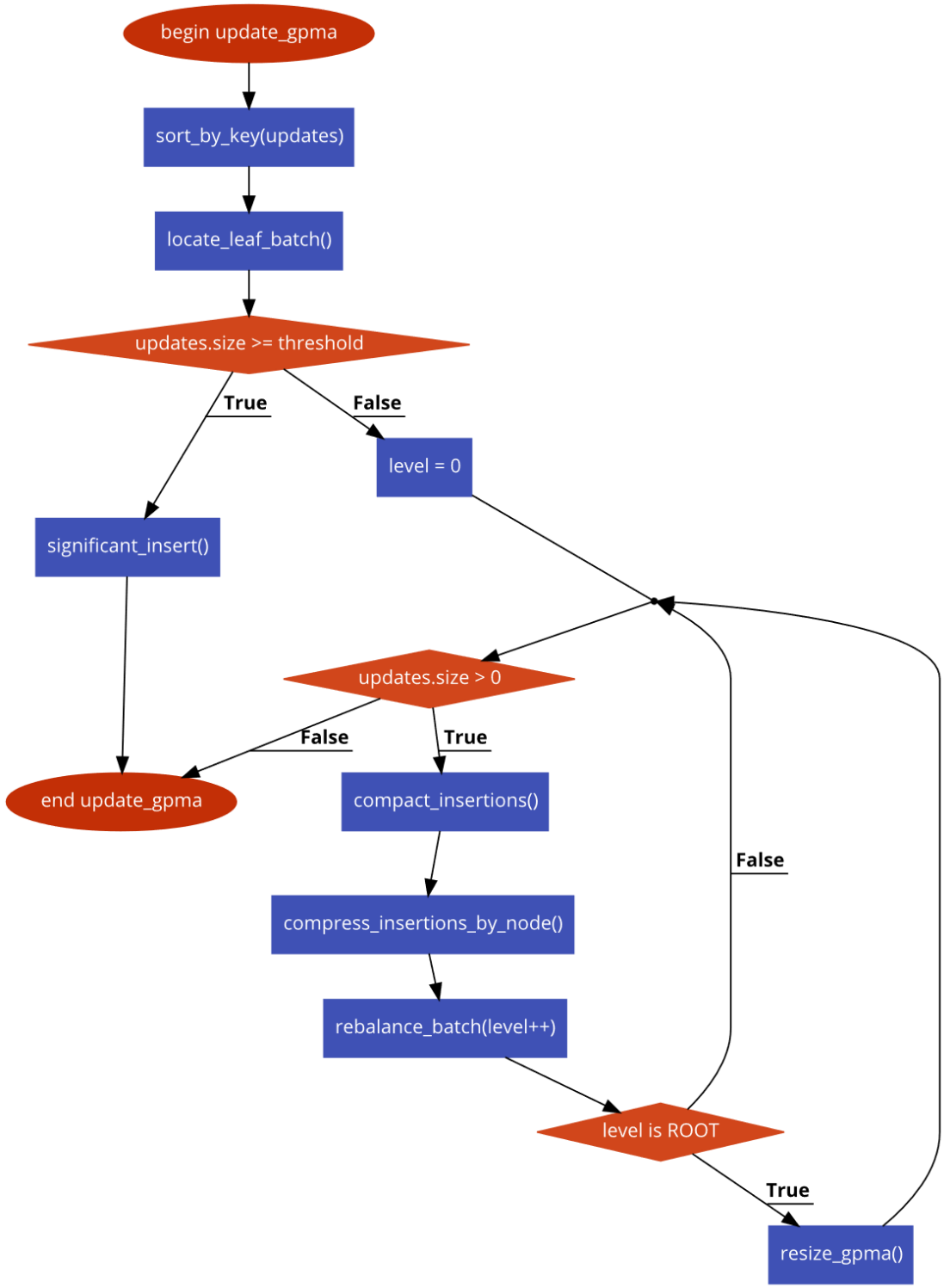


图 4.1 GPMA+插入算法工作流程图

在 GPMA+算法的图更新过程中，大致流程如图 4.1 所示。首先用 sort_by_key 将输入的更新进行排序，然后定位 key 所对应的叶子节点。随后对远大于当前

gpma 容量的巨大更新进行特殊处理，通过 significant_insert 过程，更高效的维护 GPMA 数据结构。

在一般情形下，程序会进入逐层迭代的循环中，在树的每一层尝试将对应于本 segment 的更新进行插入。如果 segment 的密度超过限制，则会将失败的更新留在 updates 数组中，并在下一次循环中，在更高一层再次尝试处理。最终，如果一直递归到根节点，都仍有更新未能被合并，就说明 GPMA 的当前容量不足以必须扩充整体 GPMA 的数据结构空间，调用 resize_gpma 完成新空间的分配。

在更新过程中，rebalance_batch 是最关键的部分。它负责将要更新的数据与原 segment 进行合并。在 GPMA<GPU>的 rebalance_batch 实现中，如果更新的大小小于 CUDA 支持的 block 数量，就直接将更新按固定的方式分配给每一个 cuda block。这有利于减少处理小更新时的调度开销。如果更新数量较大，就使用 rebalancing_kernel 函数，借助 cub 库的帮助，用更灵活的方式来为 cuda 核心分配任务。

而在 GPMA<CPU>的 rebalance_batch 实现中，使用类似的思路重新实现了 GPU 上数据量大于 block 数时的更新逻辑，调用被 OpenMP 并行化的 rebalancing_impl_cpu，完成 re-balance 操作。

4.2 多设备 GPMA 和图分区的实现

4.2.1 设备管理数据结构

GPMA_multidev 代表一个分布在多个设备上的 GPMA 结构，dispatcher 代表负责进行负载分配和均衡的结构。其主要数据结构如下文所示。

```
template <size_t cpu_instances, size_t gpu_instances>
struct GPMA_multidev {
private:
    static constexpr size_t instances = cpu_instances + gpu_instances;
    // 这两个数组存储所有设备上的GPMA 实例指针。
    std::array<GPMA<CPU> *, cpu_instances> ptrs_cpu;
    std::array<GPMA<GPU> *, gpu_instances> ptrs_gpu;
public:
```

```
        // 构造和析构上面的所有 GPMA 实例指针。
        GPMA_multidev(size_t row_num) {...}
        ~GPMA_multidev() {...}
    // 用来在不同设备间分配边的负载分配器。
        gpma_impl::dispatcher<cpu_instances, gpu_instances> dispatcher;
};

template <size_t cpu_instances, size_t gpu_instances>
struct dispatcher {
private:
        static constexpr KEY_TYPE hashSize = 1024;
        // 这是一个由单个 vector 组成的哈希表，用于缓存节点分配情况。
        quick_1way_hash_table<KEY_TYPE, size_t> mapKeyToSlot;

public:
        dispatcher()
                : mapKeyToSlot(hashSize, (size_t)(-1)) {}
        static constexpr size_t gpu_factor = 7;
        [[gnu::always_inline]] size_t select_device(const KEY_TYPE &k) ;
};
```

上述 GPMA_multidev 结构通过它的两个模板参数进行实例化。在此结构被构造时，它为每一个设备分配一个单设备 GPMA 数据结构。它还含有一个 dispatcher 结构，这个对象负责在 update_batch 时决定 key 所属的设备。

由于 dispatcher 和 GPMA_multidev 时解耦的，这里提供一种较简单的 dispatcher 实现。在 dispatcher 接口中，select_device 函数用于确定一个 key 所对应的设备编号。在我们的简单实现中，有一个表示 GPU 相比于 CPU 的相对性能的系数 gpu_factor，它通过实际测试来进行确定。mapKeyToSlot 是一个当作哈希表使用的普通数组，它由 hashSize 个槽，每一个 key 会被映射到一个槽上，然后槽内存储的 dev_id，就表示对该槽对应的 key 进行负责的具体设备编号。

4.2.2 多设备图更新和图分区

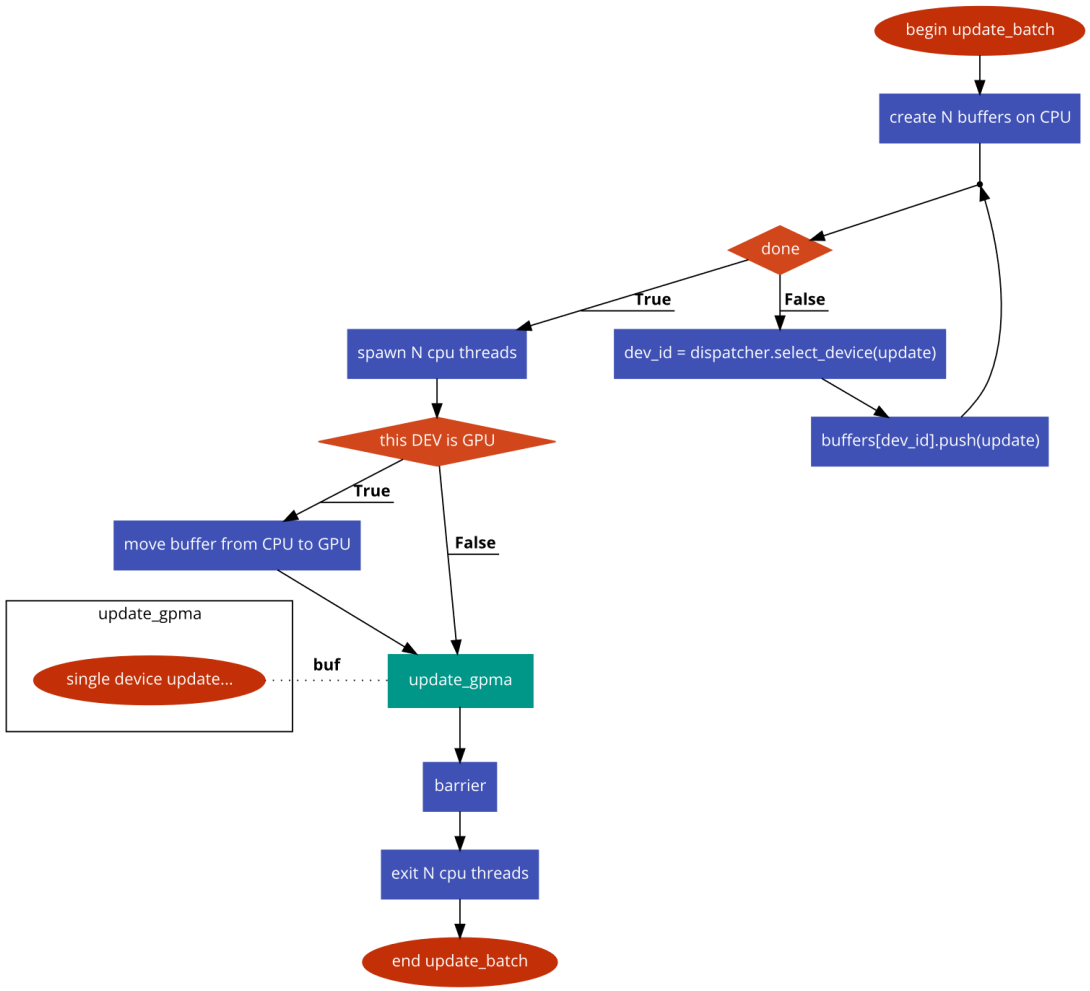


图 4.2 多设备图更新流程图

在多设备图更新中，使用一个 update_batch 代替单设备的 update_gpma。在 update_batch 过程中，首先将要更新的数据传递给 dispatcher 进行分配，然后将分配后的更新并行的发送给每一个设备，并行调用每一个设备的 update_gpma 方法，完成数据的更新。这个过程如图 4.2 所示。

在 dispatcher 的几种可能实现中，我们尝试了多种不同的分区方式。但是，注意到多设备 BFS 的实现中，存在一个在多设备间同步 node queue 和 edge queue 的过程，同时注意到对图的更新操作远大于 BFS 操作耗时。因此，当前应用中分区策略对通信模式的影响并不明显，我们选择了对更新操作更有利的分区方式。

在现在的实现中，所有的更新被分为固定数量的组，每个设备将会负责一些

任务组。任务组到设备的映射会在更新的过程中，按照 gpu_factor 系数来进行建立。而单个更新到任务组的映射，只与组的数量这个常量有关。这种做法在保证更新时性能的同时，减小了任务分配过程中的开销。

4.3 多设备 GPMA-BFS 实现

4.3.1 多设备 BFS 相关数据结构

```
template <size_t cpus, size_t gpus>
struct multidev_bfs_data {
private:
    native_vector<CPU, SIZE_TYPE> cpu_results, cpu_bitmap, cpu_nodeQ, cpu_edgeQ;
    native_vector<GPU, SIZE_TYPE> gpu_results, gpu_bitmap, gpu_nodeQ, gpu_edgeQ;
    SIZE_TYPE cpu_nodeQ_size = 0, cpu_edgeQ_size = 0;
    SIZE_TYPE *gpu_nodeQ_size, *gpu_edgeQ_size;
    std::atomic<bool> select_one_thread;
    Barrier barrier;

public:
    multidev_bfs_data(size_t node_size, size_t edge_size);
    void iteration_barrier_1(size_t dev_id);
    bool iteration_barrier_2(size_t dev_id);
};
```

BFS 所用到的数据结构主要有，表示每一个节点的层数的 results 数组，用来表示节点是否被计算过的 bitmap，表示下一层节点的 node queue，表示本层节点的所有邻居的 edge queue，加上 barrier 等用于同步的辅助数据结构。在 results 数组中，每一个节点用 0 表示未访问过，1 表示初始节点所在的层，依次类推。两个 barrier 函数用于不同设备 BFS 过程的同步，详细逻辑如 4.3.2 节所述。

4.3.2 多设备 BFS 算法实现

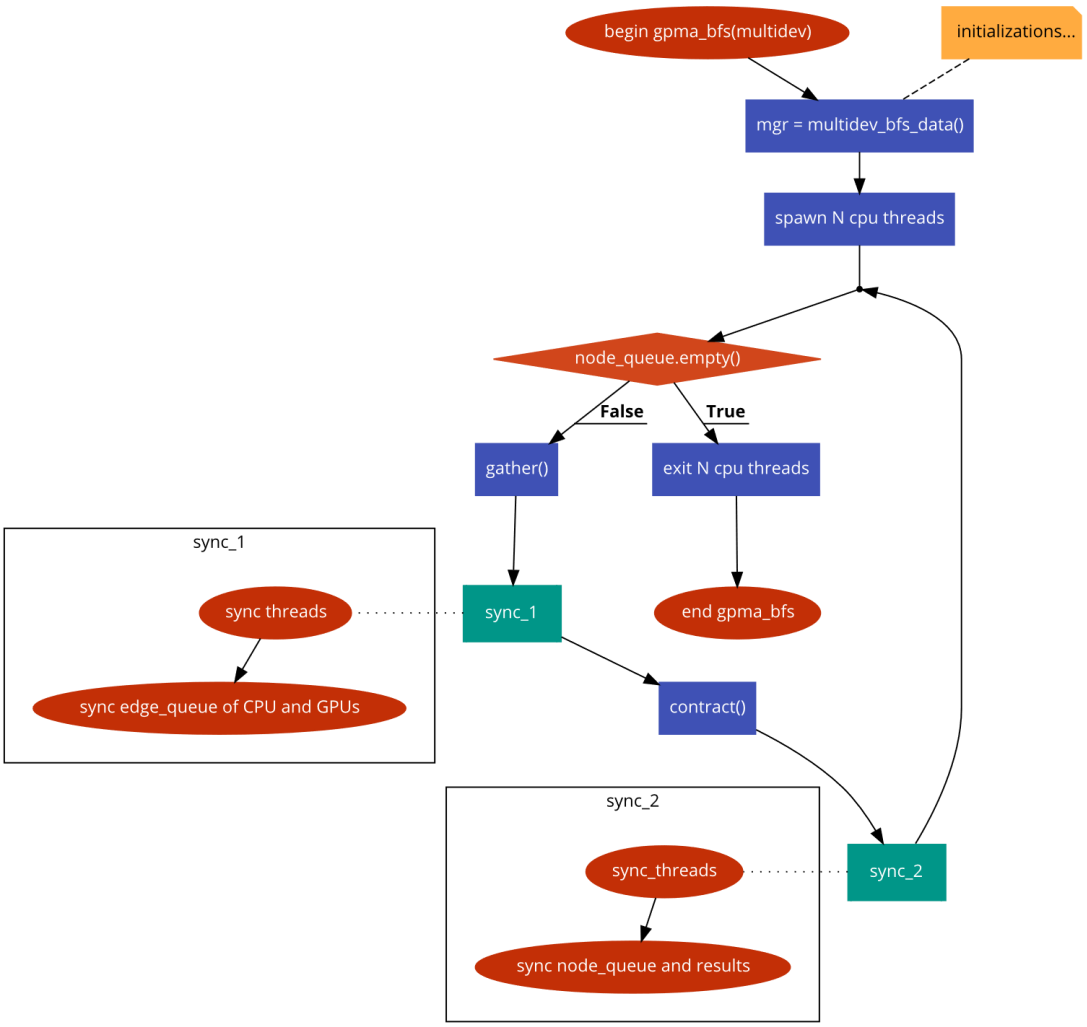


图 4.3 多设备 BFS 流程图

多设备的 BFS 主要是在，并行运行每一个单设备的 BFS 的基础上，加入必要的同步完成的。在单设备 BFS 过程中，主要有 gather 和 contract 过程。得益于原设计中良好的并行度，我们只需将 GPU 共享内存上的算法，改变为分布式共享内存上的算法即可。

在具体实现上，如图 4.3 所示。我们在每一个设备完成 gather 操作后，调用 multidev_bfs_data 的 iteration_barrier_1 方法，令调用线程等待其他线程操作完成。随后，在多设备间合并所需的数据结构，并在完成后继续进行后续任务。同样，在每一个设备完成 contract 操作后，iteration_barrier_2 方法再次进行同步，并合并 results 和中间数据结构，判断 BFS 过程是否完成。

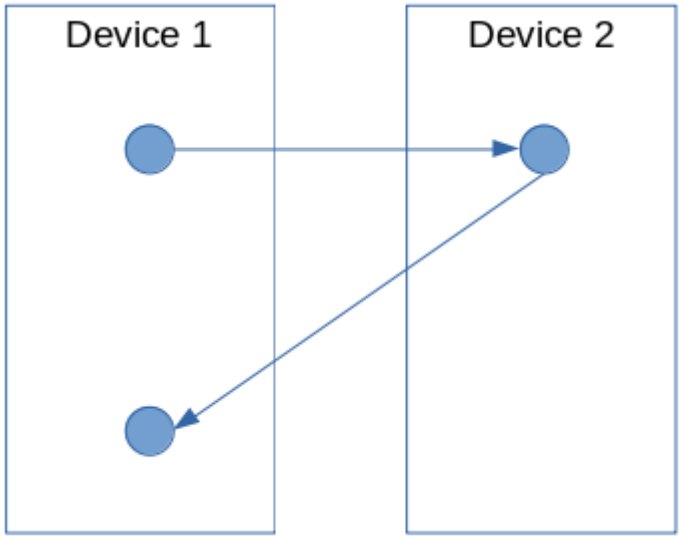


图 4.4 多设备 BFS 问题样例示意图

值得注意的是，如果 BFS 过程没有完成，即使某一个设备的子图上已经没有任务了，单个设备也不能提前结束自己的工作线程。如图 4.4 所示的情形中，如果 Device1 在完成第一次 BFS 后发现本设备上已经没有任务而退出。那么当设备 1 上的节点再次被设备 2 引用时，就会导致任务无人处理而出现错误。

4.4 本章小结

本章具体介绍了我们的多设备动态图更新系统的关键实现细节，以及它如何同时利用 CPU 和 GPU 的计算资源，进行图更新和图算法操作。同时，我们介绍了图更新和 BFS 过程中用到的具体串行与并行算法。

5 系统测试与分析

5.1 测试环境

我们首先对软件正确性进行了评估，测试平台 1 配置为: CPU 为 Intel Core i5-6500 4x 3.6GHz，8GiB DDR4 内存，Samsung 860 EVO M2 SSD，NVIDIA GeForce GTX 750，1GiB GDDR5 显存，操作系统使用安装了 CUDA 10.2 的 ArchLinux。测试使用 Pokec 社交网络数据作为测试集，这个数据集含有整个 Pokec 社交网络的关系图。我们使用其中的 soc-pokec-relationships.txt 文件，它存储了所有用户之间的朋友关系。例如其中的一行“3 5”，就表示用户 5 是用户 3 的一个好友。

随后我们对软件的性能进行评估。除了用到上面提到的测试平台 1 外，由于测试平台 1 的显存不足以运行最大的测试样例，我们会在下文的部分情况下使用服务器上的测试平台 2: CPU 为 Intel Xeon Platinum 8255C 8x 2.50GHz，32GiB 内存，NVIDIA Turing T4，16GiB GDDR6 显存，操作系统选用安装了 CUDA 10.2 的 Ubuntu 18.04 Server。

5.2 功能测试

在功能测试中，我们把测试集分为两部分。首先初始化 GPMA 数据结构，随后将前半部分数据集一次性用 GPMA+算法插入到图中。随后，我们运行一次 BFS 图算法，计算包含 0 号节点的子图的节点数，检验正确性。然后我们将数据集分为 N 个更小的集合，不断交替进行”删除一批前半部分数据集的边”和”插入一批后半部分数据集的边”的操作 N 次，直到后半部分数据集完全在图中代替前半部分数据集。此时，我们再进行一次从 0 节点开始的 BFS 算法，计算每一个节点的深度。如果结果正确，就可以使得软件正确性得到验证。

为了验证结果的正确性，我们准备了他人已有的正确工程作为 baseline。baseline 在 CPU 上运行，不涉及 CUDA，使用 PMA 数据结构来存储图数据。

我们在分别将设备数设置为 1GPU 和 1CPU，并在测试平台 2 上对完整的 P

okec 数据集进行以上所述的操作，结果如表 5.1 所示。这表明，系统中的算法的 CPU 实现和 GPU 实现均正确，而且多设备并行更新的功能也能正确运作。

表 5.1 测试平台 2 上的系统正确性测试

配置	Baseline	1CPU	1GPU	1GPU+1CPU
第一次 BFS 结果	1333870	1333870	1333870	1333870
第二次 BFS 结果	1334630	1334630	1334630	1334630
运行用时	53.28s	51.09s	8.9s	8.9s
CPU 利用率	接近 100%	接近 100%	接近 100%	接近 170%

同时，我们也在测试平台 1 上对 GTX 750 显卡运行了此测试，证明系统的运行结果也是正确的。由于测试结果类似，此处略去。

5.3 性能测试与性能分析

5.3.1 CPU 并行性测试

在完成对我们的代码正确性的测试后，我们先对图更新部分代码的并行度。表 5-2 是在测试平台 1 上进行的，设置不同数量的 GPMA<CPU>实例后，对 Pokec 测试集仅运行图更新任务的测试结果。需要注意的是，测试平台 1 上共有 4 个 CPU 核心，运行用时是指从程序启动到结束所流逝的墙上时间，CPU 利用率是指程序使用的 cpu user time 和墙上时间的比值。

表 5.2 对 4 核 CPU 的图更新测试

	运行用时	CPU 利用率
1 CPU	43.08s	111%
2 CPU	25.79s	192%
3 CPU	22.70s	261%
4 CPU	18.55s	308%
5 CPU	24.32s	251%
6 CPU	22.66s	270%
7 CPU	21.47s	287%
8 CPU	20.89s	305%

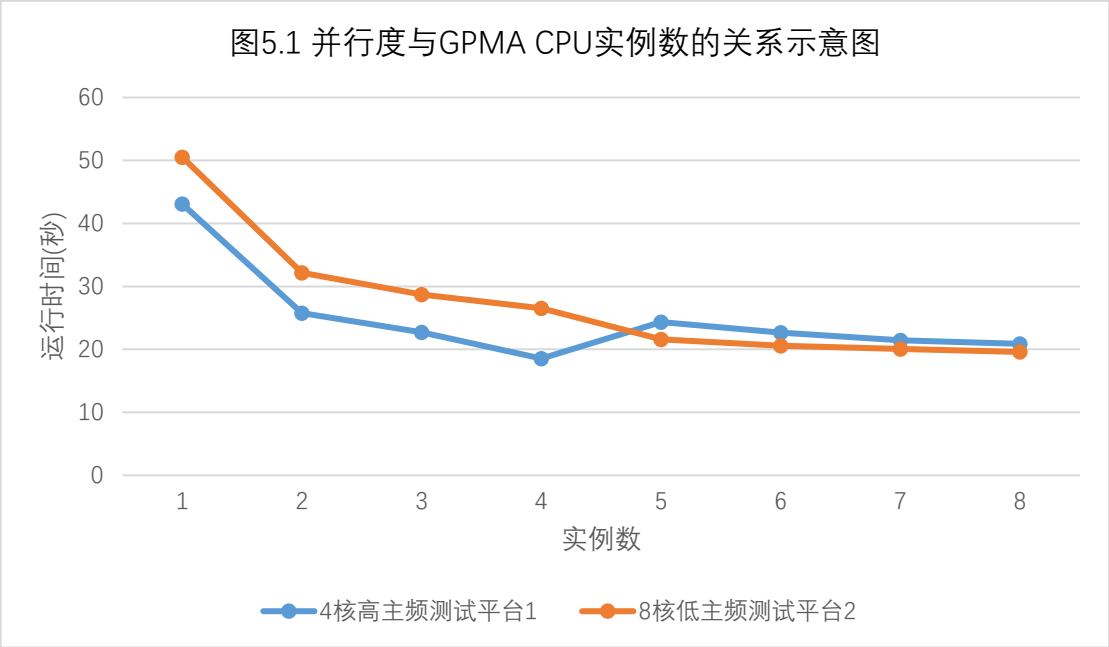
从表 5.2 的并行度测试结果中可以看出，即使只有一个 GPMA<CPU>实例，程序仍然有一定的并行度，可以达到 111%的 CPU 利用率。当程序中的 GPMA<CPU>实例数，逐渐增加到接近物理 CPU 核心数时，程序性能近似线性提高。当程序中的 GPMA 实例数超过物理核心数时，再增加程序中 GPMA 实例数就不再有性能提升了。

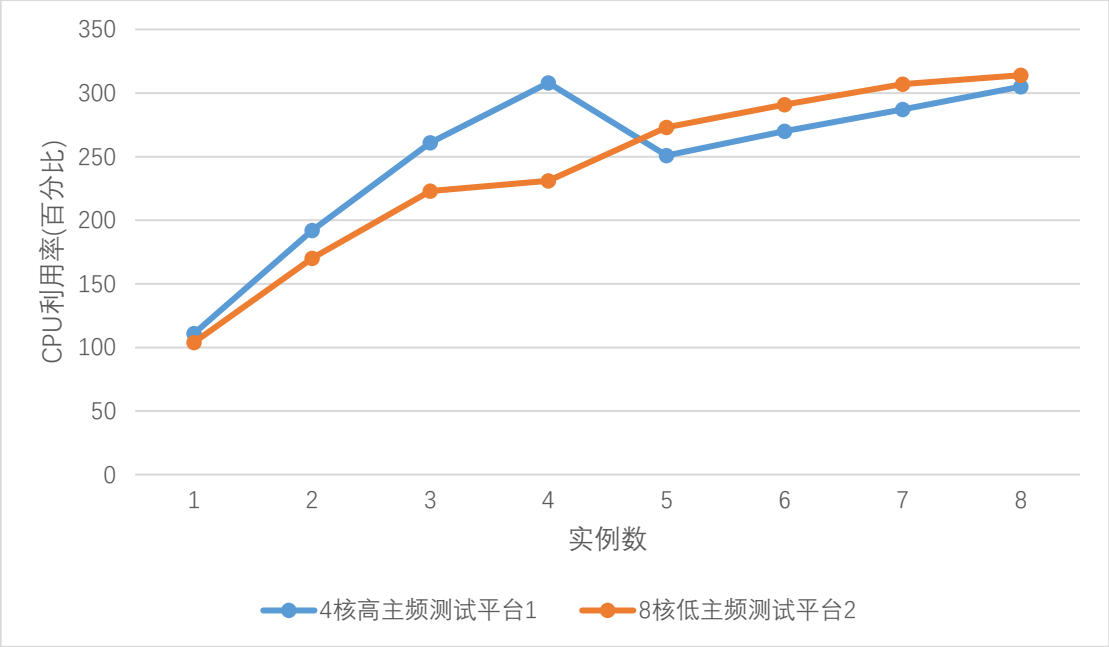
我们用相同的方式，对 CPU 核心数更高，CPU 主频更低的测试平台 2 进行了同样的并行度测试，得到的结果如表 5.3 所示，表现出了相似的规律。

表 5.3 对 8 核 CPU 的图更新测试

	运行用时	CPU 利用率
1 CPU	50.53s	104%
2 CPU	32.16s	170%
3 CPU	28.70s	223%
4 CPU	26.51s	231%
5 CPU	21.62s	273%
6 CPU	20.58s	291%
7 CPU	20.05s	307%
8 CPU	19.60s	314%

我们对表 5.2 和表 5.3 的结果进行绘图，从图 5.1 可以看出，当程序中的 CPU 实例数从 1 增加到物理核心数时，程序的性能能够以近似正比例的方式得到提高，系统的 CPU 利用率也线性增长。然而在 CPU 实例数超过物理核心数的情况下，则会造成额外的调度开销，造成了性能损失核 CPU 利用率下降，得不偿失。在我们的程序中，CPU 实例数与启用的 CPU 线程数意义基本相似，因为每一个 CPU 实例都在一个单独的线程中并行处理本分区数据。





5.3.2 多设备性能测试

由于我们的多 CPU 设计与分布式架构非常类似，不像传统并行应用那样，在程序的各个阶段使用基于多线程，共享内存，原子变量或锁同步的方法进行并行化；而是将整个 GPMA 数据结构分为多个分区，并令每一个线程对应一个 GPMA 实例，来在更高的层次上进行并行化和同步。因此出于内存同步的开销的考

虑，我们的做法可能不如传统做法性能好，但拥有更好的可扩展性：只需使用相同的模型，即可让 GPU，甚至其他主机也成为多设备的一部分。

由于测试平台 1 的显存不能完整的运行整个测试，因此从原 pokec 数据集中截取出 700M 显存所能处理的最大数据集(100000 个节点，176 万个边)。测试过程中采用 1 个 GPMA<GPU>实例，加上 3 个 GPMA<CPU>实例。为 GPU 分配不同的数据量，测试结果如图 5.2 所示。

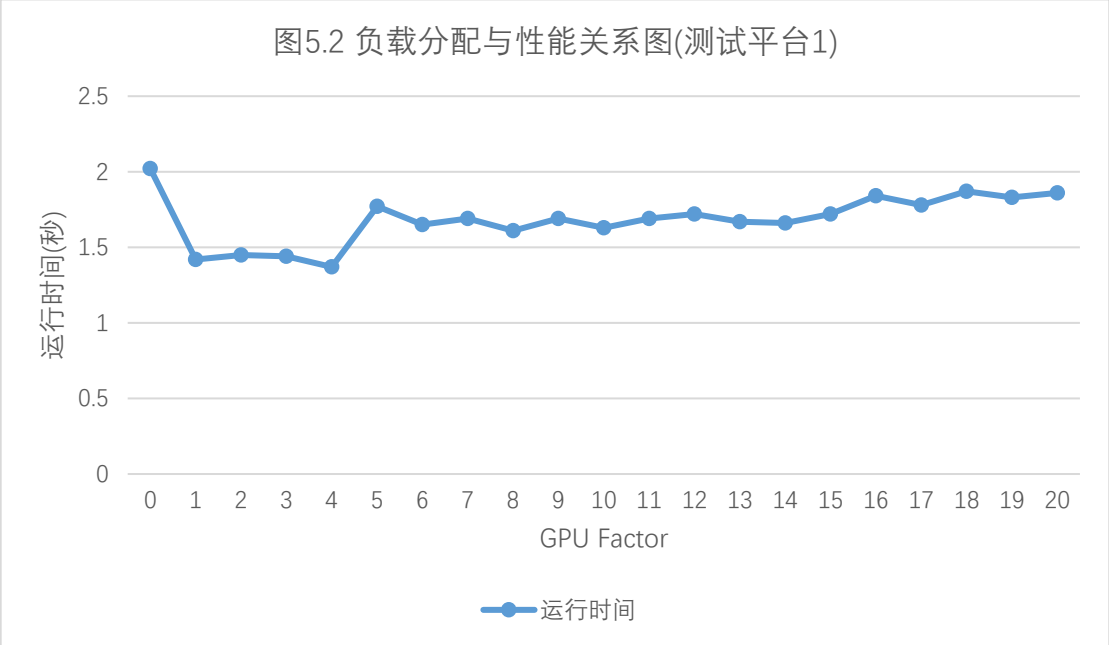
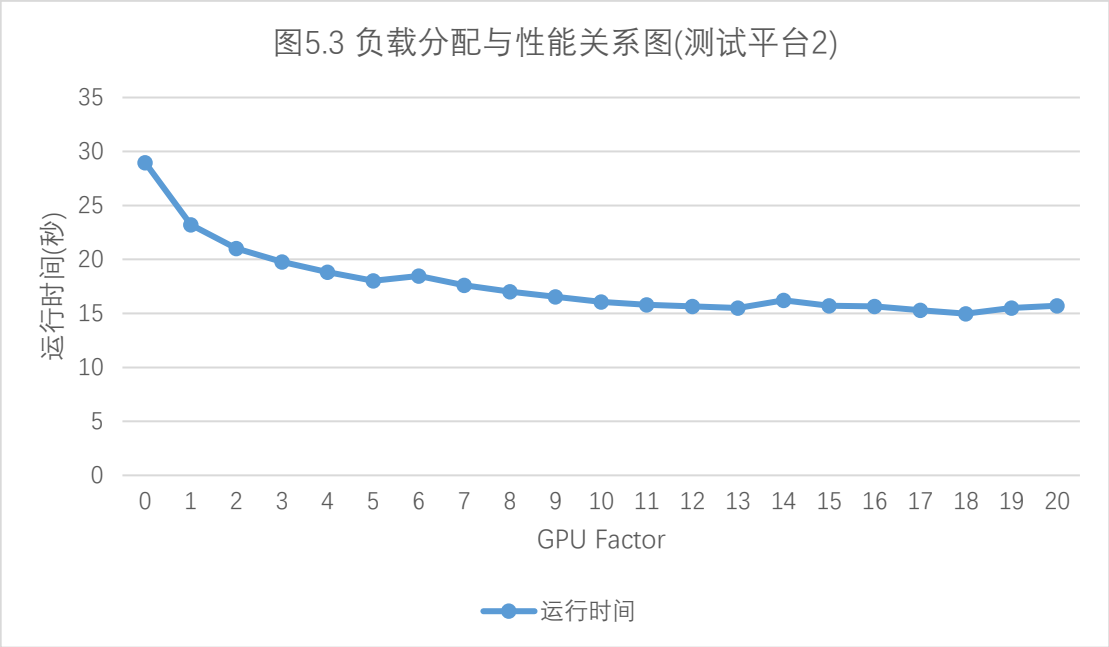


图 5.2 的测试结果表明，在这张 GTX 750 GPU 与 Intel i5-6500 CPU 的性能对比中，最佳的 GPU factor 参数为 4 左右。这意味着，应当为这张 GPU 分配相当于两个 CPU 核的任务，整个系统达到最佳状态。

同时，我们可以注意到，100000 个节点的小数据量测试中，CPU 与 GPU 对任务的不同分配比例造成的性能差距不大。因此我们在拥有更大显存的测试平台 1 上使用了完整的 Pokec 数据集进行了测试，其结果如图 5.3 所示。由此可以看出，Tesla T4 GPU 的计算能力相比 GTX 750 要强的多，最终测试平台 2 中最佳的 GPU factor 在 18 左右。



多性能设备测试的结果中，均存在一个最优的 GPU factor 值，使得 CPU 和 GPU 的任务分配达到最高效率。这表明，本文最终设计的动态图更新系统基本达到了让 CPU 和 GPU 共享任务负担的目的，成功的用多个混合类型的计算设备，来运行大规模动态图更新的计算任务；并且可以通过调节 GPU factor，来获得某种 GPU 型号搭配某种 CPU 型号的最佳任务分配比例。

5.4 本章小结

本章对动态图更新系统进行了完善的正确性和性能测试，用测试证明了程序在不同环境，不同配置下的正确性；同时，还对动态图更新系统的 CPU 部分代码的并行度进行了性能测试。同时，本章针对最终的设计目标，进行了多设备下的性能测试，展示了 GPU factor 的意义，证明本文设计的系统能够成功达到将负载分配在跨 CPU/GPU 的多个设备上的设计目标。

6 总结与展望

大规模动态图更新任务将随着大数据时代的到来而越来越重要。在本篇文章在现有的 GPMA+算法的基础上，提出了一种同时高效利用系统中的 CPU 与 GPU 资源进行动态图的更新的方法，达到了更高效的计算资源利用率，也提高了整个动态图更新系统的性能。在 CPU+GPU 混合多设备动态图处理任务中，主要需要解决三个方面的问题: 第一，如何为每种设备提供高性能的图更新算法实现; 第二，如何利用最小的内存同步开销，内存迁移开销和计算开销，在不同设备间高效的进行图的分区; 第三，如何为这种近似与分布式的系统设计算法。围绕这几个问题，本文做了如下几点研究和工作:

1) **设计实现了适合 CPU 的 GPMA 数据结构与 GPMA+算法。**我们比较了 CPU 与 GPU 作为计算设备的相同点与不同点，设计并实现了能高效在 CPU 上运行的 GPMA 数据结构与对应 GPMA+算法，单线程下与良好实现的 PMA 效率类似，为多设备 GPMA 工作打下了良好的基础。

2) **提出了将 GPMA 数据结构分布于多个设备上的一种方法。**我们考察了动态图更新任务的过程，提出了一种适合在更新操作较多，分析操作较少的动态图上进行 CPU+GPU 混合图更新的方法，经测试获得了良好的并行度。

3) **提出了在多设备 GPMA 上的负载分配与图分区的一种方法。**我们设计实现了一种高性能的多设备 GPMA 负载分配方法，成功达成了将任务负载平衡到包括 CPU 与 GPU 的混合设备中，以便同时充分利用 CPU 与 GPU 计算资源的目的。

4) **实现了可用于多设备 GPMA 数据结构的 BFS 图算法。**

本文为高性能的大规模动态图更新系统进行了有益的探索。该领域还有很多研究工作有待完成，包括:

- 为具有大量实时分析需求的大规模动态图进行针对性的优化。
- 进一步降低动态图分区造成的图更新与实时分析任务的额外性能开销。
- 利用启发式等更智能的策略进行图分区。

随着大数据时代中，数据量的不断增长，不同数据之间相互联系的维度也越

来越紧密，大规模的动态图更新与分析系统将会成为非常重要的互联网基础设施。近年来，有关大规模动态图系统的各种研究层出不穷。在大数据量的情形下，分布式和 GPU 由于他们的良好的并行性，一直是研究的热点。图系统相关技术必须不断发展，才能满足人们在大数据时代的数据存储与分析需求。

致 谢

论文完成之际，首先要感谢我的导师张宇教授。他站在学科发展的前沿，从论文的选题，研究工作逐步深入，到论文的撰写，都给我以细致的指导和建设性的意见，使我得以圆满而顺利地完成了 CPU+GPU 混合动态图处理系统。张宇教授严谨的治学态度、诲人不倦的师德和一丝不苟的工作作风将会给我留下不可磨灭的记忆。

我在华中科技大学的四年里，得到了许多老师和同学的大力帮助和支持，在此表示深深的谢意。感谢各位老师，同学，尤其是联创团队的学长对我的关心和帮助。我们大家共同创造的良好的学术氛围，对我的发展道路起到了极大的帮助作用。

最后，我要深深地感谢我的另一半，蒋伟玲，给予了我人生中前所未有的支持与美好。她陪伴我走过了大学的最美好的两个学期，陪伴我度过了无数个开发与测试的夜晚，陪伴我完成了整篇论文的撰写过程。无论多年以后我们结果如何，她的感情都会像一缕曙光，成为我终生的美好回忆。

参考文献

[1] S. Guha and A. McGregor. Graph synopses, sketches, and streams: A survey. PVLDB, 5(12):2030–2031, 2012.

[2] D. Sayce. 10 billions tweets, number of tweets per day. http://www.dsayce.com/social-media/ 10-billions-tweets/. Accessed: 2016-10-18.

[3] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, pages 5:1–5:6, 2016.

[4] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In SIGMOD, pages 251–264, 2015.

[5] A. McGregor. Graph stream algorithms: A survey. SIGMOD Rec., 43(1):9–20, 2014.

[6] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. ACM SIGMOD Record, 34(4):42–47, 2005.

[7] J. A. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. D. Liu, and W.-m. Hwu. Optimization and architecture effects on gpu computing workload performance. In InPar, pages 1–10, 2012.

[8] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. SIGPLAN Not., 50(8):265–266, 2015.

[9] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex- centric graph processing on GPUs. In Proceedings of the 23rd Inter- national Symposium on High-performance Parallel and Distributed Computing, HPDC ’14, pages 239–252, June 2014.

[10] Z. Fu, M. Personick, and B. Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In Proceedings of the Workshop on GRAph Data Management Experiences and Systems, GRADES ’14, pages 2:1–2:6, June 2014.

[11] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In Proceedings of the 18th ACM SIG-PLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, pages 135–146, Feb. 2013.

[12] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute," Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2017.

[13] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, "A study of partitioning policies for graph analytics on large-scale distributed platforms," Proceedings of the VLDB Endowment, vol. 12, no. 4, pp. 321–334, Jan. 2018.

[14] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In SC, pages 781–792, 2014.

[15] H. Liu, H. H. Huang, and Y. Hu. ibfs: Concurrent breadth-first search on gpus. In SIGMOD, pages 403–416, 2016.

[16] X. Lin, R. Zhang, Z. Wen, H. Wang, and J. Qi. Efficient subgraph matching using gpus. In ADC, pages 74–85, 2014.

[17] J. King, T. Gilray, R. M. Kirby, and M. Might. Dynamic sparse-matrix allocation on gpus. In ISC, pages 61–80, 2016.

[18] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. PVLDB, 4(4):231–242, 2011.

[19] W. Guo, Y. Li, M. Sha, and K.-L. Tan. Parallel personalized pagerank on dynamic graphs. PVLDB, 11(1), 2017.

[20] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on GPUs," Proceedings of the VLDB Endowment, vol. 11, no. 1, pp. 107–120, Jan. 2017.

[21] K. Vora, R. Gupta, and G. Xu, "Synergistic Analysis of Evolving Graphs," TACO, 2016.

[22] F. Sheng, Q. Cao, H. Cai, J. Yao, and C. Xie, "GraPU," Proceedings of the ACM Symposium on Cloud Computing, Nov. 2018.

[23] Garland M. Sparse Matrix Computations on Manycore GPU's. In: Proceedi

ngs of the 45th Annual Design Automation Conference. DAC ’08. New York, NY, USA: ACM; 2008. p. 2–6.

[24] Garland M, Kirk DB. Understanding Throughput-oriented Architectures. Co m- mun ACM. 2010 Nov;53(11):58–66.

[25] Bell N, Garland M. Efficient Sparse Matrix-Vector Multiplication on CUD A. NVIDIA Corporation; 2008. NVR-2008-004.

[26] Monakov A, Lokhmotov A, Avetisyan A. Automatically Tuning Sparse M atrix- Vector Multiplication for GPU Architectures. In: Patt Y, Foglia P, Dueste rwald E, Faraboschi P, Martorell X, editors. High Performance Embedded Archi tectures and Compilers. vol. 5952 of Lecture Notes in Computer Science. Sprin ger Berlin Heidelberg; 2010. p. 111–125.

[27] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme fo r Partitioning Irregular Graphs. SIAM J. Sci. Comput., 20(1):359–392, Dec. 19 98.

[28] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning Trillion-Edge Graphs in Minutes. In 2017 IEEE International Parallel and Distri buted Processing Symposium (IPDPS), pages 646–655, May 2017.

[29] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distribute d Graphs. In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[30] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-centric Distributed Graph Processing System. In Proceedings of the 12th USENIX Con ference on Operating Systems Design and Implementation, OSDI’16, pages 301 –316, Berkeley, CA, USA, 2016. USENIX Association