学校代号 10532　　学　　号 S12102011

分 类 号 TP302　　密　　级 普通

湖南大學

HUNAN UNIVERSITY

硕士学位论文

基于 GPU 的图计算研究

学位申请人姓名 肖军

培 养 单 位 信息科学与工程学院

导师姓名及职称 陈浩　教授

学 科 专 业 计算机科学与技术

研 究 方 向 并行计算

论文提交日期 2015 年 5 月 8 日

学校代号：10532

学　　号：S12102011

密　　级：普通

湖南大学硕士学位论文

基于 GPU 的图计算研究

学位申请人姓名：肖军

导师姓名及职称：陈浩　教授

培　养　单　位：信息科学与工程学院

专　业　名　称：计算机科学与技术

论文提交日期：2015 年　5 月　8　日

论文答辩日期：2015　年　5　月　26 日

答辩委员会主席：王东　教授

Research of Graph Computation Based on GPU

by

XIAO Jun

B.E. (ShangRao Normal University) 2012

A thesis submitted in partial satisfaction of the

Requirements for the degree of

Master of Engineering

in

Computer Science and Technology

in the

Graduate School

of

Hunan University

Supervisor

Professor CHEN Hao

May，2015

湖 南 大 学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：　　日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

1、保密□，在_____年解密后适用本授权书。

2、不保密□。

（请在以上相应方框内打“√”）

作者签名：　　日期：　　年　　月　　日

导师签名：　　日期：　　年　　月　　日

摘 要

图算法一直是学术界和工业界的研究热点。随着社交网络和大数据爆炸式增长，基于大图数据的应用逐渐增多。Google 提出了 Pregel 图计算系统，解决关于大图数据的分布式计算问题。Pregel 编程模型为用户抽象分布式计算的实现细节，提供了简明的 API 实现图算法。随着 GPU 软硬件技术的发展，与 CPU 相比，GPU 逐渐演变成为高度并行的多线程及多核处理器，同时具有强大的计算能力和超高的内存带宽。越来越多的图算法使用 GPU 来提升计算性能，并利用 GPU 的硬件特性优化图算法。

但是目前大部分基于 GPU 的图计算研究都是针对特定的图算法，缺乏基于 GPU 的类 Pregel 系统，为图算法提供基于 GPU 的通用图计算系统。而且当前的系统尚未根据图算法特点和 GPU 硬件特性，对系统性能进行优化。因此，本文主要对基于 GPU 的图计算进行研究，以实现对基于 GPU 的 Pregel 编程模型进行改进，分析图计算出现的性能瓶颈问题，提出相关的系统级优化算法。基于以上研究成果，本文实现了基于 GPU 的图计算系统 PregelGPU。论文的主要工作如下：

首先，讨论 GPU 体系结构的基础知识，归纳图计算的研究进展。着重阐述图计算系统的编程模型和执行流程，并分析相关系统的不足。

其次，GPU 线程具有细粒度并行的特点，改进传统的顶点编程模型，提出基于 GPU 的通用图编程模型—Edge-Vertex 编程模型，为图算法提供类似 Pregel 的 API。同时根据 Edge-Vertex 编程模型，设计图系统的执行流程，把整个系统执行阶段划分成预处理阶段、EdgeCompute 和 VertexCompute，并阐述各个阶段的执行任务和逻辑操作。

最后，分析 Edge-Vertex 编程模型的性能瓶颈，提出相应的解决方案。由于基于 GPU 的图计算存在线程负载不均衡问题，提出基于 GPU 的近似排序算法，缓解线程任务分配不均衡的压力。根据 GPU 合并访问全局内存的特性，提出应用于消息缓存的数据重映射算法，减少非合并访存次数，提高全局内存的实际带宽利用率。实验显示在不同的数据集和图算法下，和其他基于 GPU 的图计算系统相比，本文实现的图计算系统可以达到 1.6x—4.5x 倍的加速比。

关键词：图算法；Pregel；GPU；并行计算；负载均衡；合并访问

Abstract

Graph algorithms have always been the hot issue in academia and industry. With the explosive growth of social networks and big data in recent years, the applications based on big graph are increasing. Pregel was proposed by Google as a programming model to address distributed computing on big graph. Pregel programming model plays a significant role in abstracting architectural details of distributed computing from users and offers the simple API to express graph algorithm. With the development of the GPU hardware and software technology, GPUs have become a powerful platform for parallel computing. GPU has evolved into a highly paralleled, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth in comparison with CPU. GPU are increasingly leveraged to accelerate graph algorithms with GPU-specific optimization.

However most of graph computing research on GPU is aimed at a specific graph algorithm so far. It is lack of Pregel system based on GPU that provides the general programming model on GPU for graph algorithms. In addition, state-of-the-art systems have not yet been used to optimization according to the characteristics of graph algorithms and the GPU architecture. This paper mainly studies graph computation on GPU, to improve the Pregel programming model on GPU and propose system level optimization algorithms by analyzing the performance bottleneck. At last, based on these studies, this paper implements a graph computation system optimized for GPU, called PregelGPU. The contributions of this paper are as follows:

First, we discuss some basic background on GPU architecture, and research progress on graph computation. We focus on the programming model and execution flow of graph computing system, and analyze the weakness of related systems.

Second, in order to exploit the fine-grained parallelism on GPU, we improve the traditional vertex-centric programming model and present a general graph programming model called Edge-Vertex model, which provides Pregel-like API for developing graph algorithms. We design the execution flow of the graph system ,and partition the overall executing procedure into preprocessing, EdgeCompute and VertexCompute on the basis of Edge-Vertex model. At the same time, we elaborate

the execution task and logical operation of each execution phase.

Third, we analyze the performance bottlenecks of Edge-Vertex model and put forward the corresponding solutions. Because load imbalance among GPU threads in existing graph processing systems has important implications on performance, we introduce an approach called approximate sort to address this issue. In addition, according to the feature of coalesced memory access, we propose a data remap algorithm on the message buffer to mitigate non-coalesced GPU memory access and enhance the actual memory bandwidth. Our experiments show that the customized system implemented in this paper could achieve 1.6x to 4.5x speedup compared with state-of-the-art graph processing framework designed for GPUs with different workloads and graph algorithms.

Key Words: Graph Algorithm; Pregel; GPU; Parallel Computing; Load Balancing; Coalesced Memory Access

目 录

插图索引

插表索引

第 1 章 绪论

1.1 课题背景与意义

随着移动互联网和大数据快速地发展，基于图数据的应用也爆炸性增长。比如，在 web 链接图中计算 PageRank[1,2]，在百度地图计算两点间最短距离等。与传统图算法处理的数据量相比，这些图数据都是海量的，由几千万个顶点和几亿条边组成，在单机环境无法高效地处理如此庞大的数据集。MapReduce[3]系统是大数据的先驱，研究人员在 MapReduce 上处理图数据也取得相应的进展，比如 HaLoop[4]系统和 Pegasus[5]系统。但是 MapReduce 不适合处理关联数据集，而图数据是关联数据集的典型。如果使用 MapReduce 框架处理图数据通常需要创建一系列 mapreduce 作业，而且每次作业都需要消耗大量的网络资源，这样就导致系统性能急剧下降[6]。

Google 研究人员在 2010 年提出 Pregel[7]图计算系统，用来解决关于大图数据的分布式计算问题。和 MapReduce 类似，Pregel 主要包括两部分：高层次抽象的编程模型和负责分布式计算的运行时系统。在 Pregel 编程模型中，图算法由一系列的迭代构成。在每轮迭代过程中，顶点接受来自前一轮迭代的消息，发送消息给其他的顶点，修改顶点的状态，并且可以改变图的拓扑结构。这种编程模型可以灵活且简洁地编写图算法。Pregel 提供简单的可编程 API，把分布式相关的实现细节隐藏在运行时系统中，简化了分布式图计算的编程工作。更重要的是，Pregel 编程模型比较容易在 PC 组成的集群上实现一个高效、可扩展、容错的分布式图计算系统，并且暗含了同步原语。随后，研究人员改进 Pregel 系统，从多方面进行优化和改进系统性能，分别诞生了 GPS[8]、Kineograph[9]、Trinity[10]、Apache Hama[11]等图计算系统。

与此同时，GPU（Graphics Processing Units，GPU）已经演化成为一种用于大规模并行计算的协处理器，提供较 CPU 更高的计算峰值和超强的内存带宽[12]。例如，NVIDIAs GeForce GTX 780 型号的 GPU，包含了多达 192 个标量处理核心(Scalar Processing Cores, SPs)。这些核心被划分为 12 个流式多核处理单元(Streaming Multiprocessors, SMs)，每个 SM 又包含了 16 个 SPs。另外还有一个被所有核心共享的全局内存，容量高达 3GB。如此强大的硬件配置，使得 NVIDIAs GeForce GTX780 的并行计算能力非常出众。

为了充分利用 GPU 的并行计算能力，NVIDIA 推出了 CUDA（Compute

Unified Device Architecture，统一计算设备架构）编程模型[13]。基于该编程模型，研究人员可以使用 C 语言实现某些通用的并行计算程序，极大地降低了并行计算实现的难度[14]。另外，强劲的运算能力，原生的原子操作支持，以及共享式的内存单元，使得 GPU 在并行计算研究方面具有先天的优势。在此基础上，各种基于 GPU 的高性能算法研究，在近年来不断地呈现出来。现在 GPU 已经发展到了颇为成熟的阶段，可轻松地执行应用程序并且其运行速度已远远超过了多核 CPU 的运算速度[15]。未来计算架构将是众核 GPU 与多核 CPU 串联运行的混合型系统。

尽管如此，在 GPU 并行环境下编写高性能的图算法依然充满挑战。由于 GPU 硬件特性和 CPU 完全不同，GPU 性能优化方式和 CPU 也截然不同。目前缺乏一种基于 GPU 的类 Pregel 系统，来为图算法提供一种基于 GPU 的通用编程模型。同时当前的系统尚未根据图算法特点和 GPU 硬件特性，对系统性能进行优化。

基于此背景，本文对基于 GPU 的图计算进行详细的研究，提出并实现了一种基于 GPU 的图计算系统——PregelGPU，以实现将 Pregel 编程模型融合在 GPU 并行平台上，为用户提供一种高效且简洁的 GPU 图计算解决方案。

1.2 GPU 体系结构

本小节讨论 NVIDIA GPU 体系结构和 CUDA 并行编程模型。正如名字隐含的含义，GPU 原本是图形应用程序的加速器，主要使用 OpenGL[16]和 DirectX[17]编程接口。由于图形处理固有的并行性，GPU 一直定位为大规模并行的处理器。虽然 GPU 一开始仅仅是一种功能单一的设备，但是目前已经演变为强大的并行处理器。

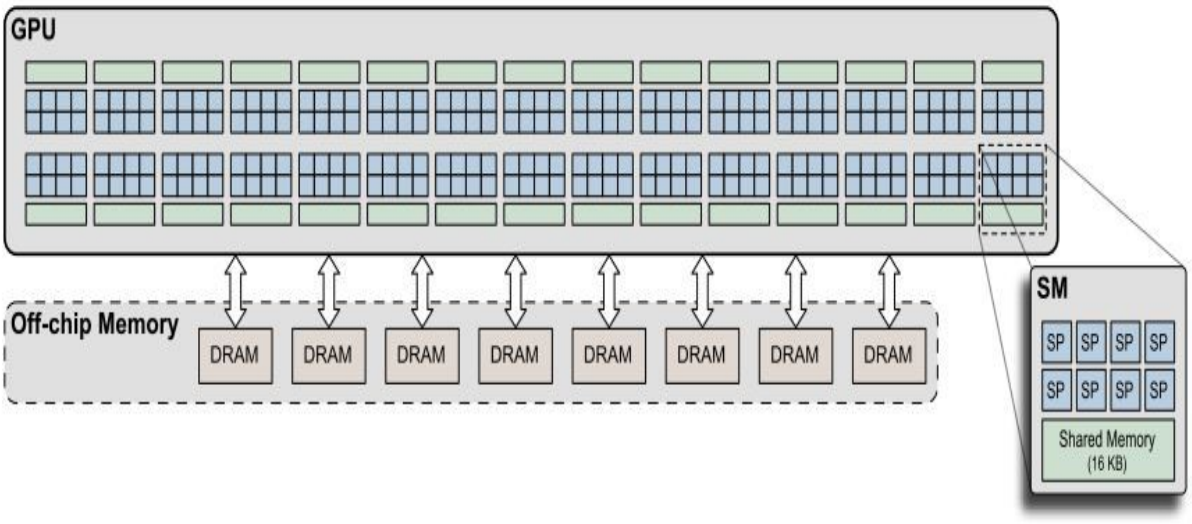


图 1.1 GeForce GTX 280 GPU硬件结构

GeForce GTX 280 是 NVIDIA 系列的 GPU。如图 1.1 所示，它配置了可编程的多核芯片，这些多核芯片组成并行处理器阵列。GPU 包含了一系列流处理器，每一个流处理器支持最高达 1024 个并发线程。NVIDIA 的现有产品线覆盖了低中高档领域，既有 1 个流处理器的低端产品，也有 30 个流处理器的高端产品。在图 1.1 中，GeForce GTX 280 GPU 的每个流处理器（SM）包含 8 个标量处理核心（SP），1024 个 32 位的寄存器[18]。并且，每个流处理器都配置了 16KB 可编程的 shared memory。和 CPU cache 类似，shared memory 也具有低延迟和高带宽的特性[19]。

流处理器通过硬件管理 GPU 线程，包括创建、调度和同步，所以线程管理工作几乎是零开销。为了更好地管理这些线程，流处理器使用单指令多线程（Single Instruction，Multiple Thread）体系结构。GPU 是以 warp 为单位进行调度，而 32 个连续线程组成了一个 warp。处于同一个 warp 内的线程执行在同一个流处理器上，即这些线程执行在同一个多线程指令单元上。流处理器可以处理 warp 内线程的条件分支，但是当执行无条件分支代码时，单指令多线程体系结构使得 GPU 获得更大的性能提升。

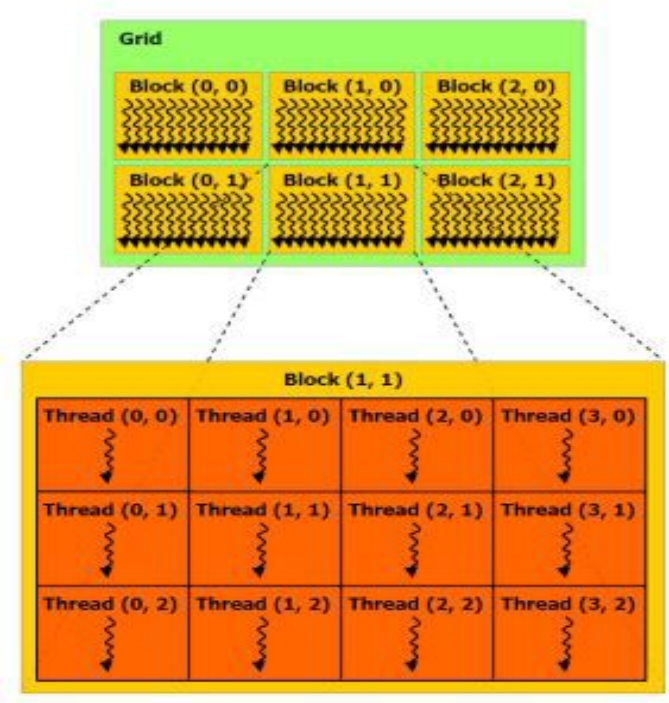


图 1.2 CUDA编程模型

CUDA 是一个完整的 GPU 通用计算解决方案，通过 GPU 来解决科学、工业以及商业方面的计算密集型问题。采用 C 语言作为编程语言，提供硬件的直接访问接口，不依赖传统的图形接口间接地使用 GPU 资源，使得用户可以方便地开发 GPU 通用计算程序，为计算密集型的应用提供一种高效的解决方案[20]。

在 CUDA 编程模型中，应用程序分成两部分：串行执行的 host 程序和执行在并行处理器的 kernel 程序。一般来说，host 部分执行在 CPU 上，kernel 部分执行在 GPU 环境上，尽管 CUDA kernel 可以编译并执行在多核 CPU 上。

如图 1.2 所示，CUDA 编程模型把线程逻辑地组织成线程块（block），一个 kernel 由一个 Grid 组成，而一个 Grid 由一个或者多个线程块组成[21]。一个线程块包含了一组并发的线程，同一个线程块的线程可以进行障碍同步（barrier synchronization），并且共享 shared memory 的内存空间。当调用一个 kernel 时，用户只需指定线程块的数目和块内线程的数目这两个参数。实际上，线程块虚拟化了 GPU 的流处理器，可以把每一个线程块看做是虚拟流处理器。

1.2.1 GPU 线程调度

本小节详细阐述 CUDA 调度模型。如图 1.3 所示，线程（thread）逻辑上划分成线程块（thread block），每个线程块都要分配给指定的流处理器（multiprocessor）。流处理器可以接受多个线程块，而线程块中的线程，又以 warp 为单位，把线程来做分组计算。流处理器通过连续的方式自动完成 warp 分组的功能。比如，若有一个线程块里有 128 个线程，则会被分成四组 warp，第 0-31 个线程是 warp 0、32-63 是 warp 1、64-95 是 warp 2、96-127 是 warp 3。如果线程块里的线程数量不是 32 的倍数，那么把不足 32 的线程独立成一个 warp。比如说线程数目是 66 的话，就会有三个 warp：0-31、32-63、64-65。由于最后一个 warp 里只剩下两个线程，所以在计算时，就相当于浪费了 30 个线程的计算能力，这是在设置块内线程数目时必须注意的问题。

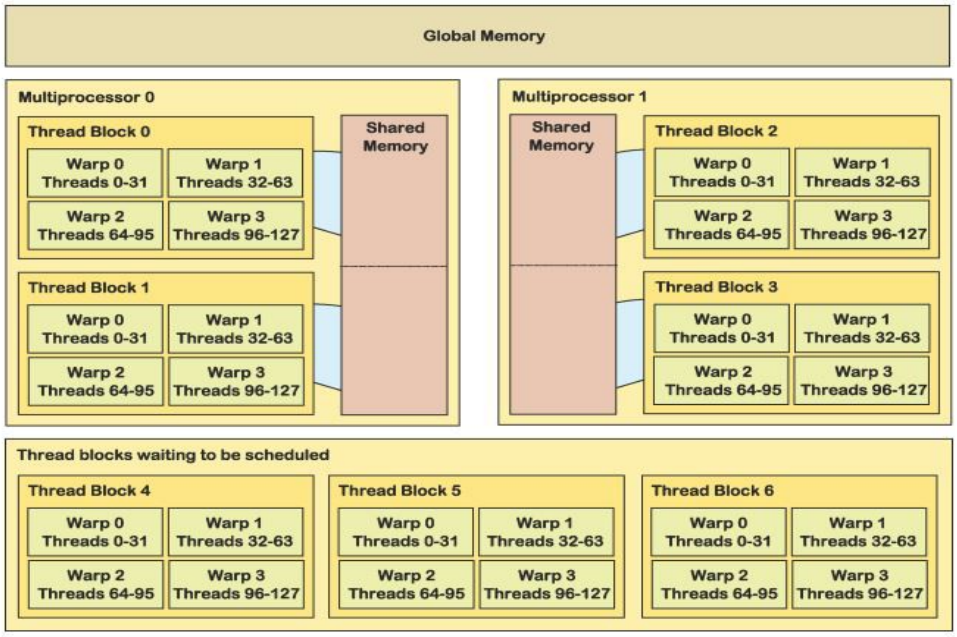


图 1.3 CUDA线程调度模型

CUDA 线程调度是由线程所属的 warp 确定的。warp 是 GPU 调度的基本单位（一个 warp 包含 32 个线程），同一个 warp 的线程并行地执行相同的指令，不同 warp 的线程有可能是串行执行。一个流处理器一次只会执行一个线程块里的一个 warp，但是流处理器并不一定一次性就把这个 warp 的所有指令都执行完。当遇到正在执行的 warp 需要等待的时候（例如线程访问全局内存需要等一段时间），就切换到别的 warp，避免由于等待而产生资源浪费。所以理论上效率最好的状况，就是在流处理器中有足够多的 warp 可以切换，在执行的时候，不会发生所有 warp 都进入等待的情形。CUDA 正是通过 warp 的切换，来隐藏线程的延迟、等待，达到线程最大并行化的目的。图 1.3 阐述了 GPU 线程的组织和调度方式，通过将所有线程组织成 warp 来进行调度，极大地简化 GPU 硬件设计[22]。

1.2.2 GPU 性能优化方式

CUDA 虽然为 GPU 提供了软件平台，但是要想充分利用 GPU 并行计算资源，必须从 GPU 硬件特性上分析，优化算法性能。目前有很多种性能优化方法，分别是：

1、尽量保证相同 warp 的线程执行相同指令：在 CUDA 编程模型中，任何控制流语句（if，switch，do，for，which）都会导致相同 warp 的线程执行分支指令（即执行不同的分支），降低指令吞吐量。如果相同 warp 的线程执行分支指令，那么不同的分支是串行执行，因为相同 warp 的线程共享程序计数器。这样就隐式增加执行指令的数目，当不同的分支都执行完成，线程收敛到相同的执行路径。为了优化性能，kernel 程序应该尽可能地避免条件分支，即保证相同 warp 的线程执行相同的指令。如果分支指令在不同 warp 的线程执行，则对性能没有影响。同时保证同一个 warp 内的线程运行时间大体相同，即相同 warp 内的线程任务负载大体均衡，也是 GPU 性能优化的方式。

2、shared memory 访问优化：shared memory 位于 GPU 片内，容量比较小，但是速度比 global memory 快很多。在不发生 bank conflict 的情况下，shared memory 的延迟几乎只有 global memory 的 1/100，访问速度与寄存器相当。位于相同线程块的线程共享 shared memory，所以可以利用 shared memory 高效地完成线程同步和互斥。或者是线程块将共享数据加载进入 shared memory，然后线程分别处理共享数据，最后将结果从 shared memory 输出到全局内存。

3、全局内存的合并访问：对全局内存的访问是否满足合并访问条件是对 CUDA 程序性能影响最明显的因素之一[23]。当来自相同 warp 的 32 个线程对全局内存进行读写访问时，如果这些线程访问连续的地址空间，GPU 将这些访存

请求合并成一次访存传输。例如，同一个 warp 内的 32 个线程同时访问不连续的内存地址单元，将生成 32 个单独的内存访问，即非合并访存（uncoalesced memory access）。如图 1.4 所示，同一个 warp 的 32 个线程访问连续的内存地址单元，32 个内存访问将会合并成 1 个内存访问，即合并访存（coalesced memory access）。

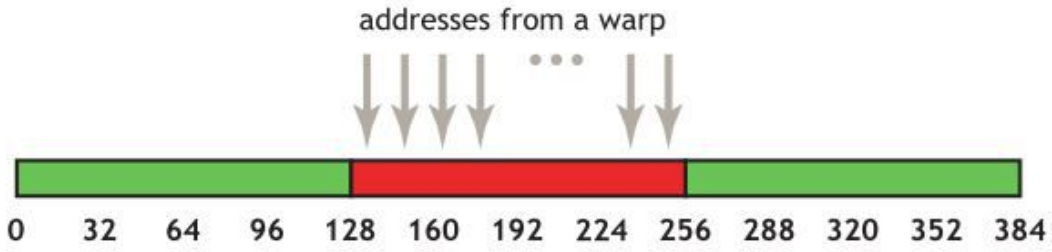


图 1.4 全局内存的访问合并

GPU 线程可以自由地加载（load）和存储（store）任何合法的地址，同时 GPU 支持聚集和分散访存。当来自相同 warp 的线程访问内存连续的字，GPU 硬件可以合并这些内存访问变成一个聚集的事务访问，这样就可以获得更高的内存带宽（memory throughput）。和非合并访存相比，合并访存能够充分利用 GPU 的高带宽。而有些程序的效率主要受内存带宽的限制，GPU 的合并内存访问模式将极大地提高这类程序的性能。

4、设置 GPU 线程数：GPU 依靠多线程来隐藏内存访问的延迟。因此设计基于 GPU 的算法，必须要创建足够多的并行任务，充分利用 GPU 强大的计算能力和内存带宽。一般情况下，现代的 GPU 至少同时发射 5,000—10,000 的线程数目，才能高效率地利用 GPU 的硬件资源。

1.3 相关研究现状

本小节从并行体系结构方面归纳图计算的相关研究进展，主要从分布式图计算、多核图计算和 GPU 图计算这三方面阐述。

1.3.1 分布式图计算

随着因特网的兴起，基于图数据应用大幅增加，比如 PageRank 算法等。学术界也在扩展 MapReduce 系统处理图数据的功能。但是研究人员发现，利用 MapReduce 系统不适合处理关联数据集，而图数据是典型的相关数据集。

针对这种情况，google 研究人员 G. Malewicz 和 M. Austern 等在 2010 提出了处理分布式图数据的 Pregel 系统。Pregel 是 PC 集群环境的分布式计算系统，和 MapReduce 相似，Pregel 同样具有自动容错能力，并提供了简单的 API。Pregel

采用简单的哈希算法用来分配节点的计算任务，简化了系统的设计，但缺乏灵活性，容易导致节点任务不均衡问题。

针对 Pregel 的不足，斯坦福大学 Semih Salihoglu 和 Jennifer Widom 提出了 GPS 分布式图计算系统，并对 Pregel 进行了三大改进：首先，GPS 是基于 Hadoop HDFS[24]分布式文件系统，而且将 Pregel 单一的以“顶点为中心”的编程模型扩展为灵活的以“多个顶点为中心”的编程模型，并加强了全局计算能力。其次，GPS 将 Pregel 的哈希任务分配算法，改为在计算过程动态分配节点的任务，以便减少网络通信开销。最后，GPS 提出了 LALP 算法（large adjacency list partitioning），以进一步减少通信量。这一系列的优化措施，使得 GPS 在离线图计算上，拥有优异的性能表现。但是，GPS 没有解决 Pregel 最大的问题，不能对图数据进行实时更新，即没有扩展 Pregel 实时图更新能力。

微软亚洲研究院的 R.Chen 和 J.Hong 等在图数据实时计算的问题上提出了 Kineograph 系统，对 Pregel 系统进行了进一步的功能性完善，提出了时间戳提交协议来解决实时更新问题。该协议的原理是，当更新数据时候，并不立即更新，而是通过缓存机制，统一到某个时间点进行更新。这种机制创造性将图计算和图更新这两个过程分离，使得更新和计算过程可以同时进行。

另外，微软研究人员 B.Shao 和 H.Wang 等提出的分布式图数据库 Trinity，创造性地引入了内存云（Memory Cloud）的概念。由于图数据具有无规律性，而图的计算和更新涉及大量的数据访问，这些访问大部分属于随机访问。尽管硬盘的技术发展迅速，但是依然不能为图计算提供高效的随机访问。所以在上述提及的分布式图系统，都是将图数据放入各节点计算机的内存中。Trinity 系统将各节点计算机的内存统一编址，并将这些内存进一步分块。Trinity 系统通过分配内存块和释放内存块，增加了对节点计算机的内存集中管理的相关机制。通过内存云机制，Trinity 系统实现了图计算和图实时更新的功能。

现有的分布式图计算系统都是将图数据进行划分，然后将划分的数据分配给相应节点，这个划分任务的过程即为预处理过程。通过这种静态地划分任务，可以平衡节点间的计算量。但是仅仅通过静态划分图数据，不能达到节点任务均衡的目的。因为在静态划分时，并不了解图数据的结构和相关算法的执行逻辑。关于节点间的任务均衡问题，Z. Khayyat 和 K. Wara 等提出基于动态任务均衡的图计算系统 Mizan[25]。和传统的 Pregel 系统相比，Mizan 系统增加了监控和动态迁移机制。首先通过监控节点的相关参数，判断是否需要迁移；如果需要，在每次迭代的末尾生成数据迁移计划，并实施数据迁移。这样，Mizan 系统比基于静态划分任务的图计算系统，提高了 84%的性能。

1.3.2 多核图计算

基于单机环境，研究人员也实现了很多图算法库，比如 JDSL[26]，LEDA[27]和 FGL[28]等。但是这些算法库无法利用多核与多处理器硬件特性，同时也无法对图数据进行扩展。

A. yrola 和 G. Blelloch 提出了基于单机环境的图计算系统 GraphChi[29]。创造性利用并行滑动窗口算法解决 PC 系统处理图数据问题。同时把图数据进行拆分处理，使得 GraphChi 系统成为一种基于磁盘的图计算系统，不需要大量内存就可以轻松地离线计算大图数据。为了充分利用多核资源，W. Han 和 S. L 提出了基于多核环境的图计算系统 TurboGraph[30]。TurboGraph 是完全意义下的并行图计算系统，体现在以下两个方面：（1）完全并行（full parallelism），包括多核并行和 I/O 并行；（2）尽可能把 CPU 处理和 I/O 处理重合。同时 TurboGraph 提出一种 pin-and-slide 并行执行模式，通过大量的优化，相比于 GraphChi，其性能了提高 4 倍多。

微软研究人员 V. Prabhakaran 和 M.Hu 等提出了 Grace[31]系统。Grace 系统把 Pregel 编程模型应用在多核及多处理器环境，并对图计算进行多方面的优化。首先，Grace 提供了多种图划分算法库，以将大图划分成小图并分配给每个线程；其次，Grace 优化了小图在内存中的存储问题，将顶点的邻接顶点存储在相邻的内存位置，进一步减少了通信开销；最后针对多核 Cache 结构，Grace 设计了相应的内存数据结构，并满足更新的 ACID 原语，保证了数据更新的一致性。另外，Grace 系统同时满足离线图计算和图实时更新两大功能。

1.3.3 GPU 图计算

由于 GPU 拥有强大的并行计算能力和超强的内存宽带等诸多优势，研究人员将众多 CPU 的算法和计算框架应用到 GPU 并行环境，取得了不俗的加速比。比如，把 MapReduce 框架运用 GPU 环境，根据 GPU 的硬件特性，提出相应的类 MapReduce 系统[32~34]。这些类 MapReduce 系统能够高效地处理数据，并提供了简洁的 MapReduce API，简化了 GPU 的编程难度[35]。

目前，基于 GPU 并行环境的图算法，也取得了相应的进展。P. Harish 和 P. J. Narayanan 首先提出利用 GPU 加速图计算过程[36]。G. J. Katz 和 J. T. Kider 提出了基于 GPU 的最短路径算法[37]。L. Luo 和 M. Wong 等提出了基于 GPU 的广度优先算法[38]；D. Merroll 和 M. Garland 等提出了基于 GPU 的可扩展的图遍历算法[39]；S. Hong 和 S. K. Kim 等提出，利用最大 warp 来提高图算法性能[40]。以上研究成果说明，可以在 GPU 环境实现图算法并取得相应的加速比。但是，这些

研究成果都有很大的不足，都跟 GPU 硬件特性相关，编写这些图算法，要求编程人员了解 GPU 硬件特性，而 CPU 与 GPU 编程有较大的区别，增加了编程人员的学习曲线。同时，现在大部分 GPU 图计算都是针对特定图算法，都不能提供通用的图编程模型。

受到 Pregel 编程模型的启发，J. Zhang 和 B. He 将 Pregel 编程模型运用于 GPU 环境，提出了 Medusa[41]系统，并针对 GPU 硬件环境进行相应的优化。首先，针对 GPU 线程具有细粒度并行的特点，提出了 EMV 模型来代替顶点模型，即将顶点模型进一步细划分为“边”、“消息”、“顶点”三个过程，利用 EMV 模型提供的 API，屏蔽 GPU 硬件细节。其次，针对 GPU 具有合并内存访问的特性，设计了特定的图数据结构。最后，根据图数据结构的特点，设计了消息缓存机制，并利用多跳冗余算法解决多个 GPU 进行图计算的消息通信问题。

Medusa 系统在 GPU 并行环境下实现了类 Pregel 编程模型，能够为用户提供统一的可编程 API，并屏蔽了相应的 GPU 硬件细节，提供了可对比的执行性能，降低了图算法的编程门槛。但是，Medusa 的 EMV 模型比 Pregel 的顶点模型复杂，比较难理解，可以考虑进一步精简。其次，将顶点模型划分为“边”、“消息”和“顶点”三个过程，但是“顶点”过程仍然存在负载不均衡问题。另外，Medusa 的消息缓存机制并不符合 GPU 合并访存访问的要求。以上的这些问题制约了 Medusa 性能的进一步提高。

相对于 Medusa 的 EMV 模型，Cusha[42]系统则是一种基于顶点模型的图计算系统。在 Cusha 系统里，使用顶点模型编程图算法，并执行在 GPU 并行环境上。Cusha 使用 G-Shards 图表示方法，减少非合并内存访问的次数。但是与传统的图表示方法相比，G-Shards 空间利用率不高。当要处理大图数据时，G-shards 将无法胜任。

1.4 本文研究内容

Pregel 图计算系统，用于解决大图数据的分布式计算问题。和 MapReduce 类似，Pregel 主要包括两部分：高层次抽象的编程模型和分布式计算的运行时系统。Pregel 在集群环境的优势，不断地激发科研人员将 Pregel 移植到其他的并行环境中。随着 GPU 相关软件包和 GPU 通用计算语言的发布，比如，CUDA 和 OpenCL，GPU 已经演化成为一种强大的并行计算环境。与 CPU 相比，GPU 逐渐演变成为高度并行的多线程、多核处理器，同时具有强大的计算能力和超高的内存带宽。可是对于程序员来说，迁移到 GPU 并行环境比较棘手，因为要理解 GPU 硬件特性还是比较困难。针对这些问题，本文研究了基于 GPU 的图计算，提出 Edge-Vertex 编程模型，极大地简化了 GPU 程序的编写难度。同时，为了解

决线程负载不均衡和合并访问内存这两个问题，提出了基于 GPU 的近似排序算法和数据排列重映射算法。基于以上研究成果，实现了基于 GPU 的图计算系统—PregelGPU。最后通过一系列的实验，表明：PregelGPU 相比于其他基于 GPU 的图计算框架，获得了 1.6x—4.5x 的加速比。本文的主要工作包括：

1、对 Pregel 的编程模型和执行流程进行分析和研究；

2、对比现有基于 GPU 的图计算系统，提出 Edge-Vertex 编程模型，同时详细阐述 PregelGPU 的系统设计；

3、为了优化 GPU 线程负载不均衡问题，提出基于 GPU 的近似排序算法；

4、根据 GPU 合并访问全局内存的特性，提出应用于消息缓存的数据重映射算法，减少非合并访问次数，提高全局内存的实际带宽

5、通过以上的研究，实现 PregelGPU 系统，并对 PregelGPU 系统进行性能分析与评估，证明了系统的高效性。

1.5 本文结构

第 1 章：讨论了本文的选题背景和科学意义，同时分析了 GPU 体系结构和图算法的研究现状。

第 2 章：研究了分布式图计算系统的编程模型及相关系统构架，并详细地阐述了基于 GPU 的图计算系统和相关优化方法，最后指出以上两个系统的不足和相应的改进策略。

第 3 章：详细分析了 PregelGPU 的系统设计。首先阐述 Edge-Vertex 编程模型和 PregelGPU 执行流程，然后展示 PregelGPU API 并用通过样例说明 API 的使用方式，最后分析了 Edge-Vertex 模型的性能瓶颈及优化从发点。

第 4 章：对本文提出的 PregelGPU 系统进行性能优化和加强。首先针对 GPU 线程负载不均衡问题，提出基于 GPU 的近似排序算法，缓解相同 warp 的线程负载不均衡的压力。然后根据 GPU 合并访存的硬件特性，提出应用于消息缓存的数据重映射算法，减少非合并访存次数。

第 5 章：对 PregelGPU 系统进行实验分析。首先详细说明实验平台和实验数据，然后进行性能评估。实验评估主要集中在四方面：(1) PregelGPU 和 Medusa 总体性能评估；(2) 分析系统预处理的效率；(3) 分析近似排序的时间消耗和对系统图计算性能的影响；(4) 分析数据重映射算法对系统图计算性能的影响。

最后，对全文所做的工作进行了总结，并指出了下一步的研究方向。

第 2 章 图计算系统研究

本章对 Pregel 图计算系统进行，包括 Pregel 执行模型、Pregel 编程模型、Pregel API、Pregel 系统构架以及 Pregel 性能瓶颈。然后阐述基于 GPU 的图计算系统 Medusa，内容涵盖 Medusa 编程模型、API、消息缓存机制以及 Medusa 的性能瓶颈和缺陷。最后，针对这两个图计算系统的优缺点，提出相应的解决方案。

2.1 Pregel 概述

Pregel 是可扩展、自动容错的分布式图计算系统，使用其提供的 API 可以方便地编写图算法，并运行在分布式集群环境中，主要应用于图遍历（BFS）、最短路径（SSSP）和 PageRank 计算等。

2.1.1 Pregel 执行模型

Pregel 把图计算的运算模型归结为：在图顶点(Vertex)上迭代执行特定的算法。每次迭代称为一个超步（super step）。如图 2.1 所示，每一个超步都包含本地计算、通信、同步这三个过程。由图 2.1 归纳 Pregel 执行模型的四大特点：输入为有向图（directed graph）；计算分成一系列超步；以节点为中心，在超步内，每个节点执行自己的计算任务；两个连续超步间进行通信和同步。

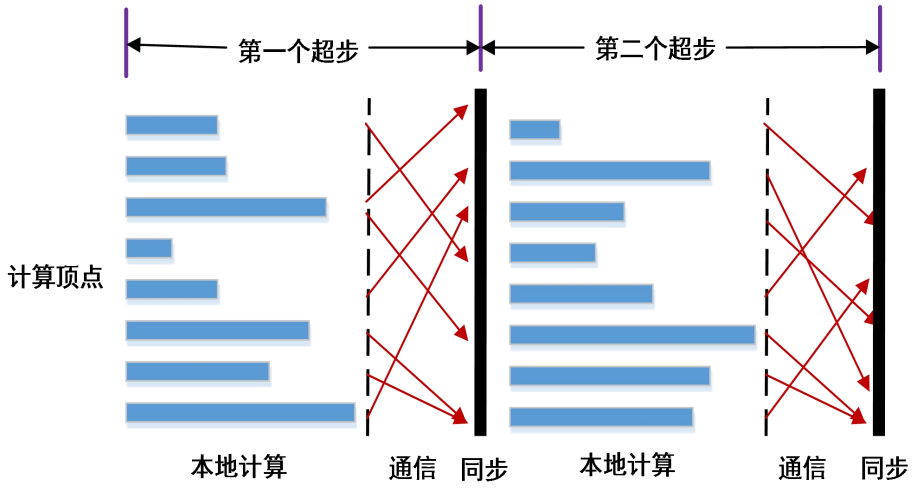


图 2.1 Pregel执行模型

在每一轮超步中，Pregel 系统都为每一个顶点触发用户定义的函数，这些顶点在概念层次上进行并行运算。这个用户定义的函数要包含顶点 V 和超步 S 的

执行代码。该函数首先接收超步 S-1 发送给顶点 V 的消息，然后发送消息给其他顶点（这些消息将会在超步 S+1 被其他顶点接收），最后更改顶点 V 的值和状态。一般消息沿着顶点 V 的出边（从顶点指出的边）方向发送，但是消息也可以发送给其他顶点，改变图的拓扑结构。

Pregel 执行模型非常容易在分布式集群环境实现（implementation）。因为整个计算流程中，数据的接收和处理是以超步为节拍来同步的，在超步 S 中各个顶点所发送的消息，直到超步 S+1 才会被目标顶点所接收和处理并触发状态变更。这种基于节拍的处理流程，很大程度上简化了数据同步的处理。所以在同一轮超步里，不需要同步机制来保障代码的执行顺序，所有的网络通信都集中在超步 S 和超步 S+1 的过渡阶段。同时，Pregel 执行模型暗含了同步原语，这就保证 Pregel 程序可以避免死锁（deadlock）和数据竞争问题（data race），用户就比较容易实现算法的语义。

但是在典型的图计算中，顶点数目都远远大于计算节点的数目，所以必须尽量平衡节点的任务，即在同一个超步内，尽量保证节点与节点的执行时间大致相同，这样能减少超步间同步的延迟时间。

2.1.2 Pregel 编程模型

有向图是 Pregel 计算的输入数据，每个顶点都会用唯一的符号标识。同时，每个顶点都有一个用户定义且可更改的值关联。有向边（directed edge）包括源顶点标识符、边的权值、目的顶点标识符这三个最基本的属性。典型的 Pregel 计算通常先初始化图数据，然后进行一系列全局同步的超步，直到达到算法停止的要求，最后输出结果。

在每一轮超步里，用户定义的函数（user-defined function）描述了给定算法的逻辑，顶点并行地调用用户定义的函数进行计算。顶点编程模型是以顶点为中心，进行相关计算，即顶点可以改变其值和状态，接受前一个超步发送给该顶点的消息，然后发送消息给其他顶点（消息将在下一个超步接受），甚至改变图的拓扑结构。边(edge)并不是顶点模型的核心对象，没有相应的计算运行在边上。

在 Pregel 系统里，算法是否结束取决于所有顶点是否同意终止（vote to halt）。在刚开始的第一个超步，所有顶点处于活跃状态（active state），活跃顶点都会参与超步计算。如果顶点同意终止，那顶点变成非活跃状态（inactive state），这就意味着如果没有外部条件触发这个顶点，这个顶点将不参与接下来的超步计算。即 Pregel 系统在后续的超步将不会计算这个顶点，除非这个顶点接受到一个新消息，使得顶点再次达到活跃状态。算法达到终止的条件有两个：（1）

所有顶点同时处于非活跃状态；（2）同步时没有消息在传送。如图 2.2 所示，解释顶点状态的相互转换条件。

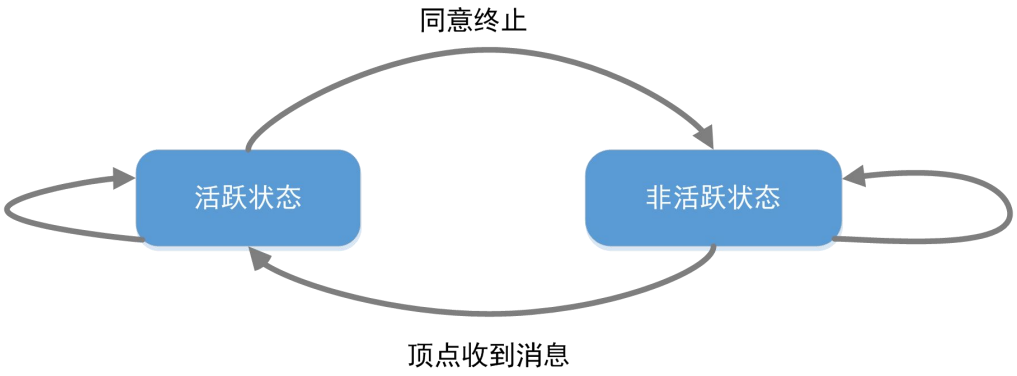
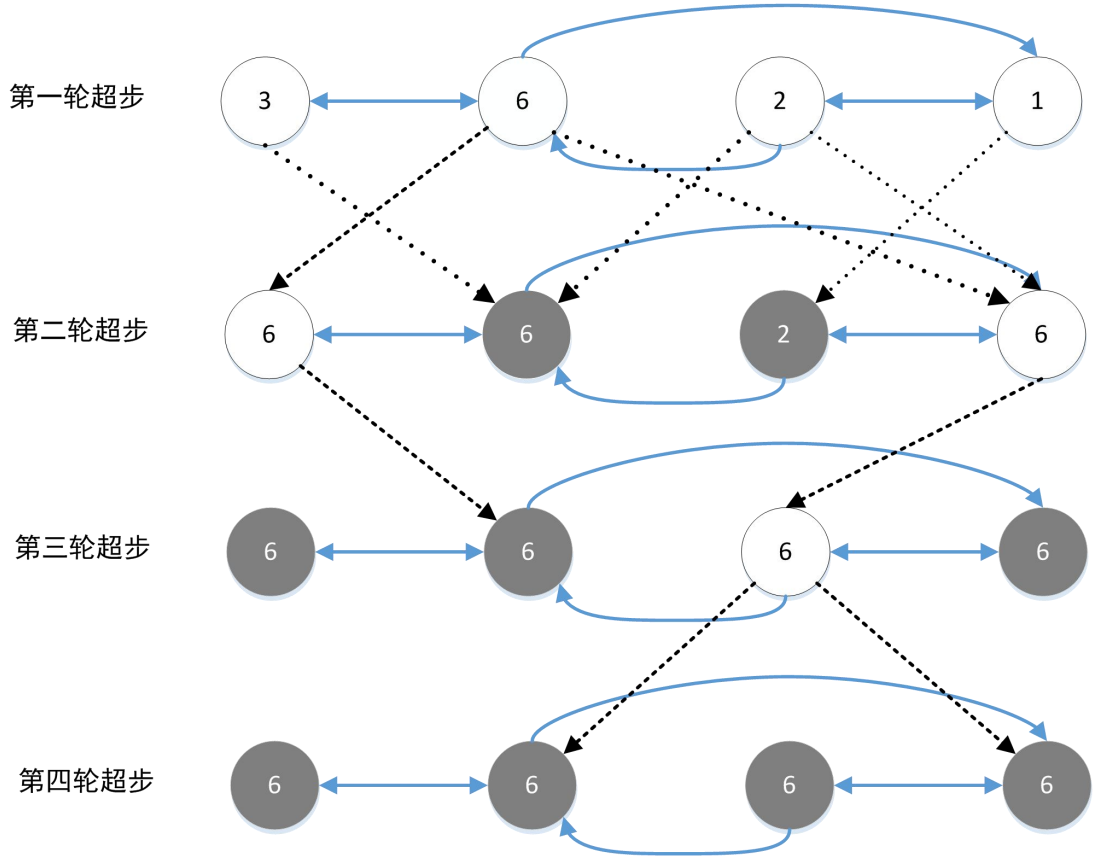


图 2.2 顶点状态转换



白色节点表示活跃节点，灰色节点表示非活跃顶点，虚线表示消息

图 2.3 Pregel编程模型求解最小值

整个 Pregel 程序的输出，是所有顶点输出的集合。通常来说 Pregel 程序的输出图和输入图是同构关系，但是这并不是系统的一个必要属性，因为顶点和边可以在计算的过程中进行添加和删除。基于图数据的挖掘算法就可能仅仅是输出从图中挖掘出来的聚合数据。比如聚类算法，就是从一个大图中生成非连

通顶点组成的小集合。

图 2.3 是一个求解最大值的简单例子，可以简单地说明上述的基本概念。给定一个强连通图，图中每个顶点都包含一个值，最终将会把最大值扩散到每个顶点。在每一个超步中，任何一个顶点从消息中接收到一个比其当前顶点值更大的值，那么把这个值设置为该顶点值并且发消息给其所有的相邻顶点；如果顶点没有接受到比它的值更大的消息，则该顶点设置为非活跃状态，直到该顶点接收消息且消息的值大于该顶点值，重新把该顶点设置为活跃状态。当某一个超步中已经没有顶点要更新其值，而且没有消息在传送，那么宣告计算结束。

Pregel 系统选择纯消息模式（pure message passing model）作为消息通信机制。这样做有两个原因：第一，任何图算法都可以使用纯消息模式表达；第二，出于性能考虑，避免远程系统调用的开销。但是在集群环境，从远程节点传递消息还是有比较高的延迟。作为系统优化，Pregel 通过异步和批量消息通信方式，减少这种消息通信的延迟。

其实，图算法也可以被写成是一系列的链式 mapreduce 作业。但是 Pregel 采用一种完全不同的编程模式和执行方式，这是基于可用行和性能考虑。在 Pregel 系统，顶点和边在本地节点进行计算，仅仅利用网络来传输信息，而不是传输计算数据。而 MapReduce 本质上是面向函数的，所以图算法用 mapreduce 来实现就需要将整个图的状态从一个阶段传输到另外一个阶段，这样就需要进行许多的通信和随之而来的序列化和反序列化的开销。另外，在一连串的 mapreduce 作业中各阶段需要协同工作也给编程增加了难度，这样的情况能够在 Pregel 系统中避免。

2.1.3 Pregel API

本小节主要分析 Pregel C++ API，并用 PageRank 算法和单源最短路径算法（SSSP 算法）阐述如何灵活地运用这些 API。要利用顶点模型编写图算法需要继承 Pregel 系统预定义的顶点基类。如图 2.4 所示，顶点基类的模板参数（template parameter）定义了三个类型参数，逻辑上分别表示顶点（vertex），边（edge）和消息（message）的数据类型。顶点类型默认有一个顶点的值属性与之关联。虽然这种设计方式可能看上去有一些局限性，但用户可以通过继承基类的方法管理或者增加的其他属性和方法。同样边和消息类型的定义也类似这样。

用户通过继承顶点基类定义派生类，并重写虚函数 Compute()，该函数会在超步中对每一个顶点进行调用。预定义的顶点基类定义一系列方法，包括查询当前顶点的信息和其出边的信息，发送消息到其他的顶点。在定义 Compute()方法的过程中，可以通过 GetValue()方法得到当前顶点的值，或者调用

MutableValue()方法来修改当前定点的值。同时还可以利用顶点的出边迭代器（out-edge iterator）修改出边的权值，这种修改是即时生效的。由于这种可见性是仅限于被修改的顶点，所以不同的顶点进行并发地数据访问时，不存在数据的竞争关系。

```
template<typename VertexValue,
         typename EdgeValue,
         typename MessageValue>
class  Vertex
  {
     public:
         virtual void Compute(MessageIterator* msgs)=0;

         const string& vertex_id() const;
         int64  superstep() const;

         const VertexValue& GetValue();
         VertexValue* MutableValue();
         OutEdgeIterator GetOutEdgeIterator();

         void SendMessageTo(const string& dest_vertex,
                            const MessageValue& message);
         void VoteToHalt();
  };
```

图 2.4 顶点基类

```
class PageRankVertex:public Vertex<double,void,double>
{
   public:
        virtual void Compute(MessageIterator* msgs)
        {
            if(superstep() >1)
            {
                double  sum=0;
                for(; !msgs->Done(); msgs->Next())
                      sum +=msgs->Value();
                *MutableValue()=0.15/NumVertices() + 0.85 * sum;
            }
            if(superstep()<=30)
            {
                const int64 n = GetOutEdgeIterator().size();
                SendMessageToAllNeighbors(GetValue()/n);
            }
            else
             {
                VoteToHalt();
             }
         }
};
```

图 2.5 Pregel API实现PageRank算法

顶点与顶点之间的通信是纯消息传递的方式，每一个消息都包含了消息的值和目的顶点标识符。消息值的数据类型则由用户通过 Vertex 类的 template 参数来确定。顶点可以发送多个消息到超步中，在超步 S+1 中使用顶点 V 的消息迭代器（Message Iterator）接收超步 S 发送给顶点 V 的消息。消息迭代器并不保证消息的顺序，但是可以保证消息一定会被传送并且不会重复。在 Pregel 系统，图算法通常的实现方法是：对于一个顶点 V，遍历其自身的出边，向每条出边发送消息到该边的目的顶点，比如在图 2.5 中，展示了 PageRank 算法的实现过程。

在图 2.5，PageRankVertex 类继承顶点基类而且重写 Compute()函数。PR 值为 double，把顶点值和消息的类型都设置为 double。在 PageRank 算法中，边不存储信息，所以把边的类型设置 void。在程序开始时初始化数据（superstep 1），PageRank 算法把每一顶点的值初始化为 1/NumVertices()。在前 30 轮超步，每个顶点都沿着出边发送消息，消息的值为该顶点的值除以出度。从第二轮超步开始，每个顶点要把发送给它的消息求和，然后通过 0.15/NumVertices()+0.85×sum 这个公式计算顶点的新 PageRank 值。当运行第 30 轮超步之后，不需要再发送消息，且每个顶点同意终止，那么 Pregel 程序停止运行。而在实际情况下，PageRank 算法可以自动达到收敛状态，通过收敛条件，判断算法是否结束。

```
class ShortestPathVertex:public Vertex<int,int,int>
{
    public:
        virtual void Compute(MessageIterator* msgs)
        {
          int mindist=IsSource(vertex_id()) ? 0: INF;
           for(; !msgs->Done(); msgs->Next())
                  mindist = min(mindist , msgs->Value());
           if(mindist < GetValue())
           {
                  *MutableValue() = mindist;
                  OutEdgeIterator iter = GetOutEdgeIterator();
                  for( ; !iter.Done(); iter.Next())
                         SendMessageTo(iter.Target() , mindist + iter.GetValue());
           }
           VoteToHalt();
        }
};
```

图 2.6 Pregel API实现单源最短路径算法

当一个顶点发送消息，尤其是发送消息给位于另一节点的顶点时，会产生网络通信开销。比方说，顶点收到许多的整数型消息（integer message），如果算法仅仅关心这些消息的总体，而不是单个消息。这种情况下，系统可以把多个消息合并成一个消息，该消息仅包含顶点消息总体特性，这样就减少网络传

输和本地缓存的开销。Combiner 在默认情况下没有被开启，这是因为要找到一种对所有图算法都合适的 Combiner 是不可能的。如果用户想要开启 Combiner 的功能，可以继承 Combiner 类，重写其虚函数 Combine()。框架并不会确保哪些消息会被合并，也不会确保传送给 Combine()的值和合并操作的执行顺序。所以 Combiner 仅仅被用来减少网络通信量。图 2.6 使用 Pregel API 实现单源最短路径算法，和图 2.7 解释如何使用 Combiner 合并消息。

最短路径问题是图论著名的研究问题，有着广泛的应用，并存在一系列的变种算法[43]。比如，单源最短路径问题就是给定源顶点，找到这个顶点到图中其他顶点的最短距离。在图 2.6，首先把源顶点的值设置为 0，其他顶点值初始化为 INF 或者其他值（这个值要比图中源顶点到其他顶点的距离都要大）。在每一轮超步，顶点从邻接顶点接收消息，计算该顶点至源顶点距离的最小值 mindist。如果 mindist 小于顶点值，那么将顶点值设置为 mindist，同时把发送消息给其他邻接顶点(消息值为 mindist 加上边的权值)。在第一轮超步，源顶点值为 0 和其他顶点的值设置 INF，源顶点发送消息给它的邻接顶点并更新其值。然后在随后的超步，源顶点的邻接顶点又发消息给它的邻接顶点并更新其值，如此循环反复，在整个图产生泛洪式更新。这个算法的停止条件是顶点的值不再更新，即每个顶点都找到和源顶点的最短距离（此时顶点值为 INF 表示源顶点和该顶点不连通）。

```
class MinIntCombiner : public Combiner<int>
{
    public:
        virtual void Combine(MessageIterator* msgs)
        {
            int mindist=INF;
            for(; !msgs->Done(); msgs->Next())
                    mindist = min(mindist , msgs->Value());
            Output( “combined_source” ,mindist);
        }
};
```

图 2.7 利用Combiner合并消息

在这个算法里，消息包含潜在的最短距离。因为接收消息的顶点最终只要最小值，所有可以在加入 combiner 优化算法性能。实现代码如图 2.7 所示，利用 combiner 极大地减少节点间的网络通信和节点的本地消息缓存。

单源最短路径算法的 Pregel 版本经常要和串行算法比较，比如 Dijkstra 最短路径算法[44]和 Bellman-Ford[45]算法，这些传统算法无法扩展到集群环境处理大图数据。

2.1.4 Pregel 系统架构

Pregel 运行时系统把图分解成许多的分区（partition），每一个分区包含了一些顶点和以这些顶点为起点的边。顶点分配到某个分区取决于该顶点的 ID，这就表示即使分配到的其他节点上，也可以通过顶点的 ID 确定该顶点是属于哪个分区，甚至在顶点尚不存在时就可以知道其所属的分区。Pregel 默认的顶点分区是通过把顶点 ID mod N 生成，N 为所有分区总数，即利用简单的哈希函数分配节点的计算任务。有些应用程序对默认的任务分配策略不太支持，而定义用户自定义的分配函数可以减少由于分配策略带来的开销。用户可以根据图数据和算法特点，重新定义任务分区函数。比如，一种典型的启发式 Web graph 就可以将来自同一个站点的网页顶点的数据分配到同一台计算节点上进行计算。

和 MapReduce 系统类似，计算节点也分为 master 和 worker。在不考虑系统容错的情况下，Pregel 程序的执行过程分为如下几个步骤：

（1）用户定义的程序分别在集群中的某些计算节点上开始执行。其中有一个进程将会作为 master，master 并不分配数据计算任务，只是负责管理和协调计算节点任务。worker 进程利用集群管理系统中提供的名字服务来定位 master 进程是运行在哪台计算节点上，并发送注册信息到这个 master 进程中。

（2）master 进程决定了对这个图进行分区， 并分配一个或多个分区到 worker 上进行运算。一个 worker 上有多个分区的情况下，可以提高分区间的并行度，更好地达到负载平衡，并产生更高的执行效率。每一个 worker 都需要记录在其分区任务的状态，对该分区中的顶点执行 Compute()函数，并管理从其他 worker 上传过来的消息。worker 向 master 汇报任务的完成进度，然后 master 向所有 worker 广播，让每个 worker 都了解所有 worker 的任务完成进展。

（3）master 进程将用户的输入数据中的一部分分配到集群的 worker 中，这些输入当做记录（record）的集合，每一条记录都包含一些顶点和边。对输入数据的分割和整个图是垂直正交的，而且通常都是基于文件边界。如果 worker 加载的顶点恰好在该 worker 分配的分区中，只需将该顶点加入计算队列。否则，这个 worker 需要发送消息到该顶点所属的 worker。当所有的输入数据都被加载完成后，所有的顶点就都已经处于活跃状态，等待执行用户定义的 Compute()函数。

（4）master 给每个 worker 发指令，让其运行超步，worker 轮询在其分区的顶点，每个分区会创建一个线程来做轮询。worker 调用每个 active 顶点的 Compute()函数，接收从上一轮超步发送的消息。消息是被异步发送的，这是为了使得计算和通信变得并行化，但是消息在超步末尾将会被发送完。当一个 worker 完成超步中所有顶点的计算后，会告诉 master 当前分区在下一轮超步开

始时活跃顶点的数目。这个过程被不断的重复，只要有顶点还处在活跃状态，或者还有消息在传送。

（5）当所有的顶点都处于终止状态，master 给所有的 worker 发指令，让 worker 保存计算结果。

2.1.5 Pregel 性能瓶颈

Pregel 系统作为分布式图计算系统，存在的最大问题就是 worker 负载不均衡。Pregel 主要采用简单的哈希算法根据顶点 ID 划分任务，虽然分区的顶点数目大致相同，但是分区图的稠密性却无法确定。稠密图的计算量远远大于稀疏图，所以负载均衡问题是系统性能优化的重要方面。

2.2 Medusa 系统概述

Medusa 基于 GPU 的图计算系统，把 Pregel 的执行模型和编程模型应用于 GPU 并行环境，针对 GPU 的硬件特性和图数据特点进行以下两点改进和优化：（1）提出细粒度并行的 EMV 编程模型及 API，代替传统 Pregel 系统的顶点编程模型；（2）根据图数据结构特点，设计基于数组的消息缓存机制，把同一个顶点的消息放置在消息缓存相邻的位置，这样避免了顶点接收消息的合并操作（grouping operation），实现消息写入和消息读取的功能。由于 Pregel 和 Medusa 基于不同的并行环境，下面章节着重阐述 Medusa 系统设计的改进之处，并分析 Medusa 的性能瓶颈。

2.2.1 Medusa 编程模型及 API

正如 2.1 小节所讨论的内容，传统的 Pregel 编程模型基于顶点模型并提供以顶点为中心的 API，用户使用其 API 可以访问相关边和消息的属性（边/消息迭代器）。尽管分布式图计算系统广泛地采用顶点编程模式，并具有良好的可编程性和优秀的性能表现，但是这种粗粒度的模型并不适合 GPU 并行环境，容易产生 GPU 线程负载不均衡。因为对于无规律图，每个顶点的边和消息数目不太相同，以顶点为单元分配任务，顶点接收消息、遍历边和发送消息数目也不一样，如果这些数目相差比较大，将导致严重的任务负载不均衡。

Medusa提出细粒度并行的EMV（Edge-Message-Vertex）编程模型，把以顶点为中心的API拆分成边(Edge)、消息（Message）和顶点（Vertex）这三类API。Edge API负责发送消息，Message API 对消息进行操作，Vertex API接收消息，用户定义这三类API的逻辑代码。在每个超步中，Edge API、Message API 和Vertex API 分

别以GPU kernel方式依次调用。EMV编程模式通过任务分解的方式，能够缓解GPU线程的负载不均衡问题，提高GPU并行线程资源的利用率。

如表 2.1 所示，Medusa 提供 6 个可定义的 EMV API，每个 API 用于处理顶点（VERTICES）、边（ELIST，EDGE）或消息（MESSAGE、MLIST）。用户定义 API 分为两类（variant）：集合对象（collective）API 和单个对象（individual）API。集合对象 API 允许访问与顶点关联的所有边或者消息（即 Edge-list 和 Message-list），而单个对象 API 是仅仅可以访问单独的一条边或者消息。Medusa 系统同样提供 Combiner 函数，用户定义 Combiner 函数实现对同一个顶点的边或者消息进行汇集。这些用户定义的 API 都不需要 CUDA 编程经验，用户编写传统的串行代码实现这些 API 接口。

表 2.1 Medusa用户定义API

API Type	Parameters	Variant	Description
ELIST	Vertex v,Edge-list el	Collective	Apply to edge-list el of each vertex v
EDGE	Edge e	Individual	Apply to each edge e
MLIST	Vertex v,Message-list ml	Collective	Apply to message-list ml of each vertex v
MESSAGE	Message m	Individual	Apply to each message m
VERTEX	Vertex v	Individual	Apply to each vertex v
Combiner	Associative operation	Collective	Apply an associative operation to all edge-lists or message-lists

在表 2.2 展示了 Medusa 提供的系统级 API，控制 Medusa 程序的执行状态。Medusa 利用 EMV<type>::Run()函数配置 GPU 线程参数和调用用户定义的 API。同时表 2.2 也说明了 Medusa 存储模块，系统提供两个系统 API：AddEdge 函数和 AddVertex 函数。通过循环添加顶点或者边的方式初始化图结构，然后把图数据从主存传输到 GPU 全局内存，GPU 全局内存和主存之间的数据传输是透明地完成。

表 2.2 Medusa系统API

API/Parameter	Description
AddEdge(void* e),AddVertex(void* v)	Add an edge or vertex into the graph
InitMessageBuffer(void* m)	Initiate the message buffer
maxIteration	The maximum iterations that Medusa executes
halt	A flag indicating whether Medusa stops the iteration
Medusa::Run(Func F)	Execute F iteratively according to the iteration control
EMV<type>::Run(Func *f*)	Execute EMV API *f* with type on the GPU

Medusa 还负责将用户定义 API 并行地执行在 GPU 并行环境，为此 Medusa 提供两个系统 API：Medusa::Run(Func F)和 EMV<type>::Run(Func *f*)。Medusa 程序总入口是 Medusa::Run(Func F)，F 通常包括 EMV 函数的调用集合。单个 EMV 函数运行是通过 EMV<type>::Run(Func *f*)，type∈{ ELIST，EDGE，MLIST，

MESSAGE，MLIST}。

图算法大部分需要多次迭代达到收敛状态。为了更好地控制迭代，Medusa 提供两种方式控制算法的终止。第一种方式，用户通过设置参数 maxIteration，控制最大迭代次数，当系统运行的迭代次数达到预定义的值，系统终止运行。第二种方式，Medusa 设置一个全局变量 halt，可以通过 EMV API 修改其值，如果其值为 false，程序继续运行，直到 halt 被设置为 true。当然，也可以同时使用这两种方式控制算法终止。

```
Device code API:
struct SendRank {  /*ELIST API */
    __device__ void operator()(EdgeList el,Vertex v)
{
      int edge_count = v.edge_count;
      float msg = v.rank/edge_count;
      for(int i = 0; i < edge_count; i++)
          el[i].sendMsg(msg);
   }
};
struct UpdateVertex {  /*VERTEX API */
    __device__ void operator()(Vertex v,int
super_step) {
       float msg_sum = v.combined_msg();
       vertex.rank = 0.15 + msg_sum * 0.85;
    }
};
Data structure definitions:
struct vertex {
  float pg_value;
  int vertex_id;
};
struct edge {
  int head_vertex_id,tail_vertex_id;
};
struct message {
 float pg_value;
};
```

```
Iteration definition:
Void PageRank
{
    /*Initiate message buffer to 0*/
    InitMessageBuffer(0);
    /*Invoke the ELIST API*/
   EMV<ELIST>::Run(SendPank);
    /*Invoke the ELIST API*/
   Combiner();
    /*Invoke the VERTEX API*/
   EMV<VERTEX>::Run(UpdateRank);
}
Configurations and API execution:
Void main(int argc , char **argv )
{
    ……
    Graph my_graph;
   /*Load the input graph*/
    conf.combinerOpType = MEDUSE_SUM;
    conf.combinerDataType=MEDUSE_FLOAT;
    conf.gpuCount = 1;
    conf.maxIteration = 30;
   /*Setup device data structure*/
    init_Device_DS(mg_graph);
    Medusa::Run(PageRank);
    /*Retrieve results to my_graph*/
    Dump_Result(my_graph);
}
```

图 2.8 Medusa API实现PageRank算法

图 2.5 说明如何利用 Pregel API 实现 PageRank 算法。为了比较两种编程模型的差异，图 2.8 展示使用 Medusa API 实现 PageRank 算法。如图 2.8 所示，需要预先定义一些数据结构（如 vertex，edge，message）。PageRank 函数包括用户定义的 EMV API 调用：ELIST 类型的 API（SendRank），消息类型的 Combiner 和 VERTEX 类型的 API（UpdateRank）。main 函数需要配置 Combiner 的数据类型和操作类型、GPU 个数和最大迭代次数。Init_Device_DS 函数负责载入图数据并传输到 GPU 全局内存，Medusa::Run（PageRank）调用 PageRank 函数。

2.2.2 Medusa 消息缓存机制

消息缓存主要负责消息的存储和读取，通过 InitMessageBuffer 系统函数分配和初始化存储空间。消息缓存的实现机制主要通过数组和链表实现，现在主要讨论这两种实现的优缺点。基于数组的缓存必须分配连续存储区域的数组，因为数组大小是固定的，所以要预先确定缓存的大小和每一个消息在缓存的位置，避免读写冲突。如果发送给同一个顶点的消息不是连续存储的，那么需要增加合并操作支持 Combiner API 的消息处理。而基于链表的缓存通过动态分配内存，不需要预先确定缓存的大小。但是当 GPU 并行地运行成千上万的线程，基于链表的缓存需要大量原子操作（atomic operation），分配消息的存储空间。

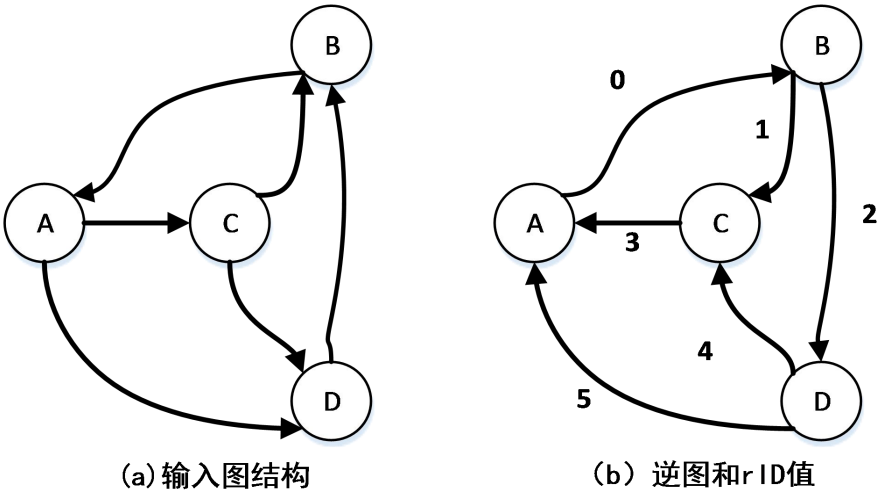


图 2.9 输入图转成逆向图

Medusa 消息缓存机制是基于数组的实现，预先确定消息缓存的大小和每一个消息在消息缓存的位置。因为可以提前确定每一条边发送消息的最大消息量，所以计算（1）总消息量的最大值；（2）每一个顶点接受消息的最大消息量。通过这种方式就可以预先分配连续的缓存空间，同时要预先计算要每条边把消息发送到缓存的位置和每个顶点从缓存读取消息的位置。

为了避免消息的合并操作，通过逆向图建立索引号方式，保证发送给同一个顶点的消息存储在相邻的位置。当把输入图载入内存时，Medusa 系统通过交换边的源顶点和目的顶点（图 2.9(a)和图 2.9(b)所示），构建逆向图（reverse graph），边按照邻接数组储存。在逆向图中，给每一条边标号 rID（reverse ID），rID 代表该边在邻接数组的位置。如图 2.9(b)所示，说明每一条边的 rID 值。

其中 rID 值隐含重要的性质：同目的顶点的边，它们的 rID 值是连续的。例如，在图 2.9(b)中，rID 值为 4 和 5 的边，它们的目的顶点都是顶点 D。利用这个特性，保证发送给同顶点的消息都是连续存储的，即顶点从消息缓存读取消

息的位置也是连续的，如图 2.10 所示。

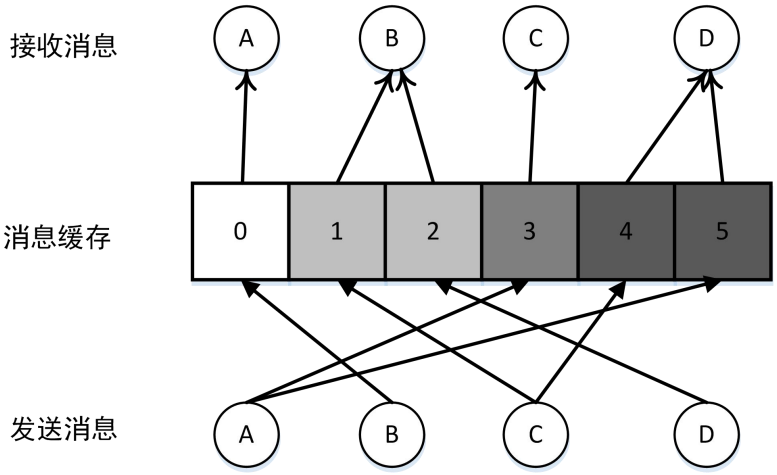


图 2.10 Medusa消息缓存机制

总结 Medusa 消息缓存的实现步骤：首先，要分配(E×m)存储单元，其中 m 表示一条边发送消息的最大数目，由用户设置 m 的值，比如，PageRank 算法，每条边至多发送一条消息，所以设置 m=1。然后，消息是通过边发送到目的顶点，如果边的 rID 值为 k，那么该消息发送在消息缓存的(k×m)位置。在图 2.10 中，展示了边把消息发送在消息缓存的位置和顶点从消息缓存读取消息的位置，其中 m=1。当发送消息时，边的 rID 值确定每条边发送消息的位置，即写消息的位置。当顶点读取消息时，发送给同顶点的消息存储在连续的位置。通过这种消息缓存机制，消息已经根据目的顶点聚集在一起，不需要额外的消息合并操作。

2.2.3 Medusa 的缺陷和性能瓶颈

Medusa 提出 EMV 编程模型代替传统的顶点模型，适合 GPU 细粒度的并行环境，但是 EMV 模型比顶点模型复杂，把太多的底层细节暴露给用户，增加编程难度。

虽然和顶点模型相比，EMV 模型缓解任务负载不均衡问题，但是任务不均衡问题依然存在，主要集中在 Vertex 计算上。因为顶点负责接收消息，而接收消息数目相差悬殊。需要进一步的轻量级算法解决 Vertex 计算的负载不均衡问题。

Medusa 的消息缓存机制把发送给同一个顶点的消息存储在相邻的位置，不需要增加额外的合并操作。但是，这种读消息的方式不能达到 GPU 合并访问全局内存的要求，无法利用 GPU 的高内存带宽。

针对 Medusa 系统的缺陷和性能瓶颈，在第三章将着重强调本文提出的

Edge-Vertex 编程模型和系统执行流程。在第 4.1 节，根据 Vertex 计算负载不均衡问题，提出基于 GPU 的近似排序算法。在第 4.2 节，提出应用于消息缓存的数据重映射算法，达到 GPU 合并访问全局内存的要求。

2.3 小结

本章首先讨论 Pregel 图计算系统，主要从包括 Pregel 执行模型、Pregel 编程模型、Pregel API、Pregel 系统构架和 Pregel 性能瓶颈这五方面分别进行阐述。然后，分析基于 GPU 的图计算系统 Medusa，Medusa 系统把传统 Pregel 编程模型应用到 GPU 并行环境，根据 GPU 硬件特性和图数据结构的特点进行改进和优化，主要集中在提出 EMV 编程模型和 API，以及 Medusa 消息缓存机制。最后分析 Medusa 的缺陷和性能瓶颈，如何消除这些性能瓶颈的影响，进一步提高系统性能，是本文的研究重点。

第 3 章 PregelGPU 系统设计

本文提出的 PregelGPU 系统是基于 GPU 的图计算系统，主要由两部分构成：类似于 Pregel 的 API 和针对 GPU 并行环境进行优化的运行时系统（runtime system）。类 Pregel API 可以为基于 GPU 环境的图算法提供统一的编程接口，降低图算法的编程难度，同时充分利用 GPU 的并行资源。运行时系统屏蔽 GPU 的底层细节，根据相关硬件特性进行性能优化。本章详细阐述 PregelGPU 的系统设计，主要内容包括 Edge-Vertex 编程模型、系统执行流程和 PregelGPU API，最后分析 Edge-Vertex 模型潜在的性能瓶颈。

3.1 Edge-Vertex 编程模型

首先，解释以顶点为中心的编程模型为何不能直接用于 GPU 体系结构。如 2.1 节所分析的内容，在顶点编程模型中，系统通过运行用户定义函数（重写顶点基类的虚函数 Compute）来执行每个顶点的计算任务（例如顶点发送消息或接收消息）。为了充分利用 GPU 的计算资源，需要将 Compute 函数映射到每个 GPU 的线程上，以便能够充分挖掘和利用 GPU 的并行能力。然而，现实世界的图通常具有幂率分布（power-law degree distribution）的特征[46]，即每个顶点的入度或者出度相差很大。具有这种分布特征的图导致分配给每个 GPU 线程的任务负载不均衡，极大的影响系统性能。

为了缓解线程负载不均衡问题，Medusa 系统提出 EMV 编程模型，把以顶点为中心的计算划分为多个计算部分，分别处理边、顶点、以及消息，为用户提供适合 GPU 细粒度并行的 API。但是 EMV 编程模型暴露消息缓存的内部结构，允许用户显式地对消息进行管理，增加用户利用 EMV API 编写图算法的难度。

通过研究分析和实践发现，把以顶点为中心的计算拆分成边（Edge）和顶点（Vertex）的计算，足够表达大多数的图算法，满足 GPU 细粒度并行的要求，而且能够抽象消息缓存的物理结构。在运行时系统封装消息缓存结构不仅能够减少编程的复杂性，同时为针对不同的 GPU 硬件体系结构进行性能优化提供可能性。

因此，本论文把基于顶点编程模型的计算过程分解为 EdgeCompute 计算和 VertexCompute 计算两个过程，称之为 Edge-Vertex 编程模型。这种编程模型是基于以下假设：由 Edge 负责将消息发送到消息缓存，而 Vertex 则对从消息缓存中接收到的消息进行计算。换言之，消息缓存的构建和管理对于用户是不可见

的。这样就能够在优化消息缓存性能和简化图算法编程难度这两方面取得一定平衡。而且，这种抽象并不会导致性能的下降，因为消息缓存的优化空间还是比较大，4.2 节将重点阐述消息缓存的优化算法。

3.2 系统执行流程

图 3.1 展示了 PregelGPU 系统的执行流程。首先，运行时系统载入图数据，构建 CSR（Compressed Sparse Row，CSR）图存储结构，执行通用的数据管理任务，包括 GPU 的内存分配，主存和 GPU 之间的数据交换等，这些都是由运行时系统完成。然后，进入预处理阶段，最后开始图计算的超步迭代过程。在 PregelGPU 系统，每一个超步包含发送消息到消息缓存的 EdgeCompute 阶段和从消息缓存接收消息的 VertexCompute 阶段。

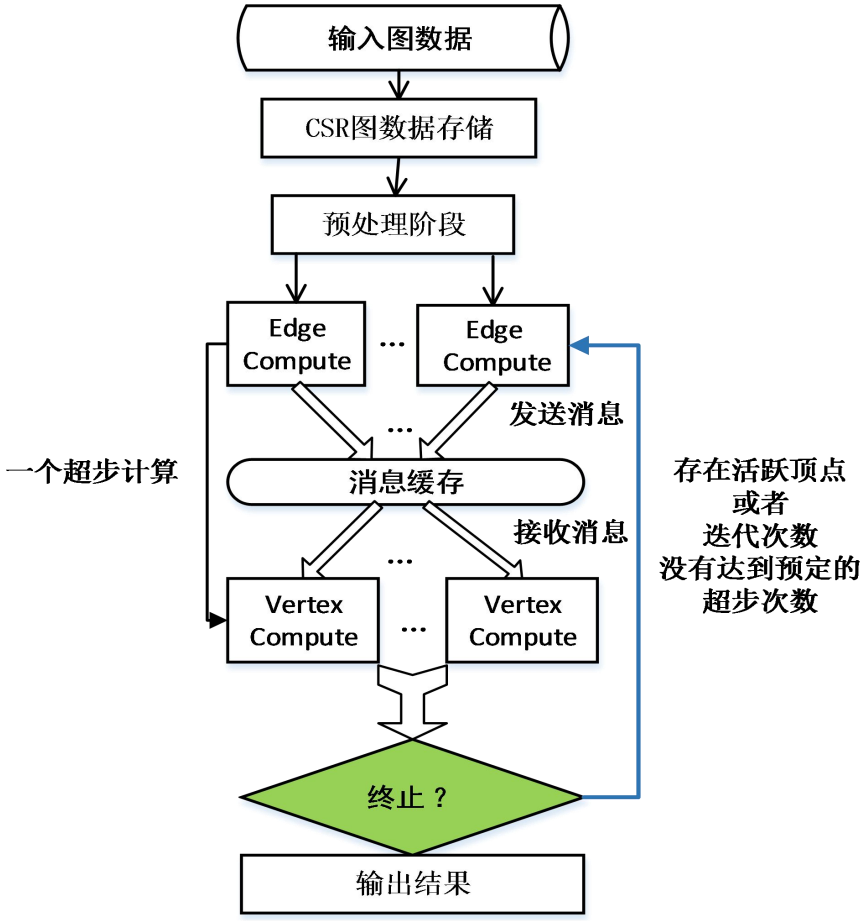


图 3.1 系统执行流程

3.2.1 预处理阶段

在 PregelGPU 系统中，预处理阶段是由运行时系统进行控制和管理，对用户透明。消息缓存的物理结构是一个基于数组的实现，负责消息的存储和读取任务。在进入图计算超步迭代过程之前，首先需要确定消息缓存的大小。然后

预先计算边发送到消息缓存的位置和顶点从消息缓存读取消息的位置，这些预先处理的计算都是在预处理阶段完成。Medusa 消息缓存机制把同一个顶点的消息放置在相邻的位置，这样无法利用 GPU 合并访存的特性，在 4.2 节将详细阐述应用于消息缓存的数据重映射算法，达到 GPU 合并访问全局内存的要求。

3.2.2 EdgeCompute 阶段

在 EdgeCompute 阶段，每个 GPU 线程处理一条或多条边，并按照以下三个步骤顺序执行：

1、读取出边的源顶点值和状态；

2、获取边的权重属性；

3、基于源顶点的值和边的权重进行相关计算，把计算结果作为消息传送给边的目的顶点。

在刚刚开始的第一个超步，每个顶点的都处于是活跃状态，这一点和以顶点为中心的编程模型相同，用户根据算法语义，决定该顶点是否要处于活跃状态。

3.2.3 VertexCompute 阶段

在 VertexCompute 阶段，一个或多个顶点被分配给一个 GPU 线程，该线程也顺序执行以下 3 个类似于 EdgeCompute 的步骤：

1、GPU 线程从消息缓存中读取发送给该顶点的消息；

2、使用这些接收的消息来计算该顶点的值和顶点出边的权值；

3、根据需要可以把新的状态或值赋给该顶点。

在 Pregel 系统中，图算法通常包含一系列超步。而在 PregelGPU 系统，算法也包含一系列超步，但是每一个超步包含 EdgeCompute 和 VertexCompute 这两个阶段，直到顶点都处于非活跃状态或者迭代次数达到预定超步的最大数目，系统停止计算。

3.3 PregelGPU API

Pregel 系统提供的 API 如表 3.1 所示。和现有的图计算框架类似，把 PregelGPU API 划分为两大类：用户定义 API（user-implemented API）和系统 API（system-provided API）。

在用户定义 API 中，initData 函数是必须被调用的，完成数据初始化工作，并配置 GPU 线程数目和 block 参数。正如 3.2 小节阐述的内容，用户通过定义两

个独立的函数（EdgeCompute 和 VertexCompute）来执行相应的图计算。在每一个超步，边和顶点都会调用用户定义的 EdgeCompute 和 VertexCompute 方法，直到满足特定的终止条件。

系统 API 作为库调用（library call）屏蔽 GPU 特有的编程细节。系统提供 startGPU 函数执行特定的 GPU 设备代码（device code），该函数轮流地调用用户定义的函数。而且系统提供两个版本的 startGPU 函数，使得用户可以通过不同的方式控制算法终止。其中一个版本可以指定超步的最大迭代数目，作为算法终止的条件。而另一个版本通过检查下一轮超步中是否存在活跃顶点，来决定是否结束运算。PregelGPU API 也拥有 Combiner 的功能，合并同一个顶点的消息。

表 3.1 PregelGPU API

User-implemented API
/*EdgeCompute function, processing one edge*/ ***void EdgeCompute(Edge e);***
/*VertexCompute function, processing one vertex*/ ***void VertexCompute(Vertex v);***
/*Initialize the value of edges and vertice , configure threads and blocks number*/ ***void initData(void* graph,void* initialValues);***
System-provided API
/*Start the runtime system, and stop until reaching maxSupersteps */ ***void startGPU(int maxSupersteps);***
/*Start the runtime system, and stop until no active vertex exist*/ ***void startGPU();***
/*Set the state of the vertex to inactive*/ ***void voteToHalt(Vertex v);***
/*Set the state of the vertex to active*/ ***void voteToActive(Vertex v);***
/*Combine the incoming messages sent to the same vertex*/ ***Message Combiner(Vertex v);***

3.4 样例

单源最短路径算法（single source shortest paths, SSSP）是一个非常有名并且容易理解的图算法。对于 SSSP 算法，需要指定一个顶点作为单源顶点，并计算图中该顶点到其他所有顶点的最短路径。在 2.1.3 小节，分析了使用顶点编程模型的 API 实现 SSSP 算法。作为比较，在算法 3.4.1 使用 Edge-Vertex API 实现 SSSP

算法。重点说明如何定义 EdgeCompute、VertexCompute 和 Combiner 三个函数，具体实现过程如算法 3.4.1 所示。算法的实现过程充分说明使用 Edge-Vertex 编程模型表达图算法的简洁性和方便性。

算法 3.4.1 PregelGPU API 实现单源最短路径算法

```
void EdgeCompute(Edge e)
1: Vertex srcVertex=e.getSrcVertex();
2: if srcVertex.active then
3:     e.sendMsg(srcVertex.value+e.weight);
4: end if
void VertexCompute(Vertex v)
1: msg_min=Combiner(v);
2: if msg_min<v.value then
3:     vertex.value=msg_min;
4:     voteToActive(v);
5:     stillContinueNextSuperstep(true);
6: else
7:     voteToHalt(v);
8: end if
Typedef:Message int
Message Combiner(Vertex v)
1: min=INF;
2: for each i∈v.inEdgesNum do
3:     if min>v.currentMsg() then
4:         min=v.currentMsg();
5:     end if
6:     v.nextMsg();
7: end for
8: return min;
```

3.5 性能瓶颈分析

Edge-Vertex 编程模型可以保证在 EdgeCompute 阶段 GPU 线程负载均衡。因为 EdgeCompute 函数负责边处理，包括发送消息到边的目的顶点，而一条边只有一个目的顶点，即在 EdgeCompute 阶段，GPU 线程是负载均衡的。但是，在 VertexCompute 阶段，存在严重的 GPU 线程负载不均衡问题。例如，SSSP 算法的 VertexCompute 阶段，需要合并属于同一顶点的消息（Combiner 函数）。但是由于图的入度分布变化比较大，导致顶点接收消息的数目差距悬殊，这样不仅造成在同一个 warp 内 GPU 线程负载不均衡，导致 GPU 计算资源使用率降低。4.1 小节将提出一个轻量级且非常有效的优化算法—基于 GPU 的近似排序算法，解决同一个 warp 的 GPU 线程负载不均衡问题。

另外一个潜在的性能瓶颈是 Combiner 函数。因为 GPU 线程从消息缓存读取

同一顶点的消息进行某种操作（如求最小值），而消息缓存在 CPU 上初始化，按照基于行优先的数据存储存放消息并传输到 GPU 内存，即同一顶点的消息放置在相邻位置。在传统 CPU 环境，这样的存储结构可以利用 cache 优化性能。但是在 GPU 并行环境，基于行优先的数据存储导致大量的非合并内存访问，降低内存带宽的利用率。在 GPU 并行环境上，通常建议采用基于列优先的数据存储结构，以满足 GPU 对访存优化的实际需求。在 4.2 小节，将提出应用于消息缓存的数据重映射算法，优化消息缓存的合并访存问题。

3.6 小结

本章主要阐述 PregelGPU 的系统设计，提出 Edge-Vertex 编程模型，缓解 GPU 线程负载不均衡的压力。并设计系统执行流程及 API，说明系统各个阶段的执行任务和逻辑功能。最后分析 Edge-Vertex 模型的性能瓶颈，在第 4 章提出相应的解决方案，优化系统性能。

第 4 章 PregelGPU 系统性能优化

针对 Edge-Vertex 编程模型的性能瓶颈，本章提出两个优化算法，提高 PregelGPU 运行时系统的性能。首先提出基于 GPU 的近似排序算法，解决在 VertexCompute 阶段同一个 warp 的 GPU 线程负载不均衡问题，并详细阐述算法核心思想和基于 GPU 并行环境的实现。然后分析消息缓存的数据存储方式，研究基于行优先存储和基于列优先存储的优劣，提出应用于消息缓存的数据重映射算法，将消息缓存的存储方式转化成列式存储，解决消息缓存的合并访存问题。

4.1 基于 GPU 的近似排序算法

4.1.1 GPU 精确排序

许多研究人员提出 GPU 排序算法，将排序算法的粗粒度并行转化成 GPU 细粒度并行。快速排序是著名的排序算法，Cederman 实现了基于 GPU 的快速排序算法，利用 GPU 加速排序过程[47]。Satish 研究了基于 GPU 的基数排序和归并排序，运用 GPU 的 shared memory 提高算法性能[48]。GPU 排序算法的研究主要集中在双调排序，快速排序，基准排序和归并排序。

这些 GPU 排序算法都属于精确排序。精准排序是指排序之后的序列严格升序或者降序。但是，一些现实应用并不需要严格的升序或者降序序列，允许某种程度的无序。这样，近似升序或者降序序列就满足这些应用的需要，而精确排序的开销相对比较大。基于 warp 的线程负载不均衡问题就属于此类应用。

4.1.2 背景

由于某些应用存在无规律的属性，引起 GPU 线程任务负载不均衡，进而导致无法充分利用 GPU 的计算资源。在第三章研究的 Edge-Vertex 编程模型，顶点从消息缓存读取消息，而顶点读取消息的数目取决于该顶点的入度。在输入图数据里，如果顶点的入度分布变化大，顶点线程负责顶点处理（比如读取该顶点消息），将导致同一个 warp 的线程任务负载不均衡（warp 是 GPU 线程调度的基本单位）。例如，如果顶点线程处理的消息量比较少，但是它的运行时间会被同一个 warp 的处理最多消息量的顶点线程“拖累”，整个 warp 的执行时间

是以处理时间最多的线程为基准。

在 1.2.1 小节讨论了 GPU 线程调度的内容，warp 是 GPU 线程调度的基本单位（一个 warp 包含 32 个线程）。现今的 GPU 并不支持细粒度的线程级任务调度。如果 GPU 线程负载不均衡，无法充分利用 GPU 计算资源。这就要求用户理解 GPU 硬件体系结构，微调算法，以达到算法性能最佳，这样加重了用户的负担。这种 GPU 线程负载不均衡问题也存在于其他应用中，最佳的解决方案是，存在一种优化算法，在负载不均衡任务上取得性能最佳，同时对于常规任务保证最小的时间开销。

为了解决这个难题，尝试不同的解决方案，探求其可行性和效率。一开始采用的方案是识别入度异常的顶点，然后把这些顶点推迟到随后的 kernel 执行。比如，如果顶点的消息过多或者过少，推迟计算这些顶点。但是这个方法的开销太大，实际意义不大，转向其他方案。同时由于顶点数目比较大，都达到百万级别的任务量，如果采用传统的调度算法解决此类负载均衡问题，引入的调度算法开销无法忽视。虽然现今的研究工作显示，在提高 GPU 处理网络数据包能力时，没有必要通过 GPU 排序把网络数据包大小相同或者相近的放置在一起。但是，通过实验发现，如果使用传统的 CPU 排序算法，把顶点按照顶点入度的属性进行排序，那么在 VertexCompute 阶段的性能可以得到较大的提升。但是因为 GPU 线程基于 warp 调度，并不要严格的全局有序，只需要保证同一个 warp 的 32 个线程任务不会相差太大，即同一个 warp 的 32 个线程任务负载不需要严格有序。根据以上需求，提出基于 GPU 的近似排序算法。

4.1.3 近似排序核心思想

对于一个待排序的数据序列，通过一次遍历，容易确定该序列的最大值 Max 和最小值 Min，进而得知所有数据的分布区间为[Min，Max]，区间长度为 Max-Min。如图 4.1 所示，对于输入数组(10，8，2，6，3，1)，其最大值 Max 为 10，最小值 Min 为 1，则该数组的数据分布在[1，10]这一区间内。相应地，区间长度为 Max-Min=9。

对于数据的分布区间[Min，Max]，可以将其划分为多个小区间，这样所有的数据都将分布到这些小区间中。然后通过一次遍历，可以将所有的数据映射存储到各个小区间。在映射后，小区间内的数据是按照遍历的先后顺序存入的，因此可能处于无序状态。但是区间之间却能够保证是有序的。如图 4.1 为例，把数据的分布区间划分为 3 个小区间，则小区间分别为[1，4)，[4，7)和[7，10]。按遍历顺序访问，原数组元素的(2，3，1)映射到区间[1,4)，(10，8，9)映射到区间[7，10]。这样映射完成后，映射到[4，7)的元素必然比[1，4)的元素大，而[7，

10]的元素必然比[4，7)的大。这样就使数据处于一种近似排序的状态。

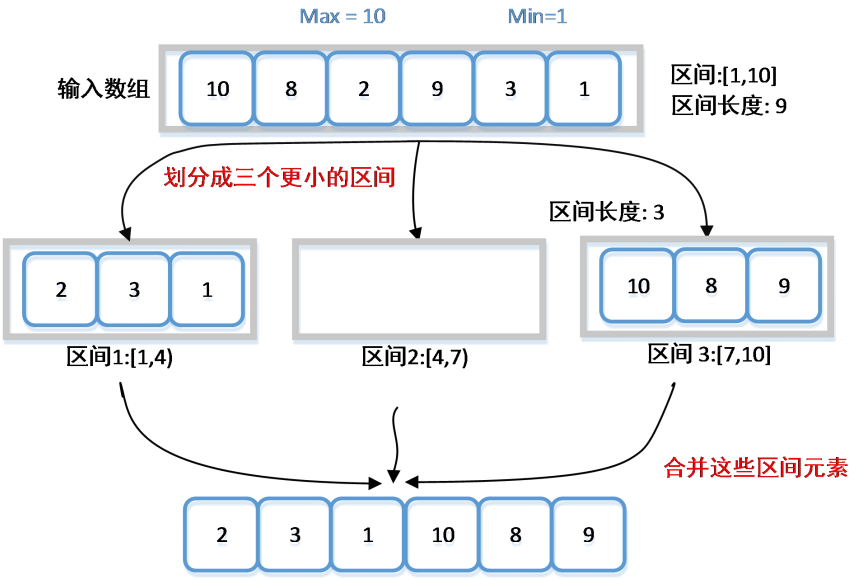


图 4.1 近似排序核心思想

在将所有的数据映射完成后，依次合并这些小区间，就得到了最终的近似排序结果。如果划分的小区间的个数为 Max-Min，则排序后的结果就是完全排序。

4.1.4 算法实现

根据近似排序的核心思想，利用 GPU 的并行计算能力，使用大量 GPU 线程，分别处理输入数据的不同部分，并将数据映射到各个区间，可以极大地提高近似排序的速度。为此，设计了一种基于 GPU 的近似排序算法。

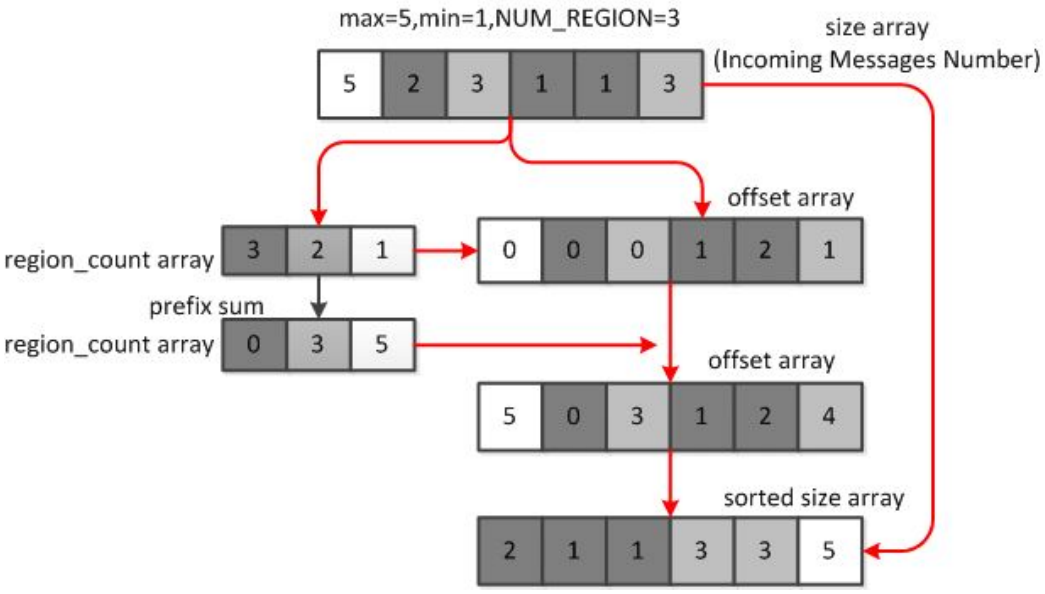


图 4.2 基于GPU的近似排序

如图 4.2 所示，把基于 GPU 的近似排序算法分为三个步骤。size 数组表示顶点的入度，即顶点接收消息的数目。首先 NUM_REGION 表示划分的区间的个数，使用 offset 数组记录每个元素在小区间的次序，并使用 region_count 数组记录映射到每个区间的元素个数。这样通过一次遍历，可以得到各元素进入小区间的次序（offset 数组），以及每个区间包含的元素个数（region_count 数组）。然后，通过计算 region_count 数组的前缀和(prefix sum)并将其存入 region_count 数组中，这一步骤的目的是确定各个小区间在输出数组中的起始位置。最后利用 region_count 数组记录的各区间起始位置，以及 offset 数组中记录的数据在小区间内的偏移位置，可以把数据近似有序地存入输出数组，最终完成排序。三个步骤的具体过程如下所示：

算法 4.1.1 元素映射过程

```
1: _global_ void assign_region(uint *input, uint *region_index, uint *offset,
2:                             uint *region_count, uint min, uint max, uint length)
3: {
4:     int idx=threadx.x+blockDim.x*blockIdx.x;
5:     uint region_idex;
6:     for(;idx<length; idx+=total_threads)
7:     {
8:         uint value=input[idx];
9:         region_idx=(value-min)*(NUM_REGION-1)/(max-min);
10:        region_index[idx]=region_idx;
11:        offset[idx]=atomicInc(&region_count[region_idx], length)
12:    }
13: }
```

步骤 1：与其他许多并行算法相似，基于 GPU 的近似排序算法，也是将输入数组划分成长度相等的小区间，然后并行地对这些小区间排序。所以，首先将输入数组的每个元素映射到相应的一个小区间。如算法 4.1.1 所示，小区间数目是一个固定值 NUM_REGION，元素映射过程是将输入数组的每一个元素线性地映射到这 NUM_REGION 个区间。线性映射的代码位于算法 4.1.1 的第 9、10 行，变量 min 和 max 分别表示输入数组的最大元素值和最小元素值，可以通过基于 GPU 的 CUDPP 库[49]中 reduce 函数求得。这样，每个小区间是区间[min，max]中的一小部分，同时所有小区间都有相同的区间长度。特别地，数组 region_count 记录了落入每个小区间的元素数目，数组 offset 记录了元素进入相应小区间的次序。在算法 4.1.1 第 11 行代码，使用了基于 CUDA 提供的原子操作函数（atomicInc），这个原子操作可以避免由并发访问造成写冲突。函数 atomicInc 完成原子性自加功能，利用这个函数记录每个元素进入相应小区间的次序。和 Fermi 体系结构的 GPU 相比，Kepler 体系结构的 GPU 已经显著地优化了全局存储器的原子操作,这一点可以在基于 GPU 的近似排序实现中看到。

步骤 2：通过步骤 1 已经得到了每个小区间元素数目的数组 region_count 和元素进入相应小区间的次序的数组 offset，然后对 region_count 数组执行前缀和操作（prefix sum）[50]，并将其存入 region_count 中，这样可以确定每个小区间在输出数组的开始位置。前缀和就是根据已有的数组，生成一个新数组，新数组中的第 i 个元素是原数组第 1 到 i-1 个元素之和。因为数组 region_count 的长度与区间数相同，远小于输入数据的长度。所以 CPU 执行前缀和要比 GPU 快。但是，由于数据在显存与内存之间的传输开销较大。同时，若把 GPU 的运行代码和 CPU 的运行代码混合运行的话，会导致排序算法性能的急剧下降。为此，在具体的代码实现中，本文使用基于 GPU 的 CUDPP 库来计算前缀和。

步骤 3：如算法 4.1.2 第 11-14 行代码所示，通过步骤 1 生成的元素在小区间偏移位置的 offset 数组和步骤 2 得到的存储各小区间起始位置的 region_count 数组，可以很容易计算出元素在输出数组的位置（如算法 4.1.2 第 15-16 行），进而完成近似排序。通过近似排序，在同一个 warp 内处理的消息量相差不太，缓解在 VertexCompute 阶段同一个 warp 的线程负载不均衡的问题。但是排序后会出现一个副作用，无法通过边访问边的源顶点，因为顶点的位置经过排序已经发生变化。所以使用 oldToNew 数组记录旧顶点 ID 和新顶点 ID 之间的映射关系。利用这个数组，需要把系统所有使用旧顶点 ID 的数据替换成新顶点 ID。

算法 4.1.2 基于 GPU 的近似排序

```
1: _global_ void   appr_sort(uint *key, uint *key_sorted, uint *oldToNew,
2:                    uint *region_count, uint *region_index, uint *offset,
3:                    void *value, void *value_sorted, uint length)
4: {
5:     int idx=threadx.x+blockDim.x*blockIdx.x;
6:     uint count=0;
7:     for(;idx<length; idx+=total_threads)
8:     {
9:         uint tKey=key[idx];
10:        uint tValue=value[idx];
11:        uint tRegion_index=region_index[idx];
12:        count=region_count[tRegion_index];
13:        uint off=offset[idx];
14:        off=off+count;
15:        key_sorted[offset]=tKey;
16:        value_sort[off]=tValue;
17:        oldToNew[idx]=off;
18:    }
19: }
```

在以上的过程中，选择一个合适的区间数目，对于基于 GPU 的近似排序算法影响比较大。随着区间数目的增加，近似排序算法就越来越接近完全排序，但是前缀和的计算开销也相应地增大。如果区间数目不断减少，则会得到一个

粗密度的近似有序序列。但是，数据进入同一区间的概率会增加，导致并发写冲突增大，进而增加运行时间。同时，基于 warp 的线程负载均衡并不需要细粒度的近似有序序列，本论文将在第五章节进行相关实验分析。

4.2 数据重映射算法

在传统的 Pregel 系统架构中，消息是发送到消息缓存，该消息缓存由运行时系统分配和管理，对用户抽象实现细节。在第 2.2.2 小节分析了 Medusa 的消息缓存机制是基于数组的实现，而基于数组的消息缓存在数据存储方面，主要分为两大类：行优先数据存储（row-major data layout）和列优先数据存储（column-major data layout）[51]。如图 4.3 所示，这两种数据存储方式都可以用一维数组表示，间接地利用物理上的一维数组表示逻辑上的二维数组。但是，GPU 线程在这两种数据存储的访存模式上完全不一样。在行优先的存储结构中，如图 4.3(a)所示，同一个线程所处理的数据是存放在相邻的位置，不同线程处理不连续的数据，然而在列优先的存储结构中，则是由不同的线程来处理连续的数据。在 GPU 并行环境，同一个 warp 内的 GPU 线程访问连续的地址空间，GPU 硬件把这些访问请求将会合并成一个内存访存事务。否则，产生非合并访问请求，即生成多个不同的访存事务。所以，列式优先存储可以利用 GPU 全局内存合并访问的特性，提高内存带宽。而在行式优先存储，不同 GPU 线程处理的数据不连续，违背 GPU 内存合并访问原则，导致大量非合并访存。

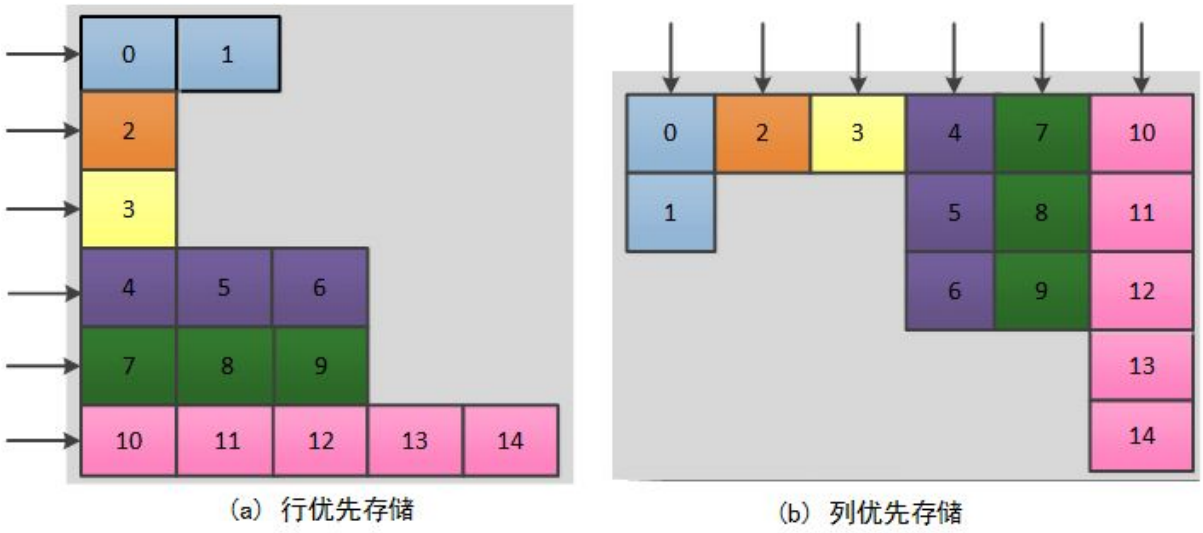


图 4.3 两种数据存储的访问方式

Medusa 系统采用基于数组的缓存机制，实现消息的存储和读取功能。在 Medusa 系统，首先根据需要输入图的边总数，预分配一个固定大小的消息缓存。然后在 CPU 环境下计算，确定每条边发送消息到消息缓存的位置和每个顶点从

消息缓存接收消息的位置。通过这种预先确定消息在消息缓存的读写位置方法，可以避免动态分配内存和原子操作，提高 GPU 多线程的执行效率。如图 4.4 所示，Medusa 系统的消息缓存机制确保每个顶点接收的消息都处于相邻的位置，这就相当于消息缓存是按照行优先存储的，导致 GPU 线程在读取顶点消息的时候，无法利用 GPU 合并访存的特性，产生大量的非合并访存，降低 GPU 全局内存带宽的使用率。

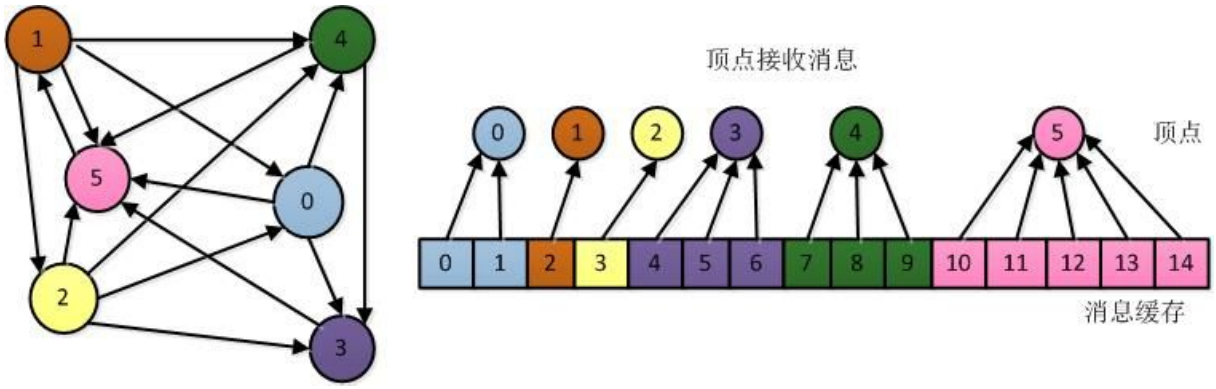


图 4.4 Medusa系统消息缓存机制

4.2.1 背景

尽管已经有很多文献采用简单的 one-vertex-per-thread 模型，即一个线程负责处理一个顶点。但是在处理消息缓存的消息数据包时，存在潜在的性能问题。通过上述分析，某些特定的应用（比如线程读取顶点消息的过程），为了利用相关硬件的体系结构特性，在 CPU 和 GPU 环境通常采用不同的数据存储方式优化性能。在 CPU 计算环境，一般采用行式优先存储，因为可以利用局部性原理和 cache 提高性能。但是，在 GPU 并行环境，通常应该采用列式优先存储，因为利用 GPU 合并访存特性，同一个 warp 的线程访问连续的存储区域的请求可能被合并成一个访问事务。本小节提出应用于消息缓存的数据映射算法，充分利用 GPU 内存合并访问特性，尽量减少内存非合并访问次数。

由于 CPU 和 GPU 通常采用不同的数据存储方式优化性能，但是在现今的 GPU 并行平台上并不存在这样轻量级算法转换这两种数据存储方式。因此，在下一小节提出基于 GPU 的轻量级数据重映射算法，把消息缓存的行式优先存储转换成列式优先存储，优化消息缓存的合并访问，同时保证较低的额外内存消耗。

4.2.2 基于 GPU 的数据重映射算法

如图 4.3 所示，对于行优先的数据存储，GPU 线程迭代地从头到尾扫描同一

个顶点的消息包，这种访问方式违反 GPU 全局内存合并访问特性，导致大量的非合并访存。然而，在一个数组中，实现严格的列优先数据分布，会消耗大量的额外存储空间。由于只需要保证同一个 warp 的线程都访问连续的存储区域，就满足 GPU 内存合并访问的要求，所以没有必要采用严格的列优先存储保证所有 GPU 线程都访问连续的存储区域。

本论文提出基于 GPU 的数据重映射算法，其核心的思想是，将顶点以 warp 为单位分组，组内的顶点消息采用列优先存储，而组与组之间的顶点消息则采用行式优先存储，这样既能保证同一个 warp 的顶点线程合并访问，同时又可以降低 GPU 内存的额外消耗。下面使用三个步骤详细说明基于 GPU 的数据重映射算法。为了简单起见，假设每 3 个顶点划分为一组。

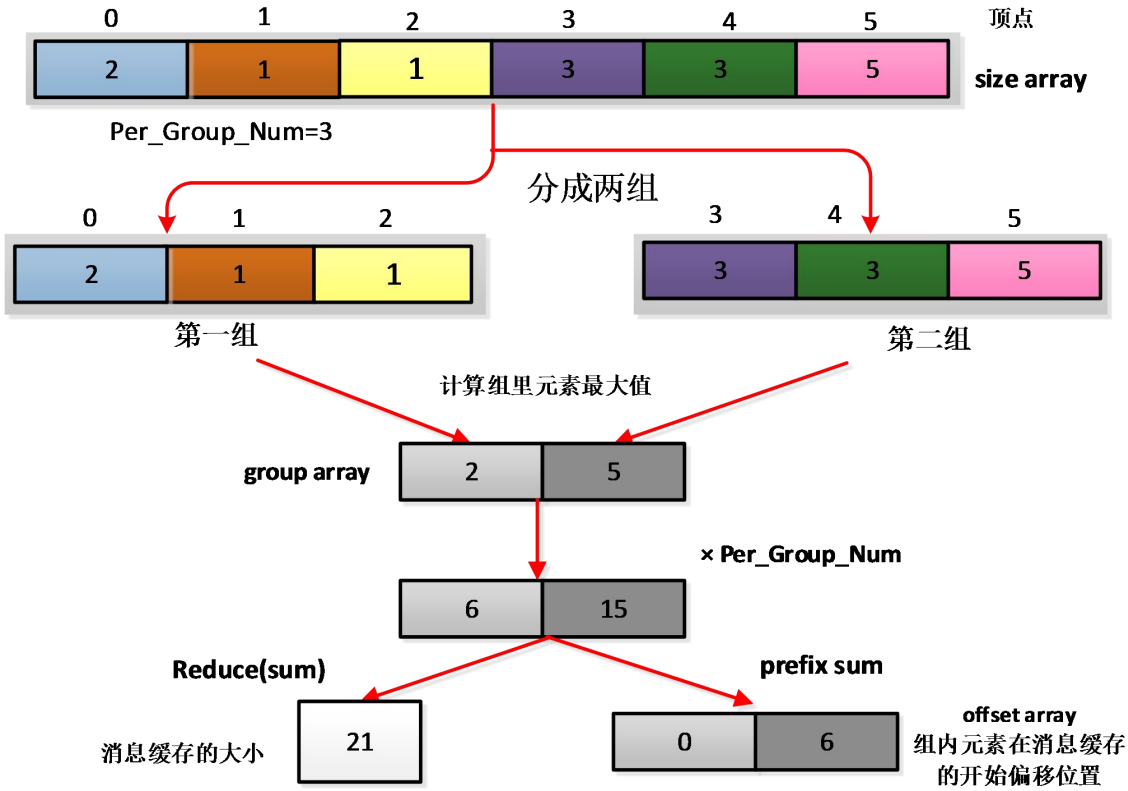


图 4.5 基于GPU的数据重映射算法（步骤1-2）

第一步，size 数组表示顶点的入度，即顶点接收消息的数目。类似于近似排序算法将 size 数组的元素映射到不同的区间内，首先将 size 数组划分成不同的组，每组内含有相同的元素个数。如图 4.5 所示，每个组内的元素的个数是一个常数 Per_Group_Num，为了简单地阐述算法过程，这里把该常数设置为 3。利用 $\lceil length(size)/Per\_Group\_Num \rceil$ 这个公式，计算得到分组的个数，并使用组内的最大值构建 group 数组，group 数组表示每个组内顶点接收消息的最大容量。这些计算可以在 GPU 上实现，从而避免在 CPU 与 GPU 间的数据传输开销。每个顶点的消息最大容量由该顶点所属组的最大顶点入度决定。

第二步，基于数组的缓存还需要计算消息缓存分配的内存大小，以及为分组构建 offset 数组。具体而言，某个顶点的接收消息最大容量大小，由它所隶属的组以及在 group 数组中相应的值所决定。group 数组中的每个元素都被乘以 Per_Group_Num，这样就产生了一个新数组。新数组中的每个元素表示每个组所需要分配的内存大小。通过 reduce 函数求和，获得消息缓存的总存储区域大小。然后利用前缀和，求得各个分组在消息缓存的开始偏移位置（offset 数组）。并行化的 reduce 和前缀和函数可以很容易的在 GPU 环境高效地实现，这里同样使用 CUDPP 库来实现算法的第二步过程。

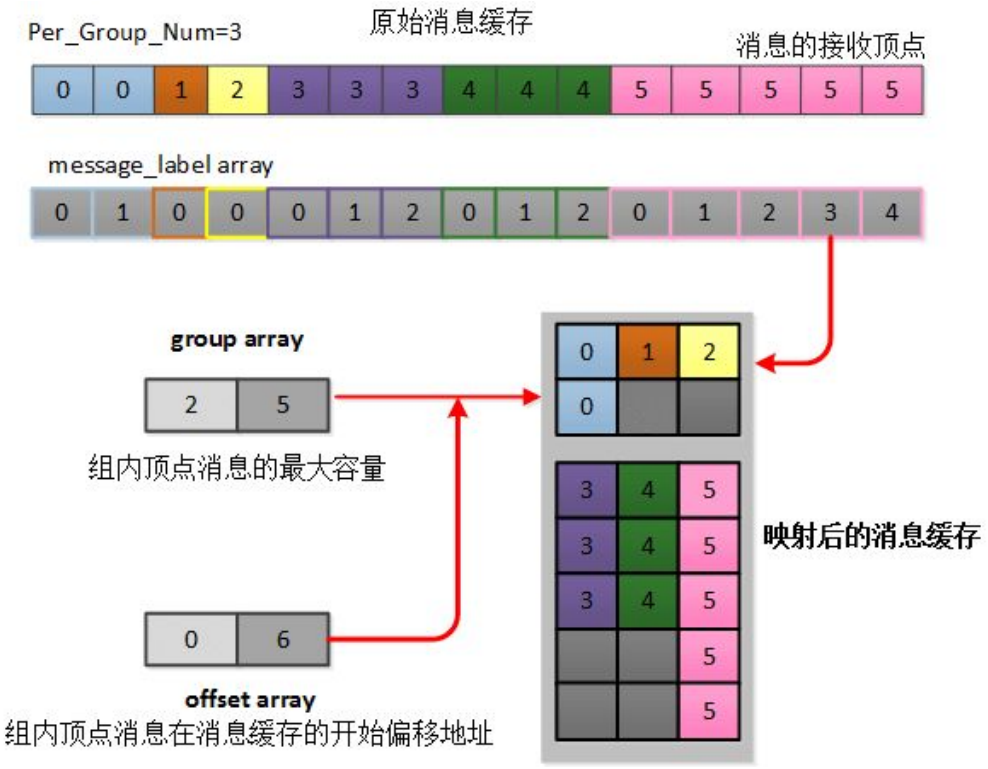


图 4.6 基于GPU的数据重映射算法（步骤3）

第三步，算法的最终目的是在原始消息缓存上执行数据重映射操作。如图 4.6 的描述，accept_vertex 和 message_label 两个数组是原始消息缓存的两个属性，用来帮助消息进行映射的辅助信息。accept_vertex 数组记录消息要被哪个顶点接收，message_label 记录消息缓存中消息在每个区段（即每个顶点消息包）的内部序号。在图 4.6 中说明数据重映射的关键过程，但是值得注意真实的数据重映射算法是执行在消息缓存的消息上，而不是在 accept_vertex 属性数组上。在这一步中，需要使用到第二步计算生成的 offset 数组。如算法所示，首先在重映射之前，系统需要获得消息的接收顶点（accept_vertex 数组内容），以及存储在 message_label 数组的消息内部序号（参考算法 4.2.1 第 8-9 行）。其次，需要确定顶点所属的组，以及由 accept_vertex%Per_Group_Num 得到消息的顶点组内序

号（参考算法 4.2.1 第 11-12 行）。最后计算得到数据重映射的新位置，即列优先存储的新位置（参考算法 4.2.1 第 13-14 行），把消息缓存的内容直接拷贝到新的位置（参考算法第 15 行）。在图 4-6 展示了数据重映射算法后，消息缓存的物理结构。每 Per_Group_Num 个线程分别读取顶点的消息，这些线程访问的消息存储单元是连续的。在 GPU 并行环境下，通常把 Per_Group_Num 设置为 32，即同一个 warp 的 32 个线程读取连续存储单元的消息，这时候达到 GPU 合并访存的要求。

算法 4.2.1　数据重映射

```
1:  _global_ void   remap(int *accept_vertex_array, int *message_label_array,
2:                        int *message_content, int *remap_message_content,
3:                        int *offset_array, int *group_array, int length)
4: {
5:      int idx=threadx.x+blockDim.x*blockIdx.x;
6:      for(;idx<length; idx+=total_threads)
7:      {
8:          int accept_vertex=accept_vertex_array[idx];
9:          int message_label=message_label_array[idx];
10:         int group_id=accept_vertex/Per_Group_Num;
11:         int group_inner_id=accept_vertex % Per_Group_Num;
12:         int offset_group=offset_array[group_id];
13:         int new_position=offset_group+group_inner_id;
14:         new_position +=message_label*Per_Group_Num;
15:         remap_message_content[new_position]=message_content[idx];
16:     }
17: }
```

尽管以上的过程涉及到多个 GPU kernel 的调用，但是由于重映射算法的轻量级设计以及对 GPU 并行计算资源的充分挖掘，该方法未导致明显的性能开销。数据重映射的收益是以增加 GPU 全局内存的使用率为代价。然而，可以通过设置 Per_Group_Num 参数的值，来控制额外存储空间消耗。例如，当 Per_Group_Num=1 时，数据的存储方式保持原样，不需要消耗额外的存储空间。因此，为 Per_Group_Num 选择一个合适的值，对于本文的数据重映射算法的效率有着重要的影响。随着 Per_Group_Num 的增加，对于全局内存的合并访问更加有效，但是内存的消耗会相应的随着增加。由于 GPU 线程以 warp 为单位访存，出于性能的考虑，把 Per_Group_Num 设置为 32。另外，更详细的关于性能和存储空间开销的分析将会在第 5 章进行分析。

本小节的数据重映射算法是在近似排序之后执行的，因为近似排序后，同一组内的顶点所接收的消息大致相同，这样可以减少内存空间的消耗。在 3.2.1 节讨论了预处理阶段，系统需要确定消息缓存的大小，以及边消息发送和顶点

接收消息的位置信息。由于，数据重映射算法也可以做这些工作，所以把其集成在预处理阶段，那么不需要在每一个超步都对消息缓存进行重映射，消息缓存直接按照列式存储，不需要行式存储进行过渡。

4.3 小结

本章是 PregelGPU 运行时系统的性能优化部分，针对 Edge-Vertex 编程模型潜在的瓶颈，进一步提出性能优化方案。由于在 VertexCompute 阶段，GPU 线程存在负载不均衡问题，根据 GPU 线程以 warp 为单元调度，提出基于 GPU 的近似排序算法，缓解同一个 warp 的线程负载不均衡的压力。研究消息缓存的数据存储方式，发现不同的数据存储方式对系统性能有影响。Medusa 消息缓存机制采用行式存储，无法利用 GPU 的合并访存特性，提出应用于消息缓存的数据重映射算法，对行式存储进行数据映射。由于数据重映射算法同样可以计算消息缓存的大小，边消息发送和顶点接收消息的位置，所以作为优化，把数据重映射算法集成在预处理阶段，消息缓存直接按照列式存储，不需要每一个超步都进行数据重映射。

第 5 章 实验结果与分析

本文实现了基于 GPU 的图计算系统—PregelGPU。本章将从四个方面对 PregelGPU 系统进行实验验证。首先，使用常见的图算法测试 PregelGPU 系统的总体性能，并与 2.2 小节的 Medusa 系统进行性能对比。然后，研究系统预处理阶段的效率。接着对本文提出的基于 GPU 的近似排序算法进行实验评估。最后分析数据重映射算法对图计算的影响。

5.1 实验准备工作

在实验数据方面，分别用真实的图数据集和人工生成的图数据集进行实验测试。表 5.1 总结了测试数据集的相关特征，其中这六个测试数据集都是有向图。为了更准确地分析实验结果，d 表示顶点的入度，则 Max(d)表示图中顶点入度的最大值。同样，代表顶点入度大于 1 的个数，并用η表示顶点入度大于 1 的顶点所占的比重。ego-Gplus[52]数据集来自 Google+的社交圈数据集，其η为 99.9%，说明在这个图中几乎所有顶点的入度都大于 1。soc-Pokec[53]和 com-LiveJournal[54]数据集都是社交网络类型的数据，这两个数据集的η几乎一样（76%）。cit-Patents[55]数据集是美国国家经济研究局公开的数据集，其中η比较小，表示这个图有 38.1%的顶点，其顶点入度小于 2。以上图数据都是真实的图数据集，并且符合幂律度分布（power law degree distribution）。

表 5.1 测试数据集

数据集	简称	顶点数	边数	Max(d)	η
ego-Gplus	ego	107,614	13,673,453	17058	99.9%
soc-Pokec	soc	1,632,803	30,633,564	6808	76.3%
com-LiveJournal	com	3,997,962	34,681,189	2454	76.7%
cit-Patents	cit	3,774,768	16,518,948	779	61.9%
RMAT	RMAT	1,000,000	16,000,000	555	52.4%
Random	Rnd	1,000,000	16,000,000	41	99.9%

RMAT 和 Random 是通过 Gtgraph[56]生成两种不同分布的图数据集。RMAT 数据集符合幂率度分布，并且η是 52.4%，即几乎有一半的顶点入度小于 2。Random 数据集满足正态分布特点，它的 Max(d)比其他的数据集低很多。这两种生成数据集的顶点平均度数 16。由于本文主要研究 GPU 线程之间的负载均衡，以及 GPU 合并访存的优化问题，实验所选择的数据集主要是符合幂律度分布，

即存在顶点任务不均衡问题。

在表5.2说明了本论文的硬件实验平台。实验中的工作站配备Xeon E5 2648L CPU，具有8GB的DDR3内存，并且装备了具有12个多处理器的GeForce GTX 780GPU,每个处理器又包含16个处理核心,同时该GPU具有3GB大小的DDR5内存。

表 5.2 硬件实验平台

硬件参数	Intel Xeon E5 2648L	GeForce GTX 780
核数/处理器	8	12
硬件线程数/核数	2	192
频率（GHz）	1.8	0.9
存储（GB）	8G DDR3	3G DDR5
存储带宽（GB/s）	57.6	288.4

5.2 总体性能评估

分别使用PregelGPU API和Medusa API实现三个具有代表性的图算法，并且以2.2小节的Medusa系统为比较基准，对本系统的性能进行比较和分析。在这小节主要比较系统运行的总时间。系统运行的总时间包括预处理时间和系统图计算时间，这两个评估指标将在5.3小节和5.4小节进行分析。实验测试的三个图算法分别是：PageRank，SSSP算法和HCC算法。PageRank主要是根据图的拓扑结构计算每一个顶点的PageRank值。SSSP算法是用于寻找单个顶点到其他所有顶点的最短路径。HCC算法则是查找图中的各个连通分量。

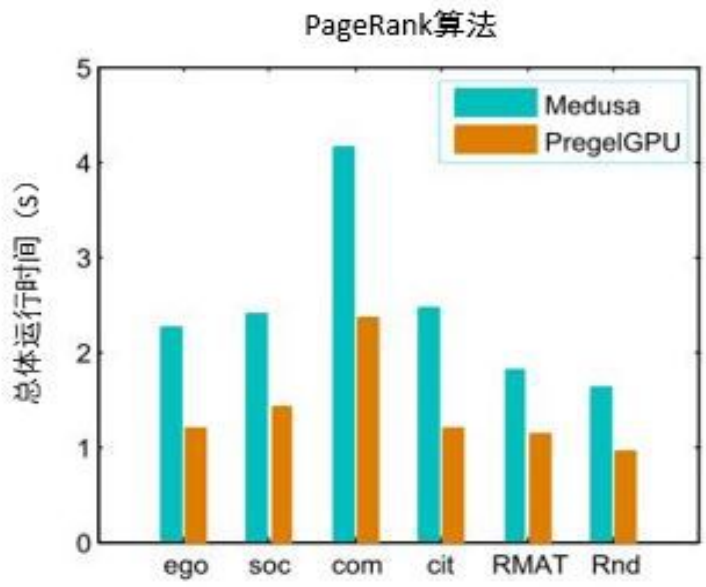


图 5.1 PageRank在两种系统的执行时间对比

图 5.1 展示了在不同数据集上，PageRank 算法在两种平台上的执行时间对比。在这个实验中，设置 PageRank 算法迭代次数为 50 次。通过图 5.1 可以看到，和 Medusa 系统相比，PageRank 在 PregelGPU 系统可以达到 1.7 到 2.3 倍的加速比。

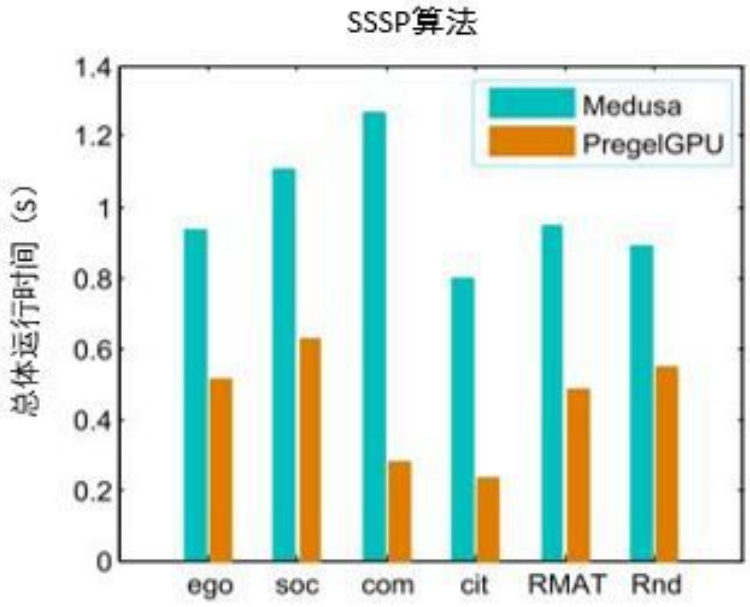


图 5.2 SSSP算法在两种系统的执行时间对比

在测试 SSSP 算法的时候，随机地选择一个顶点作为源顶点，并计算此节点到其他所有顶点的最短路径。图 5.2 展示了 SSSP 算法在六种数据集上的执行时间。通过观察发现，SSSP 算法在本系统中的运行时间相较于 Medusa 系统提高 1.6 到 4.5 倍。

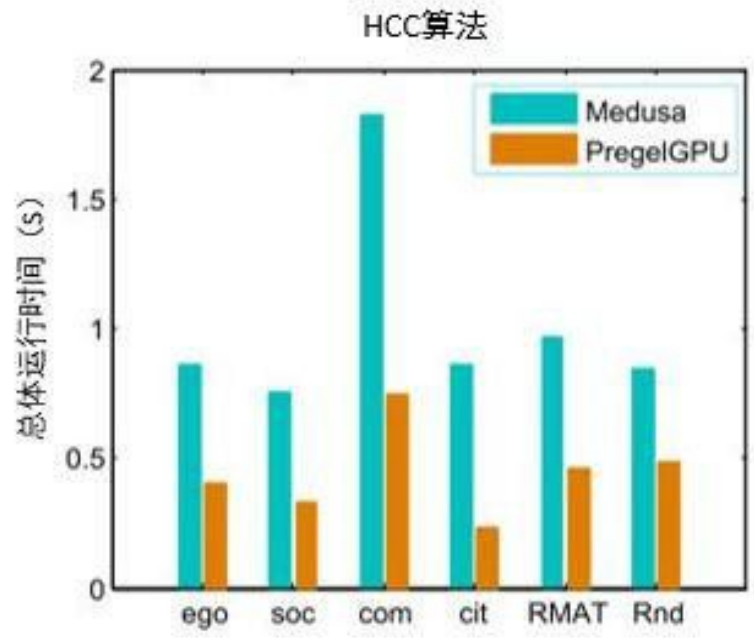


图 5.3 HCC算法在两种系统的执行时间对比

如图 5.3 所示，可以看到 HCC 算法在两种系统中的性能表现，同样和 Medusa 系统相比，PregelGPU 的执行性能可以达到 1.7 到 3.7 的加速比。通过以上三组的实验对比，我们发现，相对于 Medusa 系统，本系统可以获得大约 1.6 到 4.5 倍的性能优势。

5.3 预处理性能分析

本小节将对预处理阶段的性能开销进行测试分析。在本系统和 Medusa 系统中，都是由边负责把消息发送给消息缓存，然后顶点读取消息缓存中的消息并进行相关计算。因此，需要预先确定边把消息发送到消息缓存的位置和顶点从消息缓存读取消息的位置。Medusa 系统提出了一种图感知的消息缓存机制（Graph aware buffer scheme）来确定这些位置。然而，这种该机制是运行于在 CPU 环境，不能充分利用 GPU 的并行计算能力。因为本文提出的数据重映射算法同样可以计算这些位置信息，所以把数据重映射算法集成在预处理阶段，这样就避免了每一个超步都要进行数据重映射过程，只需要在预处理阶段进行一次位置计算即可。同时，为了减少数据重映射算法的额外空间的浪费，近似排序算法要执行在数据重映射算法之前，所以本文的预处理阶段集成了近似排序算法和数据重映射算法。与之相比，本文的预处理阶段，全部运行在 GPU 上。

表 5.3 预处理开销对比(秒)

数据集	本系统	Medusa	加速比
ego-Gplus	0.016	0.21	13
soc-Pokec	0.020	0.449	22
com-LiveJournal	0.030	1.002	33
cit-Patents	0.012	0.665	55
RMAT	0.012	0.305	25
Random	0.011	0.328	30

如表 5.3 所示，可以观察到本系统在预处理阶段的性能相对于 Medusa 系统非常优越。这是因为本系统的预处理过程移植到了 GPU 并行环境，充分利用了 GPU 的并行处理能力。在这个实验测试中，把参数 Per_Group_Num 设置为 32，所获得的性能加速比约为 10 倍到 50 倍。一个非常显著的特点是，在不同拓扑状态的图数据集上，数据重映射算法的开销始终保持稳定。例如，在 ego-Gplus 数据集上的开销，与 soc-Pokec 的开销基本一样，但是后者的顶点数量是前者的 16 倍，而边的数量是前者的 2.2 倍。相反，Medusa 预处理的开销则与图的规模紧密相关，当图中节点和边的数量增加时，开销也随之增加。

5.4 近似排序测试与分析

本节对本文提出的近似排序算法进行实验分析。在实验中，主要研究两个基本问题：（1）通过排序到底可以提高多少系统性能；（2）近似排序和基于 GPU 的精确排序算法比较，近似排序到底有多快。在图 5.4 中，可以看到未排序的图数据和排序后的图数据对图计算时间的影响。在该实验测试中，使用 PageRank 算法迭代 50 次进行测试，可以发现，排序后的 PageRank 的执行效率提升了 8%到 20%。并且对于符合幂率度分布的图数据集，这种性能提升更加明显。

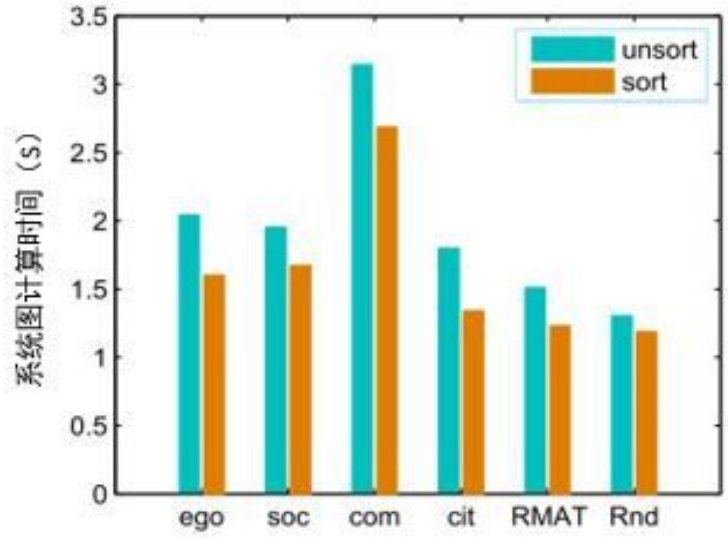


图 5.4 排序对图计算执行时间的影响

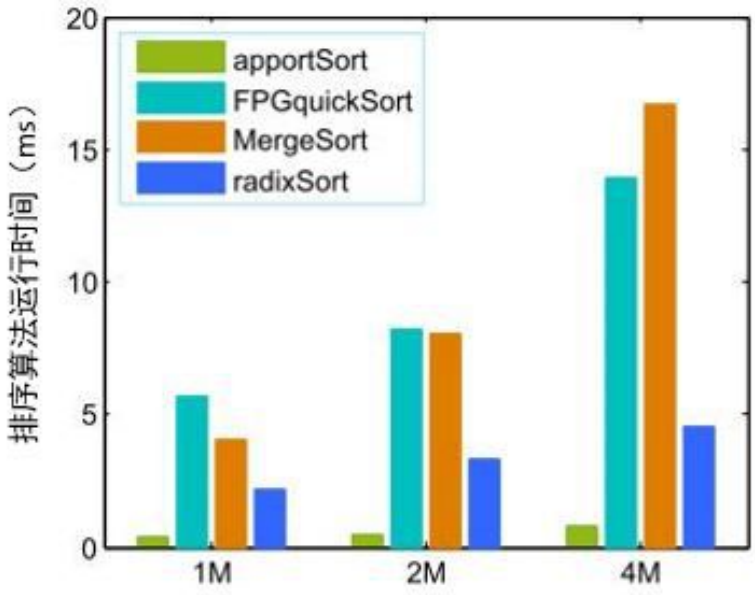


图 5.5 近似排序与其他排序算法的性能比较

然而，同样需要关注带来这种性能提升所产生的额外开销，也即排序开销。目前存在很多基于 GPU 的排序算法，把本文提出的近似排序算法与三种常见的 GPU 排序算法进行比较（FPGquick sorting，merge sorting 和 radix sorting）。假设实验中的所有数据都符合正态分布。并准备了三份数据集，其大小分别为 1M，2M 和 4M（M 表示 10^6），同时把近似排序中的 BUCKET_NUMBER 设置为 1000。如图 5.5 所示，本文提出的近似排序方法在运行时间上具有明显的优势。在最好的情况下，其执行速度是 merge sorting 的 21.7 倍，最坏情况下也达到了 6.1 倍。值得注意的是，近似排序算法的执行时间，随着数据集规模的增大，增长非常缓慢。而其他排序算法则会随之快速增加，尤其是 FPGquick sorting 算法和 merge sorting 算法。

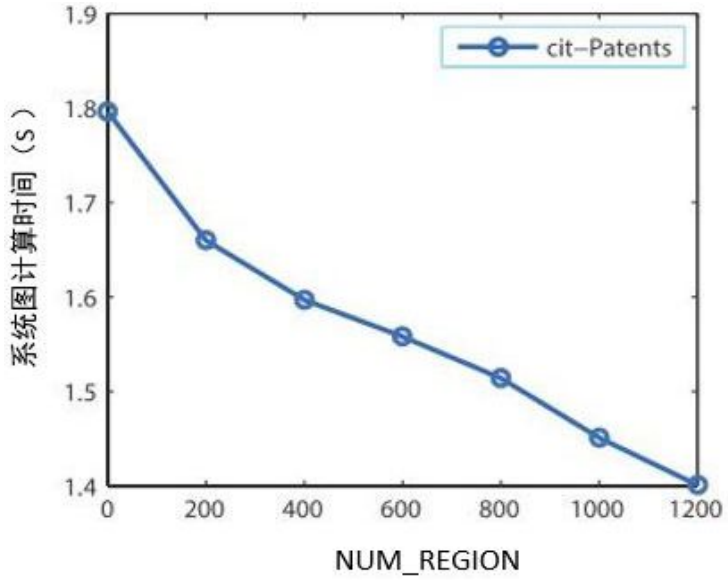


图 5.6 NUM_REGION对执行时间的影响

接着继续研究 NUM_REGION 对系统图计算时间影响。如图 5.6 所示，使用 cit-Patents 数据集作为无规律图计算任务的代表，运行 PageRank 算法迭代 50 次，分别在不同的 NUM_REGION 值下进行测试，观察系统图计算时间。当 NUM_REGION=0 时，表示不会对数据进行排序。而随着 NUM_REGION 的数量增长，系统图计算时间随之下降。这是因为 NUM_REGION 越大，使得数据更加近似有序，缓解了同一个 warp 的 GPU 线程任务负载不均衡的压力，进而加快了 PageRank 的运行速度。

5.5 数据重映射算法性能分析

GPU 线程访问全局内存，如果同一个 warp 的线程访问连续的存储单元，将会合并成一个访存事务，提高带宽的利用率。在本小节研究数据重映射算法对系统图计算时间的影响。如图 5.7 所示，再次使用 PageRank 算法迭代 50 次进行实验对比分析。map 实验组表示在预处理阶段使用了数据重映射算法，消息缓存

以列优先的方式进行存储。作为对比实验，non_map 实验组采用 Medusa 消息缓存机制，即消息缓存以行优先的方式进行存储。一个 warp 包含 32 个线程，所以在 map 实验组把 Per_Group_Num 设置为 32。

如图 5.7 所示，在预处理阶段使用数据分布重映射算法之后，消息缓存按照列式优先方式进行存储，PageRank 算法在不同的数据集上的执行效率，提高了 10%到 25%。另一个值得关注的现象是，对于符合幂率分布的数据集，效率的提升与η的大小紧密相关。对于η较小的数据集，性能提升的效果较小。但是当η较大时，则性能的提升非常明显。这是因为当η较大时，说明顶点入度大于 1 的顶点越多，消息数目大于 1 的顶点越多，那么同一个 warp 的线程读取相邻位置消息的机会越多。例如，对于具有最大η值的 ego 数据集，其性能提升达到了 25%。

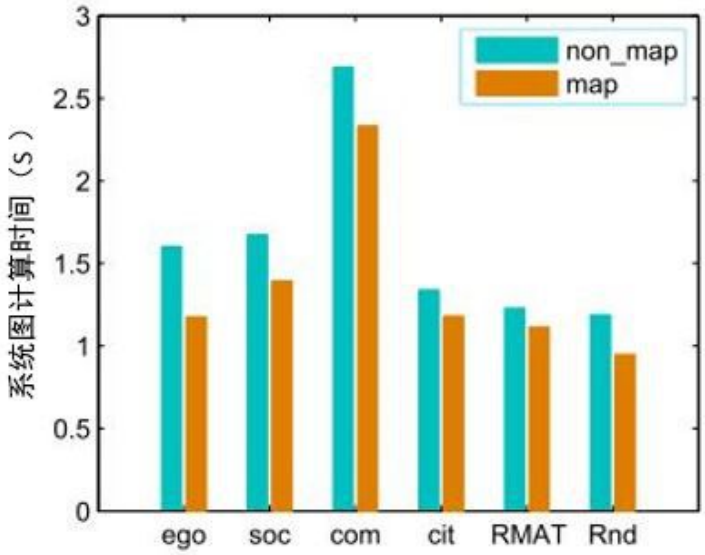


图 5.7 数据重映射对性能的影响

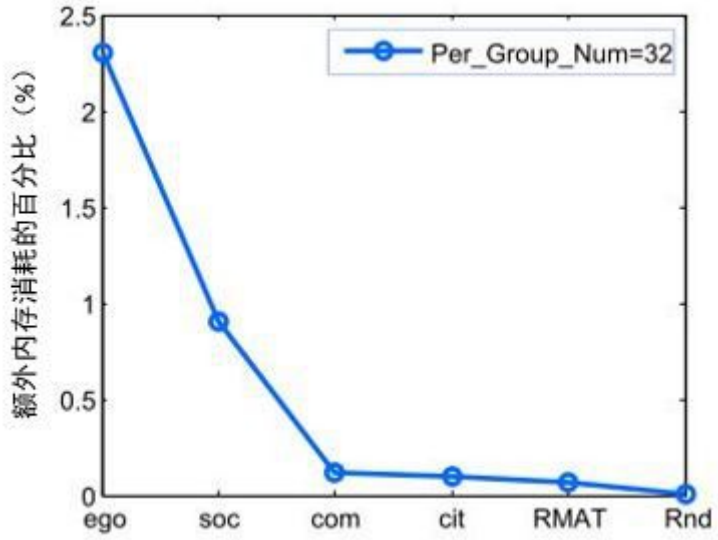


图 5.8 数据重映射算法消耗的额外内存

另外，继续分析由于数据重映射算法把消息缓存转化成列优先存储，导致

额外的内存消耗，这里的参照对象是行优先存储的空间大小。总体来说，实现数据重映射算法之后，列优先存储的额外内存消耗是非常有限的，其与 Max(d)的值密切关联。在图 5.8 所示，ego 数据集的 Max(d)=17058,需要额外消耗 2.3%的内存。而对于 soc 数据集，其 Max(d)=6808，额外消耗的内存则降低到 0.89%。对于符合正态分布的 Rnd 数据集，其内存开销几乎可以忽略不计。

5.5 小结

本章首先详细说明了实验测试的软硬件环境，为了充分验证系统性能，数据集涵盖真实的数据集和人工生成的数据集，并且定义 Max(d)和η描述数据集的特征。然后分别从系统总体性能评估，预处理性能分析，近似排序测试与分析，数据重映射算法性能研究四个方面对本图计算系统进行实验测试与分析，并与 Medusa 系统进行对比研究。相对于 Medusa 系统，本系统可以获得大约 1.6 到 4.5 倍的性能优势。

本章的内容说明本文提出的图计算系统，系统性能可以获得大幅度的提升。首先，本系统的预处理过程移植到了 GPU 并行环境，充分利用了 GPU 的并行处理能力。其次，基于 GPU 的近似排序缓解了同一个 warp 的线程负载不均衡压力，同时排序引入很小开销。最后，数据重映射算法使得消息缓存按照列优先的方式存储，减少了 GPU 线程非合并访存的次数，同时额外内存消耗几乎可以忽略不计。

结论

随着社交网络和大数据爆炸式增长，基于大图数据的应用逐渐增多。与传统图算法处理的数据量相比，这些图数据都是海量的，由几千万个顶点和几亿条边组成，在单机环境无法高效地处理如此庞大的数据集。研究人员已证明 MapReduc 不适合处理图数据集。Google 提出了 Pregel 图计算系统，解决关于大图数据的分布式计算问题。Pregel 编程模型为用户抽象分布式计算的实现细节，提供了简明的 API 实现图算法。

随着 GPU 软硬件技术的发展，GPU 逐渐演变成为高度并行的多线程及多核处理器，同时具有强大的计算能力和超高的内存带宽。但是，GPU 在线程组织结构、线程管理与调度、线程访存模式、shared memory 等都与 CPU 截然不同。尽管如此，越来越多的图算法使用 GPU 来提升计算性能，并利用 GPU 的硬件特性优化图算法。

但是目前大部分基于 GPU 的图计算研究都是针对特定的图算法，缺乏基于 GPU 的类 Pregel 系统，为图算法提供基于 GPU 的通用图计算系统。而且当前的系统尚未根据图算法特点和 GPU 硬件特性，对系统性能进行优化。

本文在讨论 GPU 硬件结构和图计算研究进展的基础上，重点探究 Pregel 系统和 Medusa 系统的执行模型和编程模型，并分析这两个系统的优缺点。针对 Medusa 系统的缺陷和性能瓶颈，结合 GPU 的体系结构特性，提出基于 GPU 的图计算解决方案。首先改进以顶点为中心的编程模型，阐述基于细粒度并行的 Edge-Vertex 编程模型及 API，为基于 GPU 的图算法提供通用的编程接口。其次，提出基于 GPU 的近似排序算法，缓解同一个 warp 的线程负载不均衡问题。最后，分析消息缓存的数据存储方式，提出数据重映射算法把行优先存储映射成列优先存储，增加 GPU 线程合并访存的次数，提高全局内存带宽的利用率。为了进一步优化运行时系统性能，把数据重映射算法集成在系统预处理阶段，充分利用 GPU 并行资源，完成消息发送和读取的位置计算。实验证明，通过以上优化算法，相对于 Medusa 系统，本系统可以获得大约 1.6 到 4.5 倍的性能优势。本文的主要工作包括以下几个方面：

1.对 GPU 体系结构进行分析，总结了 GPU 线程调度和 GPU 性能优化方式。从并行体系结构方面概括图算法的研究进展，并从研究内容方面对图计算系统做了简要分析。

2.研究了图计算系统相关知识，重点阐述 Pregel 执行模型、Pregel 编程模型、Pregel API、Pregel 系统构架和 Pregel 性能瓶颈这五方面。分析基于 GPU 的图计

算系统 Medusa，主要的研究内容集中在 EMV 编程模型及 API 和 Medusa 消息缓存机制，总结 Medusa 的缺陷和性能瓶颈。

3.改进传统的顶点编程模型，提出基于 GPU 的通用图编程模型—Edge-Vertex 编程模型，为图算法提供类似 Pregel 的 API。同时根据 Edge-Vertex 编程模型，设计图系统的执行流程，把整个系统执行阶段划分成预处理阶段、EdgeCompute 和 VertexCompute，并阐述各个阶段的执行任务和逻辑操作。

4.分析 Edge-Vertex 编程模型潜在的性能瓶颈问题，优化运行时系统的性能，提出并实现了基于 GPU 的近似排序算法和数据重映射算法。由于在 VertexCompute 阶段，GPU 线程存在负载不均衡问题，根据 GPU 线程以 warp 为单元调度，提出基于 GPU 的近似排序算法，缓解同一个 warp 的线程负载不均衡的压力。研究消息缓存的数据存储方式，发现不同的数据存储方式对性能有影响。Medusa 消息缓存机制采用行优先存储，无法利用 GPU 的合并访存特性，提出应用于消息缓存的数据重映射算法，把消息缓存映射成列优先存储。

5.通过实验验证本文提出的图计算系统，系统性能可以获得大幅度的提升。首先，本系统的预处理过程移植到了 GPU 并行环境，充分利用了 GPU 的并行处理能力。与 Medusa 系统的预处理比较，本系统的预处理过程提高 10~50 倍。其次，基于 GPU 的近似排序缓解了同一个 warp 的线程负载不均衡压力，同时排序引入很小开销。近似排序前与近似排序后比较，系统图计算性能提高了 8~20%。最后，数据重映射算法使得消息缓存按照列优先的方式存储，减少了 GPU 线程非合并访存的次数，同时额外内存消耗几乎可以忽略不计。消息缓存按列式存储和消息按行式存储作比较，系统图计算性能提升了 10%~25%。系统运行的总时间包括预处理时间和系统图计算时间，测试 PageRank、SSSP 算法和 HCC 算法，相对于 Medusa 系统，本系统可以获得大约 1.6 到 4.5 倍的性能优势。

虽然本论文针对基于 GPU 的图计算进行研究并取得了一定的成果，本系统仍然存在一些不足之处。许多功能需要进一步完善。具体地，今后可从以下几个方面着手研究：

（1）本系统目前是基于单个 GPU 环境，并不支持多个 GPU 的计算。因此可以设计一种消息通信机制，完成多个 GPU 在超步与超步之间的消息通信，并且要尽可能地减少通信量。

（2）由于消息缓存是基于数组的实现，所以本系统并不支持动态的图算法计算。所以可以尝试设计基于链表的消息缓存，扩展上述功能。

（3）目前的图计算任务由 GPU 完成，可以考虑 CPU+GPU 混合计算模型，对图计算任务进行合理分配，实现 CPU+GPU 图计算系统，并扩展到集群环境，充分利用 CPU 和 GPU 的计算资源。

参考文献

[1] Brin S, Page L. The anatomy of a large-scale hypertextual Web search engine. Computer networks and ISDN systems, 1998, 30(1): 107-117.

[2] Langville A N, Meyer C D. Google's PageRank and beyond: the science of search engine rankings. Princeton University Press, 2011.

[3] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107-113.

[4] Bu Y, Howe B, Balazinska M, et al. HaLoop: efficient iterative data processing on large clusters. In: Proceedings of the VLDB Endowment. 2010, 3(1-2): 285-296.

[5] Deelman E, Singh G, Su M H, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming, 2005, 13(3): 219-237.

[6] Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In: Proceedings of the VLDB Endowment. 2012, 5(8): 716-727.

[7] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. 2010: 135-146.

[8] Salihoglu S, Widom J. GPS: A graph processing system. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management. 2013: 22.

[9] Cheng R, Hong J, Kyrola A, et al. Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European conference on Computer Systems. 2012: 85-98.

[10] Shao B, Wang H, Li Y. Trinity: A distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. 2013: 505-516.

[11] Apache hama. http://hama.apache.org/.

[12] Owens J D, Houston M, Luebke D, et al. GPU computing. In: Proceedings of the IEEE. 2008, 96(5): 879-899.

[13] Nvidia cuda. http://www.nvidia.com/object/cuda/.

[14] Nvidia C. Nvidia cuda c programming guide. NVIDIA Corporation, 2011, 120.

[15] Nickolls J, Buck I, Garland M, et al. Scalable parallel programming with CUDA. Queue, 2008, 6(2): 40-53.

[16] Stone J E, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 2010, 12(1-3): 66-73.

[17] Luna F D. Introduction to 3D game programming with DirectX 10. Jones & Bartlett Publishers, 2008.

[18] Lindholm E, Nickolls J, Oberman S, et al. NVIDIA Tesla: A unified graphics and computing architecture. Ieee Micro, 2008, 28(2): 39-55.

[19] Nickolls J, Dally W J. The GPU computing era. IEEE micro, 2010, 30(2): 56-69.

[20] 贾 佳, 杨学军, 李志凌. 一种基于冗余线程的 GPU 多副本容错技术[J]. 计算机研究与发展, 2013, 50(7): 1551-1562.

[21] 陈钢. 众核 GPU 体系结构相关技术研究: [复旦大学博士学位论文]. 上海：复旦大学, 2011, 12-33.

[22] 唐滔. 面向 CPU—GPU 异构并行系统的编程模型与编译优化关键技术研究: [国防科学技术大学博士学位论文]. 长沙: 国防科学技术大学, 2012, 17-39.

[23] NVidia C. C best practices guide. NVIDIA, Santa Clara, CA, 2012.

[24] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system. In: Proceedings of 26th Symposium on MSST. 2010: 1-10.

[25] Khayyat Z, Awara K, Alonazi A, et al. Mizan: a system for dynamic load balancing in large-scale graph processing. In: Proceedings of the 8th ACM European Conference on Computer Systems. 2013: 169-182.

[26] Goodrich M T, Tamassia R. Data structures and algorithms in Java. John Wiley & Sons, 2008.

[27] Mehlhorn K, Näher S, Uhrig C. The LEDA platform for combinatorial and geometric computing. Automata, Languages and Programming, Springer Berlin Heidelberg, 1997:7-16.

[28] Erwig M. Inductive graphs and functional graph algorithms. Journal of Functional Programming, 2001, 11(05): 467-492.

[29] Kyrola A, Blelloch G E, Guestrin C. GraphChi: Large-Scale Graph Computation on Just a PC. In: Proceedings of OSDI. 2012, 12: 31-46.

[30] Han W S, Lee S, Park K, et al. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In: Proceedings of the 19th ACM

SIGKDD international conference on Knowledge discovery and data mining. 2013: 77-85.

[31] Prabhakaran V, Wu M, Weng X, et al. Managing Large Graphs on Multi-Cores with Graph Awareness. In: USENIX Annual Technical Conference. 2012: 41-52.

[32] Jiang H, Chen Y, Qiao Z, et al. Accelerating MapReduce framework on multi-GPU systems. Cluster Computing, 2014, 17(2): 293-301.

[33] Chen L, Agrawal G. Optimizing MapReduce for GPUs with effective shared memory usage. In: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. 2012: 199-210.

[34] Basaran C, Kang K D. Grex: An efficient MapReduce framework for graphics processing units. Journal of Parallel and Distributed Computing, 2013, 73(4): 522-533.

[35] Hong C T, Chen D H, Chen Y B, et al. Providing source code level portability between CPU and GPU with MapCG. Journal of Computer Science and Technology, 2012, 27(1): 42-56.

[36] Harish P, Narayanan P J. Accelerating large graph algorithms on the GPU using CUDA. High performance computing–HiPC, Springer Berlin Heidelberg, 2007. 197-208.

[37] Katz G J, Kider Jr J T. All-pairs shortest-paths for large graphs on the GPU. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. 2008: 47-55.

[38] Luo L, Wong M, Hwu W. An effective GPU implementation of breadth-first search. In: Proceedings of the 47th design automation conference. 2010: 52-55.

[39] Merrill D, Garland M, Grimshaw A. High performance and scalable gpu graph traversal. Department of Computer Science, University of Virginia, Tech. Rep, 2011.

[40] Hong S, Kim S K, Oguntebi T, et al. Accelerating CUDA graph algorithms at maximum warp. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. 2011: 267-276.

[41] Zhong J, He B. Medusa: Simplified graph processing on GPUs. IEEE Transactions on Parallel and Distributed Systems, 2014, 6(25):1543–1552.

[42] Khorasani F, Vora K, Gupta R, et al. CuSha: vertex-centric graph processing on GPUs. In: Proceedings of the 23rd international symposium on High performance parallel and distributed computing. 2014: 239-252.

[43] Sedgewick R. Algorithms in C++, Part 5 Graph Algorithm. Addison Wesley, 2002.

[44] Pemmaraju S, Skiena S. Implementing discrete mathematics: Combinatorics and graph theory with mathematica. Cambridge University Press, 2003.

[45] Goldberg A V, Radzik T. A heuristic improvement of the Bellman-Ford algorithm. Applied Mathematics Letters, 1993, 6(3): 3-6.

[46] Aiello W, Chung F, Lu L. A random graph model for power law graphs. Experimental Mathematics, 2001, 10(1): 53-66.

[47] Cederman D, Tsigas P. A practical quicksort algorithm for graphics processors. Algorithms-ESA, Springer Berlin Heidelberg, 2008. 246-258.

[48] Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of International Symposium on Parallel & Distributed Processing. 2009: 1-10.

[49] Harris M, Owens J, Sengupta S, et al. CUDPP: CUDA data parallel primitives library. http://gpgpu.org/developer/cudpp, 2007.

[50] Harris M, Sengupta S, Owens J D. Owens. Parallel prefix sum (scan) with CUDA. GPU gems, 2007, 3(39): 851-876.

[51] Coleman S, McKinley K S. Tile size selection using cache organization and data layout. In: Proceedings of Programming language design and implementation. 1995, 30(6): 279-290.

[52] Leskovec J, Mcauley J J. Learning to discover social circles in ego networks. In: Advances in neural information processing systems, 2012: 539-547.

[53] Takac L, Zabovsky M. Data analysis in public social networks. In: International Scientific Conference and International Workshop Present Day Trends of Innovations. 2012: 1-6.

[54] Yang J, Leskovec J. Defining and evaluating network communities based on ground-truth. Knowledge and Information Systems, 2015, 42(1): 181-213.

[55] Leskovec J, Kleinberg J, Faloutsos C. Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. 2005: 177-187.

[56] Bader D A, Madduri K. Gtgraph: A synthetic graph generator suite. Atlanta, GA, February, 2006.

附录 A 攻读学位期间所发表的学术论文

[1] Jun Xiao, Hao Chen, Jianhua Sun. High Performance Approximate Sort Algorithm using GPUs. 2015 International Conference on Computer Science and Intelligent Communication，EI 会议，已录用，文章编号：CS208。

附录 B 攻读学位期间所参与的科研活动

[1] 国家自然科学基金项目：GPU通用计算系统检查点方法研究(项目编号：61272190)，2013.01-2016.12

[2] 国家自然科学基金，基于程序分析方法的Web安全研究(61173166)，2012.1-2015.12

[3] 新世纪优秀人才支持计划：基于多核不对称特性的虚拟机优化方法研究，2013.1-2015.12

致　谢

光阴似箭，看似漫长的三年研究生生涯即将结束。三年来，我无时无刻不在感受着老师、同学、朋友和亲人们的指导与关心，在此谨向他们表示最衷心的感谢和最诚挚的谢意！

在此论文完成之际，首先感谢我的导师陈浩教授在科研工作、学习等方面给予耐心指导和帮助。本论文是在陈老师的精心指导下完成的。从论文的选题、论文内容、实验的进展以及文章的修改等环节，陈浩老师不辞辛苦，多次与我就论文中许多核心问题作深入细致地探讨，给我提出切实可行的指导性建议。陈老师严谨的治学态度和对研究工作的热情让我收益匪浅，是我以后工作生活中的榜样。另外还要感谢孙建华副教授，孙老师事实求是的科学态度和认真勤奋的工作作风，永远值得我学习和效仿。

其次，我要感谢常诚、陈志文、李文涛等师兄在科研和论文工作中给予了大量帮助、指导和有益讨论，同他们的交流讨论提高了学术水平。我也要感谢我的同门周东伟、刘祎璠、谢劲、谭元元、余玲军等，在三年学习生活期间给予的关怀与鼓励，在业余时间相互探讨也给了我不少启迪。我还要感谢我的室友伍菊红、邓颖卓、万勤和研 12 级计算机科学与技术一班的同学们，感谢他们在生活中给我带来了快乐和安慰，让我在学习的闲暇能有一个舒适的生活环境和平和的心态。特别感谢我的男朋友宋艳超，完成了本论文的排版工作。

再次，感谢我的家人和亲友对我学习的大力支持，感谢他们在我多年的求学期间，对我生活无微不至的关怀和照顾。

此外，我要给在百忙之中抽出时间为我审阅论文的专家学者们致谢，感谢他们对我的论文提出意见与建议，是他们的指导让我能够不断进步，谢谢！

最后，再次向所有关心和帮助我的老师和朋友们表示深深的谢意，真心祝福他们健康快乐、万事顺利！

肖军

2015 年 5 月