本科生毕业设计

基于 GPU 的 SSSP 算法优化研究

院　　系	计算机科学与技术
专业班级	计算机 1808
姓　　名	马忠平
学　　号	U201814719
指导教师	张宇

2022 年 05 月 30 日

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，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：　　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障、使用学位论文的规定，同意学校保留并向有关学位论文管理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。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，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：　　年　　月　　日

导师签名：　　年　　月　　日

摘 要

随着 GPU 设备运算速度的不断提高，GPU 并行加速技术也越来越多地应用于科学计算、学术研究等领域。同时，多 GPU 协同工作也成为提高计算机运算速度的一种重要方式。如何利用 GPU 异构并行性对算法进行加速优化以及如何协调多 GPU 联合计算成为一个非常重要的研究课题。

针对上述课题，基于 CUDA 编程环境，实现了对于 SSSP 典型求解算法 Dijkstra 以及 Bellman-Ford 算法的并行异构加速。此外，还基于前人提出的 Groute 模型实现和测试了多 GPU 联合计算下的 SSSP 算法加速效果。实验结果表明，GPU 的并行优化效果对于并行性较差的 Dijkstra 算法加速效果有限，但十分契合于 Bellman-Ford 算法，并行加速后的加速比甚至能达到 500 以上。另一方面，多 GPU 联合计算不失为一种有效的算力提升手段，但其通信开销不容小觑。

关键词： GPU，CUDA 编程， 单源点最短路径，Groute 编程框架，联合计算

Abstract

With the continuous improvement of the computing speed of GPU devices, GPU parallel acceleration technology is increasingly used in scientific computing, academic research and other fields. At the same time, multi-GPU collaboration has also become an important way to improve the speed of computer computing. How to use GPU heterogeneous parallelism to accelerate the optimization of algorithms and how to coordinate multi-GPU joint computing has become a very important research topic.

In view of the above problems, based on the CUDA programming environment, parallel heterogeneous acceleration of the typical solution algorithm Dijkstra and Bellman-Ford algorithm for SSSP is realized. In addition, based on the previous grute model, the acceleration effect of SSSP algorithm under multi-GPU joint computing is implemented and tested. Experimental results show that the parallel optimization effect of GPU is limited to the acceleration effect of Dijkstra algorithm with poor parallelism, but it is very suitable for bellman-Ford algorithm, and the acceleration ratio after parallel acceleration can even reach more than 500. On the other hand, multi-GPU joint computing is an effective means of improving computing power, but its communication cost should not be underestimated.

Keywords: GPU, CUDA Programming, SSSP, groute programming framework, joint computing

目 录

1 绪 论

本章我们首先介绍了当前单源点最短路径算法的应用场景，简述了 SSSP 算法的研究背景和发展趋势，详细阐述了基于多 GPU 的不规则编程模型 Groute 以及用于减少无权图搜索邻域空间的 Hub-Accelerator 编程框架。其次，简述了研究目的和主要内容，梳理了论文结构，表明了课题来源。

1.1 课题背景

1.1.1 研究背景和趋势

社交网络正在变得无处不在，其数据量正在急剧增加。流行的在线社交网络网站，如 Facebook，Twitter 和 LinkedIn，如今都拥有数亿活跃用户。谷歌的新社交网络 Google+吸引了 2500 万独立用户，并在推出后的第一个月以每天约 100 万访问者的速度增长。实现这些海量图形的在线和交互式查询处理，特别是快速捕获和发现实体之间的关系，正在成为从社会科学到广告和营销研究再到国土安全的新兴应用不可或缺的组成部分。

最短路径计算是管理和查询社交网络最基本但最关键的问题之一。社交网络网站 LinkedIn 开创了著名的最短路径服务"你如何连到 A"，它在 3 个步骤内表示出了你和用户 A 之间友谊链的精确描述。微软的 Renlifang（EntityCube）记录了超过 1000 万个实体（人员，位置，组织）的十多亿个关系，允许用户在距离小于或等于 6 时检索两个实体之间的最短路径。新出现的在线应用程序"Six Degrees"提供了一种互动方式来展示您如何与 Facebook 网络中的其他人建立联系。此外，最短路径计算在确定信任和发现网络游戏中的朋友方面也很有用[1,2]。

另一方面，具有多个附加加速器的节点在高性能计算中逐渐崭露头角，由于图形加速处理器（GPU）的效率、并行性、可扩展缓存机制以及区别于通用计算机的专用高效率指令等在图形计算方面的优势，图形加速器的运用逐渐变得流行起来。由于单个 GPU 设备运算能力终究有限，为了扩展主机的计算能力，多 GPU 的节点成为一种有效的算力提升方式。

多 GPU 节点由一个主机（CPU）和多个图形加速器（GPU）组成，这些设

备通过低延迟、高吞吐量的总线进行连接（如图 1-1 所示），这些互联允许并行应用程序有效地交换数据，并利用 GPU 进行并行计算。

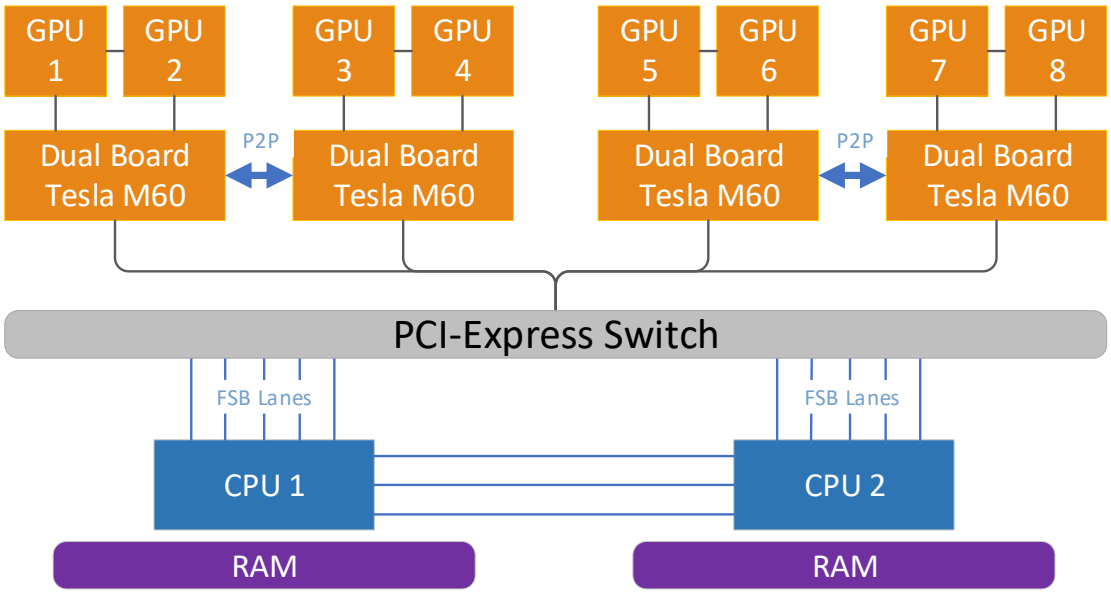

图 1-1 多 GPU 节点架构

1.1.2 提高并行图计算的途径

多 GPU 节点通常使用以下两种方法之一进行编程：

（1） 简单方法

简单方法之中，每个 GPU 都是单独管理的，每个 GPU 设备运行一个进程[3, 4]或者使用批量同步并行的编程模型（BSP）[5]，其中的应用程序按照回合形式执行，每个轮次需要完成本地计算和全局通讯的过程[6,7]。该方式会受到来自各种源（如 OS 等）的开销影响，并且需要消息传递接口进行通信。另一方面，BSP 模型可能会在实现基于轮次执行的全局程序中引入不必要的序列化。

上述两种编程方法都可能导致多 GPU 平台利用率的低下。特别是对于不规则应用程序（负载不平衡以及常发生不可预测通讯的程序），多 GPU 平台难以发挥其并发性优势，极端情况下，甚至同一时间只能有一台设备运行，其他设备则等待数据同步。

（2） 异步编程模型

原则上，异步编程模型可以减少其中一些问题。与基于回合形式的通讯不同，处理器可以自主计算和进行异步通讯，而无需等待其他处理器达到全局同步。但

是，很少有应用程序能够进行异步并发执行。开发异步并发程序时，需要深入了解底层体系结构和通讯网络，并且涉及对代码进行复杂的修整。

1.1.3 面临的问题和挑战

本课题从两个方向入手，分别从硬件、软件方面尝试对单源点最短路径算法进行优化。

硬件方面拟采用多 GPU 加速器架构，采用良好并行性的硬件设备。软件方面拟使用尽量高效的算法，并且在算法涉及过程中更多考虑到充分利用硬件设备的可能性。硬件、软件以及软硬件的交互等方面都有着一些需要注意的难点。

硬件方面首先需要考虑多 GPU 的异步控制方式，其次还需要解决多 GPU 之间的高效率通讯的问题。软件方面需要尽量减低算法时间复杂度，尽量提高算法效率。另外，为了充分利用硬件的高并发性，很明显需要考虑对图数据进行拆分，以满足能执行并行处理条件，为了保证算法的正确性，图结构拆分后的合并处理也是需要解决的一大问题。

1.2 国内外研究现状

2012 年，郭绍忠等人进行了基于 GPU 的并行 Moore 算法的初步研究，针对图形处理器(GPU)上的动态数据处理问题,在分析已有的并行单源最短路径(SSSP)算法的基础上,对 GPU 上的 Moore SSSP 算法进行并行化设计与实现。并行的 Moore 算法有效降低了空线程开销、访问开销以及同步时间[8]。

2013 年 Ruoming Jin 等人完成了一种基于集线节点（具有大量边的点）为中心的新的最短路径的计算技术——Hub-Accelerator 框架[9]，用于计算 k 度最短路径。该技术使我们能够通过大大限制集线点的邻域扩展范围或完全修剪搜索中的集线器（使用集线器标记方法）来显著减少搜索空间。Hub-Accelerator 方法用于最短路径计算比 BFS 和最先进的近似最短路径方法 Sketch 快两个数量级以上。Hub-Network 方法不会引入额外的索引成本和轻度的预计算成本;Hub2-Labeling 的索引大小和索引构建成本也适中，算法效率优于或可与近似索引 Sketch 方法。

2018 年，吴曼等人提出了割点与点割集的概念，将寻找割点与点割集的算法，与经典的 Dijkstra 算法结合，形成改进的并行算法并予以实现与应用，为寻

找无向图的最短路径提供了理论依据，并用其改进了路由协议 OSPF 中的路由选择算法，降低了算法的时间复杂度[10]。

2019 年，向志华等针对大规模网络中所有节点的全源最短路径的计算需求基于广度优先遍历（BFS）思想，在计算过程中设置存储队列，引入阻断路径，限制后续图节点的扩展范围，完成了图的减枝，大幅度降低最短路径计算的时间复杂。经测试，文中所设计的算法相较于传统 Dijkstra 算法在高、中、低规模的数据集上均可降低 50%以上的运算时间；相较于 BFS 算法，可以降低 20%以上的运算时间[11]。

2020 年 Tal Ben-Nun 等人提出了用于不规则计算的异步多 GPU 的 Groute 编程模型，还实现了通用的集合操作和分布式工作列表，无需大量编程工作即可开发不规则的应用程序。该方法为 8-GPU 和异构系统上的一套不规则应用程序提供了强大的扩展能力，使某些算法的执行效率提高了 7 倍以上[12]。

陈智康等人针对在已知环境地图中的单个陆地移动机器人路径规划求解问题，将蚁群算法的信息素思想加入到经典 Dijkstra 算法中，实验结果表明，优化后的算法能够在很大程度上减少路径规划过程中产生的冗余点，减少机器人寻路的移动代价[13]。

2021 年，曹大有等人通过利用遗传算法的框架模型建立了一个好理解、快速的最短路径问题求解过程[14]。

本文重点研究 Groute 编程模型和 Hub-Accelerator 编程框架。

1.2.1 Groute 编程模型

Groute —— 一种异步编程模型和运行时环境，可用于在多 GPU 系统上开发各种应用广泛的应用程序。基于底层网络的概念，Groute 旨在降低多 GPU 和异构平台上异步应用程序的编程复杂性。Groute 的通信结构很简单，但它们可用于有效地应用于从常规应用程序和 BSP 应用程序到特殊不规则算法的多类型程序。运行时环境的异步特性还促进了负载 平衡，从而更好地利用了异构多 GPU 节点。

（1）GPU 间通信

GPU 之间的内存传输可以由主机启动或设备启动。特别是，显式复制命令支

持主机启动的内存传输（对等传输），而设备启动的内存传输（直接访问，DA）则使用不同 GPU 间的内存访问来实现。请注意，并非所有 GPU 对都可用对等内存的直接访问，具体取决于总线拓扑的结构。但是，可以从所有 GPU 访问特定的主机内存。

设备启动的内存传输通过虚拟寻址实现，虚拟寻址将所有主机和设备内存映射到单个地址空间。虽然 DA 比对等传输更灵活，但它对内存对齐、合并、活动线程数和访问顺序有高度敏感性。为了支持设备启动的传输，在无法访问彼此内存的 GPU 之间，可以执行两阶段间接复制。在间接复制中，源 GPU 首先将信息"推送"到主机内存，然后由目标 GPU"拉取"，使用主机标志和系统范围的内存栅栏进行同步。在图 1 所示的拓扑中，GPU 一次只能传输到一个目标。这阻碍了异步系统的响应能力，尤其是在传输大型缓冲区时。解决此问题的一种方法是当消息在网络中传输时将其切分为多个数据包。经过实验证明，使用分组内存传输而不是单个对等传输时，开销随着数据包大小的增加而线性减少，对于 1-10MB 数据包，开销范围在 1~10%之间，此参数可根据各个应用程序进行调整来平衡延迟和带宽。另外，直接（推送）和间接（推/拉）传输的传输速率也存在一定差异，分组设备启动的传输和细粒度控制是有效的，即使主机管理的分组对等传输也是如此。由于设备启动的内存访问是用户代码编写的，因此可以在传输过程中同时执行其他的数据处理。

多 GPU 通信的另一个重要方面是多个源/目标传输，就像在集合操作中一样。由于互连和内存复制引擎的结构，优化较差的应用程序可能会使总线拥堵。NCCL 库[15]中使用的一种方法是在总线上创建环形拓扑。在这种方法中（如图 1-2 所示），每个 GPU 都传输到一个目标，通过直接或间接的设备启动传输进行通信。这可确保使用每个 GPU 的两个内存复制引擎，并充分利用总线。数据包化还用于在从上一个流水线操作接收信息时开始将信息传输到下一个 GPU。

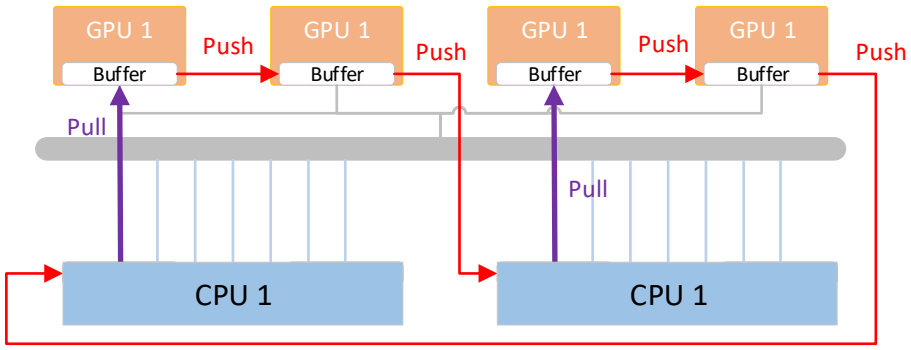

图 1-2 DA 环拓扑技术

（2）Groute 编程模型

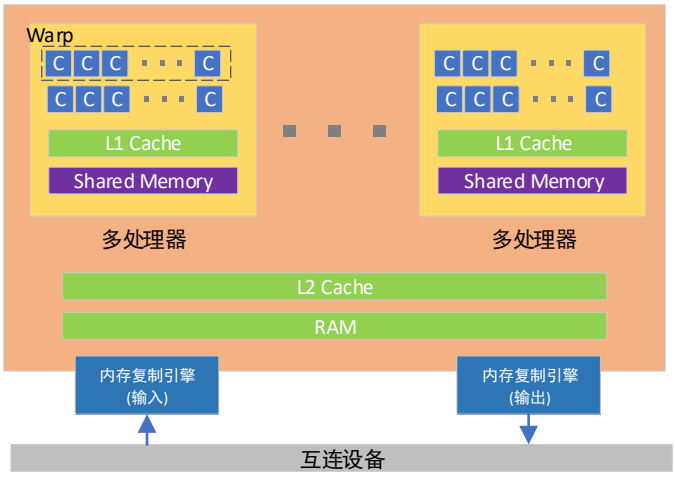

图 1-3 单个 GPU 的标准架构

单个 GPU 设备的结构如图 1-3 所示。每个 GPU 都包含一组固定的多处理器（MP）和一个 RAM 单元（称为全局内存）。GPU 过程（内核）通过调度由多个线程组成的网格（分组为线程块）在 MP 上并行运行。在每个分配给单个 MP 的线程块中，Warp（通常由 32 个线程组成）在内核上同步执行。此外，同一线程块中的线程可以通过共享内存进行同步和通信，以及使用自动管理的 L1 和 L2 高速缓存。为了支持并发内存写入，在共享内存和全局内存上定义了原子操作。

内核调用和主机启动的内存传输通过命令队列（流）执行，这些命令队列可用于表达任务并行性。一个或多个 GPU 之间的流同步通常使用事件执行，这些事件在一个流上运行，并在另一个流上等待。

GPU 调度程序按线程块分派内核，当没有更多线程块可供正在运行的内核调度时，通过调度其他内核的线程块，在同一 GPU 上实现多流并发。但是，高优

先级流允许应用程序开发人员立即调用内核，先调度来自新内核的线程块，然后再调度来自正在运行的内核的线程块。

虽然流/事件构造提供了对内核调度的细粒度控制，但在编写涉及多个 GPU 的高级功能时会出现困难。我们列出了一个编程表 Groute 编程抽象接口（如表 1-1 所示），它补充了现有模型，以简化多 GPU 开发，使用来自微基准测试得出的结论来最小化通信延迟。

表 1-1 Groute 编程接口

构造	描述
基本构造	
Context	表示运行时的单例
Endpoint	可以通信的实体（例如 GPU、CPU、路由器）
Segment	封装缓冲区、其大小和元数据的对象。
通讯设置	
Link(**Endpoint** src, **Endpoint** dst, **int** packetsize, **int** numbuffers)	将 src 连接到 dst，使用多缓冲区和 numbuffers 缓冲区和数据包大小的数据包。
Router(**int** numinputs, **int** numoutputs, **RoutingPolicy** policy)	多个端点连接在一起，实现动态通信。
通讯调度	
EndpointList RoutingPolicy(**Segment** message, **Endpoint** source, **EndpointList** routerdst)	程序员定义的函数，用于根据发送终结点和路由器目标终结点的列表确定可能的消息目标。路由器将按可用性选择目的地。
异步对象	
PendingSegment	当前正在接收的段。
DistributedWorklist(**Endpoint** src, **EndpointList** workers)	管理由路由器和每 GPU 链路组成的全对全工作项分发。

Groute 编程模型提供了多种构造来促进异步多 GPU 编程。表 1-1 列出了这些构造及其编程接口的简述。Groute 应用程序的运行包括两个阶段：数据流图构造和异步计算。Groute 程序首先指定计算的数据流图。这个有向图中的节点（我们称之为端点）表示（i）CPU 和 GPU 等物理设备，或（ii）称为路由器的虚拟设备，它们是实现复杂通信模式的抽象节点。数据流图中的边缘表示端点之间的通信链路，只要没有自循环（端点直接连接到自身）或多图中的多个相同边（即路由器只能有一个到同一端点的传出边），就可以创建边。注意，为了支持多任

务处理，可以从同一物理设备创建多个虚拟终结点。发送和接收方法允许端点在链路上发送和接收数据;收到数据后，端点可以使用回调对其执行操作。创建路由器时，程序员会指定路由策略，以确定接收到输入时的行为。例如，输入可以发送到单个终结点，也可以根据其可用性发送到终结点的子集。创建链路时，将指定该链路的分组和多缓冲策略。

1.2.2 Hub-Accelerator 框架

大规模社交网络中最短路径计算的核心问题来自 Hub 点（具有大量边的顶点）。与整个图网络规模相比，Hub 点的数量可能很少，但是，它们几乎是任何点的邻接点。事实上，中心在局部子图中扮演着非常重要的角色。它们充当连接顶点之间最短路径的常见中介，就像航空公司航班小世界中的枢纽城市一样。事实上，理论分析表明，少量的 Hub（由于幂律度分布）显着缩短了顶点之间的距离，并使大型的复杂网络显得更加“小”[16]。然而，枢纽是导致搜索空间爆炸的关键因素。对于 Hub 点的一步 BFS 将会极大扩展搜索的邻域空间，因此，Hub 点的处理方式非常重要，充分利用 Hub 点的特性，可以大大简化搜索空间，进而提高搜索效率。

Hub-Accelerator 框架是一种以 Hub 点为中心的新技术，该技术使我们能够通过大大限制 Hub 点的扩展范围（使用新颖的保持 距离的集线器网络概念）或完全修剪在线搜索中的集线器（使用 Hub2 标记方法）来显着减少搜索空间。Hub-Accelerator 方法平均比 BFS 和最先进的近似最短路径方法快两个数量级以上，包括 Sketch[17]，Tree Sketch[17]和 Rigel Paths[18]。集线器网络方法不会引入额外的索引成本和轻度的预计算成本；Hub2-Labeling 的索引大小和索引构建成本也适中，优于或可与近似索引 Sketch 方法相媲美。我们注意到，尽管最短路径计算已经过广泛研究，但大多数研究仅关注道路网络[19,20,21,22,23,24,25,26]或大规模社交网络上的近似最短路径（距离）计算[17,18]。据我所知，这是第一部明确解决这些网络中确切最短路径计算的工作。Hub-Accelerator 技术也很新颖，距离保持子图（Hub-network）发现问题本身对于图的挖掘和管理具有理论和实践的重要性。

Hub-Accelerator 框架的设计旨在利用枢纽点进行最短路径计算，而无需完全

扩展其邻域。该技术解答了限制 Hub 点邻域扩展限制、高效利用 Hub 网络进行最短路径搜索等问题，使得利用 Hub 点加速最短路径搜索过程成为可能。该技术分为两个步骤：Hub 网络发现：用于构建包含 Hub 节点的子图；基于 Hub 网络的双向 BFS：算法的计算核心，与一般的 BFS 不同的是，它在遇到 Hub 点时，不会无限制的进行扩展。另外，另一项技术用于对上述方法进行进一步优化：Hub2 标记技术，通过完全避免扩展任何集线器来进一步推动最短路径计算的速度边界。为了实现这一目标，使用更昂贵但通常负担得起的预计算和内存成本来加快在线搜索速度。

1.3 研究目的和主要内容

了解 SSSP（单源最短路径）算法和 GPU 的基础相关知识，学习并了解现有的 SSSP 算法，基于现有 SSSP 算法，探索新的 SSSP 算法。

1.4 论文结构

论文全文共六个章节，按照以下方式组织。

第一章为绪论，介绍了单源点最短路径求解算法的历史和发展趋势，分析了并行图计算的提高途径，分析了面临的问题和挑战，对 Groute 多 GPU 编程模型进行了简单阐述。

第二章为具体背景技术概述，本章中介绍了单源点最短路径的传统算法，分析了算法的优劣势和可能的优化方向，对 CUDA 架构和并行编程环境进行了基于自己的理解的简单介绍，对 Groute 编程模型的具体实现进行了研究和分析。

第三章为 SSSP 算法优化设计，提出了针对于 Dijkstra 算法、Bellman-Ford 算法的优化方案，并对优化方案进行了论证分析。

第四章为 SSSP 算法优化实现，详细介绍了使用 CUDA 并行编程对传统算法的优化实现，详细描述了 Groute 编程模型单源点最短路路径求解算法的实现过程。最后对设计实现过程中针对制约隐私做出的考虑，并使用 COCOMO 模型估算了实现方案的成本。

第五章为测试与分析，列出了测试使用的算例信息，介绍了针对基于 GPU 的 SSSP 算法性能测试的云端环境。对第四章实现的优化求解算法进行测试，展现

出测试成果，并对测试结果进行的一些理论分析。

第六章为总结与展望，对通篇文章展开的工作进行汇总总结，分析出自己研究工作中存在的不足之处和问题所在，并对 GPU 并行技术的未来发展提出了展望。

1.5 课题来源

TODO

本课题受国家自然科学基金项目“XXXXXX”（项目编号：60273073）和“973”国家重点基础研究发展项目“XXXX”（项目编号：200418203）资助。课题来源于香港大学工程学院计算机系系统与网络实验组的网络功能虚拟化项目。

2 具体背景技术概述

传统单机带权路径单源点最短路径求解常见的算法有迪杰斯特拉算法以及贝尔曼-佛德算法，对于无权图，广度优先搜索也是求解单源点最短路径的常用算法之一。

本章对传统的单源点最短路径求解算法进行分析。

2.1 迪杰斯特拉算法

迪杰斯特拉算法是求解单源点最短路径最常见的算法之一。迪杰斯特拉算法由一名荷兰科学家于 1956 年发现，此算法使用类似于广度优先搜索思想，这是一个贪心算法。

迪杰斯特拉算法是一个迭代算法，在每一轮基于上一轮迭代的结果以确定一个节点与源点的最短距离与路径。算法伪代码如下。

Algorithm 1 Dijkstra Algorithm

Input: G = (V,E), s ; {graph and source node}

Output: dist , pre; {shortest distance and path}

```
1: initialize: Set dist[u]⇐ ∞,pre[u]⇐-1,visited[u]⇐ false for each u ∈ V;
2: while true do
3:     Find the vertex u∈V which have min{dist} and visited[u]=false;
4:     visited[u]⇐ true
5:     if  dist[u]=∞  then
6:         break;
7:     end if
8:     for  each v ∈ neighbor(u) { (u,v) ∈ E }  do
9:         if dist[v] ≥ dist[u] + |u,v| then
10:            dist[v] ⇐ dist[u] + |u,v|
11:            pre[v] ⇐ u
12:        end if
13:    end for
14: end while
15: return dist,pre;
```

对于最差情况下（连通图），算法一共需要 n-1 轮迭代，每轮迭代中需要完成如下操作：（1）查找最小 dist 对应的节点编号；（2）遍历节点的每一个邻接点，对距离执行松弛操作。对于传统实现，若使用数组储存距离信息，则算法时间复杂度$O\big((n-1)*(n+e)\big)=O(n^2)$。值得注意的是，迪杰斯特拉算法仅适用于无负权边的图的计算。若需要计算负权图，需要使用贝尔曼-佛德算法。

2.2 贝尔曼-佛德算法

贝尔曼-福得算法类似于缔结特斯拉算法，都是一种贪心算法。算法以松弛操作为基础，即用更小更准确得最短路径值替代估计的最短路径值，多次迭代直到得到最优解。在两个算法中，计算过程中每两个点之间的估计距离值都大于或者等于真实值，在算法执行过程中逐渐被新找到最短路径替代。不同点在于，迪杰斯特拉算法以贪心法选取未被处理的具有最小与源节点估计距离的节点，然后依次对点的所有出边依次进行松弛操作；而贝尔曼-福德算法直接对所有边进行松弛操作，共 n-1 次（其中 n 是图的点的数量）。在迭代计算过程中，已计算得到正确的距离的边的数量不断增加，直到所有边都计算得到了正确的路径。这样的策略使得贝尔曼-佛德算法比迪杰斯特拉算法适用于带有负权边的图。

算法伪代码如下。

Algorithm 2 Bellman-Ford Algorithm

Input: G = (V,E), s ; {graph and source node}

Output: dist , pre; {shortest distance and path}

```
1: initialize: Set dist[u]⇐ ∞,pre[u]⇐-1,visited[u]⇐ false for each u ∈ V;
2: for i=1 →|V|-1 do
3:     for each u ∈ V do
4:         for each v=neighbor(u) (u,v)∈E do
5:             if dist[v]¿dist[u]+|u,v| then
6:                 dist[v]=dist[u]+|u,v|
7:                 pre[v]=u
8:             end if
9:         end for
10:    end for
11: end for
12: return dist,pre;
```

贝尔曼-佛德算法需要进行|V|-1 轮循环，循环中对每个节点的每条边执行松弛操作。若图中存在负权回路且回路中的点可达，则不存在最短路径，该算法无法收敛。若采用邻接矩阵，算法的时间复杂度为$O(|V|^3)$，若采用邻接表，时间复杂度约为O(|V||E|)。

2.3 广度优先搜索算法

广度优先搜索算法也经常用于最短路径求解，单仅适用于无权图。算法伪代码如下。

Algorithm 3 BFS Algorithm

Input: G = (V,E), s ; {graph and source node}

Output: dist , pre; {shortest distance and path}

```
1: initialize: Setpre[u]⇐-1,visited[u]⇐ false for each u∈V,dist[s]⇐0
2: visited[s]⇐true
3: queue.push(s) {queue for BFS}
4: while queue is not empty do
5:     u⇐queue.pop()
6:     for each u=neighbor(u) do
7:         if visited[v]=true then
8:             continue
9:         end if
10:        visited[v]=true
11:        dist[v]=dist[u]+1
12:        pre[v]=u
13:        queue.push(v)
14:    end for
15: end while
16: return dist,pre;
```

BFS 算法利用队列和访问标记完成了搜索空间的剪枝，且无向图无需进行松弛操作能保证在较时间复杂度下完成单源点最短路径的求解。

2.4 开发工具分析及选择

课题目的在于研究基于图形显卡对单源点最短路径求解算法的优化和加速，因此需要进行配置可用的显卡驱动和编程环境。

当下常用的为 CUDA(Compute Unified Device Architecture)。目前为止基于 CUDA 的 GPU 销量已达数以百万计，软件开发商、科学家以及研究人员正在各个领域中运用 CUDA，其中包括图像与视频处理、计算生物学和化学、流体力学模拟、CT 图像再现、地震分析以及光线追踪等等。

为了完成此次算法优化研究，分别在本地和云端配置了 CUDA11.6 开发环境，使用 C++语言进行代码的开发，使用 NVCC、GCC 编译器完成代码编译，使用 ssh 等工具连接云端将代码上传，并使用云端服务器完成代码程序的运行测

试。

2.5 计算机统一设备框架（CUDA）

计算机统一设备架构（CUDA）是由 NVIDIA 在 2007 年推向时常的并行计算架构，CUDA 作为 NVIDIA 图形处理器的通用计算引擎，提供给我们利用图形加速卡进行 GPGPU（General Purpose Graphics Process Unit）开发的全套工具。

CUDA 不仅仅是一种编程语言，它包括 NVIDIA 对于图形加速卡的完整的解决方案：从支持通用计算并行架构的 GPU，到实现计算所需要的硬件驱动程序、编程接口、程序库、编译器、调试器等。NVIDIA 提供了一种较为简便的方式编写 GPGPU 代码：CUDA C。CUDA 为 C 程序员提供了一个完整的接口，程序员可以利用接口访问本地 GPU 的内存、命令集以及并行计算元素，CUDA 为 GPGPU 编程提供了一个开放的体系结构，并且可以支持多达上万个线程的同时运行。

2.5.1 CUDA 架构

CUDA 的软件架构如图所示，CUDA 编程框架可以分为三个部分：编程接口（API）、运行时需要的 runtime 库和设备驱动。如图 2-1 所示。

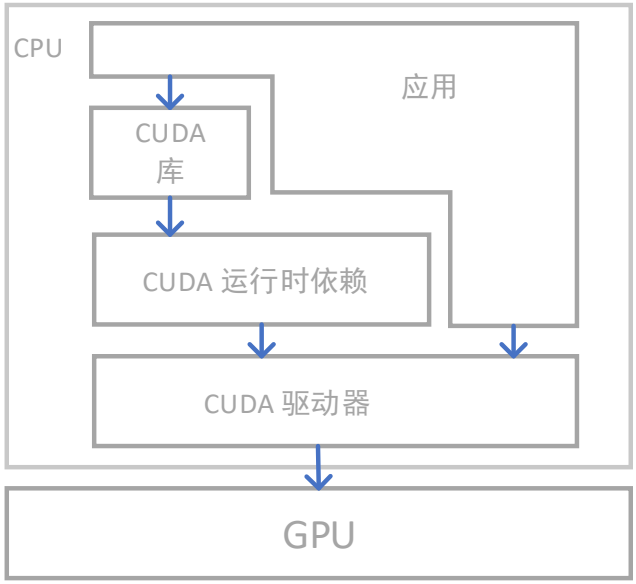

图 2-1 CUDA 软件架构

提供 CUDA 给出的应用开发库是 API 层的主要任务，大规模并行计算问题由它负责解决。

开发和运行 CUDA 程序所需要的相关组件和环境只要由 Runtime 库提供，包括定义的各种基本数据类型、设备访问、转换数据类型、调度函数等等。

2.5.2 CUDA 线程和内存结构

GPU 已经逐渐成为一个并行通用的高效计算平台，在某些领域甚至已经超越了 CPU 的使用率。基于此，越来越多的研究人员开始热忠于一个新的研究方向：GPGPU——主要研究 GPU 如何在其他科学计算应用领域内来进行更为广泛的应用计算，从而使得传统的计算技术和算法优化问题在高性能计算平台上可以获得更好的效果。当代可编程 GPU 已经发展成为了一种计算能力强大、并行性高以及拥有极高的内存带宽的高性能计算设备。

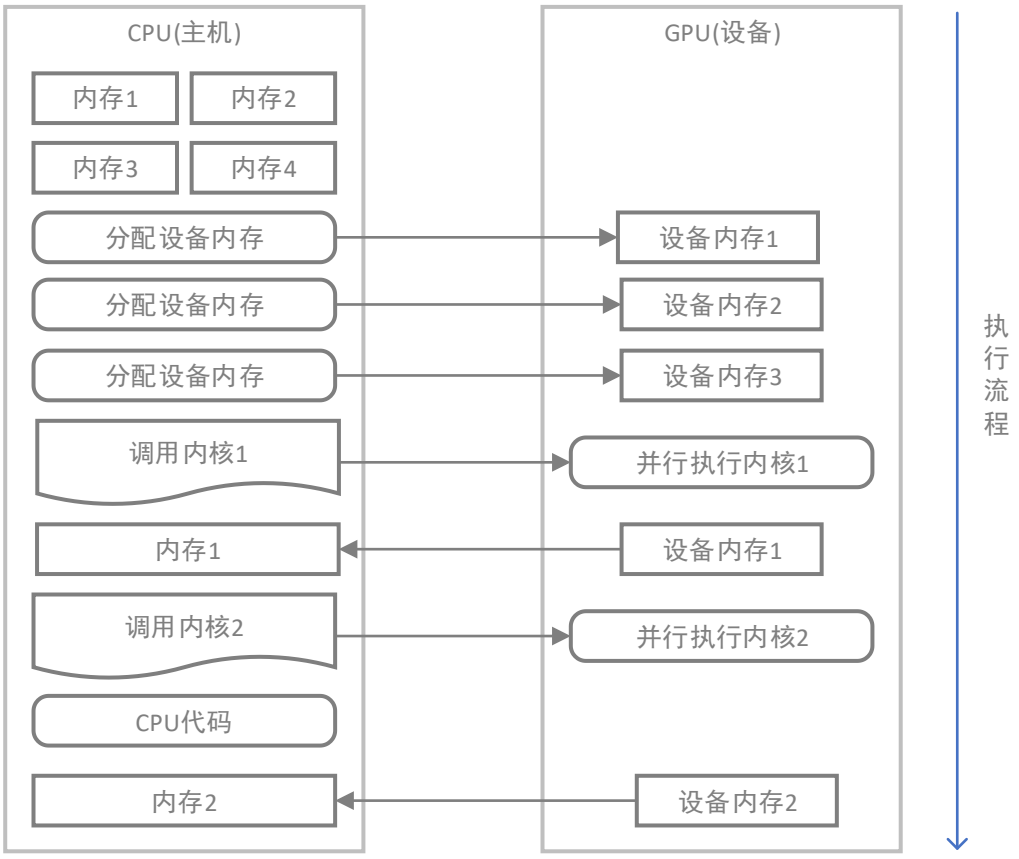

图 2-2 CUDA 计算模型

如图 2-2 所示为 CUDA 计算模型。基于 CUDA 的计算模型，我们将一个程序分为两部分：主机端（host）和设备（device）端。主机端的是 CPU 执行的程序部分，设备端的是 GPU 上执行的程序部分。内核（kernel）是设备端程序的另

外一种叫法。一般情况下，CPU 执行主机端的程序会准备好数据并将其复制进显卡内存中，然后设备端的程序由 GPU 执行完后，主机端程序会将生成的数据结果从显卡的内存中拿回。

CUDA 还指基于扩展 C 编程语言的直观和可扩展的编程模型，该模型将 CPU 和 GPU（所谓的主机和设备）统一到异构计算系统中，在两者中实现最佳优势。具体来说，CUDA-C 包含三种类型的函数：

(1) 主机函数，它的调用、执行都仅由主机端来完成；

(2) 内核函数，定义时必须要加上_global_限定符，它由主机端调用，设备端执行；

(3) 设备函数，定义时必须要加上_device_限定符，仅由设备端调用、执行。顺序操作应被编程为主机函数，可并行化操作应被编程为内核函数或者设备函数。主机函数和内核函数都将在 main 函数中被封装和调用。

事实上，一次内核调用将会在 GPU 上并行执行大量的线程。取决于计算能力的等级，一个 grid 对应于一个 GPU 或者多个多处理器。而一个 block 中所有的线程在一个流多处理器（SM）上并发地执行。当这个 block 中的所有线程都执行完毕并终止之后，GPU 会激活一个新的 block 并将其分配到当前空闲的 SM 上。处于同一个 block 里的多个 block 可以在一个 SM 上同时被激活。

当然在同一个 SM 上可能会同时被激活，但一个 block 里的线程不会被拆分并在多个 SM 上执行。同一个 block 里的线程可以被同步，并且它们可以访问 SM 里的共享存储器。SM 将每个线程与一个标量处理器核心（SP）映射起来，线程可以在 SP 上独立地执行任务，并使用独自的指令地址和寄存器。

CUDA 线程被组织为“grid-block-thread”的层次结构，具有不同计算能力的 GPU 有不同的对 block 和 grid 维数和尺寸的限制。这样的线程组织在硬件上也对应着一个有层次的实现。一个 GPU 由一个或多个 SM 组成，而每个多处理器由多个 SP 组成。标量处理器又称作 CUDA 核。每个标量处理器可以独立地执行计算任务并使用指定范围的计算资源。整个系统运作于 CUDA 的 SIMT 体系之上。

在 CUDA 计算结构中，内存的层次机制是一个重要架构技术，内存分为三个层次，分别是块内本地内存、共享内存和全局内存。在同一线程块中，每个线程

都有属于自己的块内本地内存，同时开辟一块共享内存共每一个线程块使用，线程块内的所有线程都可以访问自己块的共享内存，但是不能访问别的线程块所属的共享内存，因此，不同块之间的共享内存是互相隔离的。对于全局内存，在内核程序中的所有线程块中的线程都可以访问。

2.5.3 单指令多线程（SIMT）模式

在运行主机-设备机制的并行计算时，CPU 主机的主程序使用设备内核的计算网格，在计算网格上部署的所有线程块不是指定到一个处理器上，而是由 CUDA 分配给多处理器并执行，这样使得一个线程块中的所有线程可以执行在一个多处理器中。并多处理器管理线程的方式是使用单指令多线程 SIMT（Single Instruction MultiThreads）架构实现的，它有一个标量处理器核心，所有线程都映射到这个核心上成为标量线程，每一个标量线程在执行时都是独立地分配和使用各自的寄存器状态和指令地址。

SIMT 的模式与单指令多数据 SIMD 的向量式组织结构相似，虽然是控制多个处理单元，但是用单指令来完成的。不过 SIMT 让我们可以对独立的标量线程进行线程级细粒度的并行编码，也可以实现数据并行的编码，服务于协同线程，这些是 SIMD 向量机模式做不到的。

将一个 CUDA 应用程序分为主机端和设备端两个部分，主机端一般是指设备宿主主机的 CPU 处理器，设备端则指 CUDA 所能访问管理的显卡设备。一般情况下，一个系统最多同时只能拥有一个主机端，但是可以同时拥有多个设备端。主机端程序主要是通过 CPU 负责实际计算任务的执行，因此，主机端一般负责程序中的串行部分的业务处理，而 GPU 则主要用于进行并行计算任务的处理，一般以多线程的形式执行，由于主机端和设备端调用不同的处理单元，即前者调用主机的 CPU 处理单元，后者调用显卡设备的处理单元，因此二者访问的存储器空间是不同的，前者访问主机内存空间，而后者访问显卡设备内存空间。主机端负责的任务除了一般的与显卡设备交互以及 CUDA 程序的串行部分的计算任务之外，还负责对显卡设备调用前的环境初始化以及相关数据预处理等工作。

一个优秀的 CUDA 程序应该由主机端负责程序中串行任务的执行，并回收分配给已经执行结束的设备端程序的资源，并初始化下一个设备端内核函数的执

行环境启动执行任务，从而减少 host 与 device 之间的数据传输，在 device 上一定时间内执行尽量多的运算。

2.6 Groute 编程多 GPU 并发编程模型

环境由三层组成，如图 2-3 所示。底层包含节点拓扑和 GPU 间通信的低级管理，中间层实现 Groute 通信构造，使用拓扑优化内存传输路径，顶层实现异步常规和不规则应用中常用的高级操作。为了直接控制系统，程序员可以手动访问每个层。

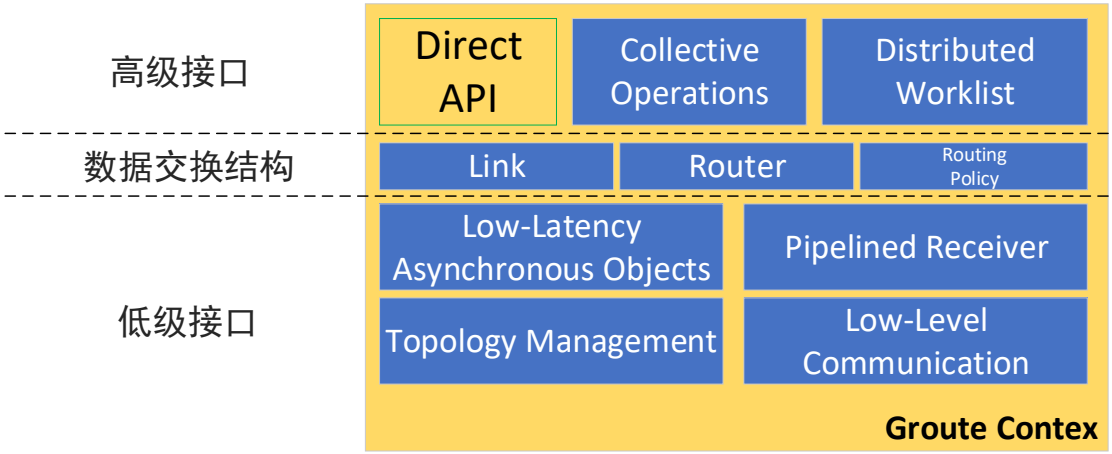

图 2-3 Groute 编程库

2.6.1 底层接口

低级层提供程序员可访问的接口，以实现高效的点对点传输。具体而言，该层提供低级通信 API 以减少延迟;流水线接收机，增加计算通信重叠;节点互连层次结构内省的拓扑管理和低延迟异步对象，以减轻系统开销。

低级通信接口提供主机和设备启动的内存复制功能，抽象化分组和条件传输。在 Groute 中，发送和接收操作被分割成数据包，以提高节点的整体响应能力，并实现多个设备之间的重叠通信。如果没有分组化，偶尔的小传输（例如，GPU 向 CPU 发送计数器）可能会受到常规大型传输（例如 GPU-GPU 传输）后面的线头阻塞的影响。

在低级接口之上，使用流水线接收器对象进行抽象异步通信，该对象通过使用双缓冲和三缓冲有效地利用了两个内存复制引擎和计算引擎。多缓冲的实现对读取缓冲区进行分配，排队虚拟的"未来读取操作"。将数据发送到流水线接收方

时，将从队列中删除读取操作，并将相应的缓冲区分配方式给发送方。同时，读取器会收到传入的挂起操作的通知，可以使用接收流等待或异步操作。

异步程序通常依赖于大量细粒度（非批量）同步点和实时内存分配来正常运行。为了最大限度地减少产生的驱动程序开销和非自愿的系统层级的同步，Groute 提供了几个低延迟异步对象，其中包括事件池(Event-Pool)和未来事件（Event-Futures）。Event-Pool 有助于通过预先分配的方式创建许多短期事件；而 Event-Futures 是实现未来/承诺模式的可等待对象，用于维护 CPU 和 GPU 之间的两层同步。特别是，事件未来处理已知将在将来记录但尚未创建的 GPU 事件。这些对象用作排队各种操作（如设备（CPU 和 GPU）的未来接收操作）的主要构建基块，并且可以与 CPU 线程或 GPU 流同步。

2.6.2 通信与调度

一个链路只能连接一对源端点和目标端点。在 Groute 中，有两种方法可以创建链接：直接创建或使用现有路由器。具体而言，每个链路应指定其最大可接收数据包大小和可能的流水线接收操作数。后者是可选的，如果未给出，则自动确定。Send 和 Receive 方法启动内存传输并返回 Event-Futures。特别是，接收操作将未来事件返回到 PendingSegment，其中包含可能尚未准备好进行处理的事件和分段。链路还提供类似套接字的 Shutdown 函数，指示不会发送进一步的信息。

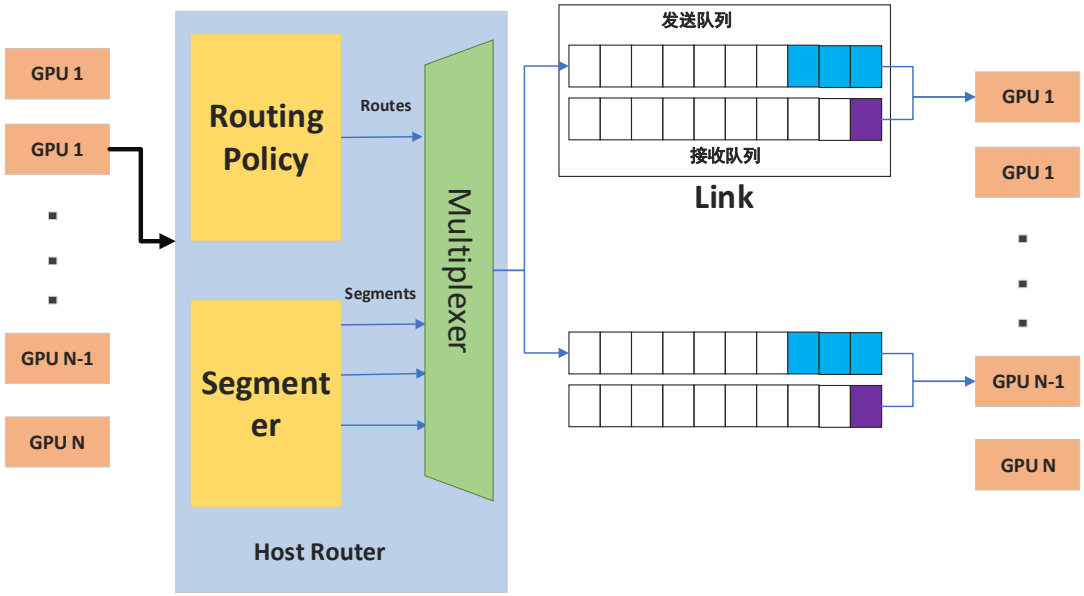

图 2-4 主机控制路由示意图

路由器的内部结构和工作流程如图 2-4 所示。该图显示路由器包含三个主要组件。Segmenter 组件控制根据目标设备功能将消息分解为分段；路由策略控制邮件目的地；多路复用器 Multiplexer 负责将消息分配给可用的 GPU。

给定要发送的消息，路由策略将依靠程序员回调确定其一个或多个目标。然后，路由器控制发送操作调度，根据可用性分配目的地。请注意，路由性能取决于底层拓扑。例如，在某些节点中，使用多对一操作进行简化是有效的，而在其他节点中，最好使用分层树进行并发规约。

在收到分段消息及其可能的目标后，调度由多路复用器管理。如图 8 的右上方所示，该实现涉及将发送操作排队到每个目标设备的发送队列。由于链路使用流水线接收器，因此设备也维护接收队列。如果发送操作和接收操作之间存在匹配项，则执行传输分配，从链接的发送和接收队列中取消一个项目的排队。此外，排队到其他设备的冗余发送操作在检查时被标记为陈旧并被每个设备删除。这可确保路由器不需要集中锁定机制，并且每个设备的队列是唯一应实现线程安全的构造。

2.6.3 分布式工作列表

Groute 提供的高级接口实现了可在多 GPU 应用程序中找到的可重用操作，例如广播和全规约。本节详细介绍了分布式工作列表的实现，其经常用于不规则

算法和图形分析。

分布式工作清单维护应处理的计算（工作项）的全局列表。反过来，每个这样的计算可能会创建排队到同一列表中的新计算。例如，广度优先搜索遍历节点的邻节点，通过为每个邻节点创建新的工作项来遍历整个图。

实现高效的分布式多 GPU 工作列表是一项具有挑战性的任务。由于每个设备可能包含输入数据的不同部分，因此只有某些设备能够处理特定的工作项。因此，分布式工作列表需要多对多的通信。Groute 使用路由器和总线拓扑，实现了分布式工作列表的管理。

在实施过程中，全局协调和工作计数由主机集中进行管理。在运行时，设备会定期报告生成和使用的工作项以进行跟踪。一旦总工作项数变为零，处理就会停止，并通过路由器链路向所有参与设备发送关机信号。

图 2-5 从单个设备的角度说明了 Groute 中分布式工作列表的实现。如图所示，工作列表是通过单个系统范围的路由器实现的。为了支持高效的全通通信，默认情况下，环形拓扑用于路由策略。除路由器外，每个设备还包含一个本地管理的工作列表，该工作列表由一个或多个用于本地任务的多生产者-单消费者队列组成。该实现由每个设备两个线程组成：工作线程和接收器线程，它们控制 GPU 间通信和工作项循环。

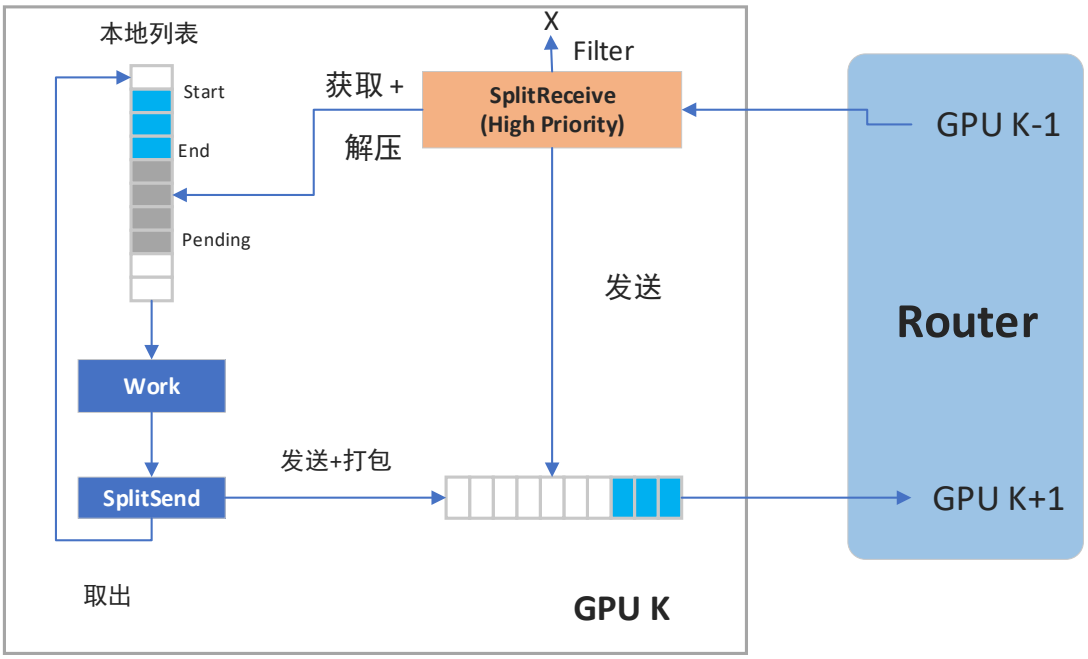

图 2-5 分布式工作清单实现

在环形拓扑中，图 2-5 中所示的工作流实现方式如下：每个设备从环拓扑的前一个设备接收信息。然后将接收到的数据进行过滤和分离（Split Receptive），将不相关的信息传递到下一个设备。与当前设备相关的项目被解压缩并"推送"到其本地工作列表中，向工作线程发出新工作即将到来的信号。同时，工作线程处理现有工作项，将生成的项分离到本地和远程工作（Split Send），并根据需要打包传出信息。请注意，Split Receive 内核在单独的高优先级流上排队。这会导致内核在现有工作处理期间被调度，从而提高系统的性能和响应能力。

为了在分布式工作清单上实现算法，程序员必须给出五个函数，如表 2-1 中所列：Pack、Unpack、OnSend、OnReceive 和 GetPrio。这些函数通常由一行代码组成，但若处理不慎可能会对工作项传播产生负面影响。在 OnSend 和 OnReceive 函数中，Flags 返回值是控制工作项目标（例如，传递到下一个设备、保留、复制或完全删除）的位图。然后，使用 GetPrio 获得的工作项的优先级用于在低优先级项之前安排较高优先级的工作，如下所述。

为了实现本地工作列表队列，Groute 使用 GPUbased 无锁循环缓冲区。此类缓冲区对于异步应用程序非常有用，因为它们消除了在运行时动态分配缓冲区的需要。如图 2-5 所示，每个工作列表队列由多个生产者和一个使用者组成。我们

的实现包含一个内存缓冲区和三个字段：开始、结束和挂起。工作消耗是通过原子增加起始字段来执行的。为避免消耗未就绪的项目，通过原子方式增加挂起字段来控制生产，从而在缓冲区中保留空间。在生产者完成追加其工作后，end 将增加一个写入器线程，与挂起进行同步。

表 2-1 分布式工作列表程序员回调函数

函数	描述
RemoteWork Pack(**LocalWork** item);	打包工作项并发送到另一个设备。
LocalWork Unpack(**RemoteWork** item);	解包接收到的工作项。
Flags OnSend(**LocalWork** item);	确定发送项目的目的地址。
Flags OnRecsive(**RemoteWork** item);	确定接收项目的目的地址。
Priority GetPrio(**RemoteWork** item);	获取接收项目的优先级。

对循环缓冲区的其他优化在 Groute 中完成。如果使用 GPU 流也产生工作（如在 SplitSend 中），则工作将通过预置信息（即减少开始）的方式推送到队列，从而避免生产者冲突。还值得注意的是，循环缓冲区使用 warp-aggregated atomics，通过将原子操作的数量限制为每个 warp 来提高追加工作的效率。

2.6.4 软优先级调度

在异步工作清单算法中应考虑的一个陷阱是中间值传播导致的冗余工作。与所有设备都同意算法中的全局状态的批量同步并行性相比，异步并发可能会由于滞后设备而传播过时的信息（"无用的工作"）。反过来，此类工作项会产生额外的中间工作，这些工作可能会随着设备数量的增加而呈指数级增长。

例如，在批量同步广度优先搜索（BFS）中，当前遍历的级别是全局算法参数。如果给定节点有两条路径，其中一条路径比另一条长，则仅注册源中边数最少的路径，并将最终值写入节点。但是，在异步 BFS 中，如果边缘数最少的路径位于滞后设备上，则将首先写入"不正确"路径（中间值）。反过来，这将使用中间值作为输入遍历图形的其余部分。具有短路径的设备完成其处理后，它将覆盖节点值，实质上是重新计算所有遍历的值。

缓解此问题的一种方法是为每个工作项分配软优先级，将怀疑是"无用工作"生成器的项推迟到稍后阶段，在该阶段中它们可能会被过滤掉。在 Groute 中，

软优先级调度是使用程序员提供的 GetPrio 回调实现的。在应用程序的运行期间，仅处理高优先级工作项，其中优先级阈值由系统范围的共识决定。完成所有可处理项目后，系统将修改阈值，分布式工作列表将处理每个设备上的延迟项目。使用软优先级调度可减少中间工作量，从而提高整体性能。

2.7 基本方案制定

本次优化实验拟对已有算法尝试从多个不同方向进行优化和对比，借助图形加速卡对单源点最短路径的求解过程进行并行加速，再次基础上，利用 Groute 多 GPU 异步编程模型完成算法优化，对完成的算法进行测试对比，得出优化结论。

2.8 本章小结

本章对已有的部分单源点最短路径算法包括迪杰斯特拉算法、贝尔曼-佛德算法和广度优先搜索算法进行了简单阐述；介绍了所用的开发工具和 CUDA 图形加速卡编程框架，对 CUDA 的基本原理进行了简单阐述；对 Groute 异步编程模型底层实现进行了详细表述；此外，制定了基本方案，为后续工作的进行划定了基本方向。

3 SSSP 算法优化优化设计

第二章简单介绍了单源点最短路径常见算法，尝试在本章节对图数据结构进行设计，对常见算法进行优化和实现。并在第五章对程序的执行结果进行对比和总结。

3.1 行压缩存储图（CSR）结构

常见图数据结构有邻接矩阵、邻接表、十字链表等。邻接矩阵用于处理图数据时，空间复杂度可达到$O(|V|^2)$，对于稀疏图，这种表示方法将会带来极大的无意义的开销。

邻接表为顺序结构和链式结构的结合，使用顺序结构表示每一个点，使用链式结构储存每一条边，空间复杂度为$O(|V|+|E|)$，用于表示稀疏图结构时，会比邻接矩阵节省很大储存空间。

十字链表也可用于储存图数据结构，与邻接表不同的是，十字链表提供了一种从边的目的节点访问边的方法，空间复杂度为$O(|V|+|E|)$。

这里提出一种高效的按行压缩的图存储结构（Compressed Sparse Row）。CSR 数据结构如下。

```
class csr_graph{
    index_t nodeNum,edgeNum; // 节点数量、边数量
    index_t *rowOffsets;  // 行偏移，即编号为 i 的节点的第一条边的索引
    index_t *columnIndices; // 列号，即边的目的节点
    distance_t *values; // 数值，即边的权值
};
```

CSR 时一种比较标准的图压缩储存数据结构，其本质是基于邻接矩阵的压缩存储，需要三类数据来表达：数值、列号以及行偏移。CSR 不是三元组，而是整体的编码方式。行偏移表示某一行第一个元素在 values 数组中的起始偏移位置，即某一个节点的第一条边在 values 中的存储索引；列号表示 values 数组中对应的值的列号，即 values 中边的目的节点；values 数组储存邻接矩阵中每个非无效

元素的值，即为边的权值。值得注意的是，为了便于图处理接口的使用，行偏移矩阵的数组长度为|V|+1，最后一个元素储存的是边的数量。

csr_graph 类提供了如下访问接口。

```
// 获取编号为vertex 的第一条边在values 数组中的索引位置
index_t csr_graph::getFirstEdgeIndex(index_t vertex);
// 获得编号为vertex 的最后一条边的下一个位置的索引
index_t csr_graph::getLastEdgeIndex(index_t vertex);
// 获取索引为index 的边的目的节点
index_t csr_graph::getEdgeTail(index_t index);
// 获取索引为index 的边的权值
distance_t csr_graph::getWeight(index_t index);
```

为了便于储存和访问，在实验过程中也尝试将已有的结构储存为 CSR 压缩结构并将其输出为压缩后的文件，便于后续的图结构的读入与创建。

3.2 迪杰斯特拉算法分析和优化

3.2.1 使用堆结构加速对距离最小值的查找

迪杰斯特拉算法的求解过程涉及到两个迭代过程中的循环：（1）当前最小距离节点的查找，（2）当前节点邻接点最短距离和路径的更新。

当前最小距离节点的查找可以使用推数据结构来进行加速。这里需要使用的结构为小顶堆。小顶堆为一个完全二叉树结构，其性质在于对于小顶堆中的每一个子树，总有根节点的元素值小于左右子树。其优势在于每次取出最小元素之后，仅需要很小的代价来恢复原有的小顶堆结构，维持一个小顶堆结构，可以在较低时间开销的情况下很快地获取每一轮迭代中的最小距离的节点，用于每一轮的迭代。

此外，可以很容易地发现，每一轮迭代中并不是需要完全检索每一个节点的当前距离，而是仅需要检索从开始到本轮迭代过程中触及的邻域空间即可。因此，可以将所有节点分为“已打开”节点和“已关闭”节点，在处理一个节点时，将其邻接点设置为“打开”，每次查找最小距离时，从“已打开”的节点进行查找即可。这样可以大大降低搜索空间，降低维护堆结构的时间复杂度和空间复杂度，

尤其是对于稀疏图，这一手段有着较强的加速效果。

3.2.2 基于 CUDA 的并行加速

CUDA 编程主要利用 GPU 执行的告诉并行性，对于迪杰斯特拉算法，其贪心性质不允许每一轮迭代并行执行，而最小距离的查找过程和当前节点邻接点的松弛操作是可以并行执行的，因此，欲使用 CUDA 加速迪杰斯特拉算法的求解算法，只能从这两个方面入手。这里很需要注意的是，使用 CUDA 加速时涉及到内存的拷贝操作，最好尽量减少内存拷贝过程，提高加速比。

（1）最小距离查找过程的并行加速

最小距离的并行查找可以使用分治的思想，运行多个线程，每个线程比较器比较两个距离并获得最小值，将最小值传递给下一级比较器，可以很快得到最小值。

（2）松弛操作的并行加速

松弛操作加速操作是简单直观的，为每一个邻接点创建一个线程用于执行松弛操作即可。

3.3 贝尔曼-佛德算法分析和优化

3.3.1 使用队列优化边的搜索空间

虽然 Bellman Ford 算法由于它要遍历所有的边，时间复杂度着实有点高了。认真思考后会发现，只有当 d[u]的值改变了，与之相连的 v 点 d[v]才有可能改变。根据该发现进行一个小的优化：建立一个队列，每次取出队首顶点 u，对 u 出发的所有边进行松弛操作（即判断 d[u]+dis < d[v]，对 d[v]进行优化。此时如果 v 不在队列中，就将其加入队列。因为 d[v]改变了，与之相连的边也可能改变）直到队列为空（说明图中没有后从源点可达的负权），或者是某个顶点的入队次数超过|V|-1（说明图中存在从源点可达的负环）。

优化后的算法伪代码如下。

Algorithm 4 optimized Bellman-Ford Algorithm

Input: G = (V,E), s ; {graph and source node}

Output: dist , pre; {shortest distance and path}

```
1: initialize: Set pre[u]⇐-1,visited[u]⇐ false,count[u]⇐0 for each u∈V,dist[s]⇐0
2: visited[s]⇐true
3: queue.push(s) {queue for BFS}
4: while queue is not empty do
5:     u⇐queue.pop()
6:     for each u=neighbor(u) do
7:         if dist[u]+|u,v| < dist[v] then
8:             dist[v]=dist[u]+|u,v|
9:             if v not∈queue then
10:                queue.push(v)
11:                count[v] ++ {times of visited,if count>|u,v|,there exists Negative weight loop}
12:            end if
13:        end if
14:    end for
15: end while
16: return dist,pre;
```

3.3.2 基于 CUDA 的并行加速

尝试对算法进行逐步的优化。

（1）引入单片 CUDA 内核，其中图形的每个顶点都分配给一个单独的线程。每个线程都会放宽由线程 ID 标识的顶点的所有输出边缘。 在这种方法中，GPU 块的数量是在运行时根据输入图中顶点的数量计算的。 这种类型的内核假定我们有足够的线程来覆盖输入图的所有顶点。

（2）在 CUDA 内核中使用了网格跨度。使用 blockDim.x*gridDim.x 计算步幅，它等于网格中的线程总数。例如，如果网格中有 1024 个线程，则线程 0 将在索引 0、1024、2048 等处处理顶点。此网格跨度循环方法可提供可伸缩性和线程重用。 它还确保没有线程处于空闲状态，并且每个线程都执行相同的工作量。

（3）为了进一步优化性能，引入了大小为|V|的布尔数组 F。对于图的所有顶点，此数组在开始时都初始化为 false。 源设置为 true。当顶点作为松弛的一部分以距源更短的距离更新时，其在 F 数组中的对应标志将设置为 true。 这表明该顶点的所有输出边缘在下一次迭代中需要进一步松弛。放松内核使用此信息，并且仅在将相应顶点的标志设置为 true 时才放松。这种方法与网格跨度循环相结合，减少了每个线程不必要的工作，因此确保了整体执行的进一步加快。

3.4 基于 Groute 多 GPU 编程模型的 SSSP 算法优化

使用 Groute 编程模型同样可以对贝尔曼-佛德算法进行优化，利用编程模型的高并发特性，可以对算法进行比较高效的优化，由于 Groute 可以同时操作多个图形显卡，算法的加速比也会随着图形显卡的数量增加而增加。

3.5 本章小结

本章先比较了各种图数据结构的优劣，确定了行压缩存储的图数据存储方式，此外还详细描述了迪杰斯特拉算法、贝尔曼-佛德算法的优化思路，也对 CUDA 对算法的加速方式进行了简述，本章节表述的算法将在第四章进行实现。

4 基于 SSSP 的 SSSP 算法优化和实现

4.1 基于 GPU 的迪杰斯特拉算法优化实现

尝试第三章提及的优化思路进行实现。由于需要使用堆结构获得每一个阶段的最小距离，考虑到 C++STL 中存在一个优先队列结构，该结构可以将入队列的元素进行排序，然后将最小或者最大的元素置顶，满足算法的需求，因此直接给予该结构尝试对算法进行优化。在不使用 CUDA 加速的情况下，优化成果比起传统迪杰斯特拉算法高不少。

尝试使用 CUDA 按照第三章提及的思路对算法进行加速，实时对算法的效率进行测试和尝试进一步，实现和优化过程如下。

（1）使用 CUDA 并行进行加速查找当前迭代的最小距离，使用 CUDA 并行加速当前节点邻接点的松弛操作。事实上，即使使用并行加速的比较过程，在使用稀疏图作为算例时，查找最小距离的效率仍旧比不上优先队列。

（2）仍旧尝试使用优先队列结构获得每轮迭代过程中的最小距离。由于 CUDA 核函数中不能直接调用 STL 库函数，因此仅将松弛过程编写成核函数使用 CUDA 进行并行加速。经过测试，由于涉及到多次内存的拷贝过程，算法的效率较低。

（3）自己将操作堆数据结构的实现为核函数以供核函数调用，将整个算法算法置入核函数中，一次性将图数据拷贝到 GPU 内存中，大大减少了内存拷贝次数，提高了算法的运行效率。

优化版迪杰斯特拉算法伪代码如下。

Algorithm 5 optimized Dijkstra Algorithm

Input: G = (V,E), s ; {graph and source node}

Output: dist

```
1: initialize: Set visited[u]⇐ false,dist[u]⇐INF for each u
2: dist[s]=0,push (0,s) to the heap
3: while heap is not empty do
4:     (u,distance)⇐heap.pop()
5:     if distnace=INF then
6:         break
7:     end if
8:     if visited[u]=true then
9:         continue
10:    end if[u]=true
11:    Relax Operation{Execution relaxation operations in parallel}
12:    cudaDeviceSynchronize() {wait for the GPU threads ending}
13: end while
14: return dist
```

4.2 基于 GPU 的算法贝尔曼-佛德算法优化实现

相较于迪杰斯特拉算法，迪杰斯特拉算法提供了最短路径实现的高效实现，而贝尔曼-佛德算法则拥有更高的并行性。使用 GPU 加速贝尔曼-佛德算法无疑可以获得更高的加速比。

首先尝试引入单片 CUDA 内核，为每一个顶点分配一个单独的线程。每个线程单独运行堆顶点的每一条出边指向的顶点执行送至操作。

在此基础上，在 CUDA 内核中使用了网格跨度。 使用 blockDim.x*gridDim.x 计算步幅，它等于网格中的线程总数。此网格跨度循环方法可提供可伸缩性和线程重用。它还确保没有线程处于空闲状态，并且每个线程都执行相同的工作量。

为了进一步优化性能，引入了大小为|V|的布尔数组 F。对于图的所有顶点，此数组在开始时都初始化为 false。源设置为 true。当顶点作为松弛的一部分以距源更短的距离更新时，其在 F 数组中的对应标志将设置为 true。这表明该顶点的所有输出边缘在下一次迭代中需要进一步松弛。放松内核使用此信息，并且仅在将相应顶点的标志设置为 true 时才执行松弛操作。这种方法与网格跨度循环相结

合，减少了每个线程不必要的工作，因此确保了整体执行的进一步加快。

算法核函数代码如下。

```
__global__ void relaxWithGridStrideV3(int N, int MAX_VAL, int *d_in_V, int *d_in_I, int *d_in_E, int *d_in_W, int *d_out_D, int *d_out_Di, int *d_out_P, int *d_out_Pi, bool *d_Flag){
        unsigned int tid = threadIdx.x + (blockDim.x * blockIdx.x);
        unsigned int stride = blockDim.x * gridDim.x;
        for (int index = tid; index < N - 1; index += stride){  // do index < N - 1 because nth element of I array points to the end of E array
            if (d_Flag[index]) {
                d_Flag[index] = false;
                for (int j = d_in_I[index]; j < d_in_I[index + 1]; j++) {
                    int u = d_in_V[index];
                    int w = d_in_W[j];
                    int du = d_out_D[index];
                    int dv = d_out_D[d_in_E[j]];
                    int newDist = du + w;
                    if (du == MAX_VAL) {
                        newDist = MAX_VAL;
                    }
                    if (newDist < dv) {
                        atomicExch(&d_out_Di[d_in_E[j]], newDist);
                        atomicExch(&d_out_Pi[d_in_E[j]], u);
                    }
                }
            }
        }
    }
```

4.3 Groute 编程框架下的 SSSP 算法优化实现

为了进一步加速单源点最短路径的求解算法，可以利用多 GPU 编程框架，更加充分地利用贝尔曼-佛德算法的并行优势，使用多个 GPU 的强大算力进一步加速单源点最短路径的求解过程。

Groute 编程模型提供了一个不规则应用程序的开发入口，还实现了通用而对集合操作和分布式工作列表，为程序编写提供了方便。

4.4 设计中考虑的制约因素

安全因素考量 我的方案考量了一定程度上的安全因素，云端服务器与本地机使用网络进行交互，在登陆远端服务器时，使用口令及密码进行登录，保证了远端运行环境的安全。

法律风险规避 方案的设计和实现过程均考虑了合法合规的要求，不存在违法的功能。所使用的开发工具、软件库、测试工具等均为正版、免费版本或者是遵守开源协议的开源版本，且仅用于研究使用，规避侵权相关的法律问题。

文化风险规避 在方案设计实现的过程中，保证了对代码、文档和测试数据的审查，代码、文档和测试数据中不存在不当言论，不以任何形式存在威胁、暴力、低俗、反动、歧视、侮辱性质的词汇或语句。

其他因素考量 除上述因素之外，同时在本地及远端搭建了除硬件外基本相同的实验环境，在本地编译程序后再使用云服务器进行测试，尽量降低了与服务器的网络交互频度。同时在本地和云端按照日期和版本编号备份开发过程中的开发代码，避免因为一些突发状况影响开发进度。

4.5 成本估算

采用 COCOMO 模型对本课题中的软件开发成本进行估算。

经过统计，源代码总行数为 2587 行，除去注释信息、debug 检查代码等无效代码，实际代码行数约为 1940 行，取 DSI=1940 行，即 KDSI=1.94。

COCOMO 模型计算公式如表 4-1 所示。本次开发项目较小，开发人员对目

标理解较为充分，受硬件约束较小，故采用组织型计算方式

表 4-1 COCOMO 模型参数

总体类型	工作量
组织型	$\mathrm{MM} = 2.4 * (KDSI)^{1.05}$
半独立型	$\mathrm{MM} = 3.0 * (KDSI)^{1.12}$
嵌入型	$\mathrm{MM} = 3.0 * (KDSI)^{1.12}$

计算可得工作量 MM 约为 4.81，即约 79.42 人日。

4.6 本章小结

本章对基于 CUDA 的迪杰斯特拉算法、贝尔曼-佛德算法的优化实现进行了详细描述，还对使用 Groute 编程模型实现 SSSP 求解算法的过程进行了分析论证。此外，还对此次设计种考虑到的制约因素进行了总结描述，并采用 COCOMO 模型对工作成本进行了大致估计。

5 性能测试与分析

5.1 测试用例

测试算法流程时，重点选择了美国道路相关的测试用例。测试算例相关信息如表 5-1 所示。

表 5-1 测试用例基本信息

	测试用例	节点数	边数	大小
小型算例	Simple Graph 00	10	34	1KB
	Simple Graph 01	1,000	18,555	137KB
	Simple Graph 02	5,000	122,984	1.19MB
	Simple Graph 03	10,000	489,476	4.79MB
	Simple Graph 04	5000	1,242,105	9.60MB
中等算例	San Francisco Bay Area	321,270	800,172	15.1MB
	Colorado	435,666	1057,066	20.3MB
	Florida	1,070,376	2,712,798	53.0MB
	Northwest USA	1,207,945	2,840,208	56.5MB
	New York City	264,346	733,846	13.7MB
大型算例	California and Nevada(CAL)	1,890,815	4,657,742	95.4 MB
	Central USA(CTR)	14,081,816	34,292,496	756 MB
	Eastern USA(E)	3,598,623	8,778,114	184 MB
	Great Lakes(LKS)	2,758,119	6,885,658	143 MB
	Northeast USA(NE)	1,524,453	3,897,636	78.4 MB
	Full USA(USA)	23,947,347	58,333,344	1.29 GB
	Western USA(W)	6,262,104	15,248,146	325 MB

上述数据除小算例外主要取自美国城市道路的距离信息图。

5.2 数据处理

从数据源得到的用例图文件时标准的 gr 版本，在这里需要对其进行处理和转化，使其满足实验需求。

对于有权图，编写程序调整文件中列出的边的顺序，使其按照出节点递增的顺序排列；为了方便处理，将文件读入创建图结构后，将图结构格式化输出为按

行压缩（CSR）格式，方便后续数据的读入，也降低了储存算例数据所需的存储空间。

5.3 测试环境

为了测试 Groute 编程模型在多 GPU 架构下的性能表现，在云端搭建了测试环境，测试环境具体参数如下。

OS: Ubuntu 20.04.4 LTS (GNU/Linux 5.4.0-96-generic x86_64)

CPU:Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50GHz

Memory:92GB

GPU: NVIDIA GeForce RTX 3080*2

Driver Version: 510.47.03 CUDA Version: 11.6

5.4 性能测试

使用中等算例对 Dijkstra 算法、CUDA 加速版本的 Dijkstra 算法、Bellman-Ford 算法、CUDA 加速版本的 Berman-Ford 算法以及 Groute 算法进行测试，为了对比算法优化的加速结果，将优化前后的结果进行对比。

5.4.1 Dijkstra 算法优化前后性能对比

Dijkstra 算法执行情况如所示。

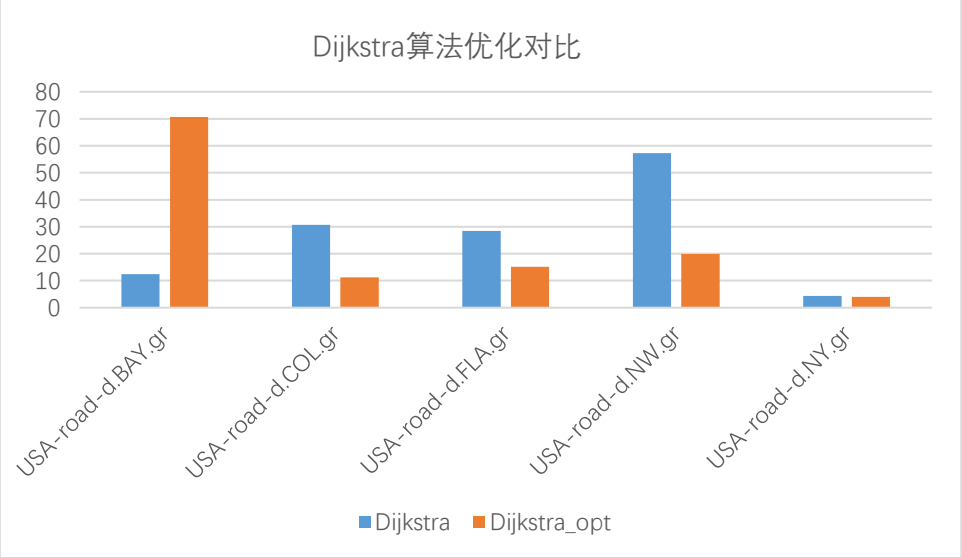

图 5-1 迪杰斯特拉算法优化前后对比

由图可见，对于较大规模算例，加速版本的算法执行效率明显优于未优化算法，在算例 Colorado 与 Northwest USA 甚至能达到三倍以上。整体加速比约为 1.75，可见 CUDA 对于 Dijkstra 算法的加速有着较为明显的效果。

对于较小算例，如 New York City ，优化效果并不明显，甚至在 San Francisco Bay Area 算例中还出现了加速比小于 1 的情况，这是由于优化的算法需要维护一定的复杂数据结构，以及包括内存拷贝、核函数调用等等，这些操作会带来一定的时间开销，对于较小算例，这些开销并不足以抵消其带来的加速效果，因此执行效率较原算法差，Dijkstra 本质是一个贪心算法，其每一轮迭代依赖于前一轮迭代的结果，算法的并行性并不算强，因而 CUDA 的并行加速效果并不能带来太多效率上的提升。

5.4.2 Bellman-Ford 算法优化前后的性能对比

相较于迪杰斯特拉算法，贝尔曼-佛德算法的时间复杂度高上一个量级，但其优势在于算法可用于求解负权图的最短路径。传统的贝尔曼算法执行效率较差，为了获得有效的可比较数据，这里采用随机生成的较小的算例对贝尔曼-佛德算法实现情况进行测试，算法执行情况如表 5-2 所示。

表 5-2 Bellman-Ford 算法性能测试

算例	Bellman-Ford	Bellman on GPU	加速比	备注
Simple-Graph-00	0.028ms	0.31ms	0.09	功能测试
Simple-Graph-01	270.345ms	8.227ms	32.86	简单图
Simple-Graph-02	8799.94ms	45.855ms	191.91	边稀疏图
Simple-Graph-03	68532.7ms	133.717ms	512.52	较大型图
Simple-Graph-04	84488.6ms	238.185ms	354.72	边稠密图

可见，对于具有一定规模的图，GPU 对于算法的加速比可以轻松达到 500 以上。Bellman-Ford 算法具有良好的并行性，GPU 设备的异构并行性可以很好地在该算法中得到发挥。

5.4.3 基于 Groute 编程模型的算法加速

Groute 编程模型充分发挥出了 GPU 的并行特性，算法的执行效率较原本的实现高上很多，因此使用大算例对算法进行测试。

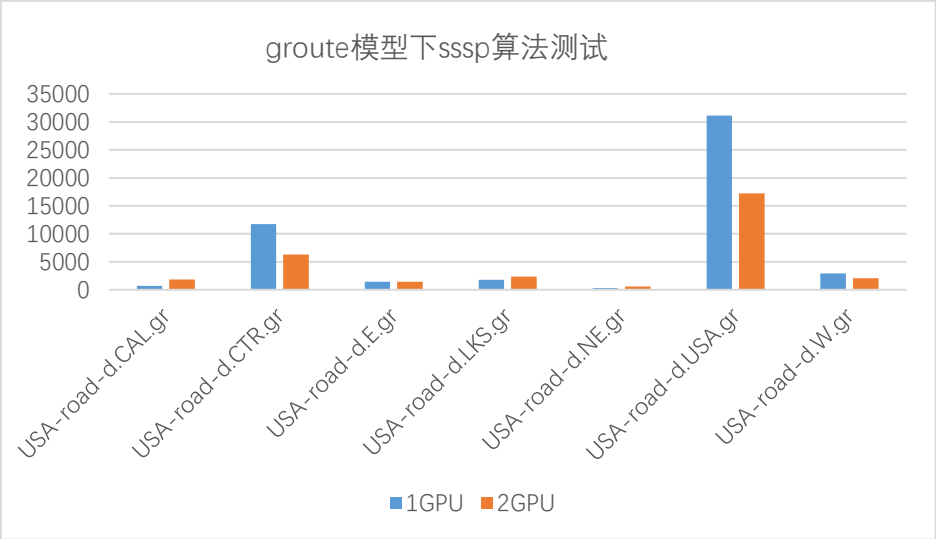

图 5-2 groute 模型下 sssp 算法性能测试

对基于 Groute 模式实现的程序进行测试，算法在单 GPU，双 GPU 下的执行情况如图 5-2 所示。

分析可知，双 GPU 在大型算例 Central USA、Full USA 中的加速比可以达到 1.8 以上，加速效果良好。但对于较小算例，Groute 模型的优化效果并不能体现出来，例如在 Eastern USA 和 Western USA 算例中加速比分别仅有 1.02 和 1.4。可见加速比与算例复杂程度有着明显相关关系。

分别取相应参数地最大值对算例尺寸、节点数、边数进行归一化操作，使用归一化结果和加速比作为横纵坐标可以得到加速比与算例复杂程度关系图如图 5-3 所示。

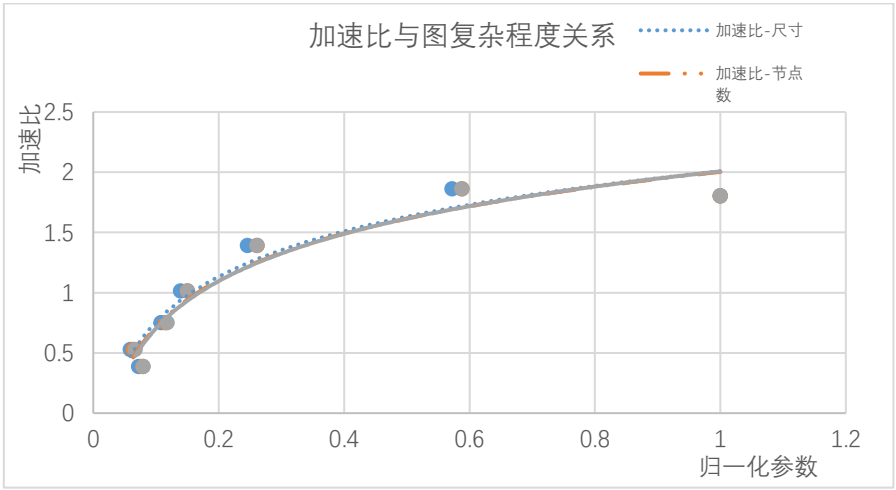

图 5-3 加速比-算例复杂程度关系图

分析可知，加速比与算例的复杂程度的关系曲线可用对数曲线进行拟合。对于较小算例（<200MB），双 GPU 并不能带来加速效果，这是由于 GPU 与 GPU 之间的通信成本较大，即使使用两倍的算力，也不能弥补通信处理成本，甚至还会出现效率更差的情况。在算例大小超过约 200M 时，GPU 数量的提升开始带来正向的算法效率的提升；在算例持续增大过程中，通信开销在整个计算过程中的占比逐渐变小，算法加速比提高，但加速曲线的斜率会逐渐变小，最终加速比会趋近于 2。

可见 Groute 编程模型能很好地协同多 GPU 的联合计算，但是通信开销依旧不小，当图数据结构较为简单时，该模型并不能获得较好的效果；但对于复杂算例，模型能带来不俗的效率提升。

5.5 本章小结

本章对不同的单源点最短路径算法进行了测试，比较了它们在优化前后的效率关系，得到了相关结论。GPU 的并行效果并不能在迪杰斯特拉算法中得到很好的应用，但其在并行效果良好的贝尔曼-佛德算法中可以得到很好地应用，算法效率得到了很大的提升。多 GPU 的协同工作会带来一定程度的通信开销，求解大、超大算例的单源点最短路径时，多 GPU 的处理效率才能体现出优势。

6 总结与展望

GPU 可以很好地实现算法的并行加速，在科学计算领域、实时计算领域都有着举足轻重的作用。当下开发者运用 GPU 算例的最常用编程环境为 CUDA 环境，使用 CUDA API 可以很好地利用 GPU 的并行异步特性，对计算过程进行基于低粒度并行计算的加速。

基于 GPU 的单源点最短路径算法优化问题，做了如下几点研究和工作。

（1）研究了传统的单源点最短路径算法，提出了在串行环境下的一些算法的优化思路，并对其进行了简单实现以及测试，获得了较为不错的提升效果。

（2）基于自己对于 CUDA 编程的理解，对传统的求解算法进行了低粒度计算的并行化，一定程度上提高了算法的执行效率。这一提升在贝尔曼-佛德算法中非常明显。

（3）测试了基于 Groute 编程模型的贝尔曼思想的单源点最短路径算法执行效果，对比了不同数量 GPU 和不同算例下的算法加速情况，体会到了多 GPU 通信开销带来的负面影响。

基于上述研究，得到了如下结论：

（1） 迪杰斯特拉算法的并行性有限，GPU 的并行特性对于迪杰斯特拉算法的优化效果有限；贝尔曼-佛德算法的并行性较强，并行加速对于负权图最短路径的求解意义重大。

（2） 多 GPU 协同工作中，GPU 之间的通信开销依然不可小觑，通信开销仍旧是多 GPU 机器的很大阻碍。

几个月时间，我做了许多，却也认识到自己的许多不足之处。

（1） 对 CUDA 编程的认识不够深刻，优化传统算法时没能最大限度地发挥 GPU 的并行优势。

（2） 毕设过程中不够主动，没有充分与导师交流。

（3） 能力和时间有限，没有时间探索更多的单源点最短路径求解算法。

随着科学并行计算需求的增长，下一代随着存储需求的爆炸性增长，下一代

异构 GPU 系统必须有更有效率的架构来应对。此外，多 GPU 协同工作具有很大意义，但其带来的通信延迟也亟需解决，从从硬件层面改进计算机总线架构，优化传输延迟，在保证访问安全性的前提下，实现 GPU 内存、Host 内存的统一编址和随机访问和或许能更好提升 GPU 应用于科学计算的性能。

致 谢

时光匆匆，转眼便是大学毕业时节，春花秋月，何曾想到离别的光景。毕业时间渐近，毕业论文的完成也随之进入了尾声。从开始课题到现在，一直都离不开导师、同学、朋友给予的热情帮助和鼓励，在这里请接受我最诚挚的谢意。

从懵懵懂懂到如今，回想自己的四年大学生活，从第一步跨入校门到即将离开，从第一节微积分到最后一节综合课设，从加入手绘社团到将部长职责转交给学弟学妹……四年，我经历了许多，也成长了许多。

也要深深感谢呵护我长征的父亲，每一次遇到困难，父亲宗室第一个给我鼓励的人，回顾 20 多年来走过的路，每一个脚印都浸满他无私的观岙和尊尊教诲。7 年的在外求学之路，寄托着父亲对我的殷切期望。

最后，向所有关心我的亲人、朋友和师长们表示深深的谢意。

参考文献

[1] Xiaohan Zhao, Alessandra Sala, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. Orion: Shortest path estimation for large social graphs. In Proceeding s of the 3rd Workshop on Online Social Networks (WOSN 2010), 2010.

[2] Xiaohan Zhao, Alessandra Sala, Haitao Zheng, and Ben Y. Zhao. Efficient shortest paths on massive social graphs. In Proceedings of 7th Internation a l Conference on Collaborative Computing: Networking, Applications and W orksharing (CollaborateCom), 2011

[3] P.-Y. Hong, L.-M. Huang, L.-S. Lin, and C.-A. Lin. Scalable multi-relaxatio n-time lattice Boltzmann simulations on multiGPU cluster. Computers & Fl uids, 110:1 – 8, 2015.

[4] E. Mej´ıa-Roa, D. Tabas-Madrid, J. Setoain, C. Garc´ıa, F. Tirado, and A. Pascual-Montano. NMFmGPU: nonnegative matrix factorization on multi-GP U systems. BMC Bioinformatics, 16(1):43, 2015.

[5] L. G. Valiant. A bridging model for parallel computation. Commun. ACM, 33(8):103–111,1990.

[6] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin. Memory access patterns: T he missing piece of the multi-GPU puzzle. In Proceedings of the Internatio nal Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, pages 19:1– 19:12. ACM, 2015.

[7] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens. MultiGPU graph ana lytics. CoRR, abs/1504.04804, 2015. URL http://arxiv.org/abs/1504.04804.

[8] 郭绍忠,王伟,周刚,胡艳.基于 GPU 的单源最短路径算法设计与实现[J].计算机工程,2012,38(02):42-44.

[9] Ruoming Jin et al. Hub-Accelerator: Fast and Exact Shortest Path Comput ation in Large Social Networks[J]. CoRR, 2013, abs/1305.0507.

[10] 吴漫,白明丽,曾咏欣,蒋峰,利叶斌.基于点割集的最短路径算法的改进与应用[J].数学理论与应用,2018,38(Z2):18-32.

[11] 向志华,赖小平.基于 BFS 算法的有阻断路径的最短路径算法研究[J].信息通

信,2019(11):41-42.

[12] Ben-Nun T,Sutton M,Pai S,et al.Groute: Asynchronous Multi-GPU Programming Model with Applications to Large-scale Graph Processing[J]. ACM Transactions on Parallel Computing, 2020, 7(3):1-27.

[13] 陈智康,刘佳,王丹丹,张运喜.改进 Dijkstra 机器人路径规划算法研究[J].天津职业技术师范大学学报,2020,30(03):30-35.10.19573/j.issn2095-0926.202003005.

[14] 曹大有,马斌.基于遗传算法的单源最短路径研究[J].汉江师范学院学报,2021,41(06):1-5.10.19575/j.cnki.cn42-1892/g4.2021.06.001.

[15] NVIDIA. NVIDIA Collective Communication Library (NCCL), 2016. URL http://www.github.com/NVIDIA/nccl/.

[16] Reuven Cohen and Shlomo Havlin. Scale-free networks are ultrasmall. Phys. Rev. Lett., 90, Feb 2003.

[17] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. Fast and accurate estimation of shortest paths in large graphs. In Proceedings of the 19th ACM international conference on Information and knowledge management, CIKM '10, pages 499–508, 2010.

[18] Xiaohan Zhao, Alessandra Sala, Haitao Zheng, and Ben Y. Zhao. Efficient shortest paths on massive social graphs. In Proceedings of 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011.

[19] N. Jing, Y. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. TKDE, 10(3):409–432, 1998.

[20] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. TKDE, 14(5):1029–1046, 2002.

[21] S. Shekhar, A. Fetterer, and B. Goyal. Materialization trade-offs in hierarchical shortest path algorithms. In SSD '97, 1997.

[22] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In 17th Eur. Symp. Algorithms (ESA), 2005.

[23] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Ne t works with Transit Nodes. Science, 316:566–, April 2007.

[24] R. J. Gutman. Reach-based routing: A new approach to shortest path algo r ithms optimized for road networks. In ALENEX/ANALC, pages 100–111, 2004.

[25] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search m eets graph theory. In SODA ’05, 2005.

[26] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In SIGMOD’08, 2008.