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 06 月 01 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。如何利用 GPU 异构并行性对算法进行加速优化成为一个非常重要的研究课题。最短路径问题是数据结构最进本但最关键的问题之一，在交通领域、社交领域甚至是广告营销、国土安全中都不乏其身影。使用 GPU 优化单源点最短路径算法，也是众多研究者孜孜不倦的重要研究课题。

针对上述课题，基于 GPU-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，使用不同的优化策略，提高了 Dijkstra 算法的并行性，降低了 Bellman-Ford 算法的冗余计算，提高了并行性。

实验结果表明，优化后的 Dijkstra 算法执行效率能在收缩检索域的基础上再提升约 40%左右，而拥有较强并行性的 Bellman-Ford 算法优化并行后较串行算法能够轻松达到约 300 的加速比。

关键词： GPU，CUDA 编程， 单源点最短路径，联合计算

Abstract

With the continuous improvement of the computing speed of GPU devices, GPU parallel acceleration technology is also increasingly used in scientific com puting, academic research and other fields. How to use GPU heterogeneous par allelism to accelerate the optimization of algorithms has become a very importa nt research topic. The shortest path problem is one of the most advanced but critical problems in data structures, and it is widely used in transportation, soci al networking, and even advertising and homeland security. Using GPU to opti mize single-source shortest path algorithm is also an important research topic o f many researchers tirelessly.

Aiming at the above issues, based on the GPU-CUDA programming enviro nment, the parallel heterogeneous acceleration of the typical SSSP algorithm Di jkstra and Bellman-Ford algorithm is implemented. Different optimization strate gies are used to improve the parallelism of the Dijkstra algorithm and reduce t he Bellman-Ford algorithm. The redundant computation improves parallelism.

The experimental results show that the execution efficiency of the optimize d Dijkstra algorithm can be improved by about 40% on the basis of shrinking the retrieval domain, while the Bellman-Ford algorithm with strong parallelism can easily reach about 300 times the efficiency of the serial algorithm after op timization and parallelism. Speedup ratio.

Keywords: GPU, CUDA Programming, SSSP, joint computing

目 录

摘 要 I
Abstract II
目 录 1
1 绪 论 1
1.1 课题背景 1
1.2 国内外研究现状 3
1.3 研究目的和主要内容 5
1.4 论文结构 5
1.5 课题来源 6
2 具体背景技术概述 7
2.1 迪杰斯特拉算法 7
2.2 贝尔曼-佛德算法 8
2.3 广度优先搜索算法 9
2.4 开发工具分析及选择 9
2.5 计算机统一设备框架（CUDA） 10
2.6 行压缩存储 14
2.7 基本方案制定 16
2.8 本章小结 16
3 基于 GPU 的 SSSP 算法优化设计 17
3.1 Dijkstra 算法分析和优化设计 17
3.2 Bell-Ford 算法分析和优化设计 22
3.3 本章小结 24
4 基于 GPU 的 SSSP 算法优化实现 25
4.1 基于 GPU 的 Dijkstra 算法优化实现 25
4.2 基于 GPU 的算法 Bellman-Ford 算法优化实现 28
4.3 设计中考虑的制约因素 33
4.4 成本估算 34
4.5 本章小结 34

5 性能测试与分析.....35
5.1 测试用例.....35
5.2 数据处理.....35
5.3 基准算法.....36
5.4 测试环境.....36
5.5 性能测试.....36
5.6 本章小结.....38
6 总结与展望.....39
致 谢.....41
参考文献.....42

1 绪 论

本章首先介绍了当前单源点最短路径算法的应用场景，简述了 GPU 并行化计算越来越广泛的应用，简述了 SSSP 算法的研究背景和发展趋势，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]。

并行计算是加速计算机运算速度的一种经典手段[3]，它可以将大问题分成许多小问题，各个小问题的计算是同时进行的，或者说程序的执行过程是同时进行的。与经典计算[4]相比，并行计算可以获得更高的计算性能，但一般需要特殊的硬件架构支持。到今天为止，并行计算已经逐渐成为计算机体系结构中占主导地位的流行方式。并行计算主要以多核处理器并行计算的形式出现，例如集群计算、大规模并行计算（MPPs）、网格计算、图形处理单元（GPU）等等。最近，单核处理器性能的进步几乎达到了物理极限，摩尔定律在提高更复杂算法的计算可行

性方面变得不那么有效了。因此，科学家和工程师必须将日益复杂的算法转移到并行计算架构上，以减少算法的实际处理时间。

在过去的几年中，由于用于数据并行计算的相对便宜和高性能的计算平台，特别是在海量数据集的医学图像重建方面，GPU 的数量得到了惊人的增长。GPU 上的通用计算 (GPGPU) 是最近并行计算研究中的一个明显的趋势。在 GPGPU 框架中，由于每个芯片的特殊设计，在一台计算机中使用大量的多个图形单元可以进一步提高 GPU 的并行特性。GPU 提供带来的高性能计算能力的一些优势是多 CPU 无法提供的[5]。

然而，GPU 或 GPGPU 上的高性能计算利用需要根据图形基元重新表述当前的顺序计算问题，图形处理器的两个主要 API 库 OpenGL 和 DirectX 已在 GPU 上支持这些问题的处理。这种从通用编程到 GPU 硬件的繁琐转换被 Sh/RapidMind、Brook 和 Accelerator 等晦涩的 GPU 编程语言所消除[6,7]。但是这些语言对于没有相应的编程培训的普通程序员来说很难应用。为了进一步为 GPU 程序员提供真正的便利，NVIDIA 的 Compute Unified Device Architecture (CUDA)、Microsoft 的 DirectCompute 和 Apple/Khronos Group 的 OpenCL 三个库为程序员提供了更可行的 GPU 编程框架，以简化了需要对数据转换为图形形式的过程，为利用 GPU 上的高性能计算速度提供了方便[8]。

实际上，SGI 公司的一个小组在 1994 年首先使用 RealityEngine2.5 在 Onyx 原始工作站上实现了基于 GPU 的图像侦察处理计算。但由于当时的图形硬件限制，SGI 图形硬件实现比 2004 年的单核 CPU 处理器慢 100 倍[9]。然而，这之后单核 CPU 处理器的性能发展比 GPU 处理器的多核慢得多。如今，GPU 已成为当前计算机用于图形处理的标准硬件部分，并被进一步设计为处理数据并行问题的相对独立的标准框架，GPU 可以很轻松地将单个元素数据分配给单独的逻辑核心以进行复杂处理[10]。如果在芯片与芯片的层面上对 GPU 与 CPU 进行比较，由于 GPU 的特殊架构，其在计算速度(FLOPS)和数据移动速度(GB/s)这两个关键指标上都可以拥有更优秀的处理能力，因此，GPU 和 CPU 之间的这种发展差异为研究人员重新考虑在 GPU 框架上并行计算图像应用程序提供了新的动力。通过将数据并行计算部分直接输入到 GPU 上，可以将一台计算机中的物理计算机

数量大大减少到最少。其好处不仅在于降低计算机设备成本，而且在任何机构、学校或医院内，整个系统运行所需的维护、空间、电力和冷却成本都更少。

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）[11]，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[12,13]。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。

1.2 国内外研究现状

现实世界网络中出现的许多问题都意味着计算最短路径及其从源到一个或多个目标点的距离。一些示例包括汽车导航系统[14]、交通模拟[15]、空间数据库[16]、互联网路线规划器[17]和网络搜索[18]。解决最短路径问题的算法计算成本很高，并且在许多情况下，商业产品实施启发式方法来生成近似解决方案。尽管启发式算法通常更快并且不需要大量的数据存储或预计算，但它们并不能保证最优路径。

对于单源点最短路径求解，最经典的算法为 Dijkstra 算法以及 Bellman-Ford 算法，前人对这两个算法的优化做出了大量的工作，并且许多有识之士提出了许

多新颖的求解算法，也对算法的并行化做出了大量贡献。

2012 年，郭绍忠等人进行了基于 GPU 的并行 Moore 算法的初步研究，针对显卡上动态数据的处理问题，在分析已有的并行 SSSP 算法之后，基于串行 Moore 算法，使用 GPU 对 Moore 算法进行了并行化优化与实现，并利用该算法思想求解单源点最短路径问题[19]。优化后的 Moore 算法有效降低了无意义的空线程开销、访问开销以及线程同步时间，一定程度上提高了求解效率。

2013 年 Ruoming Jin 等人提出了一种基于集线器节点（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[20]，用于计算 k 度最短路径。Hub-Accelerator 框架是一种以 Hub 点为中心的新技术，该技术能够通过大大限制 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，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[21]方法。

2018 年，吴曼等人基于图论理论，将割点与点割集的概念与经典的 Dijkstra 算法结合，形成了改进的并行算法，并进行了基于 GPU 的实现与应用，为计算无向图的最短路径提供了理论支持，并利用该算法改进了路由协议 OSPF 中的路由选择算法，降低了路由选路的时间开销[22]。

2019 年，针对大规模网络中的全源点最短路径的计算要求，向志华等引入了广度优先遍历中储存队列的思想，同时引入了阻断路径，限制了搜索域的展开范围，完成了图检索的剪枝操作，大幅度降低了最短路径求解过程的时间复杂度。经过测试，其设计的算法相较于传统 Dijkstra 算法在小、中、大规模的数据集上均可降低一半以上的运算时间。对于 BFS 算法，可以将运行时间优化到原来的 80%以下[23]。

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，使最短路径求解等算法的执行效率提高了 7 倍以上[24]。

陈智康等人将蚁群算法中的信息素思想加入到经典 Dijkstra 算法，用于计算在确定环境的二维地图中单个陆地机器人移动路径规划问题，实验结果表明，优化后的算法能够很大程度上减少路径规划中产生的冗余点，减少机器人单位寻路的计算代价[25]。

Harish 等人使用 CUDA 框架为 APSP 问题实现了两种算法：FW 算法和迭代算法，该算法通过改变源顶点重复计算单源最短路径 (SSSP)。他们表明，基于 SSSP 的算法比 FW 算法快大约六倍。但是，可以进一步改进该现有算法，以实现对 GPU 中可用内存资源的充分利用。例如，只使用片外存储器，因为没有从多个 SP 同时访问的公共数据。因此，可以使用更高速度（但很小）的片上共享内存来节省片外内存和 SP 之间的带宽[26]。

前人的一系列工作，为 SSSP 求解提供了大量有效的并行算法，但时至今日，对于基于 GPU 并行的 SSSP 算法优化实现的重难点仍旧集中在如何提高算法的并行性和尽量减少冗余计算等几个方面。

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，探索和优化出新的 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，介绍了基于 GPU 的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，分析了当前面临的问题和挑战，提出了优化基于 SSSP 算法的几个基本方向。

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，此外，也对算法程序进行了选择。

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，。最后对设计实现过程中针对制约因素做出的考虑，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，对整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，并对 GPU 并行技术的未来发展提出了期望。

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，这是以贪心思想为基础的算法。

迪杰斯特拉算法也是一个迭代算法，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。算法伪代码如算法 1 所示。

算法1 Dijkstra算法

```
输入：G=(V, E) ,s;                                    //图数据和源节点
输出：dist, pre                                       //最短距离和路径向量
1：  for each u ∈ V do
2：        dist[u]←∞；pre[u]=-1；visited[u]=false     //初始化
3：  end for
4：  dist[s]=0
5：  while true do
6：        Find the vertex u∈ V which have the min dist[u] and visited[u]=false
7：        if dist[u]=∞ or each visited = true then
8：              break
9：        end if
10：        visited[u]←true
11：        for each v ∈ neighbor(u) {(u,v)包含在边集中} do
12：              if dist[v] ⩾ dist[u] + |u,v| then
13：                    dist[v] ← dist[u] + |u,v|
14：                    pre[v] ← u
15：              end if
16：        end for
17：  end while
18：  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

Bellman-Ford 算法类似于 Dijkstra 算法，以松弛操作为基础，即用更小更准确的最短路径值替代估计的最短路径值，在多次迭代过程中逐渐得到最优解。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，该过程一共需要执行|V|-1 次（|V|为节点的数量）。在迭代计算过程中，已计算得到正确的距离的节点的数量会逐渐增多，直到所有的点都通过迭代计算得到了最短的路径。这样的策略使得贝尔曼-佛德算法适用于带有负权边的图。而迪杰斯特拉算法没有这样的特性，因而无法用于求解负权图。

算法伪代码如算法 2 所示。

算法2 Bellman-Ford算法

```
输入：G=(V, E) ,s;                                  //图数据和源节点
输出：dist, pre                                     //最短距离和路径向量
1：  for each u ∈ V do
2：        dist[u]←∞；pre[u]=-1；visited[u]=false   //初始化
3：  end for
4：  dist[s]=0
5：  for i=1 to |V| do
6：        for each (u, v) ∈E do
7：              if dist[v] ＞ dist[u] + |u,v| then
8：                    dist[v] ← dist[u] + |u,v|
9：                    pre[v] ← u
10：             end if
11：       end for
12：end for
13：return dist, pre
```

贝尔曼-佛德算法需要执行|V|-1 轮循环，循环中图中各个点的各条出边执行松弛操作。若图中存在负权回路且回路中的点与源节点连通，则该算法无法收敛，不存在最短路径。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，但仅适用于无权图。算法伪代码如算法 3 所示。

算法3 BFS算法

```
输入：G=(V, E) ,s;                                  //图数据和源节点
输出：dist, pre                                     //最短距离和路径向量
1：  for each u ∈ V do
2：        dist[u]←∞；pre[u]=-1；visited[u]=false   //初始化
3：  end for
4：  dist[s]=0，xisited[s]=true，queue.push(s)
5：  while queue is not empty do
6：        u ← queue.pop()
7：        for each v = neighbor(u) do
8：              if visited[v]=true then
9：                    continue
10：             end if
11：             visited[v]=true
12：             dist[v]=dist[u]+1
13：             pre[v]=u
14：             queue.push(v)
15：       end for
16：end while
17：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。BFS 算法相较于 Dijkstra 算法与 Bellman-Ford 算法而言，其并行性较好且易于理解，但该算法仅用于无权图最短路径的求解，其应用领域远远不及上述两个算法。

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)编程环境，CUDA 是 NVIDIA 提出的面向 C/C++编程人员的编程接口环境。

为了完成此次算法优化研究，分别在本地和云端分别配置了 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）

随着从单核处理器到多核处理器的过渡基本完成，现在几乎所有的商用 CPU 都是并行处理器。提高并行度，而不是提高时钟频率，已经成为处理器性能增长的主要方式，而且这种趋势很可能会持续下去。这就提出了许多重要的问题，即如何有效地开发高效的并行程序，这些程序将在日益并行的处理器上很好地扩展[27]。

一段时间以来，现代图形处理单元 (CPU) 一直处于提高芯片级并行性的前沿。当前的 NVIDIA CPU 是多核处理器芯片，从 8 个内核扩展到 240 个内核。这种程度的硬件并行性反映了 CPU 架构不断发展以适应实时计算机图形的需求这一事实，这是一个具有巨大内在并行性的问题域。随着 GeForce 8800（第一个基于 NVIDIA 的 Tesla 统一架构的 CPU）的出现，可以直接将普通的 CPU 程序实现为大规模并行处理处理程序，而不仅仅是显示图形的计算。

NVIDIA 开发了 CUDA 编程环境，为程序员提供了 C/C++编程的扩展支持。CUDA 编程模型指导程序员公开足够的细粒度并行性，以充分利用大规模多线程 CPU，同时提供跨 CPU 设备范围内可用的广泛物理并行性的可扩展性。因为它提供了一个相当简单、极简的并行抽象并继承了所有众所周知的 C 语义，它让程序员可以相对轻松地开发大规模并行程序。

2.5.1 CUDA 架构

CUDA 的软件架构如图所示，CUDA 编程环境框架可以分为三个层次：编程接口（API/CUDA 库）、运行时依赖（Runtime 库）和设备驱动（CUDA 驱动器）。

如图 2-1 所示。

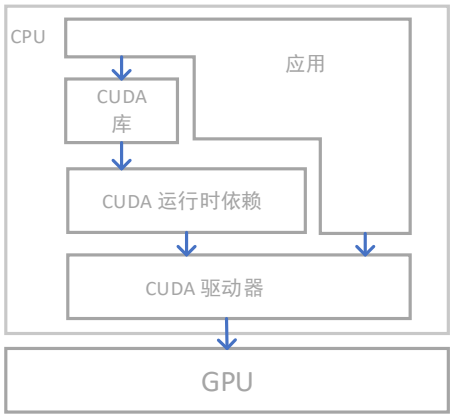


图 2-1 CUDA 软件架构

编程接口层主要用于提供应用开发依赖库，负责处理和解决大规模并行计算过程。运行时依赖库（Runtime）提供 CUDA 程序开发和运行时需要的组件和环境，其中包括定义的不同的基本数据类型、设备驱动入口、转换数据类型接口、调度方法函数等等。

2.5.2 CUDA 线程和内存结构

如图 2-2 所示为 CUDA 的程序模型。基于该模型，可以将一个程序分为两个相互不影响的并行执行部分：主机端（host）部分以及设备（device）端部分。主机端的执行流程是 CPU 执行的程序流，设备端的执行流程是 GPU 上执行的程序流。标准的编程流程是，CPU 首先执行主机端的程序，准备好待处理数据，将其复制到 GPU 显存中，然后从主机端调用设备函数启动 GPU 处理流，此时 CPU 可以并行执行一些与 GPU 无关的操作。当设备端的程序流执行完毕后，主机端程序将 GPU 显存中处理完成的数据拷贝回 GPU 端，并输出结果或者执行下一步的操作过程。

CUDA 是一种基于 C 语言扩展的简单易用的一种编程规范模型，该模型将传统的处理器（CPU）和并行图形加速卡（GPU）统一到一个异构的计算机系统中，实现在两者间灵活自如的调度过程。具体来说，CUDA-C 包含三种类型的函数：

(1) host 函数，主机端运行的函数，由 CPU 负责调用执行。

(2) kernel 函数，_global_标识符定义的函数，由主机端传入相关参数调用，启动 GPU 设备并执行，是 CPU 启动 GPU 并行计算的入口，相当于是 GPU 的 m

ain 函数。

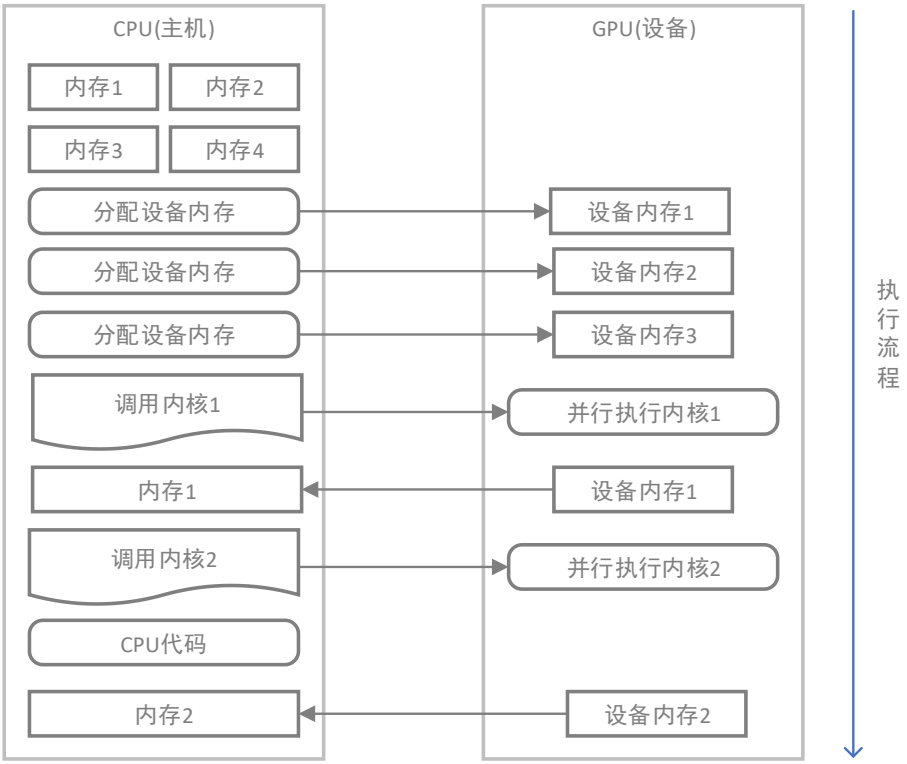


图 2-2　CUDA 计算模型

(3) device 函数，由_device_标识符定义的函数，由 global 函数或者 device 函数调用。

一般来说，串行操作过程应被实现为 host 函数，而并行化操作可被实现为 kernel 函数或者 device 函数。

事实上，一次核函数调用可以在 GPU 上并发执行大量的线程，具体数量取决于传入的参数，受限于 GPU 的计算能力，一个 grid（网格）对应于一个 GPU 或者多个多处理器。而一个 block（块）中所有的线程会在一个流式处理器（SM）上并发地执行。与操作系统对 CPU 管理过程不同的是，当占用 SM 的 block 中的所有线程都执行完毕之后，GPU 才会为该空闲的 SM 激活并分配一个新的块继续执行，即 GPU 中处理器的调度方式为不可剥夺式，而 CPU 的管理方式多为分时可剥夺式，这在一定程度上降低了进程和线程切换开销，提高了效率。

CUDA 线程被组织为从低到高的 Thread-Block-Grid 层次结构，Block 和 Grid 维数和尺寸受到 GPU 计算能力的限制。这样的线程组织软件架构也同样体现在

硬件架构上。一个 GPU 中包含若干流式多处理器（SM），而每个多处理器由多个流处理单元（SP）组成。每个流处理单元可以使用限定的计算资源独立地执行一个计算任务。整个 GPU 系统运作于单指令多线程（SIMT）体系之上。

由多个 Thread 组成的 Block 作为 SM 调度和激活的单位，一个 SM 可以同时调度一个较大或者多个较小的 Block，但不能对其进行拆分，只调度一个 Block 中的部分。CUDA 为同一个 Block 里的线程引入了同步机制，并且他们可以通过访问同一个块中共有的内存进行通信或者数据传输。SM 在调度过程中，会将每个线程与每个流处理单元（SP）建立一一映射关系，每个 SP 拥有各自的寄存器，是一个功能较为完善的处理核心，可以单独执行任务，这也是 GPU 高并发性能的来源，但与普通 CPU 处理核不同之处在于，GPU 的流处理单元在硬件架构方面对于一些常用科学计算函数定义了专有的数学计算指令，如三角函数计算指令 sinf、指数计算函数 powf、绝对值计算指令 sqrtf。对于某些精度要求不是那么高的计算过程，也可以采用精度差点但速度更快的优化版计算指令，如__sinf、__powf、__expf 等等。

在 CUDA 架构中，按层划分内存的机制是一种值得学习的架构技术。CUDA 将内存分为块内本地内存、共享内存和全局内存三个层级。块内本地内存以 Thread 为访问单位，共享内存以 Block 为访问单位，全局内存以 Grid 为访问单位。内存与线程组织对应的层次架构逻辑如图 2-3 所示。

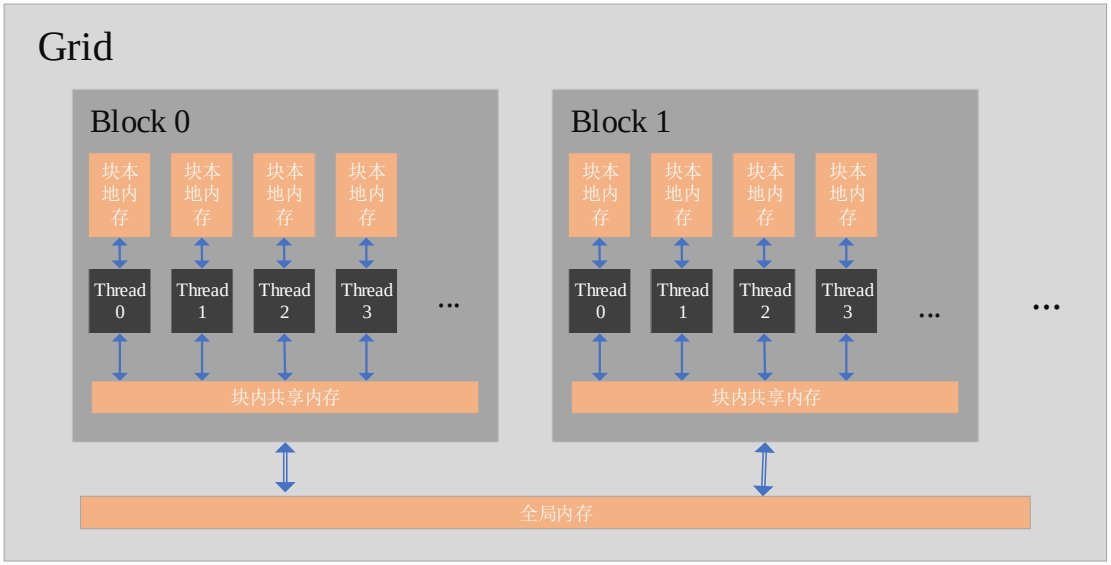


图 2-3 层次内存架构

每一个线程都拥有自己独立的块内本地内存，同一个块内的线程共享同一个块内共享内存，一个网格中的所有块中的所有线程共享同一块全局内存。这种层次结构也影响了访问各层内存的效率，由于多线程访问时需要互斥读写，因此可访问的入口越多，内存的访问效率就越差。对于该结构，内存访问效率有：块内本地内存 > 共享内存 > 全局内存。

另一方面，GPU 的内存也设置了多内存设备联合编址的机制，类似于冯诺依曼架构中内存的低位多体交叉架构，但 GPU 中内存的内存设备数量要大得多。因此对于并发的线程，按照内存地址线性进行访问，可以充分利用设备读写带宽，提高访存效率。

2.5.3 单指令多线程（SIMT）模式

CUDA 使用单指令多线程 SIMT（Single Instruction MultiThreads）架构实现了 GPU 的并行多处理器的线程管理过程。SIMT 与单指令多数据 SIMD 的 CPU 向量式组织结构相似，但不同之处在于，SIMT 是用一条指令处理多个线程，处理不同的数据，而 SIMD 使用同一条指令操作同一个线程处理不同的数据；SIMT 的每一个处理逻辑单元都是接近于完整的单元，而 SIMT 仅由一个控制器同时处理多个相同的逻辑单元；因此 SIMT 可以对单独的线程执行不同的处理，降低线程细粒度，但 SIMD 不可以。

2.6 行压缩存储

常见图数据存储结构有邻接表、十字链表、邻接矩阵等。邻接矩是一个二维数组，使用元素的行列位置来表达图的拓扑结构，优点在于所及访问效率高，但空间复杂度高达$O(|V|^2)$，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 是一种图压缩储存结构，其本质是对邻接矩阵进行压缩存储，需要个数组来表达图结构：各行偏移数组、列号索引以及边权值。

行偏移的第 i 各元素表示第 i 行第一个元素在 values 以及 columnIndices 数组中的起始偏移位置，即编号为 i 的节点的第一条边在 values 和 columnIndices 数组中的下标索引；列号表示对应的元素的列号，即边的目的节点；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，对于任何一个节点 i，可以用 rowOffset[i]和 rowOffset[i+1]来获得其第一条边和最后一条边的储存位置。

若邻接表数据域、指针域占用相同的储存空间，则 CSR 压缩储存结构可以比邻接表节省约$\frac{|V|*2+|E|*3-|V|+2*|E|}{|V|*2+|E|*3}=\frac{|V|+|E|}{2|V|+3|E|}$，对于大多数图数据，该储存结构可以降低约 40%的空间占用。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。

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，借助图形加速卡和 CUDA 编程环境对单源点最短路径的求解过程进行并行加速，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；对选择的图储存结构进行了描述，此外，还制定了基本方案，为后续工作的进行划定了基本方向。

3 基于 GPU 的 SSSP 算法优化设计

第二章将介绍单源点最短路径常见算法，对 Dijkstra 和 Bellman-Ford 算法进行分析并提出优化方案，本章节的优化方案将在第四章中得到实现。

3.1 Dijkstra 算法分析和优化设计

Dijkstra 算法是一个贪心的串行算法，提高算法的并行性一直是基于 GPU 实现并行 Dijkstra 算法关注的重难点。Dijkstra 算法的求解过程是一个重复的迭代过程，每轮迭代过程需要完成以下两个操作：（1）检索：查找最短距离未确定的节点中拥有当前备选最小距离的节点；（2）松弛：针对该节点的每一条出边执行松弛操作。重复该过程，即可在 n 次迭代后，确定源点到所有节点的最短距离和最短路径。

3.1.1 标记节点状态收缩检索域

对于传统表述的 Dijkstra 算法，操作（1）将在整个 dist 数组中查找最小值，但事实上，检索过程的搜索域可以依据执行过程进行限缩。可以将图的所有节点描述为以下三种互相对立的状态——“未打开”、“打开”、“已关闭”。已关闭状态表示该节点已经确定最短距离，每轮迭代过程会将检索到的节点状态置为“已关闭”并对其执行松弛操作。打开状态表示当前已关闭状态节点可直接到达的节点，即已所有已关闭节点的每一条出边指向的节点。未确定最短距离且当前已关闭节点不可直接到达的节点即为未打开状态。可以很容易发现，根据 Dijkstra 的执行过程，检索过程的搜索域可完全等价于“打开”节点的集合。在迭代开始前，源节点将被置为打开状态，接下来的检索和松弛过程将在已打开节点的集合中进行，每轮将检索到的节点从已打开节点集合中移出置入已关闭节点集，执行松弛操作时将出边指向的未打开节点集中的节点移入到已打开节点集合，重复迭代直到已打开节点集为空即可。

对于如图 3-1 所示的图数据，记未打开、打开、已关闭的节点集合分别为集合 U、O、C。开始时集合 U 将包含图中所有节点，假设源节点为 0，首轮迭代前，编号为 0 的节点将被从 U 中取出添加到集合 O 中。迭代执行时，检索过程

将从 O 中取出节点 0，将其加入集合 C，对于节点 0 的每一条出边指向的所有集合 U 中的节点 2、3、4，将会执行松弛操作更新 dist 信息并将 2、3、4 节点从集合 O 中取出添加到 O 中。下一轮迭代将从（2、3、4）中查找最小距离节点并继续迭代过程。

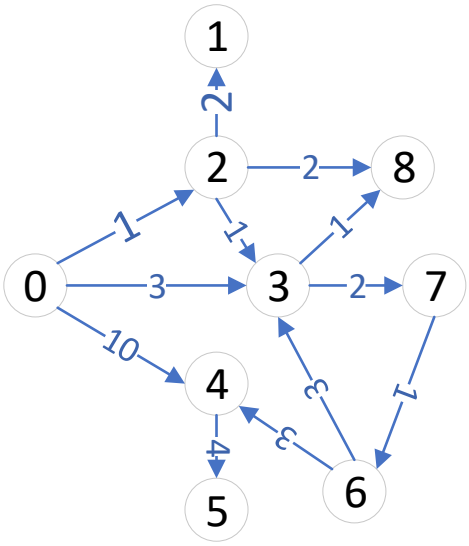


图 3-1 例图

3.1.2 串行化堆结构的引入

对于检索步骤，若采用朴素的顺序查找方式，时间复杂度为 O(n)。为了提高查找效率，考虑到集合 O 的的动态性，可以为集合 O 维护一个小顶堆结构，用于加速检索过程。维护堆的时间复杂度为 O(logn)，获取堆中最小元素的时间复杂度为 O(1)，小顶堆结构可将最短距离查找过程的时间复杂度降低到 O(logn)，可在一定程度上提高算法时间效率。

3.1.3 基于评估值的已打开节点的并行批量松弛

GPU 计算的优势在于程序的高度并行性，然而 Dijkstra 算法是一个贪心的迭代过程，每一步仅仅可以扩展当前确定的最短距离的邻接点，这大大限制了算法的并行性。如何提高算法的并行性成为提升并行 Dijkstra 算法效率的重点，当然这也是难点。

对于无权图的 SSSP 求解的典型算法 BFS，有着与 Dijkstra 相似的节点打开过程，BFS 的每一轮迭代过程中，由于无权图的特性，相同层级的节点具有相同

的与源节点的距离，因此这些节点可以并行进行邻域的扩展。与之不同的是，Dijkstra 算法是一个贪心的迭代过程，每一步仅仅可以扩展当前确定的最短距离的邻接点，算法只能串行执行。

为了在 Dijkstra 算法中实现与 BFS 相似的并行扩展过程，可以很容易想到的是，Dijkstra 算法中的一轮迭代之前，具有相同的暂定最小距离的节点可能不止一个，这时候就可以为这些节点执行并行扩展，这样可以一定程度上提高算法的并行性。但是对于大多数图数据，这种方法的提升效果微乎其微。

为了进一步提高并行性，这里可以采用更激进的策略：设定一个与距离相关的评估值，所有当前距离小于该评估值的节点都将得到扩展。在每轮迭代前，计算该评估值，所有当前距离小于该评估值的已打开节点会并行执行松弛操作。此方法可以很好提高 Dijkstra 算法的并行性，提高 GPU 执行 Dijkstra 算法的执行效率。

（A）评估值计算

这里引入一个一组符号 D(i)、dist(i)，D(i)表示第 i 轮迭代过程中的评估值，dist(i)表示节点 i 的当前暂定距离。节点被分为三个状态：未打开，已打开，已关闭，分别用 U、O、C 集合表示，U(i)中储存第 i 轮迭代中尚未确定距离并且未打开的节点，O(i)中储存示第 i 轮迭代中的边界节点，C(i)中储存第 i 轮迭代中已经确定距离的节点。

第 i 轮的评估值计算公式如下。

$$D(i) = \min\{\, dist(u) + \min\{\, |u,v|\}, u \in O(i), v \in U(i)\,\}$$

每轮迭代开始前计算以获得 D(i)，对于 O(i)中每一个 dist(i)小于 D(i)的节点，可以确定其距离，将其从 F 中取出放入 C 中，并且使用并行算法对这些节点进行邻域扩展操作。

若节点 u 没有出边，则可以不参与计算，直接进行标记，下一轮迭代中可直接确定最短距离并从集合 O 中取出加入到集合 C 中。

对这些节点进行扩展，可以保证得到正确的最短距离值，这里可以用反证法和归纳法来证明算法的正确性。

例如对于如图 3-2 所示图数据。

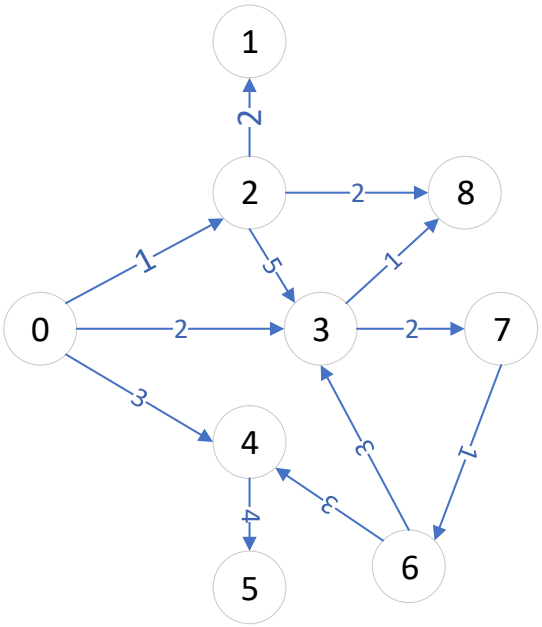


图 3-2 例图数据

以编号为 0 的节点为源节点，则第一次迭代打开的节点为 0，一轮迭代过后，节点 2,3,4 将被打开，dist[2]=1，dist[3]=2,dist[4]=3。

计算可得 D(1)=min{1+2,2+1,3+4}=3。暂定距离小于 3 的节点由 2、3、4，下一轮迭代，节点 2、3、4 将并行执行扩展和松弛操作。算法的整个执行流程如表 3-1 所示。

表 3-1 迭代计算过程

迭代次数	dist 数组									D	已关闭节点 C	已打开节点集 O	未打开节点集 U
	0	1	2	3	4	5	6	7	8				
0	0	∞	∞	∞	∞	∞	∞	∞	∞	1		0	1-8
1	0	∞	1	2	3	∞	∞	∞	∞	3	0	2,3,4	1,5-8
2	0	3	1	2	3	7	∞	4	3	5	0,2-4	1,5,7,8	6
3	0	3	1	2	3	7	5	4	3	5	0-5，7-9	6	
4	0	3	1	2	3	7	5	4	3		0-8		

可见算法在例图数据中运行可在 4 轮并行迭代中快速确定 8 个节点的最短距离。大大减少了迭代次数。

（B）批量打开与并行扩展的正确性证明

假设当前为第 i 轮迭代，并且前面各轮的迭代过程均正确确定了每个已关闭节点的 dist。未打开节点集为 U(i)，打开的节点集为 O(i)，当前 D(i)可以通过公式计算出来。若存在一个已打开节点 u∈O(i)，且 dist(u) < D(i)，即节点 u 将在此轮迭代中确定距离 dist(u)，记正确的距离值为 dist_correct(u)，若 dist_correct(u) ≠ dist(u)，则计算过程出错。假设 dist_correct(u) < dist(u)，则说明这一轮打开的节点中存在一个节点 v，存在 dist_correct(u) = dist(v) + |v,u| < dist(u) <= D(i)，但是这与 D(i)的定义相矛盾。

迭代过程是一个缩小备选距离值得过程，因此也不可能出现 dist_correct(i) > dist(i)的情况。因此必定满足 dist_correct(i) = dist(i)。

由上述证明可知，若第 i 轮迭代正确执行，则第 i+1 轮迭代也将正确执行。第一轮迭代中，仅源节点 source_node 被打开，确定的最短距离为 dsit[source_node] = 0 = dist_correct(source_node)（初始化过程中赋值），第一轮迭代必定正确执行。

归纳即可证明算法的正确性。

（3）性能分析

依据评估值对节点进行批量扩展的算法比较充分地提高了 Dijkstra 算法的并行性，在每一轮迭代过程中，可以并行扩展多个节点。假设平均一次可以扩展 k 个节点，算法可以将原本算法的效率提升 k 倍，k 的大小与图数据有关，经过测试，对于美国道路图数据，k 值约为 2 左右。

3.1.4 评估值的并行计算

基于节点的批量打开方式，每一轮迭代过程之后，每个节点的 min{dist(u)+|u,v|}都会发生改变，因此不能使用小顶堆对其进行实时维护。这里可以采用分治思想对数据进行并行规约比较，可将检索计算评估值的过程降低到 O(logn)的时间复杂度，提高算法的执行效率。

3.2 Bell-Ford 算法分析和优化设计

Bellman-Ford 算法用于计算有权图从起始节点到其他节点的最短距离和路径，它比 Dijkstra 算法更灵活，并且能够处理具有负权边的图数据。Bellman-Ford 算法的复杂性需要从最优或者最坏的情况分别讨论。

最好的情况下，初始化节点距离和源节点距离需要 O(n)时间。第一次迭代耗时 O(n)，起始所有迭代需要恒定时间来完成一次松弛操作，需要对于每一个节点的每一条出边执行上述操作，整个算法时间复杂度为 O(VE)。

最坏情况下，前两次迭代花费与最佳情况下相似的时间，并且所有剩余的迭代将会花费最大 O(n)时间来对 n 个任何节点的边执行松弛操作。该算法的时间复杂度将达到 $O(V^2)$。

Bellman-Ford 算法拥有着良好的并行性。但不可否认的是，在每一轮运算过程中，都执行了许多无意义的冗余操作，如何避免冗余计算，同时充分利用 Bellman-Ford 算法的并行性，是优化算法的重点，但同时也是难点。

3.2.1 使用队列结构减少冗余计算

由于 Bellman-Ford 算法由于要遍历所有的边，最坏情况下时间复杂度为 $O(n^2)$，尤其是对于密集图，算法的处理效率会进一步降低。

分析很容易可以发现，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。根据其特性进行如下优化：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（即取 d[v] = min{dist[u]+edgeDis, d[v]}，并更新前驱节点，此时如果此时的队列中不包含 v 节点，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），重复上述操作直到队列为空或者是某一个顶点的入队次数超过|V|-1，若队列为空则说明求解完毕，若入队次数超过|V|-1，则说明图中存在负权回路，没有最短路径。

改进后的算法流程如下：

（1）初始化：设 dist(s)=0，对于其余每一个节点 i，置 dist(i)=∞。将 s 加入队列 Q。令迭代轮次 i=1。

（2）在第 i 轮迭代过程中，从队列中取出一个节点 v，按如下公式进行计算。

$$dist(u) = \min\{dist(u), dist(v) + |u,v|\}, (u,v) \in E$$

如果 dist(v)改变，且队列 Q 中不存在节点 v，则将 v 加入到 Q 中。

（3）若队列 Q 为空，则算法结束，dist 中将记录最短距离信息；若 i=|V|-1 但队列 Q 非空，则可确定图中存在负循环，无法得出最短路径；若 Q 非空，且 i<|V|-1，执行（4）。

（4）i=i+1，执行步骤（2）。

算法伪代码如算法 4 所示。

算法4 优化搜索空间的Bellman-Ford算法

```
输入：G=(V, E) ,s;                                    //图数据和源节点
输出：dist, pre                                       //最短距离和路径向量
1：  for each u ∈ V do
2：        dist[u]←∞；pre[u]=-1；visited[u]=false     //初始化
3：  end for
4：  dist[s]=0
5：  queue.push(s)
6：  for iter=0 → |V|-1 do
7：        while queue is not empty do
8：              u ← queue.pop()
9：              for each u = neighbor(u) do
10：                   if dist[u]+|u,v| < dist[v] then
11：                        dist[v]=dist[u]+|u,v|
12：                        pre[v]=u
13：                   if v is not in queue then
14：                        queue.push(v)
15：                   end if
16：             end for
17：       end while
18：end for
19：return dist, pre
```

3.2.2 Bellman-Ford 的并行性优化

对于 Bellman-Ford 算法，这里主要对其 CUDA 实现过程进行优化，以充分发挥其并行性能。

（1）访存优化

CUDA 编程遵循着一套固定的规范流程，需要在主机分配 GPU 内存，拷贝

GPU 数据，在 CPU 端调用 GPU 核函数，发起多个线程分别对图数据进行处理。由于 CPU 和 GPU 的异构属性，不能在 GPU 核函数中直接访问主机内存，而是需要在执行核函数前，将必要数据拷贝到 GPU 内存中，以供并行核函数读取计算和写回。

另外，GPU 内存可分为全局内存、共享内存以及块本地内存，所有线程可访问全局内存，一个块内所有线程可访问所属块的共享内存，且对于共享内存的访问效率高于全局内存。设计一套适合于 Bellman-Ford 算法的图数据储存和访问模式，对于算法执行效率有着值得期待的积极影响。

（2）线程数量优化

在第 3.2.1 节中，描述了减少冗余计算的方法，在 GPU 核函数中直接使用队列可能会遇到一些访存互斥方面的问题，反而会导致性能下降。这里采用标记数组标记需要松弛的节点，并使用规约方式进行统计，避免冗余计算，减少线程数量。

（3）优化数据传输开销

在迭代开始前，需要将图数据拷贝到 GPU 内存中，这里提出了分流数据拷贝的优化方案，可以一定程度上提高传输效率。

3.3 本章小结

本章详细描述了迪杰斯特拉算法、贝尔曼-佛德算法的优化思路。基于评估值提高 Dijkstra 算法的并行性，使用队列结构降低 Bellman-Ford 算法冗余计算等，本章节表述的算法将在第四章进行实现。

4 基于 GPU 的 SSSP 算法优化实现

按照第三章描述的 Dijkstra 算法以及 Bellman-Ford 算法的优化方案，在本章节将会描述实现和处理细节。

4.1 基于 GPU 的 Dijkstra 算法优化实现

在第三章中详细描述了 Dijkstra 算法的实现思想，分别对传统 Dijkstra 算法描述了基于评估值的并行优化和评估值的规约比较过程，在这里将详细描述优化算法的实现细节。

4.1.1 评估值的计算

评估值决定着下一次迭代过程中能够并行的节点，因此需要在每一轮迭代开始前完成评估值的计算过程。由第三章可得评估值计算公式$D(i) = \min\{ dist(u) + \min\{ |u,v|\}\}$，这里可将评估值的计算分为两个过程：（1）节点本地评估值 dist(u)+min{|u, v|}的计算过程，（2）全局评估值 min{dist(u)+min{|u, v|}}的计算过程。

第一个步骤的计算过程将在松弛过程完成时计算并将中间结果储存到数组中，计算时需要检查该节点的每一条出边，找出指向已打开节点或者未打开节点最小边，若该节点的出边数量为|e|，则检测次数必为|e|。

为了减少检测次数，结合 CSR 压缩存储格式，可以改变每一条边的排列顺序，将边的顺序按照权值增大的顺序进行储存。查找最小边时从前向后查找，选定第一个指向已打开或者未打开节点的边即可。

对于如图 4-1 所示图数据，若采用调整前的图数据，在对节点 m 执行松弛操作后更新局部评估值时，需要依次检查 column 和 values 数组的全部 5 个元素，选取 column 中指向非已关闭节点的最小 values 值，这一过程需要访存 10 次。若采用调整后的图数据，假设此时已关闭节点集中包含 7、40，则只需要依次检查节点 7、40、17 的状态，检查过程不需要访问 values 值，仅需要在查找到符合要求的边后，获得其权值即可，整个过程仅需要访存 4 次即可。

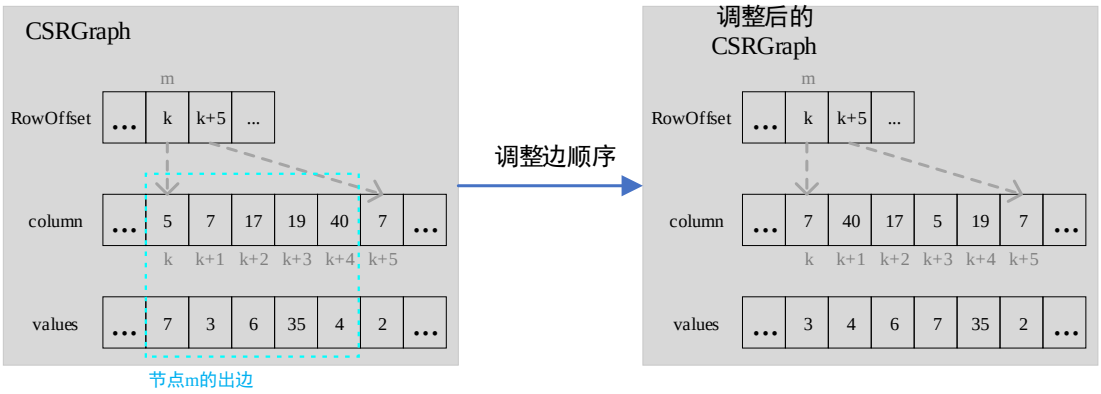


图 4-1 调整顺序前后的 CSR 格式

图数据结构转换的过程可以在执行迭代之前进行处理，转换过程不会改变图的拓扑形状，且可以将中间结果存入文件。该过程可视作预处理过程，不计入算法的额外的时间开销。

4.1.2 并行比较过程设计

完成节点本地评估值的计算之后，需要收集所有 O 集合内的节点的评估值，比较获得最小的评估值作为全局评估值，用于指导下一轮松弛。

为了加速计算流程，这里采用并行的规约比较方式，规约比较原理如图 4-2 所示。

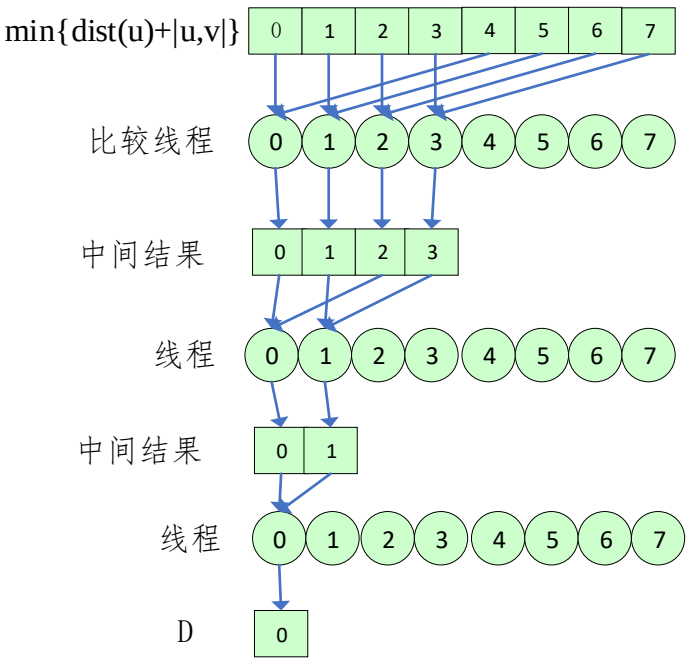


图 4-2 规约比较

并发执行多个比较线程，每个线程选出两两比较的较小结果，并将其作为下一轮比较的输入。逐级迭代，最终可获得全局评估值 D。整个规约比较过程可以将算法时间开销降低到同小顶堆相同的 O(logn)级别，一定程度上提高算法冰箱效率。

4.1.3 整体算法

整体的算法伪代码如算法 5 所示。

算法5 基于GPU的并行Dijkstra算法

```
输入：G=(V, E) ,s;                                    //图数据和源节点
输出：dist, pre                                       //最短距离和路径向量
1： Initial<<<(G.vertexNum-1+warp)/warp, warp>>>(G,dist,pre,s)
2：{将s添加到集合O}
3：while O is not empty do
4：      Iter(G,dist,pre)
5：end while
6：return dist,pre
Procedure Initial(G,dist,pre,s)
7：if tid=s do                                        //tid为线程唯一编号
8：      dist[tid]=0
9：else do
10：      dist[tid]=∞
11：end if
12：pre[tid]=-1
Procedure Iter(G,dist,pre)
13：getD<<<(|O|-1+warp)/warp, warp>>>(G)        // 规约计算评估值
14：{统计集中执行松弛操作的所有节点}
15：Relax<<<(nodeNum-1+warp)/warp,warp>>>(G,dist,pre,nodes)// 执行松弛操作
Procedure RelaxNode(G,dist,pre,nodes)
16：node=nodes[tid]
17：edgesNum=G.rowOffset[node+1]-G.rowOffset[node]
18：RelaxEdge<<<(edgesNum-1+warp)/warp, warp>>>(G,dist,pre,node,)
Procedure RelaxEdge(G,dist,pre,node)
19：edge=G.column[G.rowOffset[node]+tid]
20：if dist[edge.tail]>dist[node]+|edge| do
21：      dist[edge.tail]=dist[node]+|edge|
22：      if edge.tail is not in O do
23：            {将尾节点添加到O中}
24：      end if
25：      {依次访问尾节点的每一条边，更新节点本地评估值}
26：end if
```

4.2 基于 GPU 的算法 Bellman-Ford 算法优化实现

在第三章中详细描述了 Bellman-Ford 算法的实现思想，分别对传统 Dijkstra 算法描述了基于评估值的并行优化和评估值的规约比较过程，将在本节详细描述实现细节。

4.2.1 Bellman-Ford 算法的并行框架

CUDA 架构可以很好地调用显卡以发挥出其良好的并行计算性能，CUDA 编程的流程遵循着一套固定的规范流程，对于优化后的 Bellman-Ford 算法，按照如下流程执行 CUDA 程序。

（1）主机分配内存，读入图数据，这里可以使用 cudaMallocHost 函数分配夜锁定内存，避免操作系统的页表管理机制将数据换出内存，影响数据访问和传输的速率。

（2）分配 GPU 全局内存，从 Host 拷贝图数据到 Device。

（3）CPU 和 GPU 配合进行计算，每轮迭代过程中，CPU 调用 GPU 核函数，发起多个 GPU 线程分别处理不同节点执行松弛操作。

（4）GPU 完成一次迭代，向 CPU 反馈迭代次数，CPU 判断并决定下一步执行流程。

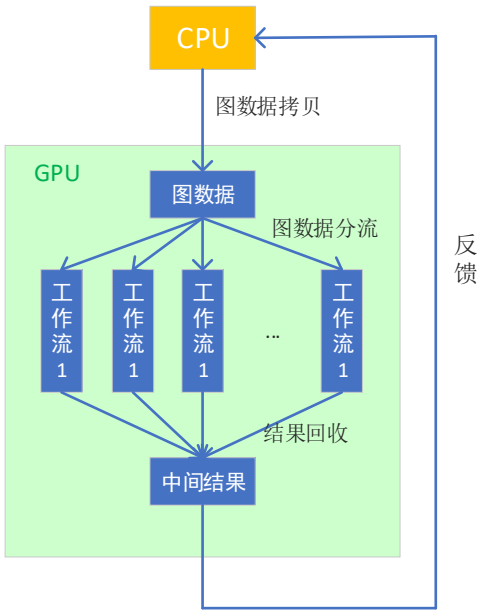
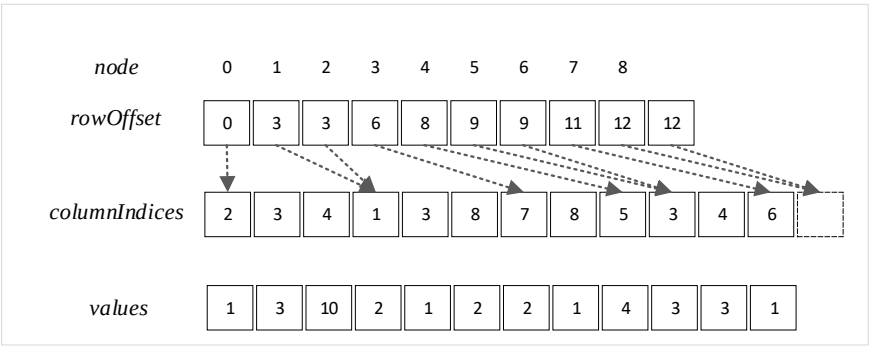


图 4-3 图数据分流处理

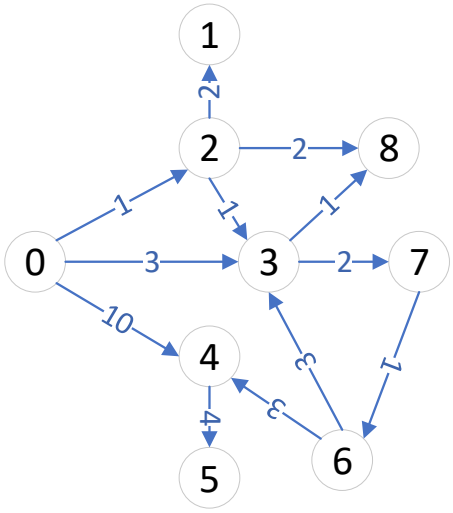
GPU 在每一轮迭代前将图数据分流传输，并行处理，并将处理结果反馈给 GPU 端，迭代过程如图 4-3 所示。

4.2.2 访存优化

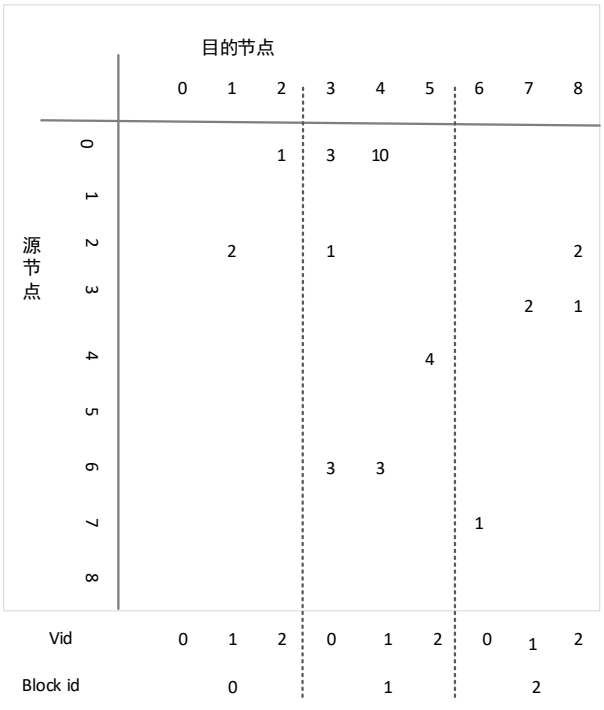
GPU 核函数在迭代过程中会频繁访问全局内存，这会带来一定的访存开销。这里提出一种访存优化策略，减少全局内存的访存次数，提高算法迭代过程的效率。



(A)图数据CSR压缩存储结构



(B)原图



(C)展开为邻接矩阵

图 4-4 示例图数据

各个线程块拥有自己的块内共享内存，且其访问速度高于全局内存，因此，在每轮循环开始前将全局内存中的 dist 数组向复制到共享内存中，每个线程在本

地完成数据处理，计算出第 i 次迭代中每个节点 v 的当前候选距离 dist[i][v]和 pre 节点 pre[i][v]。

以图 4-4 所示图数据为例，将 CSR 行压缩转化为邻接矩阵拷贝到 GPU 显存之中，如图 4-5 所示，假设 GPU 为该图数据创建 3 个线程块，每个线程块包含三个线程，如图所示。在第二次迭代过程中计算节点 v 的最短距离 dist[2][v]，该计算过程依赖于上一轮迭代的数据 dist[1][u]，u∈V，而求取每一个不同节点的 dist[2][v]的过程互不影响。这里假设创建规模为 3 线程块*3 线程的线程组进行计算，dist 矩阵中的 dist[1][v]可以预先读入到 block 的共享内存中，如图所示，block 中线程完成计算后再将计算完成后的结果 dist[2][v]写回到全局内存。而对于 pre 矩阵，无需保存副本，只需要在更新后将结果写回全局内存。

图 4-5 迭代过程中的 dist 矩阵

图的邻接矩阵在全局内存中按行存储，为了提高随机访问速度，这里可以预处理时将矩阵转置后再读入 GPU 的共享内存。处理迭代之前创建多个单线程的 Block 将最短距离矩阵和前驱节点数组读入到每个块的共享内存中，后续的计算过程将在共享内存中进行，这可以降低了访问全局内存的频率，一定程度上提升了访存效率。但因为共享内存大小有限，当图规模较大时需要调整每个 Block 中处理的节点的数量，防止出现内存溢出的现象。

4.2.3 基于标记数组优化线程数量

引入标记数组结构是为了避免个线程遍历全部节点的问题，减少 Bellman-Ford 算法的冗余计算过程。在第 i 轮迭代中，如果对所有节点创建|v|各线程进行松弛操作，会带来一些闲置线程开销，使同一个 warp 中的各个线程出现执行路径

分流的情况，导致线程执行效率下降。而事实上，仅在 i-1 轮中 dist 矩阵发生改变的节点才需要执行松弛操作，因此可以在第 i 轮迭代前，统计并集中 dist 改变的节点，集中化需要计算的节点，从而减少复制到 GPU 共享内存中距离元素的数量，并且可以减少该次迭代中启动的并行线程数量，如图 4-6 所示。

图 4-6 优化线程发散设计

这里对更新标记进行排序，也可以采用类似于图所示的并行规约比较方式。经过固定排序后若更新标记为 1 的节点数量为 k，考虑到线程调度基本单位是 warp，为了充分利用 GPU 处理核心，应该创建恰好能够容纳全部线程数的 warp 单位，这样的方式可能会在最后一个 warp 中空余出一些空线程，这些空线程不会带来较大的冗余操作，但能够一定程度上提高 warp 运行效率。创建线程数目 K 可满足下列公式，这里的除法为整型除法，会丢掉小数部分。

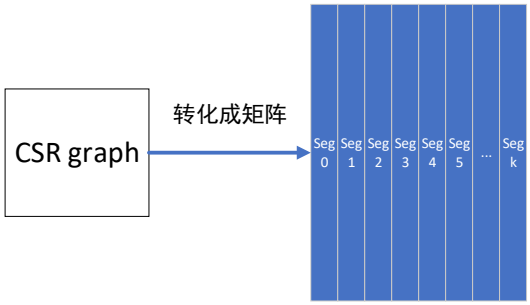
$$K = ((m + warp - 1)/warp) * warp$$

4.2.4 基于 CUDA 分流优化数据传输开销

在迭代计算之前，需要将 CPU 中转化为邻接矩阵的图数据拷贝到 GPU 的全局内存中，邻接矩阵是二维数组，空间复杂度达到 $O(|v|^2)$级别，随着图规模的增大，矩阵尺度也会大幅增大。未完成传输前，依赖于图数据的迭代过程便无法开始，因此，优化传输过程可以很好地降低整个处理过程的时间开销。

基于 CPU-GPU 的并行异步性，可以采用并行分流的方式优化数据的传输过程，在 CPU 处串行启动多个数据传输函数，将图数据划分为多个数据块，如图 4-7(a)所示，分别将不同数据传输到 GPU 全局内存中。在分流的同时，还可以进

一步细化粒密度，降低每个线程拷贝数据的容量，提高并行性，从而提高算法效率，如图 4-7(b)所示。



（a）图数据转换与划分

（b）图数据分流并行传输

图 4-7 分流优化数据传输开销

4.2.5 整体算法

结合上述优化方案，给出 Bellman-Ford 算法在主机和 GPU 之间进行配合迭代的流程图如图 4-8 所示。

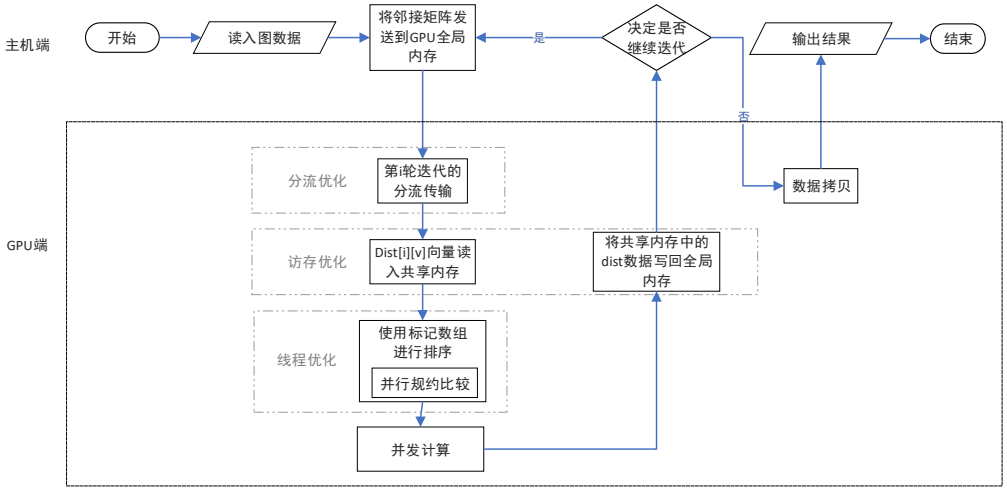


图 4-8 Bellman-Ford 算法流程图

优化后的算法伪代码如算法 6 所示。

算法 6 优化版Bellman-Ford并行算法

```
Procedure Initial(G, dist, pre, s)                    // 并行初始化核函数
1：dist[0][tid]←∞,pre[0][tid]←-1                      //初始化操作，tid为线程标号
2：if tid = s do
3：        d[0][tid]=0
4：end if
Procedure FixedOrder(G,dist,pre)                      // 规约排序模块
5：merge_sort_gpu<<<1,1024>>>(G,dist)                 // 按照更新标志排序
Procedure Iter(G,weight,dist,pre)                     //迭代计算模块
6：{分配共享内存数组dist_last储存第i-1次的迭代结果}
7：{分配共享内存数组pre储存第i-1次迭代后的结果}
8：{分配共享内存数组dist储存第i次迭代后的结果}
9：{分配共享内存数组pre_last储存第i-1次迭代前的前驱节点}
10：idx ← threadIdx.x                                 // 线程id
11：copyData()                                        //将数据从全局内存复制到共享内存
12：{并行执行松弛操作并更新标记数组}
Procedure StreamCall                                  // cuda流调用
13：streamNum←分流数; warp←每个warp含有的线程数（硬件限制）
14：cudaStream_t stream[streamNum];
15：for i=0 to streamNum do
16：        Iter<<<(G.vertexNum-1+warp)/warp,warp,stream[i]>>> (G,weight,dist,pre)
17：end for
18：{流同步}
Procedure Main(G,weight,dist,pre,s)                   // 算法入口函数
19：Initial<<<(G.vertexNum-1+warp)/warp,warp>>>(G,dist,pre,s)
20：for i = 1 to G.vertex-1 do
21：        if i=1 do
22：                StreamCall()                      //分流迭代计算
23：        else do
24：                Iter<<<(G.vertex-1+warp)/warp,warp>>>(G,weight,d,p)
25：         end if
26：        {检查迭代条件}
27：  end for
27：  {获取并输出结果}
```

4.3 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。

4.4 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。

4.5 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。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

本章将对实现的优化算法选择算例进行测试，在说明测试用例、测试平台和测试方案之后，会按照测试方案给出测试结果并且对结果进行分析。

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

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，使其按照边的头节点递增的顺序排列重新输出到文件种，读取文件创建 CSR 数据结构将会很困难；为了方便后续处理，读入文件并创建图结构后，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 基准算法

对于 Dijkstra 算法，这里采用的是基于已打开节点池优化的 Dijkstra 算法作为基准程序，该程序中缩减了当前最短路径的搜索域，较传统表述的 Dijkstra 有一定效率上的提升。

对于 Bellman-Ford 算法，这里采用传统表述的朴素 Bellman-Ford 实现作为基准程序。

5.4 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 2080

Driver Version: 510.47.03 CUDA Version: 11.6

5.5 性能测试

使用中等算例对 Dijkstra 算法、并行优化版本的 Dijkstra 算法、Bellman-Ford 算法、并行优化版本的 Berman-Ford 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。

5.5.1 Dijkstra 算法优化前后性能对比

Dijkstra 算法执行情况如图 5-1 所示。

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。整体平均加速比约为 1.75，可见 GPU 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优化效果，因此执行效率较原算法差。

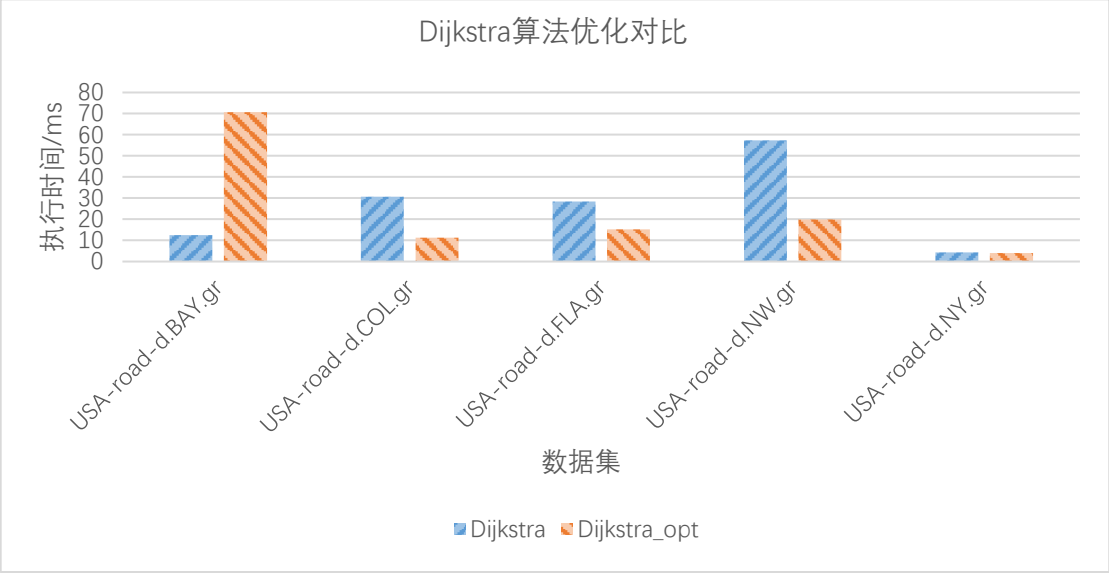


图 5-1 迪杰斯特拉算法优化前后对比

5.5.2 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。

5.6 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，提出了在串行环境以及并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）对传统算法提出了一些优化策略：对 Dijkstra 算法提出了状态标记策略，基于评估值的概念提高了算法的并行性效能，利用并行规约比较方式将比较算法的时间效率提高到了 O(logn)级别，算法的并行效率较传统算法达到了接近 50%的效率提升；对 Bellman-Ford 算法提出了标记数组的优化策略，并针对其在 CUDA 上的实现做出了访存优化、线程数量优化、流优化等策略，充分利用了算法并行性，减少了冗余计算，降低了时间开销。

（3）基于自己对 CUDA 的理解，实现并测试了优化算法的性能，并行下 Dijkstra 算法最终达到了接近 50%的效率提升，而并行下的 Bellman-Ford 算法加速比可以轻松达到 300 以上。

基于上述研究，得到了如下结论：

（1） Dijkstra 算法的并行性有限，通过提高算法的并行性可以一定程度上提高算法效率，但提升效果很大程度上取决于算例特点。

（2） Bellman-Ford 算法的并行性很强，通过调整实现方式，充分发挥其并行特性，并且减少冗余计算，就可以很大程度上降低时间消耗，获得不错的提升效果。

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，CPU-GPU 协同工作具有很大意义，但其带来的庞大通信开销也亟需解决，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，转眼便是临近毕业时节，春花秋月，何曾想到离别的光景。

四年前，背负着亲友长辈的殷切期望，独自一人踏上了远离故乡的城市，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，第一次登上喻家山，第一次环游东湖，还有第一次社团活动……四年，我经历了许多，也成长了许多。

大学四年，是一段漫长的时光，也是一段充满温情的岁月，老师的教导、同学朋友的帮助、室友的关心和包容……都些是支撑着我走过这四年不可或缺的东西。在这里，我希望向所有帮助过我的老师同学还有我的三位室友表达最诚挚的感谢。

完成毕设和编写论文的几个月，或许是大学四年最难熬的年华，在这期间，我走了许多弯路，做了许多无用功，论文初稿也的确难以让人满意。在这里非常感谢张宇导师的指导，导师以其渊博的知识经验、严谨的治学态度、活跃的学术思想和默默的奉献精神时刻激励着我前进。此外，还要感谢姜新宇学长，在我遇到困难的时候给了我许多帮助和可行的建议。

值此论文完成之际，衷心向所有关心、帮助、支持和鼓励我的老师、同学朋友致以最真诚的协议，他们的帮助推动着我在人生路上前行，也让我在学习生活、人生领悟中受益匪浅。

最后，我要感谢我的父亲，大学四年，无时无刻不在关心着我的学习和生活，在我进步的时候提醒我不要骄傲，在我失败时安慰我不要灰心，在我愤懑时倾听我的吐槽和抱怨，在我踌躇时鼓励着我脚踏实地追逐梦想……无时无刻，他都是我最坚强的后盾，激励着我不断前行。

参考文献

[1] ZHAO X, SALA A, WILSON C, et al. Orion: Shortest Path Estimation fo r Large Social Graphs[C]. in: Proceedings of 3rd Workshop on Online Soci al Networks (WOSN 2010), Boston, United States, Massachusetts, 2010. U SENIX Association, 2010: 1-9.

[2] ZHAO X, SALA A, ZHENG H T, et al. Efficient shortest paths on massiv e social graphs[C]. in: Proceedings of the 7th International Conference On Collaborative Computing: Networking, Applications and Worksharing (COL LABORATECOM), Orlando, Florida, 2011. Inst Computer Sciences Social I nform & Telecommun Eng-Icst, 2011: 77-86.

[3] ALMASI G S, GOTTLIEB A. Highly Parallel Computing[M]. Redwood Ci ty, United States: Benjamin-Cummings Publishing Co, 1989. 383-383.

[4] SHI L, LIU W, ZHANG H, et al. A survey of GPU-based medical image computing techniques[J]. Quant Imaging Med Surg, 2012, 2(3): 188-206.

[5] MITTAL S, VETTER J S. A Survey of CPU-GPU Heterogeneous Computi ng Techniques[J]. ACM Computing Surveys, 2015, 47(4): 47-69.

[6] TARDITI D, PURI S, OGLESBY J. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses[C]. in: Proceedings of the 12th I nternational Conference on Architectural Support for Programming Languag es and Operating Systems (ASPLOS), San Jose, California, United States, 2006. Assoc Computing Machinery, 2006: 325-335.

[7] CHE S, BOYER M, SHEAFFER JW, et al. A performance study of gener al-purpose applications on graphics processors using CUDA[C]. in: Proceedi ngs of ACM Conference on Computing Frontiers, Ischia, Italy, 2008. Acad emic Press Inc Elsevier Science, 2008: 1370-1380.

[8] DU P, WEBER R, LUSZCZEK P, et al. From CUDA to OpenCL: Toward s a performance-portable solution for multi-platform GPU programming[J]. Parallel Computing, 2012, 38(8): 391-407.

[9] PRATX G, LEI X. GPU computing in medical physics: A review[J]. Medi cal Physics, 2011, 38(5): 2685-2697.

[10] BOYD C. Data-parallel computing[J]. Queue, 2008, 6(2):30-39.

[11] VALIANT L G. A Bridging Model for Parallel Computation[J]. Communications of the ACM, 1990, 33(8):103-111.

[12] BEN-NUN T, LEVY E, BARAK A, et al. Memory access patterns: The missing piece of the multi-GPU puzzle[C]. in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Austin, Texas, 2015. New York: Assoc Computing Machinery 2015: 19-19.

[13] PAN Y, WANG Y, WU Y, et al. Multi-GPU Graph Analytics[C]. in: Proceedings of 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, Florida, 2017. New York: IEEE, 2017: 479-490.

[14] SANDERS P, SCHULTES D, Vetter C. Mobile route planning[C]. in: Proceedings of 16th Annual European Symposium on Algorithms (ESA), Karlsruhe, Germany, 2008. Berlin, Germany: Springer-Verlag Berlin, 2008: 732-743.

[15] BARCELO J, CODINA E, CASAS J, et al. Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems[J]. Journal of Intelligent & Robotic Systems, 2005, 41(2-3):173-203.

[16] PAPADIAS D, ZHANG J, MAMOULIS N, et al. Query Processing in Spatial Network Databases[C]. in: Proceedings of International Conference on Very Large Data Bases (VLDB), San Francisco, United States, 2003. Morgan Kaufmann, 2003: 802-813.

[17] RETVARI G, Biro JJ, CINKLER T. On shortest path representation[J]. IEEE-ACM TRANSACTIONS ON NETWORKING, 2007, 15(6): 1293-1306.

[18] BARRETT C, JACOB R, MARATHE M. Formal-language-constrained path problems[J]. Siam Journal on Computing, 2000, 30(3): 809-837.

[19] 郭绍忠, 王伟, 周刚等. 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.

[20] JIN R, RUAN N, YOU B, et al. Hub-Accelerator: Fast and Exact Shortest Path Computation in Large Social Networks[J]. arXiv, 2013:1-12.

[21] ANDREY G, SRIKANTA B, STEPHAN S, et al. Fast and accurate estimation of shortest paths in large graphs. in: Proceedings of the 19th ACM i

nternational conference on Information and knowledge management (CIKM), New York, USA, 2010. Association for Computing Machinery, 2010: 499–508.

[22] 吴漫, 白明丽, 曾咏欣等. 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.

[23] 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.

[24] 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.

[25] 陈智康, 刘佳, 王丹丹等. 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.

[26] HARISH P, NARAYANAN P J. Accelerating large graph algorithms on the GPU using CUDA[C]. in: Proceedings of 14th International Conference on High Performance Computing (HiPC 2007), Goa, India, 2007. Berlin, Germany: Springer-Verlag Berlin, 2007: 197-208.

[27] GARLAND M, LE GRAND S, NICKOLLS J, et al. PARALLEL COMPUTING EXPERIENCES WITH CUDA[J]. IEEE Micro, 2008, 28(4): 13-27.