摘　要

业务量工程（Traffic Engineering）能够通过为业务选择合理的网络路由来达到充分利用网络资源，提高网络性能，满足 Qos 需求等目的。在 SDN 网络中，控制器能够获取全网信息，能够在全局拓扑上进行业务量工程，提高业务量工程效果。然而，由于互联网应用的快速增加，短时间内会有大量业务到达 SDN 网络，同时，SDN 网络的规模也相应增大，这要求业务量工程需要在短时间内在大拓扑上为大量业务计算路由，业务量工程的计算面临着时间上的挑战，为了加速业务量工程算法，我们使用基于 GPU 的并行算法来解决业务量工程问题，利用 GPU 的大规模计算能力来加速算法。本文主要研究 SDN 网络下适合于并行计算的业务量工程算法框架，并且针对这些框架，设计并行的路由算法。

在 IP 网络中，本文首先提出了一个带链路容量约束的业务量工程模型，模型中的效用函数为最小化路由代价，为了求解这个模型，本文提出了两种并行算法 GA-PTEA（Genetic algorithm based parallel traffic engineering algorithm）和 LR-PTEA（Lagrange relaxing based parallel traffic engineering algorithm)。GA-PTEA 采用了基于备选路径的模型，利用并行的遗传算法来求解业务量工程问题，加速比可达到 10 倍以上。LR-PTEA 采用了基于拉格朗日松弛的模型，首先通过松弛链路容量约束将业务量工程问题分解为一批业务的路由计算问题，然后设计了基于 GPU 的并行算法来加速路由计算，本文采用次梯度下降方法来求解拉格朗日对偶问题，为了加速次梯度算法的收敛速度，LR-PTEA 采用了高效的次梯度步长更新方法，同时，LR-PTEA 在求解对偶问题的过程中采用了高效的路径调整策略来获得对原问题目标函数的优化解，本文的实验发现 LR-PTEA 可以在短时间了得到业务量工程问题的优化解，实验发现基于 GPU 的 LR-PTEA 并行算法对串行算法 LR-STEA(Lagrange relaxing based parallel route optimization algorithm) 的加速比可达到 6-7 倍。

针对 SDN 弹性光网络，本文采用分层图模型来进行业务量工程优化，为了优化弹性光网络中的资源使用，降低阻塞率，本文提出了 TESAA（traffic engineering and sptrum allocate algrithm）算法框架，并对框架进行并行设计，分别针对带权图和无权图设计了基于 GPU 的并行路由算法。在无权图情况下，本文采用并行 BFS 算法来加速无权图的 TESAA 框架。在带权图情况下，本文首先提出了带权带跳数约束的最短路动态规划模型，并对动态规划算法进行并行设计来加速带权图下的 TESAA 算法框架。实验发现 TESAA 和一般的贪心路由算法相比较，可以大大降

低路径的代价，路径跳数，节省网络资源，同时 TESAA 还能有效降低业务的阻塞率，实验发现基于 GPU 的并行的 PTESAA（parallel TESAA）算法对串行算法 STESAA(serial TESAA) 的加速比可达到 5 倍。

关键词： GPGPU，并行算法，业务量工程，路由优化，路由算法，SDN，弹性光网络，RSA，动态规划

ABSTRACT

Traffic engineering refers to the process of optimizing the network utilization, minimizing network costs, improving network performance, and satisfying QoS requirements by selecting a reasonable network route for the service. However, due to the rapid increase of Internet applications In a short period of time, there will be a large number of services reaching the SDN network. At the same time, the size of the SDN network will also increase accordingly.This makes service measurement needs to calculate large-scale routing in a short time. To solve the problem of large-scale routing calculation in traffic engineering, we use GPU-based parallel algorithms to calculate the traffic engineering problem and use the large-scale computing capacity of the GPU. To speed up the algorithmic process. This paper mainly studies the framework of business engineering algorithms suitable for parallel computing, and designs parallel routing algorithms for these frameworks.

In a general SDN network, two parallel algorithms GA-PROA (Genetic algorithm based parallel route optimization algorithm) and LR-PROA (Lagrange relaxing based parallel route optimization algorithm） are proposed in this paper. Among them, GA-PROA is similar with the thesis [17], but this article uses a better objective function than the paper [17]. A new parallel optimization algorithm detail is designed for this objective function. GA-PROA using the alternative path-based model, the parallel genetic algorithm is used to solve the traffic engineering problem. The speedup ratio can be up to 20 times. LR-PROA uses a Lagrangian relaxation model,LR-PROA first decomposes the routing optimization problem into a set of path calculation problems for the traffic demands by relaxing the link capacity constraints, then the path calculation tasks are dispatched to GPU and executed concurrently on GPU,this paper designs a GPU-based multi-service parallel Bellman-Ford algorithm to accelerate the routing calculation,this paper uses the sub-gradient method to solve the Lagrangian dual problem, in order to accelerate the convergence of the sub-gradient algorithm, LR-PROA uses an efficient sub-gradient update method. At the same time, LR-PROA uses an efficient path adjustment strategy to solve the dual problem to obtain a optimal solution of the original problem objective function.The experiment shows that LR-PROA can get the optimization solution of the traffic engineering problem in a short time,and that the GPU-based LR-PROA parallel algorithm can accelerate the serial algorithm LR-SROA (Lagrange relaxing based serial route optimization algorithm) up to

6-7 times.

For SDN elastic optical network, this paper uses layered graph model to optimize the traffic engineering. In order to optimize the use of resources in elastic optical network and reduce the blocking rate, this paper proposes a route optimization routing and spectrum allocation (RO-RSA) algorithm framework. In parallel design of the framework, a parallel GPU-based routing algorithm is designed for weighted graphs and unweighted graphs. In the case of weighted graphs, the parallel BFS algorithm is used in this paper to accelerate the RO-RSA framework without weighted graphs; in the case of weighted graphs Next, this paper first proposes the shortest path dynamic programming model with weighted hops constraint, and parallel design of dynamic programming algorithm to accelerate the RO-RSA algorithm framework under the weighted graph. In experiments,RO-RSA is compared with the general greedy routing algorithm. It can greatly reduce the cost of the path, the number of path hops, and save network resources. At the same time, RO-RSA can effectively reduce the blocking rate of the services. Experiments have found that a GPU-based parallel PRO-RSA (parallel RO-RSA) algorithm can speed up the serial algorithm S-ROSA (serial RO-RSA) for 6 times.

Keywords: GPGPU, Parallel Algorithm, Traffic Engineering, Route Optimization, Routing Algorithm, SDN, EON, RSA, Dynamic Programming

目 录

图目录

表目录

第一章 绪 论

1.1 研究背景与意义

软件定义网络 (Software Defined Network，SDN) 将网络的控制平面和数据平面分离开来，以集中的控制器视角为网络用户提供各种应用服务 [1] [2]。SDN 网络中的控制器负责对网络的控制和调度，同时控制器通过向上提供编程接口，使得网络控制规则可以根据不同的需求进行设计，使得网络的控制设计更加灵活，网络管理人员可以更方便的更新各种网络应用和服务，而无需关心底层的数据平面管理。在 SDN 底层数据平面中，各种网络设备被抽象简化，到达网络设备的数据流通过匹配控制器下发在设备中的流表规则来进行快速转发，和传统网络相比，SDN 架构大大简化了网络设备的复杂功能，减小了网络成本。SDN 架构中的控制器可以集中控制网络，控制器可以获得全网络的拓扑信息，同时，控制器能够通过下发流表规则来对网络流进行细粒度的调度，所以在 SDN 架构下可以很容易地实现基于全局网络的负载均衡，路由优化，Qos 等功能。

业务量工程（Traffic Engineering，TE，又被称为路由优化 routing optimization，RO）是指通过为业务选择合理的网络路由来达到充分利用网络资源，最小化网络代价，提高网络性能，满足 Qos 需求等目标的优化过程。在 SDN 网络控制器中能够实现基于全网信息的路由优化，微软 [7] 和谷歌 [8] 的实验结果证明，在 SDN 网络结构的数据中心网络中，业务量工程能够在网络吞吐量和链路利用率上达到接近最优化性能的表现。

但是另一方面，SDN 网络下的业务量工程为 SDN 控制器带来了较大的计算压力。这是因为，第一，随着网络应用的快速增加，在 SDN 网络中短时间内可能会有大量业务到达控制器 [9]，所以控制平面必须短时间内为大量业务计算路由。第二，为了适应大量业务的加入，网络规模也快速增大 [10] [11]。第三，SDN 技术使网络能够几乎满负荷运行 [12]，这意味着一旦网络流量发生重大变化或者网络链路出现故障，SDN 控制器需要在短时间内重新计算出路由。因此，在大网络下对大量业务进行快速和高效的业务量工程成为了一个重要却困难的问题。

为了缩短 SDN 网络下业务量工程的计算时间，设计高效的并行算法是一种常见的解决思路。另外，由于业务量工程中大量业务之间的路由计算是相互独立的，这也为并行业务量工程算法的设计提供了可能性。同时，现今出现的商业服务器具有多核 CPU 和强大的 GPU，为并行算法提供了一种低成本、高性能的计算环境，尤其是 GPU，他具有大量的计算单元，现今的 GPU 能够同时调度和执行上千

个线程，具有强大的并行计算能力。

GPGPU(general purpose programming on GPU) 是指利用 GPU 进行通用计算（而不仅仅是图形学计算）的算法设计思想，为了简化 GPU 的通用程序设计模式，2007 年，Nvidia 发布了一种新的的 GPU 编程模型 CUDA(Compute Unified Device Architecture)[13]。与传统的 GPU 通用计算开发模式相比，CUDA 编程更简单，功能更强大，应用领域更广泛，支持 CUDA 的硬件性能更强。随着 GPU 硬件计算能力的不断提高和 CUDA 通用计算模型的流行，利用 GPU 来加速并行算法成为一个研究热点[15]。结合 SDN 网络中业务量工程的可并行性质和 GPU 强大的通用并行计算能力，来设计高效的基于 GPU 的并行业务量工程算法能够大大提高业务量工程的计算时间，具有很重要的研究意义。

1.2 国内外研究现状

过去十年间，随着互联网中的网络流量的快速增加，业务量工程得到了大量的研究，论文 [18–20] 对业务量工程的研究进行了详尽的综述。大部分情况下业务量工程问题是一个 NP 难问题 [21,22]，大量启发式算法[18–20] 被提出来解决这类业务量工程问题。然而，这些算法基本上都是串行算法，在大网络和大业务量的情况下，在 CPU 上实现这些算法需要几分钟甚至是几个小时[16,22,23]。由于执行时间较长，这些路由优化算法仅作为离线工具使用[24]。为了减少计算时间，多篇论文[25,26] 研究了分布式的路由优化算法，在分布式的路由优化模型中，假设每个业务有一系列的备选路径，业务将总流量分配到各个路径上，各个路径上所承载的流量根据路径上的链路容量情况实时进行调整。然而分布式的路由优化问题依赖于精确实时的网络状态信息，而且在实际实现中收敛较慢。

另一方面，过去十年，GPU 的计算能力得到大幅度提高，GPU 通用计算模型得到大力发展 [13,27]，GPU 的理论计算能力提高速度大大高于 CPU 的计算能力提高速度 [27]。因此，许多基于 GPU 的并行算法被提出来求解一些整数规划问题，比如说旅行商问题[28]，路线规划问题[29]，最短路问题[30,31]，最小生成树问题 [32]，现存的研究[28–32] 表明基于 GPU 的并行算法可以比基于 CPU 的算法快几十倍以上。

现今，对并行业务量工程算法的研究较少，[16,17] 对业务量工程问题提出了并行的算法，在论文[16] 中，作者为 alpha 公平的路由优化问题提出了一种很适合在 FPGA 和 GPU 上实现并行的算法，为了达到 alpha 公平的目标，论文 [16] 假设一个业务需求需要被分到多条路径上，但是这个假设在实际中很难实现，这是因为，第一，如果业务流是细粒度的流（TCP/UDP），把业务分到不同的路径将导致网络数据包的乱序传递。第二，SDN 网络需要使用更多的流表规则来将聚合的流分离

到不同的路径，但是 SDN 交换机上的流表资源 TCAM(ternary content addressable memory) 是有限的，分拆流将可能因为 TCAM 的不足而变得不可行。第三，仅仅通过添加一系列流表规则来细粒度地分割流量是很难实现的。在论文 [17] 中提出一种基于 GPU 加速的遗传算法来加速路由优化问题，其目标函数是最小化最大链路利用率（MLU），但是大量基于实际拓扑的实验表明链路利用率效用函数，特别是链路利用率，在网络利用率没有达到拥塞程度的时候，不是对网络优化的较好评价函数 [37]，反之，在网络链路发生大量拥塞的情况下 MLU 只去优化最大的链路利用率，而不能给出一个可行（满足容量约束）的解。

随着视频流、云计算服务和移动应用的普及，互联网的流量不断增加 [42]，业务量的持续增长给互联网的传输带来了巨大的挑战，为了满足日益增长的容量要求，波分复用（WDM）系统已部署在骨干网络，其每个通道带宽高达 40 GB/s 或 100 Gb/s。然而，传统的 WDM 光网络严格遵循 ITU-T 的固定均匀间距（通常为 50GHz 或 100 千兆赫) [45]，这样会导致低效的频谱利用率，比如，一个较大的波长可能会被分配给一个低速率的业务，即使这个业务根本就不能占满整个波长。很明显，传统 WDM 光网络的不灵活和粗粒度的带宽控制会导致显著的频谱浪费，限制了提高其网络容量的潜力。

为了应对 WDM 网络的低敏捷性和频谱浪费问题，近年来，弹性光网络 (EON) 的架构被提出，在 EON 架构中，存在细粒度的带宽间隙（比如，12.5GHZ），他比 WDM 网络所遵循的 ITU 标准的 50GHZ 或者 100GHZ 的带宽粒度要小很多，而且，这些带宽间隙可以根据需要被组合在一起以提供更宽的通道，以灵活地适应各种速率的业务，提高频谱利用率。为了在 EON 网络中加入业务，控制平面必须在网络中找到一条路径，同时，还需要在此路径上的链路上分配足够的频谱带宽，来创建一个合适的端到端光路连接，这被称为路由和频谱分配（RSA）问题。

弹性光网络下的业务量工程问题需要同时考虑 RSA 问题，目前对于弹性光网络下的业务量工程研究较少，论文 [48] 中提出一种分层图模型来解决 WDM 网络中的路由和波长分配问题，这种分层图模型也可以很容易推广到 EON 网络中，分层图模型将 RSA 问题统一成路由问题。因此，使用分层图模型可以将弹性光网络下的业务量工程问题转化为一般的业务量工程问题。

1.3 论文内容及结构安排

本文各个章节的内容如下：

第一章简要介绍了 SDN 网络下的基于 GPU 的并行业务量工程算法的研究现状和意义。

第二章简略介绍了 GPU 的相关基础原理，包括 GPU 架构和 CUDA 编程模型，以帮助理解第三，第四章的 GPU 程序设计思路。

第三章首先提出了一个新的业务量工程优化目标，改进了传统的基于最小化最大链路利用率的优化目标，其次，采用备选路径模型设计遗传算法，并设计了这个遗传算法的 GPU 并行版本对目标函数进行优化，然后，采用基于拉格朗日松弛的模型，把路由优化问题转化为一系列的多业务路由问题，对多业务的路由问题设计基于 GPU 的并行算法进行加速。最终将遗传算法结果和基于拉格朗日松弛的结果和 Cplex 结果进行比较。

第四章首先使用分层图模型将频谱分配问题和业务量工程问题统一起来，并且针对分层图模型设计业务量工程算法框架。分别在无权图和有权图两种情况下设计了基于 GPU 的并行路由算法来加速这个框架。

第二章 GPU 硬件结构与 CUDA 编程模式

本文中的并行算法均在 Nvidia GPU 上通过 CUDA 编程模式实现，为了帮助理解本文的并行算法设计思路，我们在本章介绍 GPU 的硬件架构和 CUDA 编程模型。

2.1 CPU 与 GPU

2.1.1 CPU 与 GPU 区别

图为 2-1 为 CPU 和 GPU 的架构比较，可以看到，CPU 和 GPU 的差异主要有两点，第一，GPU 的 ALU 单元远远多于 CPU。第二，GPU 每个核心可用的逻辑控制单元和缓存相对 CPU 要少很多。CPU 上大面积晶体管是逻辑控制单元和缓存单元，这是因为 CPU 的设计需要兼容多方面的应用，比如在桌面应用中就具有大量的分支控制操作和存储操作，而真正的数值计算操作却很少，所以 CPU 上具有大量的逻辑控制相关的实现，比如分支预测（Branch Prediction），乱序执行（OoO 来满足这些分支控制需求，同时增加缓存大小来加速存储过程，但是这样的设计也使得留给 CPU 上的计算单元的晶体管面积较少，浮点计算能力较差，现在 CPU 为了弥补其计算能力得不足，产商常常在同块芯片上集成多个 CPU 核心，组成多核处理器，但是这样的设计并不能提高晶体管的利用率。

GPU 最初被设计来进行图像渲染，图像渲染具有高度的并行性，而且大部分操作是浮点计算操作，逻辑控制较少，所以 GPU 在设计上采用简化控制单元，增加计算单元的设计思想，这使得 GPU 在进行大规模浮点运算的任务时大大优于 CPU 的执行速度。

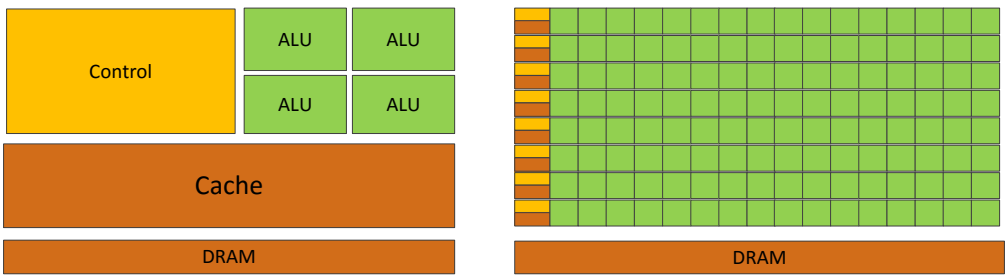

图 2-1 GPU 与 CPU 的区别

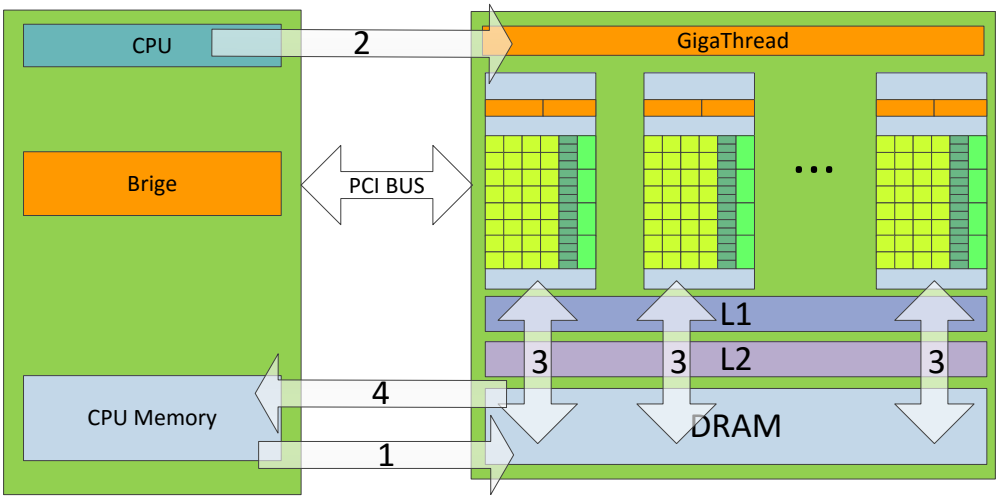

图 2-2 GPU+CPU 异构计算模型

2.1.2 CPU+GPU 异构计算模型

通过上面的分析可以看到 GPU 适用于线程数目多，浮点计算密集和逻辑较简单的并行任务，而 CPU 更加适应与串行的分支较多，计算较少的任务。所以在实际中，常常把 CPU 和 GPU 结合起来，使用 CPU 和 GPU 两个部分协作共同完成并行计算任务，在这些并行任务中，CPU 主要用于逻辑控制部分，而 GPU 作为一种设备被 CPU 控制调用来做并行计算工作，这种协同工作模型被称为 CPU+GPU 异构计算模型。如图 2-2所示为 CPU+GPU 异构计算结构示意图，GPU 和 CPU 通过 PCI 总线进行连接。CPU 调用 GPU 执行一共需要以下步骤：

1. 把输入数据从 CPU 主机内存拷贝到 GPU 的内存（全局内存）中。

2. 把执行程序加载到 GPU 上然后执行。

3.GPU 上的程序在执行过程中需要从主存中读取数据，为了加快线程访问内存的速度，数据将通过多级缓存进行缓存。

4.GPU 上程序执行完毕，将结果写在 GPU 主存中，这时需要将结果重新拷贝回 CPU 主机端进行处理。

2.2 GPU 硬件架构

虽然 GPU 相对于 CPU 在并行算法方面有很多优势，但是，由于 GPU 的出现是为了加速图形渲染问题，当想要使用 GPU 进行图形学之外的通用计算时，我们需要将问题转换为图形学问题，并且通过 OpenGL 或者 DirectX 等 API 来访问 GPU，这对普通的开发人员提出了更高的要求，限制了 GPU 程序的设计自由度，使

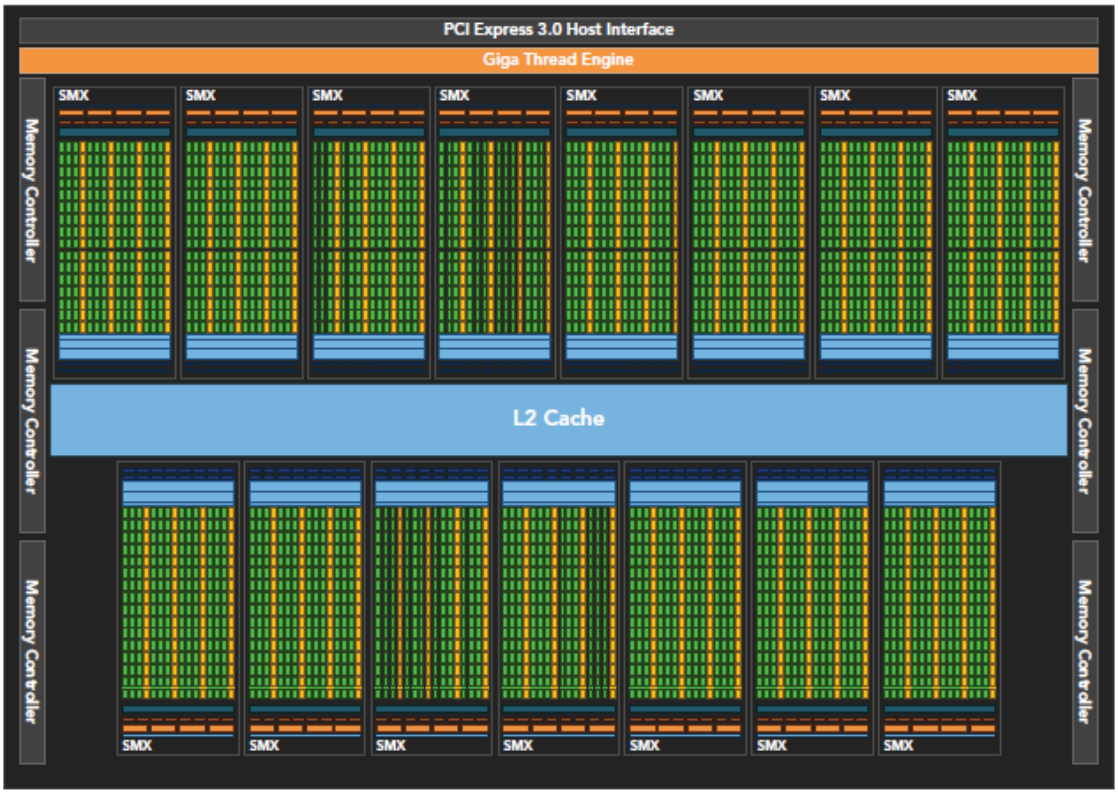

图 2-3 Kepler 架构

得 GPU 通用计算变得困难。2007 年 6 月，NVIDIA 推出 CUDA （Compute Unified Device Architecture，统一计算设备架构）[13]。CUDA 简化了在 GPU 上的通用计算流程，使用 CUDA 进行并行算法设计，不需要把问题转化为图形学问题去调用图形学 API，同时，CUDA 采用类似于 C 语言的语言结构进行开发，这使得开发人员可以很快地熟悉和使用 CUDA 进行开发。但是，要设计出快速高效的 CUDA 程序，开发人员需要掌握 GPU 架构上的相关知识，利用 GPU 的特殊架构来优化算法的设计。

我们以 NVIDIA Kepler 架构为例子来介绍 GPU 的硬件架构。图 2-3 为 Kepler 的 GPU 架构模型细节，Kepler 架构包含两个部分，流处理器阵列（core 部分）和存储系统（memory）部分，Kepler 架构中的 GPU 上一共包含 15 个流多处理器（SM），所有流多处理器共享一个 L2 缓存。

2.2.1 流处理器

我们看到 GPU 中主要组成部分是 SM，一个 SM（stream mutiprocessor）中含有大量的 sp（stream processor），sp 为 GPU 的计算核心，又称为流处理器，sp 是最基本的处办理单元，指令和任务最终都是 sp 上处理的，我们可以将在一个 sp 上执

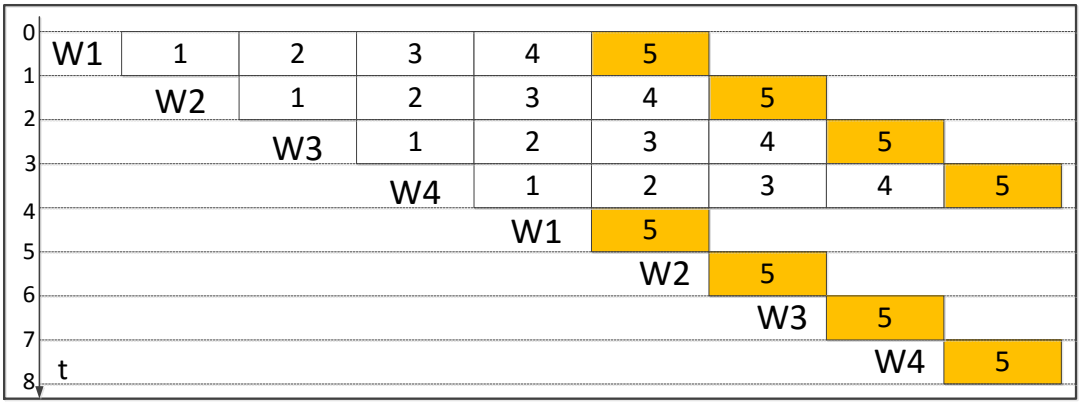

图 2-4 warp 切换示意图

行的流看做一个独立的线程（thread），如果一个 GPU 中有更多的 SM，一个 SM 中有更多的 sp，那么就意味着这个 GPU 在相同的时间内可以并行处理更多的任务。

2.2.2 线程束 (Wrap)

warp 是 SM 调度和执行的单位，一个 SM 中所有线程会被分成一个个 warp 组，通常一个 warp 组包含 32 个线程。同一个 warp 中的线程始终是同步的，这些线程都执行相同的指令，也就说同一个 warp 中的线程必须等待所有线程都执行完了同一个指令后，才会去执行下一个指令，这样的设计虽然节省了硬件的控制单元的复杂程度，但是也限制了 GPU 在处理分支时的并行粒度，如果同一个 warp 中的线程出现分支，每个线程执行的分支条件不一样，为了保持同步，不进入分支的空闲 sp 也必须等待进入分支的线程执行完毕后才能继续执行，这样使得 sp 的利用率下降。

我们知道 GPU 的浮点计算能力很强，但是在一些应用中常常需要进行内存的读写操作，这些访存操作需要大量的时间延迟，从而限制了 GPU 的浮点计算吞吐量，为了充分利用 GPU 上大规模的 sp，GPU 采用 warp 切换来隐藏这些延迟。同一个 SM 上可以存在多个 warp 的程序上下文，但是同一时间只有一部分 warp 被执行，下图 2-4 展示了 warp 上下文的切换过程，假设每个 warp 执行相同的代码，代码中需要先进行 Load 操作，再进行计算操作，Load 操作需要等待 4 个时钟周期，而计算操作只需要使用一个时钟周期，W1-W4 表示 4 个不同的 warp 线程组，开始时 W1 先执行 Load 操作，而 load 操作阻塞，这时 warp 调度器将 W2 调度到 sp 上进行执行，继续阻塞，直到第 5 个周期，W1 的数据已经准备好了可以进行计算操作，当 W1 的计算操作结束后，继续切换计算 W2，W3，W4。最终程序花费 8 个周期完成了对 4 个任务的计算，隐藏了 12 个周期的延迟，提高了 sp 的利用率。

2.2.3 存储结构

GPU 中的每个 SM 中含有多种存储单元，包括寄存器，共享内存，常量内存，纹理内存。寄存器用来存储程序执行中的临时变量，同一个 SM 中的线程都能够访问共享内存，共享内存的访存速度大大高于主存的访存速度，所以在 GPU 并行代码设计时，我们常常先把频繁访问的数据提取到共享内存中，以减小延迟。另外，将中间结果存储在共享内存中，以减小对主存的访问，但是共享内存的大小是有限的，在 Kepler 架构中共享内存的大小为 64KB。常量内存为只读内存，顾名思义他是用来缓存计算时需要的一些常量而专门设计的，常量内存大小为 48KB。纹理内存也是一种只读缓存，纹理内存是为图形学任务而设计的，在一些特殊的具有大量空间局部性的内存访问模式中能够提升性能并且减少内存流量。

2.2.4 流多处理器细节

通过上面对 sp，warp 和存储器的介绍，我们再介绍下 Kepler 架构下的 SM 细节。从图 2-5 中，可以看到，Kepler 架构下的 SMX 中包含 192 个单精度的 CUDA core，64 个双精度的单元（DP unit）和 32 个 SFU(Special Function Unit, 特殊函数单元)，32 个存储 (load/store) 单元，其中 SFU 用来执行超越函数、插值以及其他特殊运算。Kepler 架构下的 GPU 的每个 SM 包含了 4 个 warp 调度器和 8 个指令发射器。Kepler 架构下的每个 SM 可以同时调度 64 个 warp 共计 2048 个 thread，而 Kepler 架构中一共有 15 个 SM，也就是说，同一时刻最大可以支持 30720 个线程在 GPU 上调度执行，所以 GPU 的并行能力相当强大。

2.2.5 执行模型

GPU 的执行模型被称为 SIMT(Single Instruction ,Mutiple Thread)，SIMT 是对 SIMD 的一种改进[15]。在 CPU 的 SIMD 中，向量宽带是受限的，在 Intel 的 SSE 指令集中，假设一条 SSE 的指令宽带为 128bit，那么它一次可以处理 4 个单精度浮点数，如果要用 SSE 指令处理一个单精度浮点数组，必须将数组打包成 4 个一组，然后才能把这些组交给 CPU 进行计算。但是在 CUDA 的模型中，我们为相同的指令产生不同的线程，这些线程的数量可以自由设置。在 Kepler 架构中，同一时刻 GPU 上可以调度执行 30720 个线程，这比 CPU 的 SIMD 的并行粒度高很多。

2.3 CUDA 编程模式

CUDA 是一种异构计算模型，CUDA 程序在 CPU 和 GPU 上交互执行，CUDA 的程序可以分为主机部分和设备部分，设备代码在一个或者多个 GPU 上执行，主

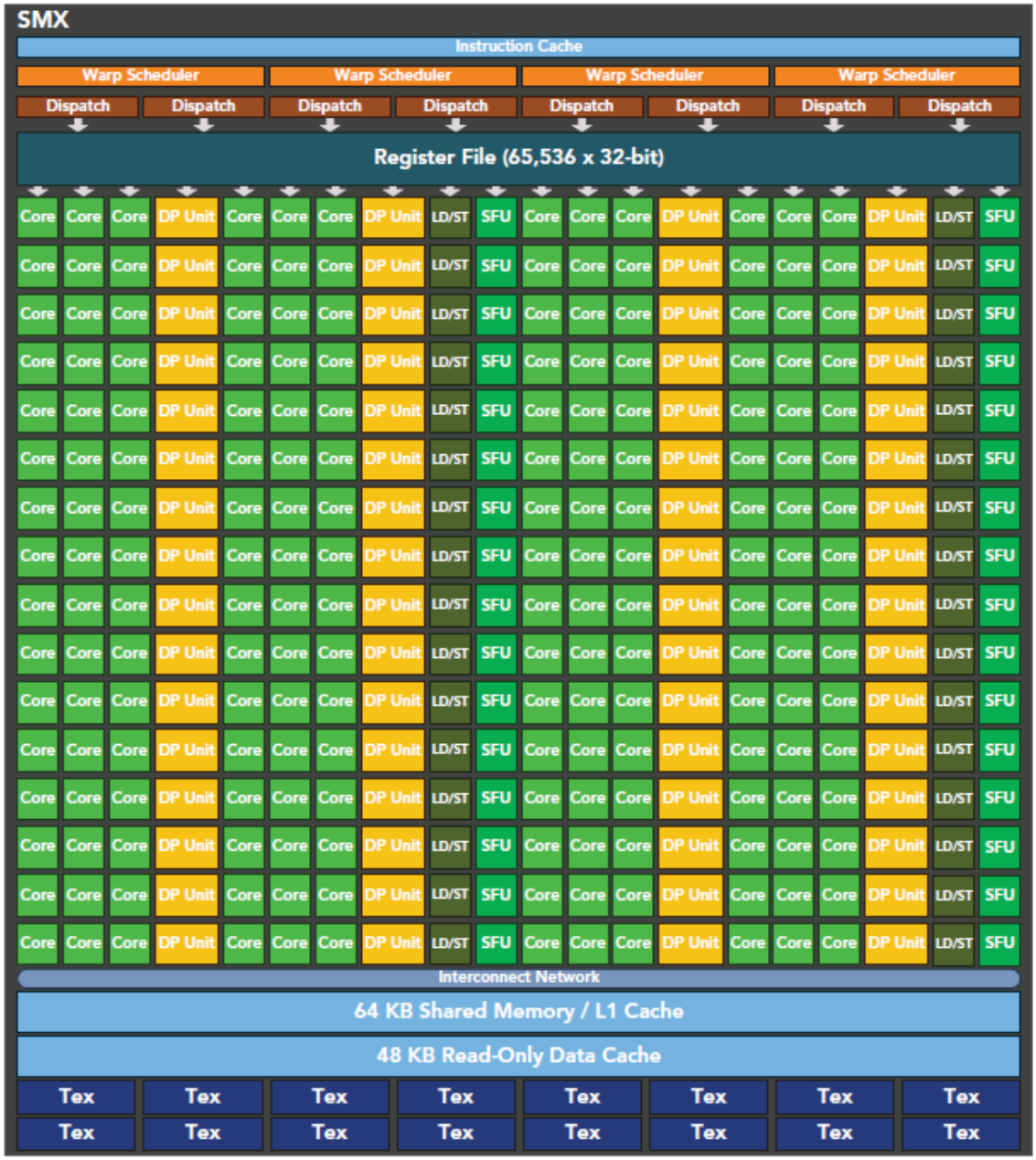

图 2-5 Kepler 架构下的 SM 细节

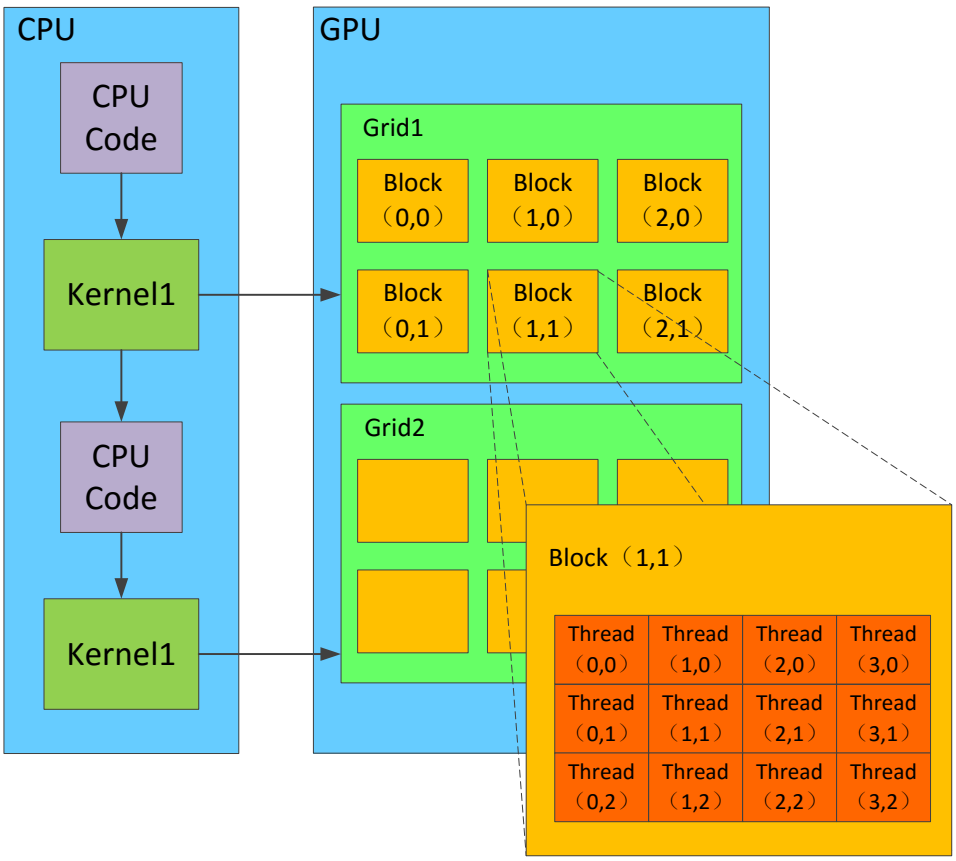

图 2-6 CUDA 异构执行流程和线程组织

机部分的代码在主机上执行，负载调度 GPU 上的设备代码执行过程。图 2-6 展示了 CUDA 异步执行流程，CPU 通过调用 kernel 函数的方式控制 GPU 执行并行程序。

2.3.1 CUD 软件线程组织

一个 CUDA 程序包含大量并行执行的线程，在软件层面上，我将多个线程组成一个 block，同一个 block 中的线程又被分成不同的 warp 组，一个 block 中的线程只能在同一个 SM 中执行，同一个 block 中的线程可以通过调用同步函数(__syncthreads()) 进行同步，同一个 blcok 中的线程可以共同访存 shared_memory，通过 shared_memory 实习通信。一个 kernel 函数可以在软件层面上产生大量的线程，这些线程被组织成一个一个的 block，如果 SM 足够，这些 block 中的线程将被加载到 SM 上调度执行，如果 SM 不足，block 需要排队等待其他 block 完成执行。多个 block 组成一个 Grid，block 的维度可以是 1-3 维，图 2-6展示了一个二维

的 block 组织，每个 block 有自己的二维标号，同样每个 block 内部的线程也可以被组织成 1-3 维，图 2-6展示了一个 block 内部的 thread 二维标号。

2.3.2 kernel 函数

kernel 函数是 GPU 上每个线程执行的函数，kernel 函数在主机端调用，在 GPU 端执行，在调用 kernel 函数时需要为 kernel 函数设置一些参数，包括 Grid 维度大小和 block 维度大小。在 kernel 函数中使用 blockIDx.x,blockIDx.y,blockIDx.z 三个标号来访问当前 block 在 Grid 中的三维坐标，使用 threadIDx.x,threadIDx.y,threadIDx.z 三个标号来访问当前线程在 block 中的三维坐标的。kernel 函数中使用 blockDim.x,blockDim.y,blockDim.z 分别表示 block 的三个维度的宽度。

2.3.3 CUDA 线程同步

2.2.2 介绍过同一个 warp 中的 32 个线程的执行是天然同步的，但是如果程序中存在更多的线程需求的话，我们就不能仅仅依赖于 warp 的性质进行同步，CUDA 支持同一个 block 中的所有线程进行同步，通过调用同步函数 __syncthread(), 一个 block 中的线程可以实现同步，它表示 block 中所有的 thread 都要同步到这个点（__syncthread() 执行处）才能继续执行，这种同步操作相当与 CPU 上多线程的 barrier(屏障) 操作。GPU 上的线程同步的另一个问题是线程的写操作的同步问题，多个线程同时写相同地址的数据不是线程安全的。CUDA 提供了一系列的原子操作来对多个线程间的共享数据的读写进行互斥保护，比如 atomicAdd 函数，他对目标地址的值进行原子加法操作，保证每次只有一个线程对数据进行加法操作。

2.3.4 CUDA 流并行

CUDA 程序通过流来管理并发，CUDA 流代表一系列的 GPU 操作组成的队列，这些队列内部的操作按照顺序在 GPU 上执行，当我们调用 kernel 函数或者调用 CUDA 内部函数（cudamemcpy（），cudasyncthread() 等）时，就是把一系列的 GPU 操作加到一个 CUDA 流队列中，这个流队列的操作会在 GPU 上按照加入的顺序执行。GPU 上可以同时存在多个流队列，不同流之间的操作是无关的，流之间可以相互切换以充分利用 GPU 上的计算资源，虽然同一个流中的操作必须顺序执行，但是不同流之间的操作顺序是乱序的，而且还有可能是同时并行执行的，所以流并行也是 GPU 上的一个并行层次，我们可以把它视为任务上的并行，每个流代表了不同的任务，不同任务间的操作可以不断切换来隐藏延迟，从而提高了 GPU 的 SM 利用率，增大了并行粒度。同时，任务上的流并行，对编程人员来说是比较容易理解的，给 GPU 的并行代码设计带来了方便，我们可以为相似的任务

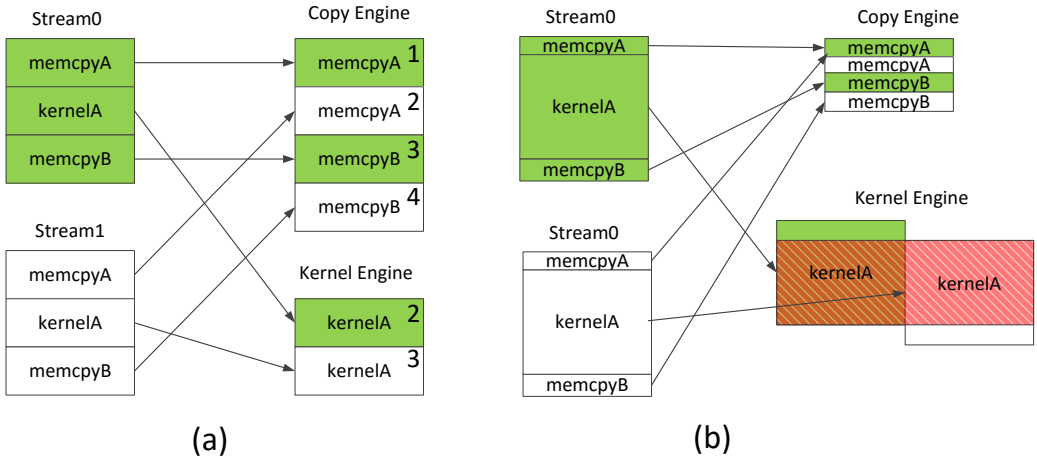

图 2-7 流并行示意图

设计相同的代码，通过流并行提高并行粒度，减小了设计难度。

我们观察下面图2-7中简单的流并行例子，来分析流并行的优势，图2-7中的 Copy Engine 表示 GPU 上的内存拷贝单元，负责和 CPU 端进行数据传输，Kernel Engine 表示 GPU 上的 kernel 执行单元，负责 kerenl 的调度执行。在图 2-7(a) 中一个有两个流，两个流的操作相同，假设图中的每一个操作的执行时间相同，那么如果两个流串行执行，则总共需要 6 个时间单位，但是如果两个流相互切换，当 Copy Engine 在进行复制操作的同时，stream0 的 kernelA 在 Kernel Engine 上执行，当 stream0 的 kernelA 执行完毕后，stream1 的 kernelA 也已经准备好可以执行了，这样 Copy Engine 的数据传输带宽和 GPU 的计算带宽都得到充分利用，一共只需要 4 个时间单位，就可以完成两个流的操作。图 2-7(b) 中，假设两个流的复制操作比较少，而 kernel 需要的计算时间较长，这个时候由于两个流的 kernel 都会很快准备好发射，如果 SM 资源足够，两个 kernel 将同时在 GPU 上调度执行，如图2-7(b) 中的红色部分表示重叠执行的时间，可以看到这种情况下多个流的并行可以大大节省运算时间，充分利用 SM 资源。

2.3.5 本章总结

本章简略介绍了 GPU 的体系架构和 CUDA 编程框架，说明了 GPU 的的强大计算能力和一些 GPU 程序设计上需要注意的问题，为后面的 3,4 章介绍做铺垫。

第三章 SDN 网络下的并行路由优化算法设计

3.1 引言

业务量工程问题可以被建模成一个多商品流问题，将网络业务作为问题的输入，寻找最优的路由来最优化效用函数，效用函数通常设置为对网络拥塞程度的评价水平，比如，最常用的效用函数是最小化最大链路利用率（MLU），简单地被定义为利用率最高的那条链路的链路利用率 [33,34]，另外一些把所有链路的链路利用率的和作为效用函数 [35,36]，这些效用函数的逻辑是：（1）低链路利用率意味着低的网络延迟。（2）维持低的链路利用率意味着预留更多的空间给其他将来到达的业务。但是大量基于实际拓扑的实验表明链路利用率效用函数，特别是 MLU，在网络利用率没有达到拥塞程度的时候，不是对网络优化的较好评价函数 [37]。在这个实验 [37] 中，当链路利用率低于 0.9 的时候会造成不可忽略的网络性能中断。反之，在网络发生大量拥塞的情况下 MLU 只去优化最大的链路利用率，而不能给出一个可行（满足容量约束）的解。所以作为替代，本文采用路由代价作为目标函数，我们假设已经知道短时间内到达的一批业务，控制器需要计算出满足链路容量约束的路径，并且最优化总的路由代价。为了使得加入网络的业务尽量多，我们设定被阻塞的业务代价为一个较大值。

本章主要设计了两种业务量工程算法，第一种是基于备选路径模型的业务量工程算法，采用遗传算法来优化目标函数，并且设计了遗传算法的并行版本，获得 10 倍以上的加速比。第二种是基于拉格朗日松弛的优化算法，算法把链路容量约束松弛到目标函数，并把业务量工程问题分解成一批业务的路由计算问题，从而采用 GPU 进行并行计算。

3.2 网络模型和问题建模

3.2.1 网络模型

本文将 SDN 网络建模成有向图 $G(V,E)$，V 表示所有的点集合，E 是所有边的集合，$n=|V|$ 和 $m=|E|$ 分别表示点数和边数。对每一条边 $(i,j)\in E$，w_{ij} 表示此边 (i,j) 上的权重（传输一单位的流量需要的代价），不失一般性，我们假设每条链路上的 w_{ij} 是整数，对每一条边 (i,j)，c_{ij} 表示此边上的容量，假设 D 表示需要被路由的业务需求集合，业务 $d\in D$ 是一个元组 (s_d,t_d,bw_d)，其中，s_d 表示业务的源节点，t_d 表示业务的目的节点，bw_d 表示业务 d 需要的流量带宽。业务量工程问题将网络业务需求和网络拓扑作为输入，计算出每条业务的路由路径以使得效用函数

代价最小化，在 SDN 网络中，业务的路径在中心控制器上计算出来。

3.2.2 问题建模

本小节，我们把业务量工程问题建模成一个混合整数规划模型（MILP），本文采用新的效用函数，如下 3-1：

$$\sum_{d\in D} f(\mathbf{d}) \tag{3-1}$$

$$f(\mathbf{d}) = \begin{cases} c(p_d)\cdot bw_d & \text{如果业务可以被加入} \\ W\cdot bw_d & \text{如果业务被阻塞} \end{cases} \tag{3-2}$$

其中 p_d 是计算出来的对应于业务 d 的路径，$c(p_d)$ ($c(p_d)=\sum_{(i,j)\in p_d} w_{ij}$) 表示的是此路径 p_d 的代价，W 为一个较大的值，比如 W 远远大于业务所有可能的路径代价值。因为不知道带宽是否足够容纳所有业务，3-1有两个分支，为了使得表达式同一，方便表示，我们首先构建辅助图 $G_a(V_a,E_a)$，对每个点 $v\in V$ 和 $u\in V$，在 $G_a(V_a,E_a)$ 中添加一条链路 (v,u)，并且设置链路 (u,v) 的容量和代价分别为 ∞ 和 W(W 为一个较大的值)，$G_a(V_a,E_a)$ 为一个全连通图。然后，我们把 $G_a(V_a,E_a)$ 和原图 $G(V,E)$ 合并成一个新图 $G_b(V_b,E_b)$，那么图 $G_b(V_b,E_b)$ 就有足够的容量来容纳业务需求，如果某条业务被路由到 $G_a(V_a,E_a)$ 的链路上，那么就表示这条业务被阻塞了，构建了辅助图 $G_b(V_b,E_b)$ 后，路由优化的效用函数可以表示为：

$$z^* = minimize\, f(\mathbf{d}) = \sum_{d\in D} c(p_d)\cdot bw_d = \sum_{d\in D}\sum_{e\in p_d} w_e\cdot bw_d \tag{3-3}$$

在本文的业务量工程问题中，每个业务只能够路由到一条路径上，如第二章中讨论的那样，这更符合于 SDN 网络的实际情况，以下整数约束能够保证每个业务只走一条路径

$$\sum_{(i,j)\in E_b} x_{ij}^d - \sum_{(j,i)\in E_b} x_{ji}^d = \begin{cases} 1 & \text{if } i = s_d \\ -1 & \text{if } i = t_d \\ 0 & ohterwise \end{cases}$$
$$\forall i\in V_a, \forall d\in D \tag{3-4}$$

其中 x_{ij}^{d} 是一个 0，1 整数变量，$x_{ij}^{d}=1$ 表示业务 d 路由经过链路 (i,j)，为了避免链路拥塞，路由路径需要满足以下的链路容量约束:

$$\sum_{d\in D} x_{ij}^{d}\cdot bw_{d}\le c_{ij}\ \ \forall(i,j)\in E_{b} \tag{3-5}$$

在这个模型中，变量的数量随着业务量大小和网络规模大小呈倍数增长，所以这个 MILP 模型在大规模情况下很难求解。

3.3 基于遗传算法的路由优化算法

3.3.1 备选路模型

模型 (3-3,3-4,3-5) 的一种常见的简化模型被称为基于备选路径的模型[22]，在备选路径模型中，为每一个业务 $d\in D$，预先产生了 K 条不同的路径作为备选路径集合 $P_{d}=\{p_{d}^{1},p_{d}^{2},p_{d}^{3}\cdots p_{d}^{K}\}$，通过在这些备选路径中进行选择来优化目标函数3-3，为了表示业务被阻塞的情况，我们在业务的备选路径集合 P_{d} 中添加一条路径 p_{d}^{0}，p_{d}^{0} 是图 $G_{a}(V_{a},E_{a})$ 上的路由，其代价为 W，所经过的链路容量为 ∞。于是，模型可以被描述如下：

$$z^{*}=minimize\, f(\mathbf{d})=\sum_{k\in[1,K]}\sum_{d\in D} c(p_{d}^{k})\cdot x_{k}^{d}\cdot bw_{d} \tag{3-6}$$

$$\sum_{k\in[0,K]} x_{k}^{d}=1 \tag{3-7}$$

$$\sum_{k\in[0,K]}\sum_{d\in D} y_{ijk}^{d}\cdot x_{k}^{d}\cdot bw_{d}\le c_{ij}\ \ \forall(i,j)\in E_{b} \tag{3-8}$$

其中，x_{k}^{d} 为一个 0,1 变量，当 $x_{k}^{d}=1$ 时表示业务 d 选择其备选路径中的第 k 条路径，y_{ijk}^{d} 是一个 0,1 变量，$y_{ijk}^{d}=1$ 表示业务 d 的第 k 条路径经过了链路 (i,j)，约束3-7表示每个业务只能选择一条路径，约束3-8表示经过链路上的总流量大小不能超过链路上的容量大小。在这个模型中，由于我们把业务的路径限制在其备选路径集合中，所以他的变量个数比模型 (3-3,3-4,3-5) 要少很多，本节将通过遗传算法来求解这个简化的模型，设计出基于遗传算法的并行业务量工程算法 GA-PTEA(Genetic algorithm based parallel traffic engineering algorithm)。

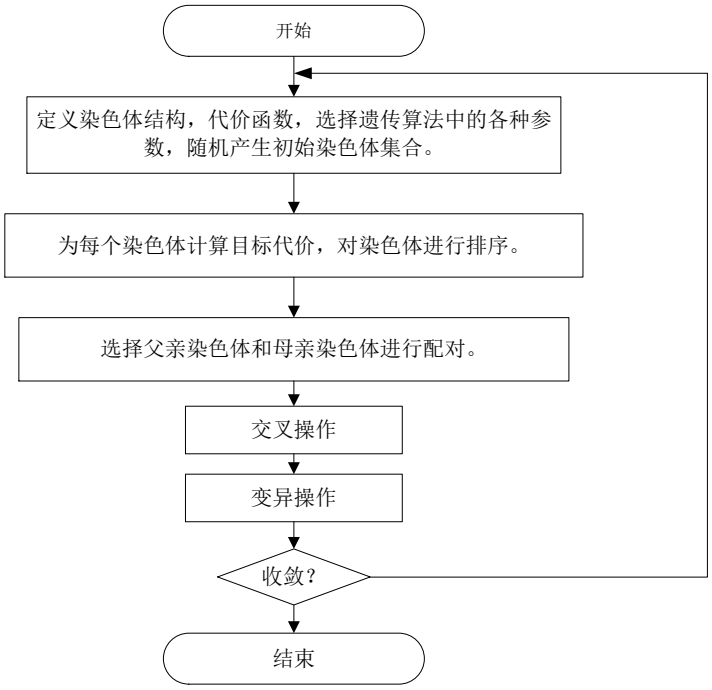

图 3-1 GA 算法流程图

3.3.2 遗传算法设计

遗传算法是一种模拟自然进化过程搜索问题最优解的启发式算法，遗传算法模仿达尔文进化论和自然选择过程来评价挑选最优解集合，从而找寻较优化的解，遗传算法从一个代表问题的可行解的种群出发，一个种群中不同个体代表了不同的解，每个个体实际上是一个染色体，染色体携带表达当前解的信息编码，初代种群产生后，对每个染色体个体进行评价，按照适者生存，优胜劣汰的原则，使得较优的个体更有可能把自己的遗传信息传递给下一代，从而得到更优化的后代，算法过程中，对一部分基因进行变异，好的变异能够提高解的质量，增加算法的搜索空间，避免算法收敛于局部最优解。遗传算法的步骤流程图如下 3-1 所示：

3.3.2.1 染色体结构

假设业务数量为 $|D|$，初始染色体集合大小为 POP，集合 C 表示染色体种群集合，C_j 表示第 j（$j \in [1:POP]$) 号染色体，$C_j^i = k, k \in [1:K]$ 表示在第 j 号染色体中，业务 i 选择了第 k 条备选路径，$C_j^i = 0$ 表示业务 i 被阻塞（在辅助图 $G_a(V_a, E_a)$ 上路由)，不占用链路资源，$|p_d^i|$ 表示业务 d 的第 i 条备选路的路径代价，rp_d^i 表示路径 p_d^i 上的最小可用容量,$rp_d^i = min(r_e | e \in p_d^i)$，其中 $e \in p_d^i$ 表示路径 p_d^i 上的边，r_e 表示此边 e 上的剩余容量。

3.3.2.2 初始可行解的生成

可行解表示满足容量约束的解，为了使得遗传算法有效，初始解的质量很重要，产生的初始解要尽量好，要有更多的业务要能加入网络中，而且保证业务的路径代价较小，本文采用一种简单的贪心算法来产生出初始可行解，算法如下3-1所示：

Require: $G(v,E)$: 网络拓扑;P: 备选路径集合;C: 未初始化的染色体集合;
Ensure: C: 可行染色体集合;

1: **for** each $c_j \in C$ **do**
2: **for** each $c_j^d \in c_j$ **do**
3: $c_j^d \leftarrow k_j^d$，其中 k_j^d 为 1 到 K 之间的随机值，随机选择一条备选路
4: **end for**
5: 对染色体中的的每个业务需求按照值 $\frac{bw_d}{\sqrt{|p_d^{k_j^d}|}}$ 进行降序排序。
6: **for** each $c_j^d \in c_j$ **do**
7: **if** $rp_d^{k_j^d} >= bw_d$ **then**
8: 加入路径 $p_d^{k_j^d}$ 到网络，更新网络链路容量。
9: **else**
10: $c_j^d \leftarrow 0$
11: **end if**
12: **end for**
13: **end for**

算法 3-1 初始可行解产生

对每一个染色体，算法随机地为每个业务选择备选路径标号，但是这样选择出来的路径集合有可能会超过网络链路的容量限制，从而使得解变得不可行，要得到可行解，必须从解中剔除一部分业务，使得他们阻塞，本文使用一种启发式策略来确定需要剔除的业务，一方面，要使得目标函数变小，那些流量需求较大的业务应该优先被加入到网络中，但是。另一方面，如果大流量的业务的路由代价很大，经过了一条很长的路径，就会大量地浪费网络中的链路容量资源。所以算法对当前染色体 j 中的业务和其路径按照 $\frac{bw_d}{\sqrt{|p_d^{k_j^d}|}}$ 的值进行排序，其中 bw_d 代表当前业务 d 所需要的流量大小，$|p_d^{k_j^d}|$ 代表集合 P_d 中的第 k_j^d 条路径的代价大小，观察目标函数，目标函数是最小化路由代价，而 $\frac{bw_d}{\sqrt{|p_d^{k_j^d}|}}$ 较大意味着较大的流量经过较小代价的链路进行路由，这种路由是很理想的，尽量节省网络的链路使用资源的同时，又减小了总体目标函数，所以这个比例值是对业务路由优劣程度的较好评价。算法按照比例值排序的结果依次尝试将路径 $p_d^{k_j^d}$ 加入到网络中，如果 $rp_d^{k_j^d} >= bw_d$，表示路径经过的链路有足够的容量来容纳这一业务，所以加入业务到网络中，并

且更新网络的链路容量大小，反之，如果 $rp_d^{k_j^d} < bw_d$，这个业务选择这一条路径会超过网络链路的容量限制，于是这条业务被阻塞，染色体中的相应基因位置被设置为 0。我们重复上面的过程 *POP* 次，就得到一个初始可行染色体集合 *C*。

3.3.2.3 评价

评价过程对每一个染色体计算其相应的目标效用函数值，并且对染色体按照效用函数值进行降序排序，由于遗传算法中的交叉和变异步骤可能会产生不可行的染色体解（链路容量超限），我们把这样的染色体评价为一个很大的代价（INF），从而在选优时被排除掉。我们把排序好的染色体分为三组：

精英染色体集合：目标值排在最前面的 α 个染色体构成精英染色体集合 *A*，精英染色体将直接保留到下一轮迭代，而且精英染色体可以产生后代。

次优染色体集合：精英染色体后的 β 个染色体构成次优染色体集合 *B*，次优染色体集合中的染色体不会直接保留到下一轮，但是他们可以和精英染色体进行交叉产生后代。

劣等染色体集合：最后剩余的 γ 个染色体构成劣等染色体集合 *G*，由于其目标函数值一般较大，其选路策略不可取，算法直接扔掉这一部分劣等染色体，不允许劣等染色体产生后代。

3.3.2.4 交叉

交叉过程从精英染色体集合中 *A* 中随机选取一个染色体 $c_i \in A$ 作为父亲，从精英染色体集合和较优染色体集合的并集 $A \cup B$ 中随机选取一个染色体 $c_j \in A \cup B$ 作为母亲，将 c_i 和 c_j 进行均匀交叉得到新的染色体 *s*，均匀交叉的过程是，对 *s* 的每一个基因点位以 %50 的几率选择继承父亲或者母亲的相应点位的路径选择。重复以上过程 β 次，从而产生 β 个新的子染色体。

3.3.2.5 变异

变异过程采用随机变异，随机在集合 $A \cup B$ 中选取 γ 条染色体，对某一选定的染色体 $c_j \in A \cup B$，随机选取 *m* 个业务基因点位，进行变异，将当前已经选择的路径编号随机改变为备选路集合中的另外一个值，由于变异过程是为了提高算法的的搜索空间，避免算法陷入局部最优解，但是实际实验过程中发现如果 *M* 和 *m* 值设置较大，可能使得算法收敛较慢，因为大量的变异可能会导致较优秀的可行解变成不可行，因此会丢掉这些优秀的解，因此实验中 *m* 的值设置得较小。

3.3.2.6 终止条件

假设第前 k 次迭代找到的最优解为 B^*，第 k+1 次迭代找到最优解为 b_{k+1}^*，如果 $b_{k+1}^* < B^*$，那么就更新 $B^* = b_{k+1}^*$，如果连续迭代 L 次，B^* 都不被更新，则判定算法收敛，算法停止。

3.3.3 基于 GPU 的并行遗传算法设计

3.3.3.1 并行评价算法设计

遗传算法中最消耗时间的部分是染色体评价部分，由于需要评价大量的染色体，评价每个染色体都需要大量计算开销，但是幸运的是遗传算法具有天然的并行性，每个不同的染色体评价可以并行执行，更进一步，每个染色体中的不同基因的计算也可以并行执行，这样并行粒度是达到 $|POP| \cdot |D|$。在具体介绍并行算法之前，我们先引入一些符号，如下：

染色体（chromosomes）相关符号：C 表示染色体种群集合，c_j 表示第 j 个染色体，c_i^d 表示第 i 个染色体的第 d 个基因位置。

备选路径（paths）相关符号：P 为所有业务的备选路径组成的集合，P_i 表示第 i 业务的所有备选路径集合，$p_i^k \in P_i$ 表示集合 P_i 中的第 k 条路径。

业务带宽（bandwidth,bw），bw_i 表示第 i 个业务需要的带宽大小。

链路流量（flow,f），f_e 表示链路 e 上占用的流量大小。

链路单位代价（weight,w），w_e 表示链路 e 上的代价。

链路流量（capacity,ca），ca_e 表示链路 e 上的容量大小。

共享内存中间数组（shared,sh），sh_e 表示链路 e 上的总代价。

目标函数值数组（objectives,ob），ob_j 表示第 j 个染色体对应的目标函数值。

由于每个染色体的计算过程是独立的，算法为每一个染色体开辟一个 block，一个 block 内部的第 d 号线程负责当前染色体上第 d 号业务的计算。算法 3-2展示了负责评价的 kernel 函数 evaluate，算法一共可以分为两个部分：

第一，流量统计部分：算法需要对染色体进行可行性判断，也就是判断是否有链路的流量大小超过其容量大小，为此算法需要统计每一条边上的流量大小，我们在共享内存中开辟数组 f 用以统计流量，f 数组被初始化为 0。一个 block 中的每一个线程首先通过寻址备选路径集合找到业务选择的路径，业务每经过一条链路都需要占用链路上的容量，所以如果路径经过了链路 e，f_e 就需要加上业务的流量大小，block 中的每一个线程都会遍历业务所选择的路径，对 f 数组进行加法操作，但是多个线程可能同时对 f 数组中的同一个地址进行加法操作，这不是同步安全的，所以算法使用 atomicAdd 操作进行原子加法，避免同步问题。当线程完成对 f

function EVALUATE(C,P,bw,w,ca,ob)
    在 block 上分配大小为 |E| 的共享内存空间组成数组 f，初始化数组 f 中的所有值为零。
    $j \leftarrow$ block ID
    $d \leftarrow$ thread ID
    $k \leftarrow c_j^d$
    for $e \in p_d^k$ **do**
        $atomicAdd(f_e, bw_d)$
    end for
    调用 block 线程同步函数 __syncthreads()
    在 block 上分配大小为 |E| 的共享内存空间组成数组 sh，初始化数组 sh 中的所有值为零。
    $e \leftarrow threadID$
    if $f_e < ca_e$ **then**
        $sh_e \leftarrow f_e \cdot w_e$
    else
        $sh_e \leftarrow INF$
    end if
    调用 block 线程同步函数 __syncthreads()
    $s \leftarrow |E|$
    while s>1 **do**
        if e<s/2 **then**
            $sh_e \leftarrow sh_e + sh_{(e+(s+1)/2)}$
        end if
        调用 block 线程同步函数 __syncthreads()
        $s \leftarrow (s+1)/2$
    end while
    $ob_j = sh_0$
end function

算法 3-2 kernel 函数 evaluate

数组的统计之后，必修进行同步（syncthreads()），同步操作保证 block 内部的所有线程执行到同一步骤，也就是先统计完成的线程必修等待其他统计线程也执行完成后才能继续执行，因为只有所有线程都完成对 f 数组的加法操作，f 数组的值才完整，才能够进行下一步操作。

第二，效用函数计算部分：效用函数为链路的代价和，算法需要先计算每一条链路上的代价值，我们在共享内存中开辟数组 sh 来对每一条边的代价进行统计，每一个线程负责对一条链路代价的计算。比如，线程 e 负责第 e 条边的代价计算，线程 e 首先判断链路的流量是否溢出（$ca_e < f_e$），如果溢出则说明此染色体表示的解不可行，设置 sh_e 为 ∞，这使得这个染色体在排序时会被归类到劣等集合，从而在下一次迭代之前被剔除掉。反之，如果 $f_e \leq ca_e$，则说明链路容量足够，链路

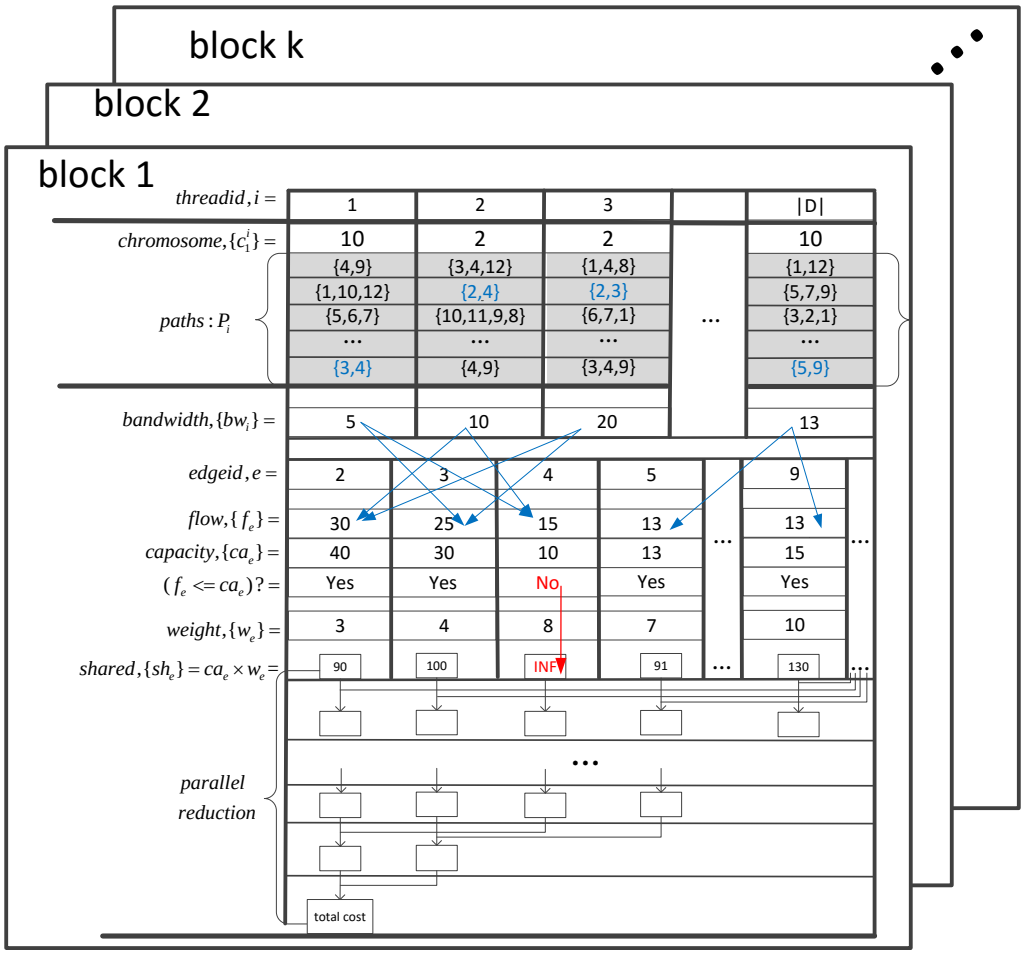

图 3-2 GPU 并行评价示例

的代价为 $f_e \cdot w_e$，设置 sh_e 为 $f_e \cdot w_e$。和 f 数组的统计一样，当线程统计完成 sh_e 之后，也需要进行同步操作，等待其他线程完成统计。sh_e 数组计算完成之后，还需要把 sh_e 中的值进行加和才能得到效用函数的值，为了充分利用 GPU 多线程，算法最后进行并行规约操作（20-26 行）进行求和，for 循环中每次将后一半的 *sh* 数组加到前一半，规约过程中必修进行同步（syncthreads()），以保证加法过程计算完整，最终求和值规约到 sh_0，将 sh_0 中的值写入到 *ob* 数组中。

图3-2为 GPU 上并行评价算法的一个具体例子，图中被标位蓝色的数组表示业务选择的路径，蓝色的箭头表示流量统计时的加法操作，红色箭头表示链路流量超过链路容量，使得链路代价被设置为 ∞。

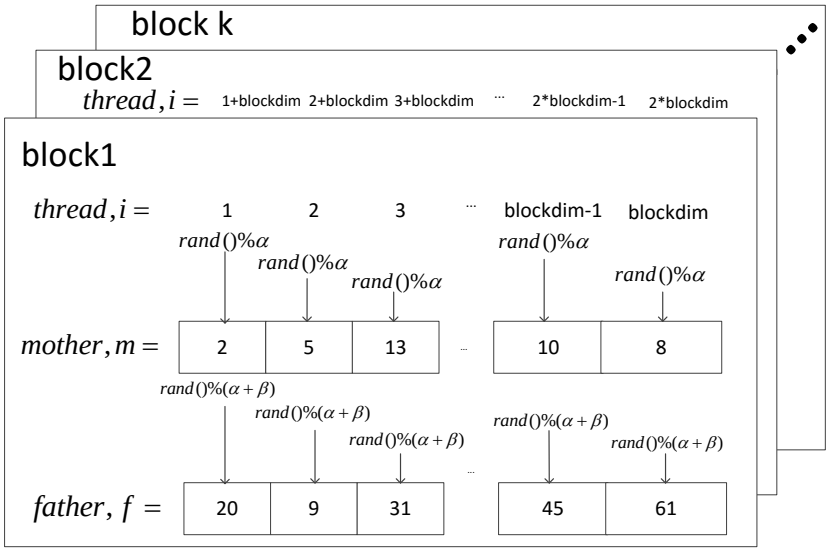

图 3-3 GPU 并行父母选取过程

3.3.3.2 并行排序，变异与交叉

由于遗传算法中最消耗时间的部分是评价部分，本设计中对其他部分的并行步骤采用较简单的算法。在评价部分结束后，需要对所有的染色体按照效用函数的大小降序排序，本设计采用 CUDA 提供的 Thrust [14] 中提供的排序函数对 GPU 上的染色体进行排序，Thrust 是基于标准模板库（STL）的 CUDA C++ 模板库，Thrust 为程序员提供常见的 CUDA 算法库，能够减少程序员工作量并且提高应用程序效率，Thrust 库中的排序函数已经针对 GPU 架构做了很好得优化，所以本文不在对排序部分进行优化，排序函数细节可参考 NVDIA 官方文档 [14]。

交叉过程分为父母选取和交叉计算两个 kernel，父母选取过程并行地选取 β 对父母，每个线程负责选取一对父母，并且将选取的父母下标记录到 *father* 和 *mother* 数组中, 父母选取 GPU 计算示意图如下图3-3所示： 每个线程进行两次随机，*mother* 标号只能在前 α 个精英染色体中选取，*father* 表号的选取在前 $\alpha+\beta$ 中选取，这样既能够给予精英染色体更大的繁殖几率，也可以保证解的搜索空间变化足够大，避免陷入局部最优值。

父母选取过程结束后，在 GPU 端记录了所选取的 *mother* 和 *father* 数组，*mother* 和 *father* 数组将用于交叉部分的计算，交叉部分的并行粒度更高，其中每个基因点的计算都是并行执行的，如下图 3-4 所示： 每个 block 负责一个新染色体的生成，

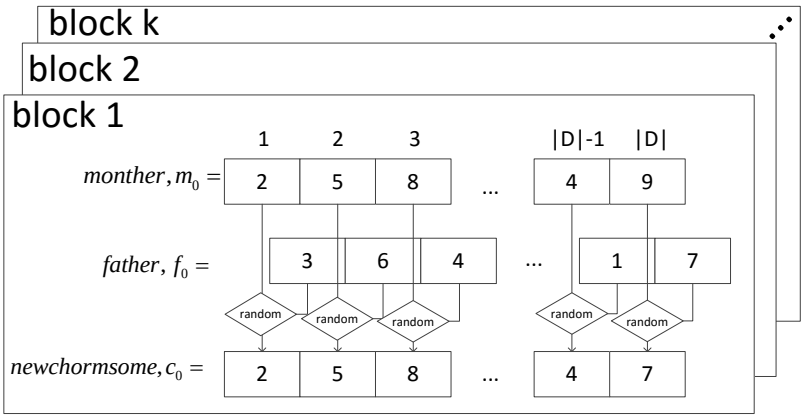

图 3-4 GPU 并行均匀交叉过程

通过之前的分析，一共需要生产 β 个新的染色体，所以 GPU 端一共需要分配 β 个 block，其中每一个 block 中有 |D| 个线程来同时负责随机从 *mother* 和 *father* 的相应点位选择一个来作为新染色体相应点位的值，如前所说采用均匀交叉策略，选择父亲母亲遗传基因的概率都是 50%，这样 block 内部的并行度达到 $|D|$，总的并行粒度达到 $\beta \cdot |D|$。

最后，对于变异部分，我们一共开辟 γ 个线程，每个线程随机在 $A \cup B$ 集合中选取一个染色体，并随机选取 m 个点位进行变异，并行粒度为 γ。

3.4 基于拉格朗日的优化算法设计

GA-PTEA 算法虽然简单，但是其具有以下缺点：第一，需要事先计算大量备选路径，不能很好地适应网络的动态变化，一旦网络链路发生变化，又要重新计算备选路径。第二，遗传算法从开始到收敛需要大量的迭代次数，虽然经过 GPU 加速，但是由于收敛缓慢，仍然需要大量的计算时间。第三，遗传算法容易陷入局部最优解。

本节采用基于拉格朗日松弛的模型来解决业务量工程问题，并根据这个模型设计出并行算法 LR-PTEA(Lagrange relaxing based parallel traffic engineering algorithm)，LR-PTEA 相对与 GA-PTEA 有以下优点：第一，LR-PTEA 能够快速收敛。第二，LR-PTEA 求得的解大大优于 GA-PTEA。第三，LR-PTEA 不需要事先产生备选路径，业务路由是实时计算出来的。

3.4.1 基于拉格朗日松弛的模型

在模型 (3-3,3-4,3-5) 中，网络容量约束 (式子 3-5)，把所有的路由变量联系到一起，因为这些变量的取值必修保证每一条链路上占用的流量小于链路的容量大小，正是由于存在链路容量约束，每个业务的路由选取才变得不相互独立，但是要利用 GPU 的并行特性，需要寻找独立计算的可能性，因此，本文采用拉格朗日松弛方法，把一个业务量工程问题分解成一批业务的路由计算问题，而这些业务的路由计算问题是相互独立的，很适合并行计算。

将模型 (3-3,3-4,3-5) 中的网络容量约束松弛进目标函数得到如下拉格朗日子问题：

$$L(\boldsymbol{\lambda}) = min \sum_{d\in D} \sum_{(i,j)\in E_b} w_{ij} x_{ij}^{d} bw_d + \sum_{(i,j)\in E_b} \lambda_{ij} \left(\sum_{d\in D} x_{ij}^{d} bw_d - c_{ij} \right) \tag{3-9}$$

其中 λ_{ij} 表示链路 (i,j) 的拉格朗日乘子。

表达式 (3-9) 还可以表示为：

$$L(\boldsymbol{\lambda}) = min \sum_{d\in D} \sum_{(i,j)\in E_b} (w_{ij} + \lambda_{ij}) x_{ij}^{d} bw_d - \sum_{(i,j)\in E_b} \lambda_{ij} c_{ij} \tag{3-10}$$

受限于：

$$\sum_{(i,j)\in E_b} x_{ij}^{d} - \sum_{(j,i)\in E_b} x_{ji}^{d} = \begin{cases} 1 & \text{if } i = s_d \\ -1 & \text{if } i = t_d \\ 0 & otherwise \end{cases}$$
$$\forall i \in V_b, \forall d \in D \tag{3-11}$$

拉格朗日子问题的目标函数中的 $\sum_{(i,j)\in E_a} \lambda_{ij} c_{ij}$ 这一项，不随着拉格朗日乘子的变化而变化，本文将其作为常数项而丢掉不讨论，丢掉 $\sum_{(i,j)\in E_a} \lambda_{ij} c_{ij}$ 这一项后，拉格朗日子问题的目标函数中只含有代价部分 $w_{ij} + \lambda_{ij}$ 和 $x_{ij}^{d} bw_d$ 的乘积。注意到，$\sum_{(i,j)\in E_a} (w_{ij} + \lambda_{ij}) x_{ij}^{d} bw_d$ 表示业务 d 的路由代价，因此，拉格朗日子问题的目标函数是最小化所有业务的路由代价总和，观察这个子问题的约束，我们发现，每一个约束都只含有一个和业务需求相关的变量，所以，这个拉格朗日子问题可以被分解成一系列独立的最短路径问题（每个业务需求对应于一个最短路问题），只是这些最短路径问题的链路代价发生了改变，链路代价变得和拉格朗日乘子 $\boldsymbol{\lambda}$ 相关，也

就是说给定一个拉格朗日乘子 $\boldsymbol{\lambda}$，我们可以将拉格朗日子问题看成一批单业务的最短路径问题，我们可以通过并行地计算一系列的最短路径来解决这个拉格朗日子问题。

因为把容量约束松弛进效用函数中后，不会增加目标函数的值，$L(\boldsymbol{\lambda})$ 成为原问题最优目标函数值的下界，$z^* \geq L(\boldsymbol{\lambda})$，为了得到最紧的的下界值，我们要解决以下这个优化问题：

$$L^*(\boldsymbol{\lambda}^*) = maximize_{\boldsymbol{\lambda}} L(\boldsymbol{\lambda}) \tag{3-12}$$

受限于: (3-11)

以上的这个优化问题也被称为原来业务量工程问题 (式子3-3,3-4,3-5) 的对偶问题 [38]，其中 $\boldsymbol{\lambda}^*$ 表示最优拉格朗日乘子，为了得到最优乘子 $\boldsymbol{\lambda}^*$，可以使用次梯度优化算法来解决，次梯度优化计算时，第一次先初始化乘子 $\boldsymbol{\lambda}^0$, 然后通过以下过程进行迭代求解：

$$\boldsymbol{\lambda}_{ij}^{(k+1)} = \boldsymbol{\lambda_{ij}}^{(k)} + \theta_k g^{(k)} = \boldsymbol{\lambda_{ij}}^{(k)} + \theta_k [(\sum_{d \in D} x_{ij}^d bw_d - c_{ij})]^+ \tag{3-13}$$

其中，$\boldsymbol{\lambda}_{ij}^{(k)}$ 表示第 k 次迭代的对应于边 (i,j) 的拉格朗日乘子，$g^{(k)}$ 是 $L(\boldsymbol{\lambda})$ 对 $\boldsymbol{\lambda}^k$ 的任意一种次梯度，θ_k 表示第 k 次的迭代的步长，标记 $[\alpha]^+$ 表示 α 中符号为正的部分，也就说 $[\alpha]^+ = max(\alpha, 0)$，从表达式 Eq. (3-13) 可以看出来如果链路 (i,j) 上的流量总和超过链路 (i,j) 上的容量，链路 (i,j) 上的 λ_{ij}^k 拉格朗日乘子会增加，也就是表示一些业务流量需要从链路 (i,j) 上移除，另外，为了避免产生负权重的链路代价，当链路容量大于其上的流量时，我们并不去减小此链路 (i,j) 上的 λ_{ij}^k。根据以上讨论，我们给出基于拉格朗日乘子法的并行路由优化算法的框架，如图3-5所示，LR-PROA 主要包括以下步骤：

步骤一，为 $G_a(V_a, E_a)$ 初始化链路权重。

步骤二，计算所有业务的最短路径，其中路径计算任务被分配到 GPU 进行并行计算。

步骤三，为了从当前计算出来的路径中得到原问题的优化目标函数值，对步骤二中计算出来的路径进行调整。

步骤四，更新链路权重，更新完毕后，如果停止条件不满足，则回到步骤二，进入下一轮迭代。LR-PROA 如果在连续 L 次成功的迭代后依然不能找到更优的全局解，则停止算法过程。

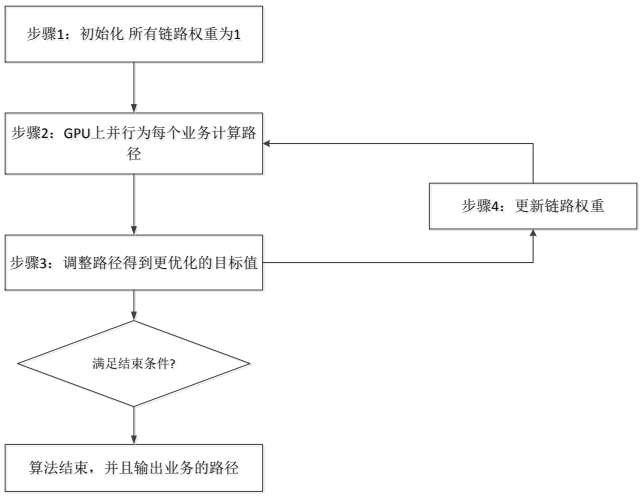

图 3-5 LR-PROA 算法流程图

3.4.2 基于 GPU 的并行路由计算

在每次迭代中，LR-PROA 为每个业务 $d \in D$ 在图 $G_a(V_a, E_a)$ 中计算最短路径，显然，丢掉链路容量约束后，不同业务的最短路径计算可以独立在 GPU 上并行执行，但是，最短路算法的逻辑对于 GPU 来说太过复杂，GPU 最初是被设计来做大规模的数值计算问题，其只实用于逻辑比较简单，但是数值计算量较大的任务，所以在 GPU 上直接开辟一个线程来计算一个业务的路径，不仅仅在计算上是低效的，而且这样的并行粒度也不能充分利用 GPU 的大规模并行能力。为了充分提高最短路径的计算速度，LR-PROA 对最短路径算法进行并行化设计。 文章 [30] 提出一种 Dijkstra 最短路径算法在 GPU 上的并行实现，但是从算法结构上分析，Dijkstra 最短路径算法并不适应于并行算法的设计，所以 Dijkstra 最短路径算法在 GPU 上的实现不能得到很好的加速效果，为了期望得到更好的加速效果，LR-PROA 选择 Bellman-Ford [38] 最短路算法来进行并行实现，Bellman-Ford 最短路算法逐步地减小距离标记 $Dist[v], v \in V$，直到其收敛到真实的最短距离。Bellman-Ford 算法过程如算法3-3，其中 $Dist[v]$ 表示距离起点 s 到 v 的最短路径距离,$Pre[v]$ 表示点 s 到点 v 的最短路径上 v 的前驱节点，在初始化好了所有节点的距离数组和前驱节点数组之后，Bellman-Ford 算法最多迭代 $|V|$ 次，每一次迭代算法松弛一次图 $G(V, E)$ 中的所有的边（第 10-11 行）。Bellman-Ford 算法的算法复杂度为 $(|V| \cdot |E|)$，他的复

Require: 网络拓扑:$G(V,E)$; 源点:s;
Ensure: 从 s 开始到其他点的路径集合 P;
1: **for** each node $v \in V$ **do**
2: $Dist[v] \leftarrow \infty$
3: $Pre[v] \leftarrow$ NIL
4: **end for**
5: $Dist[s] \leftarrow 0$
6: $Mark \leftarrow 1$
7: **while** $Mark > 0$ **do**
8: $Mark \leftarrow 0$
9: **for** each link $(u,v) \in E$ **do**
10: **if** $Dist[v] > Dist[u] + w_{uv}$ **then**
11: $Dist[v] \leftarrow Dist[u] + w_{uv}$
12: $Pre[v] \leftarrow u$
13: $Mark = 1$
14: **end if**
15: **end for**
16: **end while**
17: 根据前驱数组 Pre, 重新构建最短路集合，输出路径到集合 P

算法 3-3 Bellman 最短路算法

杂度高于 Dijkstra 最短路径算法的复杂度，但是，因为 Bellman-Ford 算法每次松弛边的操作都是独立无关的，通过为每一条边的松弛操作分配一个独立的线程执行，Bellman-For 算法很容易在 GPU 上实现并行化。

图 3-6 中显示了 LR-PROA 的最短路径计算的并行实现框架。首先，根据业务的源节点将业务分配成不同的组，使得每一组内的业务的源节点相同。我们假设第 i 个组的源节点为 s_i。然后，每一组的最短路径计算使用 m 个 GPU 线程的并行 bellman-ford 算法进行计算，如图3-6所示，线程 $T_{i,j}$ 负责为对应于源点 s_i 的链路 e_j 进行松弛操作。因此总的并行执行的线程数是 $m \times k$, 其中 m 和 k 分别表示链路数目和组的数目。

在本文的最短路算法的并行实现中，每个 block 内部的线程都松弛同一条链路，比如，在图 3-7 中，集合 $\{T_{1j}, T_{2j}, \cdots, T_{ij}, \cdots, T_{kj}\}$ 都在 $block_j$ 上执行，其中 T_{ij} 为对应源节点为 s_i 的链路 e_i 执行松弛操作，其中，我们设链路 e_i 的头节点和尾节点分别为 h_i 和 t_i，可以看到，当这次迭代存在链路更新，那么标记 $Mark$ 会被设置成 1，这是为了优化算法的迭代次数，当某次迭代结束 $Mark = 0$ 则表示这次迭代没有边进行了更新操作，说明 Bellman 算法已经提前结束，实验证明这一个优化可以大大地减小 Bellman 算法的迭代运行次数。

最短路径算法 CUDA 实现算法伪代码如下所示。需要注意的是由于线程在

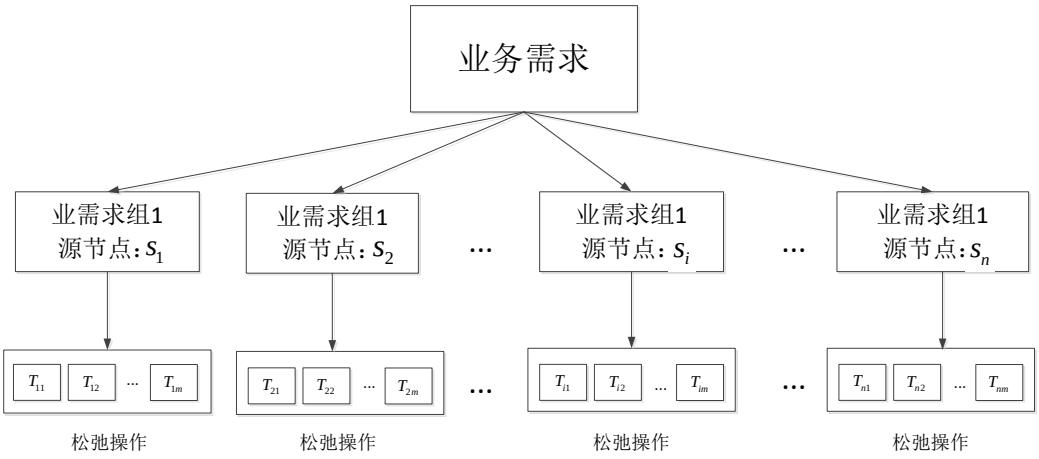

图 3-6 并行业务计算框架

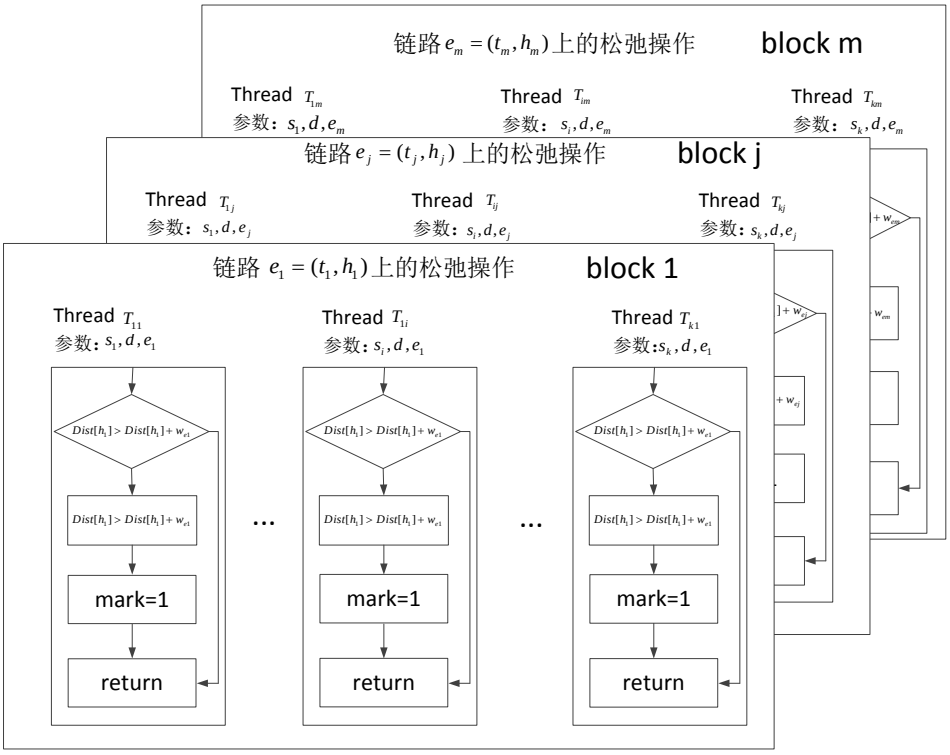

图 3-7 GPU 上 Bellman 算法的实现

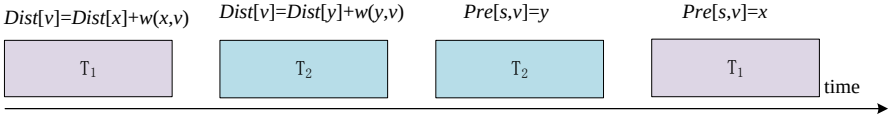

图 3-8 同步问题的例子

Require: 业务需求集合 D; 链路集合 E;
Ensure: 业务需求的最短路径结合 P
1: 将业务的源节点加入到集合 S 中
2: $Mark \leftarrow 1$
3: **while** $Mark > 0$ **do**
4: $Mark \leftarrow 0$
5: 发射 kernel_distance_update(S, E, $Dist$)
6: **end while**
7: 发射 kernel_predecessor_update(S, E, $Dist$, Pre)
8: 根据前驱数组 Pre 重建业务的最短路径，并把路径加入到集合 P 中

算法 3-4 并行最短路计算

1: **function** KERNEL_DISTANCE_UPDATE(S, E, $Dist$)
2: $bid \leftarrow$ block ID
3: $tid \leftarrow$ thread ID
4: 将 (bid, tid) 映射到 id sid
5: $s \leftarrow S[sid]$
6: $e \leftarrow E[bid]$
7: **if** $Dist[s][e.tail] + e.weight < Dist[s][e.head]$ **then**
8: $Mark \leftarrow 1$
9: $Dist[s][e.head] \leftarrow Dist[s][e.tail] + e.weight$
10: **end if**
11: **end function**

算法 3-5 kernel 函数 kernel_distance_update

GPU 上是独立执行的，在更新节点的距离标记和前驱标记的时候会出现同步问题，假设线程 T_1 为链路 (x, v) 执行松弛操作，而线程 T_2 为链路 (y, v) 执行松弛操作，假设两个线程更新点 v 的距离标记和更新前驱标记的顺序如图3-8 所示，如果 $Dist[y] + w(y, v) < Dist[x] + w(x, v)$, 那么 $Dist[v]$ 被更新为 $Dist[y] + w(y, v)$，但是由于更新的顺序发生交叉，节点 v 的前驱节点被更新成了 x, 而不是真正正确的前驱节点 y。为了避免这个同步问题，我们使用两个 kernel，一个用来更新距离标号，一个用来更新前驱节点。

```
1: function KERNEL_PREDECESSOR_UPDATE(S, E, Dist,Pre)
2:     bid ← block ID
3:     tid ← thread ID
4:     将 (bid, tid) 映射到 id sid
5:     s ← S[sid]
6:     e ← E[bid]
7:     if Dist[s][e.tail] + e.weight = Dist[s][e.head] then
8:         Pre[s][e.head] = e.tail
9:     end if
10: end function
```

算法 3-6 kernel 函数 kernel_predecessor_update

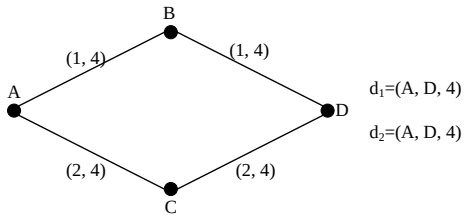

图 3-9 链路更新例子 (1)

3.4.3 链路权重更新

3.4.3.1 权重更新步长

在 LR-PROA 算法中，在第 $(k+1)$ 次迭代，链路 (i,j) 的权重被更新为 $w_{ij}^{k}+\lambda_{ij}^{k+1}$，其中 λ_{ij}^{k+1} 被更新为：

$$\lambda_{ij}^{k+1}=\lambda_{ij}^{k}+\theta_k[(\sum_{d\in D}x_{ij}^{d}bw_d-c_{ij})]^{+}. \tag{3-14}$$

为了保证收敛性，第 k 次迭代的更新步长 (θ_k) 可以被设置为 $\frac{1}{k}$ [38]。然而，通过仿真发现，当 θ_k 被设置为 $\frac{1}{k}$ 时，其收敛缓慢。让我们考虑图 3-9的例子，其中链路 (i,j) 上的标记分别表示链路权重和链路上剩余的容量大小。假设现在有两个业务需求 d_1 和 d_2，他们的源都是 A，目的节点都是 D，且每一个业务的流量大小都是 4 个单位。为了展示这个迭代过程，我们把算法前 5 次的迭代结果表示在表 3-1中，从表中可以看到，业务计算出的最短路径一直在 $A-B-D$ 和 $A-C-D$ 之间徘徊。算法必修等到 $A-B-D$ 和 $A-C-D$ 的两条路的权重相等时才能停止，只有这样这两个业务才可能分离，其中一个选择 $A-B-D$，而另一个选择 $A-C-D$。但是，正如表中所示这需要大量的迭代才能达到。

表 3-1 拉格朗日更新过程（1）

Iteration number	Calculated paths for the two demands	Path weights	θ_k
0	A-B-D A-B-D	2(1+1) 2(1+1)	1
1	A-C-D A-C-D	4(2+2) 4(2+2)	1
2	A-B-D A-B-D	10(1+4+1+4) 10(1+4+1+4)	0.5
3	A-C-D A-C-D	12(2+4+2+4) 12(2+4+2+4)	0.33
4	A-B-D A-B-D	14(1+6+1+6) 14(1+6+1+6)	0.25
5	A-C-D A-C-D	14.66(2+5.33+2+5.33) 14.66(2+5.33+2+5.33)	0.2

另外一种常用的步长选择是：

$$\theta_k = \frac{\rho[UB - L(\boldsymbol{\lambda}^k)]}{||\mathbf{Ax^k} - \mathbf{b}||^2} \tag{3-15}$$

其中 UB 是最优化目标函数的上界，ρ 是一个取值范围为 0 到 2 的常数，**A** 和 **b** 分别是链路相关矩阵和链路上的剩余容量向量。但是从实验中发现设置这种步长迭代效果也不令人满意，因此，本设计采用一种简单但是有效的链路权重更新步长，假设 θ_k^{ij} 为第 kth 次迭代时链路 (i,j) 上的需更新的步长，那么 θ_k^{ij} 为：

$$\theta_k^{ij} = \frac{1}{|c_{ij} - \sum_{d \in D} x_{ij}^d bw_d|} \tag{3-16}$$

从式子 3-16，我们可以看到，如果一条链路上承载的流量大小超过了这条链路上的容量大小，那么这条链路上的权重在下一次迭代之前就会增加 1, 对于其他的流量满足约束的链路，其权重不会改变，如果使用3-16的步长更新方法，LR-PROA 仅仅只需要一次迭代就能够得到例子中 3-9 最优的权重，实验表明，这种粗粒度的更新操作大大的减小了算法的收敛迭代次数，从而大大缩短算法运行时间。

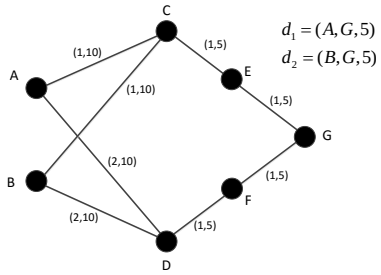

图 3-10 链路更新例子 (2)

3.4.3.2 随机更新策略

拉格朗日松弛法将原问题分解成了一个个独立的最短路径问题，这样使得算法可以并行化进行设计，但是由于每个子问题独立分离，使得每个问题在求最短路径时都是贪心的，这样可能会使得大量业务抢占同一批链路，造成拥塞，一旦拥塞，链路的权重增加，又会使得大量的业务放弃这一批链路，去抢占其他链路，而这些链路由于权重过分增加，造成没有业务去选择他们，使得链路利用出现浪费，甚至会造成其他链路发生拥塞，这样出现恶性循环，最终会使得算法提前收敛到局部最优解，另外，由上一节的分析，为了最求快速收敛，我们简单地将步长设置为 $\frac{1}{|c_{ij}-\sum_{d\in D} x_{ij}^{d} bw_d|}$，就是对把每一个超过容量约束的边简单地增加一，这样粗粒度的增加，可能会加重上面讨论的拥塞循环。 如图3-10所示，假设有存在两个业务 d_1 和 d_2，其中 d_1 的源节点为 A, 目的节点为 G, 其流量大小为 5，d_2 的源节点为 B，目的节点为 G，其流量大小为 5，开始时两个都分别贪心计算最短路径，d_1 选择路径 $A-C-E-G$，d_2 选择路径 $B-C-E-G$, 这样的话，链路 $C-E$ 和 $E-G$ 出现拥塞，表 3-2 展示了这个迭代过程，图 3-10中最优的选择是让其中一个业务经过边 $C-E-G$ 进行中继, 另一个业务经过边 $D-F-G$ 进行中继。但是的迭代过程始终无法使得两条链路发生分离，图中的链路权重始终无法达到最优条件，这是因为算法的权重迭代增加粒度太大，由于链路 $C-E-G$ 和链路 $D-F-G$ 每次超限都会为路径的总权重增加 2 个单位，假设每次迭代时链路 $C-E-G$ 和链路 $D-F-G$ 每次迭代时一共只贡献 1 个单位的权重增加，那么算法只需要一次迭代就能达到最优条件，此时链路 $C-E-G$ 由于超限，权重一共增加 1 个单位，那么路径 $A-C-E-G$ 和路径 $A-D-F-G$ 权重相等都为 4，同样，路径 $B-C-E-G$ 和路径 $B-D-F-G$ 的权重也相等了，这样两个业务才会分离开（比如业务 d_1 选择链路 $A-C-E-G$, 业务 d_2 选择链路 $B-D-F-G$）。但是在算法设计时，我们难以分辨哪些链路的组合会引起业务出现这种徘徊情况，为了解

表 3-2 拉格朗日更新过程（2）

Iteration number	Calculated paths for the two demands	Path weights
0	A-C-E-G B-C-E-G	3(1+1+1) 3(1+1+1)
1	A-D-F-G B-D-F-G	3(2+1+1) 3(2+1+1)
2	A-C-E-G B-C-E-G	3(1+2+2) 3(1+2+2)
3	A-D-F-G B-D-F-G	3(2+2+2) 3(2+2+2)
4	A-C-E-G B-C-E-G	3(1+3+3) 3(1+3+3)
5	A-D-F-G B-D-F-G	3(2+3+3) 3(2+3+3)
6	... ...	... ...

决链路增加粒度过大的情况，在本设计中我们采用随机选择执行更新的策略，也就是对一条流量超过容量约束的边 (i,j)，我们以概率 φ 来对他进行权重更新，每条边的权重增加粒度依然为 1 个单位，假设 $\varphi = 0.5$，这种方法在一定概率上保证图 3-10 中的例子可以在一次迭代中收敛。实际实验中发现这种更新策略能够保证算法得到较优解的同时，保证收敛速度较快。

3.4.4 路径调整

注意到，在最优的权重代价（最优拉格朗日乘子）下，LR-PROA 求解到的路径集合解是拉格朗日对偶问题的优化解，但是它不一定是原来路由优化问题的可行解。作为一个例子，考察图 3-11中的情况，图中的元组中数字分别表示链路的代价和链路上的剩余容量大小，我们假设有两个业务需求 d_1 和 d_2, 其中两个业务的流量需求都是 1 个单位，假设在某一个迭代过程中，LR-PROA 为两个业务算出来的链路都是 $A-C-E-D$, 这条路径选择是对偶问题的最优解，但是他不是原问题的可行解，因为链路 (C,E) 上承载的链路容量大于了链路 (C,E) 上的容

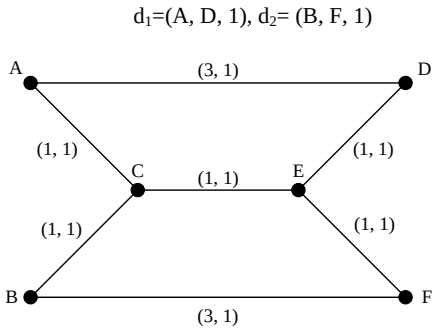

图 3-11 路径调整例子

量。这是由于每个业务在选择路径的时候没有去考虑其他业务所选择的路径，但是我们容易看到这个例子中所有路径权重都一样（都等于 3），也就是说，图中存在着大量的等价链路，但是在这个例子中两个业务恰好都选择了冲突的那一条路径 $A-C-E-D$，如果业务 d_1 的路径被调整到 $A-D$，业务 d_2 的链路被调整到 $B-F$，那么，我们可以得到原问题的最优可行解，所以通过这里的启发，我们发现需要设计一种业务路径调整算法来主动避免链路出现冲突，来得到原问题的优化的可行解。 为了说明业务路径调整算法，我们引入一些符号，假设 P 表示步骤 2 所计算出来的路径集合，其中 $p_d \in P$ 表示业务 d 的路径, rp_d 表示路径 p_d 上的可用带宽，$rp_d = min\{r_e | e \in p_d\}$，其中 r_e 表示链路 e 上的剩余带宽，D_l 表示剩余业务集合，表示这些业务不能在不违背容量约束的情况下被加入网络中。

路径调整算法的主要思想是通过调整一小部分业务的路径来得到原问题的优化可行解，算法首先对业务进行排序，这里采用 3.3.1.2 类似的思想对业务进行排序，一方面，要使得目标函数变小，那些流量需求较大的业务应该优先被加入到网络中，但是如果大流量的业务的路由代价很大，经过了一条很长的路径，就会大量的浪费网络中的链路容量资源，所以算法过程对当前解中的业务和其路径按照 $\frac{bw_d}{\sqrt{|p_d|}}$ 的值进行排序，其中 bw_d 代表业务 d 所需要的流量大小，$|p_d|$ 代表业务 d 的路径 p_d 的代价大小，这样按照顺序试图将业务加入到网络中，如果 $rp_d >= bw_d$，表示业务可以被加入到网络中，那么加入此业务并且更新网络链路的剩余容量值，反之，如果 $rp_d < bw_d$，则表示业务选择的路径上的链路容量不足以承载此业务，那么将业务加入剩余集合 D_l 中，循环结束后，得到一个剩余网络。根据前面的讨论，在剩余网络中可能存在一些等价路径，剩余链路中依然有很多可用资源，所以算法重新在剩余网络中为剩余业务计算路径，算法依次遍历集合 D_l，看能否在剩余网络中为业务寻找一条路径，首先剔除那些链路剩余容量小于业务流量 bw_d 的

Require: 网络拓扑 $G(V,E)$; 业务量需求集合 D; 步骤 2 算出来的路径集合 P_{in};
Ensure: 调整后的路径集合 AP;

把业务按照值 $\frac{bw_d}{\sqrt{|p_d|}}$ 进行降序排序
for each traffic demand $d \in D$ **do**
    if $rp_d \geq bw_d$ **then**
        把路径 p_d 加入到 AP 中
        在图 G(V,E) 中更新路径 p_d 所经过链路的剩余带宽
    else
        把业务 d 加入到剩余集合 D_l 中
    end if
end for
$G'(V',E') = G(V,E)$
for each traffic demand $d \in D_l$ **do**
    $G''(V'',E'') = G'(V',E'$
    for each link $e \in E'$ **do**
        if 链路 $e < bw_d$ **then**
            把链路 e 从图 $G''(V'',E'')$ 中移除
        end if
    end for
    在图 $G''(V'',E'')$ 中为业务 d 计算最短路径 p（链路代价都设为 1）
    if $\frac{|p|}{|p_d|} \leq \delta$ **then**
        把路径 p 加入到 AP，并把业务从集合 D_l 中移除
        在图 $G'(V',E')$ 中更新路径 p 所经过链路的剩余带宽
        $G(V,E) = G'(V',E')$
    end if
end for

算法 3-7 路径调整算法

链路，这样保证求出来的路径肯定是满足容量约束的，由于剩余链路是残余网络，所以可能会求出跳数很长的路径，如果跳数太长了，会占用太多的资源，不能达到优化的目的，所以我们对跳数进行约束，如果 $|p|/|p_d| < \delta$，则将业务加入到网络中，并更新网络链路容量，否则不加人业务到网络中。虽然这个过程是串行的，但是这个过程是很快速的，这主要有以下原因：第一，由于剩余网络中的链路容量普遍较小，能参与计算的链路很少，对一个剩余业务 d，在计算路径之前，算法会剔除那些剩余容量小于 bw_d 的链路，所以实际上参与计算的网络拓扑很小。第二，剩余业务量集合 D_l 本身较小，算法越往后面加入业务，可用链路会越来越小，网络会进一步变小。通过这个路径调整算法，可以得到原问题的一个优化可行解，这个解作为当前迭代产生的最优解，和全局最优解进行比较，如果这个解优于全局最优解则更新全局最优解。

3.4.5 仿真实验分析

3.4.5.1 仿真介绍

为了证明 LR-PROA 和 GA-PROA 的优化效果，我们把两个算法的优化结果和基于备选路径的 MILP 模型 [22] 的最优目标值进行比较，其中每个业务的备选路径数量为 30 条，我们使用 CPLEX [41] 来求解 MILP 模型，由于 CPLEX 求解 MILP 需要很大的计算量，对于大规模的网络，我们无法得到 MILP 的最优值，因此我们设置了 10 分钟的求解时间限制，当 Cplex 求解时间超过 10 分钟后，我们停止求解过程，记录求得 Cplex 求解的最优可行目标函数，以及 MILP 模型的最优值得下界。为了观察 LR-PROA 和 GA-PROA 的加速效果，我们分别设计两个算法串行版本 LR-PROA 和 GA-PROA，并把 LR-PROA 和 GA-PROA 与 LR-PROA 和 GA-PROA 进行比较，其中 LR-PROA 中的路由算法采用带堆优化的 dijkstra 算法，dijkstra 算法的复杂度为 $(|N|\lg|N|+|E|)$, 为了体现 LR-PROA 的加速效果，本文还对串行的 dijkstra 进行了进一步的优化，讨论如下：假设一批业务 D 的源节点相同，这一批业务的目的节点组成集合 D, 那么我们在求解 dijkstra 算法时，当最小堆吐出了 D 中的所有点之后，我们就提前结束了 dijkstra, 所以当集合 D 比较小时，实际的算法复杂度一般是小于 $(|N|\lg|N|+|E|)$ 的。

我们分别采用 ERdos-Renyi (ER) [39] 和 Barab asi-Albert (BA) [40] 两种模型来生成实验网络拓扑，实验中的网络拓扑中点的平均度数为 6。我们分别比较算法的目标函数，加速效果和算法的收敛性质，LR-PROA 和 LR-SROA 中的 δ 和 K 被分别设置为 10 和 30。GA-PROA 与 LR-PROA 都是通过 CUDA 8.0 进行设计，跑算法的服务器配置有四个 Intel Xeon E5-2630 CPU 和一个 NVIDIA Tesla K40M GPU。

3.4.5.2 目标函数比较

图 3-12和图 3-14分别显示了在点数为 200 和 1000 的网络中，目标函数随着业务数量规模变化的折线图，图中的网络链路容量大小为 100，业务的流量需求大小为 $[1, 100]$ 的均匀随机值。从图中我们可以看到目标函数大致随着业务数量呈现线性增加，这和目标函数的表达式相吻合。从图中可以看到 LR-SROA/LR-PORA 的算法优化目标明显优于遗传算法 LR-PROA/LR-SROA, 这是因为：1. 遗传算法提前陷入了局部最优解，导致算法提前结束。2. 遗传算法是基于备选路径的，所以其没有 LR-SPOA/LR-PROA 那么灵活，LR-SPOA/LR-PROA 会动态的寻找业务的路径，根据网络链路的动态状况重新求解路径，从而充分利用网络链路资源，从而减小目标函数。

和 MILP 的 Cplex 解相比，在小网络（点为 200）中，由于 Cplex 的计算压力

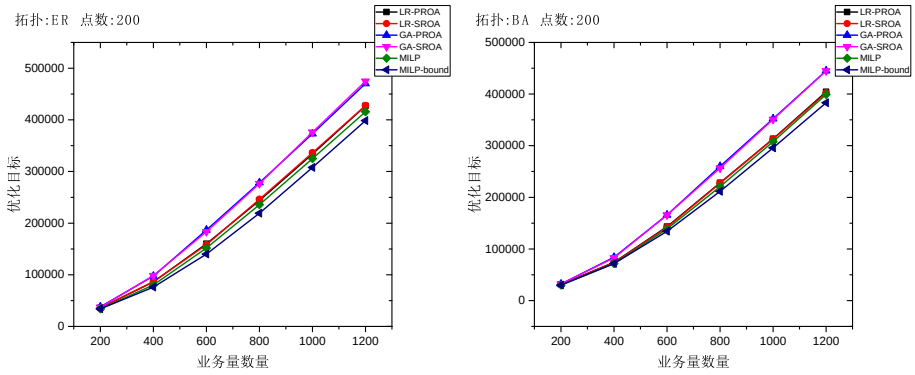

图 3-12 Object-Task(200)

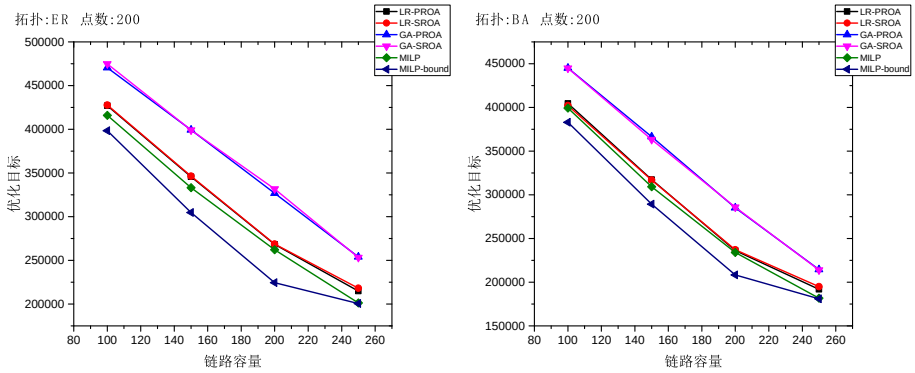

图 3-13 Object-Capacity(200)

不大，其计算出来的解略微优于 LR-SPOA/LR-PROA。可以看到图中 MILP-bound 和 LR-SPOA/LR-PROA 差距很小，这说明在小网络下，LR-SPOA/LR-PROA 算法求出的解已经很接近 MILP 的理论最优解了；在大网络中，由于 Cplex 的计算压力增大，LR-SPOA/LR-PROA 的解已经优于 Cplex 的解。比较 MILP-bound 我们可以发现 LR-SPOA/LR-PROA 的解离 MILP-bound 有一定差距，这是可能有两个原因：1.Cplex 的计算压力大，导致求得的 bound 可能不够精确。2.LR-SPOA/LR-PROA 陷入了局部最优解。图 3-13和图 3-15表示在容量变化下的目标函数值变化，其中业务数量为点数的 6 倍，网络链路容量从 100 到 250 变化。随着容量的增加目标函数大致呈现线性下降，这是因为链路容量增加后，网络资源增大，网络可容纳的业务增加，业务可选择的优化路径增增多，业务的路径代价下降，阻塞的大量减少，所以目标函数呈现大幅下降。

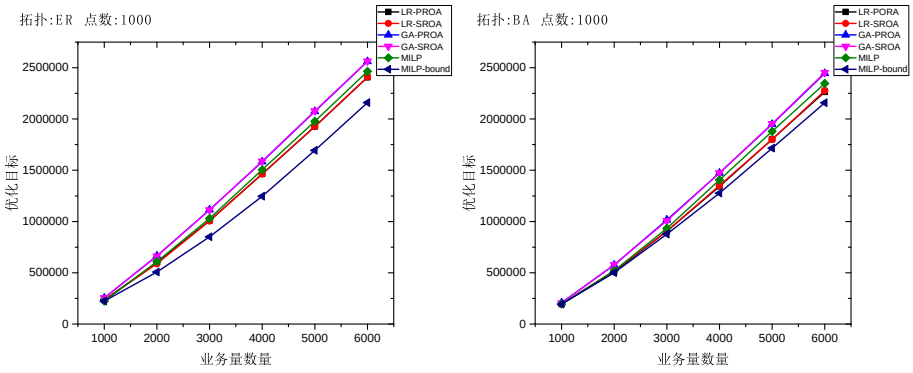

图 3-14 Object-Task(1000)

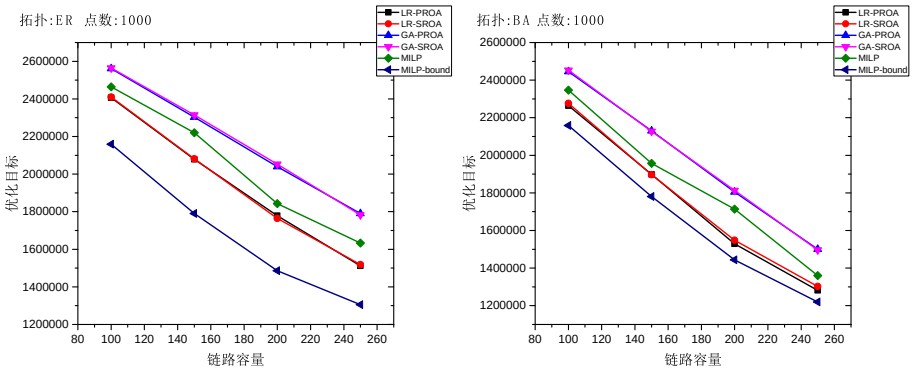

图 3-15 Object-Capacity(1000)

3.4.5.3 算法时间比较

图 3-16和图 3-17分别列出了算法在 BA 和 ER 拓扑下的计算时间随着业务数量增加的变化情况，图中拓扑节点数量为 1000，业务数量从 1000 到 6000 变化，业务流量为 1 到 100 的均匀随机值，链路的容量为 100。由于各种算法的时间差距过大，我们用三种不同的尺度进行展示。首先，CPlex 的时间限制都为 10 分钟，所以我们不再列出来。

图中我们可以看到 GA-PROA 相对于 GA-SROA 加速很多大约为 30 倍左右，但是由于遗传算法本身的缺陷，所以 GA-PROA 的计算时间是很糟糕的，这是因为基于备选路径的遗传算法的初始值染色体较差，所以需要大量的迭代次数才能收敛，而且遗传算法评价步骤的本身的计算量也很大。

观察 LR-SPOA 和 LR-PROA，发现 LR-SPOA 和 LR-PROA 的时间曲线会发生大幅度的波动，这是因为，算法的迭代次数变化较大，我们设置算法在 30 次后未找到更优解之后停止，其实 30 次设置得偏小，算法有可能会找到更好的解，但是这些解的优化程度很小（通过收敛图 3-23，可以看到算法在前 50 次内下降幅度较大，后面的下降幅度很小），所以为了优化整体时间，我们设置迭代次数为较小的 30 次，由于 30 偏小，所以可能会导致算法的提前结束，从而导致算法的迭代次数变化幅度较大，从而引起整体时间上的幅度变化。

观察图 3-16和图 3-17，我们发现当业务数量增加到两倍点数以上后，LR-PROA 相对于 LR-SROA 有较大加速，其中拓扑大小为 1000 时，LR-PROA 相对与 LR-SROA 有平均 6-7 倍的加速。

图 3-18和图 3-19表示算法时间随着网络链路容量的变化情况，其中业务数量为 6 倍网络的点数大小，在拓扑点数为 1000 的情况下，可以看到 LR-PROA 相对与 LR-SROA 有平均 6-7 倍的加速。

图 3-20和图 3-21表示算法计算时间随着网络拓扑大小的变化情况，其中加入的业务数量为网络拓扑点数目的 6 倍，可以看到随着网络拓扑变大，计算时间也相应增加，GA-PROA 的计算时间不管在在小拓扑还是大拓扑下都远远大于 LR-PROA 的计算时间。拓扑较小时 LR-PROA 相对与 LR-SROA 的加速很小，当时随着网络的拓扑规模的增加，LR-PROA 的计算时间大幅上升，而采用 GPU 上计算的 LR-PROA 上升幅度较小，使得随着在网络拓扑达到一定规模后 LR-PROA 对 LR-SROA 有较大的加速优势，加速比随着网络拓扑大小的增大从开始的 1 倍变化到最后的接近 9 倍，这充分体现出了 GPU 进行大规模计算的优势。

拓扑:ER 点数:1000

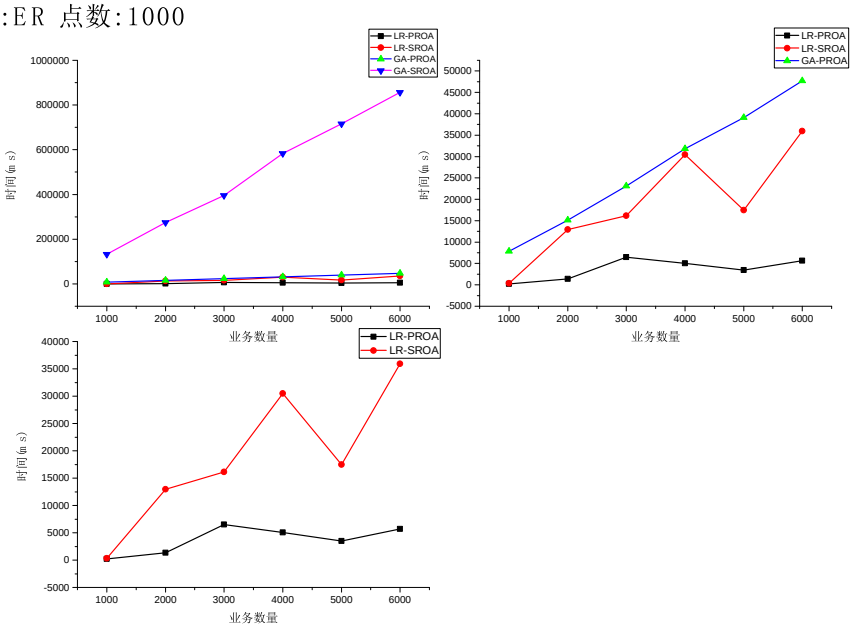

图 3-16 Time-Task(ER 1000)

拓扑:BA 点数:1000

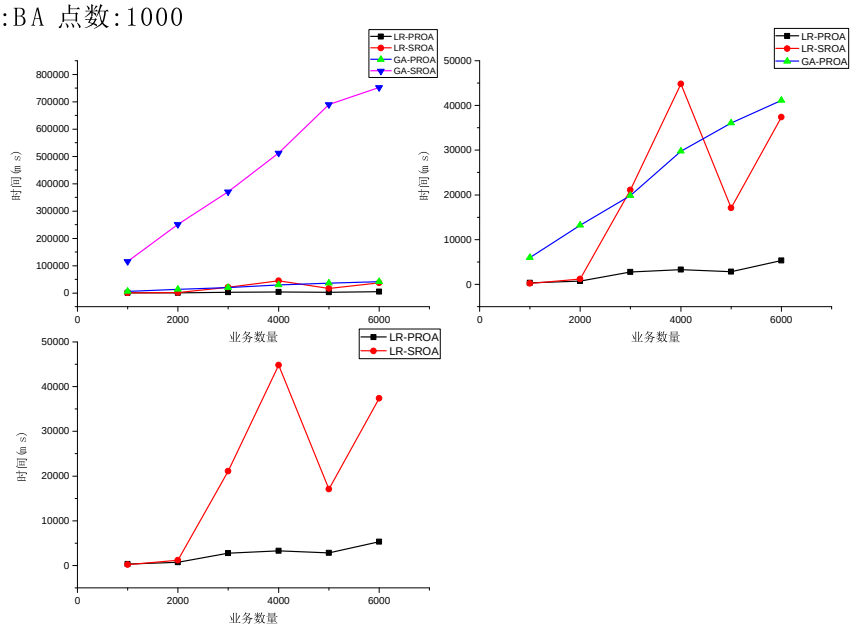

图 3-17 Time-Task(BA 1000)

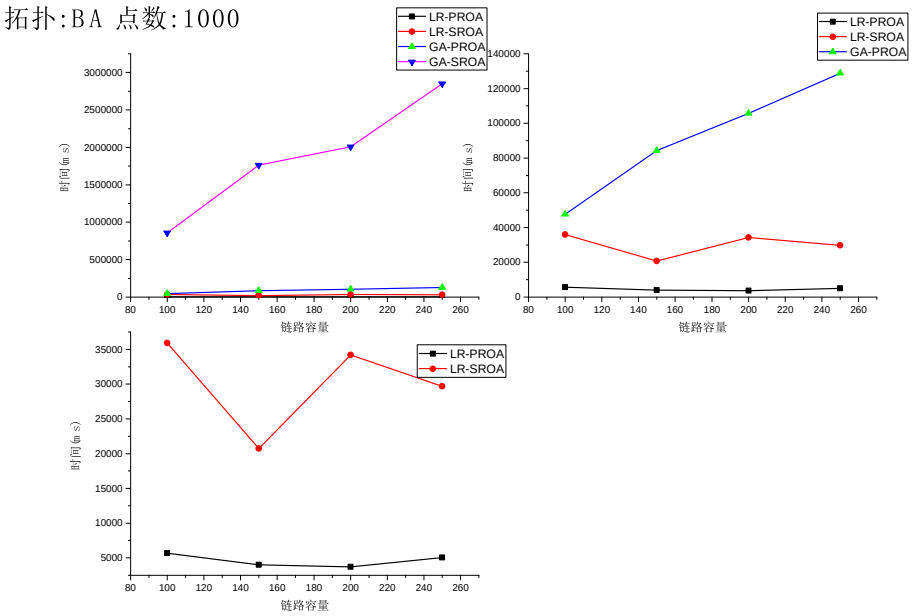

图 3-18 Time-Capacity(ER 1000)

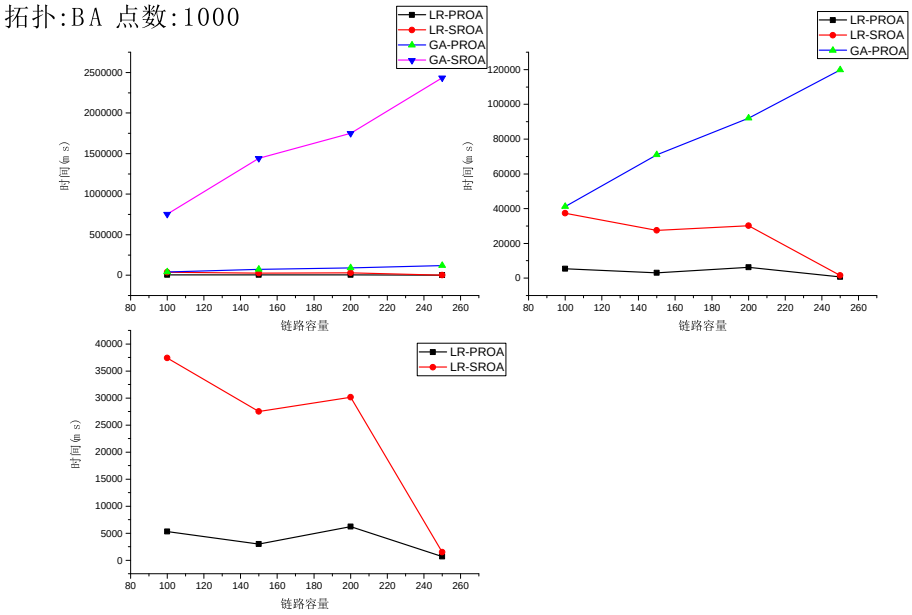

图 3-19 Time-Capacity(ER 1000)

拓扑:BA

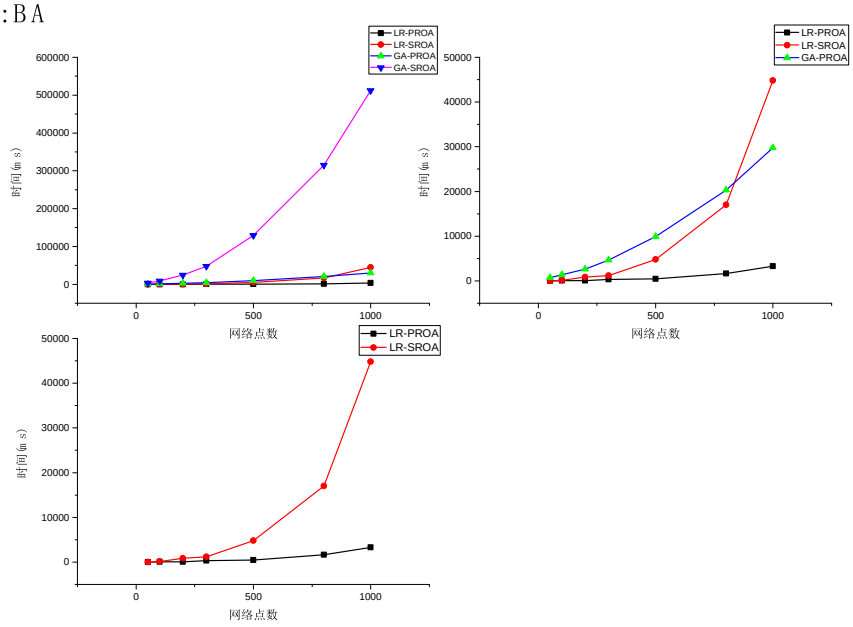

图 3-20 Time-Node(BA)

拓扑:ER

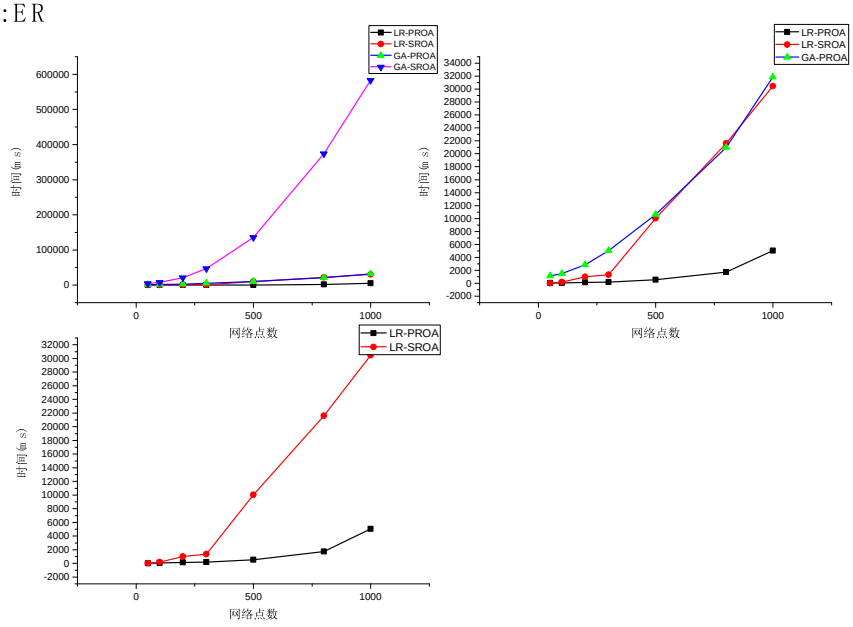

图 3-21 Time-Node(ER)

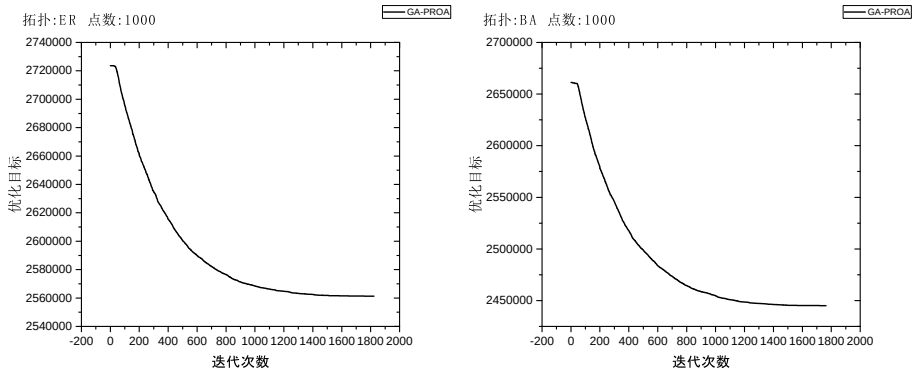

图 3-22 GA-PROA 收敛性

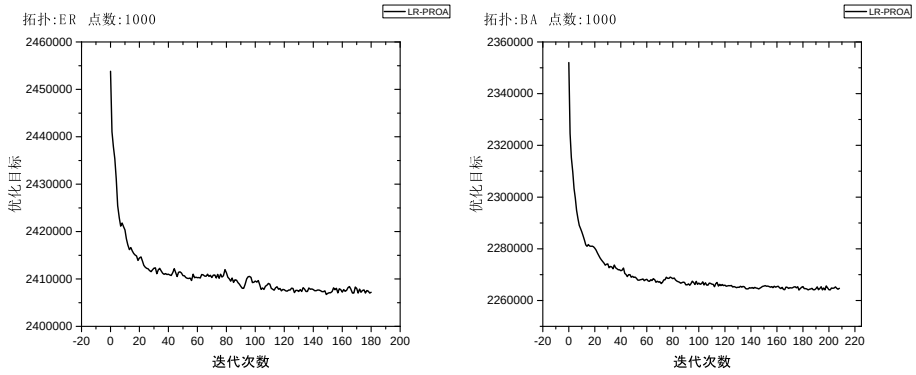

图 3-23 LR-PROA 收敛性

3.4.5.4 算法收敛性

图 3-22 和图 3-23 分别是 GA-PROA 和 LR-PROA 的收敛过程图。可以看到遗传算法的初始值较差，迭代次数较多，但是最终还是陷入了局部最优解，而 LR-PROA 的算法第一次的值已经很好，这是因为 LR-PROA 不是基于备选路径的，他会通过路径调整过程为业务重新寻找路径，从而得到一个较好的初始目标值，LR-PROA 算法的目标函数值在很短的迭代次数里快速下降，但是 LR-PROA 的曲线不够平滑，说明算法的波动很大，收敛终止条件不好控制，这也是图 3-16和图 3-17产生波动的原因。

3.5 本章总结

本章研究了 SDN 网络下的并行业务量工程算法，首先提出了新的路由优化目标函数，建立了路由优化模型，设计了两种基于 GPU 的并行优化算法方案 GA-PROA 和 LR-PROA，最后，进行实验，发现 GA-PROA 能够加速 10-20 倍，

LR-PROA 能够加速达到 6 倍，LR-PROA 能够在短时间内得到目标函数的优化解。

第四章 分层光网络下的并行路由优化算法研究

4.1 引言

在 EON 架构中，存在细粒度的带宽间隙（比如，12.5GHZ），他比 WDM 网络所遵循的 ITU 标准的 50GHZ 或者 100GHZ 的带宽粒度要小很多，而且，这些带宽间隙可以根据需要被组合在一起以提供更宽的通道。为了提高带宽利用率，在 EON 中存在混合速率，每个速率的业务需要不同数量的频谱间隙数量。理论上，EON 能够灵活地提供各种速率，但是在实际中，EON 却可能只含有很少的速率种类，这主要是因为：第一，随着频谱的增加，频谱碎片和管理复杂度将显著增加。第二，实际的 EON 从较低的线路费率升级到更高的线路费率，并存大量速率的情况已经很少见了。

为了在 EON 网络中加入业务，控制平面必须在网络中找到一条路径，同时，还需要在此路径上的链路上分配足够的频谱带宽，来创建一个合适的端到端光路连接，这被称为路由和频谱分配（RSA）问题，RSA 问题可以进一步分为静态 RSA 问题和动态 RSA 问题。静态的 RSA 问题出现在网络规划阶段，其中流量需求是已知的，这样可以离线计算出最优或者接近最优的 RSA 解，动态 RSA 问题是指在实时业务情况下光通路的路由选择和波长分配的优化问题，在动态 RSA 问题中，业务随机的到达和离开光网络，而且当业务到达网络时，控制平面需要在短时间内找到 RSA 解来安排业务。动态 RSA 问题比静态 RSA 问题更具有挑战性，因为业务需求随机到达和离开，网络状况随时发生变化，而且要求控制平面反应实时。

本章考虑在 EON 中解决动态场景下的业务量工程问题，我们采用分层图模型将频谱分配问题简化为路由选择问题，设计出基于分层图模型的频谱分配与业务量工程算法（TESAA，traffic engineering and sptrum allocate algrithm），在进行频谱分配的同时优化路由代价。我们针对算法 TESAA 进行并行加速设计，在带权图和无权图两种情况下对 TESAA 进行 GPU 并行加速，实验发现，基于 GPU 的并行的 PTESAA（parallel TESAA）算法对串行算法 STESAA(serial TESAA) 的加速比可达到 5 倍。我们将 PTESAA 与基于分层图模型的贪心算法比较，发现 PTEAA 在短时间内能够大大地优化路由代价，并且由于路由代价减小，占用网络资源较少，最终使得网络阻塞率降低。

4.2 问题描述

4.2.1 EON 中动态 RSA 问题

当业务到达网络时，SDN 控制层需要找到可行的路径，并且为路径分配合适的频谱资源。由于 EON 的物理限制，业务路由需要满足以下限制：

第一，传输距离限制：光信号的传输质量会随着传输距离的增加而下降，为了在目的点顺利恢复光信号，光链路的传输距离需要小于一个阈值 W_{max}，为了简化设计，本文假设每一条链路的长度相同，都是一个单位，那么传输路径的跳数需要小于阈值 D_{max}。

第二，频谱连续性限制：每条光连接的频谱从它的源节点到它的目的节点，在所经过的链路上保持不变。

第三，频谱不重叠限制：同一光纤链路中的频谱不能分配给不同的光路。

第四，频谱邻近限制：频谱邻近约束保证分配给一个光路径的频谱必须是一个连续的部分。

4.2.2 分层网络模型

论文 [48] 中提出一种分层图（layered Graph）模型来解决 WDM 网络中的路由和波长分配问题，分层图模型将路由选择和波长分配问题统一到一起来，使得路由和波长分配问题变得简化，这种分层图模型也可以很容易推广到 EON 的 RSA 问题中。

我们使用有向图 $N(V,E)$ 表示物理网络拓扑，其中 $V=\{v_1,v_2,...,v_N\}$ 表示节点集合，$E=\{e_1,e_2,e_3,...,e_M\}$ 表示边集合，$e_k=(i_k,j_k)$ 表示边的头节点为 v_{i_k}，边的尾节点为 v_{j_k}。假设所有光链路具有相同的频谱范围 $W=(F_{start},F_{end})$, 其中 $F_{end}-F_{start}=C$。EON 所支持的速率集合为 R, 比如，$R=40Gb/s,100Gb/s,400Gb/s$, 每一种速率 $r\in R$ 需要一个特定的频谱宽度 b_rGHz。一个源节点为 s，目的节点为 t，速率为 rGb/s 的业务被表示为 $TD(s,t,r)$。

下面我们讨论分层图的产生过程，假设第一个速率的为 r 的业务 $TD(s,t,r)$ 到达网络，我们从可用的频谱切割出一块频谱大小为 b_r 的连续带宽分配给速率为 r 的业务，假设这块频谱的起始频率为 $fs_{(r,1)}$，终止频谱为 $fe_{(r,1)}$，那么 $fe_{(r,1)}-fs_{(r,1)}=b_r$，我们把所有物理链路上频谱范围 $(fs_{(r,1)},fe_{(r,1)})$，从所有链路上切割下来，把切割下来的部分组成一个新的虚拟网络 $N^{(r,1)}(V^{(r,1)},E^{(r,1)})$，在这个新的网络中每条链路的频谱范围都是 $(fs_{(r,1)},fe_{(r,1)})$，这样切割下来之后，原来的物理网络的频谱范围更新为 $(F_{start}+b_r,F_{end})$。如果在这个复制出来的图上为业务求一条最短路径 p，那么容易知道路径 p 一定满足约束 2,3,4，这样问题被简化为单纯的路由问题。要

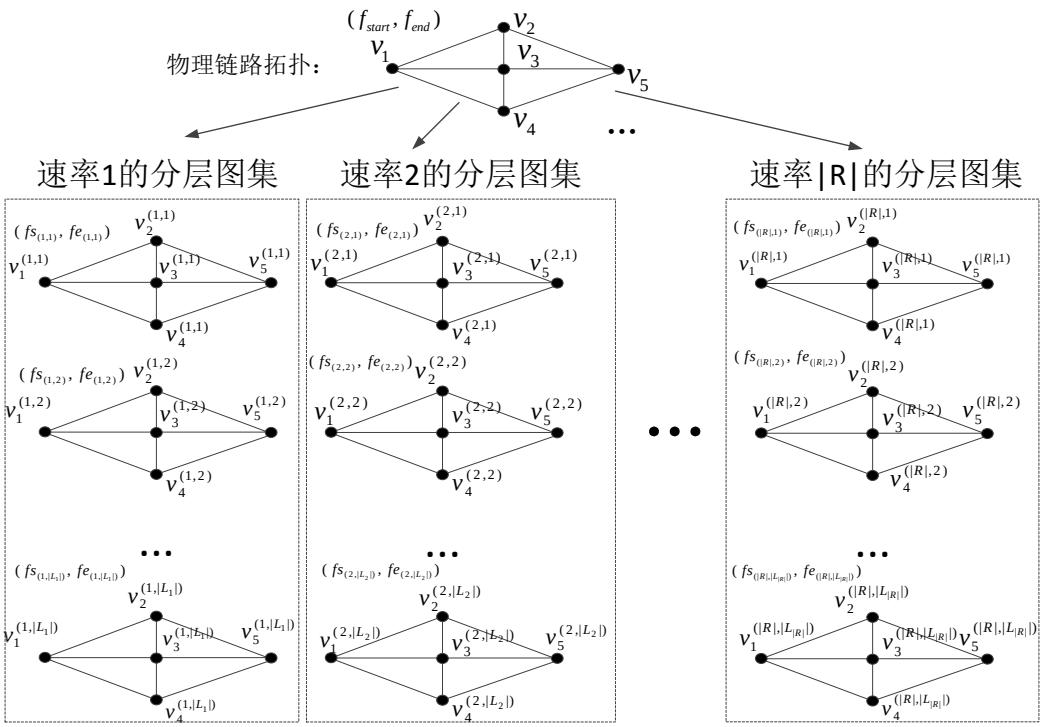

图 4-1 分层图组织结构示意

注意的是，这个虚拟图上频谱不能再分配给其他速率为 r' 的业务 $TD(s',t',r')$，即使 $b_{r'} < b_r$，因为这样会产生大量的频谱碎片，使得频谱利用率下降。所以这些分割出来给速率 r 的频谱，将继续被分配给速率为 r 的业务。当下一个速率为 r 的业务 $TD(s'',t'',r)$ 到达网络后，他首先在图 $N^{(r,1)}(V^{(r,1)},E^{(r,1)})$ 中寻找路径，如果找到合适的路径则加入网络并且更新路径链路的可用频谱为 0。反之，如果他在图 $N^{(r,1)}(V^{(r,1)},E^{(r,1)})$ 中没有找到合适的路径，那么算法会重新去切割频谱得到虚拟网络 $N^{(r,2)}(V^{(r,2)},E^{(r,2)})$，这样随着大量业务的动态到达和离去，会产生大量的虚拟图，频谱资源已经大量地分布在各个虚拟图中，我们把这些虚拟图叫做分层图（layered Graph）。

图 4-1展示了分层图的组织结构，原图被切割成一个分层图集合 L，$L_i \in L$ 表示速率 i 的分层图集合。速率 r 一共有 $|L_r|$ 个分层图，速率 r 的第 l 个分层图占用频谱 $(fs_{(r,l)},fe_{(r,l)})$。图 4-1中，点 $v_i^{(r,l)}$ 表示速率 r 的第 l 层图上的第 i 号点。

4.3 分层图模型下的业务量工程算法

根据前面的讨论，当一个速率为 r 的业务 $TD_r(s,t)$ 到达网络时，一共有 L_r 个分层平面都可以用于此业务，也就是说如果在这些图当中都能为业务找到合适的

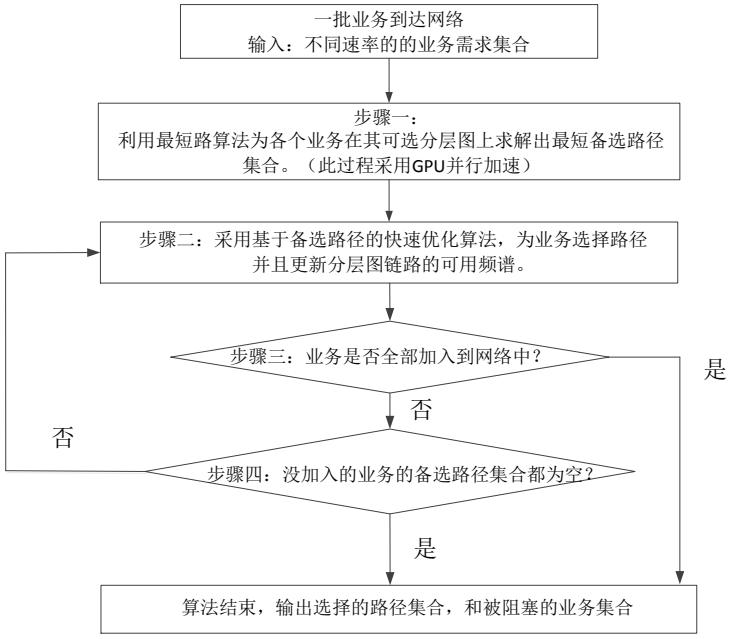

图 4-2 算法优化流程

路径的话，那么业务就有 L_r 条路径可供选择，我们可以选择代价最小的一条来优化路由。进一步，当有一批速率为 r 的业务集合 D_r 到达网络时，如果大家都选择同一层加入的话，会出现大量的冲突，而且路由代价也会很大，但是现在对每一个业务都有多个分层图平面可以选择，这就给路由代价的优化提供了可能，不同的业务可以选择不同层上的路径来进行路由，以使得总体路由代价减小，节省网络频谱资源，从而优化阻塞率。

根据以上讨论，本节提出一个基于分层图模型的频谱分配与业务量工程算法（TESAA，traffic engineering and sptrum allocate algrithm），算法在进行频谱分配的同时对业务的路由代价进行优化，算法流程如下图 4-2所示：

当一批业务到达网络时，对每一个业务在其对应速率的层上求出大量的备选路径，这个过程计算量较大，但是每一个拓扑层是独立的，所以可以采用并行算法来进行加速设计，后面 4.4 和 4.5 将讨论无权图和有权图上的基于 GPU 的并行路由算法来加速这个步骤。当备选路径都求出来了之后采用一种快速路径选择算法来把业务安排进网络，这个步骤采用一种简单的贪心算法，由于备选路径较多，实验发现这种简单的贪心算法已经可以很好的优化路径代价了。当业务的路径确定后，如果每个业务都能加入到网络，那么本轮算法结束，如果还有剩余的业务没

有被安排进网络，需要判断业务的备选路径集合是否为空，如果为空，则说明所有层都不能为业务求出路径，业务将被阻塞；反之，如果业务的备选路径集合不是空，则说明业务是因为和其他业务的路径冲突而导致不能被安排进网络的，分层图中可能还存在其他可用路径，所以需要重新在分层图中计算路径，算法回到步骤二，对这些业务进行重新计算。

步骤一的具体算法将在 4.4 和 4.5 介绍，这里介绍步骤二的快速优化算法，快速优化算法的主要思想是根据求出来的备选路径为每个业务选择路径以使得整体路径代价最小，在动态环境下需要在短时间内安排好路径，所以为了提高计算速度，这里采用一种简单但快速的贪心算法来进行解决，算法如4-1所示。

Require: $N(v, E)$: 分层图; P: 备选路径集合; D: 业务需求集合;
Ensure: AP: 加入业务的路径集合;
1: **for** $T \in D$ **do**
2: 对 P_T 中的路径按照路径的代价进行降序排序
3: **end for**
4: 对 T 按照 P_T 中的最小路径代价进行排序
5: **for** $T \in TD$ **do**
6: **for** $p \in P_T$ **do**
7: **if** 路径 p 所对应分层图上的相应链路没有被占用 **then**
8: 把 p 加入到 AP 中，并更新 p 所对应的分层网络上的链路。
9: Break
10: **end if**
11: **end for**
12: **end for**

算法 4-1 路径选择算法

算法开始对每个业务的备选路径按照其代价进行降序排序，也就是每个业务优先会选择代价较低的备选路径。然后，再对业务按照其最小代价的路径进行排序，这样保证优先考虑代价小的业务。然后开始加入业务到网络，对每一个业务，遍历其已经排序好的备选路径，如果在相应分层图上路径包含的链路都是空闲的，则加入业务到网络中，更新分层图上的相应链路为占用状态，把这个路径 p 加入到集合 AP 中。最后算法输出路径集合 AP。

4.4 无权图情况下的 GPU 算法设计

无权图情况下，路由的代价就是路径的跳数，跳数越少说明占用的网络资源越少，尽量优化路径的跳数可以节省网络资源，进一步的降低动态情况下的网络的阻塞。无权图情况下的路由算法一般使用 BFS （宽度优先搜索）算法进行求解，BFS 算法从目的节点最近的点开始，一层一层的进行扩展直到找到目的节点，每

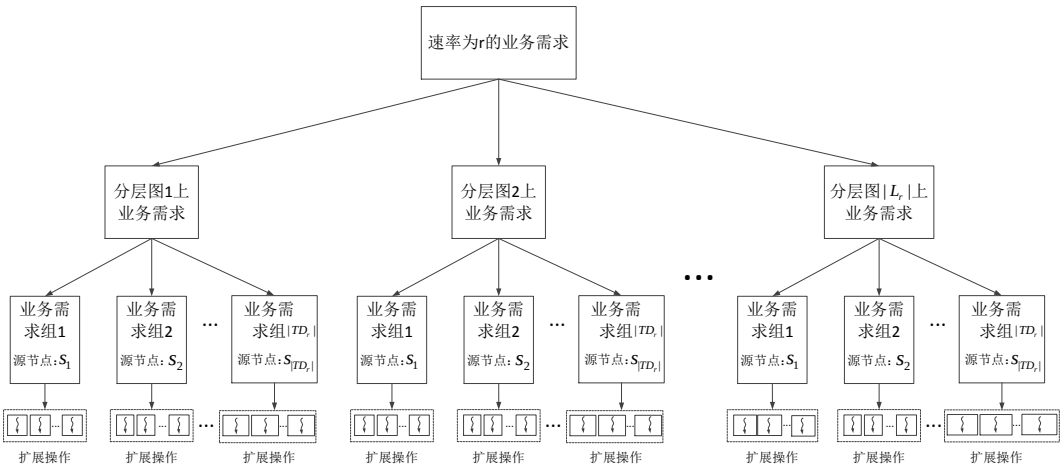

图 4-3 bfs 并行示意图

扩展一层跳数增加一跳，扩展一层需要遍历大量的边，这些边的扩展操作是相互独立的，所以为算法提供了并行设计的可能性，在加上不同速率不同层上的路由求解上的并行，总体并行粒度是很大的，下面分别对不同的并行层次的设计进行讨论。

4.4.1 相同速率业务的并行

如下图4-3所示，对速率都为 r 的一批业务，在其可用的每个分层图上并行进行计算，在每个分层图上，具有相同源节点的业务组成业务组，因为他们的路径可以一起计算，对每一个业务组，在 GPU 上开辟 $|E|$ 个线程对每一条边进行扩展操作，那么整个过程的的并行粒度为 $|L_r||D_r||E|$，其中 $|D_r|$ 为速率为 r 的业务组数量。

4.4.2 不同速率间业务的并行

由于动态业务情况下每次到达网络的业务数和速率情况都是变化的，不同速率之间业务的并行不容易直接实现在一个 kernel 中进行，我们采用 GPU 提供的流并行进行并行设计。如下图 4-4所示：我们为每一个速率建立一个 stream 来负责这个速率的业务计算，假设业务一个有 $|R|$ 个不同速率，那么需要建立 $|R|$ 个 stream，多个 kernel 同时执行，充分利用 GPU 的 SMIT 资源，当 SIMT 资源足够时，每个 kernel 将并行执行，当 SIMIT 不足时，不同的 kernel 之间可以进行快速的切换，以达到隐藏延迟的效果。

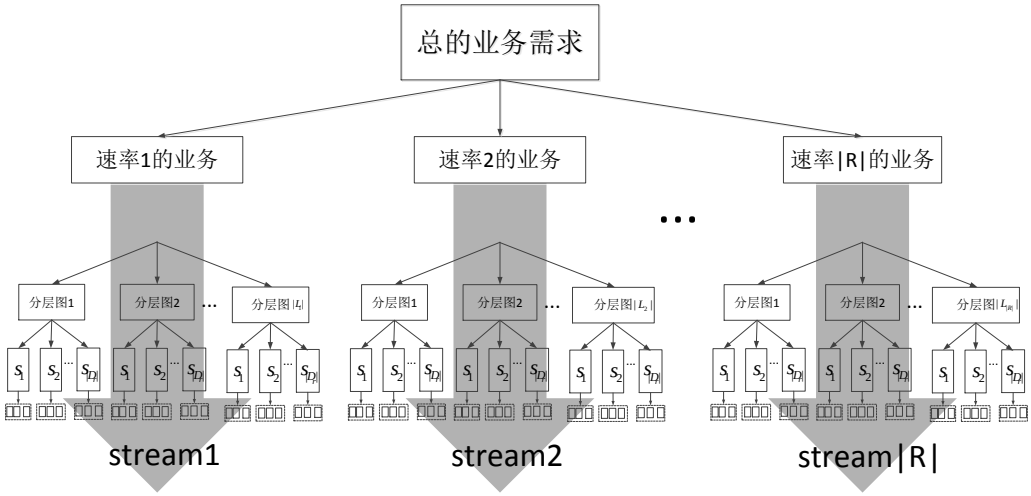

图 4-4 bfs 流并行示意图

4.4.3 GPU 上 kernel 设计

和 3.4.2 中讨论不一样，在 BFS 的并行中不会存在前驱节点更新的同步问题，但是，我们依然需要两个 kernel 函数，这是因为扩展 kernel 需要执行多次，如果扩展和更新同时执行，那么更新也会执行多次，这会增多内存访问，所以我们把两个操作分成两个 kernel，一个 *kernel_BFS_extend* 进行边的扩展操作，一个 *kernel_BFS_update* 进行前驱节点的更新操作，其中 *kernel_BFS_extend* 需要多次调用，但是 *kernel_BFS_update* 只需要在最后执行一次。在 *kernel_BFS_extend* 中，输入 E 是所有的分层图的边所组成的集合，*Dist* 是预先分配的距离标记数组，他是一个三维数组，第一维表示当前距离数组对应的分层图标号，第二维度表示源节点，第三维度表示目的节点，比如 $Dist[3][10][5] = 4$ 表示在第 3 个分层图上，点 10 到 3 的距离为 4。初始时，除了源节点距离被初始化为 0 之外，其他的距离都被初始化为无穷大。*rid* 表示当前 kernel 负责计算的业务速率标号，他是区分不同流的标记，*rid* 不同表示执行 kernel 的流不同。*round* 表示当前扩展的层数，由于 BFS 是一层一层进行扩展操作，我们需要记录当前层数来决定哪些边需要扩展。算法开始时先进行一系列映射操作，将线程映射到边标号 *eid*，当前所在分层图标号 *lid*，以及源节点标号 *sid*。由于动态情况下，边的可用状态可能发生变化，找到边之后，需要判断这条边是否可用，如果不可用则返回。接着判断当前边是否可以进行扩展操作，如果尾节点已经更新过了（$Dist[lid][sid][e.head] <= round$），则不能再更新了，反之，还需要判断头节点是否是上一次扩展层的节点（$Dist[lid][sid][e.tail] + 1 == round$），如果是的话，那么就更新尾节点的距离为当前扩展层 *round*。在 *kernel_BFS_update*

```
1: function KERNEL_BFS_EXTEND(E, Dist,rid,round)
2:     bid ← block ID
3:     tid ← thread ID
4:     用 (bid, tid, rid) 映射到边的标号 eid
5:     用 (bid, tid, rid) 映射到分层图标号 lid
6:     e ← E[gid][eid]
7:     if e.avaliable == −1 then
8:         return
9:     end if
10:    用 (bid, tid, rid) 映射到的源点标号 sid
11:    if Dist[lid][sid][e.head] > round&&Dist[lid][sid][e.tail] + 1 == round then
12:        Dist[gid][sid][e.head] ← round
13:    end if
14: end function
```

算法 4-2 kernel_BFS_extend

中，其主要逻辑和 *kerne$_B$FS_extend* 一样，在进行更新判断的时候，判断当前边的头节点是否是尾节点的上一层（Dist[lid][sid][e.head]==Dist[lid][sid][e.tail]+1）如果是的话，那么头节点就有资格作为尾节点的前驱节点，于是进行前驱更新操作。介绍完了两个 kernel 之后，算法 4-8为整个并行 BFS 的算法流程： 算法 4-8开始时先将业务按照速率的不同进行划分，再将业务按照源节点的不同划分成不同的业务组集合 D，比如，$D_r \in D$ 表示速率为 r 的业务组集合，$d_{rs} \in D_r$ 表示速率为 r 的源节点标号为 s 的业务组。划分好业务组之后，就开始初始化距离数组，将速率对应的所有分层图上的源节点距离初始化为 0，这是一个三层的 for 循环，第一层遍历速率标号，第二层遍历源节点标号，第三层遍历速率所对应的分层图标号。初始化 *Dist* 数组后，在发射 kernel 之前需要先新建 $|R|$ 个 GPU 流，不同的流将对不同速率的业务进行计算。*round* 为扩展的层数标记，开始时 *round* 初始化为 1。while 循环中进行扩展操作，其中 W_{max} 为跳数限制，最大跳数不能超过 W_{max}, 每一次扩展操作只会使得路径长度增加一跳，所以最多只能循环 W_{max} 次。while 循环中 for 循环用来发射不同的流，因为发射不同流的 kernel 是异步操作，for 循环不会去等待上一个 kernel 结束了才去执行下一个 kernel，所以可以认为所有 kernel 都是同时间发射的。while 循环结束后，算法发射 *kernel_BFS_update* 进行前驱节点记录操作，最后根据这些前驱信息，算法重新恢复出路径，组成路径集合 P，算法结束。

4.5 带权图情况下的 GPU 算法设计

在实际应用场景中，我们常常需要将不同的链路设置为不同的代价，因为链路的延迟，租用费用可能不同，所以有必要考虑链路带权情况下的优化设计，算

```
1: function KERNEL_BFS_UPDATE(E, Pre,rid)
2:     bid ← block ID
3:     tid ← thread ID
4:     用 (bid, tid, rid) 映射到边的标号 eid
5:     用 (bid, tid, rid) 映射到分层图标号 lid
6:     e ← E[gid][eid]
7:     if e.avaliable == −1 then
8:         return
9:     end if
10:    用 (bid, tid, rid) 映射到的源点标号 sid
11:    if Dist[lid][sid][e.head] == Dist[lid][sid][e.tail] + 1 then
12:        Pre[gid][sid][e.head] ← e.tail
13:    end if
14: end function
```

算法 4-3 kernel_BFS_update

法优化的目标是业务使用的链路总体代价最小化，为了简化问题模型，本节只讨论链路代价为正值的情况。本节将逐步分析和设计适用于带权图的分层网络 GPU 并行算法。

4.5.1 带跳数限制的最短路算法

带跳数约束的 BFS 路由，由于没有考虑链路的权重差异，把所有链路的权重看作一样，所以 FS 路由的优化目标就是跳数，BFS 就是寻找跳数最短的路径，跳数约束只是使得 BFS 算法提前结束。但是在带权的链路情况下，路由的优化目标是最小化链路代价和，也就是寻找一条代价最小的满足跳数约束的路径，跳数越短并不意味则代价越小，代价越小也不意味着跳数越小，这使得带权情况下比无权情况更加复杂。

为了说明带权带跳数约束下的路由算法，我们介绍一些符号，在图 $N(V,E)$ 中，有点 $i \in V$, 和点 $j \in V$, 假设集合 P_{ijk} 表示点 i 到点 j 的所有跳数为 k 的路径所组成的集合，设 p^*_{ijk} 为集合 P_{ijk} 当中代价最小的那一条路径，即 $p^*_{ijk} = \arg\min_{p \in P_{ijk}} \{Price\_Of(p)\}$，我们设最小代价数组为 $Price$，$Price$ 为三维数组，其中 $Price[i][j][k] = Price\_Of(p^*_{ijk})$，即 $Price[i][j][k]$ 表示为 p^*_{ijk} 的代价，也就是说 $Price[i][j][k]$ 表示 i 到 j 的跳数为 k 跳的最小代价路径的代价。对点 i，边集合 $PreE_i$ 表示点 i 的入边组成的集合，点集合 $PreN_i$ 表示点 i 的入节点所组成的集合。

有了上面的符号介绍后，假设业务的源节点为 s，下面给出一个动态规划递推

Require: 业务需求集合 TD; 分层图链路集合 E;
Ensure: 业务需求的最短路径集合 P
1: 将业务根据速率和源节点重新组合成业务分组集合 D
2: **for** $D_r \in D$ **do**
3: **for** $d_{rs} \in D_r$ **do**
4: **for** $l \in L_r$ **do**
5: **for** $v \in V$ **do**
6: **if** $v == s$ **then**
7: $Dist[l][s][v] \leftarrow 0$
8: **else**
9: $Dist[l][s][v] \leftarrow \infty$
10: **end if**
11: $Pre[l][s][v] \leftarrow -1$
12: **end for**
13: **end for**
14: **end for**
15: **end for**
16: 新建 $|R|$ 个流，组成集合 S
17: $round \leftarrow 1$
18: **while** $round <= W_{max}$ **do**
19: **for** r in R **do**
20: 在流 S_r 上发射 kernel_bfs_extend(E, $Dist$,r,$round$)
21: **end for**
22: round=round+1
23: **end while**
24: **for** r in R **do**
25: 在流 S_r 上发射 kernel_bfs_update(E, $Dist$,r,$round$)
26: **end for**
27: 根据前驱数组 Pre 重建路径，然后把路径加入到集合 P

算法 4-4 并行 BFS 计算

式，其中 w_{nj} 表示边 (n,j) 的代价大小：

$$Price[i][j][k] = \begin{cases} \min\limits_{n \in PreN_i} Price[i][n][k-1] + w_{nj} & \text{if } k > 0 \\ 0 & \text{if } k = 0 \text{ and } i = s \\ \infty & otherwise \end{cases} \tag{4-1}$$

设 p_{ijk}^{*} 为点 i 到 j 的跳数为 k 的最小代价路径，由于点 j 的前驱节点只可能是那些属于集合 $PreN_j$ 的点，点 j 的前驱边只可能是集合 $PreE_j$ 中的某一条边，那么最优路径 p_{ijk}^{*} 中的第 k 条边一定属于集合 $PreE_j$，于是递推式中我们遍历了这些边，另外，设经过边 $e \in PreE_j$ 到达 j 的跳数为 k 的最优路径为 p_{ijk}^{e}，那么根

据 Price 的定义我们可以得到 $Price\_Of(pijk^e) = Price[i][e.head][k-1] + w_e$，其中，$e.head$ 表示边的头结点，也就是点 i 的对应于边 e 的前驱节点。那么最优路径 $p^*_{ijk} = \arg\min_{e \in PreE} Price_Qf(p_e)$，相应的最优代价 $Price[i][j][k] = \min_{e \in PreE} Price_Qf(p_e)$，也就是 $\min_{n \in PreN_i} Price[i][n][k-1] + w_{nj}$。注意边界情况下 $(k == 0)$，由于除了源节点自己之外，源节点到其他节点的跳数不可能是零，所以开始时需要设置代价为无穷大，而源节点到他自己距离设置为 0。

上面的动态规划递推式可以求得一个点到任意点的跳数为 1 到 k 的最优路径，而我们是要求得在跳数限制下的最优代价路径，我们需要对求得的动态规划解进行处理，设二维数组 $OPCost$ 表示跳数限制下的最优路由代价值，那么跳数限制约束下的点 i 到点 j 的最小代价路径的代价为 $OPCost[i][j]$，可以得到下面的表达式：

$$OPCost[i][j] = \min_{k \in W_{max}} Price[i][j][k] \tag{4-2}$$

假设满足跳限约束的最优代价路径为 p^*_{ij}，假设路径 p^*_{ij} 的跳数为 h，$h \in [1, W_{max}]$，显然 $p^*_{ij} = p_{ijh}$，但是实际上我们并不知道 h 的值，所以我们从 $[1, W_{max}]$ 对 h 进行遍历，那么 p^*_{ij} 肯定是其中代价最小的一条，所以 $p^*_{ij} = \min_{k \in W_{max}} Price_Qf(p^*_{ijk})$，同样，$OPCost[i][j] = \min_{k \in W_{max}} Price[i][j][k]$。

观察动态规划递推式，可以看到，在递推过程不会去记录路径，这样求得路径可能会经过重复的点，但是，最优路径 p^*_{ij} 中是不可能经过重复节点的，下面本文进行证明。首先，路径跳数 $h < 4$ 的路径是不可能经过重复的节点，我们讨论 $h >= 4$ 时的情况，如果路径重复经过了一个点，那么路径中一定存在一个环，如下图 4-7，这个存在环的路径为 $L_1 - L_2 - L_3 - L_2 - L_4$，假设链路 L_2 的跳数为 h_2，代价为 C_2，由于链路的代价都是正数，所以 $C_2 > 0$，那么存在一条跳数为 $t = h - h_2$ 的，代价为 $OPCost[i][j] - C_2$ 的由 i 到 j 的路径 $L_1 - L_2 - L_4$，而这条路径的代价肯定小于等于 $Price\_Of(p_{ijt})$，即 $OPCost[i][j] - C_2 <= Price\_Of(p_{ijt}) <= OPCost[i][j]$，进一步得到 $OPCost[i][j] - C_2 <= OPCost[i][j]$，这与 $C_2 > 0$ 矛盾，所以最优路径不可能经过重复的节点。

4.5.2 并行层次

4.5.3 并行动态规划算法

上一节提出了动态规划模型，证明了算法的正确性，本节我们讨论这个动态规划算法在 GPU 上的并行设计框架，以及分层图和不同速率的并行层次设计。单个图上的动态规划的串行执行过程如下伪代码所示：

算法循环 W_{max} 次，每一次循环中对每一个点进行对应的 $Price$ 数组进行更新，

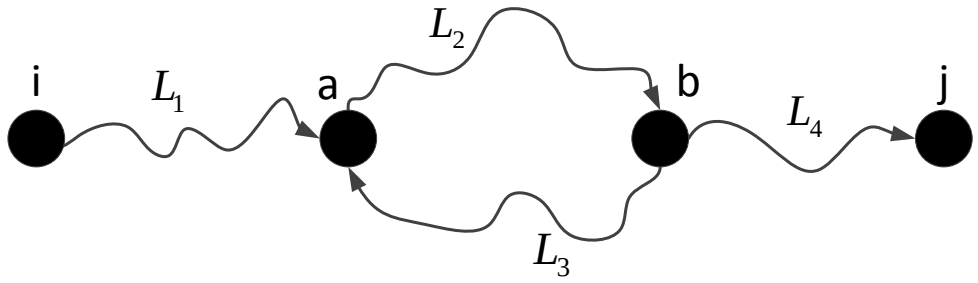

图 4-5 无环证明

Require: 点的入边集合 *PreE*; 点的入点集合 *PreN*; 源节点 *s*;
Ensure: 最优代价数组 *Price*; 最优前驱节点数组 *Pre*;

1: **for** $k \in [1, W_{max}]$ **do**
2: **for** $v \in V$ **do**
3: Price[s][v][k]= $\min_{n \in PreN_v} Price[s][n][k-1] + w_{nv}$
4: Pre[i][j][k]=arg $\min_{n \in PreN_v} Price[s][n][k-1] + w_{nv}$
5: **end for**
6: **end for**

算法 4-5 串行动态规划算法

而这些更新操作是相互独立的，所以，可以进行并行设计，对每一个点的 *Price* 数组更新操作开辟一个线程进行计算，这个线程遍历当前点的前驱边，寻找最优的那一条前驱边，这里的并行粒度只有图的点数 $|N|$，但是考虑到不同个业务和不同的分层图上计算也是并行的，总的并行粒度为 $|N||D||R|$，这样需要开辟大量的线程，充分利用 GPU 上的 SIMT 计算资源。

下面考虑单个速率情况下的并行框架，如下图 4-6所示，动态规划算法和 BFS 单个速率情况下的并行类似，只是 BFS 那里的对每一条边并行的扩展操作被替换成了对每一个点并行的 *Price* 数组更新操作。

和 BFS 一样，对不同速率的业务，本文依然采用不同的流进行并行，其并行过程如下所示：

4.5.4 GPU 上 kernel 设计

动态规划的 GPU 代码设计如下： 在 *kernel_dynamic_update* 中，输入 *PE* 是一个二维的集合数组，比如，*PE*[2][3] 表示第 2 个分层图上点 3 的入边集合。*Price* 是

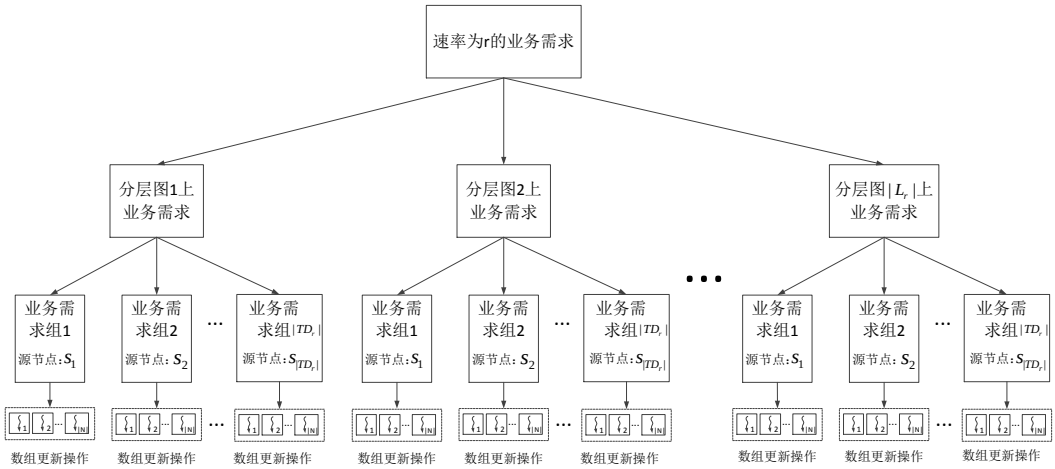

图 4-6 动态规划并行示意图

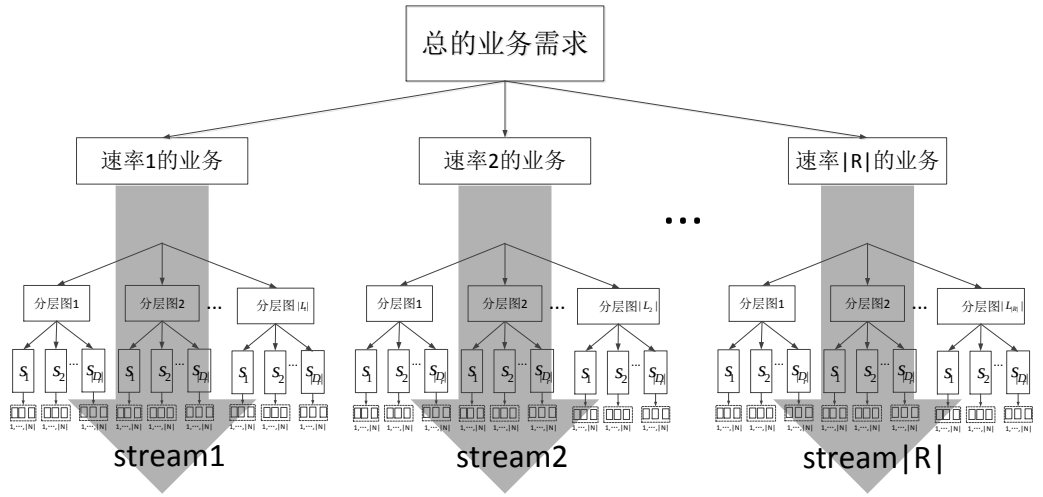

图 4-7 动态规划流并行示意图

```
1: function KERNEL_DYNAMIC_UPDATE(PE,rid,k)
2:     bid ← block ID
3:     tid ← thread ID
4:     用 (bid, tid, rid) 映射到边点的标号 nid
5:     用 (bid, tid, rid) 映射到边源节点的标号 sid
6:     用 (bid, tid, rid) 映射到分层图标号 lid
7:     for e ∈ PE[lid][nid] do
8:         if e.avalible > 0 then
9:             if Price[lid][sid][nid][k] < Price[lid][sid][e.s][k − 1] + e.weight then
10:                Price[lid][sid][nid][k] = Price[lid][sid][e.s][k − 1] + e.weight
11:                Pre[lid][sid][nid][k] = e.s
12:            end if
13:        end if
14:    end for
15: end function
```

算法 4-6 kernel_dynamic_update

预先分配的代价数组，他是一个三维数组，第一维表示当前代价数组对应的分层图标号，第二维度表示源节点，第三维度表示目的节点，第四维度表示路径的跳数，比如 *Price*[4][3][10][5]=7 表示在第 4 个分层图上，点 3 到 10 的跳数为 5 的最优路径代价为 7，初始时，当跳数等于 0 时，除了源节点代价被初始化为 0 之外，其他的距离都被初始化为无穷大。*Pre* 数组是前驱数组用以记录前驱信息来恢复路径，*Pre* 数组和 *Price* 数组一样，是一个四维数组，*Pre*[4][3][10][5]=50 表示在第 4 个分层图上，点 3 到 10 的跳数为 5 的最优路径的前驱节点 (也就是路径上的倒数第二个点) 为点 50。*rid* 表示当前 kernel 负责的计算的业务速率标号，他是区分不同流的标记，*rid* 不同表示执行 kernel 的流不同。*k* 表示当前扩展的层数，也就是跳数。算法开始时先进行一系列映射操作，将线程映射到点标号 *eid*，当前所在分层图标号 *lid*，以及源节点标号 *sid*，映射完成后就可以进行 Price 数组的更新了，也就是要去寻找最优的入边，算法遍历点的所有入边，由于动态情况下，边的可用状态可能发生变化，找到边之后，需要判断这条边是否可用，如果不可用则跳过这条边，反之，就判断更新条件是否成立 ($Price[lid][sid][nid][k] < Price[lid][sid][e.s][k-1] + e.weight$)，如果条件成立则说明这条入边比之前的要好，经过这条边到达当前点的路径代价更小，所以要更新 Price 数组，同时为了后面恢复路径需要记录前驱信息在数组 *Pre* 中。

算法开始时先将业务按照速率的不同进行划分，再将业务按照源节点的不同划分成不同的业务组集合 D，比如，$D_r \in D$ 表示速率为 r 的业务组集合，$d_{rs} \in D_r$ 表示速率为 r 的源节点标号为 s 的业务组。划分好业务组之后，就开始初始化代价

Require: 业务需求集合 TD; 各个分层图前驱链路集合 PE;
Ensure: 业务需求的最短路径集合 P
1: 将业务根据速率和源节点重新组合成业务分组集合 D
2: **for** $D_r \in D$ **do**
3: **for** $d_{rs} \in D_r$ **do**
4: **for** $l \in L_r$ **do**
5: **for** $v \in V$ **do**
6: **if** $v == s$ **then**
7: $Price[l][s][v][0] \leftarrow 0$
8: **else**
9: $Price[l][s][v][0] \leftarrow \infty$
10: **end if**
11: $Pre[l][s][v][0] \leftarrow -1$
12: **end for**
13: **end for**
14: **end for**
15: **end for**
16: 新建 $|R|$ 个流，组成集合 S
17: $k \leftarrow 1$
18: **while** $k <= W_{max}$ **do**
19: **for** r in R **do**
20: 在流 S_r 上发射 kernel_dynamic_update(PE,rid,k)
21: **end for**
22: k=k+1
23: **end while**
24: 根据前驱数组 Pre 重建路径，然后把路径加入到集合 P

算法 4-7 并行动态规划的计算

数组，将速率对应的所有分层图上的跳数为 0 的源节点代价初始化为 0，这是一个三层的 for 循环，第一层遍历速率标号，第二层遍历源节点标号，第三层遍历速率所对应的分层图标号。初始化 $Price$ 数组后，在发射 kernel 之前需要先新建 $|R|$ 个 GPU 流，不同的流将对不同速率的业务进行计算。k 为扩展的跳数标记，开始时 k 初始化为 1。while 循环中进行扩展操作，其中 W_{max} 为跳数限制，最大跳数不能超过 W_{max}，所以最多只能循环 W_{max} 次。while 循环中 for 循环用来发射不同的流，因为发射不同流的 kernel 是异步操作，for 循环不会去等待上一个 kernel 结束了才去执行下一个 kernel，所以可以认为所有 kernel 都是同时间发射的。

4.6 实验仿真分析

实验仿真中，本文假设存在两种速率的业务，并且已知两种速率的业务的大致比例为 1:3。我们初始时为速率 1 的业务分配 20 层的频谱连续分层网络，为速

Require: 业务需求集合 *TD*; 分层图集合 *G*;
Ensure: 业务需求的路径集合 *P*; 阻塞的需求集合 *Z*

```
for r ∈ R do
    flag ← 0
    for TD_r ∈ TD do
        把 TD_r 中的业务按照其在第一层分层图上的路径跳数进行排序
        for d ∈ TD_r do
            for g ∈ G_r do
                if 在分层图 g 上存在业务 d 的合法路径 p then
                    更新分层图 g 上的链路占用情况
                    把路径 p 加入到结果路径集合 P 中
                    flag ← 1
                    break
                end if
            end for
            if flag = 0 then
                把业务 d 加入阻塞集合 Z 中
            end if
        end for
    end for
end for
```

算法 4-8 贪心的分层 RSA 算法

率 2 的业务分配 60 层的频谱连续的分层网络。我们假设对业务的加入服从泊松过程，一共产生了 1000 个加入事件，每个加入事件间的时间间隔服从均值为 4 的负指数分布，当加入事件到达时，产生速率 1 的业务为 d_1 个，产生速率为 2 的业务 d_2 个，其中 $d1$ 为 2|N| 到 4|N| 中的随机值，d_2 为 6|N| 到 12|N| 之间的随机值。初始时网络的节点为 *N*，平均度数为 4。每个业务的服务时间满足均值为 *ST* 的负指数分布。我们在不同 *ST* 下进行仿真，观察算法在不同服务压力下的权值变化，跳数变化和阻塞率变化情况。

4.6.1 对比算法

为了说明 PRORSA 对路径代价和网络阻塞率带来的优化效果，我们将 PRORSA/SRORSA 和不进行路径优化的贪心算法进行比较，我们称这个算法为 GRSA(Greedy)，算法的执行伪代码如下4-8所示：

当业务到达时，GRSA 先在第一层对所有业务求最短路径，得到一个路径跳数值，按照这个跳数值对业务进行排序，这是为了优先加入较短的业务。然后逐层去寻找是否能够把业务加入到网络中，如果能加入到当前层中，那么就更新当前层上的相应链路的占用情况，反之，就在下一层进行搜寻。如果所有层都无法

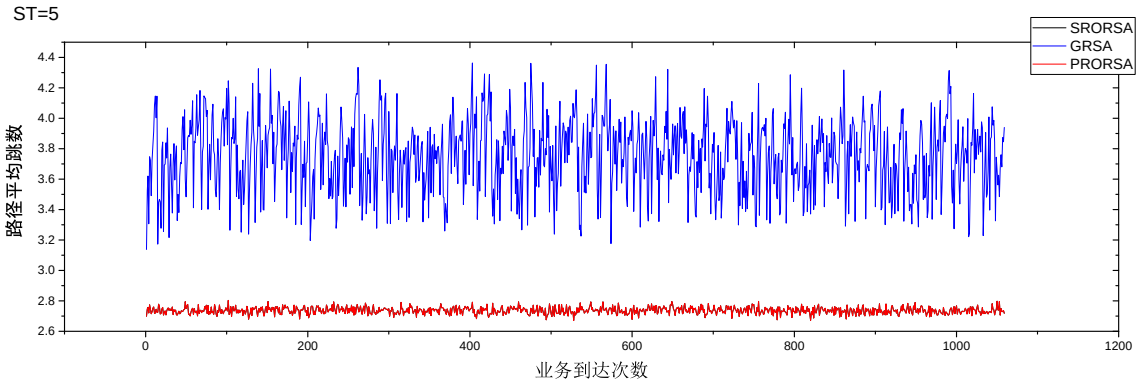

图 4-8 无权图路径跳数对比（ST=5）

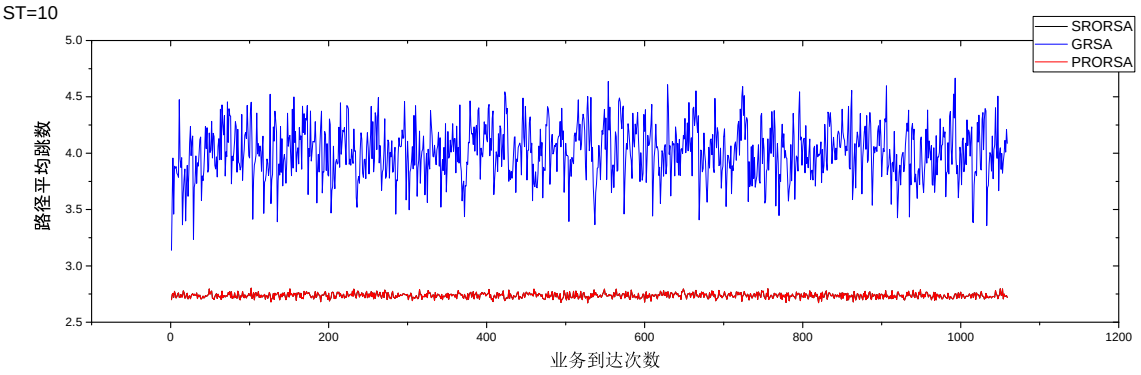

图 4-9 无权图路径跳数对比（ST=10）

加入这个业务，那么业务就被阻塞，加入到阻塞集合。

4.6.2 无权图下的仿真结果

4.6.2.1 路径优化分析

在无权图下，优化目标是最小化路径的跳数，图4-8到图4-12表示了无权图下的跳数优化结果，我们把 PRORSA 和 SRORSA 的优化结果与贪心算法 GRSA 的算法权值优化结果进行比较，图中的横坐标表示业务到达的次数，我们一共产生了 1000 个加入事件。当网络状况较好的时候，比如，平均服务时间为 $ST = 5$ 时，PRORSA/SRORSA 的平均跳数为 2.73，而 GRSA 的平均跳数为 3.73，平均跳数要少一跳左右，这样 PRORSA/SRORSA 可以大大的减小对网络资源的占用。随着平均服务时间 ST 的增加，PRORSA/SRORSA 和 GRORSA 的平均跳数都有所增加，但是 PRORSA/SRORSA 的增幅较小，从 $ST = 5$ 时的平均跳数为 2.73 增加到 $ST = 40$ 时的平均跳数为 2.82，也就是说不管在网络空闲还是繁忙的情况下，PRORSA/SRORSA 都能够很好得优化跳数。而 GRSA 的跳数随着 ST 的增加而大幅增加从 $ST = 5$ 时的平均 3.75 增加到 $ST = 40$ 时的平均 4.2。

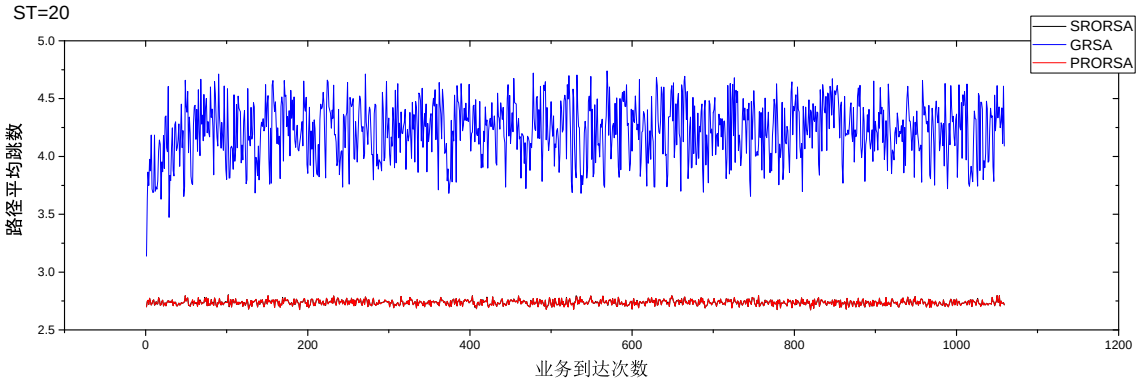

图 4-10 无权图路径跳数对比（ST=20）

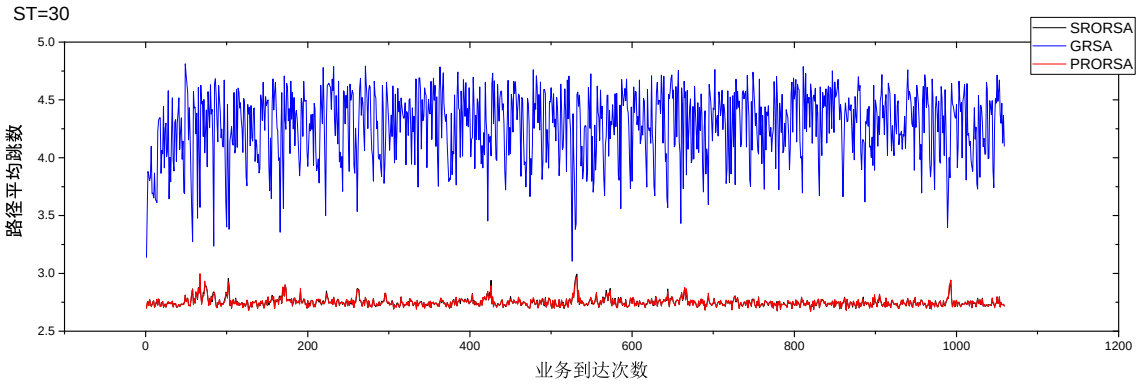

图 4-11 无权图路径跳数对比（ST=30）

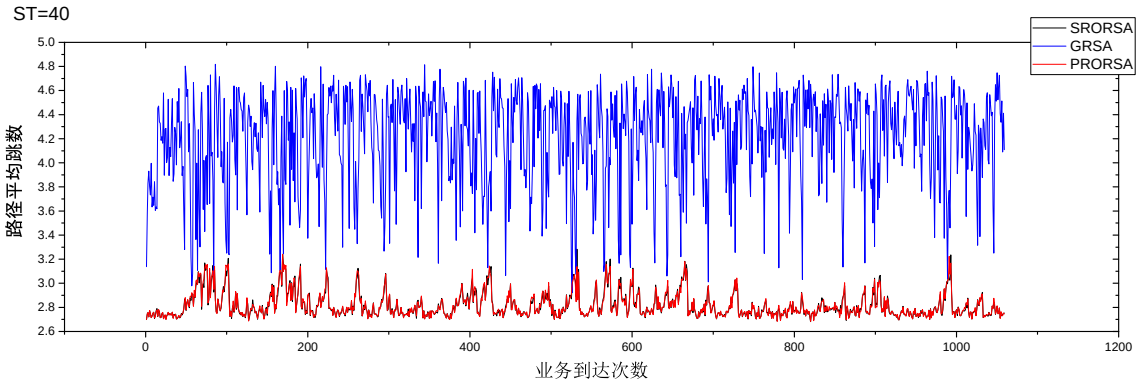

图 4-12 无权图路径跳数对比（ST=40）

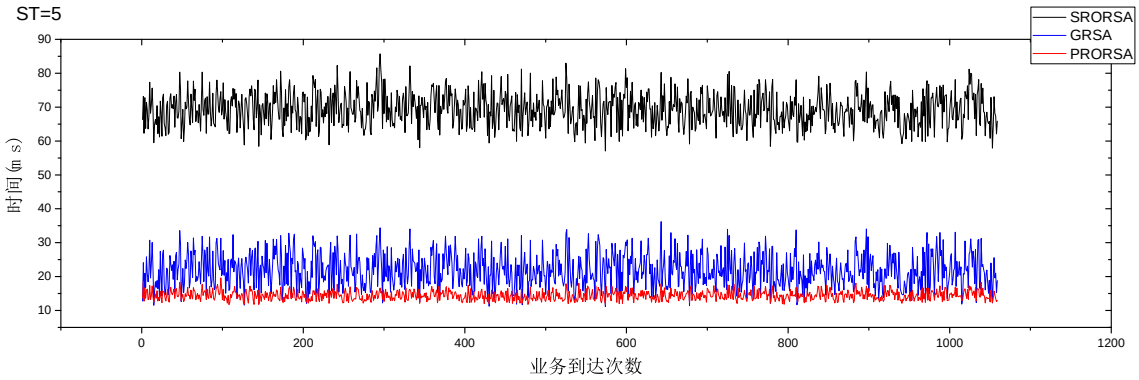

图 4-13 无权图时间对比（ST=5）

4.6.2.2 时间分析

图 4-13到图 4-17展示了在不同平均服务时间 *ST* 下的各种算法的计算时间，当 $ST = 5$ 时，我们可以看到通过 GPU 加速的 PRORSA 的计算时间是 SPROA 的计算时间的 1/4，而实际上备选路径部分的 GPU 加速达到了 7-8 倍，但是由于步骤二的快速路径选择过程也需要消耗一部分时间，这部分时间不能进行 GPU 加速，实际上 PRORSA 的大部分算法时间花在了步骤二的路径选择上，所以使得总体的加速比下降为 4 倍左右。我们发现由于 PRORSA 的波动幅度比 SRORSA 的波动幅度小很多，这是因为 SRORSA 的大部分时间花在计算备选路上, 备选路径的计算量随着业务数量和网络链路的占用情况变化较大，而 PRORSA 的计算量花在步骤二上，计算量变化较小。GRSA 的计算时间略高于 PRORSA 的计算时间，而且 GRSA 的波动幅度很大，不够平稳，这是因为 GRSA 贪心策略占用链路资源过多，容易造成分层图链路的碎片化，使得计算量变化较大。观察到随着 *ST* 的变化 PRORSA 的对 SRORSA 的加速比逐渐下降，这是因为随着网络压力的增加，可用链路变少，使得 SRORSA 的备选路径计算复杂度下降，PRORSA 加速优势变小。另外，随着 *ST* 的增加，PRORSA 和 SRORSA 的时间波动幅度增加，这是因为由于链路繁忙，使得大量业务不能一次性加入，步骤二到步骤一之间的循环次数增加，使得某些业务到达点的业务需要多次的循环，计算时间变长，从而拉大了时间变化幅度。

4.6.2.3 阻塞率分析

图 4-18到图 4-21展示了随着 *ST* 的变化，PRORSA，SRORSA 和 GRSA 的阻塞率变化情况，其中 PRORSA，SRORSA 由于是同一种算法，所以其阻塞率几乎一样。当 $ST = 10$ 时, 我们发现 PRORSA/SRORSA 的阻塞次数明显小于 GRSA。当 $ST = 20$ 时，PRORSA/SRORSA 的阻塞次数和阻塞幅度均小于 GRSA，在 GRSA 中出现了一次相对较大的阻塞，但是 PRORSA/SRORSA 中没有出现这种不平稳的阻

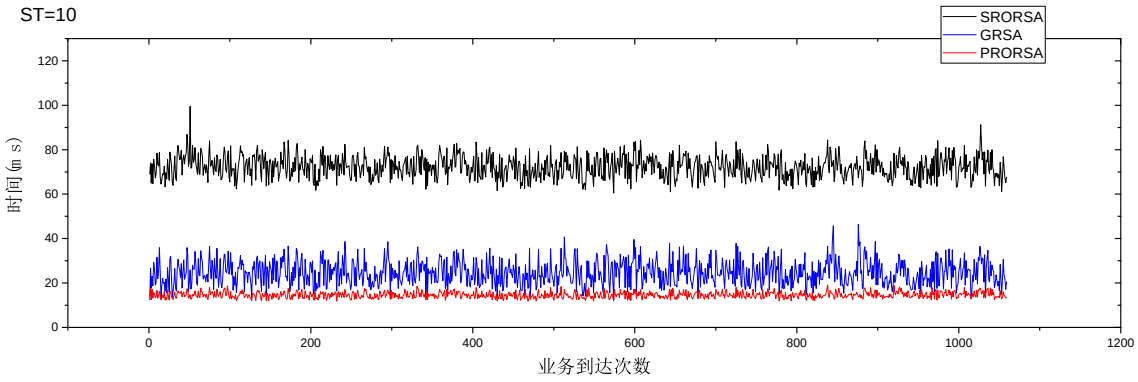

图 4-14 无权图时间对比（ST=10）

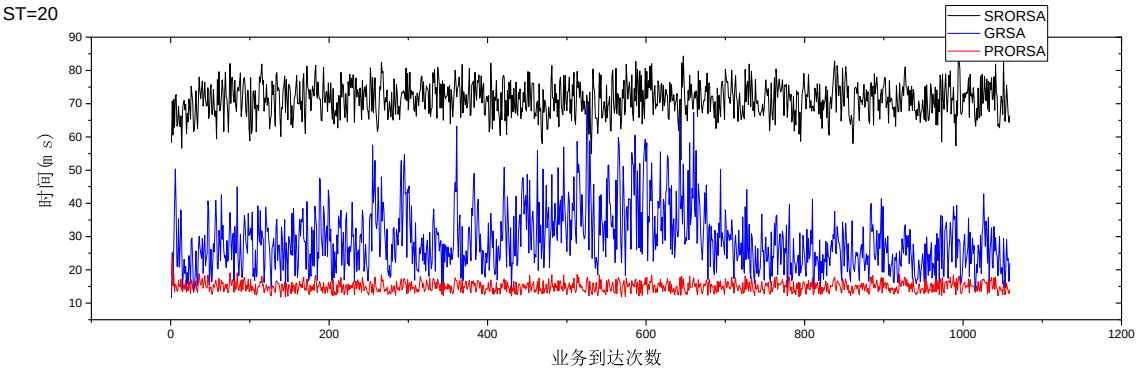

图 4-15 无权图时间对比（ST=20）

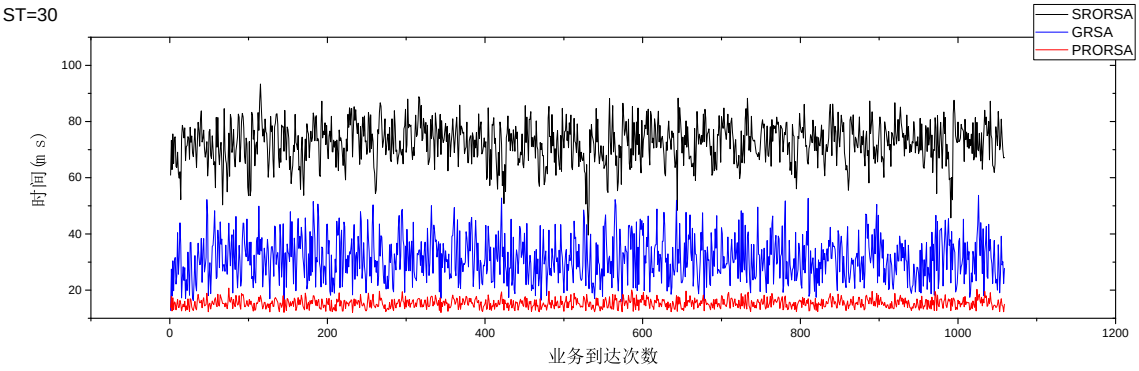

图 4-16 无权图时间对比（ST=30）

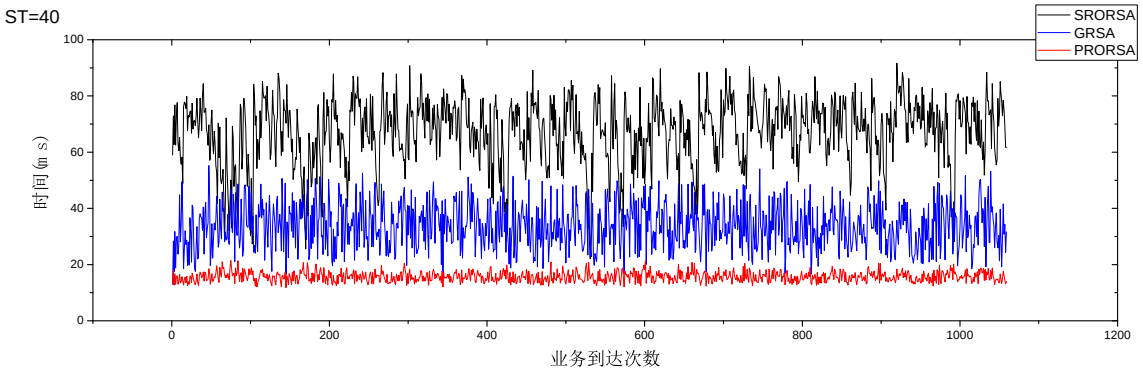

图 4-17 无权图时间对比（ST=40）

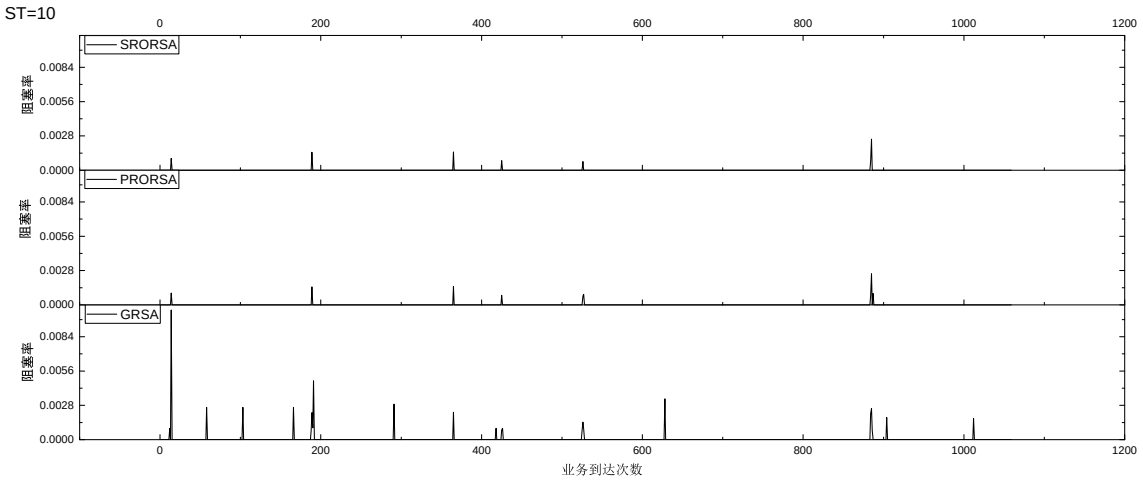

图 4-18 无权图阻塞率对比（ST=10）

塞率突变。当 $ST=30$ 时，我们发现 PRORSA/SRORSA 的阻塞次数和阻塞幅度比 GRSA 小很多，GRSA 的平均阻塞率是 PRORSA/SRORSA 的 6 倍左右。当 $ST=40$ 时，PRORSA，SRORSA 和 GRSA 的阻塞率都增加很多，但是 PRORSA/SRORSA 的阻塞情况还是大大优于 GRSA，可见在大服务压力的网络中，PRORSA/SRORSA 依然能够有效的减小阻塞率。

4.6.3 带权图下的仿真结果

4.6.3.1 路径优化分析

图 4-22到图 4-26表示了权值的优化结果，我们把 PRORSA 和 SRORSA 的优化结果与贪心算法 GRSA 的算法权值优化结果进行比较，图中的横坐标表示业务到达的次数，一共产生 1051 个加入事件，网络链路的权值设置为 1 到 10 之间的整数值，当网络状况较好的时候，比如, 平均服务时间为 $ST=5$ 时，路径权值达到了平均 3.5 倍的优化，PRORSA 和 SRORSA 的权值在 3-10 之间波动，把业务的路

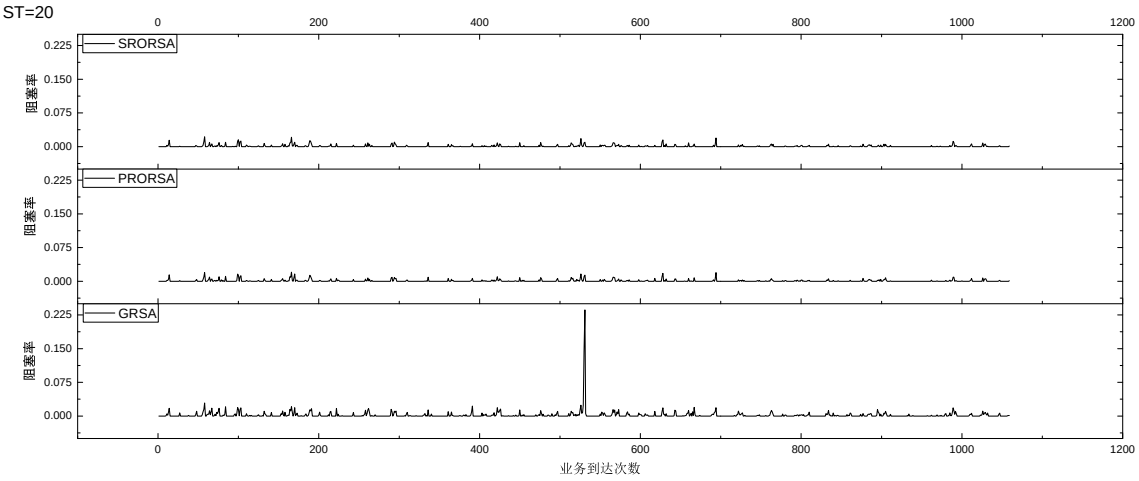

图 4-19 无权图阻塞率对比（ST=20）

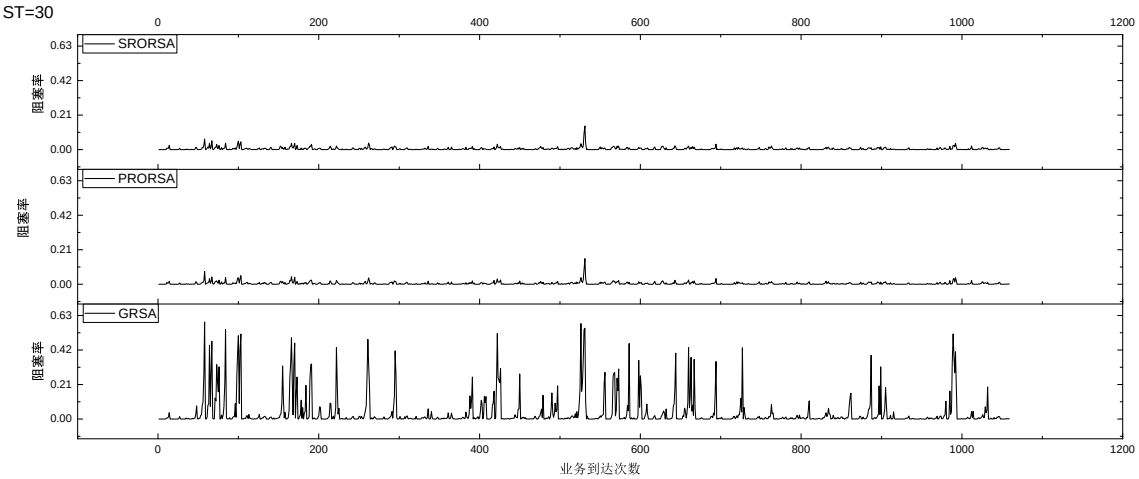

图 4-20 无权图阻塞率对比（ST=30）

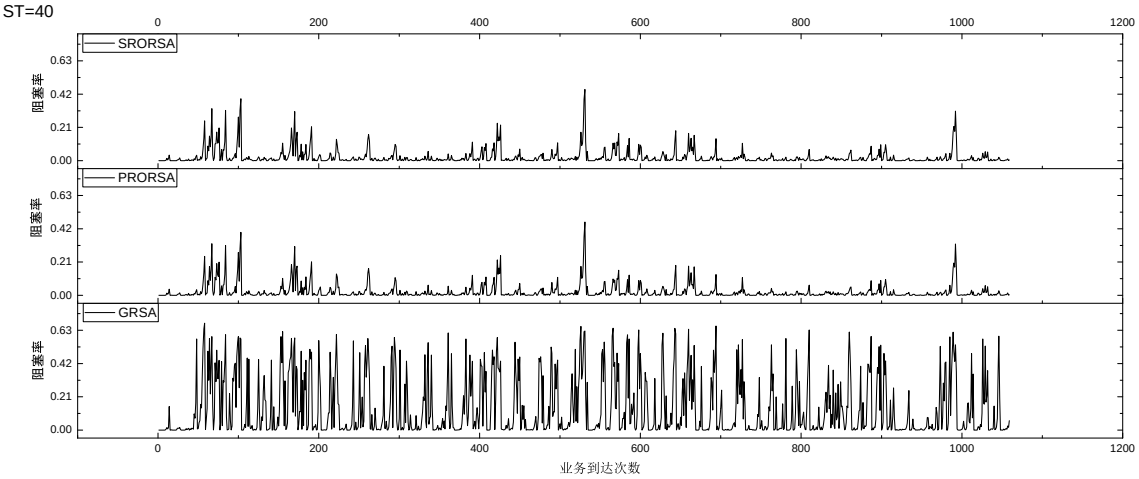

图 4-21 无权图阻塞率对比（ST=40）

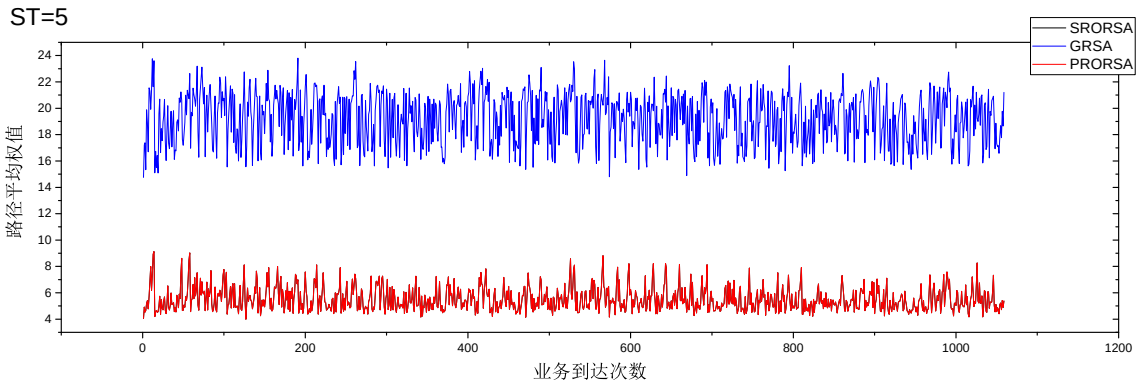

图 4-22 带权图路径权值对比（ST=5）

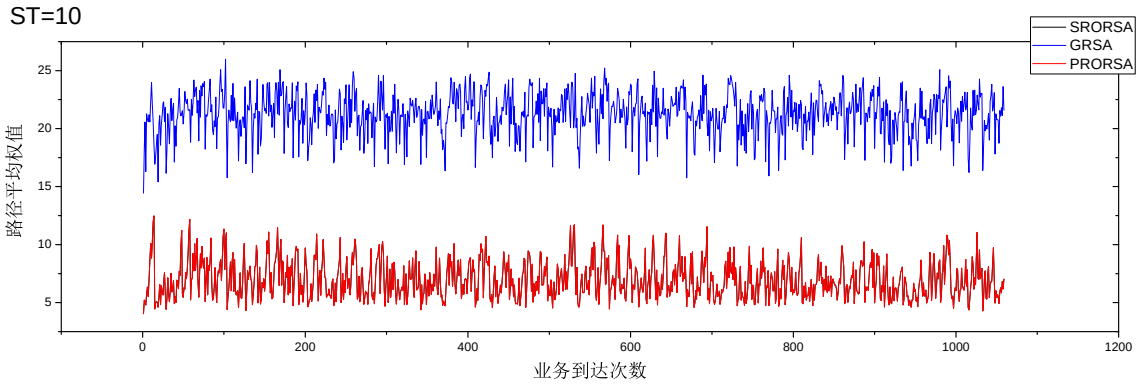

图 4-23 带权图路径权值对比（ST=10）

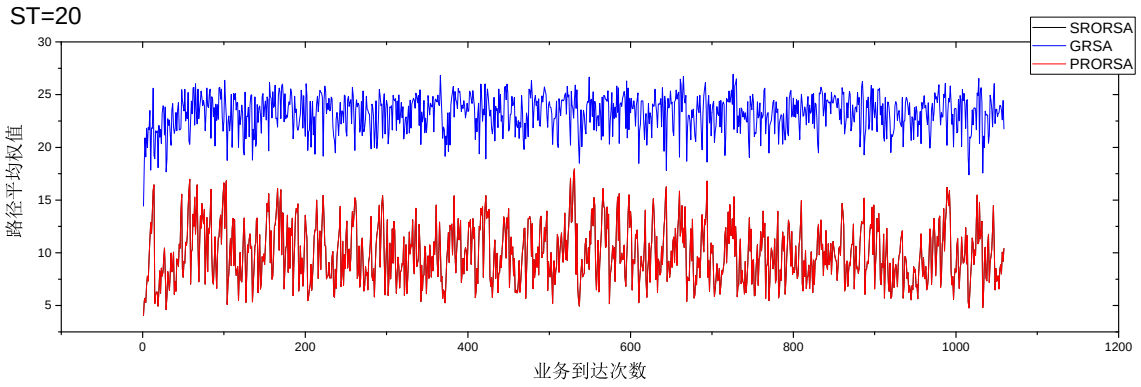

图 4-24 带权图路径权值对比（ST=20）

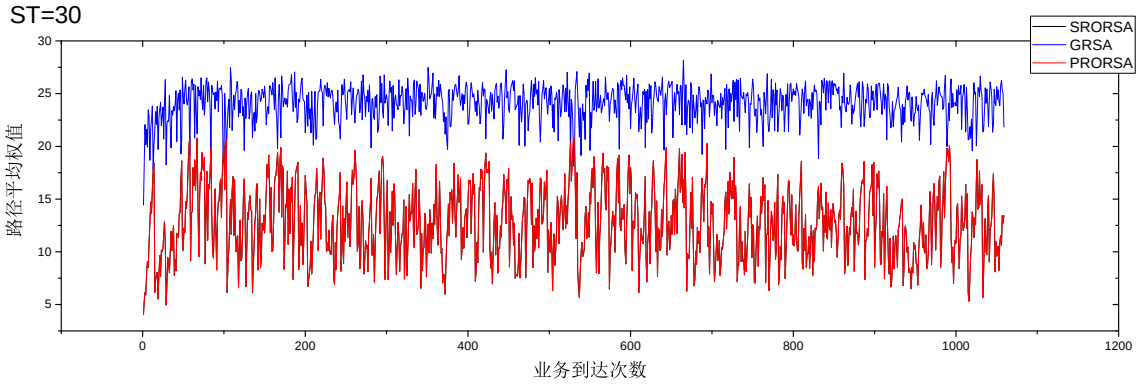

图 4-25 带权图路径权值对比（ST=30）

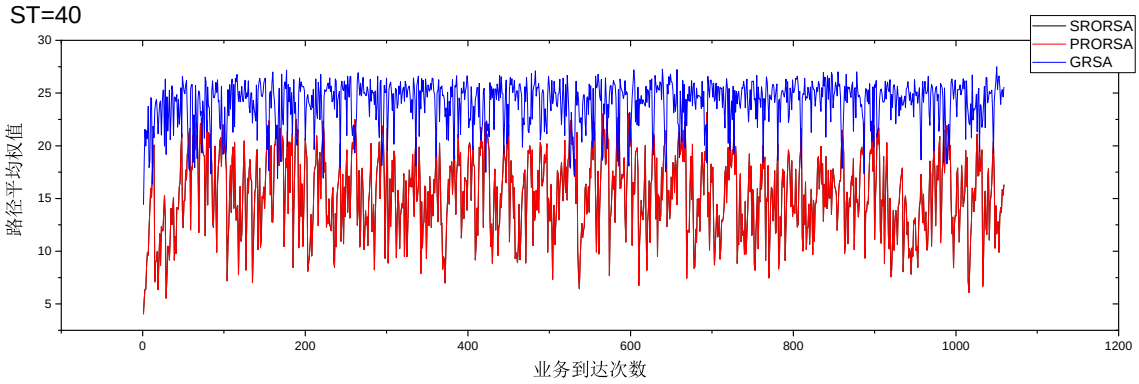

图 4-26 带权图路径权值对比（ST=40）

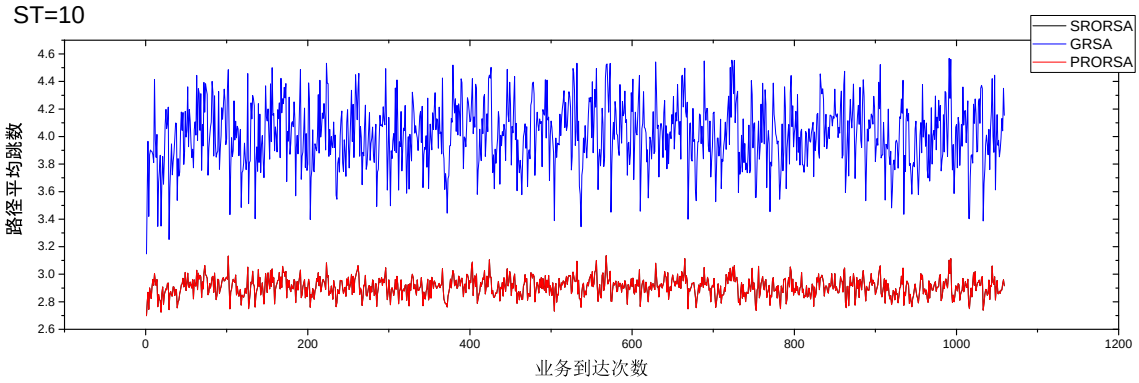

图 4-27 带权图路径跳数对比（ST=10）

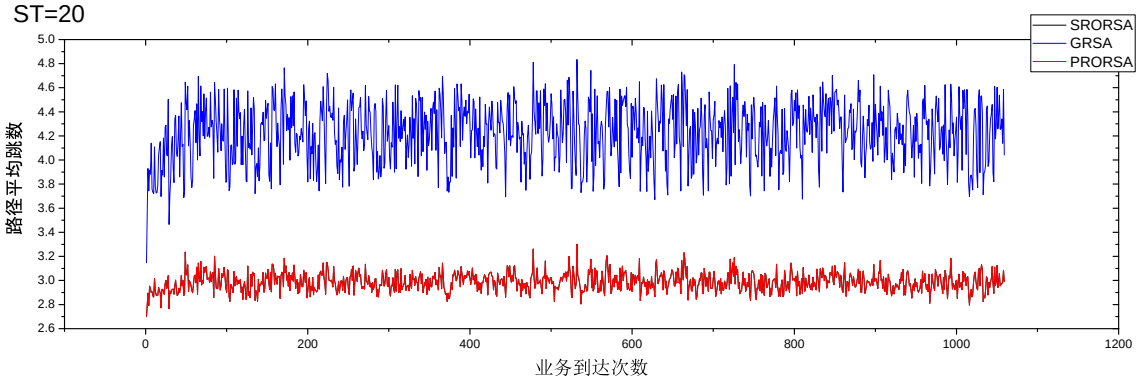

图 4-28 带权图路径跳数对比（ST=20）

径权值控制在很小的范围，路径优化得非常的理想。另外，随着平均服务时间 *ST* 的增加，贪心算法 GRSA 的路径平均权值轻微增加，由平均 20 增加到平均 25，这是因为 GRSA 是贪心的利用分层图，对链路的权值大小没有限制，不管在拥塞和不拥塞的网络条件下都会产生大量高权值的路径。观察 PRORSA 和 SRORSA 路径权值随着 *ST* 的变化发现，PRORSA 和 SRORSA 的链路权值随着 *ST* 的增加而不断上涨，从最初的均值为 7 到最后 ($ST = 40$) 时的均值为 15，这是因为当网络压力较大时，网络中的可用链路会出现匮乏，每个分层图上的路径权值都会很大，使得优化选择的空间变小。

图 4-27到 4-29显示了带权情况下的路径跳数情况，可以看到当权值优化时，业务的跳数也得到了很大的优化，这是因为权值小的路径常常路跳径较短，而且 GRSA 是会贪心的在一个分层图上加入业务，导致分层图变得碎片化，容易求得跳数较大的路径，所以即使在带权图的优化中，跳数也会减小很大。

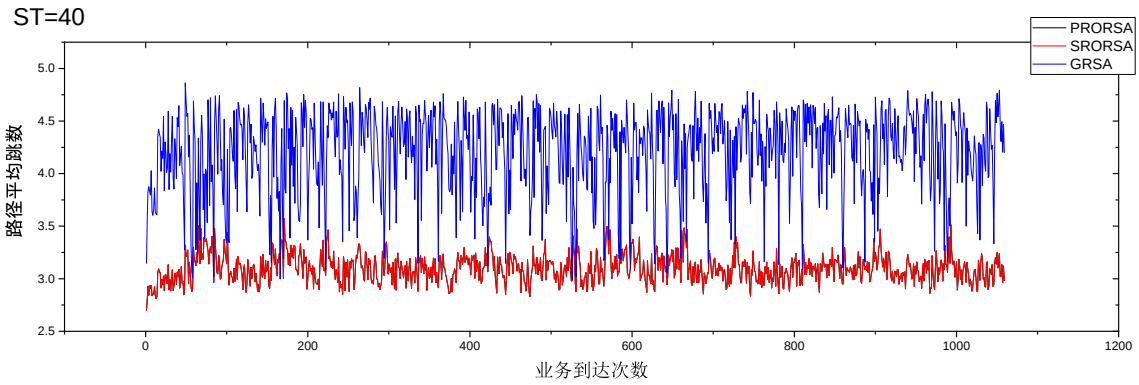

图 4-29 带权图路径跳数对比（ST=40）

4.6.3.2 时间分析

图 4-30到图 4-34展示了在不同平均服务时间 *ST* 下的各种算法的计算时间，当 $ST = 5$ 时，我们可以看到通过 GPU 加速的 PRORSA 的计算时间是 SPROA 的计算时间的 1/4，而实际上备选路径部分的 GPU 加速达到了 7-8 倍，但是由于步骤二的快速路径选择过程也需要消耗一部分时间，这部分时间不能进行 GPU 加速，实际上 PRORSA 的大部分算法时间花在了步骤二的路径选择上，所以使得总体的加速比下降为 4 倍左右。我们发现由于 PRORSA 的波动幅度比 SRORSA 的波动幅度小很多，这是因为 SRORSA 的大部分时间花在计算备选路上, 备选路径的计算量随着业务数量和网络链路的占用情况变化较大，而 PRORSA 的计算量花在步骤二上，计算量变化较小。GRSA 的计算时间略高于 PRORSA 的计算时间，而且 GRSA 的波动幅度很大，不够平稳，这是因为 GRSA 贪心策略占用链路资源过多，容易造成分层图链路的碎片化，使得计算量变化较大。观察到随着 *ST* 的变化 PRORSA 的对 SRORSA 的加速比逐渐下降，这是因为随着网络压力的增加，可用链路变少，使得 SRORSA 的备选路径计算复杂度下降，PRORSA 加速优势变小。另外，随着 *ST* 的增加，PRORSA 和 SRORSA 的波动幅度增加，这是因为由于链路繁忙，使得大量业务不能一次性加入，步骤二到步骤一之间的循环次数增加，使得某些业务到达点的业务需要多次的循环，计算时间变长。

4.6.3.3 阻塞率分析

图 4-35到图 4-38展示了带权情况下随着 *ST* 的变化，PRORSA，SRORSA 和 GRSA 的阻塞率变化情况，由于带权情况下的跳数也优化得很理想，所以其占用的链路资源也较少，阻塞情况得到改善。当 $ST = 10$ 时, 我们发现 PRORSA/SRORSA 的阻塞次数明显小于 GRSA。当 $ST = 20$ 时，PRORSA/SRORSA 的阻塞次数和阻塞幅度均小于 GRSA，在 GRSA 中出现了一次相对较大的阻塞，但是 PRORSA/SRORSA

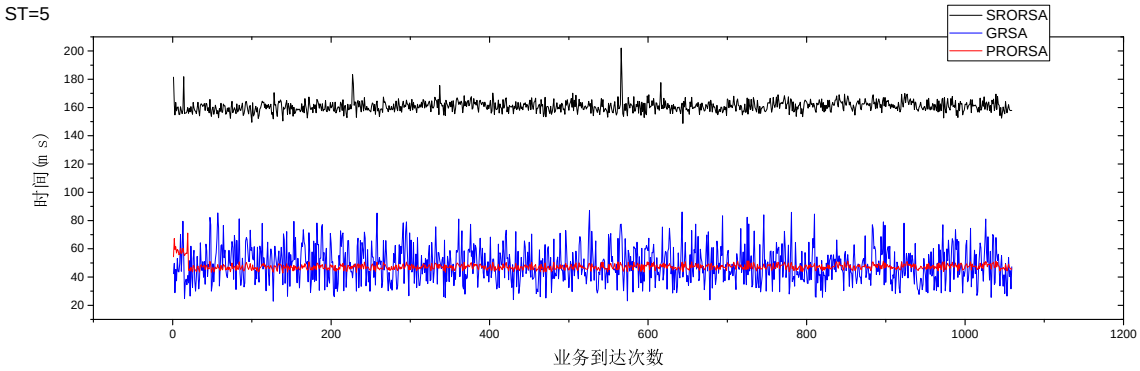

图 4-30 带权图时间对比（ST=5）

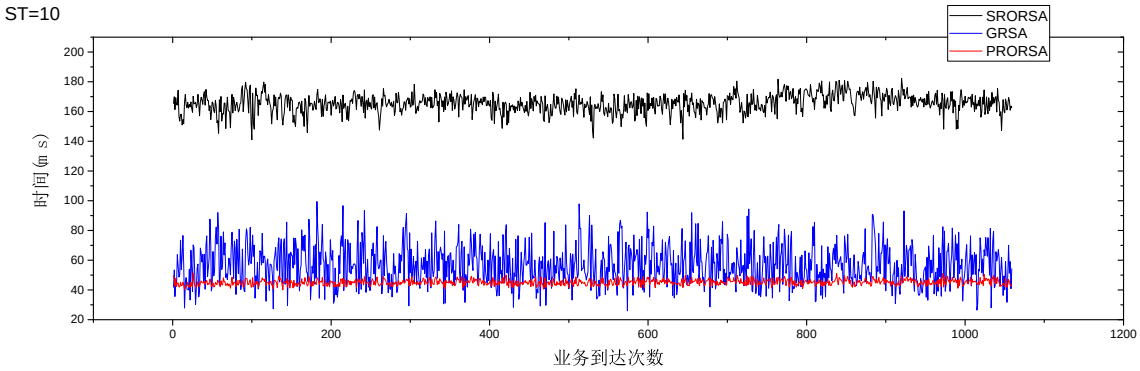

图 4-31 带权图时间对比（ST=10）

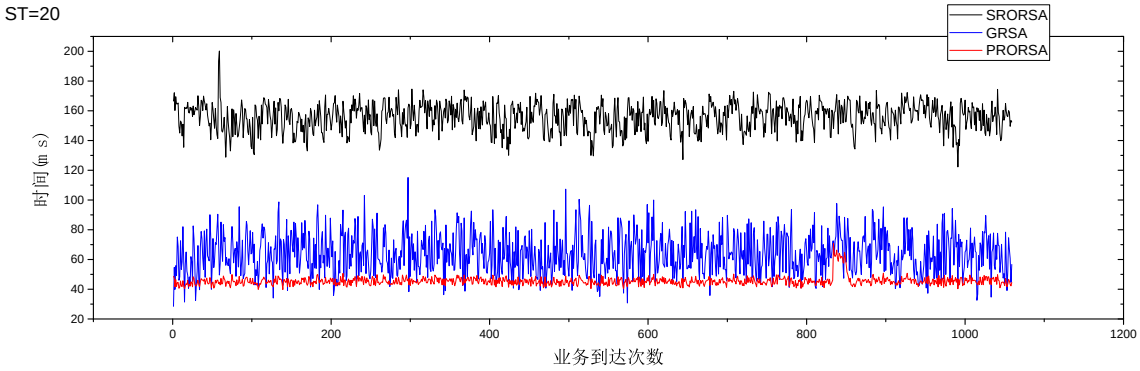

图 4-32 带权图时间对比（ST=20）

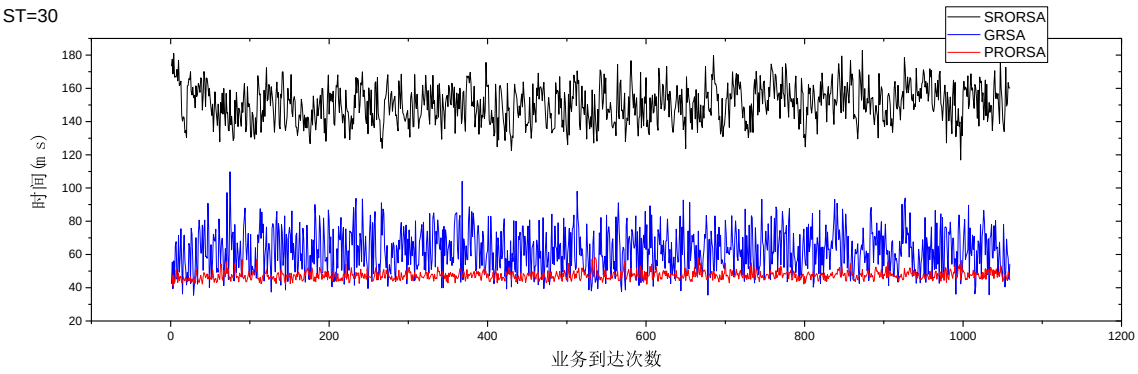

图 4-33 带权图时间对比（ST=30）

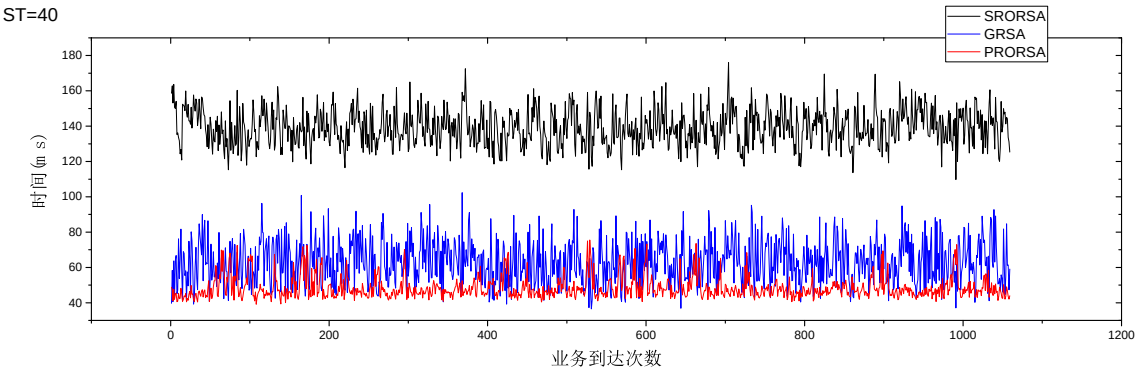

图 4-34 带权图时间对比（ST=40）

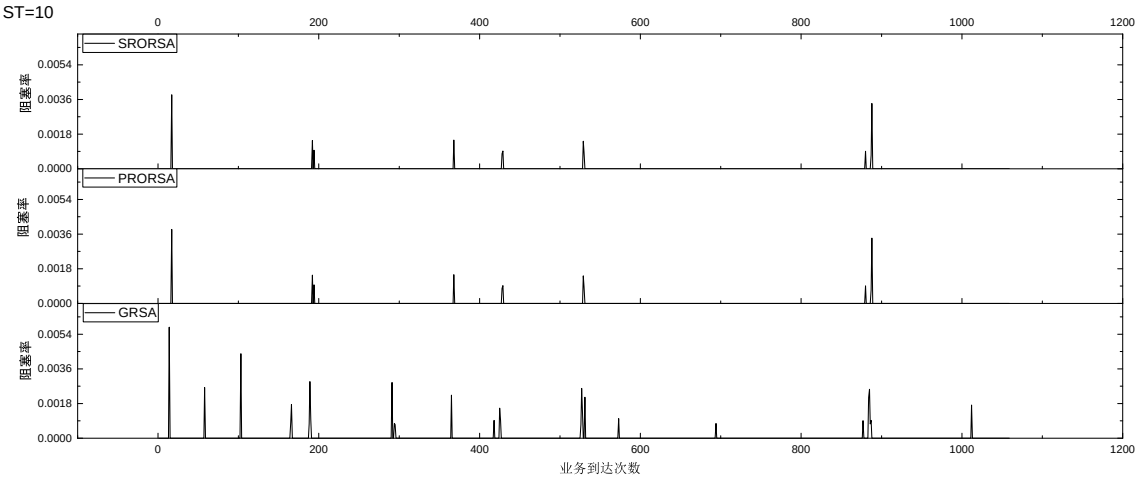

图 4-35 带权图阻塞率对比（ST=10）

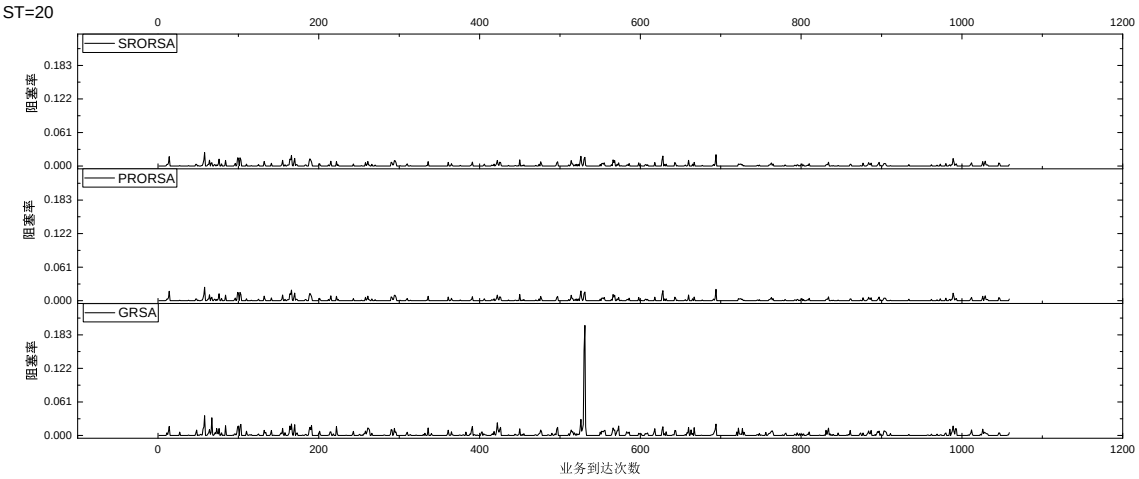

图 4-36 带权图阻塞率对比（ST=20）

中没有出现这种不平稳的阻塞率突变。当 $ST = 30$ 时，我们发现 PRORSA/SRORSA 的阻塞次数和阻塞幅度比 GRSA 小很多，GRSA 的平均阻塞率是 PRORSA/SRORSA 的 6 倍左右。当 $ST = 40$ 时，PRORSA，SRORSA 和 GRSA 的阻塞率都增加很多，但是 PRORSA/SRORSA 的阻塞情况还是大大优于 GRSA。

4.7 本章总结

本章首先讨论了 EON 中的 RSA 问题，介绍了分层图模型，针对分层图模型设计了 RORSA 的算法框架，对框架中路由计算部分进行了并行设计，分别在无权图和带权图两种情况下设计了基于 GPU 的并行算法，实验发现 RORSA 能够大大优化路径跳数，路径代价和阻塞率，同时，并行算法 PRORSA 能加速达到 5 被以上。

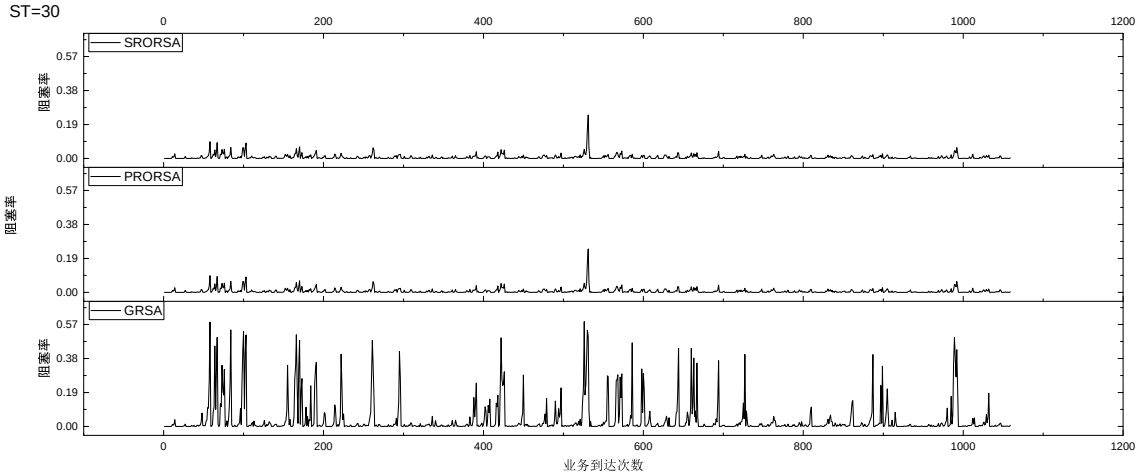

图 4-37 带权图阻塞率对比（ST=30）

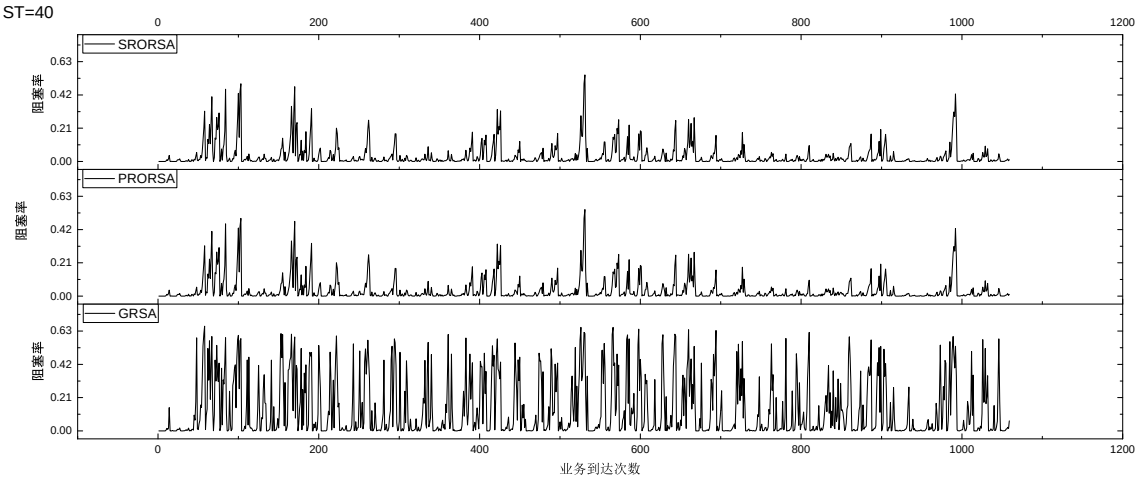

图 4-38 带权图阻塞率对比（ST=40）

第五章 全文总结与展望

5.1 全文总结

本文主要研究了 SDN 网络下适合于并行计算的业务量工程算法框架，并且针对这些框架，设计基于 GPU 的并行算法。

本文首先简要介绍了 GPU 并行优化原理，主要包括 GPU 与 CPU 的区别，GPU 的架构特点和 CUDA 编程模式，这些原理是第三章和第四章基于 GPU 的并行算法设计的理论基础。

第三章，SDN IP 网络中，本文提出了两种并行算法 GA-PROA 和 LR-PROA，GA-PROA 采用基于备选路径的模型，利用遗传算法来求解业务量工程问题，本文挖掘了遗传算法的并行特点，对 GA-PROA 进行了在 GP 的并行优化，优化加速达到 10 倍以上。LR-PROA 采用基于拉格朗日松弛的模型，利用拉格朗日松弛得到原问题的对偶问题，将对偶问题分解为对一系列业务求解最短路径计算问题，充分挖掘对业务最短路径计算的并行特点，设计了高并行粒度的多业务路由算法来加速路由计算，采用次梯度优化来求解对偶问题，对次梯度的步长设置进行讨论，最终得到收敛快速的步长更新策略，同时，为了从对偶解中挖掘出原问题的可行优化解，本文设计了路径调整的算法策略，实验表明 LR-PROA 能够在短时间内得到业务量工程问题的优化解，而且基于 GPU 的并行算法 LR-PROA 对串行算法 LR-SROA 加速可达到 6 倍。

第四章，针对 SDN 弹性光网络，本文采用分层图模型来进行业务量工程的优化，为了优化弹性光网络中的资源使用，降低阻塞率，本文提出了 RORSA 算法框架，并对框架进行并行设计，分别针对带权图和无权图设计了基于 GPU 的并行路由算法，在无权图情况下，本文采用并行 BFS 算法来加速无权图的 RORSA 框架；在带权图情况下，本文首先提出了带权带跳数约束的最短路动态规划模型，并对动态规划算法进行并行设计来加速带权图下的 RORSA 算法框架，实验发现 RORSA 和一般的贪心路由算法相比较，可以大大降低路径的代价，路径跳数，节省网络资源，同时 RORSA 还能有效降低业务的阻塞率，同时，实验发现基于 GPU 的并行算法 PTESAA 对串行算法 STESAA 加速可达到 5 倍。

5.2 后续工作展望

GPU 并不擅长分支较多的计算，GPU 更加适合逻辑简单但计算量很大的任务加速，而路由算法中计算操作并不多，而大部分是逻辑计算，这也是为什么本文中

的路由算法只能获得小于 10 倍的加速的原因，而且本文中的算法只能在业务数量达到一定规模的情况下才能达到加速效果，本文只利用了 GPU 的并行处理能力，而没有充分利用 GPU 的大规模浮点计算能力，这也是 GPU 应用在路由算法上的限制，所以在今后的工作中可以挖掘更多的计算密集型的任务进行 GPU 加速。

在研究本次毕业设计的过程中，本人除了设计 GA-PROA,LR-PROA 和 PR-ORSA 算法之外，还针对弹性光网络分层图上的网络流问题进行了基于 GPU 的并行化研究，本人对预流推进算法进行了并行化的设计并在中加入跳限约束，但是由于算法逻辑复杂，加速效果并不好，只能在一些特殊情况，比如流很多的情况下才有一定的加速比，所以这部分工作并没有写进本次设计，后续工作会继续改进这个并行预流推进算法，希望可以得到好的加速效果。

致 谢

在攻读硕士学位期间，首先衷心感谢我的导师王雄副教授三年来的关心和指导，感谢全体 KB318 教研室的同学的支持。

参考文献

[1] OpenFlow Switch Specification, https://www.opennetworking.org/images /stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Computer Communicaiton Review*,2008,38(2),69-74.

[3] N. McKeown. Software-defined networking[J]. *INFOCOM keynote talk*,2009,17(2),30-32.

[4] Kim H,Feamster N. Improving network management with software defined networking. *IEEE Communications Magazine*,2013,51(2),114-119.

[5] B. A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, A survey of software-defined networking :past, present, and future of programming networks, *IEEE Communications Surveys & Tutorials*,2014,16(3),1617-1634.

[6] R. Masoudi and A. Ghaffari, Software defined networks: a survey, Journal of Network and Computer Applications,2016,67,1-25.

[7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan, *in Proceedings of the ACM SIGCOMM*,2013,15-26.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, and M. Zhu et al. B4: Experience with a globally-deployed software defined wan, *in Proceedings of the ACM SIGCOMM*,2013,3-14.

[9] Cisco Visual Networking Index: Forecast and Methodology, 2016C2021, http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html

[10] S. S. Soliman and B. Song, Fifth generation (5G) cellular and the network for tomorrow, Journal of Network and Computer Applications,2017,85,84-93.

[11] C. Guo, L. Yuan, D. Xiang, et al. Pingmesh: a large-scale system for data center network latency measurement and analysis, *ACM SIGCOMM*, 2015.

[12] S. Gay, R. Hartert, and S. Vissicchio. Expect the unexpected:sub-second optimization ofr segment routing, *IEEE INFOCOM*, 2017.

[13] Nvidia. Programming guide: CUDA toolkit documentation, http://docs.nvidia.com/cuda/cuda_c_programming_guide/index.html#axzz4mVqP2TEC

[14] Nvidia.Programming guide: CUDA toolkit documentation, http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf.

[15] Jason Sanders,Edward Kandrot.GPU 高性能之 CUDA 实战 [M], 机械工业出版社,2011.

[16] B. McCormick, F. Kelly, P. Plante, P. Gunning, and P. Ashwood-Smith. Real time alpha-fairness based traffic engineering, *in Procedings of HotSDN*, 2014.

[17] K. Kikuta, E. Oki, N. Yamanaka, N. Togawa, and H. Nakazato. Effective parallel algorithm for GPGPU-accelerated explicit routing optimization, *in Procedings of GLOBECOM*, 2015.

[18] Y. Lee and B. Mukherjee. Traffic engineering in next-generation optical networks, *IEEE Communications Surveys & Tutorials*,2004,6(3),16-33.

[19] A. Mendiola, J. Astorga, E. Jacob, and M. Higuero. A survey on the contributions of software-defined networking to traffic engineering, *IEEE Communications Surveys & Tutorials*,2017,19(2),918-953.

[20] N. Wang, K. Ho, G. Pavlou, and M. Howarth. An overview of routing optimization for Internet traffic engineering, *IEEE Communications Surveys & Tutorials*,2008,10(1),36-56.

[21] Y. Wang and Z. Wang. Explicit routing algorithms for Internet traffic engineering, *in Procedings of International Conference on Computer Communications and Networks*, 1999.

[22] M. Pioro and D. Medhi. Routing, flow, and capacity design in communication and computer networks, Morgan Kaufmann Publishers, 2004.

[23] S. Agarwal, M. Kodialam, and T. V. Lakshman. Traffic engineering in software defined networks, *in proceedings of INFOCOMM*, 2013.

[24] J. Rexford. Route optimization in IP networks, *in Handbook of Optimizationin Telecommunications*, Springer Science + Business Media, February 2006.

[25] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS adaptive traffic engineering, *in proceedings of INFOCOMM*, 2001.

[26] J. He, M. Chiang, and J. Rexford. DATE: distibuted adaptive traffic engineering, *in proceedings of INFOCOMM*, 2006.

[27] A. R. Brodtkorb, T. R. Hagen, C. Schulz, and G. Halse. GPU Computing in Discrete Optimization Part I: Introduction to the GPU, *EURO Journal on Transportation and Logistics*,2013,2(1),129-157.

[28] K. Rocki and R. Suda. Accelerating 2-opt and 3-opt local search using GPU in the travelling salesman problem, *In Proceedings of IEEE/ACM International Conference on High Performance Computing and Simulation (HPCS)*,2012,489-495.

[29] C. Schulz. Efficient local search on the GPU - investigations on the vehicle routing problem, *Journal of Parallel and Distributed Computing*,2013,73(1),14-31.

[30] P. Harish and P. J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA. *In Proceedings of International Conference on High performance Computing*,2007,197-208.

[31] Q. N. Tran. Designing efficient many-core parallel algorithms for all-pairs shortest-poaths using CUDA. *In Proceedings of International Conference on Information Technology: New Generations*,2010,7-12.

[32] S. Rostrup, S. Srivastava, K. Singhal. Fast and memory-efficient minimum spanning tree on the GPU. *In Proceedings of International Workshop on GPUs and Scientific Applications*, 20011.

[33] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: responsive yet stable traffic engineering, *in Procedings of ACM SIGCOMM*,2005,253-264.

[34] H. Wang, H. Xie, and L. Qiu. COPE: traffic engineering in dynamic networks, *in Procedings of ACM SIGCOMM*,2006,99-110.

[35] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights, *in Procedings of IEEE INFOCOM*,2000,519-528.

[36] D. Xu, M. Chiang, and J. Rexford. Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering, *IEEE/ACM Transactions on Networking*,2011,19(6),1717-1730.

[37] A. Sharma, A. Mishra, V. Kumar, and A. Venkataramani. Beyond MLU: an application-centric comparison of traffic engineering schemes, *in Procedings of INFOCOM*, 2011.

[38] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Network flows: Theory, Algorithms, and Applications, Prentice Hall, 1993.

[39] P. Erdos and A. Renyi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*,1960,5,17-61.

[40] R. Albert and A. L. Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 2002,74,47-97, 2002.

[41] IBM ILOG CPLEX Optimization Studio, https://www.ibm.com/bs-en/marketplace/ibm-ilog-cplex.

[42] A. A. M. Saleh and J. M. Simmons,Technology and architecture to enable the explosive growth of the Internet,*IEEE Communication Magazine.*,vol,49(1),126-132.

[43] L. Zong, G. N. Liu, A. Lord, Y. R. Zhou and T. Ma.40/100/400 Gb/s mixed line rate transmission performance in flexgrid optical networks,*in 2013 Optical Fiber Communication Conference,Anaheim*,2013

[44] T. Wuth, M. W. Chbat, and V. F. kamalov, Multi-rate(100G/40G/10G) transport over deployed optical networks,*in 2008 Optical Fiber Communication Conference,San Diego*,2008.

[45] Spectral grids for WDM applications: DWDM frequencygrid,*ITU-T*,G.694.1.

[46] H. Zang, J. P. Jue, B. Mukherjee.A review of routing and wavelength assignment approaches for wavelengthrouted optical WDM networks,*Optical Networks Magazine*,2000,1,47-59.

[47] R. Ramaswami and K. N. Sivarajan,Routing and wavelength assignment in all-optical networks.*IEEE Transactions on Networking*,1995,3(5),489-500.

[48] 敖发良, 胡汉武, 全光网静态路由选择和波长分配的分层图算法. 光通信研究,vol.3,2003.