Grout：用于不规则计算的异步多 GPU 编程模型

Tal Ben-Nun Michael Sutton
以色列耶路撒冷希伯来大学
{talbn, msutton}@cs.huji.ac.il

Sreepathi Pai Keshav Pingali
美国德克萨斯大学奥斯汀分校
sreepai@ices.utexas.edu pingali@cs.utexas.edu

摘要

具有多个 GPU 的节点正在成为高性能计算的首选平台。但是，大多数应用程序都是使用批量同步编程模型编写的，这对于受益于低延迟异步通信的不规则算法可能不是最佳选择。本文提出了异步多 GPU 编程的结构，并描述了它们在称为 Groute 的精简运行时环境中的实现。Groute 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。我们证明，这种方法实现了最先进的性能，并为 8-GPU 和异构系统上的一套不规则应用程序提供了强大的扩展能力，使某些算法的加速速度提高了 7 倍以上。

类别和主题描述符：D.1.3 [编程技术]：并发编程
关键词：多 GPU，异步编程，不规则算法

1.研究动机

具有多个附加加速器的节点现在在高性能计算中无处不在。特别是，图形处理单元（GPU）因其电源效率，硬件并行性，可扩展的缓存机制以及专用指令和通用计算之间的平衡而变得流行。多 GPU 节点由一个主机（CPU）和多个 GPU 设备组成，这些设备通过低延迟、高吞吐量总线链接（参见图 1）。这些互连允许并行应用程序有效地交换数据，并利用 GPU 的组合计算能力和内存大小。

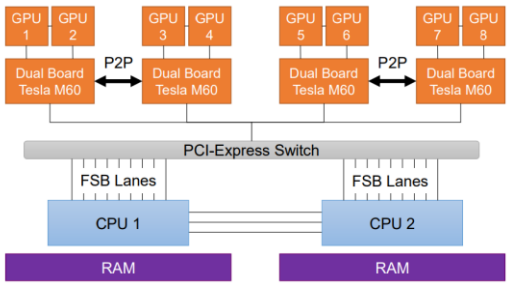


图 1 多 GPU 节点原理图

多 GPU 节点通常使用以下两种方法之一进行编程：在简单方法中，每个 GPU 都是单独管理的，每个设备使用一个进程[13，19]。或者，使用批量同步并行（BSP）[32]编程模型，其中应用程序以回合形式执行，每个轮次由本地计算和全局通信组成[6，26]。第一种方法会受到来自各种源（如操作系统）的开销的影响，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的执行的全局障碍中引入不必要的序列化。这两种编程方法都可能导致多 GPU 平台的利用率不足，特别是对于不规则的应用程序，这些应用程序可能会遭受负载不平衡的影响，并且可能具有不可预测的通信模式。

原则上，异步编程模型可以减少其中一些问题，因为与基于轮次的通信不同，处理器可以自主计算和通信，而无需等待其他处理器达到全局同步。但是，很少有应用程序利用异步执行，因为其开发需要深入了解底层体系结构和通信网络，并且涉及对代码执行复杂的修整。

本文介绍了 Groute——一种异步编程模型和运行时环境 [2]，可用于在多 GPU 系统上开发各种应用广泛的应用程序。

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

本文的主要贡献如下：

- 我们为异步执行和通信定义抽象编程结构。
- 我们证明了这些结构可用于定义各种算法，包括规则和不规则并行算法。
- 我们使用在现有框架中编写的应用程序作为基准来比较我们实现的性能方面。我们使用在现有框架中编写的应用程序作为基准，从各个方面来比较实现的性能。
- 我们证明，使用 Groute，可以实现在大多数情况下优于最先进实现的异步应用程序，与单个 GPU 上的基线执行相比，在 8-GPU 环境中实现了高达 7.28 倍的加速效果。

2.多 GPU 节点架构

通常来说，加速器的作用是通过允许它们卸载应用程序的数据并行部分来补充可用的 CPU。反过来，CPU 负责流程管理，通信，输入/输出任务，内存传输和数据预处理/后处理。

如图 1 所示，CPU 和加速器通过前端总线（FSB，实现了包括 QPI 和 HyperTransport）相互连接。FSB 通道的计数是内存传输带宽的指标，它们链接到如 PCIExpress 或 NVLink 的互连端口，这些互连同时支持 CPU 到 GPU 和 GPU 到 GPU 之间的通信。

由于硬件布局的限制，例如使用相同的主板和电源单元，多 GPU 节点通常由 1-25 个 GPU 组成。CPU、GPU 和互连的拓扑在完整的全对连接和分层交换拓扑之间可能会有所不同。在图 1 所示的树形拓扑中，每个四联 GPU（即 1–4 和 5–8）它们之间可以直接进行通信，而每一个 GPU 与另一个四元组之间的通信是间接的，因此速度较直接通信慢。例如，GPU1 和 4 可以执行直接通信，而从 GPU4 到 5 的数据传输必须通过互连通道。交换接口允许每个 CPU 以相同的速率与所有 GPU 通信。在其他配置中，CPU 可以直接连接到其四联 GPU，这会导致 CPU-GPU 通信的带宽可变，具体情况取决于进程分派。

GPU 架构包含多个内存复制引擎，可实现同步执行代码和双向（输入/输出）内存传输。下面，我们将详细阐述使用并发副本在多 GPU 节点内进行有效通信的不同方式。

2.1 GPU 间通信

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

设备启动的内存传输通过虚拟寻址实现，虚拟寻址将所有主机和设备内存映射到单个地址空间。虽然 DA 比对等传输更灵活，但它对内存对齐、合并、活动线程数和访问顺序有高度敏感性。

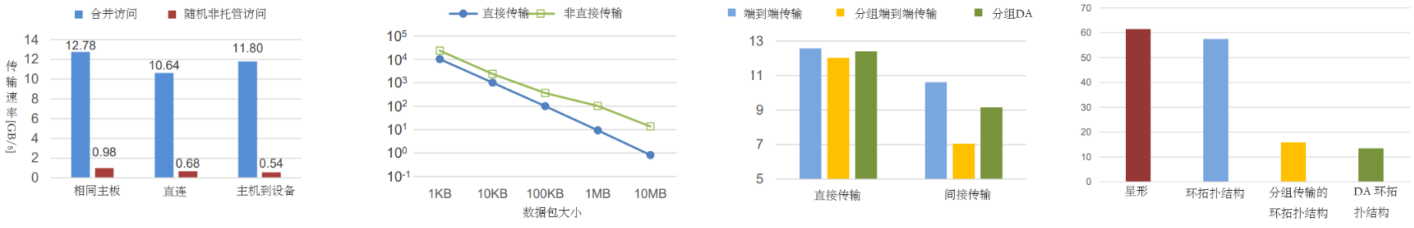
使用微基准（图 2），我们在实验设置的 8-GPU 系统上测量了 100MB 数据的传输，我们进行了平均超过 100 次试验（详细信息参见第 5 节）。

图 2a 显示了驻留在同一电路板上的 GPU 上设备启动的内存访问的传输速率;在不同的板上的 CPU-GPU 通信。该图显示了 DA 频谱的两个极端——从严格管理的合并访问（蓝色条，左侧）到随机的非托管访问（红色条，右侧）。可以观察到合并访问的执行速度比随机访问高出 21 倍。另请注意，内存传输速率与拓扑中路径的距离相关。由于增加了双板 GPU 的级别（如图 1 所示），CPU-GPU 的传输速度比两个不同的板 GPU 更快。

为了支持设备启动的传输，在无法访问彼此内存的 GPU 之间，可以执行两阶段间接复制。在间接复制中，源 GPU 首先将信息"推送"到主机内存，然后由目标 GPU"拉取"，使用主机标志和系统范围的内存栅栏进行同步。

在图 1 所示的拓扑中，GPU 一次只能传输到一个目标。这阻碍了异步系统的响应能力，尤其是在传输大型缓冲区时。解决此
问题的一种方法是当消息在网络中传输时将其切分为多个数据包。图 2b 显示了使用分组内存传输而不是单个对等传输的开销。该图显示，开销随着数据包大小的增加而线性减少，对于 1-10MB 数据包，开销范围在 1~10%之间。此参数可根据各个应用程序进行调整来平衡延迟和带宽。

图 2c 比较了直接（推送）和间接（推/拉）传输的传输速率，表明分组设备启动的传输和细粒度控制是有效的，即使比主机管理的分组对等传输也是如此。请注意，由于设备启动的内存访问是用用户代码编写的，因此可以在传输过程中执行其他的数据处理。



（a）直接内存访问顺序　（b）打包开销　（c）分组传输速率　（d）对等广播性能

图 2 GPU 间内存传输微基准

多 GPU 通信的另一个重要方面是多个源/目标传输，就像在集合操作中一样。由于互连和内存复制引擎的结构，优化较差的应用程序可能会使总线拥堵。NCCL 库[24]中使用的一种方法是在总线上创建环形拓扑。在这种方法中（如图 3 所示），每个 GPU 都传输到一个目标，通过直接或间接的设备启动传输进行通信。这可确保使用每个 GPU 的两个内存复制引擎，并充分利用总线。数据包化还用于在从上一个流水线操作接收信息时开始将信息传输到下一个 GPU。

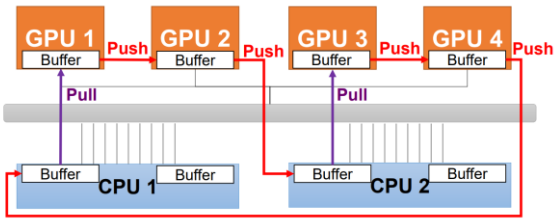


图 3 DA 环拓扑技术

图 2d 比较了实现一对全 GPU 对等广播的不同方法的性能，从来自一个源的 7 个异步传输，到完整和分组的对等传输，再到上述 DA 环方法。该图显示环形拓扑始终优于单独的直接副本。这可以归因于间接对等体传输的数据量较少（一个对等体传输到环中的第二个四元组，而不是一对一的四个）。此外，打包会触发复制流水线，从而大大减少运行时间。DA 环的性能仅略优于主机控制的环形传输，与图 2c 中更快的传输速率一致。

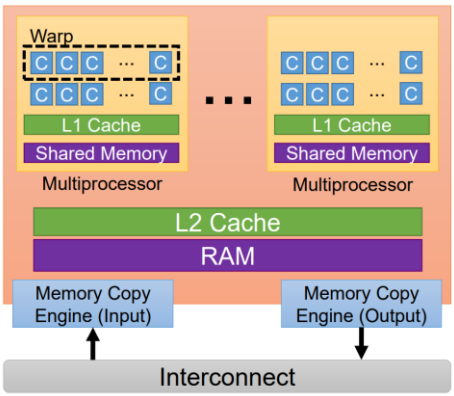


图 4 单 GPU 架构

2.2 GPU 编程模型

单个 GPU 设备的结构如图 4 所示。每个 GPU 都包含一组固定的多处理器（MP）和一个 RAM 单元（称为全局内存）。GPU 过程（内核）通过调度由多个线程组成的网格（分组为线程块）在 MP 上并行运行。在每个分配给单个 MP 的线程块中，Warp（通常由 32 个线程组成）在内核上同步执行。此外，同一线程块中的线程可以通过共享内存进行同步和通信，以及使用自动管理的 L1 和 L2 高速缓存。为了支持并发内存写入，在共享内存和全局内存上定义了原子操作。

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

GPU 调度程序按线程块分派内核，当没有更多线程块可供正在运行的内核调度时，通过调度其他内核的线程块，在同一 GPU 上实现多流并发。但是，高优先级流允许应用程序开发人员立即调用内核，先调度来自新内核的线程块，然后再调度来自正在运行的内核的线程块。

虽然流/事件构造提供了对内核调度的细粒度控制，但在编写涉及多个 GPU 的高级功能时会出现困难。在下一节中，我们列出了一个编程表 1：Groute 编程接口抽象，它补充了现有模型，以简化多 GPU 开发，使用来自微基准测试得出的结论来最小化通信延迟。

表 1 Groute 编程接口

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

3.Groute 编程模型

Groute 编程模型提供了多种构造来促进异步多 GPU 编程。表 1 列出了这些构造及其编程接口的简述。

Groute 应用程序的运行包括两个阶段：数据流图构造和异步计算。Groute 程序首先指定计算的数据流图。这个有向图中的节点（我们称之为端点）表示（i）CPU 和 GPU 等物理设备，或（ii）称为路由器的虚拟设备，它们是实现复杂通信模式的抽象节点。数据流图中的边缘表示端点之间的通信链路，只要没有自循环（端点直接连接到自身）或多图中的多个相同边（即路由器只能有一个到同一端点的传出边），就可以创建边。注意，为了支持多任务处理，可以从同一物理设备创建多个虚拟终结点。

发送和接收方法允许端点在链路上发送和接收数据;收到数据后，端点可以使用回调对其执行操作。创建路由器时，程序员会指定路由策略，以确定接收到输入时的行为。例如，输入可以发送到单个终结点，也可以根据其可用性发送到终结点的子集。创建链路时，将指定该链路的分组和多缓冲策略（第 2 节）。

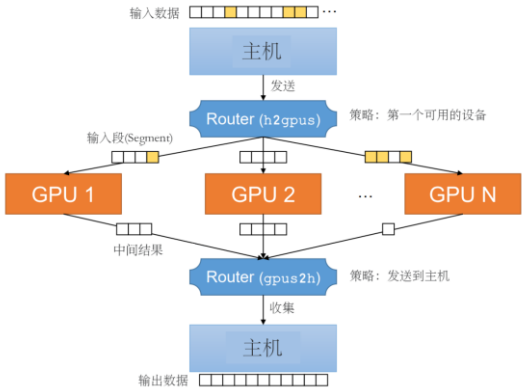


图 5 基于谓词筛选的数据流图

为了演示 Groute 模型，我们描述了使用 Groute 实现基于谓词的过滤器（PBF）的过程，如图 5 和图 6 所示。PBF 是许多应用程序中的内核，例如数据库管理和图像处理。PBF 的输入是一个大型一维数组，输出是一个数组，其中包含满足给定谓词的输入数组的所有元素。在 Groute 程序中，主机将输入数组划分为多个段，并根据需要将它们发送到空闲 GPU，以促进负载平衡。处理过的段由 GPU 传输回主机，在那里它们被组装以产生输出。图 5 描述了 PBF 的数据流图。

在图 6 中，代码在第 5-20 行中设置了 PBF 的数据流图。系统中存在的物理设备是通过访问上下文类型（第 5 行）的结构来确定的。PBF 程序创建一个名为 h2gpus 的路由器，用于将输入数组的段从主机分散到 GPU，以及一个名为 gpus2h 的路由器，用于从 GPU 收集段以创建输出数组（9-10 行).

```
1  std::vector<T> input = ...;
2  std::vector<T> output;
3  int packet_size = ...;
4
5  Context ctx;
6  auto all_gpus = ctx.devices();
7  int num_gpus = all_gpus.size();
8
9  Router h2gpus(1, num_gpus, AnyDevicePolic
10 Router gpus2h(num_gpus, 1, AnyDevicePolic
11
12 Link dist    (HOST, h2gpus, packet_size,
13 Link collect (gpus2h, HOST, packet_size,
14
15 for (device_t dev : all_gpus) {
16   std::thread t(WorkerThread,
17                 Link(h2gpus, dev, packet_
18                 Link(dev, gpus2h, packet_
19   t.detach();
20 }
21
22 dist.Send(input, input_size);
23 dist.Shutdown();
24
25 while(true) {
26   PendingSegment output_seg = collect.Receive().get();
27   if(output_seg.Empty()) break;
28   output_seg.Synchronize();
29   append(output, output_seg);
30   collect.Release(output_seg);
31 }
32 //-----------------------------------------------------
33 EndpointList AnyDevicePolicy(
34     const Segment& message, Endpoint source,
35     const EndpointList& router_dst) {
36   return router_dst;
37 }
38
39 void WorkerThread(device_t dev, Link in, Link out) {
40   Stream stream (dev);
41   T *s_out = ...;
42   int *out_size = ...;
43
44   while(true) {
45     PendingSegment seg = in.Receive().get();
46     if(seg.Empty()) break;
47     seg.Synchronize(stream);
48     Filter<<<...,stream>>>(seg.Ptr(), seg.Size(),
49                            s_out, out_size);
50     in.Release(seg, stream);
51     out.Send(s_out, out_size, stream);
52   }
53   out.Shutdown();
54 }
```

图 6 基于谓词的过滤的伪代码

第 12-13 行中的代码指定这些路由器和主机之间的链路，其中主机和 h2gpus 路由器之间的链路是在没有双重缓冲的情况下创建的。第 15-20 行中的代码使用双缓冲链路为每个 GPU 创建一个工作线程，并将输入段分散到设备（第 22 行）。在分配所有输入段时，分发器通过发送关机信号（第 23 行）通知它不再发送进一步的信息。处理的结果将在主机上获得（第 25-31 行），一旦所有 GPU 向 gpus2h 发送关机信号，程序就会停止，并表明不会再接收其他数据（第 27 行）。

两个路由器的路由策略都很简单，从所有可能的路由器目的地中选择第一个可用设备（第 36 行）。

在 GPU 端，每个设备异步处理传入消息（第 45 行）。将挂起的分段分配给设备后，它将与活动 GPU 完成流同步（第 47 行）并执行处理（第 48 行）。第 50 行将命令排队到流中，这会释放输入缓冲区以供后来的数据使用。然后使用 out 将结果传输回主机（第 51 行）。当收到关机信号时，工作线程被终止（第 46 行），并将关机信号（第 53 行）发送到 gpus2h。

内存一致性和所有权由 Groute 中的程序员进行维护。链路/路由器模型不定义全局地址空间或远程内存访问操作，而是充当分布式内存环境。在本文中，通过模型实现的算法和高级异步对象（如分布式工作清单）定义了所有权策略，而低级构造则提供了有效的通信和消息路由。

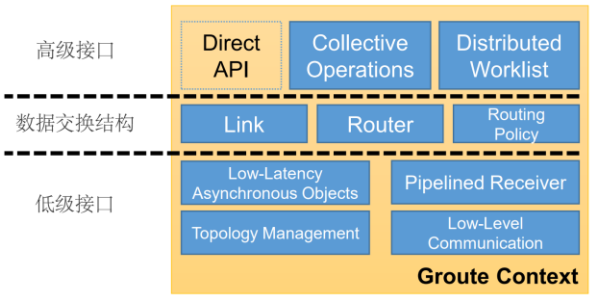


图 7 Groute 库

4. 实施细节

我们通过在标准 C++和 CUDA 上实现精简运行时环境[2]来实现 Groute 编程模型，从而实现异步多 GPU 编程。环境由三层组成，如图 7 所示。底层包含节点拓扑和 GPU 间通信的低级管理，中间层实现 Groute 通信构造，使用拓扑优化内存传输路径，顶层实现异步常规和不规则应用中常用的高级操作。为了直接控制系统，程序员可以手动访问每个层。在本节的其余部分中，我们将详细介绍 Groute 中每个层的实现。

4.1 底层接口

低级层以第 2 部分的结论为基础，提供程序员可访问的接口，以实现高效的点对点传输。具体而言，该层提供低级通信 API 以减少延迟;流水线接收机，增加计算通信重叠;节点互连层次结构内省的拓扑管理;和低延迟异步对象，以减轻系统开销。

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

在低级接口之上，使用流水线接收器对象进行抽象异步通信，该对象通过使用双缓冲和三缓冲[35]有效地利用了两个内存复制引擎和计算引擎（图 4）。多缓冲的实现对读取缓冲区进行分配，排队虚拟的"未来读取操作"。将数据发送到流水线接收方时，将从队列中删除读取操作，并将相应的缓冲区分配方式给发送方。同时，读取器会收到传入的挂起操作的通知，可以使用接收流等待或异步操作。

异步程序通常依赖于大量细粒度（非批量）同步点和实时内存分配来正常运行。为了最大限度地减少产生的驱动程序开销和非自愿的系统层级的同步，Groute 提供了几个低延迟异步对象，其中包括事件池(Event-Pool)和未来事件（Event-Futures）。Event-Pool 有助于通过预先分配的方式创建许多短期事件;而 Event-Futures 是实现未来/承诺模式[18]的可等待对象，用于维护 CPU 和 GPU 之间的两层同步。特别是，事件未来处理已知将在将来记录但尚未创建的 GPU 事件。这些对象用作排队各种操作（如设备（CPU 和 GPU）的未来接收操作）的主要构建基块，并且可以与 CPU 线程或 GPU 流同步。

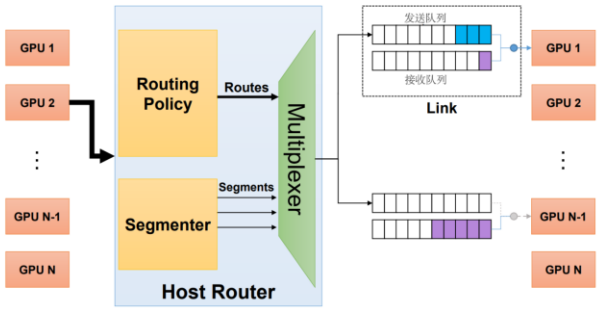


图 8 主机控制路由器示意图

4.2 通信与调度

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

路由器的内部结构和工作流程如图 8 所示。该图显示路由器包含三个主要组件。Segmenter 组件控制根据目标设备功能将消息分解为分段；路由策略控制邮件目的地；多路复用器 Multiplexer 负责将消息分配给可用的 GPU。

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

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

4.3 分布式工作列表

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

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

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

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

图 9 从单个设备的角度说明了 Groute 中分布式工作列表的实现。如图所示，工作列表是通过单个系统范围的路由器实现的。为了支持高效的全通通信，默认情况下，环形拓扑用于路由策略（有关评估，请参见第 2 节）。除路由器外，每个设备还包含一个本地管理的工作列表，该工作列表由一个或多个用于本地任务的多生产者-单消费者队列组成。该实现由每个设备两个线程组成：工作线程和接收器线程，它们控制 GPU 间通信和工作项循环。

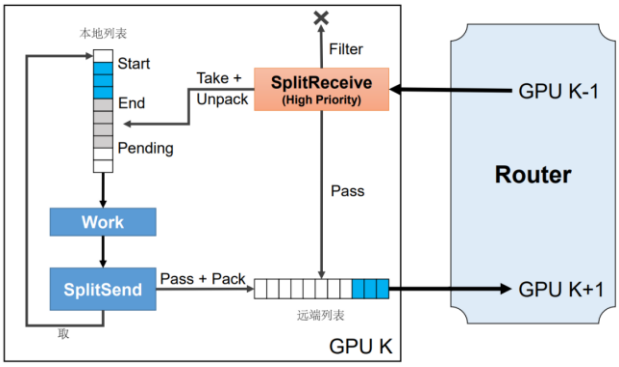


图 9 分布式工作清单实现

在环形拓扑中，图 9 中所示的工作流实现方式如下：每个设备从环拓扑的前一个设备接收信息。然后将接收到的数据进行过滤和分离（SplitReceptive），将不相关的信息传递到下一个设备。与当前设备相关的项目被解压缩并"推送"到其本地工作列表中，向工作线程发出新工作即将到来的信号。同时，工作线程处理现有工作项，将生成的项分离到本地和远程工作（SplitSend），并根据需要打包传出信息。请注意，SplitReceive 内核在单独的高优先级流上排队。这会导致内核在现有工作处理期间被调度，从而提高系统的性能和响应能力。

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

为了实现本地工作列表队列，Groute 使用 GPUbased 无锁循环缓冲区。此类缓冲区对于异步应用程序非常有用，因为它们消除了在运行时动态分配缓冲区的需要。

如图 9 所示，每个工作列表队列由多个生产者

和一个使用者组成。我们的实现包含一个内存缓冲区和三个字段：开始、结束和挂起。工作消耗是通过原子增加起始字段来执行的。为避免消耗未就绪的项目，通过原子方式增加挂起字段来控制生产，从而在缓冲区中保留空间。在生产者完成追加其工作后，end 将增加一个写入器线程，与挂起进行同步。

表 2 分布式工作列表程序员回调函数

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

对循环缓冲区的其他优化在 Groute 中完成。如果使用 GPU 流也产生工作（如在 SplitSend 中），则工作将通过预置信息（即减少开始）的方式推送到队列，从而避免生产者冲突。还值得注意的是，循环缓冲区使用 warp-aggregated atomics[4]，通过将原子操作的数量限制为每个 warp 来提高追加工作的效率。

4.4 软优先级调度

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

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

缓解此问题的一种方法是为每个工作项分配软优先级，将怀疑是"无用工作"生成器的项推迟到稍后阶段，在该阶段中它们可能会被过滤掉[17]。

在 Groute 中，软优先级调度是使用程序员提供的 GetPrio 回调实现的。在应用程序的运行期间，仅处理高优先级工作项，其中优先级阈值由系统范围的共识决定。完成所有可处理项目后，系统将修改阈值，分布式工作列表将处理每个设备上的延迟项目。正如我们将在第 5 节中所示，使用软优先级调度可减少中间工作量，从而提高整体性能。

4.5 基于工作列表的图算法

使用异步广度优先搜索（BFS），我们说明了如何使用分布式工作列表实现图遍历算法（如单源最短路径（SSSP）和 PageRank（PR）。

在 BFS 中，输入图以压缩稀疏行（CSR）矩阵格式给出。该图首先在可用设备之间进行分区，其中每个设备静态维护连续内存段的所有权，对应于顶点的一个子集。

当 BFS 处理开始时，主机会将单个工作项排入工作列表（源顶点）。Groute 确保将初始工作项发送到其所有者设备，在该设备中，通过将顶点值（即级别）设置为零并为每个相邻顶点创建级别为 1 的工作项来处理它。如果相邻顶点归处理设备所有，则该顶点将排队到设备本地工作列表（图 9 的左上角部分）。否则，它将通过分布式工作列表异步传播，直到另一个设备声明对工作项的所有权。级别值也会传播。原子操作用于检查接收的值是否较低，更新顶点的值并通过后续的工作项处理将 level+1 传播到相邻顶点。遍历所有相关边后，工作列表变为空，算法结束。

拥有的顶点之外，每个设备还存储其"halo"顶点的本地副本，即其他设备拥有的相邻顶点。halo 顶点的级别只能由本地设备更新，并且用于跳过发送不相关的更新，从而节省设备间的通信。由于 BFS 的单调性质，跳过此类更新不会改变算法的行为，即实际顶点值等于或低于 halo 值。

表 3 性能比较

Graph	BFS [ms]			SSSP [ms]		PR [ms]		CC [ms]	
	Gunrock	B40C	Groute	Gunrock	Groute	Gunrock	Groute	Gunrock	Groute
USA	617.85 (1)	**56.83** (1)	128.38 (2)	60,656.91 (8)	**725.93** (3)	1,394.25 (1)	**167.89** (8)	335.65 (1)	**15.11** (5)
OSM-eur-k	3,191.78 (1)	2,177.1 (2)	**616.4** (5)	874,083.5 (4)	**3,513.29** (8)	—	**1,045.33** (8)	—	**160.96** (4)
soc-LiveJournal1	99.11 (2)	**14.07** (4)	24.96 (6)	83.36 (5)	**30.98** (6)	2,782.06 (1)	**371.71** (5)	110.05 (1)	**14.19** (2)
twitter	—	—	**713.6** (8)	1,310.7 (7)	**649.2** (8)	—	**38,549.27** (1)	—	**384.13** (8)
kron21.sym	156.68 (3)	—	**46.55** (7)	**208.43** (2)	213.92 (8)	9,800.43 (1)	**5,342.73** (1)	—	**13.86** (8)

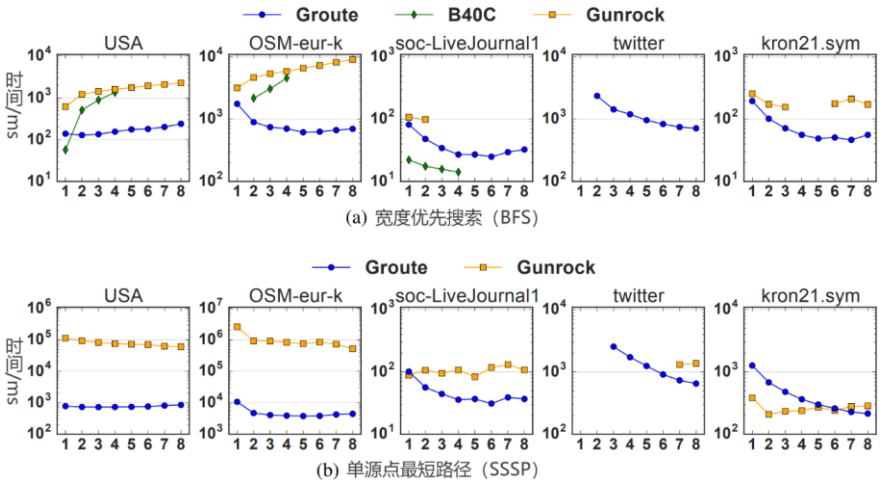


图 10 遍历算法时序（越低越好）

5. 性能评估

该节评估使用 Groute 实现的五种算法的性能：

- **广度优先搜索(BFS)**：遍历给定源节点的图形，输出从源节点遍历到每个目标节点的边数。实现是基于推送和数据驱动的，即使用分布式工作清单。。
- **单源最短路径 （SSSP）**：查找从源节点到所有其他节点的最短路径（使用边缘权重）。实现是基于推送和数据驱动的。
- **PageRank （PR）**：使用基于工作列表的算法计算给定图形的所有节点的 PageRank 度量值[34]。实现是本文中提出的基于推送的变体。
- **连接组件（CC）**：计算给定图形中连接的组件数。实现是基于拓扑的，使用两个路由器，如下所述。
- **基于谓词的过滤（PBF）**：根据给定条件筛选元素数组。GPU 内核实现基于翘曲聚合原子 [4]。

Groute 与多 GPU 并行图算法的两个基准实现进行了比较：Gunrock（版本 0.3.1）和 Back40Computing（B40C）。Gunrock [26] 是一个图形分析库，包含各种图形算法的高度优化实现。Gunrock 使用这些算法的多 GPU 实现使用批量同步并行性。由 Merrill 等人[20]撰写的 B40C 包含最先进的硬编码 BFS 实现，通过对等 GPU 之间的直接内存访问实现多 GPU 处理。

包括 Groute 在内的实现都包含以下内核优化：Warp 聚合原子操作、用于线程间通信的基于 Warp 的集合，以及用于利用嵌套并行性的 warp 和线程块级别的 GPU 内部负载平衡 [25]。此外，Groute 的异步模型允许我们执行内核融合（第 5.2 节）。

表 3 总结了 BFS、SSSP、PR 和 CC 的最佳运行时间，括号中列出了用于实现最高性能的 GPU 数量。使用 Groute 的异步实现显然优于批量同步实现，有时甚至优于硬编码版本。

我们的实验设置由两种节点类型组成。第一个是 8-GPU 服务器，由四个双板 NVIDIA Tesla M60（Maxwell 架构）卡组成，每个卡包含 16 个 MP，具有 128 个内核;和两个八核英特尔至强 E5-2630 v3 CPU。总线拓扑如图 1 所示，PCI-Express 交换机的每个 CPU 有 2 个 QPI 链路，每个链路 8GT/s。

二个节点类型是 2-GPU 异构服务器，其中包含一个 Quadro M4000 GPU（Maxwell，13 个 MP，128 个内核）;一个特斯拉 K40c GPU（开普勒架构，15MP，192 个内核）;和一个六核英特尔至强 E5-2630 CPU，总共 2 个 QPI 链路，每个链路 7.2GT/s。

表 4 图属性参数

名称	节点数	边数	平均度数	最大度数	大小 (GB)
路线图					
USA [1]	24M	58M	2.41	9	0.62
OSM-eur-k [3]	174M	348M	2.00	15	3.90
社交网络					
soc-LiveJournal1[10]	5M	69M	14.23	20,293	0.56
twitter [8]	51M	1,963M	38.37	779,958	16.00
合成图表					
kron21.sym [5]	2M	182M	86.82	213,904	1.40

表 4 列出了评估中使用的输入图的数据集信息和统计数据。除 kron21.sym 和 twitter 外，所有图都使用从 METIS[15]获得的边切割在 GPU 之间进行分区。 METIS 无法对这两个图进行分区，因此我们只需在 GPU 之间平均划分节点数组。

5.1 强大的扩展性

图 10 显示了两种图形遍历算法（BFS 和 SSSP）的绝对运行时，它们在 1 到 8 个 GPU 上运行，并与上述框架进行了比较。缺少数据点表示由于崩溃、内存不足故障或输出不正确（与外部生成的结果相比）而导致的运行失败。

总体而言，观察到在通信密集型的 BFS 和 SSSP 中，当传输超出单个 4-GPU 四联体时，总线拓扑开始影响应用程序的性能。虽然 Groute 通过优化通信路径（第 2 节）缓解了这些问题，但这种现象仍然可以在高度图中看到，例如 BFS、SSSP 和 PR 中的 soc-LiveJournal1。

5.1.1 广度优先搜索

图 10a 比较了 Groute 与 Gunrock 和 B40C。B40C 的多 GPU 实现需要所有设备之间的直接内存访问，因此最多只能运行 4 个 GPU。此外，B40C 不使用 METIS 分区，在 twitter 和 kron21.sym 上失败，并且在 OSM-eur-k 的单 GPU 版本上耗尽了内存。BFS 的 Gunrock 实现在所有 twitter 输入上都耗尽了内存，并在 kron21.sym 和 soc-LiveJournal1 上产生了不正确的结果。

该图显示，Groute 在所有情况下的表现都优于 Gunrock，道路网络（USA，OSM-eur-k）有了显着改善。这是由于异步处理启用了内核融合优化，这大大减少了高直径图中的内核启动开销（第 5.2 节）。

Groute 在多个 GPU 的道路网络上的表现也优于 B40C。但是，B40C 在 USA 输入端的一个 GPU 上更快，并且在 soc-LiveJournal1 输入端的表现始终优于 Groute 和 Gunrock。B40C 的 BFS 实现经过高度优化，包含一个混合实现，可以在不同的内核之间切换，如他们的论文[20]所述，由于其高度专业化的性质，我们没有实现这种优化。

5.1.2 单源最短路径

图 10b 显示了 Groute 中 SSSP 的强缩放。在图中，我们看到多 GPU 缩放模式与 BFS 类似。Groute 的多 GPU 可扩展性在大型图形（如 twitter 等）中尤其明显，即使在使用超过 4 个 GPU 时，性能也会提高。缺少 twitter 结果是由于内存不足造成的。

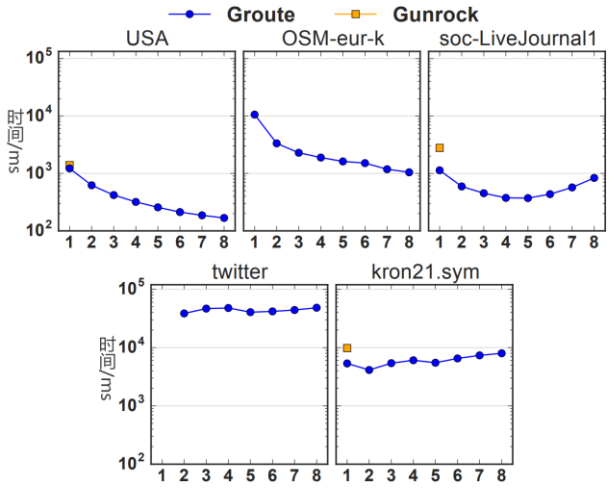


图 11 PageRank 执行时间

除了 kron21.sym 输入之外，Groute 的性能优于 Gunrock（或在 socLiveJournal1、单 GPU 上匹配）。经过深入检查，发现 Groute 中的异步实现导致执行的原子操作数量激增，从而增加了内存争用和迭代时间。

5.1.3 PageRank

PageRank（PR）的性能如图 11 所示。在单个 GPU 的情况下，结果与 Gunrock 进行比较，因为 Gunrock 的多 GPU PageRank 的评估版本产生了不正确的结果。此外，Twitter 图形不适合 1 个 GPU 的内存。

与 BFS 和 SSSP 相反，PR 是一个计算密集型问题。此外，PR 首先同时处理所有节点，因此每个 GPU 都可以得到充分利用。请注意，在图中，Groute 在所有输入上的表现都优于 Gunrock。此外，由于每个设备上执行的独立工作量以及 Groute 隐藏的通信延迟，两个道路网络都会在单 GPU 版本上生成多 GPU 扩展。在 soc-LiveJournal1 中，在单个四联 GPU 中也观察到了相同的效果。

特别是，当计算与通信的比率很高（即每次计算的通信较少）时，可以获得最佳的缩放结果。例如，在 USA 与 Groute 一起运行 PageRank，在 8 个 GPU 上比一个 GPU 产生 7.28 倍的加速，在所有多 GPU 配置上都表现出近乎线性的加速效果。

随着计算与通信比率的降低，扩展变得不那么重要，例如 socLiveJournal1，它在 4 个 GPU 上比 1 个 GPU 的速度提高了 3.02 倍。此外，在 twitter 和 kron21.sym 中几乎没有观察到加速，两者都是随机分区的（即没有 METIS）并且高度互连。

5.1.4 连接组件

我们实现了多 GPU 连接组件（CC）的拓扑驱动[27]变体，该变体不使用工作列表，以演示 Groute 异步通信构造的表现力。CC 的性能如图 12 所示。

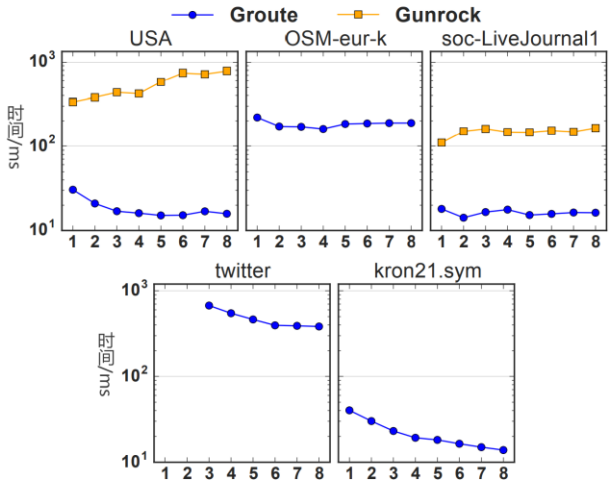


图 12 连接组件性能表现

图 13 说明了在 Groute 上由指针跳转拓扑驱动的 CC 的数据流图。在此版本中，输入图形表示形式是边列表。边分发到 GPU，每个 GPU 在其给定的图形子集中记录每个顶点的父组件。一旦 GPU 在本地聚合，其结果就会与其他 GPU 的结果合并（使用称为 Hook 和 Compress 的操作）逐渐收敛到全局组件列表。

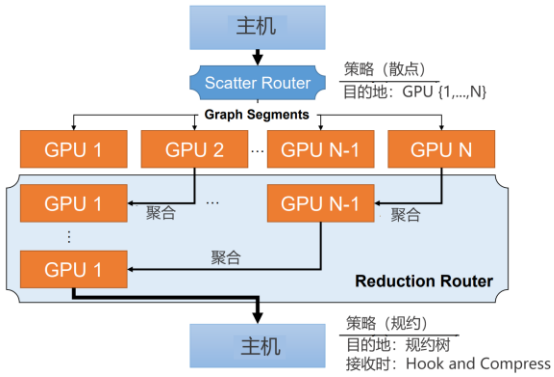


图 13 连接组件路由器结构

使用 Groute，我们创建了一个路由器，将边动态分发到多个段中的 GPU，计算并聚合每个段的本地计算结果。完成后，每个 GPU 都会使用额外的缩减路由器将其结果与其他 GPU 结果合并。根据测量的 8-GPU 节点的拓扑，减少作为并发分层操作实现。特别是，每个 GPU 将其结果与指定的"同级"GPU 合并，直到到达第一个 GPU，后者将信息发送回主机。此实现中使用的单个 GPU 内核基于 Soman 等人描述的指针跳跃方法的最先进的自适应变体 [31][29]。

图 12 中的结果表明，与 Gunrock 中的实现相比，使用异步拓扑驱动的变体在原始性能和可扩展性方面都非常有优势。具体来说，Groute 在 1 个和 8 个 GPU 上分别比 USA 算例的 Gunrock 产生 10.99 倍和 49.6 倍的加速。此外，Groute 在 kron21.sym 算例中实现了高达 2.9 倍(8GPU 超过 1)的强大扩展。

图中另一个明显的优势是内存消耗。由于 Groute 实现不使用分布式工作列表，因此测试的多 GPU 系统能够在 OSM-eur-k、twitter 和 kron21.sym 等大规模图上计算 CC。反之，Gunrock 在这三个输入上的所有 GPU 配置都耗尽了内存。

5.2 分布式工作清单性能

我们的分布式工作列表使用两种优化——软优先级调度和工作线程内核融合来显着提高异步算法的性能。

如第 4.4 节所述，朴素异步不规则应用程序会从滞后设备传播中间值，从而导致冗余计算使工作量增加。使用 METIS 可以在一定程度上减轻这种影响，因为它减少了分区之间的路径数。但是，我们的实验表明，即使使用良好的分区，通过使用软优先级调度程序来延迟可能的冗余工作也可以获得明显更好的性能。

表 5 分布式工作列表 SSSP 性能

图算例	GPU	软优先级调度	融合计算
soc-LiveJournal1	1	1.03x	1.03x
	2	0.95x	0.95x
	4	1.29x	1.31x
	8	1.29x	1.09x
kron21.sym	1	1.02x	1.01x
	2	1.10x	1.09x
	4	1.58x	1.57x
	8	1.45x	1.48x
USA	1	71.36x	166.27x
	2	58.11x	142.57x
	4	34.59x	89.94x
	8	16.22x	39.34x

表 5 比较了软优先级调度程序和融合的工作线程内核与 Groute 分布式工作列表的未优化版本的性能差距。在表中，我们看到两个版本的表现始终优于原始实现，但 soc-LiveJournal1 除外（它在 2 个 GPU 上减慢了 5%）。最引人注目的结果可以在路线图中发现，在路线图中，我们看到性能提高了两个数量级。

软优先级通过减少执行的"无用"工作量来提高性能。无用工作的这种增加可能是戏剧性的。例如，SSSP 在单个 GPU 上完整执行 USA 图上的 13，998M 个工作项。如果没有软优先级，则在四个 GPU 上增加到 15，865M 个工作项。使用软优先级，四个 GPU

版本总共只能执行 59M 个工作项。对于 SSSP，这种减少与使用具有优先级（如 SSSP NearFar [9]）的 SSSP 算法获得的减少相当。但是，BFS 没有优先级的概念，但也表现出相同的效果。USA 的单 GPU BFS 可执行 2390 万个工作项。如果没有软优先级，这会增加到四个 GPU 上的 4，244M 工作项（METIS 为 27M）。使用软优先级，这减少了四个 GPU 上的 134M 个工作项目（METIS 时为 2410 万个）。

另一方面，内核融合解决了路线图等图形所表现出的问题，这些图表在每次内核调用（约 19 微秒）时都会创建较小的工作负载，从而导致 GPU 未得到充分利用并增加通信管理开销。我们增强了工作线程内核以包含整个控制流，并使用 CPU 和 GPU 共享的标志与主机和其他 GPU 进行通信。这包括接收传入的信息信号、确定工作项优先级、处理一批工作项、运行 SplitSend（第 4.3 节）以及向路由器发出传出信息的信号。通过执行此内核融合，可以减少许多 CPU-GPU 往返，从而提高系统的整体性能。在实践中，Groute 中的内核融合会增加每个内核调用执行的工作量，这需要 10 到 100 毫秒。请注意，CPU-GPU 往返次数的减少会导致这两种优化同时改善单个 GPU 的运行时间。

5.3 负载均衡

表 6 从图 6 中测量了异构 2-GPU 节点上的 PBF 实现的性能，筛选了 250MB 的数据。每个 GPU 的运行时都显示静态调度（前三行）和 Groute 的"第一个可用设备"路由策略（底部三行）。

表 6 异构 PBF 性能

调度	GPU 类型	处理的元素	耗时(ms)
静态	Tesla K40c	12.6M	5.73 ms
	Quadro M4000	13.6M	27.04 ms
	Total Time	-	27.37 ms
Groute	Tesla K40c	20.9M	8.84 ms
	Quadro M4000	5.2M	10.90 ms
	Total Time	-	11.26 ms

该表显示，Groute 向速度较快的 Tesla K40c 分配的任务比速度较慢的 Quadro M4000 多 4 倍，从而实现了更好的负载平衡并减少了整体运行时间。请注意，在 Groute 中观察到的 2 毫秒时差在调度量程内，它比 Quadro M4000 上单个内核的运行时间短。

6.相关工作

本文所呈现的链路/路由器编程模型可以看作是发布/订阅设计模式 [11] 的近亲，在该模式中，端点订阅其他端点发布到的特定通道。链路/路由器模型与此模型的不同之处在于定义了通用策略，与具名通道相比，通用策略更适合多 GPU 节点上的低延迟通信。

最近，已经提出了简化编程并提供可重用机制的多 GPU 框架。值得注意的例子包括：NCCL[24]，它在单个节点上实现集体操作；MGPU [28]，将任务分区简化为多个 GPU；MAPSMulti [6]，它提出了一种基于内存访问模式的可扩展编程模型。由于传统的批量同步使用多 GPU 节点，这些库专注于常规计算（如模板运算符）的有效实现，而不是不规则算法。

数据驱动的图算法实现使用工作列表进行处理[22]。在一般情况下，这些实现被发现比 GPU 上的拓扑驱动对应物更快[21]。在本文中，这些结果促使人们使用分布式工作列表实施图形分析。

已经研究了其他异步图形处理框架。Galois[23]提出了一种用于异步多核 CPU 处理的工作窃取调度程序。GTS[16]中还提出了在单个 GPU（使用多个流）上的并发图形分析，这表明这种类型的编程对于单 GPU 应用程序也很有前途。已经提出了额外的单 GPU[7，12，14，33]和多 GPU[20，26]图形分析库。但是，与我们的异步方法相反，这些实现都利用了批量同步并行性。

第 5 节中提出的分布式工作清单内核融合优化类似于 Steinberger 等人提出的 Megakernel 单 GPU 方法[30]，该方法还将部分控制流传输到 GPU。

7. 结论

本文介绍了一种用于异步多 GPU 应用程序开发的可扩展编程抽象和运行时环境。对多 GPU 节点结构的深入研究表明，创建此类应用程序需要仔细调整通信拓扑和工作负载处理，特别是在不规则算法中，滞后信息可能对扩展产生重大影响。然后，该论文表明，编程抽象简单而富有表现力，能够有效地实现复杂的图分析算法，显示出强大的扩展结果。

这项研究可以向几个方向扩展。首先，链路/路由器抽象概念可以推广到其他非 GPU 架构以及分布式系统。其次，大多数路由器控制流依赖于基于

主机的决策。与工作线程内核融合类似，将这些决策移动到基于设备的路由器可能会通过进一步减少 CPU-GPU 副本来降低系统开销。第三，多 GPU 遍历算法中的负载平衡可以通过采用异步工作窃取调度程序和动态更改节点所有权来改进。

致谢

这项研究得到了德国研究基金会（DFG）优先计划 1648"百亿亿次级计算软件"（SPP-EXA）的研究项目 FFMK 的支持;NSF 拨款 1218568，1337281，1406355 和 1618425;由 DARPA BRASS 合同 750-16-2-0004;以及 NVIDIA 的设备赠款。

引用

[1] 9th DIMACS Implementation Challenge. URL http://www.dis.uniroma1.it/challenge9/ download.shtml.

[2] Groute Runtime Environment Source Code. URL http://www.github.com/groute/groute.

[3] Karlsruhe Institute of Technology, OSM Europe Graph, 2014. URL http://i11www.iti.uni-karlsruhe.de/resou rces/roadgraphs.php.

[4] A. Adinetz. Optimized filtering with warp-aggregated atomics. 2014. URL http://devblogs.nvidia.com/parallelfor all/cuda-pro-tip-optimized-filteringwarp-aggregated-atomics/.

[5] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings, volume 588 of Contemporary Mathematics, 2013. American Mathematical Society.

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

[7] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In Workload Characterization (IISWC), 2012 IEEE International Symposium on, pages 141– 151, 2012.

[8] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi. Measuring user influence in Twitter: The million follower fallacy. ICWSM, 10(10-17):30, 2010.

[9] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Workefficient parallel GPU methods for single-source shortes paths. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pages 349–359, 2014.

[10] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. ACM Trans. Math. Softw., 38(1):1:1–1:25, 2011.

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. ACM Comput. Surv., 35(2):114–131, 2003.

[12] A. Gharaibeh, L. Beltrao Costa, E. Santos-Neto, and M. Ri- ˜ peanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 345–354. ACM, 2012.

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

[14] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, pages 267–276. ACM, 2011.

[15] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput., 20(1):359–392, 1998.

[16] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pages 447–461. ACM, 2016.

[17] A. Lenharth, D. Nguyen, and K. Pingali. Priority queues are not good concurrent priority schedulers. In Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings, pages 209–221. Springer Berlin Heidelberg, 2015.

[18] B. Liskov and L. Shrira. Promises: Linguistic

support for efficient asynchronous procedure calls in distributed systems. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, pages 260–267, 1988.
[19] E. Mej´ıa-Roa, D. Tabas-Madrid, J. Setoain, C. Garc´ıa, F. Tirado, and A. Pascual-Montano. NMF-mGPU: nonnegative matrix factorization on multi-GPU systems. BMC Bioinformatics, 16(1):43, 2015.
[20] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 117–128, 2012.
[21] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on GPUs. In Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pages 463–474, 2013.
[22] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on GPUs. In ACM SIGPLAN Notices, volume 48, pages 147– 156. ACM, 2013.
[23] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In Proceedings of the TwentyFourth ACM Symposium on Operating Systems Principles, SOSP '13, pages 456–471, 2013.
[24] NVIDIA. NVIDIA Collective Communication Library (NCCL), 2016. URL http://www.github.com/NVIDIA/nccl/.
[25] S. Pai and K. Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '16. ACM, 2016.
[26] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens. MultiGPU graph analytics. CoRR, abs/1504.04804, 2015. URL http://arxiv.org/abs/1504.04804.
[27] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos, and X. Sui. The tao of ´ parallelism in algorithms. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 12–25. ACM, 2011.
[28] S. Schaetz and M. Uecker. A multi-GPU programming library for real-time applications. In Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Part I, ICA3PP'12, pages 114–128. SpringerVerlag, 2012.
[29] J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pages 1–8, 2010.
[30] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the GPU. ACM Trans. Graph., 33(6): 228:1–228:11, 2014.
[31] M. Sutton, T. Ben-Nun, A. Barak, S. Pai, and K. Pingali. Adaptive work-efficient connected components on the GPU. CoRR, abs/1612.01178, 2016. URL http://arxiv.org/abs/1612.01178.
[32] L. G. Valiant. A bridging model for parallel computation. Commun. ACM, 33(8):103–111, 1990.
[33] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the GPU. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 265–266, 2015.
[34] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali. Scalable data-driven PageRank: Algorithms, system issues, and lessons learned. In L. J. Traff, S. Hunold, and F. Versaci, ed- ¨ itors, Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Proceedings, pages 438–450. Springer Berlin Heidelberg, 2015.
[35] D. Wilson. Triple buffering: Why we love it. 2009. URL http://www.anandtech.com/show/2794.

工件描述

A.1 摘要

该工件包含编译 Groute 可执行文件并重复本文结果所需的所有源代码。该软件包还包含 shell 脚本，

用于将图形和表格生成为 CSV、获取代码依赖项以及下载基准测试的输入图。

A.2 说明

A.2.1 检查表（工件元信息）

• 算法：微基准、广度优先搜索、单源最短路径、PageRank、连接组件、基于谓词的过滤。
• 编译：对 GPU 代码使用 CUDA (nvcc)，对主机代码使用 GCC (gcc, g++)。 两个编译器都使用以下标志：-O3 -DNDEBUG -std=c++11。
• 二进制：每个算法都有一个 CUDA 可执行文件。
• 数据集：转换为二进制 CSR (Galois) 格式的公开可用图表。
• 运行时环境：Debian 8 Linux 与 CMake 3.2、GCC 4.9.3 和 CUDA 7.5。 可选要求（用于与 Gunrock 进行比较）：Boost 1.55.0。
• 硬件：支持 CUDA 的 GPU，计算能力至少为 3.5。多 GPU 测试需要同一节点上的多个 GPU。
• 输出：代表每个图形和表格的 CSV 文件。
• 实验工作流程：克隆存储库、设置环境、运行生成图形的 shell 脚本、查看生成的 CSV 文件。
• 是否公开？：是的。

A.2.2 如何交付

包含所有 Groute 代码和图形生成脚本的 Git 存储库位于：http://www.github.com/groute/ppopp17-artifact

该存储库包含设置环境、获取和编译外部代码、下载数据集和运行图形生成器的 shell 脚本。请参阅第 A.3 和 A.4 节中的附加信息。

A.2.3 硬件依赖

要运行基于 GPU 的测试，Groute 需要一个或多个计算能力至少为 3.5 的 NVIDIA GPU。多 GPU 测试不会在单 GPU 节点上运行。对于表 6 中执行的测试，建议使用具有两个异构 GPU（即一个比另一个快）的系统来重复结果。

A.2.4 软件依赖

要编译 Groute，需要 CMake3.2、GCC4.9 和 CUDA7.5。更高版本可以使用，但作者未测试。上述存储库中包含的代码包括以下外部依赖项：METIS5.1.0、gflags2.1.2、MGBench1.01、NCCL1.2.3 和 Gunrock0.3.1（可选的）。

A.2.5 数据集

该工件的数据集包含论文表 4 中列出的所有图，并转换为 Galois CSR 二进制格式（.gr 文件）。数据集（压缩后 10.6GB，未压缩 29GB）由安装脚本自动下载。每个图形都位于一个单独的子目录中，并带有三个附加文件：

1.<graph>/<graphfile>.metadata：图属性的元数据（连通分量的数量，是否使用 METIS 分区，软优先级 delta）。
2. <graph>/bfs-<graphfile>.txt：外部生成的 BFS 结果（源节点：0），用于验证。
3. <graph>/sssp-<graphfile>.txt：外部 SSSP 结果（源节点：0），也用于验证。

A.3 安装

请按照以下说明进行操作：

• 从 https://github.com/groute/ppopp17-artifact.git 递归克隆 Git 存储库
• 运行 setup.sh。该脚本将自动编译 Groute 和外部代码。
• 当提示是否下载数据集时，回答“y”或“n”。可以稍后通过运行 dataset/download.sh 手动下载数据集。

A.4 实验工作流程

设置好环境后，要么运行 runall.sh 生成所有图形和表格，要么通过调用 figures/figureXX.sh 分别运行每个单独的图形，其中 XX 是图形编号。作为 shell 命令列表的整体工作流程如下：

```
$ git clone --recursive
  https://github.com/groute/ppopp17-artifact.git
$ cd ppopp17-artifact/
$ ./setup.sh
$ ./runall.sh
$ cat output/figure10b.csv
```

注意：Gunrock 执行（用于与 Groute 比较）默认情况下未启用。若要启用，请在运行图形生成脚本之前导出 RUN GUNROCK=1。

注意 2：每个单独的测试可能需要时间，但限制为 2 小时，以避免永远等待错误的应用程序。对于较慢的 GPU，可以通过将 TIMEOUT 变量从 2h 修改为不同的值（图/common.sh 中的第 4 行）来增加此超时。

A.5 评价和预期结果

在 output/中找到的结果采用逗号分隔值(CSV)格式，并代表论文中的每个图形和表格（例如，figure2a.csv、table6.csv）。在将结果插入 CSV 之前，至少 3 次运行对结果进行平均。在此过程中，结果

也将写入控制台。