Groute: Asynchronous Multi-GPU Programming Model with Applications to Large-scale Graph Processing

TAL BEN-NUN, ETH Zurich
MICHAEL SUTTON, The Hebrew University of Jerusalem
SREEPATHI PAI, University of Rochester
KESHAV PINGALI, The University of Texas at Austin

Nodes with multiple GPUs are becoming the platform of choice for high-performance computing. However, most applications are written using bulk-synchronous programming models, which may not be optimal for irregular algorithms that benefit from low-latency, asynchronous communication. This article proposes constructs for asynchronous multi-GPU programming and describes their implementation in a thin runtime environment called Groute. Groute also implements common collective operations and distributed work-lists, enabling the development of irregular applications without substantial programming effort. We demonstrate that this approach achieves state-of-the-art performance and exhibits strong scaling for a suite of irregular applications on eight-GPU and heterogeneous systems, yielding over 7× speedup for some algorithms.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**; • **Software and its engineering** → *Massively parallel systems*; Runtime environments;

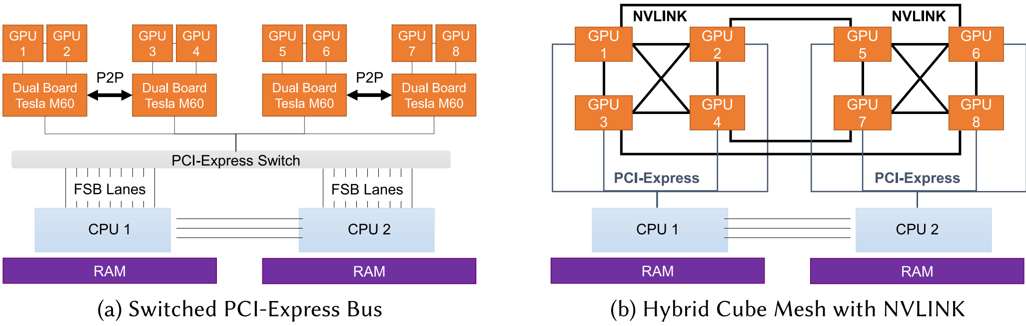
Additional Key Words and Phrases: Multi-GPU, asynchronous programming, irregular algorithms

ACM Reference format:
Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2020. Groute: Asynchronous Multi-GPU Programming Model with Applications to Large-scale Graph Processing. *ACM Trans. Parallel Comput.* 7, 3, Article 18 (June 2020), 27 pages.
https://doi.org/10.1145/3399730

1 MOTIVATION

Nodes with multiple attached accelerators are now ubiquitous in high-performance computing. In particular, Graphics Processing Units (GPUs) have become popular because of their hardware parallelism, scalable caching mechanisms, and balance between specialized instructions and general-purpose computing. Multi-GPU nodes consist of a host (CPUs) and several GPU devices linked

This research was supported by the German Research Foundation (DFG) Priority Program 1648 "Software for exascale Computing" (SPP-EXA), research project FFMK; NSF grants 1218568, 1337281, 1406355, and 1618425; by DARPA BRASS contract 750-16-2-0004; and an equipment grant from NVIDIA.
Authors' addresses: T. Ben-Nun, ETH Zurich, Department of Computer Science, ETH Zürich, Zürich, 8006, Switzerland; email: talbn@inf.ethz.ch; M. Sutton, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem 9190401, Israel; email: michael.sutton@mail.huji.ac.il; S. Pai, Department of Computer Science, University of Rochester, Rochester, NY 14627, USA; email: sree@cs.rochester.edu; K. Pingali, Institute for Computational and Engineering Sciences, The University of Texas at Austin, Austin, TX 78712-1229, USA; email: pingali@cs.utexas.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
2329-4949/2020/06-ART18 $15.00
https://doi.org/10.1145/3399730



(a) Switched PCI-Express Bus (b) Hybrid Cube Mesh with NVLINK

Fig. 1. Multi-GPU node schematics.

via a low-latency, high-throughput bus (see Figure 1). These interconnects allow parallel applications to exchange data efficiently and to take advantage of the combined computational power and memory size of the GPUs, but may vary substantially between node types.

Multi-GPU nodes are usually programmed using one of two methods. In the simple approach, each GPU is managed separately, using one process per device [19, 26]. Alternatively, a Bulk Synchronous Parallel (BSP) [42] programming model is used, in which applications are executed in rounds, and each round consists of local computation followed by global communication [6, 33]. The first approach is subject to overhead from various sources, such as the operating system, and requires a message-passing interface for communication. The BSP model, however, can introduce unnecessary serialization at the global barriers that implement round-based execution. Both programming methods may result in under-utilization of multi-GPU platforms, particularly for irregular applications, which may suffer from load imbalance and may have unpredictable communication patterns.

In principle, asynchronous programming models can reduce some of those problems, because unlike in round-based communication, processors can compute and communicate autonomously without waiting for other processors to reach global barriers. However, there are few applications that exploit asynchronous execution, since their development requires an in-depth knowledge of the underlying architecture and communication network and involves performing intricate adaptations to the code.

This article presents Groute, an asynchronous programming model and runtime environment [2] that can be used to develop a wide range of applications on multi-GPU systems. Based on concepts from low-level networking, Groute aims to overcome the programming complexity of asynchronous applications on multi-GPU and heterogeneous platforms. The communication constructs of Groute are simple, but they can be used to efficiently express programs that range from regular applications and BSP applications to nontrivial irregular algorithms. The asynchronous nature of the runtime environment also promotes load balancing, leading to better utilization of heterogeneous multi-GPU nodes.

This article is an extended version of previously published work [7], where we explain the concepts in greater detail, consider newer multi-GPU topologies, and elaborate on the evaluated algorithms, as well as scalability considerations. The main contributions are the following:

- We define abstract programming constructs for asynchronous execution and communication.
- We show that these constructs can be used to define a variety of algorithms, including regular and irregular parallel algorithms.

- We compare aspects of the performance of our implementations, using applications written in existing frameworks as benchmarks.
- We show that using Groute, it is possible to implement asynchronous applications that in most cases outperform state-of-the-art implementations, yielding up to 7.32× speedup on eight GPUs compared to a baseline execution on a single GPU.

2 MULTI-GPU NODE ARCHITECTURE

In general, the role of accelerators is to complement the available CPUs by allowing them to offload data-parallel portions of an application. The CPUs, in turn, are responsible for process management, communication, input/output tasks, memory transfers, and data pre/post-processing.

As illustrated in Figure 1, the CPUs and accelerators are connected to each other via a Front-Side Bus (FSB, implementations include QPI and HyperTransport). The FSB lanes, whose count is an indicator of the memory transfer bandwidth, are linked to an interconnect such as PCI-Express or NVLink that supports both CPU-GPU and GPU-GPU communications.

Due to limitations in the hardware layout, such as use of the same motherboard and power supply units, multi-GPU nodes typically consist of ∼1–25 GPUs. The topology of the CPUs, GPUs, and interconnect can vary between complete all-pair connections and a hierarchical switched topology, as shown in the figure. In the tree-topology shown in Figure 1(a), each quadruplet of GPUs (i.e., 1–4 and 5–8) can perform *direct* communication operations amongst themselves, but communications with the other quadruplet are *indirect* and thus slower. For example, GPUs 1 and 4 can perform direct communication, but data transfers from GPU 4 to 5 must pass through the interconnect. A switched interface allows each CPU to communicate with all GPUs at the same rate. In other configurations, CPUs are directly connected to their quadruplet of GPUs, which results in variable CPU-GPU bandwidth, depending on process placement.

The GPU architecture contains multiple memory copy engines, enabling simultaneous code execution and two-way (input/output) memory transfer. Below, we elaborate on the different ways concurrent copies can be used to efficiently communicate within a multi-GPU node.

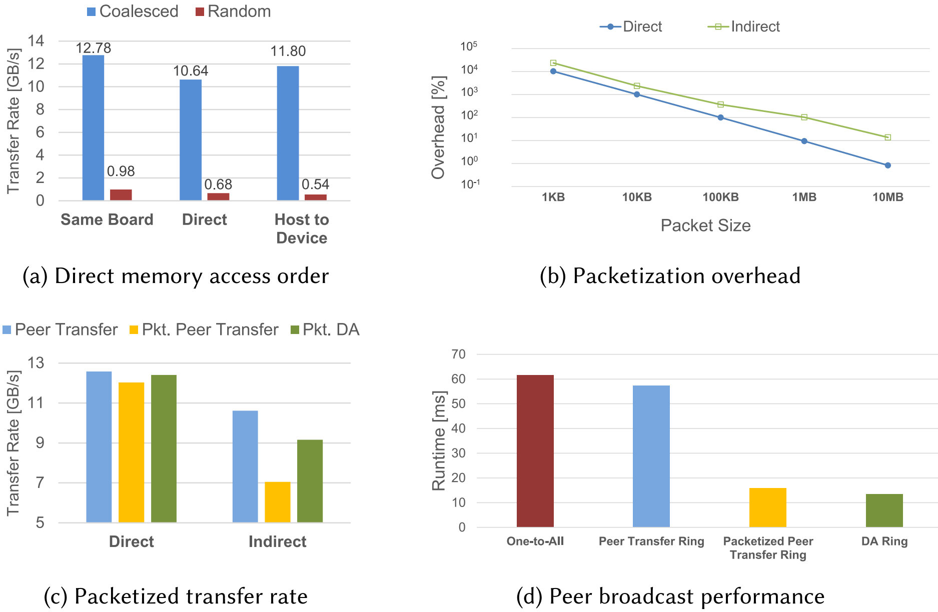
2.1 Inter-GPU Communication

Memory transfers among GPUs are provided by the vendor runtime via implicit and explicit interfaces. For the former, abstractions such as Unified and Managed Memory make use of virtual memory to perform copies, paging, and prefetching. With explicit copies, however, the user maintains full control over how and when memory is transferred. When exact memory access patterns are known, it is generally preferable to explicitly control memory movement, as prefetching may hurt memory-latency bound applications, for instance. For this reason, we focus below on explicit inter-GPU communication.

Explicit memory transfers among GPUs can either be initiated by the host or a device. Host-initiated memory transfer (*Peer Transfer*) is supported by explicit copy commands, whereas device-initiated memory transfer (*Direct Access*, DA) is implemented using inter-GPU memory accesses. Note that direct access to peer memory may not be available between all pairs of GPUs, depending on the bus topology. Access to pinned host memory, however, is possible from all GPUs.

Device-initiated memory transfers are implemented by virtual addressing, which maps all host and device memory to a single address space. While more flexible than peer transfers, DA performance is highly sensitive to memory alignment, coalescing, number of active threads, and order of access.

Using microbenchmarks (Figure 2), we measure 100 MB transfers, averaged over 100 trials, on theeight-GPU system from our experimental setup (see Section 5 for detailed specifications).



(a) Direct memory access order

(b) Packetization overhead

(c) Packetized transfer rate

(d) Peer broadcast performance

Fig. 2. Inter-GPU memory transfer microbenchmarks.

Figure 2(a) shows the transfer rate of device-initiated memory access on GPUs that reside in the same board, on different boards, and CPU-GPU communication. The figure demonstrates the two extremes of the DA spectrum—from tightly managed coalesced access (blue bars, left-hand side) to random, unmanaged access (red bars, right-hand side). Observe that coalesced access performs up to 21× better than random access. Also notice that the memory transfer rate correlates with the distance of the path in the topology. Due to the added level of dual-board GPUs (shown in Figure 1(a)), CPU-GPU transfer is faster than two different-board GPUs.

To support device-initiated transfers between GPUs that cannot access each other's memory, it is possible to perform a two-phase indirect copy. In indirect copy, the source GPU "pushes" information to host memory first, after which it is "pulled" by the destination GPU using host flags and system-wide memory fences for synchronization.

In topologies such as the one presented in Figure 1(a), GPUs can only transmit to one destination at a time. This hinders the responsiveness of an asynchronous system, especially when transferring large buffers. One way to resolve this issue is by dividing messages into packets, as in networking. Figure 2(b) presents the overhead of using packetized memory transfers as opposed to a single peer transfer. The figure shows that the overhead decreases linearly as the packet size increases, ranging between ~1% and 10% for 1–10 MB packets. This parameter can be tuned by individual applications to balance between latency and bandwidth.

Figure 2(c) compares the transfer rate of direct (push) and indirect (push/pull) transfers, showing that packetized device-initiated transfers and the fine-grained control is advantageous, even over the host-managed packetized peer transfers. Note that, since device-initiated memory access is written in user code, it is possible to perform additional data processing during transfer.

Another important aspect of multi-GPU communication is multiple source/destination transfers, as in collective operations. Due to the structure of the interconnect and memory copy engines, a naive application is likely to congest the bus. One approach, used in the NCCL library [31], creates a ring topology over the bus. In this approach, illustrated in Figure 3, each GPU transfers to

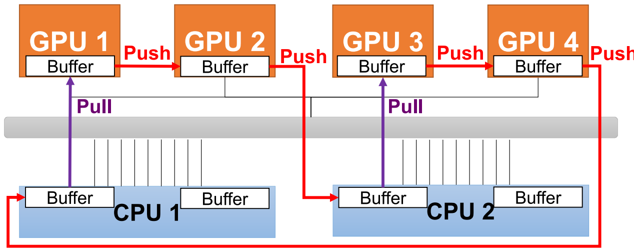


Fig. 3. DA Ring topology.

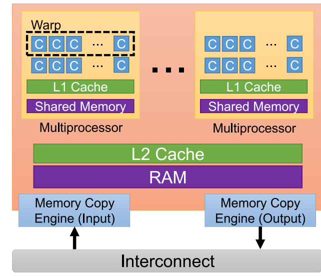


Fig. 4. Single GPU architecture.

one destination, communicating via direct or indirect device-initiated transfers. This ensures that both memory copy engines of every GPU are used, and the bus is fully utilized. Packetization is also used to begin transmitting information to the next GPU while it is being received from the previous, pipelining operations.

Figure 2(d) compares the performance of the different methods of implementing one-to-all GPU peer broadcast, ranging from seven asynchronous transfers from one source, through complete and packetized peer transfers, to the above DA Ring approach. The figure shows that the ring topology consistently outperforms separate direct copies. This can be attributed to the lower amount of indirect peer transfers (one peer transfer to the second quadruplet in ring vs. four in one-to-all). Additionally, packetization induces copy pipelining, which dramatically decreases the running time. DA Ring performs only slightly better than host-controlled ring transfer, consistent with the faster transfer rate in Figure 2(c).

2.2 GPU Programming Model

The structure of a single GPU device is depicted in Figure 4. As shown in the figure, each GPU contains a fixed set of multiprocessors (MPs) and a RAM unit (referred to as *global memory*). GPU procedures (*kernels*) run on the MPs in parallel by scheduling a *grid* of many threads, grouped to *thread-blocks*. Within each thread-block, which is assigned to a single MP, *warps* (usually composed of 32 threads) execute on the cores in lockstep. Additionally, threads in the same thread-block can synchronize and communicate via shared memory, as well as use the automatically managed L1 and L2 caches. To support concurrent memory writes, atomic operations are defined on both shared and global memory.

Kernel invocation and host-initiated memory transfers are performed via command queues (*streams*), which can be used to express task parallelism. Stream synchronization between one or more GPUs is usually performed using *events*, which are recorded on one stream and waited for on another.

The GPU scheduler dispatches kernels by thread-blocks, enabling multiple-stream concurrency on the same GPU by scheduling other kernels' thread-blocks when there are no more thread-blocks to schedule for a running kernel. However, high-priority streams allow application developers to immediately invoke kernels, scheduling thread-blocks from a new kernel prior to thread-blocks from a running kernel.

While the stream/event constructs provide fine-grained control over kernel scheduling, difficulties arise when programming higher-level functionality that involves multiple GPUs. In the following section, we present a programming abstraction that complements the existing model to facilitate multi-GPU development, using insights from the microbenchmarks to minimize communication latency.

Table 1. Groute Programming Interface

Construct	Description
	Base Constructs
Context	Singleton that represents the runtime environment.
Endpoint	An entity that can communicate (e.g., GPU, CPU, Router).
Segment	Object that encapsulates a buffer, its size, and metadata.
	Communication Setup
`Link(Endpoint src, Endpoint dst, int packet_size, int num_buffers)`	Connects `src` to `dst`, using multiple-buffering with `num_buffers` buffers and packets of size `packet_size`.
`Router(int num_inputs, int num_outputs, RoutingPolicy policy)`	Connects multiple Endpoints together, enabling dynamic communication.
	Communication Scheduling
`EndpointList RoutingPolicy(Segment message, Endpoint source, EndpointList router_dst)`	A programmer-defined function that decides possible message destinations based on sending Endpoint, and a list of the Router destination Endpoints. The Router will select destinations by availability.
	Asynchronous Objects
PendingSegment	Represents a Segment that is currently being received.
`DistributedWorklist(Endpoint src, EndpointList workers)`	Manages all-to-all work-item distribution, consists of a Router and per-GPU links.

3 GROUTE PROGRAMMING MODEL

The Groute programming model provides several constructs to facilitate asynchronous multi-GPU programming. Table 1 lists a summary of these constructs and their programming interface. The associated runtime environment [2] is implemented over CUDA, intended for NVIDIA-based multi-GPU nodes.

Groute applications consist of two phases: dataflow graph construction and asynchronous computation. A Groute program begins by specifying the dataflow graph of the computation. Nodes in this directed graph, which we call *endpoints*, represent either (i) physical devices such as CPUs and GPUs, or (ii) virtual devices called *routers*, which are abstractions that implement complex patterns of communication. Edges in the dataflow graph represent communication *links* between endpoints and can be created as long as there are no self-loops (endpoints connected directly to themselves) nor multiple identical edges as in multigraphs (i.e., an individual router can only have one outgoing edge to the same endpoint). To support multitasking on the same device, multiple virtual endpoints can be created from the same physical hardware. Another approach to multitasking is to create multiple dataflow graphs by way of additional routers, thereby enabling additional links between a pair of devices.

`Send` and `Receive` methods permit endpoints to send and receive data on a link; upon receipt of data, an endpoint may act upon it using a callback. When a router is created, a *routing policy* is specified by the programmer to determine the behavior when an input is received. For example,

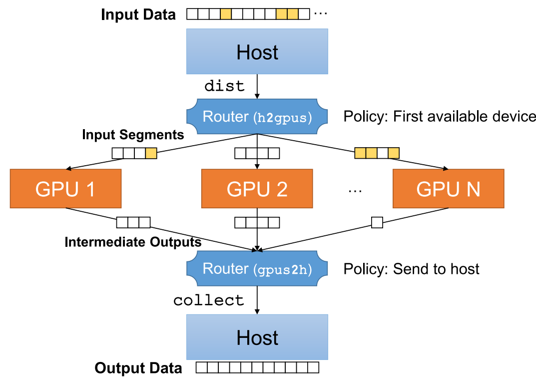


Fig. 5. Predicate-based filtering dataflow graph.

the input can be sent to a single endpoint or to a subset of endpoints according to their availability. When a *link* is created, the packetization and multiple-buffering policies for that link are specified (Section 2).

To demonstrate the Groute model, we describe the implementation of Predicate-Based Filtering (PBF) using Groute, shown in Figures 5 and 6. PBF is a kernel in many applications such as database management and image processing. The input to PBF is a large one-dimensional array, and the output is an array containing all elements of the input array that satisfy a given predicate. In the Groute program, the host divides the input array into segments and sends them to free GPUs on demand to promote load-balancing. Processed segments are transferred by the GPUs back to the host, where they are assembled to produce the output. Figure 5 depicts the resulting dataflow graph.

In Figure 6, the code sets up the dataflow graph for PBF in lines 5–19. The physical devices present in the system are determined by accessing a structure of type `Context` (line 5). The PBF program creates a router named `h2gpus` for scattering segments of the input array from the host to the GPUs, as well as a router named `gpus2h` to gather segments from the GPUs to create the output array (lines 9–10).

The code in lines 12–13 specifies the links between these routers and the host, where the link between the host and the `h2gpus` router is created without double buffering. The code in lines 15–19 creates a worker-thread for each GPU using double-buffered links, and input segments are scattered to the devices (line 21). Upon distributing all input segments, the distributor notifies that it will send no further information by sending a shutdown signal (line 22). The result is obtained at the host (lines 24–30) and the program stops once all GPUs send shutdown signals to `gpus2h`, notifying that no additional data will be received (line 26).

The routing policy for both routers is straightforward, selecting the first available device out of all possible router destinations (line 34). On the GPU end, each device asynchronously handles incoming messages (line 43). Once a `PendingSegment` is assigned to the device, it is synchronized with the active GPU stream (line 45) and processing is performed (line 46). Line 47 queues a command to the stream, which releases the incoming buffer for future use upon processing completion. The results are then transmitted back to the host using `out` (line 48). When a shutdown signal is received, the worker thread is terminated (line 44) and a shutdown signal is sent (line 50) to `gpus2h`.

3.1 Distributed Memory Model and Semantics

Memory consistency and ownership are maintained by the programmer in Groute. The link/router model does not define a global address space or remote memory access operations, but functions

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

Fig. 6. Predicate-Based Filtering (PBF) pseudocode.

as a message-passing distributed memory environment. In the article, algorithms and high-level asynchronous objects implemented over the model (such as Distributed Worklists) define ownership policies, whereas the low-level constructs provide efficient communication and message routing.

As a message-passing model, Groute translates directly to multi-node clusters. The same concepts of communication and routing can be implemented using message-passing interfaces (such as MPI) or via sockets and packets. As dataflow routers generalize commonly used collective operations, optimizing the same high-level constructs for latency/bandwidth on distributed systems must take network topology into account, which is beyond the scope of this work. For

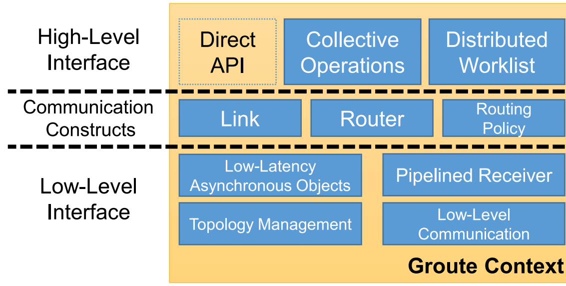


Fig. 7. The Groute library.

many algorithms, however, a hierarchical decomposition approach is beneficial, using Groute to efficiently utilize a single multi-GPU node, combined with MPI for cluster-level communication.

4 IMPLEMENTATION DETAILS

We realize the Groute programming model by implementing a thin runtime environment [2] over standard C++ and CUDA to enable asynchronous multi-GPU programming. The environment consists of three layers, illustrated in Figure 7. The bottom layer contains low-level management of the node topology and inter-GPU communication, the middle layer implements the Groute communication constructs (using the topology to optimize memory transfer paths), and the top layer implements high-level operations that are commonly used in asynchronous regular and irregular applications. For direct control over the system, each of the layers can be manually accessed by the programmer.

The runtime environment is managed by a `Context` object, which receives a list of devices to manage. Upon this context, it is possible to construct the dataflow entities (i.e., Endpoints and Routers) as well as the higher-level algorithmic data structures. The context manages its own memory, which can be configured by the user upon construction. This facilitates interoperability with other libraries as well as with multiple instances of Groute.

In the rest of this section, we elaborate on the implementation of each layer in Groute.

4.1 Low-level Interface

The low-level layer builds upon insights from Section 2 to provide programmer-accessible interfaces for efficient peer-to-peer transfers. Specifically, the layer provides *Low-level Communication* APIs for latency reduction; *Pipelined Receivers* to increase computation-communication overlap; *Topology Management* for node interconnect hierarchy introspection; and *Low-Latency Asynchronous Objects* to mitigate system overhead.

The low-level communication interface provides host- and device-initiated memory copy functionality, abstracting packetization and conditional transfer. In Groute, send and receive operations are segmented into packets to increase the overall responsiveness of the node and enable overlapping communication between multiple devices. Without packetization, occasional small transmissions (e.g., GPUs sending counters to the CPU) may suffer from head-of-line blocking behind regular large transfers (e.g., GPU–GPU transfers).

On top of the low-level interface, asynchronous communication is abstracted using a *pipelined receiver* object, which efficiently utilizes the two memory copy engines and the compute engine (Figure 4) by using double- and triple-buffering [45]. The implementation of multiple-buffering allocates read buffers, queuing virtual "future read operations." When data are sent to a pipelined receiver, a read operation is removed from the queue and the corresponding buffer is assigned to

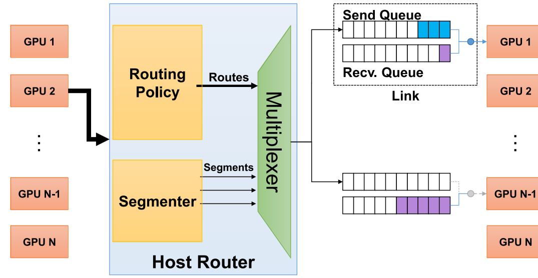


Fig. 8. Host-controlled router diagram.

the sender. Simultaneously, the reader is notified of the incoming pending operation, which can either be waited for or acted upon asynchronously using the receive stream.

Asynchronous programs often rely on a multitude of fine-grained (non-bulk) synchronization points and impromptu memory allocation to operate correctly. To minimize the incurred driver overhead and involuntary system-wide synchronizations, Groute provides several *low-latency asynchronous objects*, among which are *Event Pools* and *Event-Futures*. Event pools facilitate the creation of many short-lived events by way of pre-allocation; whereas event-futures are waitable objects implementing the *future/promise* pattern [25] to maintain two-layered synchronization between the CPUs and GPUs. In particular, event-futures handle situations where GPU events are known to be recorded in the future, but not yet created. These objects are used as the primary building block for queuing various actions, such as future receive operations for devices (CPU and GPU), and can either be synchronized with a CPU thread or a GPU stream.

4.2 Communication and Scheduling

A link can only connect one pair of source and destination endpoints. In Groute, there are two methods to create links: directly or using an existing router. Specifically, each link specifies its maximal receivable packet size and the number of possible pipelined receive operations. The latter is optional and determined automatically if not given. The `Send` and `Receive` methods initiate memory transfer and return an event-future. In particular, a receive operation returns a future to a `PendingSegment`, which contains an event and a segment that may not yet be ready for processing. Links also provide a socket-like `Shutdown` function, signaling that no further information will be sent.

The internal structure and workflow of a router is depicted in Figure 8. The figure shows that the router contains three main components. The *Segmenter* component controls breaking down messages into segments according to the destination device capabilities; the *Routing Policy* controls the message destination(s); and the *Multiplexer* is responsible for message assignment to available GPUs.

Given a message to send, a routing policy will determine its one or several destinations using the programmer callback. The router then controls send operation scheduling, assigning destinations based on availability. Note that routing performance depends on the underlying topology. For example, in some nodes it is efficient to reduce with all-to-one operations, while in others it is better to use a hierarchical tree for concurrent reductions.

Upon receiving a segmented message and its possible destinations, scheduling is managed by the multiplexer. As shown in the top-right portion of Figure 8, the implementation involves queuing send operations to each of the target devices' send queues. Since links use pipelined receivers, devices also maintain receive queues. If there is a match between a send operation and a receive

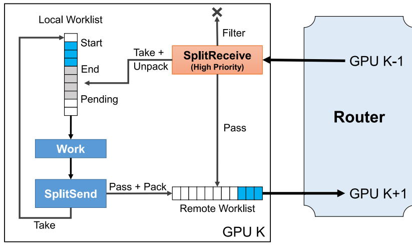


Fig. 9. Distributed worklist implementation.

Function	Description
`RemoteWork Pack(LocalWork item)`	Pack `item` to send to another device.
`LocalWork Unpack(RemoteWork item)`	Unpack received work-item.
`Flags OnSend(LocalWork item)`	Determine outgoing `item` destination.
`Flags OnReceive(RemoteWork item)`	Determine incoming `item` destination.
`Priority GetPrio(RemoteWork item)`	Obtain priority of incoming `item`.

Fig. 10. Distributed worklist programmer callbacks.

operation, transfer assignment is performed, dequeuing one item from both the send and receive queues of the link. Additionally, the redundant send operations queued to other devices are marked as stale and removed by each device upon inspection. This ensures that routers do not require a centralized locking mechanism, and per-device queues are the only constructs that should implement thread-safety.

4.3 Distributed Worklists

The high-level interface provided by Groute implements reusable operations that can often be found in multi-GPU applications, such as broadcast and all-reduction. This section details the implementation of distributed worklists, frequently used in irregular algorithms and graph analytics.

Distributed worklists maintain a global list of computations (work-items) that should be processed. Each such computation, in turn, may create new computations that are queued to the same list. For example, breadth-first search traverses a node's neighbors, propagating through the graph by creating new work-items for each neighbor.

Implementing efficient distributed multi-GPU worklists is a challenging task. As each device may contain a different portion of the input data, only certain devices are able to process specific work-items. Thus, distributed worklists require all-to-all communication. Using routers and bus topology, Groute implements distributed worklist management.

In the implementation, global coordination and work counting is centralized and managed by the host. During runtime, devices periodically report produced and consumed work-items for tracking purposes. Once the number of total work-items becomes zero, processing stops and a shutdown signal is sent to all participating devices via router links.

Figure 9 illustrates the implementation of a distributed worklist in Groute from the perspective of a single device. As seen in the figure, the worklist is implemented over a single, system-wide router. To support efficient all-to-all communication, the ring topology is used for the routing policy by default (based on our experiments in Section 2). Apart from the router, each device contains a locally managed worklist, which comprises one or more multiple-producer-single-consumer queues for local tasks. The implementation consists of two threads per device: worker thread and receiver thread, which controls inter-GPU communication and work-item circulation.

Over the ring topology, the workflow presented in Figure 9 is implemented as follows: Each device receives information from the previous device, according to the ring. The received data then undergo filtering and separation (`SplitReceive`), which passes irrelevant information to the next device. Items that are relevant to the current device are unpacked and “pushed” onto its local worklist, signaling the worker thread that new work is arriving. At the same time, the worker thread processes existing work-items, separating the resulting items to local and remote work (`SplitSend`) and packing outgoing information as necessary. Note that the `SplitReceive`

kernel is queued on a separate, high-priority stream. This causes the kernel to be scheduled during existing work processing, increasing the performance and responsiveness of the system.

To implement an algorithm over a distributed worklist, five functions must be given by the programmer, listed in Figure 10: `Pack`, `Unpack`, `OnSend`, `OnReceive`, and `GetPrio`. These functions usually consist of a single line of code but may adversely change work-item propagation. In the `OnSend` and the `OnReceive` functions, the `Flags` return value is a bit-map that controls the work-item destination(s), e.g., pass to the next device, keep, duplicate, or completely remove. The priority of a work-item, obtained using `GetPrio`, is then used for scheduling higher-priority work before low-priority items, as detailed below.

To implement local worklist queues, Groute uses GPU-based lock-free circular buffers. Such buffers are beneficial for asynchronous applications, as they eliminate the need for dynamic allocation of buffers during runtime. If a circular buffer overflows beyond its allocation, checks are triggered in host and device code that will abort program execution.

As shown in Figure 9, each worklist queue consists of multiple producers and a single consumer. Our implementation contains a memory buffer and three fields: `start`, `end`, and `pending`. Work consumption is performed by atomically increasing the `start` field. To avoid consuming items that are not ready, production is controlled by atomically increasing the `pending` field, reserving space in the buffer. After a producer has finished appending its work, `end` is increased by a single writer thread, synchronizing with `pending`.

Additional optimizations to the circular buffers are performed in Groute. If the consuming GPU stream also produces work (as in `SplitSend`), work is pushed to the queue by way of prepending information (i.e., decreasing `start`), which avoids producer conflicts. It is also worth noting that circular buffers use warp-aggregated atomics [4], which increase the efficiency of appending work by limiting the number of atomic operations to one-per-warp.

4.4 Soft-priority Scheduling

A pitfall that should be considered in asynchronous worklist algorithms is excess work resulting from intermediate value propagation. In contrast to bulk-synchronous parallelism, where all devices agree upon a global state in the algorithm, asynchronous concurrency may propagate stale information ("useless work") as a result of lagging devices. Such work-items, in turn, generate additional intermediate work that may increase the overall workload exponentially with the number of devices.

For example, in bulk-synchronous Breadth-First Search (BFS), the current traversed level is a global algorithm parameter. If there are two paths to a given node, where one is longer than the other, only the path with the least number of edges from the source will be registered, writing final values to the nodes. In asynchronous BFS, however, if the path with the least number of edges is located on a lagging device, the "incorrect" path (intermediate value) would be written first. This will, in turn, traverse the rest of the graph using the intermediate value as input. After the device with the short path completes its processing, it will overwrite the node values, essentially recomputing all traversed values.

One way to mitigate this issue is to assign *soft priorities* to each work-item, deferring items that are suspected as generators of "useless work" to a later stage, in which they will likely be filtered out [24].

In Groute, soft-priority scheduling is implemented using the programmer-provided `GetPrio` callback. During the runtime of the application, only high-priority work-items are processed, where the priority threshold is decided by a system-wide consensus. This consensus threshold reflects a priority value that is close to what all the systems are currently processing. This prevents a GPU from running too far ahead of the others and producing results that may eventually

turn out to be stale. Once items under the threshold are completed, the system increases the threshold and the distributed worklist will process the deferred items on each device. As we shall show in Section 5, using soft-priority scheduling decreases the amount of intermediate work, increasing overall performance.

4.5 Worklist-based Algorithms

Using asynchronous Breadth-First Search (BFS), we illustrate how graph traversal algorithms such as Single-Source Shortest Path (SSSP) and PageRank (PR) are implemented using distributed worklists. The corresponding code for asynchronous BFS is listed in Figure 11.

As described above, the code is composed of four parts: The GPU work kernel (lines 2–20), callbacks upon communication with the Distributed Worklist (lines 24–40), the per-GPU management procedure on the host (lines 44–51), and the host management and initialization (lines 55–66).

In BFS, the input graph is given in the Compressed Sparse Row (CSR) matrix format (`CSRGraph` in the code). The graph is first partitioned among the available devices, where each device statically maintains ownership of a contiguous memory segment, corresponding to a subset of the vertices. In the example, we assume that the partitioned graph was already copied to the GPUs.

When BFS processing starts, the host enqueues a single work-item to the worklist—the source vertex (line 65). Groute, using the partitioned graph, ensures that the initial work-item is sent to its owner device, where it is processed by setting the vertex value (i.e., *level*) to zero and creating work-items with *level*=1 for each neighboring vertex (lines 10–11). If a neighboring vertex is owned by the processing device, it is queued to the device-local worklist (top-left portion of Figure 9). Otherwise, it is asynchronously propagated through the distributed worklist until another device claims ownership on the work-item, a behavior that is specified in the `OnReceive` callback (lines 26–32). The value of *level* is propagated as well. Atomic operations are used to check if the received value is lower, updating the vertex's value and propagating *level*+1 to the neighboring vertices through subsequent work-item processing (lines 16–17). After all relevant edges have been traversed, the worklist becomes empty and the algorithm ends.

The host management (`Work` function) is invoked by Groute as long as there is a work segment to process. A library method (line 46), called `KernelSizing`, assists the programmer with choosing thread-block and grid dimensions, such that the device-level worklist will be able to operate concurrently with the work. Upon communication, the device-level worklist invokes the right callbacks in `SplitOps` (lines 26–39).

In addition to owned vertices, each device also stores a local copy of its "halo" vertices, namely, neighboring vertices owned by other devices. The level of a halo vertex can only be updated by the local device and is used to skip sending irrelevant updates, thus reducing inter-device communication. Skipping such updates does not change the algorithm's behavior, since shared value updates are well-defined in the original algorithm. In our BFS example, shared updates resolve to the lowest vertex value.

4.6 Traversal Graph Algorithms (TGA) in Groute

We implement graph algorithms in Groute to demonstrate its ability to handle fine-grained asynchronous communication and scheduling. A multi-GPU node is unlike a cluster in that it possesses both shared memory (among the CPUs) and distributed memory (across the GPUs or across the GPUs and CPU). This allows us to take an unconventional approach in structuring our implementations of graph algorithms.

Any graph algorithm can in principle use the Groute programming model, provided that it can be converted to a push/pull scheme [8]. For the algorithms to converge in an asynchronous, distributed memory setting, programmers must define conflict-resolution operations on the updates

```
1  // GPU kernel
2  __global__ void BFSKernel(CSRGraph graph, LevelsDatum levels_datum,
3                            WorkSource work_source,
4                            WorkTarget work_target) {
5      int tid = TID_1D;
6      unsigned nthreads = TOTAL_THREADS_1D;
7      uint32_t work_size = work_source.get_size();
8  
9      for (uint32_t i = tid; i < work_size; i += nthreads) {
10         index_t node = work_source.get_work(i);
11         level_t next_level = levels_datum.get_item(node) + 1;
12 
13         for (index_t edge = graph.begin_edge(node), end_edge = graph.end_edge(node);
14              edge < end_edge; ++edge) {
15             index_t dest = graph.edge_dest(edge);
16             if (next_level < atomicMin(levels_datum.get_item_ptr(dest), next_level))
17                 work_target.append_work(graph, dest);
18         }
19     }
20 }
21 
22 //--------------------------------------------------------
23 // Worklist callbacks
24 struct SplitOps
25 {
26     __device__ SplitFlags OnReceive(const remote_work_t& work) {
27         if (m_graph_seg.owns(work.node))
28             return (work.level < atomicMin(get_item_ptr(work.node), work.level))
29                 ? groute::SF_Take
30                 : groute::SF_None; // filter out
31         return groute::SF_Pass;
32     }
33     __device__ SplitFlags OnSend(local_work_t work) {
34         return (m_graph_seg.owns(work)) ? groute::SF_Take : groute::SF_Pass;
35     }
36     __device__ remote_work_t Pack(local_work_t work) {
37         return remote_work_t(work, get_item(work));
38     }
39     __device__ local_work_t Unpack(const remote_work_t& work) { return work.node; }
40 };
41 
42 //--------------------------------------------------------
43 // GPU worker
44 void Work(const Segment<local_work_t>& work, Worklist& output_worklist, Stream& stream) {
45     dim3 grid_dims, block_dims;
46     groute::KernelSizing(grid_dims, block_dims, work.GetSegmentSize());
47     BFSKernel<<<grid_dims, block_dims, 0, stream.cuda_stream>>>(
48         graph, levels,
49         WorkSourceArray<local_work_t>(work.GetSegmentPtr(), work.GetSegmentSize()),
50         WorkTargetWorklist(output_worklist));
51 }
52 
53 //--------------------------------------------------------
54 // Host side
55 Context ctx;
56 size_t num_gpus = ctx.devices().size();
57 Router<remote_work_t> router(ctx, Policy::CreateRingPolicy(num_gpus));
58 DistributedWorklist<local_work_t, remote_work_t> dwl (ctx, router, num_gpus);
59 for (size_t i = 0; i < num_gpus; ++i)
60     distributed_worklist.CreatePeer(i, SplitOps(), Work, flags...);
61 ctx.SyncAllDevices();
62 
63 remote_work_t initial_work (source_node, 0);
64 Link work_sender (HOST, router);
65 work_sender.Send(Segment(&initial_work, 1), Event());
66 work_sender.Shutdown();
```

Fig. 11. BFS distributed worklist source code.

as they are considered by the recipient worker (e.g., `minimum` for SSSP). This ensures that even if the operations are reordered within a bulk-synchronous step, or between steps in the asynchronous case, the updated values will propagate correctly. To use soft-priority scheduling, updates to algorithm state must have a notion of priority and be deferrable. Not all algorithms support this, but as we show below, certain TGAs define this priority naturally.

In essence, we structure our implementations of so-called traversal algorithms (BFS, SSSP, and PR) into three parts. The first, which we label the *problem*, is the per-GPU distributed part of the computation. Each problem instance is handed a partition of the problem that it solves independently on one GPU, independently of the other problem instances.

These per-GPU problem instances are mirrored by their CPU-side *solver* instances. Each solver is responsible for managing the CPU-side affairs of a problem instance.

Finally, the last part consists of one *algorithm* instance that maintains a global view of the computation to take full advantage of the shared memory available to the host CPUs.

In the subsections below, we explore the full responsibilities for each of these parts, in the implementation of traversal graph algorithms, starting from the algorithm.

4.6.1 Algorithms for TGA. An implementation of the algorithm for a traversal graph begins by taking the input graph in compressed sparse row (CSR) format. It then runs the partitioner (in our case, the METIS algorithm [22]) to construct partitions of the graph by assigning an equal number of vertices to each GPU device. All of a vertex's outgoing edges are retained on the device that owns it—the edge list of a vertex is not split.

Data maintained by the graph really consist of three kinds, each of which needs a different partitioning strategy. First, there is read-only data, such as the graph topology. Such data are easy to partition and distribute, since they do not need to be kept coherent.

Data that must be written can be further classified as *local* data, which belong to the vertices assigned to the GPU, and *remote* data for vertices that are resident on another GPU. We begin by first partitioning vertex data among all the GPUs, handing each GPU the partition for the data for the vertices it owns. The usual strategy for remote data allocates space only for the remote data directly reachable from this partition. The resulting sparse data structure then needs an index to lookup.

In our implementations of TGAs in Groute, we choose a simpler but faster strategy for handling remote data based on two observations. First, nearly all of the algorithms we implement mutate only vertex data, whereas edge data are usually read-only. Second, for all the graphs we are interested in, the number of vertices is relatively small compared to the number of edges. Indeed, most GPUs can store data for a few billion vertices without running out of memory. This allows us to reduce the latency of message handling for such graphs by storing vertex data for *all* nodes on a single GPU in a straightforward manner—in a directly indexed array. Reads/writes as well as atomic read/modify/writes in graph algorithms can then proceed locally on the GPU without needing to perform remote transfers. For larger graphs where this is not possible, Groute can work with other vertex partitioning methods. The translation of the vertex indices should then be provided by the `Pack` and `Unpack` callbacks (Figure 10).

While we store data for all the nodes, we only transmit remote data. As work is placed on a outgoing work list, it is separated into local work and remote work. The latter is then packaged into messages that are then sent over to the other GPUs. On the receiving end, the GPU unpacks the work (usually, the vertex and the new data value) and applies it to the graph creating work on the local worklist if needed.

4.6.2 Solvers for TGA. A solver is the CPU counterpart of the asynchronous GPU kernels. It is responsible for starting GPU kernels and keeping them running until distributed termination is detected.

The structure of a solver is general enough that we can utilize the same solver (suitably parametrized) for BFS and SSSP. This generic multi-GPU solver begins by setting up a termination detection loop over the distributed worklist. Then, as long as the distributed worklist has not indicated termination, it performs the following steps:

(1) *Obtain local work*: The solver queries the worklist to obtain the work assigned to this GPU.
(2) *Perform local work*: The work assignment arrives as a series of lists, which is then handed off to the corresponding problem kernel, which will be described in the next section. Each problem kernel operates on the local work and produces additional work that consists of both local and remote work.
(3) *Forward remote work*: The solver then splits remote work from local work, forwarding remote work to remote GPUs.
(4) *Append local work*: As part of the split procedure, local work is placed on the local worklists to be handled in the next iteration of the loop.
(5) *Participate in termination detection*: Finally, before commencing the next iteration of the loop, the solver updates its work status so that asynchronous termination detection can be used to determine if the algorithm should continue.

Note that if kernel fusion (device-based routers) is enabled, steps (2)–(4) are handled within a persistent GPU kernel.

Unlike BFS and SSSP, which start with a single node, PageRank (PR) begins by operating on all vertices. To avoid creating explicit worklists containing all the nodes, our PR problem uses a slightly different GPU kernel for the first iteration (these kernels use vertex ranges instead of worklists). Therefore, we create a custom solver for PR. However, its overall structure remains the same as that for BFS and SSSP.

4.6.3 Problems for TGA. Given our setup so far, the set of actual GPU kernels (i.e., the *problem*) for each algorithm requires very little modification from their single-GPU equivalents to operate on multiple GPUs.

Breadth-First Search. The BFS kernel uses `atomicMin` to update values in memory. This is really an artifact of running asynchronously, rather than of running on multiple GPUs.

Single-Source Shortest Path. The primary SSSP kernel, remarkably, is unchanged from its single GPU version. We also considered an implementation of the Near–Far variant of SSSP [12]. This variant examines each updated node to determine if it is local or remote before appending it to the local near/far lists or to the remote GPUs. However, ordinary SSSP coupled with soft-priority essentially behaves like SSSP Near–Far, so we do not use the explicit Near–Far version in our evaluation.

PageRank. Our PageRank kernel implementation uses a residuals-based implementation [44], and this necessitates differentiation between the single-GPU and multi-GPU kernels. For the vertices it owns, a GPU can wait until a local vertex's residual is greater than a pre-determined PageRank-specific ϵ value before re-adding the vertex to the worklist. It cannot do this for remote vertices—all GPUs may have a residual update smaller than ϵ for a particular remote vertex, but their sum can exceed ϵ. Thus, the PageRank kernel explicitly checks if a residual update is for a remote node and forwards the update regardless of the magnitude of the change. The receiving

GPU, of course, accumulates all the incoming residual updates and only schedules a vertex onto the worklist once the accumulated residual is greater than ϵ.

4.6.4 Putting Algorithms, Solvers, and Problems Together. Ultimately, once the user has provided the algorithm, solver, and problem corresponding to a graph algorithm, execution on multiple GPUs can begin. After the graph is loaded and partitioned, algorithm initialization routines are called. Solver and Problem instances are then created for each GPU. For asynchronous, parallel execution, a CPU thread is created for each GPU. Each CPU thread runs one solver instance. Once the solver instances have terminated, the GPU computation is complete. The algorithm then gathers data from across the GPUs into host memory. Only data of vertices belonging to each partition are read back into the host. Any algorithm-defined output routines are then run on the host data.

5 PERFORMANCE EVALUATION

As a multi-level infrastructure, the Groute programming environment can represent a versatile set of programs efficiently. In this section, we showcase irregular algorithms from different classes and analyze their performance in detail. The algorithms include three Traversal Graph Algorithms (TGA) that can be implemented over a worklist, a graph algorithm that is not based on worklists, and an irregular query over dense data (e.g., on a column-store database). In particular, the following five algorithms are evaluated:

- **Breadth-First Search (BFS):** Traverses a graph from a given source node, outputting the number of edges traversed from the source node to each destination node. Implementation is push-based and data-driven, i.e., using distributed worklists.
- **Single-Source Shortest Path (SSSP):** Finds the shortest path (using edge weights) from a source node to all other nodes. Implementation is push-based and data-driven.
- **PageRank (PR):** Computes the PageRank measure for all nodes of a given graph using a worklist-based algorithm [44]. Implementation is the push-based variant proposed in the article.
- **Connected Components (CC):** Computes the number of connected components in a given graph. Implementation is topology-based, using two routers as explained below.
- **Predicate-Based Filtering (PBF):** Filters an array of elements according to a given condition. GPU kernel implementation is based on warp-aggregated atomics [4].

Groute is compared with two benchmark implementations of multi-GPU parallel graph algorithms: Gunrock (version 0.4) and Back40Computing (B40C). Gunrock [33] is a graph analytics library containing highly optimized implementations of various graph algorithms. Gunrock uses bulk-synchronous parallelism for its multi-GPU implementations of these algorithms. B40C, by Merrill et al. [27], contains state-of-the-art hardcoded BFS implementations, enabling multi-GPU processing by direct memory accesses between peer GPUs. We measure each application five times, reporting the median performance and the 25th and 75th percentiles of all runs, represented by shaded error regions.

All implementations, including Groute, contain the following kernel-level optimizations: warp-aggregated atomic operations, warp-based collectives for inter-thread communication, and intra-GPU load balancing on the warp and thread-block level to exploit Nested Parallelism [32]. Additionally, Groute's asynchronous model allows us to perform kernel fusion (Section 5.2).

Table 2 summarizes the best running times for BFS, SSSP, PR, and CC, with the number of GPUs used to achieve the highest performance in parentheses. Asynchronous implementations using Groute clearly dominate over bulk-synchronous implementations and sometimes even outperform hardcoded versions.

Table 2. Best Performance Comparison [ms]

	Algorithm	Graph				
		USA	OSM-eur-k	soc-LiveJournal1	twitter	kron21.sym
BFS	Gunrock	741.50 (1)	3,170.72 (1)	40.41 (6)	–	66.41 (6)
	B40C	**56.66** (1)	2,250.40 (2)	**12.98** (4)	–	–
	Groute	129.00 (2)	**622.84** (5)	25.87 (4)	**719.32** (8)	**42.84** (7)
SSSP	Gunrock	48,417.77 (8)	526,028.06 (8)	61.44 (6)	1,446.44 (7)	**188.14** (3)
	Groute	**722.39** (3)	**3,531.47** (8)	**31.89** (6)	**645.32** (8)	245.30 (8)
PR	Gunrock	820.64 (5)	13,778.61 (1)	442.63 (8)	**14,989.14** (8)	**394.26** (8)
	Groute	**166.45** (8)	**1,034.53** (8)	**367.22** (4)	26,176.12 (4)	1,991.36 (8)
CC	Gunrock	171.46 (1)	1,510.78 (1)	67.02 (1)	–	72.58 (6)
	Groute	**14.27** (5)	**160.74** (4)	**14.77** (6)	**404.12** (8)	**13.73** (8)

Table 3. Graph Properties

Name	Nodes	Edges	Avg. Degree	Max Degree	Size (GB)
Road Maps					
USA [1]	24M	58M	2.41	9	0.62
OSM-eur-k [3]	174M	348M	2.00	15	3.90
Social Networks					
soc-LiveJournal1 [13]	5M	69M	14.23	20,293	0.56
twitter [10]	51M	1,963M	38.37	779,958	16.00
Synthetic Graphs					
kron21.sym [5]	2M	182M	86.82	213,904	1.40

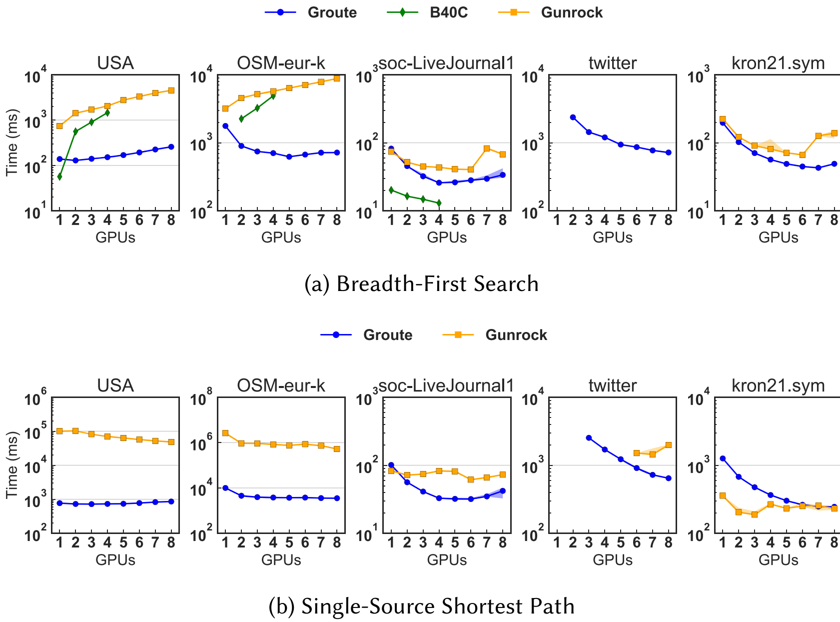
Our experimental setup consists of two node types. The first is an eight-GPU server of four dual-board NVIDIA Tesla M60 (Maxwell architecture) cards, each containing 16 MPs with 128 cores; and two eight-core Intel Xeon E5-2630 v3 CPUs. Bus topology is depicted in Figure 1(a), with two QPI links per CPU at 8 GT/s per link for the PCI-Express switch. The second node type is a two-GPU heterogeneous server that contains one Quadro M4000 GPU (Maxwell, 13 MPs with 128 cores); one Tesla K40c GPU (Kepler architecture, 15 MPs with 192 cores); and one six-core Intel Xeon E5-2630 CPU with two total QPI links at 7.2 GT/s per link.

Table 3 lists dataset information and statistics for input graphs used in the evaluation. All graphs are partitioned between GPUs using an edge-cut obtained from METIS [22] except for *kron21.sym* and *twitter*. METIS fails to partition these two graphs, so we simply partition the node array equally among the GPUs.

5.1 Strong Scaling

Figure 12 presents the absolute runtime of the two graph traversal algorithms (BFS and SSSP) running on one to eight GPUs and comparing with the above frameworks. Missing data points indicate runs that failed due to crashes, out-of-memory failures, or incorrect outputs (compared to externally generated results).

Overall, observe that in BFS and SSSP, which are communication-intensive, the topology of the bus starts affecting the performance of applications when transfers are performed beyond the single four-GPU quadruplet. While Groute mitigates these issues by optimizing communication



(a) Breadth-First Search

(b) Single-Source Shortest Path

Fig. 12. Traversal algorithm timing (lower is better).

paths (Section 2), the phenomenon can still be seen in high-degree graphs such as *soc-LiveJournal1* in BFS, SSSP, and PR.

5.1.1 Breadth-First Search. Figure 12(a) compares Groute with Gunrock and B40C. B40C's multi-GPU implementation requires direct memory access between all devices, and thus only runs up to four GPUs. Additionally, B40C does not use METIS partitioning, failed on *twitter* and *kron21.sym*, and ran out of memory on the single-GPU version of *OSM-eur-k*. The Gunrock implementation of BFS ran out of memory on all *twitter* and *OSM-eur-k* GPU configurations. Since the out-of-memory issue for *OSM-eur-k* only appears in Gunrock v0.4, multi-GPU results for this input were obtained from v0.3.1.

The figure shows that Groute outperforms Gunrock in all cases, with significant improvements in road networks (*USA*, *OSM-eur-k*). This is due to the kernel fusion optimization enabled by asynchronous processing, which dramatically decreases kernel launch overhead in high-diameter graphs (Section 5.2).

Groute also outperforms B40C on road networks on multiple GPUs. However, B40C is faster on one GPU on the *USA* input and always outperforms Groute and Gunrock on the *soc-LiveJournal1* input. B40C's BFS implementation is highly optimized, containing a hybrid implementation that switches between different kernels as described in their paper [27], an optimization we did not implement due to its highly specialized nature.

5.1.2 Single-Source Shortest Path. Figure 12(b) presents the strong scaling of SSSP in Groute. In the figure, we see that the multi-GPU scaling patterns are similar to BFS. The multi-GPU scalability of Groute is especially apparent in large graphs, such as *twitter*, in which performance increases even when using more than four GPUs. Missing *twitter* results are caused by insufficient memory. Due to the same memory issue as BFS, results for *OSM-eur-k* were obtained from Gunrock v0.3.1.

Groute outperforms Gunrock (or matches it on *soc-LiveJournal1*, single GPU) with the sole exception of the *kron21.sym* input. Upon in-depth inspection, it was found that the asynchronous

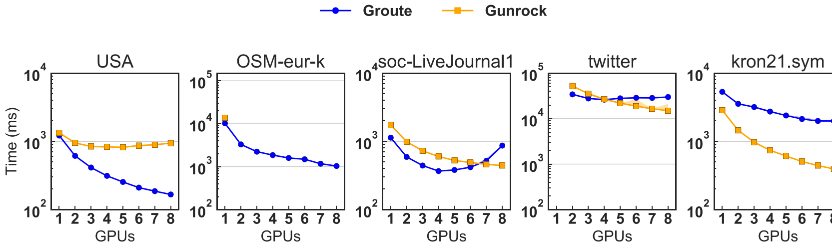


Fig. 13. PageRank Execution Time.

implementation in Groute causes an inflation in the number of performed atomic operations, which increases memory contention and iteration time. This inflation is a direct result of the cut size of the partitioned graph, which affects the number of incoming messages from different GPUs, increasing the probability of "useless" work.

5.1.3 PageRank. The performance of PageRank (PR) is shown in Figure 13. To produce consistent results on one to eight GPUs, Gunrock's multi-GPU PageRank uses a modified algorithm that normalizes the sum of ranks to 1 and compensates for 0 out-degree vertices. It also eliminates self-loops in the graph. Thus, its results are not directly comparable to Groute. Without these modifications suggested by the Gunrock authors,[1] Gunrock usually fails self-validation on multiple GPUs. Gunrock also runs out of memory on *OSM-eur-k* for more than one GPU, and the *twitter* graph does not fit in the memory of one GPU when using both Groute and Gunrock.

As opposed to BFS and SSSP, PR is a computationally intensive problem. In addition, PR starts by processing all the nodes simultaneously, so each GPU can be fully utilized. Observe that in the figure, Groute outperforms Gunrock on all inputs up to four GPUs except *kron21.sym*, again due to the abundance of atomic operations in Groute. On *twitter* and *soc-LiveJournal1*, we see that Gunrock does not stop scaling as opposed to Groute. This is a result of Groute's use of the distributed work-queue, which we observed to be nearly full in these cases. A possible solution to this is to merge multiple updates prior to sending them, as a bridge between asynchronous and bulk-synchronous parallelism.

Both road networks generate multi-GPU scaling over the single-GPU version, owing to the amount of independent work performed on each device, as well as the communication latency that is hidden by Groute. The same effect is observed in *soc-LiveJournal1* up to a single quadruplet of GPUs. In particular, the best scaling results are obtained when the ratio of computations to communications is high (i.e., less communications per computation). For example, running PageRank with Groute on *USA* yields a 7.32× speedup on eight GPUs over one, exhibiting near-linear scaling on all multi-GPU configurations.

As the ratio of computation to communication decreases, scaling becomes less substantial, as in the case of *soc-LiveJournal1*, which exhibits a 3.09× speedup on four GPUs over one. Additionally, nearly no speedup is observed in *twitter*, which is partitioned randomly (i.e., without METIS) and heavily interconnected.

5.1.4 Connected Components. We implement a topology-driven [36] variant of multi-GPU Connected Components (CC), which does not use a worklist, to demonstrate the expressiveness of Groute's asynchronous communication constructs. Its performance is shown in Figure 14.

[1] https://github.com/gunrock/gunrock/issues/191#issuecomment-251257782.

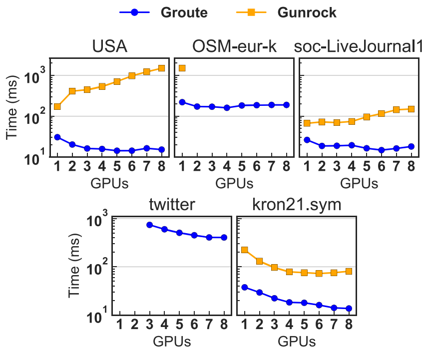


Fig. 14. Connected components performance.

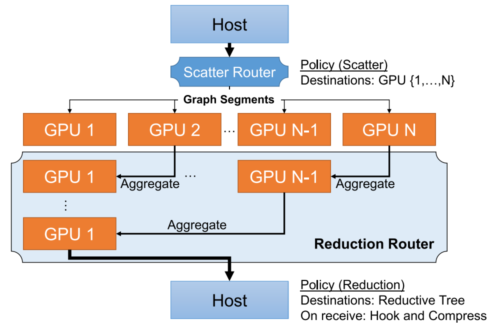


Fig. 15. Connected components router structure.

Figure 15 illustrates the dataflow graph of a pointer-jumping topology-driven CC over Groute. In this version, the input graph representation is an edge-list. The edges are distributed to the GPUs, and each GPU keeps note of the parent component of each vertex in its given graph subset using operations called Hook and Compress [40]. Once a GPU converges locally, its results are merged with the results of other GPUs to converge to the global component list. Since the pointer-jumping algorithm essentially yields parent-pointing trees with each step, we can merge partial results of CC to a global result by reusing the same Hook and Compress operations on those trees as if they were part of a standard input graph.

With Groute, we create one router to dynamically scatter edges to the GPUs in multiple segments, computing and aggregating local CC results for each segment. Upon completion, each GPU merges its results with the others using an additional reduction router. According to the topology of the measured eight-GPU node, the reduction is implemented as concurrent hierarchical operations. In particular, each GPU merges its results with a designated “sibling” GPU until reaching the first GPU, which sends the information back to the host. The single GPU kernels used in this implementation are based on a state-of-the-art adaptive variant [40, 41] of the pointer-jumping method described by Soman et al. [38].

The results in Figure 14 show that using an asynchronous topology-driven variant is highly beneficial, both in terms of raw performance and scalability, over the implementation found in Gunrock. Specifically, Groute yields 5.61× and 98.23× speedups over Gunrock in *USA* on one and eight GPUs, respectively. Furthermore, Groute achieves a strong scaling of up to 2.74× (eight GPUs over one) in the *kron21.sym* input.

Another advantage apparent in the figure is memory consumption. Since the Groute implementation does not use distributed worklists, the tested multi-GPU system is able to compute CC on large-scale graphs such as *OSM-eur-k* and *twitter*. Gunrock, however, runs out of memory for all multi-GPU configurations on these two inputs.

5.2 Distributed Worklist Performance

Our distributed worklist uses two optimizations—soft-priority scheduling and worker kernel fusion—to significantly improve the performance of asynchronous algorithms.

As explained in Section 4.4, a naive asynchronous irregular application propagates intermediate values from lagging devices, which leads to an increase in work due to redundant computations. Using METIS mitigates this effect somewhat, since it reduces the number of paths between the partitions. However, our experiments show that deferring possibly redundant work by using a soft-priority scheduler can achieve significantly better performance *even* with good partitions.

Table 4. Distributed Worklist SSSP Performance

Graph	GPUs	Unoptimized Time [ms]	Improvement	
			Soft-priority Scheduler	Fused Worker
soc-LiveJournal1	1	182.32	1.03×	1.03×
	2	135.71	0.95×	0.95×
	4	89.40	1.29×	1.31×
	8	79.20	1.29×	1.09×
kron21.sym	1	2,939.54	1.02×	1.01×
	2	1,789.01	1.10×	1.09×
	4	1,607.42	1.58×	1.57×
	8	1,077.81	1.45×	1.48×
USA	1	126,510.85	71.36×	166.27×
	2	102,076.20	58.11×	142.57×
	4	64,461.77	34.59×	89.94×
	8	32,008.69	16.22×	39.34×

Table 4 compares the performance of the soft-priority scheduler and fused worker kernel with the unoptimized version of Groute's distributed worklists. In the table, we see that both versions consistently outperform the original implementation, with the exception of *soc-LiveJournal1* (which slows down 5% on two GPUs). The most compelling results can be found in road maps, in which we see performance increase of up to two orders of magnitude.

Soft priorities improve performance by reducing the amount of "useless" work that is performed. This increase in useless work can be dramatic. For example, SSSP works on 13,998M work-items on the *USA* graph in a complete execution on a single GPU. Without soft-priorities, this increases to 15,865M work-items on four GPUs. With soft-priorities, the four-GPU version executes only 59M work-items overall. For SSSP, this reduction is comparable to those obtained by using an SSSP algorithm with priorities such as SSSP Near-Far [12]. While BFS has no notion of priorities, it too exhibits the same effect. Single-GPU BFS on *USA* executes 23.9M work-items. Without soft-priorities, this increases to 4,244M work-items on four GPUs (27M with METIS). With soft-priorities this reduces to 134M work-items on four GPUs (24.1M with METIS).

Kernel fusion, however, tackles a problem exhibited by graphs such as road maps that create small workloads with each kernel call (~19 microseconds), causing the GPUs to be under-utilized and increasing the communication management overhead. We augment the worker kernel to include the entire control-flow and communicate with the host and other GPUs using flags shared by both the CPU and GPU. This includes receiving incoming information signals, determining work-item priorities, processing a batch of work-items, running `SplitSend` (Section 4.3), and signaling the router to circulate the outgoing information. By performing this kernel fusion, many of the CPU–GPU roundtrips can be reduced, increasing the overall performance of the system, even for a single GPU. In practice, kernel fusion in Groute increases the work performed by each kernel invocation, which takes between ~10 and 100 milliseconds.

In both optimizations, we observe diminishing returns when increasing the number of GPUs. While the unoptimized version scales at a certain rate, soft-priority scheduling and kernel fusion strongly scale at a lower rate on all input graphs. As with the performance improvement, this effect could be the result of the number of work-items as well—the unoptimized algorithm scales better, because it performs more work, whereas soft-priority scheduling decreases the overall work (on

Table 5. Heterogeneous PBF Performance

Scheduler	GPU Type	Processed Elements	Time (ms)
Static	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

any number of devices) and thus starts encountering scaling issues with fewer GPUs than the baseline version.

5.3 Load Balancing

Table 5 measures the performance of the PBF implementation from Figure 6 on the heterogeneous two-GPU node, filtering 250 MB of data. The runtime of each GPU is shown with static scheduling (top three rows) and Groute's "first available device" routing policy (bottom three rows).

The table shows that Groute assigns 4× more tasks to the faster Tesla K40c than the slower Quadro M4000, achieving better load balancing and decreasing overall runtime. Note that the two-millisecond time difference observed in Groute is within the scheduling quantum, as it is shorter than the runtime of a single kernel on the Quadro M4000.

6 RELATED WORK

Groute is primarily an asynchronous communication substrate for multiple GPUs. In this work, we have demonstrated its utility for asynchronous graph processing using the ability to use multiple GPUs to process graphs that are beyond any individual GPU's capacity.

Groute's link/router programming model can be seen as a close relative of the *Publish/Subscribe* design pattern [14], in which endpoints subscribe to specific channels that other endpoints publish to. The link/router model is different in that it defines generalized policies, which are more optimized for low-latency communication on multi-GPU nodes than named channels.

Recently, multi-GPU frameworks to simplify programming and provide reusable mechanisms have been proposed. Notable examples include NCCL [31], which implements collective operations on a single node; MGPU [37], which simplifies task partitioning to multiple GPUs; and MAPS-Multi [6], which proposes a scalable programming model based on memory access patterns. Owing to the traditional bulk-synchronous use of multi-GPU nodes, these libraries focus on the efficient implementation of regular computations, such as stencil operators, rather than irregular algorithms.

Data-driven graph algorithm implementations use worklists for processing [29]. These implementations were found, in the general case, to be faster than their topology-driven counterparts on GPUs [28]. These results motivated implementing graph analytics using distributed worklists in this article.

Asynchronous graph processing frameworks have been explored for the CPU. Pearce et al. [35] describe a system that performs asynchronous BFS, SSSP, and CC on a shared memory system, augmented with solid-state disks to handle very large graphs. They also use priority queues per worker, but their use of shared memory means that they do not suffer as much from stale value

propagation (Section 4.4). Unlike this work, we operate in distributed memory and have to deal with stale values.

Galois [30] proposes a work-stealing scheduler for asynchronous multi-core CPU processing implying the use of distributed worklists. Recently, Galois has been extended to multiple GPUs using D-IrGL [11]. However, D-IrGL uses bulk synchronous execution on distributed GPUs using parallel gather and scatter orchestrated by the CPU to keep data consistent.

Harshvardhan et al. [17] present an interesting approach for handling large graphs when using vertex-centric programming models. They partition the graphs into subgraphs, which are then operated upon independently. Updates to the current subgraph are applied immediately, but other updates are buffered and applied asynchronously when the subgraph containing the updated nodes is read. However, only the updates are asynchronous—the graph algorithm remains synchronous (except it can choose to perform synchronization every k steps for some k).

Concurrent graph analytics on a single GPU (using multiple streams) has also been proposed in GTS [23], showing that this type of programming is promising for single-GPU applications as well. Additional single-GPU [9, 15, 20, 43] and multi-GPU [27, 34] graph analytics libraries have been proposed. However, as opposed to our asynchronous approach, these implementations all utilize bulk-synchronous parallelism. The distributed worklist kernel fusion optimization proposed in Section 5 is similar to the Megakernel single-GPU approach, proposed by Steinberger et al. [39], which also transfers portions of the control flow to the GPU.

Lux [21] is recent system for multi-GPU graph processing that also takes into account the memory hierarchy in multi-GPU systems. However, Lux only supports vertex programs, which only operate on a node's edges and its immediate neighbors. This precludes its use for more sophisticated graph algorithms.

Graphie [16] is a system that implements traversal-based algorithms for large graphs on a single GPU. It uses asynchronous CUDA transfers to stream edge data to the GPU. This allows it to process graphs that do not fit in the memory of a single GPU. Graphie does not support worklists and uses a flag array to detect active vertices in the graph.

MultiGraph [18] focuses on processing the sparse and dense subsets of a graph using different execution strategies. Compared to Groute, MultiGraph uses synchronous execution and is limited to graphs that can fit on a single GPU. Under these constraints, it can outperform some of our distributed implementations running on a single GPU.

7 CONCLUSIONS

The article presented a scalable programming abstraction and runtime environment for asynchronous multi-GPU application development. In-depth study of the structure of a multi-GPU node showed that creating such applications requires careful tuning with respect to communication topology and workload processing, particularly in irregular algorithms, where lagging information may have a major impact on scaling. The article then showed that the programming abstraction is simple yet expressive, enabling the efficient implementation of complex graph analytic algorithms, showing strong scaling results.

This research can be extended in several directions. First, the link/router abstraction concepts can be generalized to other non-GPU architectures as well as distributed systems. Second, the majority of the router control flow relies on host-based decisions. Similarly to worker kernel fusion, moving these decisions to a device-based router may decrease system overhead by further reducing CPU-GPU copies. Third, load balancing in multi-GPU traversal algorithms may be improved by employing asynchronous work-stealing schedulers and dynamically changing node ownership.

REFERENCES

[1] Camil Demetrescu, Andrew Goldberg, and David Johnson. 2006. 9th DIMACS Implementation Challenge. Retrieved from http://www.dis.uniroma1.it/challenge9.

[2] Groute Authors. 2017. Groute Runtime Environment Source Code. Retrieved from https://www.github.com/groute/groute.

[3] Karlsruhe Institute of Technology. 2014. OSM Europe Graph. Retrieved from http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php.

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

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner (Eds.). 2013. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop.* Contemporary Mathematics, Vol. 588. American Mathematical Society.

[6] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. 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).* ACM, Article 19, 12 pages.

[7] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17).* ACM, New York, NY, 235–248. DOI: https://doi.org/10.1145/3018743.3018756

[8] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'17).* ACM, New York, NY, 93–104. DOI: https://doi.org/10.1145/3078597.3078616

[9] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'12).* 141–151.

[10] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and P. Krishna Gummadi. 2010. Measuring user influence in Twitter: The million follower fallacy. In *Proceedings of the International Conference on Web and Social Media* 10, 10–17 (2010), 30.

[11] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, 752–768. DOI: https://doi.org/10.1145/3192366.3192404

[12] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium.* 349–359.

[13] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (2011), 25 pages.

[14] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (2003), 114–131.

[15] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. 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).* ACM, 345–354.

[16] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17).* IEEE, 233–245.

[17] Harshvardhan, B. West, A. Fidel, N. M. Amato, and L. Rauchwerger. 2015. A hybrid approach to processing big data graphs on memory-restricted systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium.* 799–808. DOI: https://doi.org/10.1109/IPDPS.2015.28

[18] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. 2017. MultiGraph: Efficient graph processing on GPUs. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17).* IEEE, 27–40.

[19] Pei-Yao Hong, Li-Min Huang, Li-Song Lin, and Chao-An Lin. 2015. Scalable multi-relaxation-time lattice Boltzmann simulations on multi-GPU cluster. *Comput. Fluids* 110 (2015), 1–8.

[20] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11).* ACM, 267–276.

[21] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *Proc. VLDB Endow.* 11, 3 (2017), 297–310.

[22] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

[23] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*. ACM, 447–461.

[24] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority queues are not good concurrent priority schedulers. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing (Euro-Par'15: Parallel Processing)*. Springer Berlin, 209–221.

[25] B. Liskov and L. Shrira. 1988. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*. 260–267.

[26] Edgardo Mejía-Roa, Daniel Tabas-Madrid, Javier Setoain, Carlos García, Francisco Tirado, and Alberto Pascual-Montano. 2015. NMF-mGPU: Non-negative matrix factorization on multi-GPU systems. *BMC Bioinformatics* 16, 1 (2015), 43.

[27] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. 117–128.

[28] R. Nasre, M. Burtscher, and K. Pingali. 2013. Data-driven versus topology-driven irregular computations on GPUs. In *Proceedings of the IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS'13)*. 463–474.

[29] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Morph algorithms on GPUs. *ACM SIGPLAN Not.*, Vol. 48. ACM, 147–156.

[30] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 456–471.

[31] NVIDIA. 2016. NVIDIA Collective Communication Library (NCCL). Retrieved from http://www.github.com/NVIDIA/nccl/.

[32] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM.

[33] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2015. Multi-GPU graph analytics. *CoRR* abs/1504.04804 (2015).

[34] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2017. Multi-GPU graph analytics. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. IEEE, 479–490.

[35] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE Computer Society, Washington, DC, 1–11. DOI: https://doi.org/10.1109/SC.2010.34

[36] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, 12–25.

[37] Sebastian Schaetz and Martin Uecker. 2012. 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)*. Springer-Verlag, 114–128.

[38] J. Soman, K. Kishore, and P. J. Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*. 1–8.

[39] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (2014), 11 pages.

[40] Michael Sutton, Tal Ben-Nun, and Amnon Barak. 2018. Optimizing parallel graph connectivity computation via subgraph sampling. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*.

[41] Michael Sutton, Tal Ben-Nun, Amnon Barak, Sreepathi Pai, and Keshav Pingali. 2016. Adaptive work-efficient connected components on the GPU. *CoRR* abs/1612.01178 (2016).

[42] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[43] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2015. 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'15)*. 265–266.

[44] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. 2015. Scalable data-driven pageRank: Algorithms, system issues, and lessons learned. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing (Euro-Par'15: Parallel Processing)*, Larsson Jesper Träff, Sascha Hunold, and Francesco Versaci (Eds.). Springer Berlin, 438–450.

[45] Derek Wilson. 2009. Triple buffering: Why we love it. Retrieved from http://www.anandtech.com/show/2794.

Received July 2018; revised October 2019; accepted December 2019