Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations

Tal Ben-Nun Michael Sutton
The Hebrew University of Jerusalem, Israel
{talbn, msutton}@cs.huji.ac.il

Sreepathi Pai Keshav Pingali
The University of Texas at Austin, USA
sreepai@ices.utexas.edu pingali@cs.utexas.edu

Abstract

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 paper 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 8-GPU and heterogeneous systems, yielding over 7x speedup for some algorithms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Multi-GPU, Asynchronous Programming, Irregular Algorithms

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 power efficiency, 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 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.

PPoPP '17, February 04-08, 2017, Austin, TX, USA
 ISBN 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018743.3018756

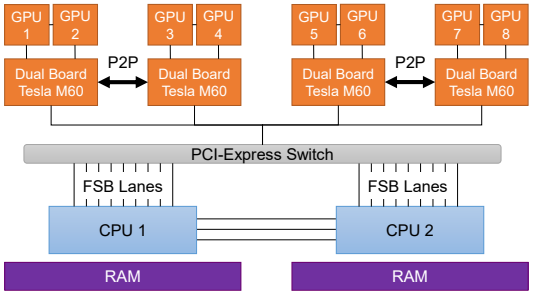


Figure 1: Multi-GPU Node Schematic

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 [13, 19]. Alternatively, a Bulk Synchronous Parallel (BSP) [32] programming model is used, in which applications are executed in rounds, and each round consists of local computation followed by global communication [6, 26]. The first approach is subject to overheads from various sources, such as the operating system, and requires a message-passing interface for communication. The BSP model, on the other hand, 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 these 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 paper 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.

The main contributions of this paper 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 7.28× speedup on 8 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, 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 can either be initiated by the host or a device. In particular, 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.

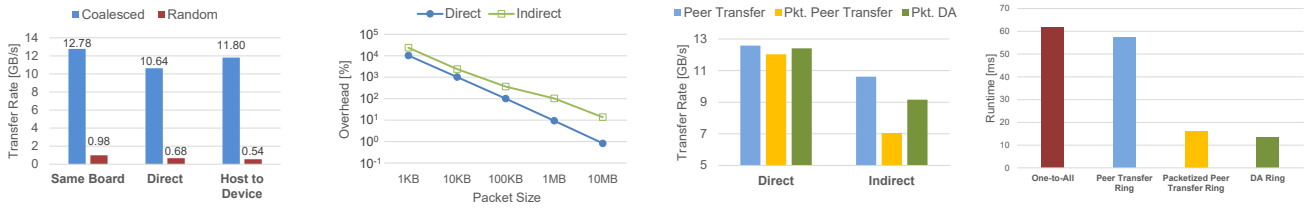
Device-initiated memory transfers are implemented by virtual addressing, which maps all host and device memory to a single address space. While more flexible than peer transfers, DA performance is highly sensitive to memory alignment, coalescing, number of active threads and order of access.

Using microbenchmarks (Figure 2) we measure 100 MB transfers, averaged over 100 trials, on the 8-GPU system from our experimental setup (see Section 5 for detailed specifications).

Figure 2a 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), CPU-GPU transfer is faster than two different-board GPUs.

In order 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, 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 2b 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 ∼1-10% for 1-10 MB packets. This parameter can be tuned by individual applications to balance between latency and bandwidth.



(a) Direct memory access order (b) Packetization overhead (c) Packetized transfer rate (d) Peer broadcast performance

Figure 2: Inter-GPU Memory Transfer M

Figure 2c 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 [24], creates a ring topology over the bus. In this approach, illustrated in Figure 3, each GPU transfers to one destination, communicating via direct or indirect device-initiated transfers. This ensures that both memory copy engines of every GPU are use
to
bei

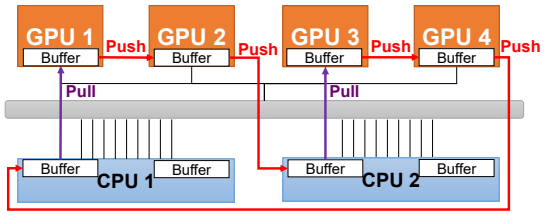


Figure 3: DA Ring Topology

Figure 2d compares the performance of the different methods of implementing one-to-all GPU peer broadcast, ranging from 7 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 2c.

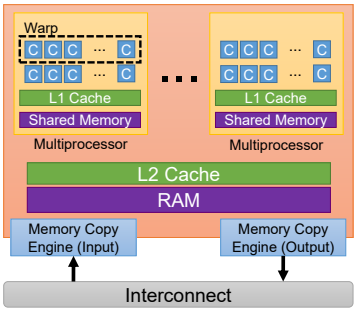


Figure 4: Single GPU Architecture

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 comprised 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

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.

abstraction that complements the existing model to facilitate multi-GPU development, using insights from the microbenchmarks to minimize communication latency.

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.

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 like 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., a router can only have one outgoing edge to the same endpoint). Note that to support multitasking, multiple virtual endpoints can be created from the same physical device.

`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, the input can be sent to a single endpoint, or to a subset of endpoints according to their availability. When a *link* is created,

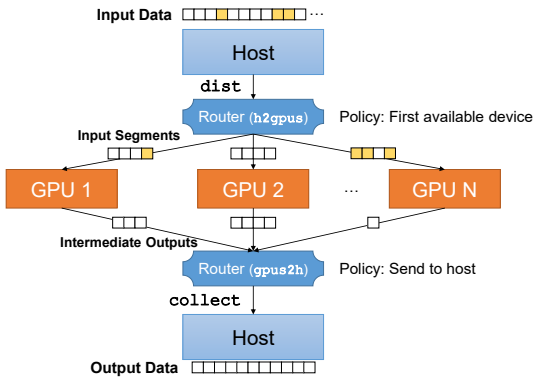


Figure 5: Predicate-Based Filtering Dataflow Graph

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 5-20. 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 15-20 creates a worker-thread for each GPU using double-buffered links, and input segments are scattered to the devices (line 22). Upon distributing all input segments, the distributor notifies that it will send no further information by sending a shutdown signal (line 23). The result is obtained at the host (lines 25-31) and the program stops once all GPUs send shutdown signals to `gpus2h`, notifying that no additional data will be received (line 27).

The routing policy for both routers is straightforward, selecting the first available device out of all possible router destinations (line 36). On the GPU end, each device asynchronously handles incoming messages (line 45). Once a `PendingSegment` is assigned to the device, it is synchronized with the active GPU stream (line 47) and processing

```
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 }
```

Figure 6: Predicate-Based Filtering Pseudocode

is performed (line 48). Line 50 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 51). When a shutdown signal is received, the worker thread is terminated (line 46) and a shutdown signal is sent (line 53) to `gpus2h`.

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 as a distributed memory environment. In the paper, 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.

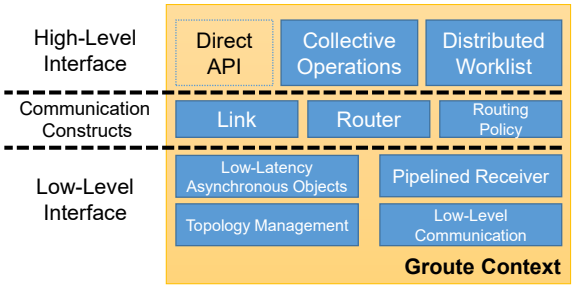


Figure 7: The Groute Library

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. 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 [35]. 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

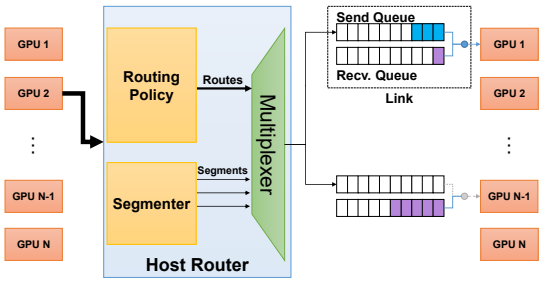


Figure 8: Host-Controlled Router Diagram

buffer is assigned to 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. In order 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 [18] 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 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. In order to support efficient all-to-all communication, the ring topology is used for the routing policy by default (see Section 2 for evaluation). Apart from the router, each device contains a locally-managed worklist, which comprises one or more multiple-producer-single-

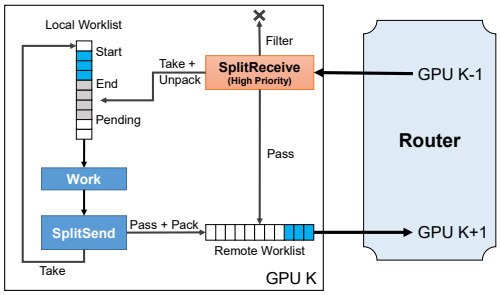


Figure 9: Distributed Worklist Implementation

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 undergoes 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.

In order to implement an algorithm over a distributed worklist, five functions must be given by the programmer, listed in Table 2: `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.

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,

Table 2: Distributed Worklist Programmer Callbacks

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`.

`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 [17].

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. Upon completion of all processable items, the system modifies the threshold and the distributed worklist will process the deferred items on each device. As

Table 3: Best Performance Comparison

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)

we shall show in Section 5, using soft priority scheduling decreases the amount of intermediate work, increasing overall performance.

4.5 Worklist-Based Graph 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.

In BFS, the input graph is given in the Compressed Sparse Row (CSR) matrix format. 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.

When BFS processing starts, the host enqueues a single work-item to the worklist – the source vertex. Groute 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. 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. 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. After all relevant edges have been traversed, the worklist becomes empty and the algorithm ends.

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 conserving inter-device communication. Skipping such updates does not change the algorithm's behavior, due to the monotonic nature of BFS, i.e., the real vertex value is either equal or lower than the halo value.

5. Performance Evaluation

This section evaluates the performance of five algorithms implemented using Groute:

- **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 [34]. Implementation is the push-based variant proposed in the paper.
- **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 to two benchmark implementations of multi-GPU parallel graph algorithms: Gunrock (version 0.3.1) and Back40Computing (B40C). Gunrock [26] 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. [20], contains state-of-the-art hardcoded BFS implementations, enabling multi-GPU processing by direct memory accesses between peer GPUs.

All implementations, including Groute, contain the following kernel 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 [25]. Additionally, Groute's asynchronous model allows us to perform kernel fusion (Section 5.2).

Table 3 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.

Our experimental setup consists of two node types. The first is an 8-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, with 2 QPI links per CPU at 8 GT/s per link for the PCI-Express switch.

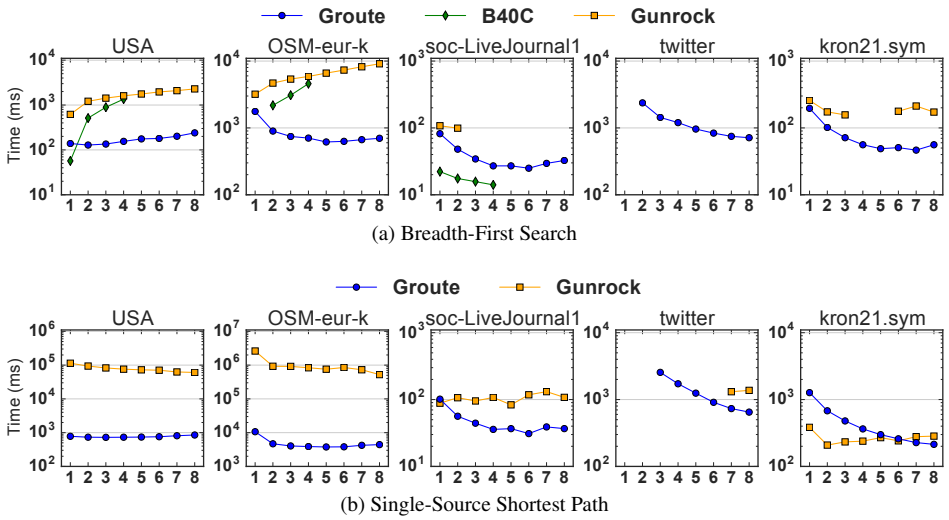


Figure 10: Traversal Algorithm Timing (lower is better)

The second node type is a 2-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 2 total QPI links at 7.2 GT/s per link.

Table 4: 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 [10]	5M	69M	14.23	20,293	0.56
twitter [8]	51M	1,963M	38.37	779,958	16.00
Synthetic Graphs					
kron21.sym [5]	2M	182M	86.82	213,904	1.40

Table 4 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 [15] 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 10 presents the absolute runtime of the two graph traversal algorithms (BFS and SSSP), running on 1 to 8 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 4-GPU quadruplet. While Groute mitigates these issues by optimizing communication 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 10a compares Groute with Gunrock and B40C. B40C's multi-GPU implementation requires direct memory access between all devices, and thus only runs up to 4 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* inputs, and produced incorrect results on *kron21.sym* and *soc-LiveJournal1*.

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 [20], an optimization we did not implement due to its highly specialized nature.

5.1.2 Single-Source Shortest Path

Figure 10b 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 4 GPUs. Missing *twitter* results are caused by insufficient memory.

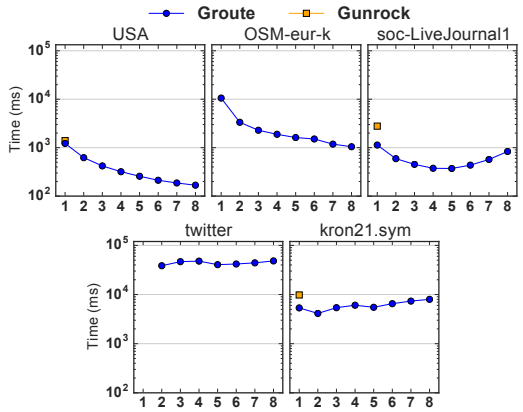


Figure 11: PageRank Execution Time

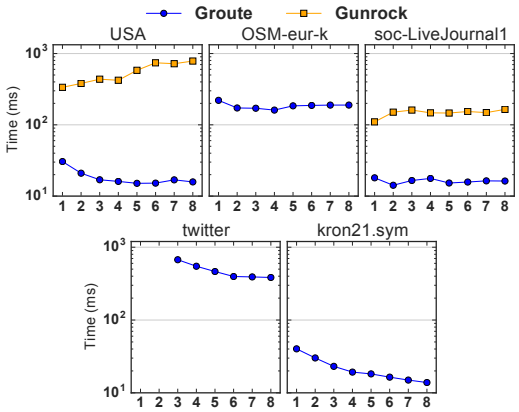


Figure 12: Connected Components Performance

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 implementation in Groute causes an inflation in the number of performed atomic operations, which increases memory contention and iteration time.

5.1.3 PageRank

The performance of PageRank (PR) is shown in Figure 11. Results are compared with Gunrock in the single GPU case, since the evaluated version of Gunrock's multi-GPU PageRank produced incorrect results. Also, the *twitter* graph does not fit in the memory of 1 GPU.

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. Additionally, 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 7.28× speedup on 8 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 3.02× speedup on 4 GPUs over one. Additionally, nearly no speedup is observed in *twitter* and *kron21.sym*, both of which are partitioned randomly (i.e., without METIS) and heavily interconnected.

5.1.4 Connected Components

We implement a topology-driven [27] variant of multi-GPU Connected Components (CC), which does not use a worklist, in order to demonstrate the expressiveness of the Groute asynchronous communication constructs. The performance of CC is shown in Figure 12.

Figure 13 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. Once a GPU converges locally, its results are merged (using operations called Hook and Compress) with the results of other GPUs to converge to the global component list.

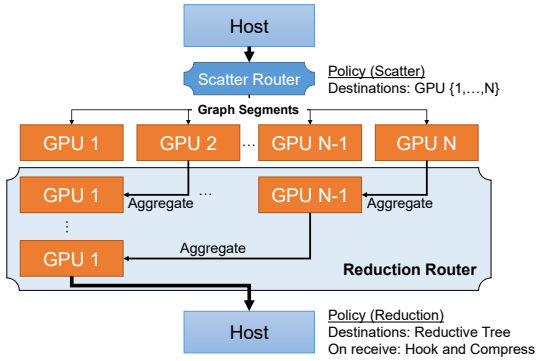


Figure 13: Connected Components Router Structure

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 8-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 [31] of the pointer-jumping method described by Soman et al. [29].

The results in Figure 12 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 10.99× and 49.6× speedups over Gunrock in *USA* on 1 and 8 GPUs respectively. Furthermore, Groute achieves a strong scaling of up to 2.9× (8 GPUs over 1) 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*, *twitter* and *kron21.sym*. Gunrock, on the other hand, runs out of memory for all GPU configurations on these three 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 5: Distributed Worklist SSSP Performance

Graph	GPUs	Soft Priority Scheduler	Fused Worker
soc-LiveJournal1	1	1.03×	1.03×
	2	0.95×	0.95×
	4	1.29×	1.31×
	8	1.29×	1.09×
kron21.sym	1	1.02×	1.01×
	2	1.10×	1.09×
	4	1.58×	1.57×
	8	1.45×	1.48×
USA	1	71.36×	166.27×
	2	58.11×	142.57×
	4	34.59×	89.94×
	8	16.22×	39.34×

Table 5 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 2 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 workitems on the *USA* graph in a complete execution on a single GPU. Without soft-priorities, this increases to 15,865M workitems on four GPUs. With soft-priorities, the four GPU version executes only 59M workitems overall. For SSSP, this reduction is comparable to those obtained by using an SSSP algorithm with priorities such as SSSP Near-Far [9]). However, BFS has no notion of priorities, but also 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 workitems on four GPUs (24.1M with METIS).

Kernel fusion, on the other hand, tackles a problem exhibited by graphs such as road maps which 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. In practice, kernel fusion in Groute increases the work performed by each kernel invocation, which take between ∼10 and 100 milliseconds. Note that the reduction of CPU–GPU roundtrips causes both optimizations to also improve the runtime of a single GPU.

5.3 Load Balancing

Table 6 measures the performance of the PBF implementation from Figure 6 on the heterogeneous 2-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).

Table 6: 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

The table shows that Groute assigns 4 times more tasks to the faster Tesla K40c than the slower Quadro M4000, achieving better load balancing and decreasing overall runtime. Note that the 2 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

The presented link/router programming model can be seen as a close relative of the *Publish/Subscribe* design pattern [11], in which endpoints subscribe to specific channels that other endpoints publish to. The link/router model differs from this model by defining generalized policies, which are more optimized for low-latency communication on multi-GPU nodes than named channels.

Recently, multi-GPU frameworks that simplify programming and provide reusable mechanisms have been proposed. Notable examples include NCCL [24], which implements collective operations on a single node; MGPU [28], 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 [22]. These implementations were found, in the general case, to be faster than their topology-driven counterparts on GPUs [21]. These results motivated implementing graph analytics using distributed worklists in this paper.

Other asynchronous graph processing frameworks have been researched. Galois[23] proposes a work-stealing scheduler for asynchronous multi-core CPU processing. Concurrent graph analytics on a single GPU (using multiple streams) has also been proposed in GTS [16], showing that this type of programming is promising for single-GPU applications as well. Additional single-GPU [7, 12, 14, 33] and multi-GPU [20, 26] 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. [30], which also transfers portions of the control flow to the GPU.

7. Conclusions

The paper 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 paper 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.

Acknowledgments

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.

References

[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/resources/roadgraphs.php`.

[4] A. Adinetz. Optimized filtering with warp-aggregated atomics. 2014. URL `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-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. Work-efficient parallel GPU methods for single-source shortest

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. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. 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 multi-GPU 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. Mejía-Roa, D. Tabas-Madrid, J. Setoain, C. García, F. Tirado, and A. Pascual-Montano. NMF-mGPU: non-negative 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 Twenty-Fourth 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. Multi-GPU 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. Méndez-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. Springer-Verlag, 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. Träff, S. Hunold, and F. Versaci, editors, *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`.

A. Artifact description

A.1 Abstract

This artifact contains all the source code necessary to compile the Groute executables and repeat the results of this paper. The package also contains shell scripts to generate the figures and tables as CSVs, obtain code dependencies, and download input graphs for the benchmarks.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithm:** Micro-benchmarks, Breadth-First Search, Single-Source Shortest Path, PageRank, Connected Components, Predicate-Based Filtering.
- **Compilation:** Using CUDA (`nvcc`) for GPU code and GCC (`gcc`, `g++`) for host code. Both compilers use the following flags: `-O3 -DNDEBUG -std=c++11`.
- **Binary:** One CUDA executable for each algorithm.
- **Data set:** Publicly-available graphs converted to binary CSR (Galois) format.
- **Run-time environment:** Debian 8 Linux with CMake 3.2, GCC 4.9.3, and CUDA 7.5. Optional requirement (for comparing with Gunrock): Boost 1.55.0.
- **Hardware:** CUDA-capable GPU with compute capability of at least 3.5. Multiple GPUs on the same node are required for multi-GPU tests.
- **Output:** CSV files representing each figure and table.
- **Experiment workflow:** Clone repository, setup environment, run figure-generating shell scripts, review generated CSV files.
- **Publicly available?:** Yes.

A.2.2 How delivered

A Git repository that contains all the Groute code and figure-generating scripts can be found in:
`http://www.github.com/groute/ppopp17-artifact`
The repository contains shell scripts that set up the environment, obtain and compile external code, download the dataset, and run the figure generators. See additional information in Sections A.3 and A.4.

A.2.3 Hardware dependencies

To run GPU-based tests, Groute requires one or more NVIDIA GPUs with compute capability of at least 3.5. Multi-GPU tests will not run on a single-GPU node. For the test performed in Table 6, a system with two heterogeneous GPUs (i.e., one faster than the other) is recommended to repeat the results.

A.2.4 Software dependencies

To compile Groute, CMake 3.2, GCC 4.9 and CUDA 7.5 are required. Later versions can be used but were not tested by the authors.

The code contained in the above repository includes the following external dependencies: METIS 5.1.0, gflags 2.1.2, MGBench 1.01, NCCL 1.2.3, and Gunrock 0.3.1 (optional).

A.2.5 Datasets

The dataset for this artifact contains all graphs listed in Table 4 of the paper, converted to the Galois CSR binary format (.gr files). The dataset (10.6 GB compressed, 29 GB uncompressed) is automatically downloaded by the setup script. Each graph is located in a separate subdirectory, with three additional files:

1. `<graph>/<graphfile>.metadata`: Metadata of the graph properties (number of connected components, whether to use METIS partitioning, soft-priority delta).
2. `<graph>/bfs-<graphfile>.txt`: Externally generated BFS results (source node: 0), used for validation.
3. `<graph>/sssp-<graphfile>.txt`: External SSSP results (source node: 0), also used for validation.

A.3 Installation

To install, follow the instructions below:

- Clone the Git repository recursively from `https://github.com/groute/ppopp17-artifact.git`
- Run `setup.sh`. The script will automatically compile Groute and the external code.
- When prompted whether to download the dataset, respond 'y' or 'n'. The dataset can be downloaded manually by running `dataset/download.sh` at a later time.

A.4 Experiment workflow

After setting up the environment, either run `runall.sh` to generate all figures and tables, or run each individual figure separately by calling `figures/figureXX.sh`, where `XX` is the figure number. The overall workflow as a list of shell commands is as follows:

```
1 $ git clone --recursive
2   https://github.com/groute/ppopp17-artifact.git
3 $ cd ppopp17-artifact/
4 $ ./setup.sh
5 $ ./runall.sh
6 $ cat output/figure10b.csv
```

Note: Gunrock execution (for comparison with Groute) is not enabled by default. To enable, `export RUN_GUNROCK=1` prior to running the figure-generating scripts.
Note 2: Each individual test may take time, but is limited to 2 hours to avoid waiting forever for a faulty application. For slower GPUs, this timeout can be increased by modifying the `TIMEOUT` variable from `2h` to a different value (line 4 in `figures/common.sh`).

A.5 Evaluation and expected result

The results, found in `output/`, are in Comma-Separated Values (CSV) format and represent each figure and table in the paper (e.g., `figure2a.csv`, `table6.csv`). Prior to being inserted to the CSV, the results are averaged over at least 3 runs. The results will also be written to the console during the process.