Scalable Irregular Parallelism with GPUs: Getting CPUs Out of the Way

Yuxin Chen
University of California, Davis
Davis, USA
yxxchen@ucdavis.edu

Benjamin Brock
University of California, Berkeley
Berkeley, USA
brock@cs.berkeley.edu

Serban Porumbescu
University of California, Davis
Davis, USA
sdporumbescu@ucdavis.edu

Aydın Buluç
Lawrence Berkeley National Laboratory
Berkeley, USA
abuluc@lbl.gov

Katherine Yelick
University of California, Berkeley
Berkeley, USA
yelick@berkeley.edu

John D. Owens
University of California, Davis
Davis, USA
jowens@ece.ucdavis.edu

***Abstract*—We present Atos, a dynamic scheduling framework for multi-node-GPU systems that supports PGAS-style lightweight one-sided memory operations within and between nodes. Atos's lightweight GPU-to-GPU communication enables latency hiding and can smooth the interconnection usage for bisection-limited problems. These benefits are significant for dynamic, irregular applications that often involve fine-grained communication at unpredictable times and without predetermined patterns. Some principles for high performance: (1) do not involve the CPU in the communication control path; (2) allow GPU communication within kernels, addressing memory consistency directly rather than relying on synchronization with the CPU; (3) perform dynamic communication aggregation when interconnections have limited bandwidth. By lowering the overhead of communication and allowing it within GPU kernels, we support large, high-utilization GPU kernels but with more frequent communication. We evaluate Atos on two irregular problems: Breadth-First-Search and PageRank. Atos outperforms the state-of-the-art graph libraries Gunrock, Groute and Galois on both single-node-multi-GPU and multi-node-GPU settings.**

***Index Terms*—PGAS, distributed GPUs, asynchronous, irregular application**

I. INTRODUCTION

The bulk-synchronous communication model [1] is one of the most popular models for distributed CPU computing. By splitting up applications into bulk phases of communication and computation, the bulk synchronous parallelism (BSP) model produces communication patterns that are a good match for modern inter-node communication networks, which require large messages to achieve peak communication bandwidth. The BSP model performs particularly well for programs with regular, fixed communication patterns, but can encounter difficulties when communication volume and timing becomes irregular and dynamic.

More recently, the distributed implementations of many Partitioned Global Address Space (PGAS) languages such as UPC, UPC++ [2], and OpenSHMEM [3] have found success in adopting an alternative communication model: one that issues many independent small communication messages. This model works well in applications where independent computation can be scheduled simultaneously with communication, allowing these applications to overlap communication latency with computation time. In addition, small one-sided communication removes the need for remote processor synchronization, and thus reduces overall synchronization overhead. And small communications can often be evenly spread over the runtime of the program, thus smoothing network usage, compared to the more variable network usage in many bulk-synchronous programs. These benefits are especially significant for dynamic, irregular applications that often involve fine-grained communication at unpredictable times and without predetermined patterns.

These advantages often allow PGAS implementations to achieve speedups over the traditional BSP approach [4]–[8]. The chief disadvantage of small-grained communication is lower bandwidth utilization, but in spite of this, Bell et al. [9] showed that this communication model can provide significant performance advantage even on interconnection bandwidth-limited applications.

Despite these PGAS successes on distributed CPUs, they are still not the most common target on distributed GPUs for three reasons:

1) Historically, GPU memory was not directly accessible by other GPUs in a distributed system. This precluded efficient, fine-grained asynchronous GPU-to-GPU communication, since these had to be routed through the CPU, increasing latency.
2) Traditionally, GPU communication has taken place at kernel boundaries. Relaxing this restriction significantly complicate data consistency. A single large kernel is unabled to achieve much overlap, since communication becomes much more coarse-grained, while running many small kernels usually leads to significant kernel launch overheads and low GPU utilization.
3) The most prevalent formulations of many distributed algorithms use the BSP model. These formulations may not map well to one-sided communication. In particular, algorithmic formulations with more inherent

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/31.00 ©2022 IEEE

asynchronous operations are better suited for one-sided communication, but many (BSP) formulations may not be structured in this way.

Consequently, GPUs on distributed systems have predominantly been used in a bulk-synchronous way for both computation and communication. A PGAS-like programming model however, can be a superior alternative especially for *irregular applications*: those with varying communication and synchronization patterns. Thus, in this work, we describe "Atos", a PGAS-style framework for asynchronous execution across multiple GPUs in a cluster. Atos supports fine-grained, one-sided communication, along with the following features:

1) Though direct GPU-to-GPU *data movement* is now widespread on modern systems, the communication *control path* (including communication preparation and triggering, possibly matching and synchronization) is even today typically run on CPUs. Our work moves the communication control path to the GPU and achieves authentic one-sided communication between GPUs by leveraging NVIDIA's OpenSHMEM-based NVSHMEM, which maps GPU memories to the NIC (and vice versa) so that GPU memory is directly accessible by remote GPU threads without local or remote CPU involvement.
2) Traditionally, ensuring data consistency before communication is implemented by synchronizing CUDA kernels from the CPU. This implicitly couples data consistency with synchronization. To avoid this coupling and the resulting overheads, Atos implements asynchronous distributed queues to ensure data consistency and enable one-sided kernel communication without any need for synchronization.
3) Communication traditionally occurs only after synchronization of CUDA compute kernels. This forces a trade-off between (a) large, high-utilization GPU kernels with delayed messages and little communication-computation overlap and (b) small, low-utilization GPU kernels with more frequent communication and more communication-computation overlap. Atos avoids this tradeoff by enabling high-utilization GPU kernels with fine-grained lightweight communication from *within* the kernel. This leads to more overlap of communication and computation, as smaller communication sizes make it easier to find sufficient computation to hide latency.
4) Enabling communication without explicit synchronization enables the deployment of synchronization-free algorithms, with potential performance benefits due to reduced synchronization cost. In contrast, traditional BSP approaches on the GPU preclude the use of synchronization-free algorithms.
5) Finally, our approach is robust enough to work efficiently without changes on multi-GPU systems with two different interconnect families, NVLink and InfiniBand (IB).

Atos's approach is distinct enough from traditional approaches that we can evaluate our system and show appreciable differences with previous work even with relatively simple algorithms. For our evaluation, we implemented breadth-first search (BFS) and PageRank and compare our single-node, multi-GPU NVLink implementation against Groute [10] and Gunrock [11] and our multi-node, multi-GPU InfiniBand (IB) implementation against Galois [12].

In our work, we examine NVLink and InfiniBand.

a) NVLink: Using an NVLink interconnect, we show that the fine-grained one-sided communication in our BFS and PageRank implementations enables superior performance and scalability compared to other frameworks because of better latency hiding and better use of the network.

b) IB: InfiniBand is less efficient for small messages than NVLink. To optimize communication for InfiniBand, Atos implements a communication aggregator that runs concurrently with application code, transparently aggregating messages together and sending them off in larger message bundles. We show that this aggregator effectively addresses bandwidth underutilization on an IB system due to fine-grained one-sided communication. Our Atos BFS and PageRank implementations also outperform other frameworks with respect to runtime and scalability on an IB-based system.

II. PGAS PROGRAMMING MODEL ON GPUS

Many of the fastest supercomputers available today use GPUs for the bulk of their computational performance, while CPUs are responsible for communication. GPUs achieve high performance and energy efficiency through simple throughput-optimized cores and parallelism, but they have limited support for OS-level features such as interrupts, single helper threads, or the large memory space needed for dynamic message buffering. Common communication protocols, such as two-sided send/receive or collective communication, are complex and require message matching and synchronization; these typically run on CPUs but are a significant challenge to implement efficiently on GPUs.

PGAS programming models instead feature one-sided communication primitives in the form of implicit communication (pointer and array references) and explicit put and get calls. This approach is a good fit for modern GPU-to-GPU communication mechanisms such as NVLink or PCIe within a single node. Recent advances enable GPUs in separate nodes to perform one-sided memory operations directly into and out of the InfiniBand network. Furthermore, both PGAS and the GPU use a relaxed memory model, which automatically unifies the guarantees for ordering, synchronization, and atomic operations.

On distributed CPU systems, from the memory model perspective, the PGAS model allows users to directly access the union of shared memories across nodes. Users can access global pointers and distributed arrays and connect them into a single distributed data structure. From the programming model perspective, PGAS languages use a SPMD approach where a fixed set of processes start together at the beginning of the computation and terminate together at the end. Some PGAS frameworks such as Legion use a sophisticated runtime

to map work to hardware automatically with a default task mapper or a user-provided mapper. The use of accelerators complicates the PGAS programming model because GPUs have a hardware scheduler internally (scheduling CTA to SMs within a kernel, or concurrently running two kernels). PGAS has a uniform model for parallelism within and between nodes, but does not have a natural way to deal with GPUs. Our work here proposes a single integrated programming model that is efficient, flexible, and principled, contrasting against the current disjoint model for on-node GPU parallelism that is tacked onto a SPMD model between nodes.

Previous asynchronous multi-GPU frameworks have not yet reached this goal. For example, Galois [12], PTask [13], and StarPU [14] each use a similar approach: treating each GPU as the equivalent of a CPU process. Within these frameworks, each GPU kernel collects all the communication generated during the kernel and issues it in bulk at the end of the kernel. This approach is typically used with large kernels, which naturally results in large bulk communications. It forces application/framework implementers into an unappetizing tradeoff: high GPU utilization but high-latency communication, or low GPU utilization with lower-latency communication.

An alternative is to treat each GPU thread as the analog of a CPU process, and allow each GPU thread to independently issue communication requests. However, an individual GPU thread is rather weak, and coordinating the hundreds of thousands of concurrently running threads required to fully utilize a GPU is a significant challenge.

What programmers want is more flexibility: the ability of the programmer to specify a group of threads with a size that is the best fit for the application. The GPU already has abstractions (warp, CTA, cooperative thread group) that help organize these groups. The abstraction we choose for Atos is a "worker": a set of GPU resources, including a configurable number of CUDA threads, shared memory, coupled with the number of tasks that this worker will target. One of the significant advantages of specifying groups of threads as a worker is that collective loads/stores among the threads of a worker can be issued as coalesced accesses that are necessary to make the most of the GPU memory system. Listings 1 and 2 show an intuitive example, simplified for clarity, of multi-GPU BFS programmed in the traditional way and with a PGAS style.

Listing 1 Traditional approach to a multi-GPU BFS.

```
BFS(graph G) {
  frontier F = {source_node};
  Vertex recv_buf[], send_buf[];
  while(!F.empty()) {
   GPU::one_step_traversal<<stream>>>(F, G, send_buf);
   // Launch CUDA kernel
   cudaStreamSynchronize(stream);
   MPI_Irecv(recv_buf, request);
   MPI_Isend(send_buf);
   MPI_Wait(request);
   GPU::merge_recv_buf_update_depth<<<stream>>>(F,G,recv_buf);
  }
}
__global__ void
one_step_traversal(frontier F, graph G, Vertex *send_buf)
{
    for (thread in all_threads) {
      Vertex node = F.exclusive_pop();
      for (j in node.neighbor()) {
        if (j.remote())
          send_buf.append(j);
        else if (F.depth_update(j, cur_depth) == SUCCEED)
          F.push(j);
      }
    }
}
```

Listing 2 PGAS approach to a multi-GPU BFS.

```
BFS(graph G) {
  GPU::BFS_traverse<<stream>>>(F, G);  // Launch CUDA kernel
  cudaStreamSynchronize(stream);
}
__global__ void BFS_traverse(frontier F, graph G) {
  frontier F = {source_node};
  while (!F.empty()) {
    for (thread in all_threads) {
       Vertex node = F.exclusive_pop();
       for (j in node.neighbor()) {
           if (j.remote())
             if (F.depth_update_remote(j, cur_depth)==SUCCESS)
               // one-sided remote update
               F.push_remote(j);  // one-sided remote update
           else if(F.depth_update_local(j, cur_depth)==SUCCESS)
             F.push_local(j);
       }
     }
  }
}
```

III. Atos's Design Decisions

A key feature of PGAS programming models is the decoupling of *communication* and *synchronization*, which are inherently intertwined in the BSP model [15]. Unlike BSP, which requires global synchronization in order to perform communication operations like all-to-all data redistributions, one-sided memory operations allow PGAS programs to send data asynchronously, without synchronization. This can enable significant performance improvements for applications like graphs [16] and genomics [17] algorithms. The *many-to-many* communication pattern [17], [18] is a well-studied alternative to BSP's synchronous all-to-all that uses *asynchronous queue insertions* to distribute data without global coordination. In this work, we take advantage of asynchronous communication to enable better performance on irregular graph problems.

Our GPU dynamic scheduling framework, Atos, implements a PGAS programming model. Inspired by thread-parallel CPU programming systems, Atos is programmed in a task-parallel way, maintaining a distributed queue of tasks. GPU workers fetch a task from the queue, then process it. Any newly generated tasks are added to the local distributed queue if they belong to a local process, and otherwise added to the remote distributed queue of the owner process. The program runs until either a stop condition is met or the entirety of the distributed queue is empty (Listing 3). In our system, we use the following terminology:

- worker: one or a group of GPU threads that work together as a single unit (optionally leveraging shared memory).
- task: one or more pieces of work that are scheduled as a single unit in our system. Tasks may consist of one or multiple data elements.

- application function $f()$: the code that processes each task. Each application function declares the worker size it requires to run.

Listing 3 Simplified task parallelism in Atos for illustration.

```
------------------------CUDA_KERNEL-------------------
for each worker:
  while not dist_queue.empty():
   task = dist_queue.concurrent_pop(task.size())
   new_tasks = f(task)
   if(new_tasks.local())
     dist_queue.concurrent_push(new_tasks)
   else
     dist_queue.concurrent_push(new_tasks, findPE(new_tasks))
------------------------------------------------------
```

Atos is a configurable framework with three key configuration decisions:

1) Kernel implementation strategy: Atos can use either discrete kernels or persistent kernels [19]. In the latter case, only one kernel is launched, which remains resident on the GPU until the program finishes. This strategy, while more complex, is better suited for applications that are otherwise dominated by kernel launch overhead. Listing 3 is written using a persistent kernel. We could alternatively interchange its outer and inner loops, making one discrete kernel call per iteration.
2) Queue architecture: a standard distributed queue vs. a distributed priority queue. In the standard queue, tasks are processed in FIFO order, while a more complex priority queue can prioritize tasks marked with higher priority. As we will show later, this capability is important in an asynchronous setting to reduce the cost of speculation.
3) Worker size: Atos provides thread-, warp- and CTA-sized workers, to support tasks of different sizes and different synchronization requirements.

Listing 4 shows Atos distributed queue APIs. `launch*` API functions are used to launch workers that repeatedly pop tasks from the local and remote receive queues; each worker then applies function `f1` to the popped task. When the worker fails to pop, it runs function `f2` (default noop) instead. In the `launchCTA` API, the choice of `numThread` determines the number of threads used for each worker. `launchThread` and `launchWarp` use worker sizes with 1 and 32 threads respectively and can each be implemented more efficiently using warp intrinsic instructions. Under the persistent-kernel mode, `numThread` $\times$ `numBlock`, by default, is set to the maximum number of threads that can concurrently reside on the GPU based on the application's register and shared memory usage, but can be overridden by users.

To illustrate the use of the framework API, we use the example of BFS with warp worker granularity. In Listing 5, we create a `BFS` class and define and allocate memory for its relevant variables. The code defines a SIMD function `BFSWarp`, in which all threads in a warp participate, collectively iterating over all neighbors of `node` and updating their `depth` values. If the `depth` of a `neighbor` is improved, `neighbor` is pushed into the local queue if it is a local vertex, otherwise into the remote

Listing 4 Atos framework APIs.

```
template<typename RECV_T, typename LOCAL_T, typename COUNTER_T>
class DistributedQueues {
public:
  __host__ void init (int my_pe, int n_pes, COUNTER_T local_cap,
  COUNTER_T recv_cap, int num_queues, int iteration);

  template<typename F1, typename F2, typename... Args>
  __host__ void launchThread (bool ifPersist, int numBlock,
  int numThread, int shareMem, F1 f1, F2 f2, Args... arg);

  template<typename F1, typename F2, typename... Args>
  __host__ void launchWarp (bool ifPersist, int numBlock,
  int numThread, int shareMem, F1 f1, F2 f2, Args... arg);

  template<int FETCH_SIZE,
  typename F1, typename F2, typename... Args>
  __host__ void launchCTA (bool ifPersist, int numBlock,
  int numThread, int shareMem, F1 f1, F2 f2, Args... arg);
};

template<typename RECV_T, typename LOCAL_T,
    typename THESHOLD_T, typename COUNTER_T>
class DistributedPriorityQueues {
public:
  __host__ void init (int my_pe, int n_pes, COUNTER_T local_cap,
  COUNTER_T recv_cap, THRESHOLD_T threshold,
  THRESHOLD_T threshold_delta, int num_queues, int iteration);

#Same thread-,warp- and CTA-launch APIs as in DistributedQueues
};
```

Listing 5 Atos BFS (worker size: warp).

```
struct BFS {
  int my_pe;
  int n_pes;
  int total_nodes;
  int total_edges;
  CSR *csr;
  int *depth;
  DistributedQueues<int, int, int> worklists;

  BFS(Csr my_csr, int local_cap, int recv_cap, int my_pe,
  int n_pes, int num_qs) {
    csr = &my_csr;
    worklists.init(my_pe, n_pes, local_cap, recv_cap, num_qs);
    nvshmem_malloc(&depth, sizeof(int) * total_nodes);
  }

  void BFSStartWarp(int numBlock, int numThread) {
    worklists.launchWarp(1, numBlock, numThread, 0, BFSWarp(), *this);
  }
};

class BFSWarp {
 public:
  __device__ void operator()(int node, BFS bfs) {
    int depth = bfs.depth[node];
    int node_offset = bfs.csr.neighborlist_start(node);
    int neighborlen = bfs.csr.neighbor_list_length(node);
    for (int item=LANE; item<neighborlen; item = item + 32){
      int neighbor = bfs.csr.get_neighbor(node_offset + item);
      int old = atomicMin(bfs.depth+neighbor, depth+1);
      if (old > depth + 1) {
        if(bfs.iflocal(neighbor))
          bfs.worklists.push_warp(neighbor);
        else {
          int pe = bfs.csr.findPE(neighbor);
          if(atomicMin(bfs.depth+neighbor, depth+1, pe) > depth+1)
          // remote RDMA direct atomic operation
            bfs.worklists.push_warp(neighbor, pe);
            // RDMA push to remote receive queues
        }
      }
    }
    __syncwarp();
  }
};
```

receive queue of its owner GPU. Lastly, we pass `BFSWarp()` and its arguments onto `launchWarp` and invoke it.

A. Implementing One-Sided GPU Communication

The three most important design decisions we make in implementing Atos focus on enabling lightweight, efficient one-sided GPU communication:

1) GPU-Based Control Path: Data transfers require two paths: the path by which data actually moves, and the communication control path that includes preparing messages, triggering data movement, and signaling the remote process. Most frameworks enable a GPU-to-GPU data movement path, but implement the control path on the CPU. Atos implements both on the GPU, which both reduces control path latency and lowers overhead. Together these improvements make fine-grained communication feasible and potentially desirable. Briefly, on NVLink systems, we leverage CUDA's unified memory to allow reads from and writes to remote GPU memory on the same node. On IB systems, we turn to NVIDIA's NVSHMEM and its put, get, and atomic operations. Both mechanisms keep all control on GPUs and do not require CPU intervention.

2) Ensuring Data Consistency with an Asynchronous Distributed Queue: We implement a lock-free queue that is able to replace the heavyweight synchronization between CPUs that would traditionally occur at the end of CUDA kernels before initiating communication. Our design is able to avoid two resulting overheads: the cost of synchronization and the lack of overlap of communication and computation.

Maintaining data consistency in a queue without kernel synchronization where multiple asynchronous Atos workers are both reading and writing is a challenge. Because our queue is implemented as a single block of memory and respects FIFO ordering, we use a counter-based mechanism (Listing 6) to carefully manage the queue's start and end states to guarantee consistency, preventing invalid data from being popped before it is either processed locally or sent to remote GPUs. In our design, all data before `end` in the queue is valid and ready to pop. `end_max` and `end_count` are used to update `end`; Listing 6 details the mechanism. In contrast, other concurrent queue implementations such as Troendle et al. [20] and the broker queue [21] address this problem by wrapping each queue item in a tuple with a flag. Pushing to their queues requires 3 steps: (1) write the item to the reserved spot in the queue; (2) call `__threadfence()`; (3) set the corresponding flag to ready. Popping from their queues requires reading a valid flag beforehand. Our choice of additional global counters instead of per-item flags results in two benefits: (1) the flag solution consumes unnecessary memory (typically an entire word to ensure alignment), and (2) querying new work items can be done via a single `end` counter broadcast, which consumes less memory bandwidth than polling a different flag for each item. In addition, our choice of an `atomicAdd` synchronization primitive instead of `atomicCAS` [22], [23] enables higher performance under high-contention concurrent popping, as CAS failure probability increases significantly with increasing contention.

Our implementation differs from recent queue designs [20], [21] that also use `atomicAdd` in two ways. (1) We use the GPU thread hierarchy to reduce contention. More specifically, each (warp- or CTA-sized) worker computes the total number of push/pop requests for the entire worker first, then only the worker's leading thread atomically increases the counter. (2) We pad the memory to ensure `end`, `start`, `end_alloc`, `end_max`, and `end_count` are stored in different cache lines because those counters are each updated through atomics and storing them in the same cache line would otherwise serialize the updates.

Listing 6 Update counter end mechanism.

```
void Queue::push_warp(T item)
    unsigned mask = __activemask();
    uint32_t total = __popc(mask);
    int rank = __popc(mask & lanemask_lt());
    int leader = __ffs(mask)-1;
    uint32_t reserv_index = -1;
    if (rank==0) reserv_index = atomicAdd(&end_alloc, total);
    reserv_index = __shfl_sync(mask, reserv_index, leader);
    queue[reserv_index+rank] = item;
    if (rank==0) {
        atomicMax(&end_max, reserv_index+total);
         __threadfence();
    }
    __syncwarp(mask);
    if (rank == 0) {
        if (atomicAdd(&end_count, total) + total == end_max)
           atomicMax(&end, end_max);
    }
```

These two differences enable our queue implementation to perform better than the alternatives. We characterize queue performance with three experiments, each with high contention: (1) n concurrent threads each push to the queue 10 times; (2) n concurrent threads each pop from the queue 10 times; and (3) n concurrent threads each push and then pop from the queue 10 times without synchronization between push and pop. We compare two implementations of our queue—warp-API and CTA-API—with the open-source broker queue [21] and with our own implementation of an `atomicCAS`-based queue[1]. Figure 1 shows the runtime of concurrent push, concurrent pop and, concurrent pop-and-push as the number of threads, and hence contention, increases. In all benchmarks, both the warp and CTA implementations of our queue are faster than the broker queue and CAS-based queue and show better scalability.

3) Integrating a communication aggregator: Different applications result in different native message sizes. In order to minimize programmer burden, the ideal programming model would allow applications to use their native message sizes. This presents a challenge, however, when an application prefers small, irregular message sizes, as is the case for the applications considered in this work. Such communications may be, from a performance perspective, poorly suited for a

[1]Troendle et al.'s queue [20] would be an interesting comparison, but only supports AMD GPUs and lacks warp intrinsics. Our `atomicCAS` queue implementation leverages warp intrinsics to avoid inter-warp contention.

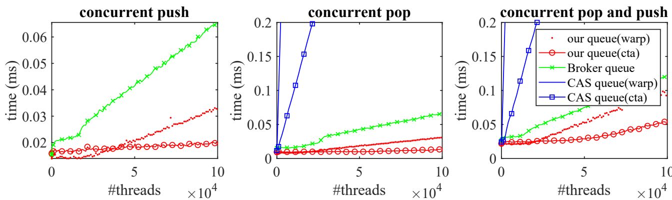

Fig. 1. Runtime performance of our queue with warp- and CTA-sized workers against the broker queue [21] and our implementation of a CAS-based queue.

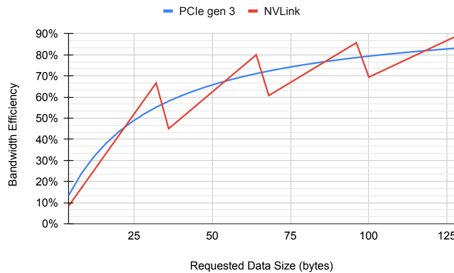

Fig. 2. Bandwidth efficiency (fraction of message size occupied by payload) vs. requested bytes on PCIe Gen 3 and NVLink. The minimum payload size on NVLink is a 32-byte sector. A NVLink package can contain up to 4 sectors.

particular communication technology (primarily, in our case, InfiniBand). We next note design issues for each of our communication technologies:

a) Choosing communication size on NVLink: For NVLink-connected single-node GPUs, remote memory accesses essentially look like GPU loads and stores. With them, remote GPU memory access latency can be hidden by other instructions in a kernel, taking advantage of dynamic instruction scheduling. Any adjacent remote memory accesses within a warp (a group of 32 neighboring threads) will first be aggregated before issue ("coalescing"). Unlike other communication technologies such as PCIe or InfiniBand with a wide range of payload sizes, NVLink packages are more restricted in payload size (only up to 128 bytes). Figure 2 shows that even modest payload sizes achieve relatively good bandwidth efficiency over NVLink. Coupling this bandwidth efficiency with a relatively low latency and high throughput (even a 32 byte payload has more than 50% efficiency), NVLink is a good solution for the kind of small random accesses we expect to target.

b) Choosing communication size on InfiniBand: Infiniband presents a more significant challenge. IB messages pass through the NIC and are not able to take advantage of instruction-level parallelism (ILP) in the way NVLink messages do. IB bandwidth is lower, and messages have longer latencies. IB supports a smaller number of operations than NVLink (e.g., lacking atomicMin). IB does have one advantage: because IB memory requests are offloaded to the NIC, IB requests require only one or a few threads to initiate data transfer, in contrast to the many threads involved in an NVLink request. In summary, more severe latency and bandwidth constraints mean that if we follow the same strategy as NVLink communications with small messages, we are likely to make poor use of IB and gate our overall performance.

c) Communication Aggregator: We implement a communication aggregator that runs transparently alongside application code to aggregate individual requests into larger messages. By achieving larger message sizes, we are able to improve bandwidth utilization. Figure 3 shows the workflow for our communication aggregator, which bundles messages together locally until either a maximum message size or a maximum wait time is reached. Once one of these user-configurable parameters is exceeded, the runtime will send the bundled messages over the wire. Our aggregator allows us to achieve higher IB bandwidth utilization, through larger messages sizes, at the cost of higher message latency.

Our aggregator is transparent to the programmer[2], who implements their application by writing tasks. Inside tasks, users can use a combination of local memory operations, new task launches, and PGAS-style one-sided memory operations, to implement their applications. Our aggregator is critical for performance, as it allows us to decouple the code that generates new messages from the code that actually sends them out over the network. Users can then impelment their tasks using the task granularity most natural for their application, and these can be sent out over the network in batches.

An ideal batch size will generate messages that are large enough to saturate the network bandwidth while maintaining a relatively low latency. In order to determine the optimal batch size, we perform two experiments. (1) Measure the *latency* cost at different message sizes (Figure 4, left). (2) Measure the *bandwidth* achieved at different message sizes (Figure 4, right). In these experiments, each send is performed as a blocking send operation followed by a system memory fence (necessary to ensure completion of the send) and a remote counter update. When initiating communication on the GPU in our IB system, the optimal message size balances between minimizing latency and maximizing bandwidth. Figure 4 shows this tradeoff on our system; we choose a 1 MiB message size, with near-minimal latency and high bandwidth. This message size is consistent with previous studies of optimal message size for GPU-initiated communication on IB networks [24]. While larger message sizes always achieve higher bandwidth *during the time the data is being sent*, smaller messages are preferred when there are not enough tasks to fill the buffer. In such a case, without another mechanism for triggering a message, the queue batching might wait forever. If the application takes long to fill the buffer, communication latency would increase, which hampers increased throughput because longer latency hinders the generation of new tasks. We expect to see this problem particularly if the application is limited by available parallelism. To address this problem, we enable a second mechanism for triggering a message send: a maximum wait time. We implement this using a *WAIT_TIME* counter, which counts each query to the queue to see if it is full. After *WAIT_TIME* visits, the data is sent out, whether it

[2]This is not entirely true; our applications are currently implemented using separate push-to-network-via-aggregator and push-directly-to-network calls, but we believe integrating these would be straightforward.

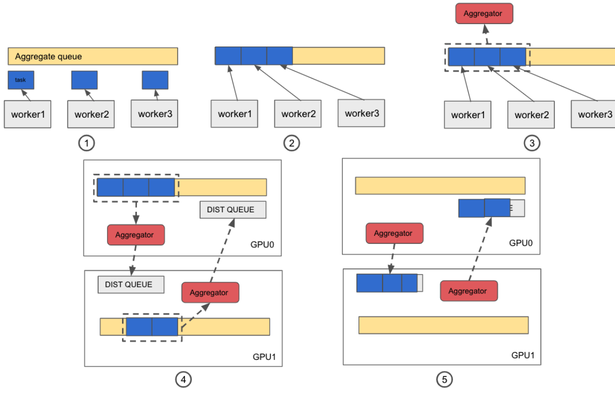

Fig. 3. Step 1: instead of directly sending to remote GPUs, Atos workers push the messages to an aggregate queue for accumulation. Step 2: Workers return immediately when they finish writing the messages to the aggregate queue. Step 3: Our aggregator, running persistently and concurrently alongside Atos workers, monitors message accumulation count. Steps 4 and 5: If accumulated messages reach a *BATCH_SIZE*, or if enough time passes, the aggregator writes the accumulated messages to the remote GPU distributed queue.

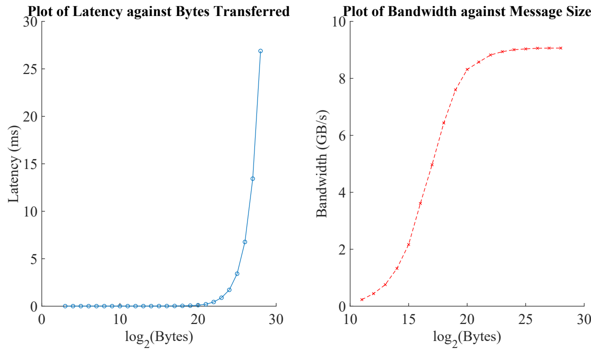

Fig. 4. On our IB system, a batch size of 2^{20} B allows us to achieve both near-peak bandwidth and relatively low latency.

meets the maximum message size or not. Programmers can thus utilize a "eager" mode that minimizes latency by setting the wait time to be very low.

B. Related Work

Substantial prior work on GPUs has described alternate design approaches to the traditional BSP-focused programming model. In general, these prior frameworks [10], [12]–[14] adopt a master-slave model, where the CPU orchestrates the executions, streams the data to the GPUs alongside running GPU computation kernels, so that the communication can be overlapped with the computation. This is the same benefit we advocate in this paper. However, because of the design decisions discussed in Section III-A, we are able to send many smaller messages with much less overhead than previous frameworks. The frequency and granularity of communication matters as it increases the depth of message pipelines, allows more communication and computation overlap. Particularly the difference between Atos and asynchronous graph frameworks Groute, Galois will be discussed in more detail in section IV.

IV. Results and Analysis

In order to evaluate the performance of Atos, which enables PGAS-style communication for dynamic, irregular applications, we select two graph algorithms—BFS and PageRank—as representatives of irregular applications.

BFS: Our BFS "push" asynchronous implementation begins by adding a single vertex to the queue. Workers continuously pop vertices from the queue. Once a worker succeeds in popping work, workers propagate the popped vertex's depth to its neighbors using an `atomicMin` operation and add the neighbor to the queue if that neighbor's depth has been modified. BFS finishes when the queue becomes empty. We discuss this BFS formulation, with pseudocode, in more detail in our Atos single-GPU work [16].

PageRank: PageRank computes the importance (rank) of nodes in a graph with the assumption that more links to a node from other highly-ranked nodes indicate greater importance. A PageRank "push" implementation begins by assigning each node to an initial PageRank and residue value and pushing all vertices into the work queue. Workers continuously pop from the queue. If the pop succeeds, the worker adds the popped vertex's residue to its rank and propagates some of the residue to all its neighbors. If the pop fails, the worker loops over its assigned vertices and pushes the vertices that have residue higher than the convergence threshold and are not in the queue. The PageRank finishes when all vertices have converged and the queue becomes empty. We discuss this PR formulation, with pseudocode, in more detail in our Atos single-GPU work [16].

We designed Atos to be able to achieve high performance on systems with different interconnect characteristics. Thus we test on a single-node system with 4 GPUs connected by NVLink and on a multi-node system with 8 GPUs connected by InfiniBand (IB).

1) **NVLink System ("Daisy")**: Daisy is an NVIDIA DGX Station running Linux with 4 V100 GPUs each with 32 GB memory, two 2.20 GHz Intel hyper-threaded E5-2698 v4 Xeon CPUs, and 128 GB of main memory. The 4 V100 GPUs on Daisy are all-to-all connected via NVLink; each GPU has one dual-link (50 GB/s) connection to one GPU peer and two single-link (25 GB/s) connections to the others. We use NVIDIA's nvcc version 11.1.168 and gcc 9.3.0, both with the -O3 flag. All tests were run 10 times with the average runtime used for results.
2) **IB System ("Summit")**: Each node of Oak Ridge Leadership Computing Facility's Summit supercomputer [25] is equipped with two IBM POWER9 processors and six NVIDIA Tesla V100 accelerators, each with 16 GB of memory. Each GPU connects to one of the POWER9 processors, as well as the other two GPUs in its NUMA domain, using a bidirectional 50 GB/s NVLink link. In order to isolate the effect of inter-node communication over InfiniBand, we run our experiments using a single GPU per Summit node. This ensures that all commu-

TABLE I
SUMMARY OF THE DATASETS USED IN OUR EXPERIMENTS.

Dataset	Vertices	Edges	Diam.	Max. indeg.	Max. outdeg.	Avg. degree	type
soc-LiveJournal1	4.8M	68M	20	13,905	20,292	14	scale-free
hollywood_2009	1.1M	11M	11	11,467	11,467	105	scale-free
indochina_2004	7.4M	191M	26	256,425	6,984	8	scale-free
twitter50	51M	1.9B	12	3.5M	0.77M	38	scale-free
road_usa	23.9M	57M	6,809	9	9	2	mesh-like
osm_eur	174M	348M	21,158	15	15	2	mesh-like

nication between GPUs is performed using InfiniBand. Each Summit node is connected to the network using dual-rail EDR InfiniBand, with each rail providing 12.5 GB/s of unidirectional injection bandwidth.

We test BFS and PageRank on two graph types: scale-free datasets (primarily generated from social networks) and mesh-like datasets (primarily road networks), summarized in Table I. In all experiments, we use 512-thread CTA workers, which achieve the best performance for both BFS and PageRank. More details on the choice of the worker size can be found in our Atos single-GPU work [16].

A. Evaluation on NVLink

We evaluated two different configurations of Atos (discussed in Section III) on the 4-GPU NVLink system: *Atos-standard-persistent*, which uses a distributed standard queue with persistent kernels, and *Atos-priority-discrete*, which uses a distributed priority queue with discrete kernels. The most direct comparison possible is against single-node, multi-GPU frameworks, and we choose two leading frameworks for which source code is available: Gunrock [11] and Groute [10]. All experiments use the same graph partitionings[3]. Out of the 6 tested datasets, Groute cannot run twitter50 due to an out-of-memory error. We summarize performance results in Tables II and IV.

a) Summary of NVLink Results: BFS Atos achieves speedup over Groute over all tested graphs except indochina-2004 with 1 and 2 GPUs; BFS Atos achieves speedup over Gunrock over all tested graphs except twitter50. PageRank Atos achieves speedups over both Gunrock and Groute over all tested graphs. Atos's speedup is greatest when compared to Gunrock for BFS on mesh-like datasets and when compared to Groute for PageRank on all tested datasets. In addition to superior runtime performance, Atos achieves better overall strong scaling.

1) BFS on NVLink: Of the two Atos configurations: on mesh-like datasets, *Atos-standard-persistent* is faster than *Atos-priority-discrete*; conversely, on scale-free datasets, *Atos-priority-discrete* is faster. The primary reason for this difference is that BFS on mesh-like datasets suffers from a lack of parallelism, thus underutilizing GPUs; conversely BFS on scale-free datasets has more parallelism and is more bounded by bandwidth [16], [26].

The asynchronous BFS algorithm used in Atos and Groute exposes additional parallelism at the cost of redundant work.

[3]Groute requires Metis, so for all tests that Groute can run, we use Metis partitionings; twitter50 uses a random partitioning.

TABLE II
BFS RUNTIMES IN MS (SPEEDUP VS. GUNROCK IN PARENTHESES) ON DAISY (NVLINK). PERFORMANCE LEADERS ARE BOLDED. GRAPH TYPES ARE S (SCALE-FREE) AND M (MESH-LIKE).

Application: BFS on Gunrock				
Dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs
soc-LiveJournal1^s	13.4 (x1)	10.0 (x1)	8.15 (x1)	8.03 (x1)
hollywood_2009^s	6.28 (x1)	5.38 (x1)	5.62 (x1)	5.39 (x1)
indochina_2004^s	11.0 (x1)	12.8 (x1)	13.6 (x1)	14.9 (x1)
twitter50^s	**906 (x1)**	**477 (x1)**	**330 (x1)**	**258 (x1)**
road_usa^m	604 (x1)	917 (x1)	963 (x1)	1009 (x1)
osm-eur^m	2094 (x1)	3163 (x1)	3282 (x1)	3442 (x1)
Application: BFS on Groute				
Dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs
soc-LiveJournal1^s	19.0 (x0.71)	10.8 (x0.93)	10.2 (x0.80)	12.6 (x0.64)
hollywood_2009^s	7.17 (x0.88)	5.81 (x0.93)	5.82 (x0.97)	8.63 (x0.62)
indochina_2004^s	**7.55 (x1.47)**	**7.43 (x1.73)**	23.2 (x0.59)	29.7 (x0.50)
road_usa^m	144 (x4.42)	145 (x6.32)	152 (x6.32)	163 (x6.17)
osm-eur^m	570 (x3.66)	507 (x6.22)	502 (x6.53)	512 (x6.71)
Application: BFS on Atos (queue+persistent kernel)				
Dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs
soc-LiveJournal1^s	12.4 (x1.08)	9.00 (x1.12)	6.87 (x1.19)	6.33 (x1.27)
hollywood_2009^s	6.27 (x1.00)	7.90 (x0.68)	6.86 (x0.82)	6.77 (x0.80)
indochina_2004^s	8.03 (x1.38)	9.44 (x1.36)	8.43 (x1.62)	7.38 (x2.03)
twitter50^s	1412 (x0.64)	841 (x0.57)	587 (x0.56)	452 (x0.57)
road_usa^m	**46.5 (x13.7)**	**57.5 (x15.9)**	**63.6 (x15.1)**	**62.0 (x16.2)**
osm-eur^m	**247 (x8.47)**	**218 (x14.5)**	**236 (x13.8)**	**227 (x15.1)**
Application: BFS on Atos (priority queue+discrete kernel)				
Dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs
soc-LiveJournal1^s	**11.3 (x1.18)**	**6.45 (x1.56)**	**5.01 (x1.63)**	**4.01 (x2.00)**
hollywood_2009^s	**5.77 (x1.09)**	**5.14 (x1.05)**	**4.69 (x1.20)**	**3.84 (x1.40)**
indochina_2004^s	9.68 (x1.15)	9.21 (x1.39)	**7.23 (x1.89)**	**6.48 (x2.31)**
twitter50^s	1052 (x0.86)	506 (x0.94)	348 (x0.95)	270 (x0.96)
road_usa^m	189 (x3.38)	181 (x5.05)	200 (x4.81)	207 (x4.86)
osm-eur^m	518 (x4.04)	617 (x5.12)	623 (x5.26)	709 (x4.85)

TABLE III
NORMALIZED WORKLOAD WITHOUT → WITH PRIORITY QUEUE

Dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs
soc-LJ1	1.063 → 1.003	1.26 → 1.06	1.34 → 1.10	1.42 → 1.141
hollywd	1.168 → 1.197	1.36 → 1.11	1.42 → 1.21	1.57 → 1.248
indoch	1.004 → 1.00	1.03 → 1.03	1.03 → 1.04	1.05 → 1.047
twtr50	1.237 → 1.008	1.29 → 1.16	1.31 → 1.26	1.34 → 1.305

Redundant work occurs because out-of-order iterations may require visiting vertices multiple times in order to find the shortest path. In practice, we observe that BFS on mesh-like datasets benefits from this tradeoff and are instead sensitive to kernel launch overhead which can be reduced directly by using persistent kernels.

BFS on scale-free datasets is not limited by parallelism (e.g., twitter50 shows excellent strong scalability), but instead by bandwidth. Here, the tradeoff of redundant work for more parallelism is unfavorable and worsens the bandwidth bottleneck. We thus directly mitigate the cost of redundant work with a priority queue in order to give vertices with lower depth values higher processing priority. We quantify this by counting the total number of vertices visited and normalizing against an ideal traversal that only visits each vertex once (without p.q. → with p.q.) in Table III.

a) Atos vs. Groute/Gunrock, mesh-like datasets: Groute and Atos use the same algorithm (asynchronous BFS) and kernel strategy (persistent kernel), so these factors do not

contribute to the performance difference. BFS on mesh-like datasets suffers from lack of parallelism, and interconnection latency further hinders saturation of the workers by slowing the rate that individual processes can push or pop to the work queue. Atos's performance advantage comes from its lower communication latency. Why? Atos sends communication immediately when communication data is available. This stands in constrast to Groute's control path, which passes through the CPU, while Atos's entire control path is on the GPUs.

Gunrock's control path also passes through the CPU, but Gunrock has additional complications. It implements a BSP version of BFS, incurring further latency due to waiting for kernel synchronization. In BSP BFS, the parallelism in any given iteration is limited to only the vertices in the current step; in contrast, asynchronous BFS uses speculation to expose more parallel work [16]. Finally, Atos's persistent-kernel formulation reduces the large kernel launch overhead seen in Gunrock on mesh-like datasets [16], [26]. These additional factors lead to Atos's much larger speedup over Gunrock (13.8x) compared to Groute (2.44x).

b) Atos vs. Groute/Gunrock, scale-free datasets: BFS on scale-free datasets is more limited by interconnection bandwidth. Priority queues mitigate the amount of redundant work, directly reducing bandwidth cost. Atos's performance advantage over Groute (1.71x) and Gunrock (1.29x) is due to Atos's aggressive overlap of communication and computation during kernel execution. This allows consistent communication throughout the entire program and best utilizes the most bottlenecked resource, interconnect bandwidth.

c) Strong Scaling: Figure 5 (left) shows strong scaling results for the four BFS implementations. Not surprisingly, all frameworks scale better on bandwidth-limited scale-free graphs than parallelism-limited mesh-like graphs. Regardless of the type of dataset, the Atos configuration with priority queue and discrete kernel achieves similar or better scalability than both Gunrock and Groute on all datasets.

2) PageRank: Excluding indochina_2004[4], Groute generally performs worse than Gunrock (0.85x). Both Atos implementations outperform Gunrock (2.59x for *Atos-standard-discrete* and 2.37x for *Atos-standard-persistent*). Gunrock uses the BSP model for both computation and communication. The two Atos implementations and Groute all use asynchronous PageRank. We conclude that the overall performance difference is less due to algorithmic differences (asynchronous vs. BSP) and more due to implementation differences in the frameworks.

Compared to BFS, PageRank generally has more parallelism as well as communication volume. For instance, on the twitter50 dataset, on {2, 3, 4}-GPU configurations, Atos's PageRank has {10, 13, 14}x the workload of Atos's BFS. We see three reasons for Atos's performance advantage: (1) Atos communications are spread out, smoothing the spikes in network communication that typically occur when communication is isolated in a single phase; (2) small messages are better able to overlap with computation, hiding latency; and (3) a GPU control path reduces the latency comparing to routing through CPUs (Groute).

[4]On PageRank, Groute is very slow on indochina_2004 (more than 200 times slower than Gunrock).

TABLE IV
PAGERANK RUNTIME IN MS (SPEEDUP VS. GUNROCK IN PARENTHESES) ON DAISY (NVLINK). PERFORMANCE LEADERS ARE BOLDED. GRAPH TYPES ARE S (SCALE-FREE) AND M (MESH-LIKE).

Application: PageRank on Gunrock				
Dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs
soc-LiveJournal1^s	262 (x1)	188 (x1)	89.8 (x1)	75.3 (x1)
hollywood_2009^s	87.3 (x1)	51.7 (x1)	44.8 (x1)	33.8 (x1)
indochina_2004^s	159 (x1)	120 (x1)	105 (x1)	100 (x1)
twitter50^s	25483 (x1)	15075 (x1)	8996 (x1)	6998 (x1)
road_usa^m	220 (x1)	189 (x1)	143 (x1)	122 (x1)
osm-eur^m	2784 (x1)	2253 (x1)	1650 (x1)	1373 (x1)
Application: PageRank on Groute				
Dataset	1 GPU	2 GPUs	3 GPUs	4GPUs
soc-LiveJournal1^s	259 (x1.01)	165 (x1.14)	132 (x0.68)	132 (x0.57)
hollywood_2009^s	115 (x0.76)	109 (x0.47)	102 (x0.44)	105 (x0.32)
indochina_2004^s	31933 (x0.00)	31845 (x0.00)	31396 (x0.00)	31360 (x0.00)
road_usa^m	479 (x0.46)	232 (x0.81)	150 (x0.96)	114 (x1.08)
osm-eur^m	2414 (x1.15)	1224 (x1.84)	829 (x1.99)	661 (x2.08)
Application: PageRank on Atos (discrete kernel)				
Dataset	1 GPU	2 GPUs	3GPUs	4GPUs
soc-LiveJournal1^s	**116 (x2.26)**	58.8 (x3.20)	**35.6 (x2.52)**	**26.3 (x2.86)**
hollywood_2009^s	**75.1 (x1.16)**	**27.9 (x1.85)**	**21.75 (x2.06)**	**18.9 (x1.79)**
indochina_2004^s	**50.8 (x3.14)**	**30.8 (x3.90)**	**24.1 (x4.39)**	**19.8 (x5.07)**
twitter50^s	11291 (x2.26)	6332 (x2.38)	4521 (x1.99)	3582 (x1.95)
road_usa^m	**111 (x1.98)**	76.0 (x2.49)	51.2 (x2.80)	38.9 (x3.16)
osm-eur^m	991 (x2.81)	785 (x2.87)	**525 (x3.14)**	**408 (x3.39)**
Application: PageRank on Atos (persistent kernel)				
Dataset	1 GPU	2GPUs	3GPUs	4GPUs
soc-LiveJournal1^s	117 (x2.23)	**58.4 (x3.23)**	40.0 (x2.24)	32.2 (x2.33)
hollywood_2009^s	90.8 (x0.96)	33.3 (x1.55)	31.4 (x1.43)	26.2 (x1.29)
indochina_2004^s	53.4 (x2.98)	37.0 (x3.24)	35.0 (x3.02)	30.1 (x3.34)
twitter50^s	**11037 (x2.30)**	**5802 (x2.59)**	**4016 (x2.24)**	**3077 (x2.27)**
road_usa^m	128 (x1.72)	**69.5 (x2.72)**	**47.3 (x3.03)**	**36.2 (x3.39)**
osm-eur^m	**923 (x3.01)**	**729 (x3.09)**	590 (x2.80)	508 (x2.70)

a) Strong Scaling Tests: Figure 5 (right) shows the strong scaling for the four implementations for PageRank on different datasets. We highlight two interesting results. (1) It is possible to have strong scaling beyond the perfect scaling line because asynchronous PageRank may lead to less total workload than BSP PageRank. (2) Compared to BFS, PageRank generally sees better strong scalability. This is because PageRank generally has more opportuntiy for parallelism than BFS: each vertex is pushed multiple times in PageRank, whereas most vertices are only visited once in BFS.

On all datasets, both Atos implementations achieve better absolute runtime compared to Gunrock and Groute. On the soc-LiveJournal1 datasets, Atos achieves better strong scaling as well. On the twitter50, road_usa, and osm-eur datasets, Gunrock/Groute achieve similar/better strong scaling respectively, when compared to Atos; this is because Gunrock and Groute's single-GPU time is very slow.

3) Latency Hiding: To test if small-grained one-sided communication has better latency tolerance, we compare Gunrock and Atos on both BFS and PageRank on two different NVLink topologies (Figure 6). Summit's topology requires more than half of all GPU-to-GPU communications to pass between sockets and thus incurs a latency penalty. Figure 7 shows that for both BFS and PageRank, Gunrock's strong scaling drops

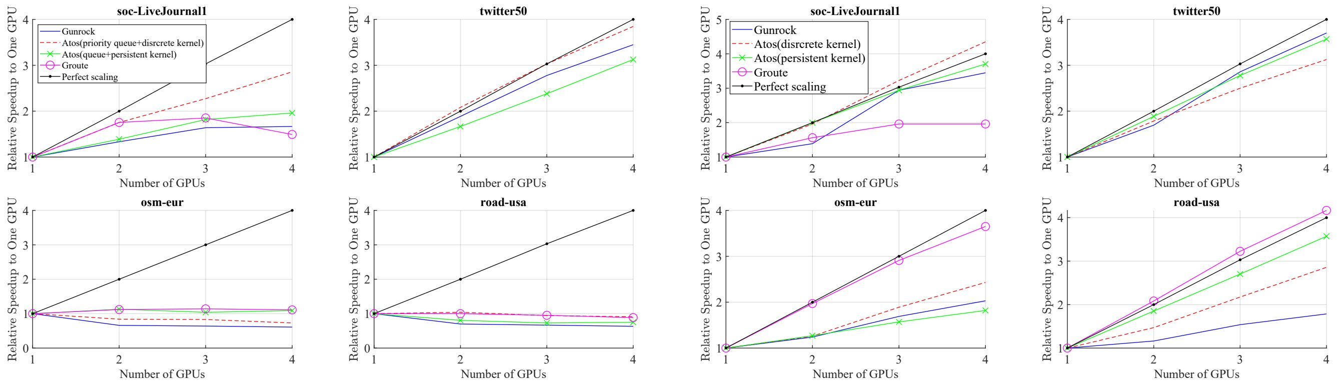

Fig. 5. Strong scaling test for BFS (left) and PageRank (right) on 4 datasets on an NVLink system. The plot shows relative speedup for each framework, as a function of the number of GPUs, comparing to its own single-GPU implementation (a self-to-self comparison). The black solid line in all plots shows perfect strong scaling. The top two datasets are scale-free and the bottom two are mesh-like.

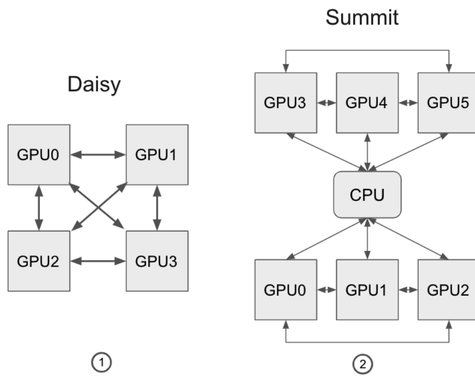

Fig. 6. The left topology (Daisy) is an all-to-all NVLink connection. The right topology (Summit) has three GPUs on each of two different sockets, with longer latency between GPUs than on the fully connected topology.

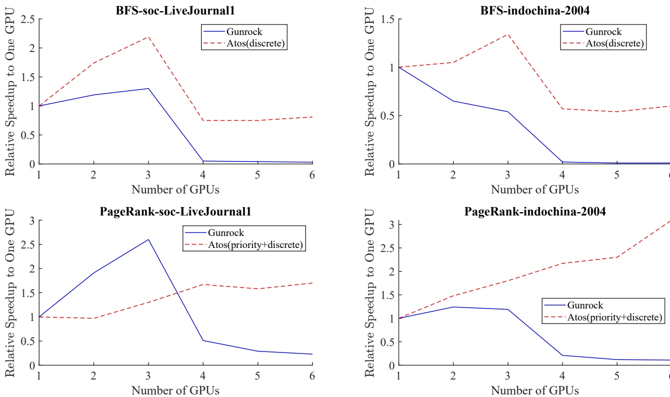

Fig. 7. Strong-scaling Gunrock and Atos on a single multi-GPU Summit node.

beyond 3 GPUs. For the more interconnection-latency-limited application BFS, Atos's scaling also drops beyond 3 GPUs, but less than Gunrock. For the interconnection-bandwidth-limited application PageRank, Atos continues to achieve speedups beyond 3 GPUs. We conclude that our implementation of small-grained one-sided communication has better latency tolerance than previous work.

B. Evaluation on InfiniBand

We run BFS and PageRank with Atos and Galois [12] on an InfiniBand (IB)-connected multi-GPU system (Summit), using one GPU per node. Neither Groute nor Gunrock run on multiple nodes, so we compare to another framework Galois. Galois uses the Gluon communication layer and is a non-blocking bulk-asynchronous graph library. We also considered Lux, built atop the Legion framework, but despite our best efforts and requests to its authors, we were unable to compile Lux. Galois's own experiments [12] compare favorably to Lux, however, so we believe we have chosen the most appropriate comparison. The principal difference between Atos and either Galois or Lux is the smaller granularity of communication enabled by Atos. All runtime results below use the best measured runtime among all available partition schemes.

a) Summary of InfiniBand Results: Atos shows runtime speedup and better scalability over Galois on all tested graphs on both BFS and PageRank: twitter50 demonstrates a modest speedup and all other datasets show large speedups (Table V).

Our NVLink and IB Atos implementations are identical except for the following consideration. It is easier to achieve scalable communication on the more capable NVLink interconnect than on IB. Bandwidth on our IB system is more of a constraint and small communications are particularly inefficient. Hence, in our IB implementation, we do not communicate by directly writing to/reading from remote GPU memories but instead route all communication messages through Atos's communication aggregator. The aggregator (Section III-A3) runs as a persistent kernel concurrently with the application code, checking the aggregated messages and sending them to remote GPUs when the messages reach a *BATCH_SIZE* or timeout threshold. We summarize performance results for Galois and Atos in Table V.

1) BFS: For BFS on all datasets, we set the aggregate message threshold (*BATCH_SIZE*) to 1 MB and *WAIT_TIME* to 4 to enable eager mode (minimal accumulation), because BFS is more bounded by interconnection latency.

For BFS, Atos generally performs better than Galois on both scale-free datasets and mesh-like datasets, even though Galois

Application: BFS on Galois

dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
soc-LiveJournal1^s	19.8 (x1)	19.1 (x1)	361 (x1)	382 (x1)	476 (x1)	470 (x1)	587 (x1)	636 (x1)
hollywood-2009^s	24.6 (x1)	204 (x1)	263 (x1)	403 (x1)	466 (x1)	499 (x1)	542 (x1)	545 (x1)
indochina-2004^s	49.0 (x1)	88.4 (x1)	667 (x1)	724 (x1)	858 (x1)	931 (x1)	953 (x1)	985 (x1)
twitter50^s	**465 (x1)**	**533 (x1)**	500 (x1)	591 (x1)	638 (x1)	699 (x1)	809 (x1)	702 (x1)
road_usa^m	4392 (1x)	24661 (1x)	36891 (1x)	37258 (1x)	143830 (1x)	53299 (1x)	173400 (1x)	65332 (1x)
osm-eur^m	86516 (1x)	76359 (1x)	105660 (1x)	135425 (1x)	148622 (1x)	165393 (1x)	176689 (1x)	180735 (1x)

Application: BFS on Atos

dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
soc-LiveJournal1^s	**11.3 (x1.74)**	**7.34 (x2.60)**	**5.69 (x63.6)**	**4.87 (x78.6)**	**4.29 (x110)**	**3.97 (x118)**	**3.69 (x159)**	**3.72 (x171)**
hollywood-2009^s	**5.77 (x4.26)**	**4.19 (x41.7)**	**4.22 (x62.4)**	**3.61 (x111)**	**3.11 (x150)**	**2.94 (x169)**	**3.31 (x163)**	**3.17 (x172)**
indochina-2004^s	**9.68 (x5.06)**	**9.35 (x9.45)**	**7.71 (x86.5)**	**6.77 (x107)**	**7.14 (x120)**	**6.97 (x133)**	**6.75 (x141)**	**7.12 (x138)**
twitter50^s	1052 (x0.44)	539 (x0.99)	**366 (x1.37)**	**338 (x1.75)**	**298 (x2.14)**	**286 (x2.44)**	**329 (x2.46)**	**286 (x2.46)**
road_usa^m	**46.5 (x94.4)**	**40.3 (x609)**	**49.0 (x752)**	**49.4 (x753)**	**57.1 (x2515)**	**64.2 (x829)**	**74.2 (x2336)**	**79.0 (x826)**
osm-eur^m	**247 (x349)**	**220 (x345)**	**226 (x466)**	**253 (x534)**	**278 (x534)**	**260 (x633)**	**268 (x657)**	**269 (x671)**

Application: PageRank on Galois

dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
soc-LiveJournal1^s	1066 (x1)	1059 (x1)	661 (x1)	662 (x1)	669 (x1)	672 (x1)	666 (x1)	634 (x1)
hollywood-2009^s	454 (x1)	702 (x1)	796 (x1)	808 (x1)	814 (x1)	810 (x1)	1042 (x1)	997 (x1)
indochina-2004^s	2950 (x1)	2614 (x1)	2926 (x1)	2657 (x1)	1995 (x1)	2957 (x1)	2133 (x1)	2208 (x1)
twitter50^s	15103 (x1)	14626 (x1)	8396 (x1)	7349 (x1)	6466 (x1)	6176 (x1)	5869 (x1)	5547 (x1)
road_usa^m	133 (x1)	795 (x1)	816 (x1)	805 (x1)	1024 (x1)	927 (x1)	907 (x1)	900 (x1)
osm-eur^m	1010 (x1)	2688 (x1)	2254 (x1)	2199 (x1)	2090 (x1)	2110 (x1)	2109 (x1)	2029 (x1)

Application: PageRank on Atos

dataset	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
soc-LiveJournal1^s	**112 (x9.44)**	**55.8 (x18.9)**	**41.5 (x15.9)**	**36.6 (x18.0)**	**34.1 (x19.5)**	**28.7 (x23.4)**	**30.0 (x22.1)**	**30.7 (x20.6)**
hollywood-2009^s	**74.1 (x6.13)**	**39.7 (x17.6)**	**35.2 (x22.6)**	**30.6 (x26.4)**	**30.3 (x26.8)**	**29.0 (x27.9)**	**28.8 (x36.0)**	**29.8 (x33.4)**
indochina-2004^s	**51.2 (x57.5)**	**66.0 (x39.6)**	**48.2 (x60.6)**	**32.3 (x82.2)**	**36.8 (x54.0)**	**36.2 (x81.5)**	**34.1 (x62.4)**	**30.2 (x73.0)**
twitter50^s	**11046 (x1.37)**	**5535 (x2.64)**	**3894 (x2.16)**	**3022 (x2.43)**	**2496 (x2.59)**	**2144 (x2.88)**	**1887 (x3.11)**	**1688 (x3.29)**
road_usa^m	**101 (x1.31)**	**62.1 (x12.8)**	**42.8 (x19.0)**	**33.0 (x24.3)**	**26.9 (x38.0)**	**22.3 (x41.9)**	**22.2 (x40.8)**	**22.3 (x40.3)**
osm-eur^m	**991 (x1.02)**	**874 (x3.07)**	**659 (x3.42)**	**512 (x4.29)**	**335 (x6.23)**	**294 (x7.16)**	**199 (x10.56)**	**251 (x8.06)**

TABLE V

BFS AND PAGERANK RUNTIMES IN MS (SPEEDUPS IN PARENTHESES ARE VS. GALOIS) ON SUMMIT (IB). S IS SCALE-FREE, M IS MESH-LIKE.PERFORMANCE LEADERS ARE BOLDED.

uses direction-optimized BFS and Atos only uses push BFS. On mesh-like datasets, Atos achieves a 268x geomean speedup over Galois. This high factor is because BFS on mesh-like datasets is limited by interconnection latency, and Atos is able to send messages to remote GPUs more quickly than Galois because of Atos's smaller message sizes and fast control path to signal remote GPUs.

For BFS on scale-free datasets, Atos's speedup is smaller (although we achieve more than 100x speedup on smaller datasets with an 8 GPU configuration). The primary difference between Galois and Atos is much more communication overhead for Galois, which reduces its ability to fully utilize all communication bandwidth. Atos's GPU-centered communication control path and overlap of communication and computation greatly reduces the overhead of communicating, and its communication aggregation makes the most of available bandwidth.

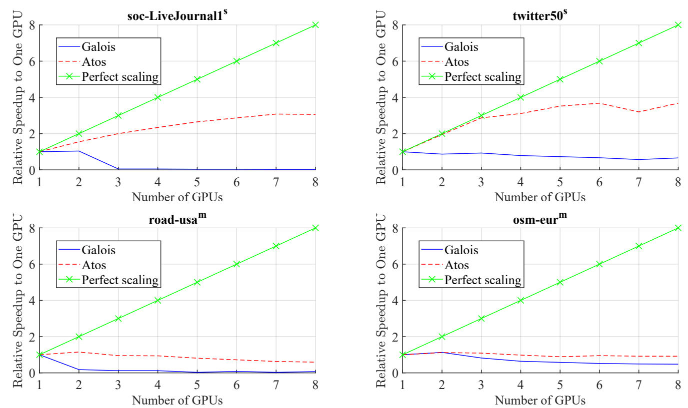

Fig. 8. Strong scaling test for BFS on 4 datasets on an 8-node, 8-GPU InfiniBand system. The plot shows relative speedup for each framework, as a function of the number of GPUs, comparing to its own single-GPU implementation. The green 'x' line in all plots shows perfect strong scaling.

a) Strong Scaling: Figure 8 shows strong scaling for Atos and Galois for BFS on two scale-free datasets (top) and two mesh-like datasets (bottom). Just as with NVLink, the lack of parallelism in mesh-like datasets results in poor strong scalability for any system; however, Atos's strong scalability is consistently better than that of Galois. On scale-free datasets, Galois cannot achieve better performance given more GPUs, primarily because of its high communication overhead. On the largest dataset, twitter50, Galois's fastest implementation

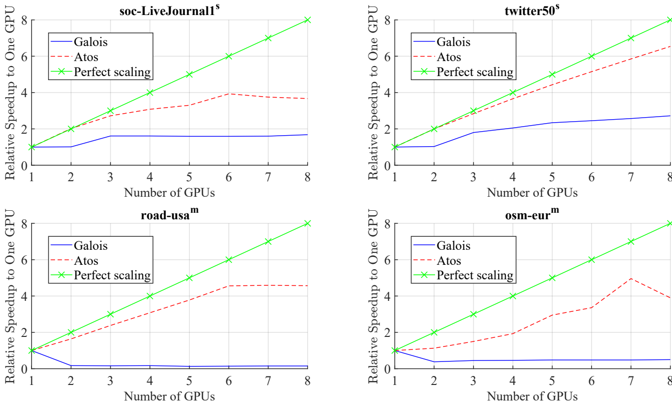

Fig. 9. Strong scaling test for PageRank on 4 datasets on an 8-node, 8-GPU InfiniBand system. The top two datasets are scale-free and the bottom two are mesh-like.

is its single-GPU implementation, which is more than twice as fast as Atos's; this is largely due to algorithmic differences as Galois leverages direction optimization. However, Atos is faster than Galois on any configuration with more than 2 GPUs.

2) PageRank: Atos configures PageRank with *WAIT_TIME* = 32 and a maximum size threshold (*BATCH_SIZE*) of 1 MB. The combination of a relatively high *WAIT_TIME* and a large message means that the aggregator batches more messages before sending. This communication choice is motivated by the fact that PageRank is primarily bounded by interconnection bandwidth and we choose a configuration that maximizes achieved bandwidth even at the cost of latency and overwork.

For PageRank, Atos generally performs better than Galois on both scale-free datasets and mesh-like datasets. Galois also uses asynchronous PageRank, so the performance difference is not due to algorithmic differences. Our IB system has less bandwidth than NVLink, but we still show a performance advantage even for the bandwidth-limited PageRank. Just as in the NVLink system, the superior performance of Atos comes from (1) spread-out communications that smooth the interconnection usage surge that typically occurs when communication is isolated in a single phase; (2) small messages that can better overlap with computation and thus better hide latency; and (3) the GPU-centric control path for communication that reduces latency compared to a CPU control path.

a) Strong Scaling: Figure 9 shows strong scaling behavior for Atos and Galois for PageRank on two scale-free datasets and two mesh-like datasets. Atos achieves better strong scalability than Galois; on all datasets, Atos becomes faster with more GPUs whereas Galois becomes slower with more GPUs.

V. Conclusion

Ken Batcher noted that "A supercomputer is a device for turning compute-bound problems into I/O-bound problems." Scaling many applications, including graph analytics, on GPUs is challenging due to the network rapidly becoming a bottleneck as problem sizes and machines become larger. Thus Atos primarily targets this communication bottleneck with a collection of techniques including one-sided communication, fine-grained message sizes, communication overhead reduction, message aggregation, and aggressive communication and computation overlap.

We learned the following lessons for applications …

- Fine-grained one-sided communication enables communication-computation overlap, and also smooths out network usage, leading to improved runtime.
- Asynchronous task-parallel computation exposes more parallelism, as well as reducing synchronization overhead. PGAS-style communication is a natural fit for this computation model.
- Applications prefer to express their communications in the most natural way for that application.
- latency-limited applications (e.g., BFS on mesh-like datasets) benefit from propagating messages as quickly as possible even at the expense of non-ideal bandwidth utilization, whereas bandwidth-limited applications (e.g., PageRank) benefit from sending larger messages to maximize bandwidth usage.

… and for frameworks that implement PGAS-style communication models for GPUs.

- A communication aggregator decouples communication granularity from computation granularity, allowing the user (or potentially the framework) to choose the optimal communication granularity.
- A framework should enable users both to send data in an eager (immediate) and, via aggregation, in a more bandwidth-efficient way.
- A GPU-centric control path for communication reduces communication latency.
- Guaranteeing consistency before communication via synchronization on the CPU is expensive. An asynchronous data structure (in Atos, an asynchronous queue) is necessary to decouple synchronization from data consistency so that communication can be issued within a kernel without synchronization. Implementing such a data structure is tricky.
- A data structure that supports scheduling preferences (such as a priority queue) can significantly improve application performance.

Acknowledgement

This work is supported by the National Science Foundation (NSF) under projects CCF-1823034, CCF-1823037, and OAC-1740333; by the Department of Defense Advanced Research Projects Agency (DARPA) under projects HR0011-18-3-0007 and FA8650-18-2-7835; by an NVIDIA gift and hardware donations; and by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231. We used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[2] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1105–1114.

[3] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, Oct. 2010, pp. 1–3.

[4] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick, "Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, ser. IPDPS 2009, May 2009.

[5] P. Husbands and K. Yelick, "Multi-threading and one-sided communication in parallel LU factorization," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07, 2007. [Online]. Available: https://doi.org/10.1145/1362622.1362664

[6] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra, "Performance of asynchronous optimized Schwarz with one-sided communication," *Parallel Computing*, vol. 86, pp. 66–81, Aug. 2019.

[7] K. Z. Ibrahim, P. H. Hargrove, C. Iancu, and K. Yelick, "An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS 2014, May 2014, pp. 1115–1125.

[8] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the UPC language," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, ser. IPDPS 2004, Apr. 2004, pp. 254:1–254:7.

[9] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS 2006, Apr. 2006.

[10] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "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*, ser. PPoPP '17, Feb. 2017, pp. 235–248.

[11] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017.

[12] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, 2018, pp. 752–768.

[13] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011, pp. 233–248.

[14] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware task scheduling on multi-accelerator based platforms," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE, Dec. 2010, pp. 291–298.

[15] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, "Bulk synchronous parallel computing—a paradigm for transportable software," in *Tools and Environments for Parallel and Distributed Systems*. Springer, 1996, pp. 61–76.

[16] Y. Chen, B. Brock, S. Porumbescu, A. Buluç, K. Yelick, and J. D. Owens, "Atos: A task-parallel GPU scheduler for graph analytics," in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP 2022, Aug./Sep. 2022.

[17] K. Yelick, A. Buluç, M. Awan, A. Azad, B. Brock, R. Egan, S. Ekanayake, M. Ellis, E. Georganas, G. Guidi *et al.*, "The parallelism motifs of genomic data analysis," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190394, 2020.

[18] E. Georganas, M. Ellis, R. Egan, S. Hofmeyr, A. Buluç, B. Cook, L. Oliker, and K. Yelick, "Merbench: Pgas benchmarks for high performance genome assembly," in *Proceedings of the Second Annual PGAS Applications Workshop*, ser. PAW17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3144779.3169109

[19] K. Gupta, J. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Proceedings of Innovative Parallel Computing*, ser. InPar '12, May 2012.

[20] D. Troendle, T. Ta, and B. Jang, "A specialized concurrent queue for scheduling irregular workloads on GPUs," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019, 2019.

[21] B. Kerbl, M. Kenzel, J. H. Mueller, D. Schmalstieg, and M. Steinberger, "The broker queue: A fast, linearizable FIFO queue for fine-granular work distribution on the GPU," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, Jun. 2018, pp. 76–85.

[22] D. Cederman and P. Tsigas, "On dynamic load-balancing on graphics processors," in *Graphics Hardware*, ser. GH '08, Jun. 2008, pp. 57–64.

[23] S. Tzeng, B. Lloyd, and J. D. Owens, "A GPU task-parallel model with dependency resolution," *IEEE Computer*, vol. 45, no. 8, pp. 34–41, Aug. 2012.

[24] T. Groves, B. Brock, Y. Chen, K. Z. Ibrahim, L. Oliker, N. J. Wright, S. Williams, and K. Yelick, "Performance trade-offs in GPU communication: A study of host and device-initiated approaches," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 126–137.

[25] Oak Ridge National Laboratory Leadership Computing Facility, "Oak Ridge Summit," https://docs.olcf.ornl.gov/systems/summit_user_guide.html, 2021, accessed: 2021-09-30.

[26] Gunrock team, "Throughput vs. frontier size," https://gunrock.github.io/docs/frontier.html, 2017, accessed: 2017-09-30.