Gravel: Fine-Grain GPU-Initiated Network Messages

Marc S. Orr
UW-Madison
morr@cs.wisc.edu

Shuai Che
AMD Research
shuai.che@amd.com

Bradford M. Beckmann
AMD Research
brad.beckmann@amd.com

Mark Oskin
AMD Research, University of Washington
oskin@cs.washington.edu

Steven K. Reinhardt
Microsoft
stever@microsoft.com

David A. Wood
AMD Research, UW-Madison
david@cs.wisc.edu

ABSTRACT

Distributed systems incorporate GPUs because they provide massive parallelism in an energy-efficient manner. Unfortunately, existing programming models make it difficult to route a GPU-initiated network message. The traditional coprocessor model forces programmers to manually route messages through the host CPU. Other models allow GPU-initiated communication, but are inefficient for small messages.

To enable fine-grain PGAS-style communication between threads executing on different GPUs, we introduce Gravel. GPU-initiated messages are offloaded through a GPU-efficient concurrent queue to an aggregator (implemented with CPU threads), which combines messages targeting to the same destination. Gravel leverages diverged work-group-level semantics to amortize synchronization across the GPU's data-parallel lanes.

Using Gravel, we can distribute six applications, each with frequent small messages, across a cluster of eight GPU-accelerated nodes. Compared to one node, these applications run 5.3x faster, on average. Furthermore, we show Gravel is more programmable and usually performs better than prior GPU networking models.

CCS CONCEPTS

Computer methodologies→Massively parallel algorithms;

KEYWORDS

message aggregation, graphics processing unit (GPU), fine-grain communication, partitioned global address space (PGAS)

ACM Reference format:

Marc S. Orr, Shuai Che, Bradford M. Beckmann, Mark Oskin, Steven K. Reinhardt, and David A. Wood. 2017. Gravel: Fine-Grain GPU-Initiated Network Messages. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.

DOI: 10.1145/3126908.3126914

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11 $15.00
https://doi.org/10.1145/3126908.3126914

1 INTRODUCTION

GPUs are becoming prominent in high-performance distributed systems. For instance, consider the Green500 list, which tracks the most energy-efficient supercomputers—nine of the top ten systems use GPUs [1]. At the commodity end, cloud platforms now offer GPU computing [2][3][4]. Multiple GPUs are now being used in a coordinated fashion to accelerate single applications ranging from high performance computing [5] to machine learning [6][7][8][9].

Nevertheless, it is surprisingly difficult to route a network message between a GPU thread (called a *work-item* or WI) and the network interface (NI). One challenge is that WIs coordinating to access the NI must accommodate the GPU's data-parallel architecture, where some WIs execute in lockstep. For example, a dependency between WIs executing in lockstep (e.g., a spin lock) can cause deadlock.

A second challenge is managing the cost of producer/consumer synchronization (e.g., reserving space in a shared message queue). In particular, synchronization becomes a bottleneck for *irregular applications*, which are characterized by frequent, small, and unpredictable (i.e., input-dependent) messages. For example, in a graph algorithm it is typical to initiate a small message (e.g., a few bytes) every time a vertex's neighbor resides on a different machine [10]. Prior CPU-based systems, such as Grappa [11] and GraphLab [12], limit synchronization by aggregating messages in per-thread buffers. However, this scheme is a poor fit for GPUs because per-thread aggregation results in branch-divergence and fails to leverage the GPU's memory coalescing hardware.

Despite these challenges, three programming abstractions have been proposed to access the network from the GPU. We consider each model and try to apply them to irregular applications. In each case, we encounter programming difficulties or, even worse, performance limitations.

The first proposal, called the *coprocessor model* [13][14], disallows GPU WIs to access to the NI. Instead, programmers write CPU code for communication before and after a GPU kernel and must manually overlap communication and computation for peak performance. This model's poor programmability is partially offset by its ability to generate large messages, which are ideal for network transmission.

In the second proposal, called the *message-per-lane model* [15][16][17], WIs independently access the NI. Compared to the

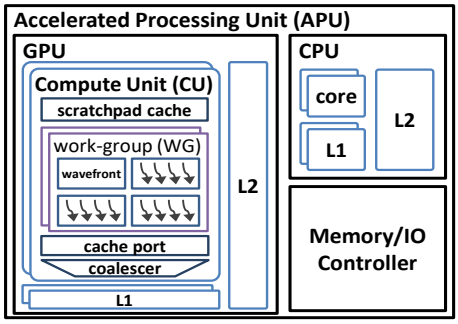


Figure 1. Generic APU architecture.

coprocessor model, this model simplifies programming, but it can generate small, high-overhead messages. For example, NVSHMEM enables GPU threads to read and write another GPU's memory using partitioned global address space (PGAS) semantics [17]. Unfortunately, NVSHMEM is limited to GPUs on the same PCIe or NVLink fabric. Furthermore, it prefers linear/coalescing access patterns, making it a poor fit for irregular workloads.

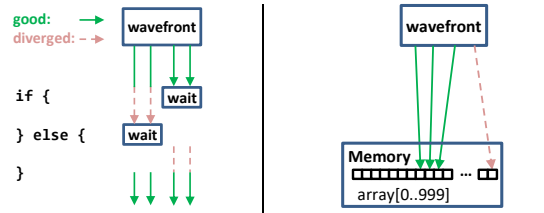
Finally, in the third proposal, called *coalesced APIs* [18][19], WIs coordinate with their neighbors to access the NI. Compared to the message-per-lane model, this model is harder to program, but it uses adjacent WIs to form larger messages. However, for irregular applications, messages are smaller than the coprocessor model.

To address the limitations of prior models and enable the GPU to efficiently initiate small messages, we introduce Gravel. Semantically, Gravel is similar to NVSHMEM in that it enables GPU threads to participate in PGAS-style communication. But, instead of relying on interconnects with limited scalability (e.g., NVLink), Gravel targets large, highly scalable interconnects such as Ethernet and InfiniBand.

Gravel draws inspiration from the GPU's hardware memory coalescer, which operates across a data-parallel instruction to combine accesses to the same cache line. Similarly, network messages incur overhead that can be amortized by combining small messages into larger messages. However, the network induces more overhead, and thus demands a coarser level of aggregation. Specifically, messages targeting the same node (not the same cache line) should be combined and message combining should occur across the GPU (not across a data-parallel instruction).

In Gravel, GPU-initiated network messages are routed through a GPU-efficient producer/consumer queue to an aggregator, which combines messages being sent to the same destination. Gravel amortizes synchronization across adjacent WIs, similar to coalesced APIs, but Gravel does not require WIs to synchronize. Instead, Gravel leverages *diverged work-group-level semantics* to enable the GPU to access the network from divergent code.

To make Gravel work on current GPUs, the aggregator is implemented with CPU threads and software predication is used to achieve the diverged work-group-level semantic. We believe that future GPUs can support these features more efficiently in hardware. For example, we suggest and evaluate two alternatives



(a) Branch divergence. *(b) Memory divergence.*

Figure 2. SIMT effects.

to software predication—a work-group-wide reconvergence stack and fine-grain barriers.

We evaluate Gravel on a cluster of eight AMD APUs connected by InfiniBand. Compared to one node, Gravel achieves a 5.3x speedup on average across six irregular applications. Furthermore, we show that Gravel is more productive and usually performs better than prior GPU networking models.

2 GPU BACKGROUND

In this section, we first describe how the GPU execution model maps GPU threads (WIs) to GPU hardware (§2.1). Next, we discuss the GPU's single-instruction/multiple-thread (SIMT) paradigm (§2.2). Finally, we describe how integrated GPUs enable fine-grain CPU-GPU synchronization, which our work leverages (§2.3).

2.1 GPU Execution Model

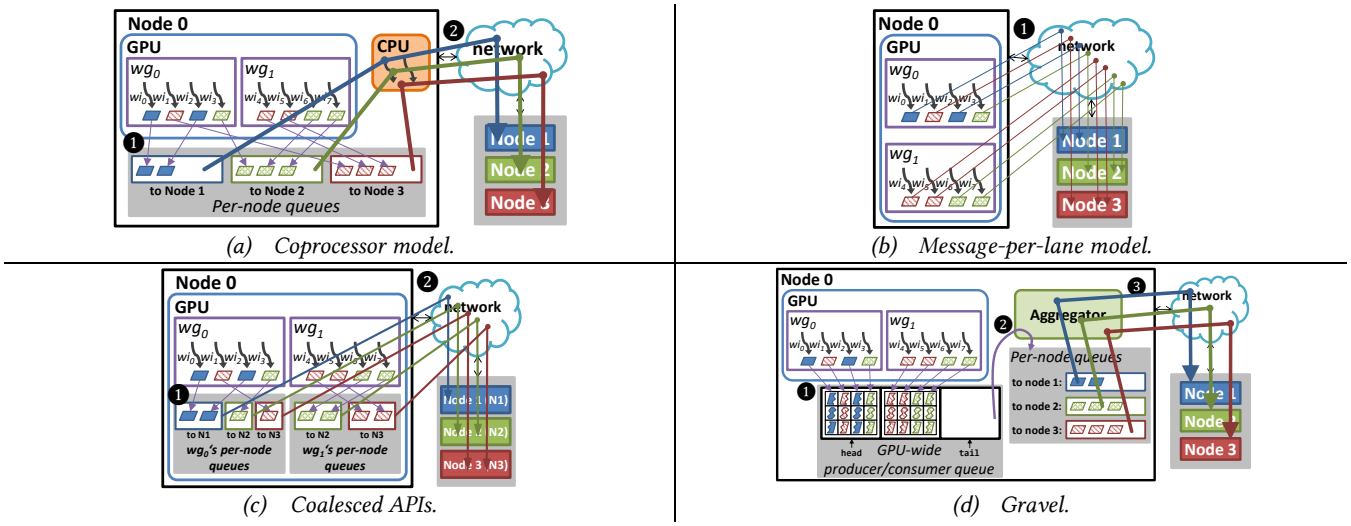
GPU hardware (Figure 1) executes *wavefronts* (WF)—a small number of WIs (e.g., 64 on AMD GPUs) that execute in lockstep. *Work-groups* (WG) comprise one or more WFs that execute on the same GPU core, called a compute unit (CU). WIs in a WG communicate using WG-level barriers and hardware caches—including a programmer-managed scratchpad cache.

These primitives enable WG-level operations, which use the WIs in a WG to index and process a data array. An important WG-level operation is *reduction*, which reduces an array to a single result (e.g., sum, maximum). For example, given the array, `A=[2,1,0,5]`, reduce-to-sum returns `2+1+0+5=8`. Another important operation is *prefix-sum*, which calculates an array's running total. For example, the prefix sum of `A` is `[0, 0+2=2, 0+2+1=3, 0+2+1+0=3]`.

2.2 SIMT Effects

GPU programming languages, like OpenCL [20] and CUDA [21], are SIMT because they present a WI as the unit of execution. But GPUs execute WFs, which exposes two performance effects. Branch divergence, depicted in Figure 2a, occurs when WIs in a WF encounter different control paths. GPUs use hardware predication to execute branches, which causes idle execution units.

Each CU has a coalescer, which operates across the CU's single WF-level cache port to combine memory operations that target the same cache line. Memory divergence, depicted in Figure 2b, occurs when WIs in a WF access different cache lines and is undesirable because WFs stall until all of their cache lines are accessed.



(a) Coprocessor model. (b) Message-per-lane model.

(c) Coalesced APIs. (d) Gravel.

Figure 3. Mapping applications with frequent, small, and unpredictable messages to GPUs in a distributed system.

2.3 Integrated GPUs vs. Discrete GPUs

We focus on systems that support fine-grain shared virtual memory (SVM) [20], where the CPU and GPU synchronize through atomic memory operations. Many integrated GPUs, including AMD's Heterogeneous System Architecture (HSA) [22] and Intel's Graphics Gen9 [23], support this feature.

In contrast, discrete GPUs rely on generated code to copy data between the CPU and GPU at kernel boundaries. This approach provides the illusion of a unified memory, but fails to support CPU-GPU communication from within a kernel. Discrete GPUs could use PCIe atomic operations to synchronize with the CPU from within a kernel. However, discrete GPUs currently lack first-class (e.g., driver and runtime) support for PCIe atomics and current PCIe atomic implementations may limit performance.

3 GPU NETWORKING MODELS

This section explores the programmability and operation of each GPU networking model for irregular applications. The three previously proposed models—coprocessor, message-per-lane, and coalesced APIs—are not designed to handle the small and unpredictable (i.e., input-dependent) messages that frequently occur in irregular applications. Thus, we first try to understand how these prior models can accommodate small messages, which incur very high overhead with a naïve implementation. A recurring theme is to amortize per-message overhead by combining messages that share the same destination into large *per-node queues*, suitable for network transmission.

Table 1. Ranking different GPU networking models.

	coprocessor	msg-per-lane	coalesced APIs	Gravel
SIMT utilization	*	✓	*	✓
large messages	✓	*	*	✓
efficient sync	✓	×	✓	✓
programmability	×	✓	*	✓

* Good for prior workloads studied; bad for small unpredictable messages.

Next, we discuss how Gravel simplifies programming an irregular application by automatically combining small messages. Specifically, we use four criteria to summarize the benefits and limitations of each model: (1) *SIMT utilization*, described in §2.2; (2) *large messages*, meaning that messages are large enough to amortize network overhead; (3) *efficient synchronization*, meaning that WIs coordinate to use the NI efficiently; and (4) *programmability*, meaning that applications are simpler (e.g., fewer lines of code). Table 1 summarizes how each model ranks across these four criteria.

Throughout this section, we use the GUPS micro-benchmark to demonstrate programmability issues. In GUPS, a distributed array, A, is incremented at random offsets obtained from a second local data structure [24]. Figure 4 shows pseudo-code for each model and Table 2 shows line counts for real code.

3.1 Coprocessor Model

In the coprocessor model, programmers write CPU code to handle network communication before and after each GPU kernel. GPUDirect RDMA [13] and CUDA-aware MPI [14] follow this model. To implement an irregular application, a programmer might manually organize messages into per-node queues (Figure 3a), which exposes several low-level issues. Specifically, the programmer must avoid overflowing a queue, manually send and receive the queues, and overlap the sends/receives with GPU execution. The GPU code must efficiently insert messages into the per-node queues.

The pseudo-code (Figure 4a) avoids overflowing a queue by chunking the updates (lines 6-7). Specifically, each chunk is sized to match the per-node queue size. This enables each queue to handle the worst case, where all WIs send messages to the same node. Chunking also helps to overlap communication (lines 8-11) and computation. On the GPU, WGs use WG-level synchronization (§4.1) to efficiently reserve space in the queues (line 4). Note, WG-level synchronization occurs once per destination (lines 2-3), which causes branch and memory divergence.

Table 2. Lines of code for GUPS for each model.

	coprocessor	msg-per-lane & Gravel	coalesced APIs
host	296	174	187
GPU	46	19	131
total	342	193	318

3.2 Message-per-lane Model

In prior work, the message-per-lane model (Figure 3b) requires programmers to manage GPU-initiated messages in a SIMT-efficient way (e.g., DCGN [16]) or requires special hardware (e.g., GGAS [15] or NVSHMEM [17]). We assume that SIMT issues are hidden from programmers and note that Gravel's producer/consumer queue (§4.2) achieves this effect in software.

Figure 4b shows pseudo-code for the message-per-lane model. After launching the GPU kernel (line 16), WIs update slices of `A` (line 15). Table 2 shows that this model (i.e., 193 lines) is more programmable than the coprocessor model (i.e., 342 lines). However, there are two performance issues. First, messages generated by WIs are too small for efficient network transmission. Second, we show in §4.1 that the WF width in a modern GPU is too small to fully amortize synchronization overhead.

3.3 Coalesced APIs.

Coalesced APIs, shown in Figure 3c, are designed to be executed by all WIs in a WG at the same time and with identical arguments (e.g., destination, command, payload). GPUnet [18] and GPUrdma [19] provide coalesced APIs. At first glance, this model seems to degenerate to the message-per-lane model for small random messages. However, the pseudo-code in Figure 4c shows that a tenacious programmer can use the GPU's scratchpad (lines 18-21) to sort a WG's messages by destination (lines 22-25). A counting sort, where the keys are the destination IDs, works well [25]. The sort outputs a contiguous list of messages for each destination targeted by a WG. The pseudo-code then uses a coalesced API, `sync_inc_list`, to send each list.

Our coalesced APIs version (318 lines) is 1.6x more code than our message-per-lane model (193 lines). One issue is the amount of scratchpad used (i.e., a WG with 256 WIs uses 4 kB of scratchpad). A second problem is that aggregating across a WG (instead of the entire GPU) generates small per-node queues. Finally, a third issue is that coalesced APIs are invoked for each destination, which degrades SIMT utilization.

3.4 Gravel

In Gravel (Figure 3d), GPU-initiated messages are routed through a GPU-efficient producer/consumer queue to an aggregator, which repacks the messages into per-node queues and sends them to the NI after they become full or exceed a timeout. The producer/consumer queue interface (§4.2) hides low-level issues like avoiding deadlock between WIs in a WF or optimizing SIMT utilization. Thus, Gravel's pseudo-code (Figure 4b) is identical to the message-per-lane model, but Gravel performs better for two reasons.

First, like the coprocessor model, Gravel's aggregator generates large messages to amortize network overhead. Second,

```
--- GPU kernel ---
 1: gups(B, C, Qs):
 2:   for each node targeted by my work-group:
 3:     if node == C[GRID_ID]:
 4:       MyOff = work_group_level_reserve(&Qs[node])
 5:       Qs[node][MyOff] = B[GRID_ID]
--- host code ---
 6: for (idx = 0; idx < len(B), idx += Q_SZ):
 7:   gups(&B[idx], C, Qs) # on GPU, GRID_WIDTH=Q_SZ
 8:   for each node:
 9:     send Qs[node] to node
10:   for each node:
11:     receive Q from node
12:     for each offset in Q:
13:       A[offset]++
```

(a) Coprocessor model.

```
--- GPU kernel ---
14: gups(A, B, C):
15:   shmem_inc(A + B[GRID_ID], C[GRID_ID])
--- host code ---
16: gups(A, B, C) # on GPU, GRID_WIDTH=len(B)
```

(b) Message-per-lane model & Gravel.

```
--- GPU kernel ---
17: gups(A, B, C):
18:   # allocate data-structures in GPU's scratchpad
19:   int64_t ptrs[WG_SIZE]
20:   int dests[NODE_COUNT]
21:   int cnts[NODE_COUNT]
22:   # After sort: ptrs -> list of per-node Qs; dests
23:   # -> destination list and cnts -> list of
24:   # per-node Q sizes. dcnt = # of destinations.
25:   dcnt = sort(ptrs, dests, cnts, A, B, C)
26:   off = 0
27:   for (d = 0; d < dcnt; d++):
28:     sync_inc_list(&ptrs[off], dests[d], cnts[d])
29:     off += cnts[d]
--- host code ---
30: gups(A, B, C) # on GPU, GRID_WIDTH=len(B)
```

(c) Coalesced APIs.

Figure 4. GUPS pseudo-code. `A` is the array being updated. There is a slice of `A`, at the same virtual address, on each node. `B` is a local array of offsets into `A`. `C` is a local array of destinations. `GRID_ID` is a per-work-item identifier used to index data.

Gravel amortizes synchronization across WGs, which is similar to coalesced APIs—but Gravel does not require WIs to operate in a WG-synchronous fashion. Instead, we leverage a diverged WG-level semantic to asynchronously offload messages to the NI (§5). Another alternative is to offload messages at wavefront granularity, which is done in prior work like GGAS [15] and channels [26], but we find that offloading messages at WG granularity is approximately 3x faster (Figure 6, explained in §4.1).

One last subtle point is that Gravel's "CPU-side aggregation" strategy scales better than the "GPU-side aggregation" strategy described for the coprocessor model and coalesced APIs. Specifically, as the number of destinations (and per-node queues) increase, GPU-side aggregation suffers low SIMT utilization because WIs in the same WG write different queues. Conversely, in Gravel the GPU always writes messages to a single queue (i.e., the producer/consumer queue).

4 PRODUCER/CONSUMER QUEUE DESIGN

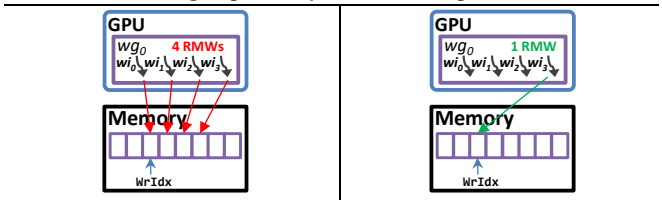
We now describe Gravel's producer/consumer queue, which acts as the GPU's interface to Gravel's aggregator. The queue differs from CPU queues in two important ways. First, it handles SIMT correctness and performance issues that occur when exporting

```
1: work_item_level_reserve(Q):                  sample run:
2:   return fetch_add(Q.WrIdx, 1);              ret: [2,3,4,5]
```

(a) work-item-level synchronization pseudo-code.

```
3: work_group_level_reserve(Q):                 sample run:
4:   lid = LANE_ID; # wi's WG offset            lid:  [0,1,2,3]
5:   max = reduce_max(lid);                     max:  [3,3,3,3]
6:   MyOff = prefix_sum(1);                     MyOff:[0,1,2,3]
7:   Qoff = 0;                                  Qoff: [0,0,0,0]
8:   if lid == max:
9:     Qoff=fetch_add(Q.WrIdx,MyOff+1);         Qoff: [0,0,0,2]
10:  return reduce_sum(Qoff)+MyOff;             ret:  [2,3,4,5]
```

(b) work-group-level synchronization pseudo-code.



(c) work-item-level behavior. *(d) work-group-level behavior.*

Figure 5. Work-item vs. work-group-level synchronization.

messages from the GPU's data-parallel hardware to the NI. Second, the queue limits the frequency of shared-memory synchronization, which is required to coordinate WIs initiating messages in parallel.

First, §4.1 explains how WG-level synchronization enables the GPU to export messages at WG granularity. Next, §4.2 details the producer-consumer synchronization algorithm used to order WIs and aggregator threads accessing the queue. Finally, §4.3 quantifies the queue's performance.

4.1 WG-level Synchronization

The GPU interacts with the aggregator through in-memory queues. For example, to send a message, a WI reserves space in a queue, deposits the message (e.g., command, payload), and notifies the NI that the message is ready to be sent. Thus, producer/consumer synchronization is required to reserve space and again to notify the NI.

Ignoring (for now) the case where the queue is full, a WI can reserve space by using a `fetch-add` to atomically increment the queue's write index. In this approach, depicted in Figure 5 (scenes *a* and *c*), shared-memory synchronization occurs at WI granularity. An alternative, shown in Figure 5 (scenes *b* and *d*), is to leverage SIMT execution so that a *leader WI* synchronizes

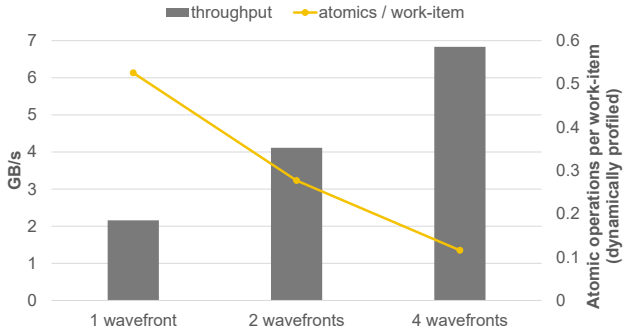


Figure 6. Producer/consumer throughput vs. work-group size.

globally on behalf of its WG. Figure 5b shows that this can be achieved using a few WG-level operations. Specifically, the leader WI is chosen to be the WI with the largest lane ID using a `reduce-max` operation (lines 4-5). Then, a prefix-sum operation is used to determine each WI's local offset (line 6); inactive WIs can cause the local offset to differ from the lane ID. Next, the leader WI reserves a slot for each WI (lines 7-9). Finally, the leader WI broadcasts the WG's queue offset, which is added to each WI's local offset (line 10).

Figure 6 shows how WG size impacts the throughput of Gravel's producer/consumer queue (§4.2) for 32-byte messages; details about the processor are in §6. Larger WGs achieve greater throughput by amortizing atomic operations across more WIs. For example, a WG with four WFs achieves about 3x more throughput than a WG with a single WF by reducing the number of atomic operations by almost 80%. We also measured the throughput of Gravel's producer/consumer queue implemented with WI-level synchronization and found that it is two orders of magnitude slower (0.06 GB/s).

One issue is that WG-level synchronization requires all of the WIs in a WG to participate. As a result, Gravel requires explicit software predication to leverage WG-level synchronization from divergent code. §5 discusses this issue in detail and explores diverged WG-level operations as an alternative for future GPUs.

4.2 Producer/consumer Behavior

The producer/consumer queue's design and operation is illustrated in Figure 7. Each queue slot is arranged as a two-dimensional array, where each column holds a WI's message. This organization enables messages to be written in a non-divergent manner. In our implementation, the first row is used to store the command (e.g., `PUT`, atomic increment), the second row stores the destination, and subsequent rows encode arguments (e.g., address, value).

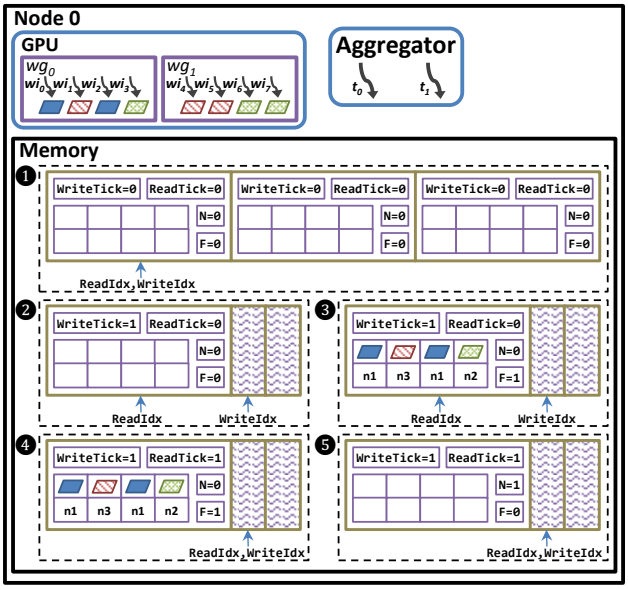


Figure 7. Gravel's producer/consumer behavior.

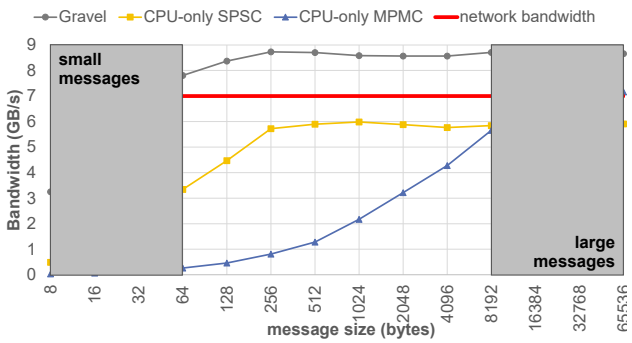


Figure 8. Producer/consumer queue throughput.

In addition to the payload, each queue slot has variables to synchronize producers (i.e., WIs) and consumers (i.e., aggregator threads) and avoid overflowing the queue. To obtain an offset into the queue, `fetch-add` is used to increment `WriteIdx` (by producers) and `ReadIdx` (by consumers). Three situations require synchronization. The first occurs when two or more producers alias to the same array slot. A ticket lock, `WriteTick`, is used to synchronize producers. The second situation occurs when two or more consumers alias to the same array slot. A second ticket lock, `ReadTick`, is used to synchronize consumers. Finally, a full/empty bit, `F`, is used to arbitrate between a producer that has the write ticket and a consumer that has the read ticket.

Figure 7, which focuses on the messages initiated by wg_0, demonstrates the queue's operation. Initially (time ❶), the queue, which has three slots, is empty. At time ❷, wi_3 obtains a write ticket of `0` after performing a `fetch-add` operation on `WriteTick`. Because the write ticket equals the current ticket, `N`, and the full bit, `F`, is clear, wi_3's WG owns the slot. All four WIs (i.e., wi_0-wi_3) write their messages into the slot and wi_3 sets the full bit, `F`, at time ❸. At time ❹, an aggregator thread, t_0, takes ownership of the slot because the full bit, `F`, is set and its read ticket equals the slot's current ticket, `N`. Finally, after the aggregator has consumed the messages, it clears the full bit, `F`, and increments the current ticket, `N`, to release the slot (time ❺).

4.3 Producer/consumer Queue Analysis

Figure 8 shows the throughput of Gravel's producer/consumer queue at different message sizes; WGs have four WFs. The left side of the figure corresponds to small messages (e.g., smaller than a cache line), which incur large overhead. The right side corresponds to larger messages that can be managed using traditional synchronization approaches. The plot demonstrates that Gravel's producer/consumer queue achieves high throughput for small messages. For example, 32-byte messages are processed at 7 GB/s, which matches the network bandwidth in our system (§6).

To put Gravel's performance into perspective, the plot shows two additional producer/consumer queues, where all producers and consumers are CPU threads. The first is a simple single-producer/single-consumer (SPSC) queue [27]. The second is a multi-producer/multi-consumer (MPMC) queue, which uses the same synchronization algorithm as Gravel. The only difference is

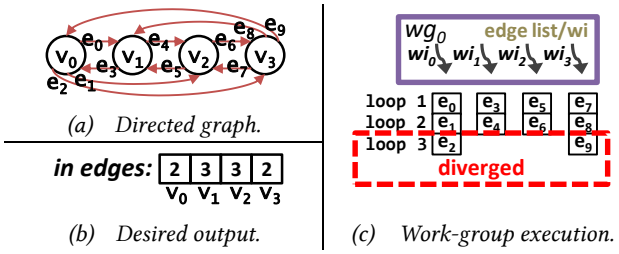


Figure 9. Using WG-level operations in diverged control flow.

that each queue slot is organized to be written by a single CPU thread instead of a GPU WG.

Two factors enable Gravel to offload small messages faster than the CPU-only queues. The first is WG-level synchronization, which amortizes producer/consumer synchronization across a WG—up to 256 messages in our system. In contrast, the other queues require producer/consumer synchronization for each message. The second factor is the payload organization, which allows the WIs in a WG to write messages into the same cache lines. This is possible because WIs in the same WG execute on the same CU. Conversely, extra bytes are appended to the payload in the CPU-only designs to avoid false sharing and this padding adds significant overhead for small messages. For example, in the SPSC queue, three cache lines are read/written to send an eight-byte message—a padded read index, a padded write index, and the padded payload. Things are worse for the MPMC queue. In contrast, Gravel's queue incurs a half-byte of overhead to send the same eight-byte message.

The performance of large messages, which is not the focus of this paper, is explained by how each queue uses the evaluated CPU, which is four-way threaded. The MPMC queue is configured with two producer threads and two consumer threads. Gravel's queue uses all four CPU threads as consumers. Thus, in the limit, Gravel is limited by the throughput of its four consumer threads, the MPMC approaches the throughput of 2 threads, and the SPSC approaches the throughput of a single thread.

5 DIVERGED WG-LEVEL SEMANTIC

Earlier, we described WG-level synchronization (§4.1) and showed that it helps to amortize synchronization (Figure 6). We also noted that software predication is required to leverage WG-level synchronization from divergent code because WG-level operations must occur within converged control flow [20].

In this section, we first provide an example that requires network access from diverged control flow, then show how software predication enables the example to work on current GPUs (§5.1). Next, we define useful behavior for WG-level operations that occur in diverged control flow (§5.2). Finally, we describe how future GPUs can provide this behavior (§5.3).

5.1 Software Predication

To understand how the current behavior of WG-level operations limits Gravel's networking capability, consider the example in Figure 9, which counts the number of incoming edges for each vertex in a directed graph. For instance, in Figure 9a, v_0 has two

```
1: count_in_edges(edge_list, visitors):
2:   for each edge in edge_list:
3:     shmem_inc(&visitors[edge.idx], edge.node)
```
(a) Ideal pseudo-code to count each vertex's in edges.

```
4: count_in_edges(edge_list, visitors):
5:   loop_cnt = reduce_max(edge_list.size)
6:   for i in range(loop_cnt):
7:     active = i < edge_list.size
8:     idx = 0, node = 0
9:     if active:
10:      idx = edge_list[i].idx
11:      node = edge_list[i].node
12:    shmem_inc(&visitors[idx], node, active)
```
(b) Pseudo-code modified to use software predication.

```
13: count_in_edges(edge_list, visitors):
14:   if LANE_ID == 0:
15:     initfbar fb # create fine-grain barrier object
16:   joinfbar fb # start with all work-items
17:   for each edge in edge_list:
18:     shmem_inc(&visitors[edge.idx], edge.node, fb)
19:     if edge + 1 == edge_list.end:
20:       leavefbar fb
```
(c) Pseudo-code modified to use fine-grain barriers.

Figure 10. Diverged work-group-level operation pseudo-code.

incoming edges—one from v_1 and a second from v_3. In general, this problem can be solved by traversing each vertex's outgoing edge list and incrementing a counter once for each neighbor encountered. Figure 9b shows the final counters for the graph in Figure 9a.

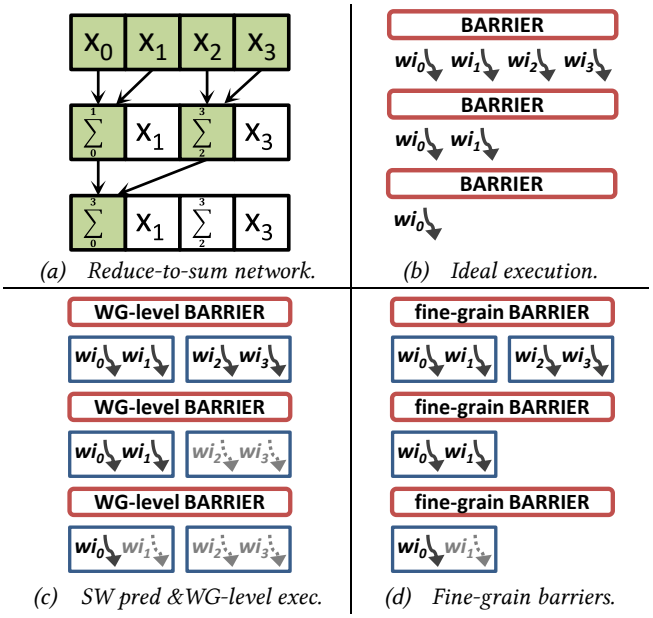
Figure 9c shows one way to distribute this problem using Gravel. Each GPU WI traverses a vertex's outgoing edge list. Figure 11a shows pseudo-code; each WI loops through its edge list and uses a network operation, `shmem_inc`, to update a distributed array of counters. Figure 9c shows that all of the WIs are active during the first two loops. In the third loop, wi_1 and wi_2 become inactive, which prevents wi_0 and wi_3 from leveraging WG-level synchronization to access the network.

Figure 11b shows how software predication can solve this problem. In the code, inactive WIs keep executing with their WG. Specifically, before entering the loop, WIs coordinate to determine the number of loop iterations to execute (line 5). Inside the loop, WIs determine whether they are active (line 7) and if so they construct a network message (lines 9-11). Finally, the network API is extended with an extra argument to differentiate active and inactive WIs (line 12).

Software predication enables WG-level synchronization in divergent code, but it requires a non-trivial code transformation and introduces software overhead. Next, we articulate software predication's behavior and then consider alternatives to achieve that behavior.

5.2 Defining Useful Behavior

This section proposes that WG-level operations occurring in diverged code execute across the *active* WIs in a WG. Specifically, a WI is active if it is predicated on when its WG executes a basic block. Note, our proposal requires a way for the GPU to view the application's control flow at WG granularity (instead of WF granularity) so that it is clear which WIs participate in a given WG-level operation. Below, we describe two useful diverged WG-level operations.



(a) Reduce-to-sum network. *(b) Ideal execution.*
(c) SW pred &WG-level exec. *(d) Fine-grain barriers.*

Figure 11. Diverged reduce-to-sum operation.

Reduction: Active WIs submit a value. Inactive WIs submit a *non-interfering value* (e.g., 0 for reduce-to-sum, `INT_MAX` for reduce-to-minimum). The reduction of these values is returned to active WIs.

Prefix Sum: Active WIs submit a value. Inactive WIs submit the non-interfering value 0. The prefix sum of these values is returned to active WIs.

More generally, non-interfering values are used to implement data-parallel operations. For example, a WG-level sort might be defined such that inactive WIs submit `INT_MAX`, which will be placed at the end of the sorted list, where it can be ignored by the active WIs.

5.3 Supporting Diverged WG-level Operations

WG-level operations use a WG's WIs to index an array and route the respective elements through a data-parallel network. For example, Figure 11a shows a reduce-to-sum network with four elements and Figure 11b shows an ideal execution with four WIs. Note that all WIs must be present to submit their values. Subsequent levels of the network, which are executed by the WIs that submitted values, are separated by a barrier.

In a diverged WG-level operation, the GPU must determine which WFs have active WIs and wait for those WFs to arrive. This is non-trivial because WFs in the same WG progress through the control flow graph at different rates. Next, we discuss three ways to determine the WFs with active WIs.

First, it may be possible to automate the code translation for software predication. One issue with software predication is that it can cause a completely inactive WF to continue executing, as depicted in Figure 11c, because it builds off of WG-level operations. Another approach is to build GPUs that track control flow at WG granularity instead of WF granularity. For example, thread block compaction, proposed to mitigate branch divergence, suggests a WG-level reconvergence stack [28]. Compared to

software predication, this approach does not add software overhead, but it does allow inactive WFs, as depicted in Figure 11c, because it essentially expands the GPU's execution granularity to the width of a WG.

Finally, fine-grain barriers (`fbar`), introduced by HSA, can be used to identify active WIs [22]. An `fbar` enables barrier synchronization across a subset of a WG's WIs. Specifically, HSA provides primitives to create/destroy an `fbar`, register/unregister WIs with an `fbar`, and synchronize the registered WIs. However, HSA's current `fbar` instruction is not able to distinguish WIs in a WF, which is required by our proposal. Thus, we argue that future GPUs should allow an arbitrary set of WIs to be registered/unregistered with an `fbar`. This would allow a compiler to instrument control flow containing WG-level operations with `fbar` operations. This idea is demonstrated in Figure 11c. Unlike the other solutions, this approach does not cause completely inactive WFs to continue executing (Figure 11d).

6 METHODOLOGY AND WORKLOADS

We prototyped Gravel on an eight-node cluster. Each node has an AMD APU with four CPU threads and an HSA-enabled integrated GPU. The nodes are connected by a 56 Gb InfiniBand link. More details can be found in Table 3.

Gravel's aggregator is realized by using the integrated CPU to consume GPU-initiated messages and repack them into per-node queues. We use MPI to send/receive the queues and allocate three queues per node (over allocation helps hide network latency). Each per-node queue is 64 kB, which we found is large enough to obtain most of the benefit of large messages on our system and does not consume an excessive amount of memory.

To obtain the necessary thread support, all network requests are funneled through a dedicated network thread [31]. Upon receiving a per-node queue, the network thread iterates through each message and resolves it as a local memory operation (e.g., load, store). The aggregator performs best with one CPU thread because there are several background threads in the system (i.e., Gravel's network thread, an HSA background thread, and an MPI progress thread).

Currently, Gravel can support the following non-blocking network operations: `PUT`, atomic increment, and a primitive active message API. `PUT` and atomic increment operate on a partitioned global address space (PGAS). Atomic operations (i.e., atomic increment and active messages) are serialized by routing them through Gravel's network thread. Thus, some operations that can execute locally are still routed through the NI. On our system, this approach is faster than using concurrent read-modify-write operations. Furthermore, it simplifies writing active messages.

Table 3. Node architecture.

Processor (AMD A10-7850K)	*CPU*: 2 cores (4 threads); 3.7 GHz; 16 kB L1D; 2 MB L2
	GPU: 8 CUs; 720 MHz; 16 kB L1D; 2 MB L2
Memory	32 GB; DDR3-1600; 2 channels
NIC	56 Gb/s InfiniBand card
Software	Ubuntu 14.04; Open MPI 1.10.1; GCC 4.9.3; HSA runtime 1.0.3
Gravel's configuration	24 per-node queues (each 64 kB; 125 µs timeout); 1 MB producer/consumer queue; 1 aggregator thread

Table 4. Application inputs.

benchmark(s)	inputs
GUPS	~180 million updates
PR-1; SSSP-1; color-1	hugebubbles-00020 [29] (~21 million vertices, ~64 million edges)
PR-2; SSSP-2; color-2	cage15 [29] (~5 million vertices, ~99 million edges)
kmeans	8 clusters, 16 million points
mer	human-chr14 [30] (3.6 GB)

Six applications are evaluated with the inputs in Table 4. The graph applications (i.e., PR, SSSP, and color) are derived from GasCL, which is a single-node graph processing system for GPUs [32]. The following text summarizes each application.

Giga-updates-per Second (GUPS): Described in §3 [24].

PageRank (PR): Ranks web pages by iteratively sending each vertex's rank through its links.

Single-source/shortest-path (SSSP): Calculates the shortest distance from a source vertex to every other vertex.

Graph coloring (color): Labels each vertex in a graph such that no two neighbors have the same color.

Kmeans clustering (kmeans): Iteratively groups a set of Cartesian coordinates into a fixed number of clusters.

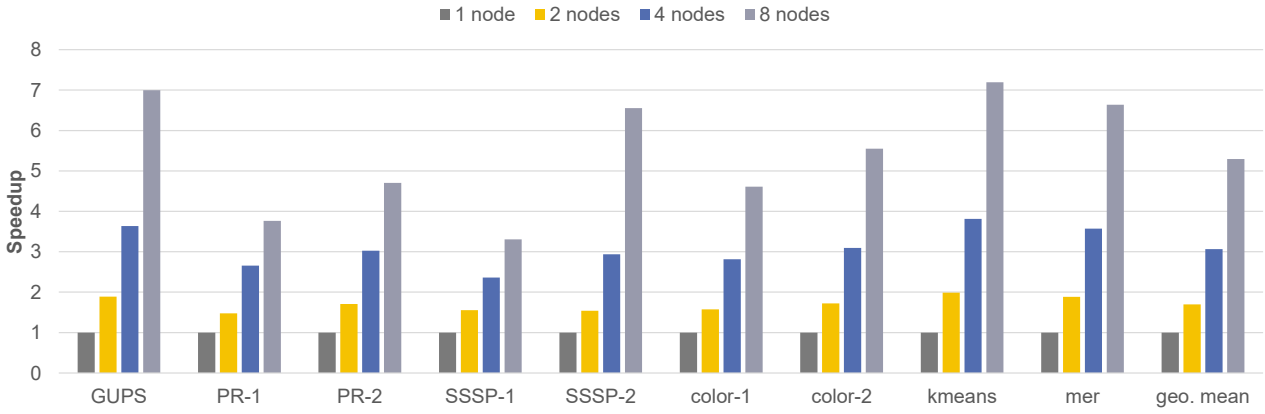


Figure 12. Gravel's scalability.

Table 5. Network statistics for Gravel at eight nodes.

	Remote access frequency	Average message size (bytes)
GUPS	87.5%	65,440
PR-1	37.7%	64,611
PR-2	16.5%	15,700
SSSP-1	30.0%	1,563
SSSP-2	16.2%	57,916
color-1	36.7%	27,258
color-2	16.5%	9,463
kmeans	87.5%	5,656
mer	87.5%	64,822

Meraculous graph construction (mer): Meraculous uses a two phases of a genome sequencing pipeline [33]. Phase 1 builds a distributed hash table and phase 2 traverses it. We evaluate phase 1 and leave phase 2, which has significant branch divergence, for future work.

7 RESULTS AND ANALYSIS

In this section, we analyze Gravel's scalability (§7.1) and then compare Gravel to prior GPU networking models (§7.2).

7.1 Scalability Analysis

Gravel's scalability is depicted in Figure 12. Two factors that impact scalability are the frequency of *remote* data access (i.e., an access through the network), and the cost of a remote access relative to a local access. For each input, Table 5 summarizes the frequency of remote data access and the average message size, which influences the cost of a remote access.

Recall that our implementation serializes atomic operations (i.e., `fetch-add` and active messages) by routing all of them—including local operations—through the NI. Thus, the throughput for atomics is similar for local and remote access. GUPS, kmeans, and mer, use atomics exclusively. Thus, even though these applications are dominated by remote accesses, as shown in Table 5, they approach the ideal speedup of 8x.

PR and color use non-atomic operations (i.e., `PUT` operations) exclusively. A local `PUT` is executed by the GPU directly as a store. Thus, for PR and color, local operations achieve more concurrency than remote operations because they execute across the GPU's massively parallel architecture. In contrast, remote operations are executed by CPU threads (i.e., Gravel's network thread) across the seven receiving machines. We experimented with helper threads

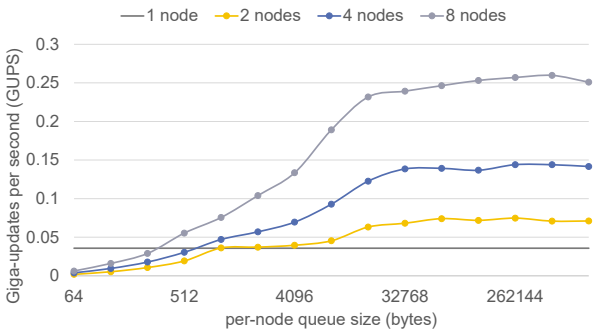


Figure 14. Gravel's aggregation sensitivity.

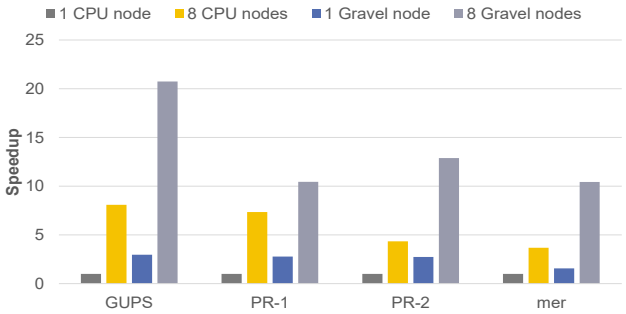


Figure 13. Gravel vs. CPU-based distributed systems.

at the receiver to recover some of the lost concurrency, but the CPU is already saturated. Thus, we observe little benefit.

Finally, SSSP uses atomic operations (i.e., active messages) and `PUT` operations. Specifically, SSSP-2 approaches the ideal speedup because remote access is infrequent and Gravel is able to combine remote accesses into large messages (i.e., ~58 kB as shown in Table 5). In contrast, remote access occurs more frequently in SSSP-1 and the cost of those accesses is higher because Gravel's aggregator is not effective for this input (i.e., messages are ~1.6 kB on average). As a result, SSSP-1 does not scale as well as other inputs.

To put these results into perspective, we compared Gravel to CPU-based distributed systems, which do not to leverage the GPU. Specifically, Figure 13 shows how Gravel compares to Grappa [11] for GUPS and PR and to UPC [33] for mer. Notice that Gravel is significantly faster on one node, where aggregation and networking are irrelevant. Fundamentally, the GPU's massively parallel architecture is better suited to the underlying data-parallel behavior of these workloads and this advantage translates to eight nodes, where Gravel continues to outperform CPU-based systems.

Finally, Figure 14 shows how the per-node queue size, which determines the maximum size of a network message, affects GUPS. In general, larger queues provide better multi-node performance, but the benefit diminishes beyond 32 kB where network overhead is sufficiently amortized. Thus, to obtain good performance without using an excessive amount of memory, we use 64 kB per-node queues.

7.2 Style Comparison

We wrote versions of each application for each GPU networking model using the methodology described in §3. Gravel performs equal to or better than the alternative prior GPU networking models in Figure 15.

First, we summarize the key insights demonstrated by Figure 15. The first two bars in Figure 15 are variants of the coprocessor model. Specifically, they show that the coprocessor model uses memory inefficiently and is not aggressive enough to overlap communication with computation. The third bar in Figure 15, which is a variant of the message-per-lane model, shows that the GPU generates messages that are too small for network transmission. The fourth and fifth bars in Figure 15 are variants of coalesced APIs. These bars show that packing messages within

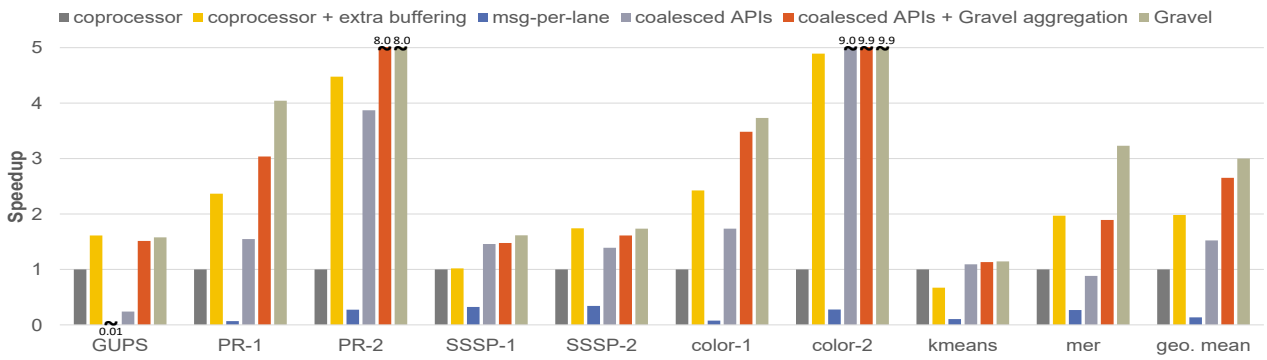


Figure 15. Style comparison at eight nodes.

a WG is not sufficient. Instead, the GPU-wide aggregation taken by Gravel is required to obtain peak performance.

The following paragraphs discuss each set of bars in Figure 15 in more detail. The first set of bars, labeled *coprocessor*, were generated by configuring the coprocessor model to use the same amount of buffering as Gravel. Recall, the number of WIs executing concurrently is limited to avoid overflowing a per-node queue. Such small per-node queues, which are sufficient for Gravel, limit the amount of parallelism on the GPU, causing this version of the coprocessor model to perform worse than Gravel in all cases. This effect is pronounced for PR and color, where WIs access the network many times.

In the second set of bars, labeled *coprocessor + extra buffering*, we allocate 1 MB for each per-node queue, which is an order of magnitude more space for per-node queues than Gravel. While this enables GUPS and SSSP-2 to perform as well as Gravel, most applications still perform worse. This is because Gravel is more effective at overlapping communication and computation. Specifically, in Gravel, per-node queues are sent through the network as soon as they become full or exceed a timeout, while the coprocessor model delays sending a message until the GPU kernel completes. This effect is pronounced for kmeans, which actually runs slower in the coprocessor model with larger per-node queues.

The third set of bars, labeled *msg-per-lane*, bypasses the aggregator (as described in §3). Figure 15 shows that sending small messages directly degrades performance. Similarly, the fourth set of bars, labeled *coalesced APIs*, shows that combining messages across a WG (as described in §3) is not sufficient in most cases. Other than SSSP-1, SSSP-2, and kmeans, coalesced APIs can lead to messages that are too small.

Finally, the second-to-last set of bars, labeled *coalesced APIs + GPU-wide aggregation*, takes messages generated by the coalesced APIs model and repacks (i.e., aggregates) them into even larger messages. The effect is very large messages that resemble those generated by Gravel. Figure 15 shows that this optimization helps coalesced APIs to perform nearly as well as Gravel. Specifically, this experiment shows that Gravel's superior performance over coalesced APIs can be primarily attributed to the fact that it aggregates messages across the GPU instead of across a WG. At the same time, we showed in Section 3.3 (Table 2) that Gravel is more productive than coalesced APIs, which force programmers to manually pack messages within a WG.

One interesting case is mer, which uses more scratchpad than other benchmarks. This scratchpad usage, in combination with the amount of scratchpad used by the coalesced APIs model, limits the number of WIs that execute on the GPU concurrently.

8 NETWORKING ON FUTURE GPUS

In this section, we explore how future GPUs can provide better support for small messages. First, §8.1 suggests replacing Gravel's CPU-based aggregator with dedicated hardware. Next, §8.2 evaluates alternatives to software predication.

8.1 Hardware Aggregator

Currently, Gravel uses the integrated CPU to aggregate messages, which enables us to use current hardware—but this approach is inefficient. Specifically, we found that, even at eight nodes, the CPU's out-of-order, multi-GHz core spends 65% of its time polling for GPU-initiated messages. Furthermore, these hardware threads cannot be used for other tasks.

Dedicated hardware could do aggregation in a more energy- and latency-efficient manner. A hardware aggregator could be fixed-function logic, but a small programmable core would provide more flexibility. For example, modern GPUs and NICs both incorporate control processors that could be used for aggregation. Placing the aggregator in the GPU would allow it be used for other purposes (e.g., task aggregation, memory allocation, etc.) and enable data-parallel optimization (e.g., a GPU-wide memory coalescer).

8.2 Diverged WG-level Operation Analysis

Gravel uses software predication to achieve the diverged WG-level semantic on current hardware, but this approach introduces overhead that could be avoided in a GPU with hardware support. To test this idea we emulate the alternatives proposed in §5.3.

To emulate a GPU that tracks control flow at WG granularity we perform synchronization at WF granularity. Specifically, code without predication (e.g., Figure 11a) works on current GPUs when the WG size is limited to one WF. Using this methodology, we observed a 1.28x speedup over software predication for a modified version of GUPS, called GUPS-mod, where each WI

performs a random number of updates and 95% of WIs perform no updates (otherwise, the benchmark is too memory bound to observe interesting performance effects).

Next, we evaluate fine-grain barriers by emulating the desired behavior in software. We found that our software-based `fbar` operations (e.g., Figure 11c) provide a 1.06x speedup over software predication for GUPS-mod. The software-based `fbar` incurs significant overhead and should be viewed as a lower bound.

9 RELATED WORK

GPUs often lack native support for WIs to initiate I/O. DCGN [16], and GPUfs [34] rely on the CPU to orchestrate I/O. GPUDirect RDMA optimizes network I/O by allowing the CPU to initiate data transfers between a discrete Nvidia GPU and the NIC [13]. GPUnet uses GPUDirect RDMA to provide a coalesced socket API for GPUs [18]. Specifically, WIs manage sockets by sending a request, over PCIe, to a CPU with GPUDirect capabilities. GPUnet does not do Gravel-style message combining and its coalesced APIs are synchronous, while Gravel's network APIs are asynchronous.

Semantically, NVSHMEM is similar to Gravel in that it enables PGAS-style distributed memory for GPUs [17]. NVSHMEM is limited to GPUs on the same PCIe or NVLink fabric. In contrast, Gravel supports large commodity networks like Ethernet and InfiniBand and optimizes for small messages.

Operationally, GGAS [15] and GPUrdma [19] resemble NVSHMEM by enabling GPU-initiated messages without CPU involvement. Specifically, the GPU driver is modified to expose the NIC's doorbell register and GPU code interacts directly with the NIC. This contribution is orthogonal to Gravel's, which focuses on providing an efficient and programmable interface to the network. For example, GGAS interacts with the network at WF granularity, which we showed is not efficient. Meanwhile, GPUrdma uses coalesced APIs, which are less programmable than Gravel.

Channels [26] and DTB [35] explore GPU-wide task aggregation, which is related to GPU-wide message aggregation. Compared to Gravel, channels was prototyped in a simulator and does GPU-side aggregation, which is less scalable than Gravel's CPU-side aggregation scheme. DTB introduces special hardware instead of using shared-memory synchronization.

Gravel's diverged WG-level semantic resembles HSA's `fbar` by enabling a subset of data-parallel lanes to coordinate [22]. It also resembles unconditional operations, described by Hillis and Steele [36], which temporarily activate inactive data-parallel lanes so that they can participate in a data-parallel computation.

CPU-based systems like Grappa [11], GraphLab [12], and GMT [37], aggregate small network messages. Gravel's GPU-compatible aggregation scheme was inspired by these CPU-based systems.

10 CONCLUSION

Gravel enables GPUs to initiate small network messages and we showed that it is more programmable and performs better than prior GPU networking models for all of the workloads and inputs that we evaluated. Our main contributions were to demonstrate that data-parallel hardware can be exploited to amortize synchronization, use this insight to efficiently offload GPU-initiated network messages to Gravel's aggregator, and to explore diverged WG-level semantics.

To prototype Gravel, we leveraged integrated GPUs to use the CPU for aggregation. Gravel could work with discrete GPUs, with the GPU writing to a producer/consumer queue in main memory over PCIe, and using PCIe atomic operations to synchronize. Current PCIe implementations may limit performance, but future interfaces such as CCIX [38] should improve this situation.

Finally, our evaluation was limited to a very small cluster (i.e., eight nodes). CPU-based system (e.g., Grappa [11]) that focus on optimizing communication via aggregation, have shown that they can scale up to 128 nodes. Larger systems could be organized in a logical hierarchy (perhaps mirroring the physical network topology but not required), with multiple levels of aggregation. For example, a two level hierarchy with each level doing a 16-node aggregation supports 256 nodes with one indirect hop

ACKNOWLEDGEMENTS

We thank Greg Rogers, Joseph Greathouse, and Brad Benton for helping to procure and configure the hardware used to prototype and evaluate Gravel. We also thank Gabe Loh and the anonymous reviewers for their helpful feedback. This work was performed while Marc Orr and Steve Reinhardt worked at AMD Research. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos.

REFERENCES

[1] The Green500 List. [Online]. http://www.green500.org/

[2] Amazon Elastic Compute Cloud User Guide for Linux Instances. [Online]. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html.

[3] Microsoft Azure, N Series, GPU enabled Virtual Machines. [Online]. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/#n-series.

[4] Google Cloud Platform: GRAPHICS PROCESSING UNIT (GPU), Leverage GPUs on Google Cloud for machine learning and scientific computing. [Online]. https://cloud.google.com/gpu/.

[5] T. Geller. 2011. Supercomputing's Exaflop Target. In *Commun. Of the ACM*.

[6] M. Abadi *et al*. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *Google preliminary whitepaper*.

[7] D. Yu, K. Yao, and Y. Zhang. 2015. The Computational Network Toolkit [Best of the Web]. In *IEEE Signal Processing Magazine*.

[8] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-Like Environment for Machine Learning. In *BigLearn NIPS Workshop*.

[9] Install GraphLab Create with GPU Acceleration. [Online]. https://dato.com/download/install-graphlab-create-gpu.html/.

[10] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proc. of the ACM SIGMOD International Conference on Management of Data*.

[11] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. 2015. Latency-tolerant Software Distributed Shared Memory. In *Proc. of the USENIX Annual Technical Conference (ATC)*.

[12] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[13] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. Panda. 2013. Efficient Inter-Node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Proc. of the International Conference on Parallel Processing (ICPP)*.

[14] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, and D. Panda. 2011. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. In *Proc. of the International Supercomputing Conference (ISC)*.

[15] L. Oden and H. Fröning. 2013. GGAS: Global GPU Address Spaces for Efficient Communication in Heterogeneous Clusters. In *Proc. of the IEEE International Conference Cluster Computing (Cluster)*.

[16] J. Stuart and J. Owens. 2009. Message Passing on Data-Parallel Architectures. In *Proc. of the IEEE International Symposium on Parallel Distributed Processing*.

[17] S. Potluri, N. Luehr, and N. Sakharnykh. 2016. Simplifying Multi-GPU Communication with NVSHMEM. [Online]. http://on-demand.gputechconf.com/gtc/2016/presentation/s6378-nathan-luehr-simplyfing-multi-gpu-communication-nvshmem.pdf.

[18] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *Proc. of the USENIX Symp. on Operating Systems Design and Implementation (OSDI)*.

[19] F. Daoud, A. Watad, and M. Silberstein. 2016. GPUrdma: GPU-Side Library for High Performance Networking from GPU Kernels. In *Workshop on Runtime and OS Support for Supercomputers*.

[20] OpenCL 2.0 Reference Pages. [Online]. http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/.

[21] CUDA C Programming Guide. [Online]. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[22] HSA Foundation. 2015. HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG) Version 1.0.1.

[23] S. Junkins. 2016. The Compute Architecture of Intel® Processor Graphics Gen9. *Intel whitepaper, v1.0*.

[24] HPC Challenge Benchmark: RandomAccess. [Online]. http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[25] Wikipedia. Counting Sort. [Online]. https://en.wikipedia.org/wiki/Counting_sort.

[26] M. Orr, B. Beckmann, S. Reinhardt, and D. Wood. 2014. Fine-Grain Task Aggregation and Coordination on GPUs. In *Proc. of the International Symp. on Computer Architecture (ISCA)*.

[27] H. Levy. 2003. Single Producer Consumer on a Bounded Array Problem. *Course notes*. [Online]. https://courses.cs.washington.edu/courses/cse451/03wi/section/prodcons.htm.

[28] W. Fung and T. Aamodt. 2011. Thread Block Compaction for Efficient SIMT Control Flow. In *Proc. of the International Symp. on High Performance Computer Architecture (HPCA)*.

[29] University of Florida Sparse Matrix Collection. [Online]. http://www.cise.ufl.edu/research/sparse/matrices/.

[30] NERSC. Meraculous Data. [Online]. http://portal.nersc.gov/project/m888/apex/Meraculous_data/.

[31] OpenMPI FAQ. [Online]. https://www.open-mpi.org/faq/?category=supported-systems#thread-support.

[32] S. Che. 2014. GasCL: A Vertex-Centric Graph Model for GPUs. In *Proc. of the IEEE High Performance Extreme Computing Conference (HPEC)*.

[33] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and Katherine Yelick. 2014. Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[34] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. 2013. GPUfs: Integrating File Systems with GPUs. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[35] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. 2015. Dynamic Thread Block Launch: a Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *Proc. of the International Symp. on Computer Architecture (ISCA)*.

[36] W. Hillis and G. Steele. 1986. Data Parallel Algorithms. In *Comm. of the ACM*.

[37] A. Morari, A. Tumeo, D. Chavarria-Miranda, O. Villa, and M. Valero. 2014. Scaling Irregular Applications through Data Aggregation and Software Multithreading. In *Proc. of the International Parallel and Distributed Processing Symp. (IPDPS)*.

[38] CCIX Consortium. Cache Coherent Interconnect for Accelerators (CCIX). [Online]. http://www.ccixconsortium.com