ComP-Net: Command Processor Networking for Efficient Intra-kernel Communications on GPUs

Michael LeBeane
The University of Texas at Austin
Advanced Micro Devices, Inc.
mlebeane@utexas.edu

Khaled Hamidouche
Advanced Micro Devices, Inc.
Khaled.Hamidouche@amd.com

Brad Benton
Advanced Micro Devices, Inc.
Brad.Benton@amd.com

Mauricio Breternitz
INESC-ID & IST
University of Lisbon
mbreternitz.ist@gmail.com

Steven K. Reinhardt
Microsoft Corporation
stever@microsoft.com

Lizy K. John
The University of Texas at Austin
ljohn@ece.utexas.edu

ABSTRACT

Current state-of-the-art in GPU networking advocates a host-centric model that reduces performance and increases code complexity. Recently, researchers have explored several techniques for networking within a GPU kernel itself. These approaches, however, suffer from high latency, waste energy on the host, and are not scalable with larger/more GPUs on a node. In this work, we introduce Command Processor Networking (ComP-Net), which leverages the availability of scalar cores integrated on the GPU itself to provide high-performance intra-kernel networking. ComP-Net enables efficient synchronization between the Command Processors and Compute Units on the GPU through a line locking scheme implemented in the GPU's shared last-level cache. We illustrate that ComP-Net can improve application performance by up to 20% and provide up to 50% reduction in energy consumption vs. competing networking techniques across a Jacobi stencil, allreduce collective, and machine learning applications.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**;

KEYWORDS

GPUs, Programming Models, RDMA networks

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243179

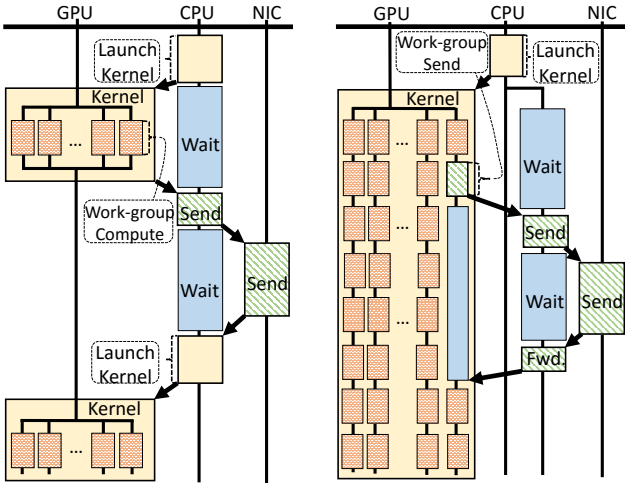
ACM Reference Format:
Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2018. ComP-Net: Command Processor Networking for Efficient Intra-kernel Communications on GPUs. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243176.3243179

1 INTRODUCTION

GPUs are pervasive in data centers and high-performance computing (HPC) environments. At the time of this writing, 98 of the top 500 supercomputers leverage some form of GPU acceleration [39]. GPU powered machines offer extreme levels of computational throughput, memory bandwidth, and energy efficiency for structured, data-parallel workloads across a very wide class of high-performance applications [24].

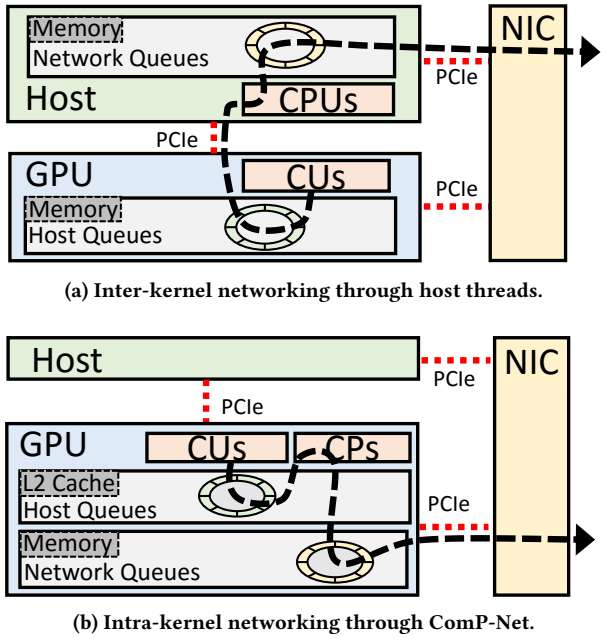
To solve the largest and most difficult problems, multiple GPUs are deployed across many compute nodes. These compute nodes typically employ high-performance network adapters to communicate with devices on remote machines. Current industry-driven technologies such as peer-to-peer data transfer from a GPU's discrete memory to the NIC [21] and direct initiation of network operations by the GPU front-end [2, 34] have optimized both the data path and portions of the control path to flow directly from the GPU to the network adapter.

Despite much progress in the area, traditional multi-node GPU clusters communicate data across the network between subsequent kernel launches. Restricting communication in this manner forces the programmer to think about communication separately from the computation in a completely different device instead of embedding network runtime calls directly within the kernel itself. Inter-kernel networking can also impose performance challenges when networking is frequent compared to computation and limits the class of algorithms that can be offloaded to a GPU. To put this into



(a) Inter-kernel networking. (b) Intra-kernel networking.

Figure 1: Comparison of control flow for inter- and intra-kernel GPU networking. Intra-kernel networking allows for computation and communication overlap at work-group granularity inside a GPU kernel.



(a) Inter-kernel networking through host threads.

(b) Intra-kernel networking through ComP-Net.

Figure 2: Comparison of ComP-Net to traditional intra-kernel networking schemes.

perspective, waiting for kernel tear-down/startup has been shown to take upwards of 10μs [19]. This is an order of magnitude greater than modern network latencies, which hover around 0.7μs at the time of this writing [22].

Recently, a number of researchers have explored initiating GPU communications from within a kernel itself [10, 12, 16–19, 27, 28, 31, 32, 37], in a similar manner that one would initiate communication on the host through a network runtime like the Message Passing Interface (MPI). While some researchers have focused on running a full network stack on the GPU, most practical solutions involve using helper threads on the host to send messages on behalf of the GPU.

Figure 1 compares and contrasts these intra-kernel networking techniques with traditional inter-kernel networking. Intra-kernel networking approaches utilize a level of indirection where the GPU communicates with the host through producer/consumer queues that exist either in GPU or host memory. The host CPU is responsible for servicing requests from these queues and placing them into traditional networking queues attached to a NIC. Requests that complete from the NIC are forwarded back to the GPU through an opposite sequence of steps. This programming model removes the high price of ending a kernel strictly for communication and allows for communication and computation to overlap at the level of a GPU work-group.

While host CPU service threads are an obvious way to enable intra-kernel networking, there are a number of critical limitations to this approach. Helper-thread-based, intra-kernel networking requires a number of long latency IO operations before a network operation even begins. In fact, synchronization between a CPU and discrete GPU may not even be possible from within a kernel depending on the supported memory consistency model of the platform. Additionally, previously proposed GPU networking solutions require a prohibitive number of host threads to scale beyond a single GPU. As the number of compute units on a GPU (or the number of GPUs in a node) continues to grow, much of the host CPUs' cycles will be spent servicing network operations. Finally, requiring the host to constantly poll on remote GPU memory consumes a non-trivial amount of energy.

In this work, we improve the performance and energy consumption of GPU-initiated communication using a little-known feature of modern GPUs: embedded, programmable microprocessors that are typically referred to as *Command Processors (CPs)*. These processors exist on the GPU device itself and are utilized to perform the serial tasks involved with launching and tearing down a GPU kernel [4, 30]. However, in the presence of intra-kernel networking, programmers are encouraged to use larger (less) kernels, as they no longer need to break down kernels across network communication points. This leaves the Command Processors otherwise idle and available to assist with GPU networking.

Our solution, called **Com**mand **P**rocessor **Net**working (ComP-Net), moves the network service thread from the host CPU to the GPU-resident CP. Figure 2 compares ComP-Net to traditional intra-kernel networking schemes where

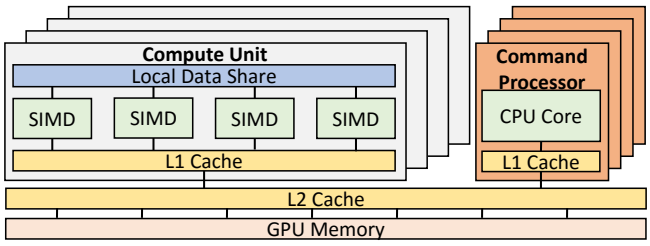


Figure 3: Graphics Core Next (GCN) GPU architecture. [4].

the network service threads reside on the host. In ComP-Net, GPU work-groups submit networking operations to the CP through the GPU's shared cache hierarchy on per-work-group command queues. By hosting the networking runtime on the CP vs. the host CPU, we achieve a reduction in latency for network operations, an increase in scalability in multi-GPU systems, and a decrease in energy consumption associated with the network service thread.

This paper describes the following contributions of the ComP-Net system:

- We describe the runtime architecture and programming interface of ComP-Net.
- We discuss practical challenges related to the relaxed memory consistency model on the CP and the GPU.
- We discuss ways to mitigate perform problems when sharing data between these two devices by applying some simple architectural enhancements to the GPU's L2 cache.
- We show that ComP-Net can improve application performance by 20% and reduce energy consumption of the network service thread by up to 50% vs. other intra-kernel networking designs on a 2D Jacobi stencil, allreduce collectives, and machine learning workloads.

2 BACKGROUND

This section describes the technology upon which ComP-Net is built.

GPU Compute Architecture: Figure 3 illustrates the relevant components of a compute optimized GPU. GPUs are comprised of a number of Compute Units (CUs), each of which are comprised of a collection of Single Instruction, Multiple Data (SIMD) units. Each CU is connected to a private L1 cache and shared L2 cache, which are maintained by explicit cache management instructions. Groups of work-items (also known as threads) are dispatched on the CUs in bundles known as wavefronts (also known as warps). These wavefronts are further bundled into work-groups (also known as thread blocks). Work-groups are guaranteed to execute on the same CU and can therefore make use of fast scratch-pad memory called the Local Data Share (LDS).

Command Processor: Of particular importance for ComP-Net is the block known as the Command Processor (CP). The primary responsibility of the CP is to manage the scheduling, launch, and tear-down of GPU kernels by serving as an intermediary between the host CPU and the GPU's work-group scheduler. In this paper, we assume a programmable CP implemented as a general-purpose CPU with private L1 instruction and data caches. The CP is hooked up to the GPU through a shared L2 cache, as described in the prior art [30].

GPU Memory Consistency Model: The GPU operates under a weak memory consistency model [14]. In our system architecture, we assume that CUs in the GPU and the CPs have private L1 caches which are not coherent with the rest of the system or with each other. To share data between CUs and CPs, the programmer needs to use scoped synchronization operations. A scope defines the level of visibility that a synchronization command operates (e.g., local, device, and system). In this paper, we assume that a synchronization can either be a release operation, which ensures that all previous memory operations have been made visible to the requested scope, or an acquire operation, which ensures that we see the newest data for all memory operations below the synchronization point. These concepts map to ISA instructions that perform cache maintenance operations and memory fences in the hardware.

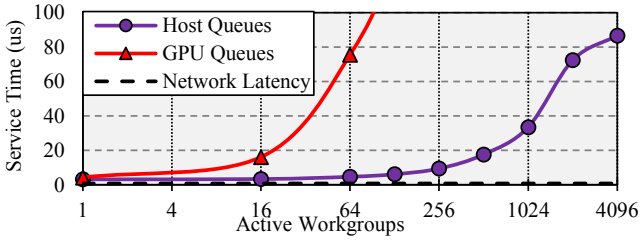
RDMA and OpenSHMEM: Remote Direct Memory Access (RDMA) technology can completely avoid the target CPU when performing network operations and are implemented in many high-performance networking protocols [15, 35]. RDMA technologies are typically used to implement one-sided communication semantics, such as those provided by OpenSHMEM [7]. OpenSHMEM is a Partitioned Global Address Space (PGAS) library specification that defines many different one-sided operations, such as remote *Puts* and *Gets*, as well as use synchronization primitives and collectives. In this paper, we have designed ComP-Net according to the semantics of the OpenSHMEM network programming standard.

3 MOTIVATING COMP-NET

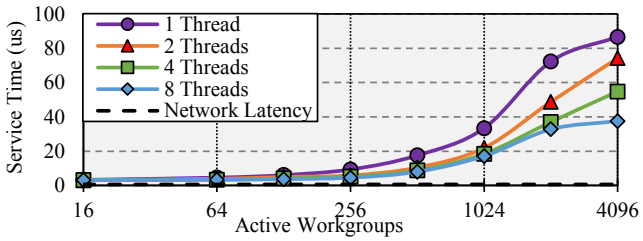
In this section, we will dive into the limitations of existing intra-kernel networking schemes that use threads on the host.

3.1 High Latencies

Most currently existing intra-kernel networking schemes require communication between CPU network service threads and a GPU's work-groups. Unfortunately, in discrete GPU form factors, these two devices are separated by a high latency IO interconnect.



(a) Average service time of CPU/GPU queues with the queue placed on the GPU and in host memory.



(b) Average service time of CPU/GPU queues with varying number of CPU helper threads.

Figure 4: Performance of intra-kernel networking using host threads

Figure 4a shows the latencies for intra-kernel communication between a host CPU and GPU, as observed by work-groups on the GPU. This experiment uses a simple producer/-consumer queue for communication between the CPU and the GPU. There are two locations in which the queue can be placed. In the first design, the command queue is placed in the GPU device memory. The queue is either mapped to the CPU's address space and accessed with loads and stores, or it is accessed by a runtime call. Either way, the performance is poor, especially when the CPU is required to monitor multiple queues at once, since the long latency reads block the CPU from making forward progress.

In the second design, the command queue is placed in host memory. The GPU maps the host memory to its address space and does PCIe stores and atomics to synchronize. While this approach performs better than the previous design, the access latency is still high, on the order of 5-80μs, which is 1 or 2 orders of magnitude higher than network latencies of 0.7μs [22].

No matter where the queue is placed, latencies are high. We are not the first to note this restriction. Previous works have illustrated considerable latencies that far surpasses the latency of a network interface. For example, DCGN [37] quotes latencies of 330μs and Gravel [31] uses a 125μs time-out to flush pending messages. Even recent works on powerful modern hardware, such as dCUDA [12], only achieve latencies of approximately 20μs in the best case.

While high latencies may not matter much when performing bulk synchronous transfers of large data, many network applications, even on a GPU, require more than just support for streaming transfers. Even applications that are mostly parallel still have frequent periods of serial behavior that cannot be easily overlapped. Consider the popular stencil pattern of computation, which is frequently accelerated on GPUs. In these applications, a reduction is typically performed after each relaxation phase to determine whether a convergence criterion has been met. After a local reduction across the GPU, each device contributes a small amount of data, such as the calculation of residuals or synchronization between time-steps of an iterative calculation. This step of the algorithm resides directly on the critical path where there is not enough parallelism for latency hiding to apply. It is precisely these use cases that we target with ComP-Net.

3.2 Poor Scalability

When using intra-kernel networking, it is very likely that there will be many work-groups which need to access the network simultaneously. Figure 4b shows an implementation of intra-kernel networking on the host that sweeps both the number of work-groups participating in a network operation and the number of host threads allocated to service these requests. We can see from this graph that a large number of host threads are required to maintain reasonable quality of service for network operations on a *single* 64 CU GPU. Requiring a large number of threads to service the GPU limits the scalability of the design. The number of required threads will only become more of an issue as the number of CUs on a GPU, and the number of GPUs attached to each CPU socket, continues to increase. Recent trends show us that many GPUs per node should be expected in the highest performing systems [26].

There are also second order effects associated with consuming many cores on the host. For workloads that could benefit from simultaneous CPU compute, these helper threads draw from resources that would otherwise be used by the application. For workloads that only use the GPU, host threads burn unnecessary power and prevent the host CPU from entering a deeper sleep state.

3.3 The Case for ComP-Net

ComP-Net implements intra-kernel networking while simultaneously addressing all the above concerns. While using a networking CP, latency is drastically improved. The CP/GPU command queue is placed directly in GPU memory, which both the CP and GPU can access without traversing an IO bus. Additionally, the CP is located behind the GPU's L2 cache. This means that the CP and the GPU can communicate on the order of hundreds of GPU cycles if the data is

```
__host__ void
hostInit()
{
    // ❶ Initialize ComP-Net
    cpnet_handle_t* cpnet_handle;
    cpnet_init(&cpnet_handle, GRID_SZ / WG_SZ);
    // ❷ Allocate symmetric heap memory
    char* buf = cpnet_shmalloc(sizeof(char) *
                               GRID_SZ / WG_SZ);
    // ❸ Initiator/target launches kernel
    if (cpnet_handle->pe == INITIATOR) {
        hipLaunchKernel(Ping, GRID_SZ,
                        GRID_SZ / WG_SZ, 0, 0,
                        cpnet_handle, buf);
    } else { /* Launch target kernel. */ }
}
```

(a) Initialization and host code.

```
__device__ void
Ping(cpnet_handle_t *cpnet_handle
     char* wg_buffer)
{
    // ❹ Extract context from global handle
    __shared__ cpnet_ctx_t cpnet_ctx;
    cpnet_ctx_create(cpnet_handle, cpnet_ctx);
    // ❺ Each WG pings target
    cpnet_shmem_char_p(cpnet_ctx,
        wg_buffer[hipBlockIdx_x],
        1, TARGET);
    // ❻ Each WG waits for pong target
    cpnet_shmem_char_wait_until(
        wg_buffer[hipBlockIdx_x, 1);
    cpnet_ctx_destroy(cpnet_ctx);
}
```

(b) Device ping to remote GPU using ComP-Net.

Figure 5: ComP-Net ping/pong example on host and device.

resident in the L2 cache. While this is still quite a bit higher than cache-to-cache communication between threads on a CPU, it is far less expensive than synchronizing over PCIe.

The careful reader will observe that an APU form factor (e.g., a CPU and a GPU integrated on the same die sharing the same memory) can also avoid expensive synchronization over PCIe. However, APUs are currently limited to low-end desktops and laptops since they do not have enough CUs or memory bandwidth for high performance applications. Additionally, APUs fix the ratio of GPU resources to CPU resources, which significantly decreases flexibility when deploying systems. Therefore, we feel that GPU solutions that communicate with the CPU over a high-latency bus will continue to dominate high performance systems, motivating the need for ComP-Net.

Scalability is also elegantly addressed by ComP-Net. As opposed to a regular CPU, CPs scale naturally with additional GPUs in a system. Since CPs are *a part of* the GPU, adding additional GPUs in a system allows you to gain more CPs for network processing. Additionally, a CP is much smaller than a core on the host, which results in significant savings in power and energy. We show all these effects across several workloads in Section 5.

4 COMP-NET ARCHITECTURE

In this section, we will discuss the device- and host-side API, the runtime architecture, and design optimizations for ComP-Net.

4.1 ComP-Net API

ComP-Net implements an OpenSHMEM-based API that is exposed to the GPU programmer through a device side library. Each ComP-Net operation is implemented as a work-group collective; the runtime executes a work-group barrier after each API call. Work-groups are a natural granularity to perform networking on a GPU. Any larger, and ComP-Net would need to synchronize across work-groups, which is expensive and limits the ability for work-groups to overlap. Any smaller, and message size would most likely be too small to saturate the network link when streaming large messages. The CPU would then need to coalesce messages together, which adds additional latency and ties up the CPU.

Each ComP-Net API call (put/get/collective/etc.) takes the same arguments as a standard OpenSHMEM implementation (source/destination/length/etc.) with the addition of a GPU-only context that provides the information needed to communicate with the CP. These arguments are placed in a producer/consumer queue and forwarded to the CP, the details of which are described in Section 4.2.

The practical details of ComP-Net communication are best described through a small example. Figure 5 illustrates a simple ping-pong benchmark between two GPUs. The example is written using AMD's Heterogeneous-compute Interface for Portability (HIP) [5], which has a very similar syntax to Nvidia's CUDA [25]. The pong step is omitted since it is similar to ping and offers no additional information regarding ComP-Net's API. Figure 5a shows the host-facing API for ComP-Net. First, the host initializes the ComP-Net runtime and creates a handle for the GPU ❶. This initialization step allocates a number of service threads on the GPU's CPs to handle messages and brings up a standard OpenSHMEM runtime under the hood. In our case, we have selected Sandia OpenSHMEM [36], due to its support for contexts and direct

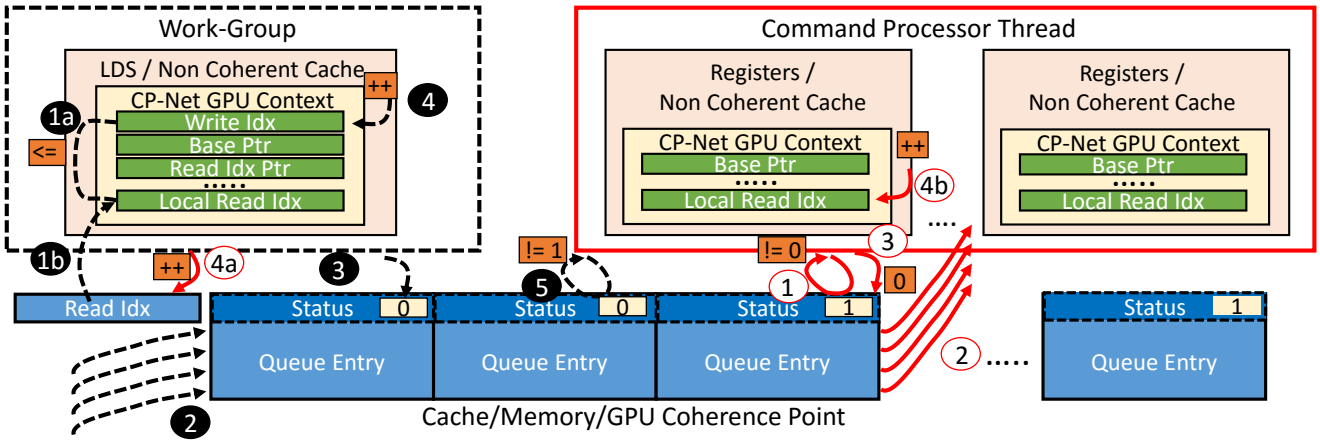


Figure 6: Illustration of work-groups and CP network service threads communicating using ComP-Net.

implementation on top of Portals 4 [35], which is the API for our simulation environment.

Next, the host allocates a network accessible buffer on a symmetric heap allocated from GPU memory ❷. We modify the symmetric heap code in Sandia OpenSHMEM to allow allocation of memory on discrete GPU (dGPU) devices. The details of this are beyond the scope of this work, but GPU-side symmetric heap allocators have been explored in the prior art [13]. Finally, a GPU kernel is launched with the ComP-Net handle and the allocated buffer ❸.

Figure 5b illustrates the device side API from ComP-Net. The GPU first calls an initialization function with the host-provided ComP-Net handle ❹. This API creates a private communication context for each work-group. This context is allocated in scratch-pad memory and initializes its data from the global ComP-Net handle. The next two steps perform standard one-sided network calls to perform a remote put on the target ❺ and wait for the corresponding ping ❻. Each work-group performs a separate ping operation on an independent buffer entry. The details of what happens internally in ComP-Net are described in Section 4.2.

One important detail of ComP-Net is the use of OpenSHMEM contexts. Contexts were recently added to the specification as a way to wait on a subset of the outstanding network operations [7]. While useful for CPUs, this becomes a critical requirement on GPUs. Work-groups should not be stalled waiting for unrelated messages, as this significantly reduces the amount of available communication and computation overlap.

Our prototype ComP-Net implementation does contain some programming model limitations. First, our initial implementation only allows for a single symmetric heap to be bound to a single process. Therefore, all allocated ComP-Net symmetric heap memory is always placed on GPU memory. This restriction is a limitation of the current OpenSHMEM API. Second, ComP-Net can deadlock if there are circular dependences with unscheduled work-groups on the local or remote GPU. This is a limitation of any GPU application that synchronizes across work-groups on the device. However, newer APIs, such as *cuLaunchCooperativeKernel* [25], ensure that GPUs are never oversubscribed. This allows forward progress in the presence of inter-work-group synchronization.

4.2 ComP-Net Design

This section describes the architectural design and optimizations for ComP-Net.

4.2.1 GPU/CP Communication. The CP and the GPU communicate through per-work-group producer/consumer queues. However, building producer/consumer queues between the GPU and the CP on top of a weakly coherent cache hierarchy is different than on a fully coherent CPU cache hierarchy. The memory consistency model of the GPU is described in detail in Section 2, and we will assume the reader is familiar with these details in this section.

Figure 6 describes the details of both the producer and consumer side of a ComP-Net operation. In ComP-Net, each work-group maintains a context that contains the write index, base pointer of the queue, a pointer to the read index that is shared between the CP and work-group, and a local copy of the read index. First, the work-group checks if the queue is full by comparing the local read index to its private write index 1a. If the work-group thinks that the queue is full using its local read index, it goes ahead and refreshes its local copy with the version in shared memory and repeats the step 1b. This reduces accesses to global GPU memory in the common case. Once there is space in the queue, the work-group then fills the slot with all the information necessary to perform a network operation (operation type, destination, length, etc.) and enqueues a release marker to ensure that the

data is visible to the CP ❷. If the data to be sent is less than 8 bytes, it is copied directly into the queue entry to enable the CP to inline the data. Otherwise, a pointer to the data buffer in GPU memory is copied. Next, the work-group sets a status bit in the queue entry to inform the CP that the data is ready for consumption with another device scope write with release marker ❸, and increments its local write index to complete the operation ❹. On a blocking call, or a quiet operation for in-flight non-blocking calls, the work-group needs to check on completion for any outstanding requests. This is done by polling on the status bits on all requests between the read and the write index ❺. An acquire marker needs to be inserted after every iteration of the loop to invalidate the non-coherent L1 cache for the work-group.

On the consumer side, the command processor also keeps a context for each work-group. The command processor adds its local read index to the base pointer of the queue and polls on the status bit of the next queue entry ①. Similarly to the work-group side, the CP needs to enqueue an acquire marker to invalidate its L1 cache. After the CP detects that a network packet is available, it reads out the appropriate data and calls into a standard OpenSHMEM implementation to perform the operation ②. If the operation is non-blocking, the CP immediately marks it as complete by setting the status bit followed by a release operation ③, and increments the read pointer in both its local memory (4b) and on the host (4a). If the operation is blocking, the CP translates the operation into a nonblocking OpenSHMEM call, but does not mark the queue entry as complete. This translation is to prevent the CP network service thread from blocking, which would leave it unable to service other requests from other work-groups. After performing a predefined number of polling rounds through all queues, the CP will complete all outstanding network requests and mark all blocking queue entries as complete.

4.2.2 CP Atomic Operations. Once the networking thread(s) has received a command from the host, it forwards it to an OpenSHMEM library. Largely, the OpenSHMEM implementation is unmodified except for the addition of acquire/release markers to communicate between the CP and the NIC. The procedure is similar to GPU/CP synchronization, except the operations are performed at system scope instead of device scope.

However, while operating in a multi-threaded environment with multiple CPs, our OpenSHMEM library makes heavy use of mutexes to protect shared network data structures across threads. Typically, GPUs resolve device scope atomics at the point of device-level coherence, which, in the case of AMD GPUs, is the L2 cache. CPUs work in an entirely different manner. For our prototype implementation, we assume a CP running an x86 instruction set, which uses

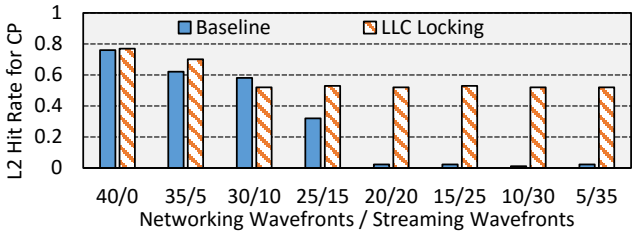


Figure 7: L2 hit rate for CP-generated accesses under different GPU load conditions.

Read-Modify-Write (RMW) prefixes on instructions to take ownership of critical sections. On a standard CPU, this is implemented by two completely different requests. A CPU maintains atomicity by locking either the cache line or the entire response path of the coherent L1 cache. On a non-coherent GPU, this implementation will not guarantee atomicity. Therefore, we introduce a locking cache state to the GPU's L2 cache to support RMW instructions. All RMW instructions issued from the CP automatically bypass the L1 cache. The read cycle in the RMW locks the cache line, and the write instruction unlocks it. This also has the side effect of ensuring that RMWs are not only atomic with respect to other CP threads, but also to the GPU itself.

4.2.3 Controlling Cache Thrashing in ComP-Net. To reduce latency between the CP and the GPU, we would like ComP-Net to leverage the shared Last-Level Cache (LLC) between the two components. Unfortunately, our preliminary design exploration of ComP-Net revealed a major limitation. In most applications that fully utilize the GPU, the time data is resident in the LLC is rather low. This is mainly due to the fact that the GPU performs a significant amount of streaming accesses coupled with the relatively small ratio of cache space to the number of GPU compute threads. This fact has major performance implications on ComP-Net. If enough wavefronts are performing streaming accesses to global memory, the data shared between the CP and the GPU through the LLC would be evicted. This forces the CP and the GPU to communicate through relatively slower GPU memory.

To illustrate this point, Figure 7 shows an experiment where the ratio of networking work-groups to streaming work-groups are sweeped. For the purposes of this experiment, we only use a single CU on the GPU and reduce the size of the L2 cache accordingly. L2 hit rates are reported only from accesses that are generated by the CP. The experiment shows that, in the absence of streaming wavefronts, the L2 hit rate for the CP is relatively high, indicating that the CP and the GPU are sharing data through the L2 cache successfully. However, as successively more streaming wavefronts are added, the CP L2 hit rate plummets to almost zero.

There are many techniques that could be used to solve the above issue. One technique would be to add a dedicated low-latency communication channel for the GPU to signal the CP. However, even simple mailbox-based designs would require significant amount of hardware modification for a narrow use case. Fortunately, the same technique we use for implementing CPU mutexes can be extended to prevent eviction of control plane data from the LLC. We extend the GPU ISA to allow a locked store operation that puts a cache line in the same lock state as a CPU-side RMW operation. This data is only unlocked when it has been accessed by the CP. Since the control plane data is small, and the CP will most likely access the data quickly, the total number of cache lines and the amount of time that a cache line is locked is very small (on the order of 800ns - 1μs).

Using this modification, one final experiment is performed and labeled 'LLC Locking' in Figure 7. LLC locking significantly improves the hit rate for networking work-groups that share an L2 cache with streaming work-groups. However, while the CP hit rate no longer plummets in the presence of streaming work-groups, it is still reduced by 20% in the worst case from the baseline with no streaming interference. This reduction is due to the fact that only data shared by the GPU and the CPU is locked. Data that is used solely by the CP that does not fit in the CPs relatively small L1 cache is spilled out to the L2 and effected by thrashing behavior. This indicates that there are still performance optimizations to be gained by drawing on more sophisticated cache partitioning or locking schemes from the literature. A deeper exploration of this interaction is left as future work.

5 EVALUATION

In this section, we evaluate ComP-Net performance and energy consumption on a number of different workloads.

5.1 Experimental Setup

We evaluate ComP-Net using the open-source gem5 simulator [8] including the AMD public GPU compute model [3] based on AMD GCN architecture [4]. For CPU/CP power numbers, we feed the gem5 output statistics into McPAT [20]. We omit GPU power analysis because we expect it to be similar across all designs.

Table 1 shows the specific configuration for the major components of our infrastructure. We configure our system as a compute node containing a CPU, GPU, CP, and NIC. The CP itself is configured according to the specifications listed in Orr et al [30].

For our experiments, each ComP-Net producer/consumer queue was 64 entries of 64B each for all simultaneously executing work-groups. All proposed designs have the NIC access the queues through PCIe, whether they reside on CPU or dGPU memory. The memory bandwidth consumed by the GPU network queues is minimal. Each network operation requires a 64B command packet, which is small compared to the size of the payload itself and any other data the GPU application may be accessing concurrently.

Table 1: Baseline simulation configuration.

CPU and Memory Configuration	
Type	8 Wide OOO, 16 cores @ 4GHz
I, D-Cache	64K, 2-way, 2 cycles
L2-Cache	2MB, 8-way, 8 cycles
L3-Cache	16MB, 16-way, 20 cycles
DRAM	DDR4, 8 Channels, 2400MHz
GPU Configuration	
Type	AMD GCN3 @1.5GHz
CU Config	12 CUs with 4 SIMD-16 engines
Wavefronts	40 Waves per SIMD (64 lanes)
V-Cache	16kB, 16-way, 12 cycles, per CU
K-Cache	16kB, 8-way, 12 cycles, per 4 CUs
I-Cache	32kB, 8-way, 12 cycles, per 4 CUs
L2-Cache	1MB, 8 banks, 16-way, 150 cycles
CP Configuration	
Type	2 Wide OOO, 2 cores @ 2GHz
D-Cache	32kB, 8-way, 4 cycles
I-Cache	16kB, 8-way, 4 cycles
L2-Cache	Shared with GPU
Network Configuration	
Speed	100ns / 100Gbps
Topology	Star

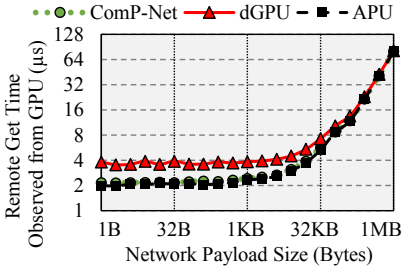
In our experiments, we compare across five different implementations of GPU networking. Here we describe all designs in detail:

CPU: Standard node with just a CPU and a NIC. OpenMP is used for thread-level parallelism, and MPI is used for multi-node communication.

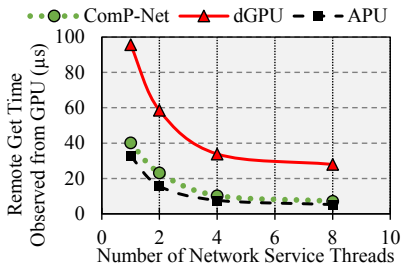
Inter-Kernel: Traditional GPU networking node representative of technologies such as GPUDirect RDMA [21]. Kernels are launched by the host to perform computation, and all networking is routed through MPI at kernel boundaries.

APU: Intra-kernel networking by placing the network thread on the CPU of an APU. The GPU can communicate through host memory and is coherent through a directory-based protocol. This is representative of the Gravel intra-kernel networking implementation for APUs, but without the coalescing [31].

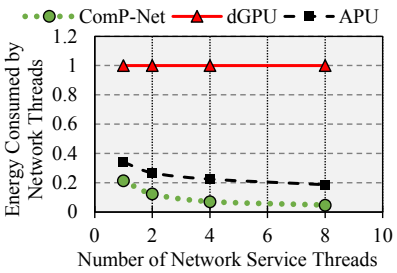
dGPU: Intra-kernel networking by placing the network thread on the CPU of a host machine in a standard off the shelf dGPU-enabled system. The CPU and GPU communicate through a non-coherent PCIe bus model. This is representative of most previous works that have attempted intra-kernel



(a) Sweep of payload size for a single WG and network service thread.



(b) Sweep of threads for 1 Byte transfers with 480 WGs.



(c) Sweep of CPU energy consumption for 1 Byte transfers with 480 WGs.

Figure 8: Microbenchmarks of ComP-Net vs other intra-kernel networking baselines.

networking using helper threads on the host [12, 16, 37]. dGPU also serves as the baseline for all results that report normalized energy consumption or speedups.

ComP-Net: Intra-kernel networking using ComP-Net. The network thread is placed on the CP. The CP and GPU communicate through a shared L2 cache on the GPU.

5.1.1 A note on APU vs. ComP-Net. We wish to address the APU vs. ComP-Net results up front, since they may be surprising to some readers. Although ComP-Net is more energy efficient, for most of our results, we notice that APU and ComP-Net have very similar performance. Since neither communicate over PCIe, this result implies that the gains in synchronizing through the GPU's L2 cache (in ComP-Net) are offset by the relative decrease in performance of a CP vs. CPU for running the network stack itself.

We do not believe that this diminishes the value of ComP-Net. Although we include APU for completeness, most GPU compute deployments employ discrete GPUs, since APU-based designs do not offer enough performance for compute applications. Additionally, APUs fix the ratio of GPU resources to CPU resources, which significantly decreases the flexibility when designing systems. With this in mind, we feel that the correct comparison point for ComP-Net is the other discrete GPU baselines (either Inter-Kernel or dGPU).

5.2 Microbenchmarks

In this section we describe the performance of ComP-Net and competing designs in a number of controlled microbenchmarks. For this section, we will compare against the three intra-kernel networking designs (i.e., dGPU, ComP-Net, and APU).

Figure 8a shows the latency of a single work-group performing remote *Get* network operations of varying payload sizes across different intra-kernel networking designs. dGPU based designs incur over twice the latency of both ComP-Net and APU designs. As the payload size increases, network bandwidth becomes the ultimate determining factor, and all three intra-kernel networking designs perform similarly.

Figure 8b shows the performance of the three intra-kernel networking schemes when fully loading the GPU with network requests. In this example, we schedule 480 simultaneous work-groups of 64 threads each, which will fully saturate our 12 CU system. We then sweep the number of network service threads. Both ComP-Net and APU perform better than dGPU. The dGPU design performs particularly poorly when there are multiple work-groups, since service threads must poll over PCIe.

Figure 8c shows the energy consumption of the previous multi-threaded experiment. For this and all subsequent energy studies, we focus just on the energy consumed by the network service thread(s). We observe that ComP-Net offers significant energy savings over both APU and dGPU. ComP-Net consumes a third of the energy of dGPU, and half the energy of APU.

5.3 Jacobi 2D Stencil

This section evaluates the performance of ComP-Net over a Jacobi relaxation problem. In Jacobi, a series of operations are performed on a local data set, followed by a halo exchange of neighboring data. In our example, a two-dimensional stencil is split in one dimension over all participating nodes. The algorithm follows three main phases. First, the next value of the local stencil is calculated (either on the GPU or the host). Each element in the stencil updates it's value based on the values of each of its 4 neighbors. Next, the halo region is exchanged with a node's adjacent peers. Finally, a residual is reduced over the stencil to determine whether to continue the relaxation.

In the intra-kernel version the host is no longer needed beyond data preparation. Since we can now perform network transfers from within a kernel, the main relaxation loop can be moved onto the GPU. Additionally, work-groups that are performing a halo exchange on the edge of the stencil

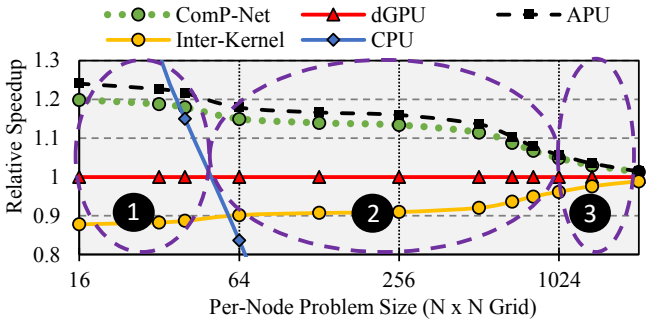


Figure 9: Performance of different networking techniques on various stencil sizes.

can automatically overlap with work-groups on the interior. Without intra-kernel networking, this overlap would need to performed using an exterior and interior kernel.

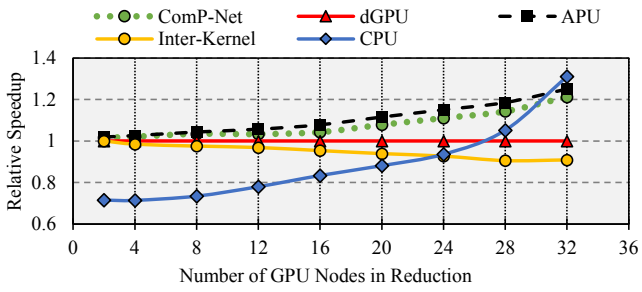
Figure 9 illustrates the results of the Jacobi relaxation on 8 nodes across all of our experimental configurations. The results are presented as speedup to the dGPU baseline and represent a single iteration of Jacobi, and the x-axis sweeps the total number of elements in the stencil per node. The figure shows three regions of interest. In Region ❶, the CPU performs best. This is because the problem size is much smaller than can be accelerated on the GPU. In Region ❷ GPUs start to become advantageous. In this design, ComP-Net and APUs perform better than dGPU and intra-kernel by 10-20%. In Region ❸, all the GPU versions start performing similarly, since the problem size is large enough that intra-kernel networking latencies are no longer the bottleneck.

5.4 Allreduce

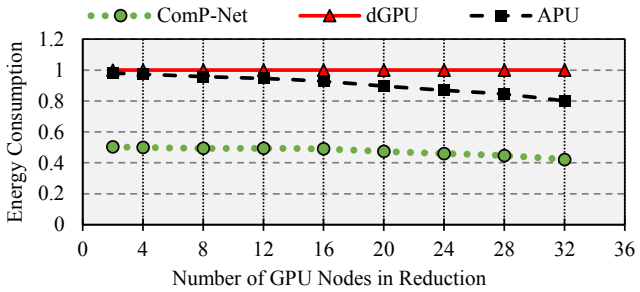
Collective operations are critical for a large number of distributed GPU applications [9, 23]. In this section, we will explore the performance of collective operations by measuring the performance of the allreduce algorithm on ComP-Net, both in isolation, and in the context of machine learning workloads.

In the allreduce collective operation, each node combines the contents of all participating nodes' buffers using a user-specified arithmetic operation. For the CPU and Inter-Kernel baselines, we use the allreduce implementation provided by Baidu [6]. This design implements an allreduce using nonblocking send and receive operations, with computation performed on the CPU or the GPU.

The proposed intra-kernel allreduce design uses a multi-ring algorithm that maximizes GPU and NIC utilization with fine-grained overlap of communication and computation. The number of rings is defined by the number of work-groups participating in the allreduce on each GPU. In $Ring_i$, WG_i on Kernel/GPU of process P receives data from WG_i of process $P-1$ and sends data to WG_i of process $P+1$. The ideal



(a) Performance w.r.t dGPU.



(b) Energy consumption w.r.t dGPU.

Figure 10: Performance and energy of different networking techniques on allreduce of different input sizes.

number of work-groups (rings) per process is a function of the message-size, chunk size, and bandwidth of the network.

Figure 10 shows a strong-scaling study of a 64MB allreduce operation on all the evaluated configurations. In Figure 10a, we see that, for small number of nodes, the GPU results all look similar, since the average problem size per GPU is large and network latencies do not significantly impact performance. As the number of nodes increase and the amount of work per GPU decreases, the performance benefits of ComP-Net over competing approaches (except APU) becomes more pronounced. Eventually, the problem size per node becomes small enough where the reduction is optimally calculated on the CPU. Figure 10b shows the energy consumption of ComP-Net compared to other approaches. We observe that ComP-Net is 50% more energy efficient than both dGPU and APU baselines.

5.4.1 Machine Learning. Deep learning workloads typically use clusters of GPUs to accelerate training and inference of neural networks. For these results, we will focus on the most computationally intensive part, which is training the networks using stochastic gradient descent (SGD) with back-propagation. In distributed SGD, an allreduce operation is used to combine the contents of each GPU's gradient matrix with that of every other GPU. Depending on the number of GPUs participating in the training phase, the allreduce

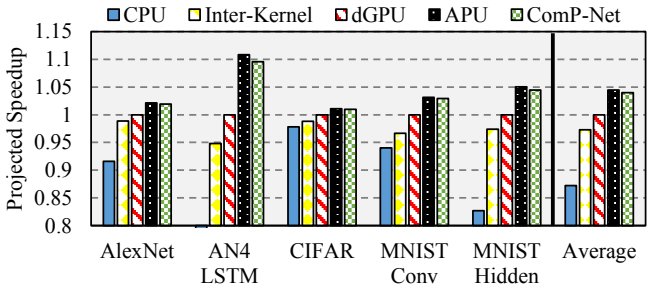


Figure 11: Projected speedups on CNTK workloads with intra-kernel allreduce on ComP-Net.

operation can become a significant bottleneck in the application. For our studies, we have selected five machine learning workloads from a variety of application domains from the Microsoft Cognitive Toolkit [1]. Figure 11 shows how ComP-Net improves performance of GPU training of neural networks on a cluster of 8 nodes with a single GPU each. The results are obtained via a combination of simulation using the parameters described in Table 1 and real hardware runs on the Stampede supercomputer [38]. The difference between the experiments is the device and networking policy (intra- vs. inter-kernel) used for the reduction.

We observe that the biggest jump in performance occurs when switching from a CPU-based reduction to a GPU-based one. Further optimizing the allreduce portion of the training phase through ComP-Net improves total workload performance over a dGPU baseline by 5% on average. However, depending on how much the application is bound by networking, that number can vary from 11% in AN4 LSTM to 2% in CIFAR.

6 RELATED WORK

We are aware of one other work that attempts to leverage an on-chip CP to accelerate GPU workloads. Orr et al. [30] implement the channels programming abstraction [11] on GPUs by using on-chip CPs to perform frequent and complex scheduling decisions.

A few industry efforts optimize GPU networking through an inter-kernel programming interface. GPUDirect RDMA [21], enables InfiniBand NICs to directly access GPU local memory without intermediate data copies. GPUDirect Async [2, 34] implements pre-registration of network operations that can occur asynchronously when a kernel completes.

A number of work implement intra-kernel networking while avoiding CPU helper threads. GPU-TN [19] provides an intra-kernel networking scheme by using a mechanism based on Portals 4 triggered operations [35]. GPU Global Address Space (GGAS) [27] implements intra-kernel networking by adding explicit hardware in the GPU to support a cluster-wide global address space. Oden et al. [29], GPUrdma [10], and Potluri et al. [32] all explore techniques to implement InfiniBand entirely on the GPU. Unfortunately, these works either have challenges with performance [29] or data visibility [10, 32] related to the GPU's relaxed memory consistency model. Klenk et al. [17, 18] explore a number of techniques and communication models to support communication directly from the GPU and show good performance in a number of cases. NVSHMEM uses an OpenSHMEM-like interface to perform intra-kernel communication with other GPUs that reside on the same node [33].

There are a number of works that support GPU networking through helper threads on the host CPU. GPUNet [16] provides a socket-based abstraction for GPUs. Both Distributed Computing for GPU Networks (DCGN) [37] and dCUDA [12] implement a device-side MPI library for GPU kernels that attempts to hide long-latency GPU network events across the cluster. Gravel [31] optimizes irregular GPU messaging applications by employing host-side coalescing of network operations. Gravel is unique among these works in that it focuses solely on APUs.

7 CONCLUSION

In this work, we improved the performance and energy efficiency of intra-kernel communication using a lesser known feature of modern GPUs: embedded microprocessors that are typically referred to as Command Processors (CPs). Our solution, which we call Command Processor Networking (ComP-Net), moves the network service thread from the host CPU over to the GPU-resident CP. In the paper, we described the ComP-Net programming model, discussed a detailed mechanism for GPU/CP synchronization, and implemented architectural modifications to reduce cache thrashing between the GPU and CP. Overall, we show that ComP-Net can improve application performance up to 20% and provide up to 50% energy reduction of networking threads vs. other GPU networking solutions on a Jacobi stencil, allreduce collective, and machine learning workloads.

ACKNOWLEDGMENTS

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Results were obtained in part using resources from the Texas Advanced Computing Center. This research was partially supported by the US Department of Energy under the PathForward program, the National Science Foundation under grants 1745813 and 1725743, and Fundação para a Ciência e a Tecnologia (FCT) under project UID/CEC/50021/2013. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the DoE, NSF, or other sponsors.

REFERENCES

[1] Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, T. Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Hari Parthasarathi, Baolin Peng, Marko Radmilac, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Huaming Wang, Yongqiang Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit.* Technical Report. Microsoft. https://www.microsoft.com/en-us/research/wp-content/uploads/2014/08/CNTKBook-20160217.pdf

[2] Elena Agostini, Davide Rossetti, and Sreeram Potluri. 2017. Offloading Communication Control Logic in GPU Accelerated Applications. In *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid).* https://doi.org/10.1109/CCGRID.2017.29

[3] AMD. 2015. The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5. http://gem5.org/GPU_Models

[4] AMD. 2017. Graphics Core Next Architecture, Generation 3 ISA. http://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual/

[5] AMD. 2018. HIP: Heterogeneous-computing Interface for Portability. http://rocm-developer-tools.github.io/HIP/

[6] Baidu. 2018. baidu-allreduce. https://github.com/baidu-research/baidu-allreduce

[7] Matthew Baker, Swen Boehm, Aurelien Bouteiller, Barbara Chapman, Robert Cernohous, James Culhane, Tony Curtis, James Dinan, Mike Dubman, Karl Feind, Manjunath Gorentla Venkata, Max Grossman, Khaled Hamidouche, Jeff Hammond, Yossi Itigin, Bryant Lam, David Knaak, Jeff Kuehn, Jens Manser, Tiffany M. Mintz, David Ozog, Nicholas Park, Steve Poole, Wendy Poole, Swaroop Pophale, Sreeram Potluri, Howard Pritchard, Naveen Ravichandrasekaran, Michael Raymond, James Ross, Pavel Shamis, Sameer Shende, and Lauren Smith. 2018. OpenSHMEM Specification. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf

[8] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, and Tushar Krishna. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1. https://doi.org/10.1145/2024716.2024718

[9] Ching-Hsiang Chu, Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, and Dhabaleswar K. Panda. 2016. CUDA Kernel Based Collective Reduction Operations on Large-scale GPU Clusters. In *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid).* https://doi.org/10.1109/CCGrid.2016.111

[10] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-side Library for High Performance Networking from GPU Kernels. In *Intl. Workshop on Runtime and Operating Systems for Supercomputers (ROSS).* 6:1–6:8. https://doi.org/10.1145/2931088.2931091

[11] Benedict R. Gaster and Lee Howes. 2012. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer* 45, 8 (August 2012), 42–52. https://doi.org/10.1109/MC.2012.257

[12] Tobias Gysi, Jeremia Bär, and Torsten Hoefler. 2016. dCUDA: Hardware Supported Overlap of Computation and Communication. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC) (SC '16).* Article 52, 12 pages. https://doi.org/10.1109/sc.2016.51

[13] Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. 2016. CUDA-Aware OpenSHMEM: Extensions and Designs for High Performance OpenSHMEM on GPU Clusters. *Parallel Comput.* 58 (2016), 27–36. https://doi.org/10.1016/j.parco.2016.05.003

[14] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[15] InfiniBand Trade Association. 2000. InfiniBand Architecture Specification: Release 1.0.2. http://www.infinibandta.org/content/pages.php?pg=technology_download

[16] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *USENIX Conf. on Operating Systems Design and Implementation (OSDI).* 201–216. https://doi.org/10.1145/2963098

[17] Benjamin Klenk, Lena Oden, and Holger Froning. 2014. Analyzing Put/Get APIs for Thread-Collaborative Processors. In *Intl. Conf. on Parallel Processing (ICPP) Workshops.* https://doi.org/10.1109/ICPPW.2014.61

[18] Benjamin Klenk, Lena Oden, and Holger Froning. 2015. Analyzing Communication Models for Distributed Thread-collaborative Processors in Terms of Energy and Time. In *Intl. Symp. on Performance Analysis of Systems and Software (ISPASS).* https://doi.org/10.1109/ISPASS.2015.7095817

[19] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2017. GPU Triggered Networking for Intra-Kernel Communications. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC).*

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Intl. Symp. on Microarchitecture (MICRO).* 469–480.

[21] Mellanox. 2017. Mellanox OFED GPUDirect RDMA. http://www.mellanox.com/page/products_dyn?product_family=116

[22] Mellanox. 2018. InfiniBand Performance. http://www.mellanox.com/page/performance_infiniband

[23] Nvidia. 2016. Fast Multi-GPU collectives with NCCL. https://devblogs.nvidia.com/parallelforall/fast-multi-gpu-collectives-nccl/

[24] Nvidia. 2017. GPU Applications. http://www.nvidia.com/object/gpu-applications-domain.html

[25] Nvidia. 2018. CUDA Toolkit 9.2. https://developer.nvidia.com/cuda-toolkit

[26] Nvidia. 2018. Nvidia DGX-2. https://www.nvidia.com/en-us/data-center/dgx-2/

[27] Lena Oden and Holger Froning. 2013. GGAS: Global GPU Address Spaces for Efficient Communication in Heterogeneous Clusters. In *Intl. Conf. on Cluster Computing (CLUSTER).* 1–8. https://doi.org/10.1109/cluster.2013.6702638

[28] Lena Oden, Holger Froning, and Franz-Joseph Pfreundt. 2014. Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU. In *Intl. Conf. on Parallel Distributed Processing Symposium Workshops (IPDPSW).* 976–983. https://doi.org/10.1109/ipdpsw.2014.111

[29] Lena Oden, Benjamin Klenk, and Holger Froning. 2014. Energy-Efficient Collective Reduce and Allreduce Operations on Distributed GPUs. In *Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid).* Institute of Electrical and Electronics Engineers (IEEE), 483–492. https://doi.org/10.1109/ccgrid.2014.21

[30] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-grain Task Aggregation and Coordination on GPUs. In *Intl. Symp. on Computer Architecture (ISCA).* 181–192. http://dl.acm.org/citation.cfm?id=2665671.2665701

[31] 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 *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*.

[32] S. Potluri, A. Goswami, D. Rossetti, C. J. Newburn, M. G. Venkata, and N. Imam. 2017. GPU-Centric Communication on NVIDIA GPU Clusters with InfiniBand: A Case Study with OpenSHMEM. In *Intl. Conf. on High Performance Computing (HiPC)*. 253–262. https://doi.org/10.1109/HiPC.2017.00037

[33] Sreeram Potluri, Nathan Luehr, and Nikolay Sakharnykh. 2016. Simplifying Multi-GPU Communication with NVSHMEM. http://on-demand-gtc.gputechconf.com/gtc-quicklink/7D7mU

[34] Davide Rossetti. 2015. GPUDirect Async. http://on-demand.gputechconf.com/gtc/2015/presentation/S5412-Davide-Rossetti.pdf

[35] Sandia National Laboratories. 2017. The Portals 4.1 Network Programming Interface. http://www.cs.sandia.gov/Portals/portals41.pdf

[36] Sandia National Laboratories. 2018. Sandia OpenSHMEM. https://github.com/Sandia-OpenSHMEM/SOS

[37] Jeff A. Stuart and John D. Owens. 2009. Message Passing on Data-parallel Architectures. In *Intl. Symp. on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/ipdps.2009.5161065

[38] TACC. 2015. Stampede Supercomputer User Guide. https://portal.tacc.utexas.edu/user-guides/stampede

[39] TOP500.org. 2018. Highlights - June 2018. https://www.top500.org/lists/2018/06/highlights/