GPUrdma: GPU-side library for high performance networking from GPU kernels

Feras Daoud
Technion - Israel Institute of Technology
ferasd@campus.technion.ac.il

Amir Watad
Technion - Israel Institute of Technology
amirw@tx.technion.ac.il

Mark Silberstein*
Technion - Israel Institute of Technology
mark@ee.technion.ac.il

ABSTRACT

We present *GPUrdma*, a GPU-side library for performing Remote Direct Memory Accesses (RDMA) across the network directly from GPU kernels. The library executes *no* code on CPU, directly accessing the Host Channel Adapter (HCA) Infiniband hardware for *both* control and data. Slow single-thread GPU performance and the intricacies of the GPU-to-network adapter interaction pose a significant challenge. We describe several design options and analyze their performance implications in detail.

We achieve 5μsec one-way communication latency and up to 50Gbit/sec transfer bandwidth for messages from 16KB and larger between K40c NVIDIA GPUs across the network. Moreover, GPUrdma outperforms the CPU RDMA for smaller packets ranging from 2 to 1024 bytes by factor of $4.5\times$ thanks to greater parallelism of transfer requests enabled by highly parallel GPU hardware.

We use *GPUrdma* to implement a subset of the global address space programming interface (GPI) for point-to-point asynchronous RDMA messaging. We demonstrate our preliminary results using two simple applications – ping-pong and a multi-matrix-vector product with constant matrix and multiple vectors – each running on two different machines connected by Infiniband. Our basic ping-pong implementation achieves 5% higher performance than the baseline using GPI-2. The improved ping-pong implementation with per-threadblock communication overlap enables further 20% improvement. The multi-matrix-vector product is up to $4.5\times$ faster thanks to higher throughput for small messages and the ability to keep the matrix in fast GPU shared memory while receiving new inputs.

GPUrdma prototype is not yet suitable for production systems due to hardware constraints in the current generation of NVIDIA GPUs which we discuss in detail. However, our results highlight the great potential of GPU-side native networking, and encourage further research toward scalable, high-performance, heterogeneous networking infrastructure.

*Corresponding author

ROSS '16, June 01 2016, Kyoto, Japan

DOI: http://dx.doi.org/10.1145/2931088.2931091

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; I.3.1 [**Hardware Architecture**]: Graphics processors

Keywords

Operating Systems Design, GPGPUs, Networking, accelerators

1. INTRODUCTION

GPUs have become an integral part of supercomputing systems, with large-scale deployments reaching thousands of GPUs installed across the compute nodes. Therefore, high-throughput low-latency inter-GPU communications are crucial for exploiting the full power of such multi-GPU systems.

The separate physical memory of discrete GPUs has traditionally been the main obstacle to achieving low latency and high throughput transfers between GPU kernels in different nodes. In older architectures this memory has not been accessible from peripheral network devices (NICs), requiring explicit buffer staging from the GPU to CPU memory and back (see Figure 1(a)) in order to transfer GPU memory buffers across the network. The introduction of GPUDirect RDMA technology [2] enables direct NIC access to GPU memory, allowing Remote Direct Memory Access (RDMA) from and to the GPU without an extra hop through the CPU memory (see Figure 1(b)). The specialized MVAPICH-2 MPI library leverages the GPUDirect RDMA mechanism to implement direct memory transfers across GPUs via standard MPI calls. Similarly, the GPI-2 communication library [1], which implements the Partitioned Global Address Space (PGAS) interface, uses GPUDirect RMDA to implement zero-copy RDMA across GPUs.

While these inter-GPU communication libraries achieve high performance and are widely used, they share one key limitation: data transfer calls must be invoked *by the CPU*, while ensuring that the GPU kernel that generates the data *terminates* before the transfer starts (or, symmetrically, that the kernel that reads the received content is started only after the transfer is complete). Such a *CPU-centric* GPU-as-co-processor design significantly constrains the design space and complicates development, for the following reasons:

1. **Bulk-synchronous design and explicit pipelining.** Applications are forced to be split into bulk-synchronous phases. Such a design makes overlapping computations and communications challenging. In particular, multiple buffers are required to implement double buffering, and computations must be broken down to work on small tiles that fit into smaller buffers.
2. **Complex synchronization.** Synchronizing between kernel execution and network operations is complicated since GPUs and NICs expose different synchronization mechanisms and

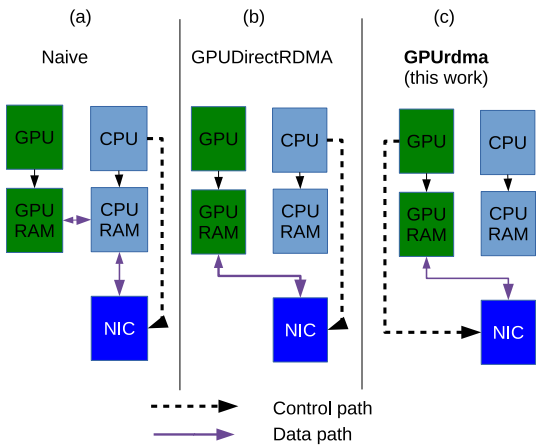

Figure 1: Evolution of the GPU-NIC interaction. (a) Prior to GPUDirect RDMA (memory staging). (b) With GPUDirect RDMA (transfer from GPU memory, CPU controls the NIC). (c) With GPUrdma (GPU controls the NIC, CPU not involved).

event interfaces, and even more challenging when executing multiple kernels and I/O calls in parallel to achieve pipelining.

3. **Kernel invocation overheads.** Kernel invocation might be relatively costly for short kernels. Furthermore, kernel termination and re-invocation make it impossible to store the kernel state in the fast scratchpad shared memory, which results in non-negligible performance loss, as we show in Section 7.
4. **Large buffers for batched transfers.** The messages to be transferred to other machines must be accumulated in internal GPU buffers during the kernel execution. This in turn increases the kernel memory consumption, or constrains the amount of data a kernel may process at once to keep the output within the memory boundaries.

Recent works introduce GPUfs [12, 13, 10, 11] and GPUnet [5] — GPU-side *native I/O libraries* for direct access to Operating System I/O services from GPU code. These libraries expose high level programming abstractions such as files and sockets to GPU kernels in order to provide the host file system and network services to GPUs via POSIX-like interfaces. These libraries enable a *GPU-centric* application design, in which I/O operations such as file access or network transfers are initiated from GPU code. They highlight the main programmability and performance benefits of GPU-native I/O services, showing that the GPU-centric design eliminates or greatly reduces the outlined above limitations inherent to the CPU-centric design.

This paper follows the GPU-centric design approach and presents **GPUrdma**: a novel GPU-side library for high-performance RDMA communications from GPU code. GPUrdma differs from previous works in that the CPU is *completely* bypassed, executing no code relevant to GPU communications. As a result, GPUrdma provides strong performance isolation of GPU communications from the CPU workloads.

Earlier attempts to build a GPU-side RDMA library showed unsatisfactory results [7], leading their authors to conclude that the GPU-native design is inferior to the traditional CPU-controlled I/O mechanism. In contrast, we show that the high-performance GPU-side RDMA is feasible, thanks to extensive optimizations in the API design, as well as advances in recent hardware. We achieve about 5 μsec latency and 50 Gbit/sec throughput for inter-GPU or GPU-to-CPU communications over the FDR Infiniband networking infrastructure.

We thoroughly analyze the design options and explain the main insights that enable high performance. In particular, we show that the slow performance of a single GPU thread dictates the use of *multiple* parallel threads to issue multiple RDMA requests in parallel. Interestingly, the throughput of the parallel GPU implementation *exceeds* that of CPU-controlled RDMA for smaller messages with similar configuration by up to $3.8\times$ for messages ranging from 2B to 1KB.

We use GPUrdma to prototype a subset of a GPU-side global partitioned address space programming interface (GPI) [1]. GPI is a popular communication infrastructure and runtime used in many large-scale supercomputing systems that implements a Partitioned Global Address Space (PGAS) abstraction. GPI implements a thin layer on top of an RDMA substrate, providing an efficient and convenient means for implementing scalable high-performance applications.

We implement a subset of essential GPI API calls on NVIDIA GPUs using GPUrdma, and evaluate the system using a ping-pong microbenchmark and a more complex matrix-vector product application. In the ping-pong microbenchmark the Master node sends 180MB from its CPU memory to the GPU memory in the Worker node, which immediately sends it back without processing. In the baseline implementation the CPU on the worker is notified when the message is received, and it then sends the same data back without any processing (and without GPU kernel invocation). Our first implementation does the same, but from GPU code using the GPU-side GPI: it receives the buffer in 30 threadblocks (each 1/30 of the total data), and then sends data back from each threadblock. The GPU-side GPI achieves throughput of 40.5Gbit/sec vs. 38Gbit/sec in the CPU version. The second, enhanced, implementation overlaps execution with communication by *independently* sending requests to each threadblock, effectively overlapping execution of different threadblocks. This implementation achieves throughput of 52Gbit/sec.

We then evaluate GPUrdma on an application that computes a product of a constant matrix by each of the short vectors that arrive in stream. We show that the GPU-side GPI implementation achieves about $4.5\times$ higher performance than the GPI-2 implementation, thanks to the ability to send small messages with high throughput and keep the matrix in shared memory while continuously receiving new inputs.

Despite the promising results we report here, the concept of GPU-side network communication libraries cannot yet be made fully available in production systems. This is due to the hardware limitation in existing NVIDIA GPUs which do not guarantee consistent update of GPU memory by the network card while the kernel is running. We discuss the implications of this issue in detail in Section 5, and show the example in Section 7. However, we believe that our current work clearly demonstrates the need for adding the missing functionality, and are encouraged to see emerging technologies [9] which might make it available soon.

2. RELATED WORK

The need for high performance GPU I/O services motivated several prior works relevant to this paper.

GPUnet [5]. Offers a native GPU networking layer that provides a socket abstraction and high-level networking APIs for GPU programs. It allows GPU threads to send and receive data using send/recv calls, and includes several advanced applications, such as a GPU

native server and GPU-native MapReduce, which demonstrate the benefits of GPU-side I/O services. However, GPUnet does not implement direct control of the HCA. Therefore, although the networking APIs are invoked from the kernel itself, all the calls to the HCA are performed by the CPU. In contrast, GPUs in our system interact with the HCA directly without CPU involvement.

GPUfs [12, 13]. Implements a file system abstraction for GPU programs to allow direct access to the host file system. GPUfs implements a buffer cache in GPU memory to achieve higher performance. However, GPUfs does not directly control the storage device and uses the CPU as a file server.

Infiniband-Verbs on GPU. [7] In this work the idea of GPU-side VERBs was investigated. However, the results showed significant performance degradation compared to the traditional CPU-originated networking, which led to the conclusion that the GPU-side VERBs are unlikely to work well.

In this paper we revise these conclusions and demonstrate the performance and programmability advantages of GPU-side RDMA networking.

GPUDirect RDMA. GPUDirect RDMA is a mechanism to expose GPU memory regions to be directly accessible on the PCIe bus via a Base Address Register (BAR) window. We discuss this mechanism in greater detail in Section 3.2. While this technology is an enabler for the GPUrdma project, it provides only the low-level infrastructure and is not intended to be used by GPU developers directly.

MVAPICH2 with GPUDirect RDMA. MVAPICH2 [3] is a high performance GPU-aware MPI library that uses GPUDirect RDMA to allow direct transfer of MPI buffers from GPU memory to the network. MVAPICH2 is a standard CPU-side library, unlike the GPU-side library described in this paper.

GPUDirect Async and NCCL. NVIDIA recently announced two new technologies for efficient inter-GPU communications. NCCL [14] is a CPU communication library for performing collective communications across multiple GPUs over NVLINK and PCIe. NCCL does not deal with cross-machine communication. GPUDirect Async [9] is a technology that offers efficient scheduling of networking with GPU computations by exposing network-related events to CUDA streams. These systems indicate the growing importance of high-performance GPU networking, however they do not yet support communications from GPU kernels.

3. BACKGROUND

3.1 GPU hardware and software

We briefly outline the basic concepts of the GPU architecture and programming model, focusing on discrete GPUs, using NVIDIA CUDA® terminology; more details about CUDA® and the GPU model can be found in [6].

Execution hierarchy. GPUs are parallel processors that run thousands of threads. Threads are grouped into *warps*, warps are grouped into threadblocks, and multiple threadblocks form a GPU kernel, which is invoked on the GPU by the CPU. A warp is a set of 32 threads executed in lockstep, i.e., at a given step all the threads in the warp execute the same instruction. Threads in a threadblock are invoked on the same core, called a streaming multiprocessor (SM), and can communicate and synchronize efficiently.

Memory hierarchy. Each thread has a set of dedicated private registers. Threads in the same threadblock share a fast, threadblock-private, on-die scratchpad called *shared* memory. The performance characteristics of shared memory are similar to those of an L1 cache

All threads in the kernel share *global* GPU memory. Global memory provides about an order of magnitude lower bandwidth than shared memory. Accesses to global memory are cached in a two-level hardware cache. The GPU can also access the CPU's memory over the PCIe bus, using pinned pages. Limitations of this access method include lack of cache coherency and atomic operations. It also incurs high latencies and is an order of magnitude slower than the GPU's internal global memory.

Inter-thread communication. Threads in the kernel may communicate via global memory. Threads in the same threadblock may also communicate via shared memory, and synchronize by using efficient hardware barriers.

3.2 GPUDirect RDMA and peer-to-peer DMA in PCIe

GPUDirect RDMA is a technology introduced by NVIDIA in Kepler-class GPUs and CUDA 5.0 that enables a direct path for data exchange between the GPU and a third-party peer device using standard features of PCI Express. The technology makes it possible to configure a desired GPU physical memory segment to be accessible via memory-mapped I/O operations through the PCIe bus. It provides a number of low-level functions that perform GPU memory address translations and mappings.

GPUDirect RDMA relies on the standard PCIe capabilities to perform peer-to-peer DMA across PCIe devices without CPU involvement. For example, one GPU may read from the memory of another GPU over PCIe, as long as the memory-mapped I/O region exposing the memory of the first is mapped into the address space of the second. These capabilities, combined with the GPUDirect RDMA API for GPU memory translations and mappings, enable any PCIe device to access GPU memory in the same way it accesses the CPU memory.

3.3 Infiniband and HCA

We briefly describe the relevant details of the Infiniband software and hardware.

Infiniband is a high-throughput low-latency network standard widely used in high performance computing, Web 2.0, storage systems and cloud services. The Infiniband standard defines the physical, data link, network and transport protocols, and requires compliant end node devices (called host channel adapters, or HCAs) to implement these layers in the HCA hardware, obviating the need for a network stack in the operating system, and freeing the data path from OS involvement.

Infiniband's transport layer supports Remote Direct Memory Access (RDMA) operations, which allow a machine to access another machine's memory for read, write and atomic operations without any involvement of the target machine's CPU.

In order to use the services of the Infiniband transport layer, a process asks the HCA to create a Queue Pair (QP) — a control object consisting of a pair of Work Queues: a Send Work Queue, which allows the process to be the initiator of transport operations, and a Receive Work Queue, which allows the process to be the target of a transport layer operation. A QP is identified by a QP number (analogous to a TCP port number).

A Work Queue is a pinned buffer in memory, accessible to the CPU process for write and to the HCA for read. A process instructs the HCA to execute a task by posting a Work Queue Element (WQE, also called a Work Request) describing the send task. The Receive Work Queue is of little relevance to RDMA operations, which are the focus of this work.

Two processes create a transport connection by instructing the HCA to pair their local QPs. Once the QPs are paired, the pro-

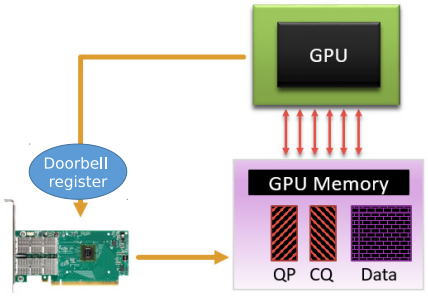

Figure 2: Host channel adapter (HCA) control structures (QP,CQ) and data buffers in GPU memory. The HCA doorbell register is mapped into GPU address space.

cesses can communicate. The HCA executes a Work Request asynchronously, and delivers a Completion Notification to the initiating process with the status of the operation. A Completion Notification (called Completion Queue Element – CQE) is delivered to a Completion Queue (CQ) associated with the initiating QP, which is usually polled by the communicating process.

In order for a process to specify a memory buffer for RDMA operations, the buffer must be mapped into the HCA's virtual memory system via an operation called Memory Registration. A user process performs Infiniband transport operations using VERBs, which is the transport layer API of the Infiniband protocol

Controlling the HCA from the CPU. Data transfer from the HCA is triggered by writing into a respective *doorbell* register in the HCA. Each QP has its own doorbell register. The write to the doorbell notifies the HCA to handle the next Work Request by advancing to the next WQE in the QP.

4. BASIC SUPPORT FOR GPUrdma

We briefly describe the low-level infrastructure for RDMA support in GPUs.

HCA control and data in GPU memory. We create QP and CQ control structures and associated data buffers in GPU memory using standard CUDA memory allocation API, and use peer DMA API [2] to expose them to the HCA. We modify the HCA initialization routines in the original network libraries [4] to make it possible to initialize QP/CQ with pointers to GPU memory.

GPU driver modifications for mapping doorbell registers. The HCA doorbell registers are mapped into CPU address space, and can be accessed as regular memory by leveraging the memory mapped input/output mechanism (MMIO). They must be mapped into GPU address space in order to be accessible from the GPU. However, the `cudaHostRegister()` call, which is used to map CPU physical memory into the GPU address space, does not support MMIO pages. This is because the current NVIDIA GPU driver requires CPU virtual memory pages to be backed by physical pages in order to map them into the GPU address space.

To overcome this limitation, we modify the open-source part of the NVIDIA closed-source driver such that the MMIO pages are determined at runtime and mapped appropriately. In addition, we obtained custom-modified Mellanox HCA firmware, which guarantees that the HCA PCIe BAR (Base Address Register) will be allocated in the address range below 42 bits. This turns out to be the maximum PCIe bus address range supported by modern NVIDIA GPUs.

Stage	Latency (μsec)	Accumulated
WQE writing	1.6	1.6
Doorbell access	1.5	3.1
CQ polling	3.1	6.2
Measurement overhead	-0.4	5.8

Table 1: Latency breakdown for the single-thread implementation

Limits on GPU network buffer size. NVIDIA GPUs prior to K40 constrain the amount of memory which can be concurrently accessed over the PCIe bus to about 220MB. Starting from NVIDIA K40 (excluding K40C), this limit has been raised to 16GB, implying that the whole GPU physical memory can be accessed via RDMA operations.

GPU in-kernel software support. Data sent from/to GPU memory buffers by GPU code must be consistently stored in GPU memory so that it can be read by the HCA over the PCIe. Two important precautions must therefore be taken. First, the memory buffers exposed to HCA must be marked `volatile`, to prevent compiler optimizations, such as keeping their contents in registers. Second, the set of threads that update the buffer must complete the write operation by issuing a `__threadfence_system()` CUDA call to guarantee that the writes are observable in the right order by the HCA. This call, however, is not necessary for the receive path.

PCIe switch. As has been noted in prior work [5], Intel PCIe switches suffer from severe read/write performance asymmetry when two PCIe peers communicate directly. In the context of the GPU-HCA communications, writing into GPU memory from the HCA (receive path) achieves close to maximum throughput (50Gbit/s), which is about twice as high as that obtained when the HCA reads from GPU memory (send path). In our work, therefore, we use PLX PEX8747 PCIe switches, which, fortunately, do not suffer from this problem.

4.1 Single GPU thread performance

We implement the basic support for GPU-side RDMA using a NVIDIA K40c GPU with 15 streaming multiprocessors (SMs) and a Mellanox Connect-IB HCA supporting FDR (56Gbit/s). In our experiments we invoke a GPU kernel with a single thread that issues 1024 RDMA write requests. This number of requests fits into the work queue of a single QP.

We run each experiment 3 times and report the average. We measure the transfer time from the point we start the writing of the first Work Queue Entry (WQE) until the point we receive a successful Completion Notification for the last sent message. We measure the time using the `clock64()` GPU intrinsic.

Naive version. The GPU thread performs the following operations for every message it sends: (1) it creates a Work Queue Entry (WQE) in the QP, with information about the source and the destination addresses and data size; (2) it updates the QP's doorbell record with the index of the last posted WQE; (3) it writes to the QP's doorbell register, triggering the HCA to start execution of the work request; (4) it waits for the completion of the request by polling the respective CQ for a Completion Queue Entry (CQE).

Figure 3 shows the bandwidth and Table 1 shows the latency breakdown for a GPU kernel with a single thread performing an RDMA write to the CPU memory of a remote host. RDMA read performance is similar and not shown.

We observe that the maximum bandwidth is about 43Gbit/sec, and it is achieved for very large (1 MB) messages. Smaller messages of up to 32KB achieve only 15Gbit/sec, less than one-third of the maximum network bandwidth. We also observe an unex-

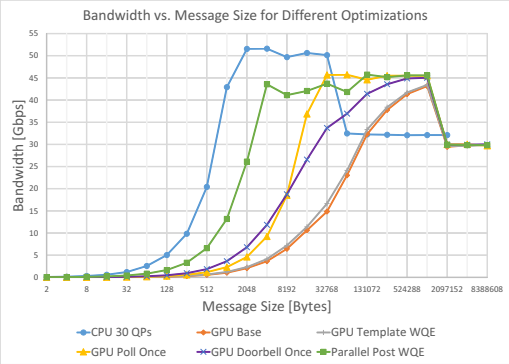

Figure 3: Single GPU-thread network bandwidth for different message sizes: naive version, WQE reuse, pipelining, reduced doorbell

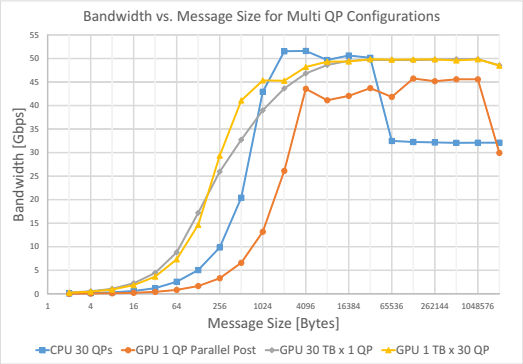

Figure 4: Bandwidth optimization via GPU and HCA parallelism.

pected performance drop for messages above 2MB. This degradation is most likely due to the PCIe switch limitations for peer-to-peer communications, since we observed no such degradation with Intel PCIe switches.

These results are clearly unsatisfactory, and warrant several optimizations, which we describe next.

WQE reuse. A WQE is a 64-byte structure that describes the request to the HCA, including, the request type, size, and local and remote memory addresses. Many of the fields in WQE remain the same across multiple requests. Pre-initializing all the WQEs to minimize the number of writes helps reduce latency and improve single-thread communication throughput. Figure 3 shows that this optimization does not improve performance for larger messages, but it does help reduce the latency of each WQE by about 0.76 μsec, yielding a total latency of 5.1 μ sec per message.

Request pipelining. The original implementation polls the completion queue after issuing each work request, waiting for each job to be received and acknowledged by the remote HCA before posting the next job. This delay is clearly unnecessary if one sends a large data stream, as we do in this bandwidth experiment. Therefore, a throughput-optimized version requests and waits for completion only when the QP's work queue is full; in this way, the queue can be reused for another batch. We show that this optimization provides a significant performance boost, making it possible to reach the maximum bandwidth of 45Gbit/sec for 32KB messages.

Reduced writes to doorbell. Doorbell registers reside across the PCIe bus in the HCA. Writing to these registers to signal the HCA to send the message is a costly operation that also incurs the overhead of the memory fences which precede it. Our naive implementation writes to the doorbell register for each message separately. Our optimization reduces the number of writes to the doorbell to a bare minimum, writing into it only after the QP is full. Figure 3 shows the results of adding this optimization to the pipelined version. We see that this optimization indeed works well for small messages. For larger messages, however, the HCA remains underutilized: it sends the data faster than the GPU produces the requests.

Conclusions. While the optimizations enable improved performance, even the optimized implementation achieves about 10% lower bandwidth than the CPU-controlled RDMA transfers.

4.2 Exploiting GPU parallelism for efficient networking

In this section we take a complementary approach to throughput optimization. We increase the number of QPs to leverage multiple GPU threads to concurrently create multiple transfer requests. It is important to note that the use of multiple QPs is a well-known optimization for CPU transfer performance. As we show, however, this optimization has a dramatically positive effect on GPU performance.

In all the following experiments we use the best pipelined single-thread configuration, but replicate it across multiple threads.

QP per warp. We run one threadblock with 30 warps. Each warp has its own QP, and all QPs are associated with a single CQ. This is convenient, since only one thread is sufficient to poll for completion. Figure 4 shows that we reach a transfer throughput of 50 Gbit/sec with 16K messages, more than 10% higher than the GPU single QP case. While this configuration performs well, its memory consumption is rather high, with 128MB for all the QPs. Using the GPU with full occupancy while assigning a QP to each warp will consume a large amount of the GPU memory.

QP per threadblock. We invoke 30 threadblocks with 1 warp each, and create one QP per threadblock. All QPs are connected to the same CQ. The threads in the threadblock perform an RDMA call in a collaborative manner. All the threads first create their own WQEs but store each in its own offset in the QP. The last thread in the threadblock writes into the doorbell. The threads poll collaboratively for a CQE containing the QP number assigned to their threadblock: each thread polls in a different offset in the CQ buffer, and the threads use CUDA® warp vote intrinsics to determine whether one of them has found the CQE. We achieve a bandwidth comparable to the QP-per-warp case, reaching 50 Gbit/sec with 16KB messages, with better scalability with respect to GPU memory consumption.

Conclusions. Packet ordering rules of the RDMA transport protocol force the HCA to serialize the processing of packets that belong to the same QP. Increasing the number of QPs allows the HCA to process different packets from different QPs in parallel.
We exploit this ability, as well as the ability to write multiple WQEs to the work queue of the same QP in parallel. This results in faster job submission, while also allowing faster processing of the network traffic by the HCA.

4.3 Understanding optimal QP/CQ locations

The intricacies of PCIe peer-to-peer transfer performance necessitate additional analysis for the optimal location of the QP/CQ structures. Specifically, our basic design moves both QP and CQ to the GPU, and forces the HCA to access them from the GPU memory. However, since such an access results in peer-to-peer transfers,

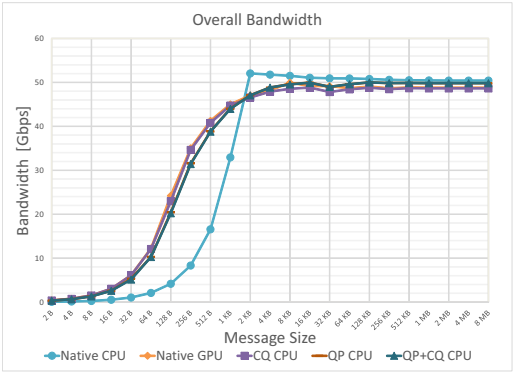

Figure 5: Transfer bandwidth as a function of different locations of HCA control structures

	QP-CPU	QP-GPU
CQ-CPU	8.6	6.2
CQ-GPU	6.8	4.8

Table 2: Transfer latency (in μsec) as a function of different locations of HCA control structures.

performance might be slower than if the HCA accessed these data structures from CPU memory.

Therefore, we evaluate the throughput and latency of all four possible designs, in which the QP and CQ buffers are located either on the GPU or on the CPU. Figure 5 shows that the configuration where the QP is located in CPU memory and the CQ is in the GPU memory achieves slightly better bandwidth than the case where both the QP and CQ are located in GPU memory. This is consistent with the observation that HCA reads from the GPU memory are slower than HCA reads from the CPU memory.

Conclusions. The best location for QP and CQ buffers in native GPU networking is the GPU memory. Moving the QP buffer to the host memory increases the latency of posting new jobs and ringing the doorbell: in this case, for each new job the GPU writes to the host memory and then to the doorbell across the PCIe. For the CQ buffer, polling a completion from a buffer which resides in host memory increases the overhead of the poll command.

5. LIMITATIONS

Our work is pushing the current GPU hardware to extremes, and requires capabilities which are not yet officially supported. Specifically, a network card, or any other third party I/O device, cannot guarantee consistent updates of GPU memory via peer-to-peer DMA while a GPU kernel is running. The implication of this constraint is that causally dependent RDMA writes into GPU memory (e.g., one for sending a data buffer and another for the consequent receiver notification) might be observed by a GPU kernel *in reverse order*. In practice, this behavior is indeed observed under very high data transfer load 7, and eventually all the data that has been written into GPU memory becomes available to the kernel. However this limitation obviously prevents current broader adoption of the GPU-side networking concepts in production.

However, we believe that early experimental uses of our work might be already advantageous today, and will likely become even more so in the future. First, RDMA calls which *read* from GPU memory are not affected by the inconsistency problem. Thus, one can use GPUrdma for sending strided or irregularly organized data directly from the kernel. Further, certain applications like asynchronous Stochastic Gradual Decent (SGD) [8] might benefit from low latency and high bandwidth of GPU memory updates via GPUrdma even without full data integrity guarantee. Finally, we believe that the recently announced GPUDirectAsync [9] technology will help solving the consistency issue.

```
__device__ stencil_compute(float* area)
{
  //wait for boundaries
   gpu_gaspi_notify_waitsome(…);
  //data parallel code per thread
    area[threadId.Idx]=1/2*(area[threadIdx.x-1]+
                            area[threadIdx.x+1]);
   // wait for all to complete
  __syncthreads();
  //collaborative send from buf
  gpu_gaspi_write_notify(…);
}
```

Figure 6: A code sketch for using GPU-side GPI calls for simple stencil computations.

6. GPI PROTOTYPE

Background. The GPI library provides a variety of naming, point-to-point and collective messaging services. However, most programs use a handful of API calls that implement asynchronous one-sided communication with a remote node (read/write), and application level synchronization via GPI's *notification* mechanism. Specifically, each communicating process declares a local memory region where it stores all the notification flags. A remote sender may update the respective notification flag in the receiver after sending a message, so that the receiver can check the flag to see that the message is available.

GPU-side GPI calls. We use GPUrdma to implement the four commonly used GPI calls: `gpu_gaspi_write`, which implements RDMA write, `gpu_gaspi_notify`, which sends remote notifications, `gpu_gaspi_notify_waitsome`, which checks notifications locally, and `gpu_gaspi_write_notify`, which combines write and notify in one call.

Threadblock-wide API calls. In our prototype we implement *coalesced* threadblock-wide function calls, which are invoked by all the threads in a threadblock at the same point in the program, with the same arguments. This design pattern has been successfully used in other GPU-side I/O libraries [12, 5]. The main intuition behind the threadblock-wide calls is that they match the common GPU kernel design pattern whereby each threadblock represents an independent task collaboratively performed by all the threads in a threadblock. Therefore, threadblock-wide I/O calls naturally match the synchronization points at the boundaries of each task, and serve for data exchange with other tasks.

We illustrate the idea in Figure 6, which shows a sketch of a function for computing a 1D stencil in one threadblock. The boundaries are received and sent from and to other nodes before and after the computation.

Future versions of the GPU-side calls, however, may need to implement warp-level API for potentially higher performance, but will also incur the cost of more complicated programming.

Implementing GPI on GPUrdma. We use a threadblock per QP/CQ design described in the previous section to prevent contention among the threadblocks. In accordance with GPI terminology, each threadblock uses its own GPI queue. This design might result in high memory consumption due to too many QP/CQ data structures, each usually occupying 128KB. We plan to revisit this design in the future.

All the calls are implemented using the standard RDMA VERB API. In addition, we leverage an optimized version of inline RDMA write to reduce the latency of `gpu_gaspi_notify`.

6.1 Discussion

GPU-side I/O and persistent kernels. The most common GPU programming practices involve invocation of a multitude of short-lived threadblocks in each kernel, each executing a small chunk of computations, after which it terminates. Such a design matches the GPU-as-co-processor model well, because the kernels cannot obtain any new input or output the results other than at the kernel invocation boundaries.

GPU-side I/O libraries enable a more natural application design in which computations and I/O are interleaved, and the I/O calls are performed directly from GPU code. As a result, rather than terminate the kernel for performing I/O, we can execute long running threadblocks throughout the entire lifetime of the application. Hence, the kernel is invoked with just as many threadblocks as can be concurrently executed by GPU hardware — the approach commonly referred as *persistent* kernels.

Persistent kernels provide many advantages, saving kernel invocation overhead and simplifying application design. One particular benefit we explore in this paper (see Section 7) is the ability to keep the entire threadblock state in fast, on-die shared memory. This memory is reset every time a threadblock terminates; therefore persistent kernels are the only way to leverage shared memory for storing the state throughout the execution.

One downside of persistent kernels is the lack of an efficient global barrier across its threadblocks, which otherwise is implemented by terminating a kernel and restarting it again. Further, using the GPU-provided 3D index space across threadblocks is sometimes more convenient than implementing one manually.

Our current prototype requires the persistent kernel design in order to work. We, however, are working on extending it to support more traditional GPU programming practices as well.

Scalability challenges. Scaling the runtime system to support hundreds and thousands of communicating GPUs poses a challenge. This is because each threadblock must have one separate QP/CQ for each peer threadblock or CPU it communicates with, which results in space requirements quadratic to the number of nodes in the system. This problem exists in the original GPI as well but is exacerbated in the case of GPUs because the number of QP/CQ structures is increased by a factor equal to the number of threadblocks.

One possible solution is to set a pool of QP/CQ structures reused across multiple connections. This idea has been employed in large-scale CPU systems; however, it becomes truly necessary only for applications running on thousands of nodes. In the case of GPUs it must be used for significantly smaller environments.

Another design we consider is the use of 1-to-N communication primitives enabled via Mellanox's Dynamically Connected QPs.

Register pressure. Register pressure is a known problem with many GPU workloads, and the use of GPU-side networking library potentially increases the number of hardware registers required for a GPU kernel. We, however, believe that the register pressure is gradually becoming less critical, since the register file is growing as GPUs are evolving. For example, in K80 NVIDIA GPU the number of registers is doubled compared to the previous K40.

7. GPI EVALUATION

In this section we implement several benchmark applications to evaluate the performance of the GPU-side GPI prototype. We design the benchmarks to obtain the lower and upper bounds on the performance of the GPU-side I/O operations. We use GPU-2 v1.1.1 for baseline implementations.

Latency Ping-Pong. We seek to measure application-level latency of GPI notification across the network. We implement a GPU kernel with a single threadblock that issues `gpu_gaspi_notify_waitsome()` to wait for the notification from a remote node, and then `gpu_gaspi_notify()` to notify the sender back. The counterpart is implemented in the CPU and performs the same operations but in reverse order. We measure the latency on the CPU.

The result is a roundtrip of 10μsec, or 5μsec one way, which is consistent with our earlier results in Section 4. GPI API adds no noticeable latency overhead.

Throughput Ping-Pong. The throughput ping-pong application transfers 180MB from CPU memory to GPU memory and back without any processing.

The baseline version is implemented using the GPI-2 framework, which uses GPUDirect RDMA to send network buffers directly from the GPU memory. It does not invoke any GPU kernel, and therefore encounters no kernel invocation overhead, which is the most optimistic scenario for the original CPU-controlled I/O design.

In both the baseline and the GPU-side GPI cases, we run a master process on the CPU. This process opens 30 GPI queues, registers a 180MB memory segment in CPU memory, and initiates the transfer from that segment to the worker's GPU memory directly via the `gaspi_write_notify` call. The master sends all the data at once using one of the queues, and notifies the worker. Then the master waits for notification from the worker that the return message has been sent.

The worker process runs on the CPU for the baseline version and on the GPU for the GPU-side GPI. A CPU worker allocates the receive buffer in GPU memory, waits for the notification from the master that the data has arrived, and then sends the same buffer back over 30 GPI queues by splitting the buffer contents evenly among all the queues. A GPU worker invokes 30 threadblocks, where each first waits for a notification (one for all the threadblocks) from the master, then selects a 1/30 slice of the received data and sends it back using its own queue.

The basic GPU-side GPI implementation results in better throughput than the CPU GPI-2 baseline, with peak throughput of 40.5Gbit/sec versus 38Gbit/sec for GPI-2.

We note that these results are particularly encouraging because this kind of application demonstrates the worst-case performance for GPUrdma and GPU-side GPI. Indeed, it provides no opportunity for computation and communication overlap. Yet, the performance is higher because there is no kernel invocation overhead which is among the important benefits of GPU-side communications.

Throughput Ping-Pong with block overlapping. We implement an enhanced version of the throughput Ping-Pong. Unlike the previous bulk synchronous implementation which waits for notifications from all the threadblocks before sending the next chunk of data, this version triggers send individually to each threadblock once it is ready, without waiting for other threadblocks to complete. As a result, the GPU is fully utilized.

At this point, unfortunately, we face the memory consistency update problem that we explain in Section 5 in detail. Our solution is application specific. To measure the throughput in this experiment we check the data correctness in GPU memory before sending it back to the master. Specifically, since the expected contents of the receive buffer are known, the threads check the buffer in data-

Batch size (KB)	60	120	240	480	960	1920
GPI	4.9	9.2	1.1	26.5	37.9	46.3
GPUrdma	22.4	35.9	48.1	55.6	57.7	60

Table 3: The system throughput in millions of vector multiplications per second as a function of batch size

parallel manner and keep waiting in a loop until the data arrives in full. Another alternative would be to use parallel CRC32, as has been done in GPUnet [5].

Despite these additional checks, this implementation provides higher throughput than any previous implementation, with the peak throughput of 52Gbit/sec.

Multi-matrix-vector product. The multi-matrix-vector product application compares the throughput of the GPI-2 and GPU-side GPI library when running GPU applications using messages of different sizes. The application multiplies each vector in a stream of short 32x1 vectors by a fixed 32x32 matrix of floats, which fits into shared memory of each threadblock.

The master node runs on the CPU. It generates a batch of vectors and sends the batch to the worker together with the GPI notification. The master waits for the notification from the worker that the results have been sent back, and then transfers the next batch.

We compare two implementations of the worker. The baseline GPI-2 implementation waits for the notification from the master in CPU code. Once the notification has been received, the worker starts a kernel with 30 blocks, each having 1024 threads. The kernel copies the matrix from global memory to shared memory, calculates the multiplications on its batch and terminates. After the GPU kernel terminates, the worker sends the data back to the master and notifies it. These operations are performed from the CPU. When the new batch is received, a new kernel is started.

The worker that uses the GPU-side GPI also invokes the kernel with 30 blocks and 1024 threads per threadblock, but does that only once. In the kernel it uses GPU-side GPI calls to receive and send the input vectors. It copies the matrix to multiply the vectors to the shared memory of each threadblock *only once*, and never reads it again. For each batch the kernel calculates the results, and sends them back to the master without requiring termination of the kernel to perform communications, and therefore eliminating the need to copy the data from global to shared memory for every kernel invocation.

We measured the throughput of these two application with 75M vectors, with total input of 2.4GB, and different batch sizes. The results are as shown in Table 3

We observe that our GPU-side GPI implementation is significantly faster for smaller batches than the one that uses GPI-2, outperforming it by up to $4.5\times$. There are two main sources of overhead in the GPI-2 implementation that are not present in the GPU-side GPI-based design: (a) the matrix is repeatedly copied from the global memory to the shared memory, as opposed to only once in the GPU-side GPI design (about 20% of the performance gain) (b) lower performance when transferring shorter messages.

8. CONCLUSIONS

GPU-side native networking is a promising direction for high performance computing systems. Efficient inter-GPU communication is critically important to achieve scaling in multi-GPU applications.

In this paper we describe GPUrdma – an RDMA library which implements high-throughput, low-latency communications for GPU kernels. GPUrdma is the first library to completely bypass the CPU and directly access the network adapter for both data and control. We leverage GPUrdma to prototype several common global partitioned address space APIs from the GPI library, and implemented two applications to evaluate its performance.

We show that our GPU-side GPI prototype can achieve higher application performance than the traditional CPU-side GPI for these workloads, also exposing several important directions for future research, such as better scaling to large numbers of GPUs.

Acknowledgements

Mark Silberstein is supported by the Israel Science Foundation (grant No. 1138/14), the National Science Foundation (grant No. CCF-1333594), as well as the Israeli Ministry of Science and the Israeli Ministry of Economics via HiPer consortium.

9. REFERENCES

[1] *GPI-2*. http://www.gpi-site.com/gpi2/.

[2] *GPUDirectRDMA technology*. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[3] MVAPICH2: High performance MPI over InfiniBand, iWARP and RoCE. http://mvapich.cse.ohio-state.edu.

[4] *OpenFabrics Enterprise Distribution*. https://www.openfabrics.org/index.php/openfabrics-software.html.

[5] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, 2014.

[6] David B Kirk and Wen-mei W Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Newnes, 2012.

[7] Lena Oden and Holger Fröning. Infiniband verbs on GPU: a case study of controlling an InfiniBand network device from the GPU. *International Journal of High Performance Computing Applications*, page 8, 2015.

[8] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[9] Davide Rossetti. GPUDirect async: integrating the GPU with a network interface.

[10] Sagi Shahar, Shai Bergman, and Mark Silberstein. ActivePointers: A Case For Software Translation on GPUs . In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 2016.

[11] Sagi Shahar and Mark Silberstein. Supporting Data-Driven I/O on GPUs using GPUfs. In *ACM International Conference on Systems and Storage (SYSTOR)*. ACM, 2016.

[12] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013.

[13] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. *ACM Transactions on Computer Systems (TOCS)*, 2014.

[14] https://github.com/NVIDIA/nccl. NCCL: optimized primitives for collective multi-GPU communication.