Analyzing Put/Get APIs for Thread-collaborative Processors

Benjamin Klenk
University of Heidelberg
Institute of Computer Engineering
Heidelberg, Germany
klenk@uni-hd.de

Lena Oden
Fraunhofer Institute for Industrial Mathematics
Competence Center High Performance Computing
Kaiserslautern, Germany
oden@fhg.itwm.de

Holger Fröning
University of Heidelberg
Institute of Computer Engineering
Heidelberg, Germany
froening@uni-hd.de

***Abstract*—In High-Performance Computing (HPC), GPU-based accelerators are pervasive for two reasons: first, GPUs provide a much higher raw computational power than traditional CPUs. Second, power consumption increases sub-linearly with the performance increase, making GPUs much more energy-efficient in terms of GFLOPS/Watt than CPUs. Although these advantages are limited to a selected set of workloads, most HPC applications can benefit a lot from GPUs. The top 11 entries of the current Green500 list (November 2013) are all GPU-accelerated systems, which supports the previous statements.**

For system architects the use of GPUs is challenging though, as their architecture is based on thread-collaborative execution and differs significantly from CPUs, which are mainly optimized for single-thread performance. The interfaces to other devices in a system, in particular the network device, are still solely optimized for CPUs. This makes GPU-controlled IO a challenge, although it is desirable for savings in terms of energy and time. This is especially true for network devices, which are a key component in HPC systems.

In previous work we have shown that GPUs can directly source and sink network traffic for Infiniband devices without any involvement of the host CPUs, but this approach does not provide any performance benefits. Here we explore another API for Put/Get operations that can overcome some limitations. In particular, we provide a detailed reasoning about the issues that prevent performance advantages when directly controlling IO from the GPU domain.

I. Introduction

In compute-intensive domains like High-Performance Computing, GPUs are pervasively used to accelerate tasks by offloading them from the CPU. This trend has been boosted by the introduction of domain-specific languages like CUDA or OpenCL, which allow programming GPUs without dealing with aspects related to graphics processing like textures or shaders. Beside computational power, GPUs also improve the performance-to-watt ratio significantly. For instance, while an Intel Xeon E5 achieves a peak of 1.44 GFLOPS/W (with 8 cores at 3.4GHz), a GPU can yield up to 14.08 GFLOPS/W for single precision computations. A quick look at the current Green500 list (Nov. 2013) [1], which lists the 500 most energy-efficient systems based on their *High Performance Linpack (HPL)* performance and sustained power consumption, reveals that the top 11 entries are all based on GPU accelerators. Although it has to be noted that GPUs only excel in performance and thus energy-efficiency for certain workload characteristics, in HPC they have proven to be highly effective.

Many of the GPU's advantages in terms of performance and power-efficiency come from their architecture, which differs significantly from traditional processors like CPUs. Most important, they rely on a thread-collaborative execution model, in which multiple thousands of threads cooperatively perform computations and memory accesses. This in combination with the warp-based scheduling make their execution model highly different to the one of a CPU. Also, GPUs only excel in performance for in-core computations, while in particular PCIe data movements can severely hurt performance. Often this fact is amplified because GPU memory is scarce and currently not exceeding 12GB, while CPUs can accommodate multiple TBs of memory, at least for enterprise-class systems.

Although GPUs come with plenty of advantages, including performance and power efficiency, their applicability in scalable systems is still challenging. Reasons include aspects like programming, productivity and porting of legacy codes, but also interactions with other system components. In this work we focus on the latter, which has up to now received rather little interest. Most IO interfaces, including networking and storage, are based on *PIO (Programmed IO)* commands and *DMA (Direct Memory Access)* descriptors, which are optimized for a single thread and actually do not take issues regarding control flow limitations into account. Thus, APIs are still solely optimized for CPUs, resulting in huge performance penalties when controlled from the GPU. GPU-controlled IO is desirable though, due to reasons that include time savings, energy savings and a reduced complexity to address the tremendous impact of hybrid programming models.

Related to issues with IO interactions, we see a huge need for GPU communication libraries. However, without advantages in terms of energy and time the future of such an approach is pretty bleak. Thus, we focus in this work on three parts:

1) An analysis of two put/get APIs from a GPU's perspective (IB and EXTOLL), in particular regarding limitations and optimization opportunities.
2) Implementing these APIs for GPU-controlled data movements.
3) A quantitative performance assessment in terms of band-

width, latency and sustained operations per second.
4) A detailed analysis using GPU performance counters to identify limitations and to highlight the most severe issues of current API designs.

We find these insights essential for future GPU libraries that abstract interactions with other IO devices, including in particular networking and storage.

The remainder of this paper is structured as follows: Section II will provide background knowledge for this work. This is followed by section III where the EXTOLL put/get API will be described. The same applies for the Infiniband API in section IV. In section V we analyze our results followed by a discussion in section VI. Related work will be shown in section VII while the last section concludes.

II. Background

In this section we provide background knowledge about GPUs and communication semantics as it will be helpful in the subsequent sections.

A. GPU Computing / CUDA

GPUs are highly parallel devices, which are composed of so-called *Symmetric Multiprocessors (SMs or SMXs since the Kepler architecture)* implementing plenty of lightweight cores. Due to plenty of cores, several thousands of threads can and should be executed in parallel. GPUs are optimized for highly parallel execution, while the single thread performance of a GPU is very low. Threads are grouped in blocks forming grids respectively kernels. Each SM implements up to four schedulers that schedule groups of 32 threads, referred as warps, at the same time. To achieve best utilization the programmer has to ensure that threads within one warp execute the same instruction, otherwise branch divergence occurs and reduces performance.

Unlike CPUs, caches are not used to minimize latency of memory accesses, but rather reduce traffic of the memory system allowing for smaller cache structures. GPUs launch plenty of threads to hide memory accesses by using a scoreboard that administrates warps that are either ready, performing arithmetic operations or waiting on memory responses. Also, the memory bandwidth is about an order of magnitude higher than for CPU memory. However, the highest bandwidth is only achieved if accesses are coalesced meaning that threads operate on consecutive addresses. As device memory is pretty far away from cores and registers, each SM also contains a fast scratchpad memory, referred as shared memory, which is explicitly managed by the programmer.

In order to allow programmers who are not familiar with graphics programming to use GPUs for general purpose tasks, NVIDA introduced a computing platform and runtime system called CUDA [2] in 2007. Applications are split into host and device code, which is compiled from a virtual instruction set (PTX) into machine code during run time. Instructions are grouped into kernels and explicit memory transfers between host and device memory. Recently, NVIDIA's CUDA 6.0 introduced Unified-Memory, removing the need of calling these explicit copy operations. This is handled by the runtime when a CUDA kernel is launched. Other approaches are OpenCL [3] or the directive based approach OpenACC [4], but for this work we rely on CUDA. Note that insights can also be applied to other languages.

B. Put/Get semantics

The common way for distributed multi-GPU programming is a hybrid programming model, where the GPU is used for computation and the CPU manages the communication.

Usually, this communication is based on message passing like it is provided by the *Message Passing Interface (MPI)*. This two-sided communication requires both the sender and receiver to call explicit send respectively receive functions. This normally adds a lot of overhead to the communication, due to tag matching or data buffering [5].

put/get communication, also known as one-sided communication, only needs the origin to issue a data transfer. This helps to reduce the communication overhead to a minimum. Using *Remote Memory Access (RMA)* capable hardware, the data transfer can completely be offloaded to the *Network Interface Controller (NIC)*. This allows overlapping of communication and computation.

A simple put/get API that supports overlapping of communication and computation requires two basic functions: one for initiating a data transfer and one to retrieve information about the current status of the data transfer.

Initiating a communication normally includes creating and posting of *Work Requests (WRs)*. Such a WR contains all information that is required for the communication, such as addresses and the payload size. The NIC's DMA engine then starts copying the data according to this information. In addition, the NIC may provide information about whether a data transfer has been started, or is locally completed, or if data has been received. This information can be used by the API to retrieve information about the communication status. In the simplest case the API is used to ensure, that the communication is locally completed.

To allow direct put/get communication on distributed GPUs, NVIDIA introduced GPUDirect RDMA which enables a third-party PCIe device to access GPU memory over PCIe BAR regions. For instance, a NIC can directly read and write into GPU memory without interference of the CPU.

Usually put/get communication is entirely handled by the CPU, even if the data is read from or written to GPU memory requiring costly context switching between the CPU and GPU domain. Examples include openShmem [6], UPC [7] or GPI2 [8].

We extend existing put/get APIs for Infiniband and EXTOLL in order to make the NIC directly accessible from the GPU avoiding these context switches. More detail on that will be given in subsequent sections.

III. EXTOLL implementation

This section will focus on the architecture and software API of the EXTOLL *Remote Memory Access (RMA)* unit. We focus

in particular on the aspects related to the GPU architecture. A more detailed explanation of the RMA unit can be found in [9], [10].

A. The EXTOLL RMA unit

The RMA unit is responsible for one-sided communication based on put and get operations. The communication between process and device is done using descriptors, for instance, WRs or completion notifications. WRs can be posted specifying where the data should be read from and where it has to be delivered to. In addition, a lightweight notification system provides information about already executed work requests.
Before communication can take place, a memory region has to be registered for the RMA unit. This is necessary as the EXTOLL NIC uses a global address space based on *Network Logical Addresses (NLAs)* to read and write into system memory. The translation is done by the *Address Translation Unit (ATU)* of the EXTOLL NIC. Once memory has been registered, it can be used for one-sided data transfers. The receiver does not have to call any routine to receive data as long as memory has been registered previously.
The RMA unit consists of three main units:

- Requester: Receives WRs and starts the data transfer. When the transfer has been started, a requester notification is created signaling the requester is able to receive another WR.
- Completer: Receives the WRs from the network. If there was a put command, the completer writes the data into the receiver's memory. In case of a get command, it reads the data from the memory and hands it over to the responder unit.
- Responder: Generates responses that are sent back to the source. This unit is only active for get commands.

The EXTOLL NIC provides a PCIe BAR slot for the RMA unit where the WR can be written to. Writing a WR to the BAR starts the data transfer. Notifications are created by the hardware and written to a queue structure in system memory. If notifications are used they have to be consumed and freed before the queue overflows.

B. RMA API

Before the RMA unit can be used, it has to be initialized. The initialization maps the PCIe BAR and the notification queues into the userspace, since the queues are allocated in kernel space at driver load time.
To set up communication between two nodes, a port has to be opened and a connection has to be established. The port gets assigned a window in the BAR where the WR can be written to.
Already allocated CPU memory can be registered for RMA transfers by calling a register function. The ATU translates the physical addresses into NLAs. RMA put and get commands use NLAs as parameters to create WRs and start data transfers. In order to check whether a command has been completed, notifications can be queried. Each unit like requester, completer or responder can create notifications. For two-sided communication, the sender can wait for requester notifications while the receiver waits for completer notifications to ensure the data has been received.

C. Extending the API for GPUs

In order to make use of the RMA API on the GPU, we had to extend the API. First, the RMA must be enabled to read and write GPU memory. Then, the memory is mapped to the PCIe BAR using GPUDirect RDMA. Finally, we map GPU memory to the user space, like described in our previous work [11].
In addition, the EXTOLL ATU needs to be able to translate MMIO addresses to NLAs to use them for RMA commands. This is done by a small driver patch that allows translation of MMIO addresses to their physical addresses. Then, they can be passed to RMA operations like put and get. However, this still needs the CPU to create and write the work request.
To avoid context switches between the CPU and GPU domain we made the RMA unit accessible by the GPU. This requires a NVIDIA kernel driver patch to map MMIO addresses into the GPU *Unified Virtual Addressing (UVA)* space, thus allowing to map the EXTOLL requester BAR as well as the notification queue to the GPU's address space. A single thread can now create the WR and hence start the data transfer. In addition, notifications can be consumed directly on GPU. This approach completely frees the CPU while communication is offloaded to the EXTOLL NIC, saving time and power.

IV. Infiniband implementation

In this section, we provide a short introduction to the Infiniband and Verbs-API for GPUs. A good and more detailed introduction into the Infiniband architecture can be found in [12].

A. Infiniband communication

All communication in Infiniband is handled between so called *queue-pairs* (QP). Essentially, a QP consists of two ring-buffers: one to send and one to receive work requests. These buffers are also called *queues* and are normally located in host memory. Each of theses queues is associated with one extra queue, the *completion queue* (CQ).
To initiate a data transfer, WRs are submitted to these queues. The Infiniband network device is notified about a new request by writing a notification to the so-called *doorbell register*. For user-space communication, the doorbell register is mapped into the user space with MMIO, so the operating system can be bypassed. The network interface reads the requests from the buffer and starts the data transfer. If the communication is completed, a notification is written to the assigned completion queue.
Infiniband supports one-sided remote read and remote write communication. The requests are submitted to the send queue and therefore only a completion entry on the active side is created.
This is different for send requests, which require a remote receive request to complete the communication. Receive requests

are submitted to the receive queue and require a destination address. If a send request is submitted without a matching receive request on the remote side, the communication fails. However, for send/receive request pairs, a completion notification is created on both sides.
Beside one- and two-sided communication, Infiniband also supports remote write requests with immediate data, which is something in between. A remote write request with immediate data requires a remote receive request to be completed, but the receive address can be set to zero, since the remote write request provides all necessary information for the communication. The advantage of this communication is that a completion element is created on both sides. However, the disadvantage is that the receive request has to be posted, otherwise the communication fails.
Similar to the EXTOLL RMA unit, for Infiniband the memory has to be registered to the network device to allow communication. In contrast to the RMA unit, Infiniband uses the virtual user space memory address and a key pair (one local and one remote key) to identify the memory and allow remote communication.

B. Infiniband Verbs on the GPU

A detailed description of Infiniband support for GPUs can be found in our previous work in [11], for brevity we describe the essential details here.
To allow the Infiniband network device to access GPU memory, GPUDirect RDMA has to be enabled. This can be done by using a patch that has been recently released by Mellanox [13]. In the next step, Infiniband resources are mapped to the GPU address space to create an Infiniband context on the GPU. The buffers for the send/receive and completion queues can be allocated on either host memory or registered GPU memory. For the latter one, an additional patch to the Infiniband device drivers is required. Also, the Infiniband doorbell register has to be mapped to the GPU address space. To allow this, the same patch for the low-level device drivers is required as described in the previous section for EXTOLL's RMA. The basic communication function, *ibv_post_send, ibv_post_receive* and *ibv_poll_cq* were ported to the GPU to enable direct communication from the GPU.

V. Performance analysis

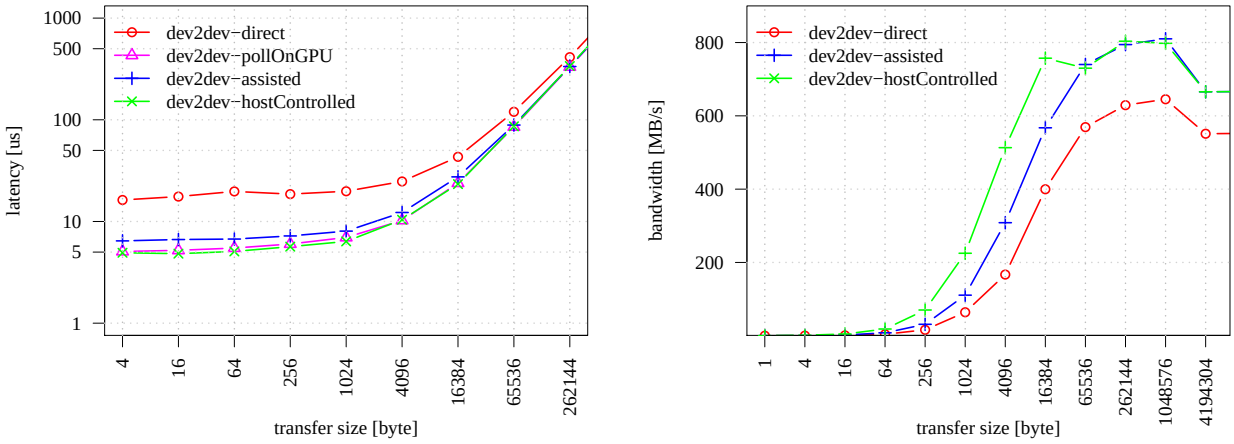
This section provides a performance analysis of the EXTOLL RMA and the Infiniband Verbs API for direct GPU communication. We present latency, bandwidth and message rate as key metrics for both implementations. In addition, we analyze performance counters to further explore issues when performance is affected negatively.
Our test bed consists of two nodes equipped with EXTOLL Galibier add-in cards, and two nodes with Infiniband 4X FDR HCAs. For the Infiniband nodes, the openstack subnet manager version 4.0.5 is used. It should be noted that the EXTOLL NIC is based on an FPGA implementation with a core frequency of 157MHz and 64bit wide datapath. We expect future ASIC implementations to improve performance significantly as core frequency will be increased to about 700MHz and internal datapaths become extended to 128bit.

A. EXTOLL RMA

In the following, we present our experiments with EXTOLL. We examined latency, bandwidth and message rate as key metrics for communication.
1) Latency and bandwidth: The latency and bandwidth experiments are conducted with following configurations:

- *dev2dev-direct*: The put operations are issued on the GPU. After the command was posted the requester notification is queried to ensure the command has been executed. Completer notifications are read to ensure data has been received.
- *dev2dev-pollOnGPU*: Instead querying notifications, the last element that is supposed to be received is periodically checked. This avoids system memory accesses and allows for polling on the GPU's device memory. However, this is only applicable for the ping-pong test.
- *dev2dev-assisted*: The actual data transfer is handled by the CPU. The CPU and GPU synchronize via a flag that is placed in host memory and mapped to the GPU's address space.
- *dev2dev-hostControlled*: The CPU issues the put command that copies the data from the sender's to the remote GPU. The control flow entirely remains on the CPU.

The results of a ping-pong as well as a bandwidth microbenchmark are shown in Fig. 1.
The latency for put operations that are executed on the GPU is almost twice as much as for host-controlled transfers (dev2dev-direct vs. dev2dev-hostControlled). The performance decrease is mostly caused by the PCIe traffic when notifications are queried on the GPU. Notifications are still placed in system memory and each query leads to PCIe transactions. For EXTOLL, notifications amount to 128 bit that are polled periodically, decreasing performance significantly. Thus, we replace the polling on notifications by polling on the last element that is supposed to be received, referred as *dev2dev-pollOnGPU* in Fig. 1. The resulting latency drops significantly and is even lower than host-assisted put operations, quite likely because no context switches are required anymore.
The corresponding bandwidth shows that there is still a gap between GPU and CPU-controlled RMA transfers. This may be caused by polling for requester notifications from the GPU. Another issue is that the bandwidth drops for message sizes larger than 1MB. This is due to a PCIe peer-to-peer issue that has already been evaluated in [14] and [15] and only occurs if data has been read from the GPU by another PCIe device like a network controller.
2) Message rate: Fig. 2 shows the message rate achieved with the EXTOLL RMA API for different approaches. The message size amounts to 64 bytes. Each message is sent over a different EXTOLL RMA port. For every port that is opened a new requester page on the PCIe BAR is allocated avoiding race conditions when multiple descriptors are posted in parallel. The examined methods are:



(a) latency **(b)** bandwidth

Fig. 1: Measured latency and bandwidth for the EXTOLL RMA API

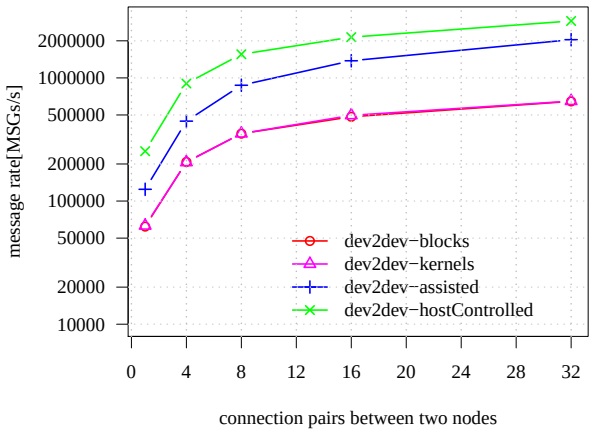


Fig. 2: Message rate for 64 byte messages achieved with the EXTOLL RMA API

- *dev2dev-blocks*: For every connection pair one CUDA block is launched. Each block posts one put command, resulting in 32 parallel put commands if 32 CUDA blocks are used.
- *dev2dev-kernels*: Instead of parallel CUDA blocks, different streams are used. For instance, if 32 connection pairs are used, 32 asynchronous kernels are launched with one CUDA block each.

The other methods are the same as for the latency and bandwidth experiment explained earlier in this section.
As a result, posting descriptors with multiple CUDA blocks performs similar as launching CUDA kernels with different streams whereby each stream posts one descriptor. The kernel launch overhead does not affect performance at this point. In addition, host-assisted transfers, where the GPU tells the CPU to perform a data transfer, performs worse than host-controlled operations due to synchronization overhead between the GPU and CPU domain. Nonetheless, both CPU-controlled data transfers are still faster.

metric	system memory	device memory
sysmem reads (32B accesses)	4,368	0
sysmem writes (32B accesses)	2,908	303
globmem64 reads (accesses)	0	1,314
globmem64 writes (accesses)	500	400
l2 read hits	0	3,143
l2 read requests	4,822	2,970
l2 write requests	5,268	404
memory accesses (r/w)	6,788	1714
instruction executed	46,413	22,491

TABLE I: Comparison of different polling approaches for the EXTOLL RMA API. The granularity of accesses is given in brackets after the metric. Device memory means the program polls on the last received element in device memory and system memory means the notifications that are created by the requester and completer unit are queried.

3) Analysis: We examine performance counters to evaluate system and global memory accesses caused by different polling approaches for the ping-pong microbenchmark with 100 iterations and a payload size of 1KB. The results are shown in Table I.
The results show why polling on system memory decreases the performance significantly. Especially system memory read operations are costly and all the accesses put a lot of pressure on the PCIe network. The system memory write operations are caused by writing the WR (192 bit) to the PCIe BAR, freeing notifications (128 bit) by resetting them to zero, and incrementing the read pointer (32 bit) of the queue structure. Polling on device memory (64 bit values) avoids consuming and freeing notifications, and therefore system memory accesses are only caused by posting WRs. According to Table I, polling on device memory causes 3 system memory write operation per iteration which is exactly the size of the WR (3x64 bit values). Because no notification is read, there are no system memory read operations. Polling on those notifications in system memory again leads to about 26 write and 43 read operation per iteration, excluding the 3 writes for posting the descriptor.
Other notable things are L2 cache accesses. Polling on the last

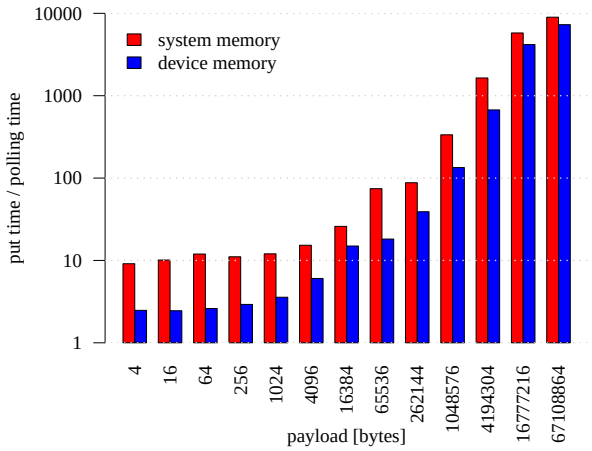


Fig. 3: Comparison of two different polling approaches for the EXTOLL RMA API. The numbers correspond to the WR generation time divided by the polling time.

received element in device memory can be kept in the L2 cache decreasing polling latency, therefore most accesses hit the L2 cache while polling on notifications in system memory cannot use the L2 cache at all. Note that for such global memory accesses, the L1 cache is bypassed. Furthermore, polling on notifications leads to twice as much instructions.

In Fig. 3 we split up the latency for both polling approaches into data transfer and polling on notifications time and divide both numbers by each other. For small messages, polling on system memory needs ten times the time than it is needed to post the WR. For example, polling on device memory consumes only about 2.5X more time than posting the WR. However, for rather large messages both approaches perform similar. The data transfer itself becomes the dominating fraction at this point.

B. Infiniband

Analogous to the previous section, this section provides results of the latency, bandwidth and message rate experiments, followed by a performance counter analysis. A detailed discussion of the latency and bandwidth results for Infiniband was presented in our previous work [11]. Here, we complement this by a detailed analysis of the message rate and the behavior, using GPU performance counters.

1) Latency and bandwidth: The latency and bandwidth results are shown in Fig. 4. Similar to the benchmarks for the EXTOLL RMA unit, we compare four different communications mechanisms:

- *dev2devBufOnGPU:* Communication is controlled by the GPU, the completion and the work requests buffers are allocated on GPU memory.
- *dev2devBufOnHost:* Communication is controlled by the CPU, the completion and the work requests buffers are allocated on host memory.
- *dev2devAssisted:* The GPU triggers the CPU to perform the communication by writing to a flag.
- *dev2dev-hostControlled:* The CPU entirely controls the communication while data is transferred between GPUs.

In all cases data is transferred between two GPUs on different nodes. We do not use the remote write with immediate value operation on the GPU, since this would add a lot of overhead to the GPU due to the generation of receive work requests. Instead, we poll on the last received element, since Infiniband guarantees in-order delivery of data, if a reliable connection has been set up between two QPs.

For host-initiated data transfers, we use the remote write with immediate value operation to synchronize the ping and the pong side, since the Mellanox GPUDirect patch does not allow to poll on GPU device memory from host. However, on host side the overhead for the work request generation is negligible. The latency for a GPU-initiated data transfer is much higher than the latency for a CPU-initiated data transfer, in particular for small messages. This is due to the higher overhead of GPU-initiated data transfers [11]. However, in contrast to EXTOLL's RMA, for Infiniband the location of the communication resources, here the queues, makes only a small difference.

For the bandwidth we see the same effect as for the EXTOLL RMA unit. The bandwidth is limited to about 1GB/s and decreases for larger messages. Again, this is due issues with PCIe.

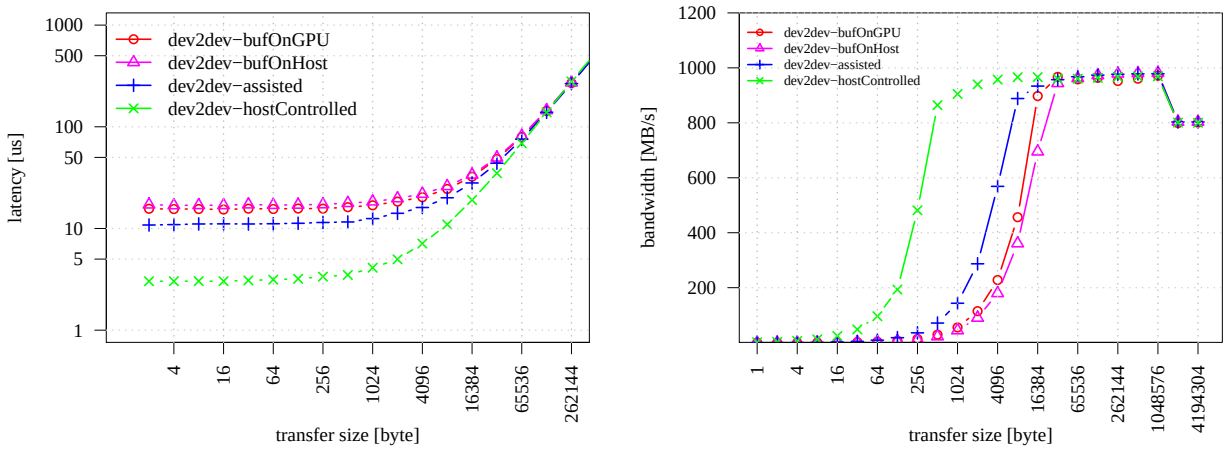
2) Message rate: In Figure 5 the results for the message rate using Infiniband on the GPU are shown. We use the same methods, *dev2dev-blocks* and *dev2dev-kernels*, like described in the previous section for the EXTOLL RMA unit. Every block respectively kernel uses another QP connection for communication. So for 32 kernels or blocks, 32 QP connections are established between the two GPUs. For the host-initiated and the host-assisted version, we do the same, but the QPs are controlled by the CPU.

There is no difference whether the communication is started from different blocks or kernels. Since we use very small kernels with only one block each, all kernels can run concurrently. The results show that for 32 connections almost the same message rate can be reached as for host-initiated data transfers. Since every kernel or block has its own QP, the work request generation can be perfectly parallelized.

We use a message size of 64 bytes for the message rate microbenchmark. This means that for 32 kernels or blocks, 2KB of data is transferred each iteration. Therefore, the results correspond to the bandwidth results, where host and GPU-initiated data transfers reach approximately the same bandwidth for a message size of 2KB.

The message rate of the host-assisted version remains constant for more than four connection pairs. The reason for this is that all connections are served by the same thread. If one block or kernel has a communication request, the thread is blocked for all other aspirants. This results in a blocking state for the GPU.

3) Analysis: We use performance counters to evaluate the behavior of our Infiniband Verbs implementation for GPUs. The results are shown in Table II. They clearly show why it makes no big difference for Infiniband if the buffers are allocated on GPU memory or on host memory. Although there are slightly more accesses to system memory if the buffers are allocated



(a) latency **(b)** bandwidth

Fig. 4: Measured latency and bandwidth for the Infiniband Verbs API

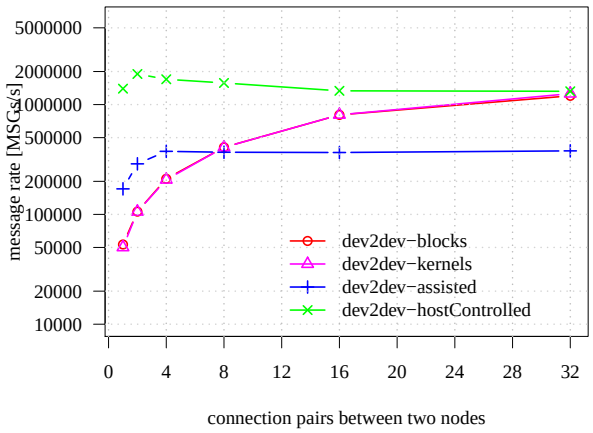


Fig. 5: Message rate for 64byte messages achieved with the Infiniband Verbs API

metric	Buffer on Host	Buffer on GPU
sysmem reads (32B accesses)	772	80
sysmem writes (32B accesses)	670	316
l2 read misses	999	1,405
l2 read hits	16,647	14,575
l2 read requests	16,657	15,110
l2 write requests	1,990	1,885
memory access (r/w)	59,937	58,905
instruction executed	123,297	110,463

TABLE II: Comparison of different buffer-placement approaches for the Infiniband VERBS API. The granularity of accesses is given in brackets after the metric. Buffer on host means the buffer is placed in host memory while the other approach places the buffer in the GPU's device memory.

on host memory, the difference is considerable smaller than for the EXTOLL RMA unit. In order to examine the cause for the high latency of Infiniband, the more interesting counters are the number of executed instructions and general memory accesses. Our ping-pong benchmark runs with 100 iterations resulting in about 110,000 instructions and 60,000 memory accesses. That means, that about 1,100 instructions and 600 memory accesses are required for one iteration. Note that most of these instructions have to be performed by a single thread, since the work request generation cannot be parallelized. This is a lot, in particular in contrast to the numbers for the EXTOLL RMA (Table I). It seems that the work request generation for Infiniband requires a lot more overhead. In another test, we measure the instructions that are required to post a single work request (*ibv_post_send*) and the number of instructions for a single successful polling (*ibv_poll_cq*).
It requires 442 instructions to post a work request and 283 to poll for the completion. A closer look to the Infiniband verbs code is very informative, what kind of instructions are required. For example, the elements for the work requests have to be converted from little-endian to big-endian to be usable for the Infiniband network device. To optimize this for the GPU, we used static converted values where possible. However, since the source and destination address and the message size may change for every communication request, these values have to be converted for every request. Older queue elements have to be *stamped* to be recognized as unused values from the prefetching unit of the network device.
The polling for a completion includes, if successful, the handling of the completion element. Thereby, the associated QP has to be picked out of the list of QPs, which also adds overhead to the completion handling process.

VI. Discussion

There are two basic functions for put/get communication: generating work requests and polling on notifications to ensure consistency. Both significantly impact performance.
We showed that EXTOLL allows to generate a work request by writing to the PCIe BAR. For Infiniband, however, issuing a work request is based on two steps: first, the work request has to be written to a queue in main memory, and second the network device has to be explicitly notified by writing to a doorbell register that is located on the device. In addition, In-

finiband requires big-endian values, causing lots of converting overhead.

On the other hand, Infiniband allows the notification queue to be dynamically allocated either in CPU or GPU memory. This reduces polling latency significantly. Contrary, EXTOLL pre-allocates notifications structures within the kernel driver. When a port is opened, these pre-allocated structures are assigned to this new port. This reduces initialization time, but the notification queue cannot be easily moved to GPU memory, resulting in significant additional overhead.

While generating work requests increases latency for Infiniband, polling on notifications in system memory decreases performance for EXTOLL. Nonetheless, CPU-controlled put/get operations always perform better than GPU-controlled operations. Controlling IO devices from the GPU needs to become more in-line with the GPU's execution model, in particular including thread-collective interfaces. Today, most APIs are optimized for single-thread performance.

Based on the explorations in this work, we summarize our findings in a set of claims for future optimizations of put/get interfaces.

1) The footprint of the interface has to be as small as possible, as GPU memory is scarce and inevitable for notifications
2) The interface of the API has to be in-line with the thread-collaborative execution model of a GPU
3) PCIe transfers for control have to be kept at a minimum, including work request generation but also notification queues in GPU memory that are being updated by the network device. Both have to be kept as small as possible

VII. Related Work

To the best of our knowledge, this is the first work that provides an in-depth analysis of put/get APIs on GPUs. GPU-initiated communication was first discussed in [16] from Owens et. al. They introduced a message passing interface on GPUs, that was, what we call "host-assisted. However, the authors strongly claim a support for direct communication among GPUs without CPU involvement.

In our previous work [11], we introduce a Infiniband Verbs implementation on the GPU, but without an in-depth analysis. In GGAS [17], we introduce a direct communication framework on GPUs that is based on remote load and store operations.

A lot of work has been done in using GPU performance counters to optimize GPU applications, but listing these works here would exceed the scope of this work.

VIII. Conclusion

In this work we investigate how GPUs can source and sink network traffic efficiently by exploring two APIs for put/get operations from a GPU's point of view. Both have been optimized for GPU-controlled data movements and performance results in terms of streaming bandwidth, ping-pong latency and sustained message rate between two distributed GPUs are reported. As the performance of such GPU-controlled data movements is still inferior to CPU-controlled ones, we perform a detailed analysis using GPU performance counters. Based on these insights, we state a set of claims for future GPU networking APIs. Although we focus on network device interactions, we think that these results are also applicable to other IO interfaces, including storage. In future work we gear to work towards GPU communication libraries that meets the previously stated claims.

Acknowledgment

We gratefully acknowledge the generous support of this research effort by Nvidia, Xilinx Inc and the EXTOLL company.

References

[1] [Online]. Available: http://green500.org
[2] *CUDA Programming guide*, Nvidia Std., 2014. [Online]. Available: http://docs.nvidia.com/cuda/index.html
[3] *The OpenCL Specification*, Khronos OpenCL Working Group Std., November 2013.
[4] *The OpenACC Application Programming Interface*, OpenACC Working Group and others Std., August 2013.
[5] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 10–pp.
[6] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D. K. Panda, "Extending openSHMEM for GPU computing," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 1001–1012.
[7] Y. Zheng, C. Iancu, P. Hargrove, S.-J. Min, and K. Yelick, "Extending unified parallel C for GPU computing," in *SIAM conference on parallel processing for scientific computing*, 2010.
[8] L. Oden, "GPI2 for GPUs: A PGAS framework for efficient communication in hybrid clusters," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing, M. Bader and A. Bode, Eds., vol. 25. IOS Press, March 2014, pp. 461 – 470.
[9] M. Nuessle, M. Scherer, and U. Bruening, "A resource optimized remote-memory-access architecture for low-latency communication," in *38th Conference on Parallel Processing (ICPP-09)*, 2009.
[10] H. Fröning, M. Nüssle, H. Litz, C. Leber, and U. Brüning, "On achieving high message rates," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.
[11] L. Oden, H. Fröning, and F.-J. Pfreundt, "Infiniband-verbs on GPU: A case study of controlling an infiniband network device from the GPU," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International*. IEEE, 2014, in press.
[12] G. F. Pfister, "An introduction to the infiniband architecture," *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001.
[13] (2014, jan) Mellanox OFED GPUDirect RDMA beta. [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family=116
[14] M. Si and Y. Ishikawa, "Direct MPI library for Intel Xeon Phi co-processors," in *Parallel and Distributed Processing Symposium Workshops & PhD Forums (IPDPSW)*, 2013.
[15] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. Panda, "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, 2013.
[16] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *International Symposium on Parallel & Distributed Processing. IPDPS2009*, 2009.
[17] L. Oden and H. Fröning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *IEEE Cluster*, 2013.