Infiniband-Verbs on GPU: A case study of controlling an Infiniband network device from the GPU

Lena Oden*, Holger Fröning† and Franz-Joseph Pfreundt*
*Fraunhofer Institute for Industrial Mathematics, Competence Center High Perfomance Computing, Kaiserslautern, Germany
oden,pfreundt@itwm.fhg.de
†University of Heidelberg, Institute of Computer Engineering, Heidelberg, Germany
froening@uni-hd.de

***Abstract*—Due to their massive parallelism and high performance per watt GPUs gain high popularity in high performance computing and are a strong candidate for future exacscale systems. But communication and data transfer in GPU accelerated systems remain a challenging problem. Since the GPU normally is not able to control a network device, today a hybrid-programming model is preferred, whereby the GPU is used for calculation and the CPU handles the communication. As a result, communication between distributed GPUs suffers from unnecessary overhead, introduced by switching control flow from GPUs to CPUs and vice versa. In this work, we modify user space libraries and device drivers of GPUs and the Infiniband network device in a way to enable the GPU to control an Infiniband network device to independently source and sink communication requests without any involvements of the CPU. Our performance analysis shows the differences to hybrid communication models in detail, in particular that the CPU's advantage in generating work requests outshines the overhead associated with context switching. In other terms, our results show that complex networking protocols like IBVERBS are better handled by CPUs in spite of time penalties due to context switching, since overhead of work request generation cannot be parallelized and is not suitable with the high parallel programming model of GPUs.**

***Index Terms*—GPUs Communication Heterogeneous Clusters Infiniband RDMA**

I. Introduction

Graphic Processing Units (GPUs) have gained high popularity in High Performance Computing. While power of single threaded CPUs stagnated and only by introducing multiple cores the CPU vendors were able to maintain performance growth, the performance of GPUs has increased dramatically in the recent years. The introduction of GPU computing languages like CUDA, openCL or directives based approaches like openACC made GPUs accessible for users who are not familiar with classical graphic aspects. Thereby, many non-graphic scientific and industrial applications were ported to GPU-based heterogeneous systems. Due to high performance per Watt GPUs become also a promising candidate for next generation exascale systems.

But GPUs only scale, if they perform their calculations in core and GPU device memory is a scarce resource. Since a single compute node only can handle a limited number of GPUs, a tightly coupled cluster of GPU-equipped nodes can help to overcome these limitations. But, despite their high popularity, data transfer and communication in GPU accelerated systems remains a problem. Recent technologies like GPUDirect RDMA help to overcome some limitations, since they allow direct data transfer between GPUs on distributed nodes.

However, this technology still requires a hybrid-programming model, where the GPU is used for computation, while the CPU and a CPU-based communication library is required for inter-GPU communication. This requires multiple context switches between the GPU and the CPU. In recent work [1], [2] and [3] the idea of direct sourcing and sinking network traffic from the GPU without any CPU involvement was claimed to be the ideal way utilize a hybrid cluster.

One of the most common interconnect in today high performance systems is Infiniband, and many GPU-accelerated clusters like the TSUBAME 2.5 [4] in Japan use Infiniband as an interconnect. In this work, we enable the GPU to source and sink network traffic directly to an Infiniband Host Channel adapter. The CPU is only required for a setup phase, but once everything is set up, the CPU is not required at all. This approach requires several changes to the GPU and network device drivers and user space libraries, but no changes to the hardware. The scope of this work is to demonstrate, which changes are required to enable the GPU to communicate with the network device, and to analyze if this approach is useful for GPU clusters. We compare our results with a host-assisted version, which uses a callback mechanism whereby the GPU initiates the CPU to perform the communication and a hybrid version, where the GPU is only used for computation, while the CPU handles the communication. However, our current implementation doesn't bring any performance improvements compared to the hybrid approaches. We analyze the bottlenecks of this behavior and claim, which requirements are necessary for efficient sourcing and sinking of network traffic from the GPU.

The rest of this paper is organized as follows. The next

section gives an overview about related work. In section 3 a short introduction into GPUs and Infiniband is given. Section 4 describes the implementation of an Infiniband context on the GPU. In section 5, we evaluate the performance and compare to other communication techniques. In section 6 we discuss the results, before we conclude in the last section.

II. Related Work

GPUs in cluster computing are pervasive, so various related work exists in this area. The idea of sourcing and sinking network traffic from GPUs existed before, in 2009 Stuart et.al. introduce DCGN [1] (Distributed Computing on GPU Networks), a framework that allows GPU threads to send and receive data with commands similar to MPI. Although the commands are called within a kernel, the data transfer itself is handled by the host system. Still, the authors clearly state that support for direct communication among GPUs without CPU involvement is desirable. They repeat this in [2] were they claim their requirements for an MPI version running on an GPU.

Another approach, which allows direct sourcing network traffic from the GPU is GGAS [3], which creates a global address space for communication in GPU-Clusters and completely avoid host accesses. However, GGAS requires a special kind of hardware, which is currently only available as an FPGA.

In [5] the efficiency of different GPU-to-GPU communication methods and their impaction application were examined. However, this work only looks on hybrid models, where the CPU is used for communication, and new techniques like GPUDirect RDMA were not considered.

The most common way to utilize a GPU cluster is a hybrid-programming model, which uses a host-based communication library to control the data transfer. In [6] Wang et al. introduce MVAPICH2-GPU, an MPI-Version for Infiniband that can use pointers to GPU memory. This work is optimized in [7] and in [8] to using GPUDirect technologies.

Another framework for GPU support is MPI-ACC, which is introduced in [9] by Aji et al. Its main focus is to be portable, so it not only supports CUDA but also OpenCL.

PGAS based communication frameworks, supporting GPUs are OpenShmem [10], GPI2 [11] for GPUs [12] or UPC for GPUs [13]. Also some of this approaches use GPUDirect RDMA technology, the communication handled by the host respectively the CPUs.

Controlling and Infiniband device from an accelerator device was discussed for example [14] for the Intel Xeon Phi . However, the Xeon Phi has a Linux runtime-system, which allows porting of device drivers and use a direct interface. This approach is currently not applicable for GPUs.

III. Background

In this section we describe the GPU architecture and the CUDA programming model. Then we provide a short introduction into the Infiniband architecture and how network traffic is sourced and sink on an Infiniband card. But first, we start to give a short introduction into the terms pinned and registered memory.

A. Pinned and registered memory

Peripheral devices like GPUs or network cards are able to direct access host memory. For this, the memory has to be pinned and registered for this device. Pinned memory means that the memory is locked and cannot be swapped. This allows static virtual to physical address translation. To register this memory for a device, the operation system kernel calculates the physical page addresses belonging to a virtual memory address. The device uses these addresses to create an own page table. By this, the peripheral device can direct access the host memory without any CPU involvement.

B. GPU and CUDA

A GPU is a peripheral device, which is connected to the host through a high speed IO slot, which is in most cases PCIe. A GPU has its own dedicated device memory, up to 12 GB on modern GPUs. Data transfer between host and GPU is usually performed by the DMA engine of the GPU, which has direct memory access (DMA) to both, host memory and device memory. To access host memory, this memory must be pinned and registered.

GPUs are high-core-count devices with multiple Streaming Multiprocessors (SMs), which are composed of a large number of processing cores. GPUs are optimized for high a high parallel execution model, which is referred as single instruction multiple threads (SIMT) model, while the single thread performance of a GPU is low compared to modern CPUs.

Threads are organized in blocks, but the scheduler of the GPU doesn't handle each single thread or block; instead threads are scheduled in warps (typically 32 threads). These warps are scheduled to the SMs during runtime. Context switching between warps comes at negligible costs; so long-latency events can easily be hidden. To maximize sustained performance, threads within a warp have to have similar control flows, as the scheduler is not able to handle such unaligned control flows efficiently. Therefore, all threads of one wrap will block if only a single thread performs a specific branch.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model for NVIDIA GPUs. It provides a virtual instruction set to manage GPU device memory and to start and synchronize computation kernel on the GPU. A more detailed description of GPUs and CUDA can be found in the excellent book by Kirk and Hwu [15]. Although we use CUDA in this work, all principles are also applicable to other GPU programming languages, including OpenCL.

C. Infiniband

Infiniband is a very popular interconnect, which is currently used by 41% of the top500 high performance computing systems [16]. Infiniband supports remote direct memory access

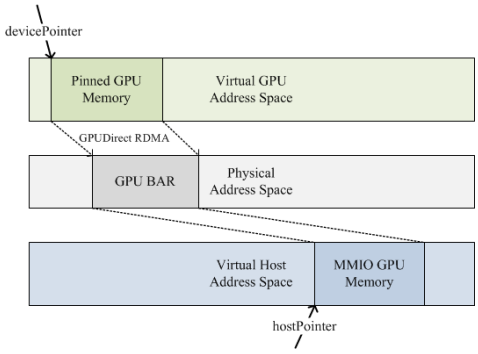


Fig. 1. Using GPUDirect RDMA by mapping GPU memory to the user space

(RDMA), which allows direct reading and writing memory of remote node. It also support send and receive and atomic operations on remote memory regions.

To allow the Infiniband device access to memory, this memory must be pinned and registered.

Communication in Infiniband is handled between so called queue-pairs (QPs). A QP consists of two queues: one for send- and one for receive-requests. RDMA request are submitted to the send queue. Every queue also has a corresponding completion queue, where the Infiniband network interface or host channel adapter (HCA) confirms the completion of an operation.

To initiate a data transfer, a work request has to be generated. This work request contains all necessary information of the communication, like the local address, the transfer size or, for RDMA operations, the remote address. This work request is submitted to the queue, a ring buffer in registered host memory.

If work request is submitted, the CPU writes to the so-called *doorbell register*. The doorbell register is a special register of the Infiniband device, which can be mapped to the user space with memory-mapped IO (MMIO). If a notification to the doorbell register is written, the Infiniband HCA reads the work request and performs the communication. If the communication is completed or if an error occurred, the HCA writes a notification to the completion queue, which can be used by the host to confirm the completion of an operation. An good and more detailed introduction into the Infiniband architecture can be found at [17].

IV. Device to device communication between GPU and the Infiniband network device

In this section we describe, how direct communication between multiple GPUs is realized. Sourcing and sinking of communication request directly from the GPU requires direct communication between the Infiniband HCA and the GPU. As both are peripheral devices, this is referred as device-to-device communication. This is originally not supported and requires several changes in the device drivers and user space libraries, which are now explained in more detail.

We have to consider two ways of communication: To directly transfer data between two GPUs via the Infiniband device, the Infiniband HCA must be able to read and write GPU device memory. To initiate the data transfer, the GPU must control the network device. We describe first, how the direct data transfer is realized, next we describe, how the GPU is enabled to control the network device.

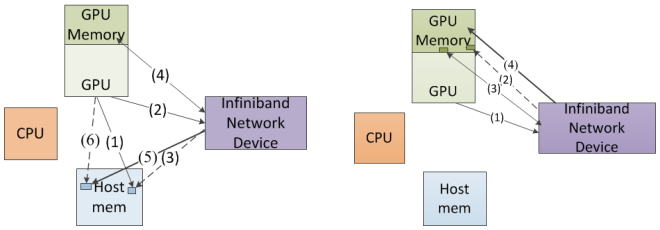
A. GPUdirect RDMA

In 2012, Nvidia released CUDA 5, together with a new feature, called GPUDirect RDMA. GPUDirect RDMA allows RDMA capable devices to direct access to GPU device memory, so data can be directly transferred between two GPUs without buffering in host memory. This technique requires some changes in the network-device drivers, which are described in the CUDA toolkit documentation and can be found at [18]. Mellanox released a patch for their Infiniband devices, which is available as a beta release on [19] and will be official released with CUDA 6. Still, we developed our own version of the patch for two reasons: The Mellanox patch only allows the registration of memory regions, which are used, for data transfer. This is sufficient for host controlled device-to-device data transfer. Our approach also allows allocating Infiniband queues on GPU device memory. Besides, our approach also allows direct polling of GPU memory from host. However, the final result of both patches is the same: the Infiniband device can access GPU device memory by using physical addresses. Therefore, there are no performance differences between the Mellanox patch and our version for GPU to GPU data transfers.

To allow an Infiniband HCA to access host memory, the device requires the physical addresses. Within the physical address space are linear windows called PCI base address registers (BAR), which are normally used to control a peripheral device from host and every common PCI/PCIe device provides at least one BAR. However, from the device points of view, there is no difference, if a physical address points to host memory or the BAR of another device. GPUDirect RDMA allows mapping GPU device memory to one of the BARs of the GPU.

Our GPUDirect solution works in two steps. We developed a small device driver and user space library, which pins GPU device memory and maps it to the BAR of the GPU. The BAR addresses of the GPU are mapped to the user space with memory mapped IO (MMIO), like shown in Figure 1. Since the BAR-addresses are pointing to the GPU memory, this allows direct access to the GPU memory from host without special communication functions.

Next, this virtual MMIO-address is handed over to the Infiniband memory registration function. This function forwards the address to the Infiniband device driver. For a normal user space address, the device driver would pin the memory and calculate the physical page addresses. This operation fails for MMIO addresses. So, we developed a small patch for the Infiniband user space driver. This patch recognizes, if a virtual address is an MMIO-address and calculates the physical addresses belonging to the address. These physical addresses,



(a) Buffers on host (b) Buffers on GPU

Fig. 2. PCIe accesses to source and sink communication requests on the GPU, solid lines are write request, dashed are read requests

pointing the GPU-BAR, are handed back to the Infiniband device driver.

The disadvantage of this solution is that it adds some overhead to the application due to address translation, since the Infiniband device driver and the host are using another address then the GPU to access the memory.

B. Creating an network interface on the GPU

GPUDirect RDMA allows a direct data transfer between the memories of two GPUs on different nodes, but the communication has to be controlled by the host. To allow the GPU to control the network device, further steps are required.

1) Setup connections on the host: The first step is to set up an Infiniband context on host. This step requires a several operation system calls and therefore cannot be performed by a GPU.

First, an Infiniband context is created, which means that network device is initialized. For our purposes the most important part is the doorbell register. Every Infiniband context gets its own doorbell register that is mapped to the user space with MMIO. This doorbell register is required to source and sink network traffic, like mentioned in section III-C.

Next, a protection domain is created. Protection domains allow associating multiple resources like completion queues and queue pairs with a single domain of trust.

For this protection domain the completion queues and the queue pairs are created. The creation includes the allocation and registration of the ring buffers.

We create QPs and the corresponding completion queues and establish connections to remote QPs. To enable RDMA access, we register memory regions on GPU devices memory and exchange the information about these memory regions with the remote nodes. A more detailed description about the setup routines for Infiniband connection on the host can be found at [20].

2) Map the resources to the GPU-Address space: To allow the GPU to directly access the resources that are required to source and sink network traffic, these resources have to be mapped to the address space of the GPU. To map the doorbell register to the GPU address space, we handle the virtual MMIO address to the CUDA function *cudaHostMemRegister*. Like the Infiniband device driver, the GPU driver normally would fail by trying to register this address. However, we apply the same patch we use for the Infiniband driver to the low level GPU device driver to allow mapping MMIO addresses to the GPU address space. Although most of the Nvidia driver is only provided as a binary, the memory-locking function *nv_lock_user_pages* has to be linked against the operation system; therefore this part of the driver can be manipulated after extracting the Nvidia driver package. There are two ways to allow the GPU to access the queue ring buffers. Either, the queue buffers are allocated on host memory and then registered for the GPU with *cudaHostMemRegister*. Or, the buffers are directly allocated on the GPU and our GPUDirect RDMA patch is used to register these buffers for the network device. By this, we avoid host memory accesses from the GPU trough the PCIe bus.

Figure 2a and Figure 2b shows how many accesses through the PCIe bus are required to source and sink a communication request from the GPU for both cases. In Figure 2a the queues are located on host memory. First, the GPU has to write the work request to host memory (1). Then, the GPU writes to the doorbell register to initiate the communication (2). The network device reads the work request from the host (3) and starts the data transfer by reading or writing the data directly from the GPU (4). If the communication is completed, the Infiniband device writes a completion to the host memory (5). To verify the completion, the GPU reads this completion from host memory (6).

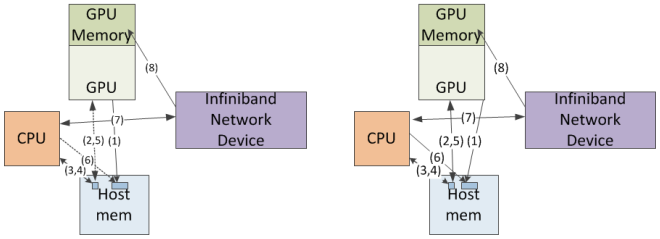
In Figure 2b the buffers are allocated on GPU device memory, and the GPU doesnt have to go trough the PCIe bus to write a request. Still, the GPU have to write to the doorbell register (1) to initiate the data transfer. The network device reads the work request from GPU memory (2). If the data transfer (3) is completed, the Infiniband device writes the completion directly to GPU memory (4), where a GPU thread can consume the completion notification without further accesses trough the PCIe-bus.

However, to allow the allocation of queue buffers on the GPU, we implemented the functions *ibv_create_gpu_cq* and *ibv_create_gpu_qp* , which creates the completion queues and QPs with ring buffers located on GPU memory.

3) Setup an Infiniband context on the GPU: In the last step, we have to setup an Infiniband context on the GPU. All necessary resources are transferred from the host to the GPU. These resources include the mapped addresses of the queues and the doorbell register, but also local identification numbers for QPs or memory regions.

To keep the overhead as small as possible, only the necessary information is ported to the GPU. We developed a small framework, which manages the Infiniband resources on the GPU and allows sourcing of communication requests and polling on the completion queue. For this, we port the following functions of Infiniband user space library to the GPU: *ibv_post_send*, *ibv_post_revecive* and the *ibv_poll_cq*.

We do this by copying most of the host code to the GPUs. Since the creation of work request and posting to the doorbell register can't be parallelized, there is hardly any optimizing for the GPU. Nevertheless, we reduce the functionality of the operations to a minimum to keep the overhead as small as



(a) Sourcing of communication requests

(b) Completion of communication requests

Fig. 3. Host assisted sourcing and sinking of a communication request

possible.

To avoid race conditions, we also make sure that only one block can access these resources concurrently and only one thread per block can source requests. We apply a mutex to theses functions using GPU atomic operations, like described in [21].

After this is done, the GPU is able to source autonomously send, receive and RDMA requests to the Infiniband HCA by completely bypassing the host CPU.

C. Host-assisted sourcing and sinking of network traffic

The previous described solution requires a lot of changes to network device drivers and user space libraries and also add a lot of overhead to the GPU due to work request generation and communication with the network device. Therefore, we developed a more general, but host assisted solution to source and sink network traffic from the GPU. The GPU still creates a work request, but the request is forwarded to the CPU, which actually performs the communication. A similar approach was described in [1] for message passing. Still, message passing adds additional overhead to the communication and techniques like new GPUDirect RDMA help to improve inter-node communication, whereof our approach can benefit. We use the GPI2 for GPUs as communication layer, since it adds very low overhead to the communication, especially compared to MPI [22].

Our solution use GPUDirect RDMA technology to direct transfer the data between two GPUs, but also copies in host memory are supported, since the complete data transfer is controlled by the CPU. We use a callback-mechanism, like described in [23] to post a communication request from the GPU to the CPU. In this approach, the CPU polls for a flag in host memory, which is also mapped to the GPU address space. The GPU writes to a flag to request a communication request. We borrow the idea of queues from the Infiniband device and the GASPI specification [24]. An Infiniband host queue pair gets its counterpart on the GPU. This counterpart is a registered host memory buffer, which is also accessible from the GPU.

In Figure 3a is shown, how the GPU initiates a data transfer. First, the GPU writes the request that contains all necessary information for the communication to the queue (1). After the request is written, the GPU writes to a callback flag on host to notify the CPU about the new work request (2). On host, a thread polls for the callback flags (3). If a new request arrives, the host accepts this work by writing a new value to the callback flag to notify the GPU that the request was accepted (4). The GPU accepts the flag (5) and can resume the computation work. Meanwhile, the CPU uses the work request in the host buffer (6) to initiate the communication (7). This may require further communication between the network device and the CPU. For a better overview we omitted these steps in Figure 3a.

For the completion of a communication request a similar approach is used, like shown in Figure 3b. The GPU sends a request to the host by writing to a request flag (1). If the CPU accepts this request (2), it communicates with the network device to verify the completion of a communication request (3). Once the completion has occurred, the CPU notifies the GPU about this by updating the flag. The GPU accept this completion and now can resume the work.

This implementation adds less communication overhead to the GPU than the previous described approach. However, the main disadvantage is that a thread is required to the control the communication. Since this thread is in a polling state, it consumes CPU cycles and thereby power.

V. Performance Results

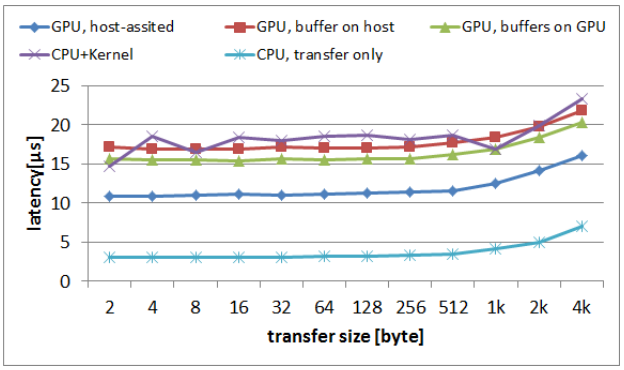
In this section we analyze the performance results of our previous described approaches. Our test system consists of two machines, each with two Intel 6-Core Xeon X5660, and directly connected using Mellanox Connect X3 Infiniband. Each machine is equipped with on Nvidia Kepler K20 GPU, which comes with 13 SMs, each with 64 double precision cores; or 832 cores in total. The GPUs are equipped with 5GB GDDR5 device memory and support GPUDirect RDMA features.

Since our approach requires a lot of changes in the device drivers and Infiniband user space libraries, we were not able to run our tests on a larger cluster. However, the scope of this work is to analyses the capabilities of GPUs to source and sink network traffic and for this, a test system with two GPUs is sufficient.

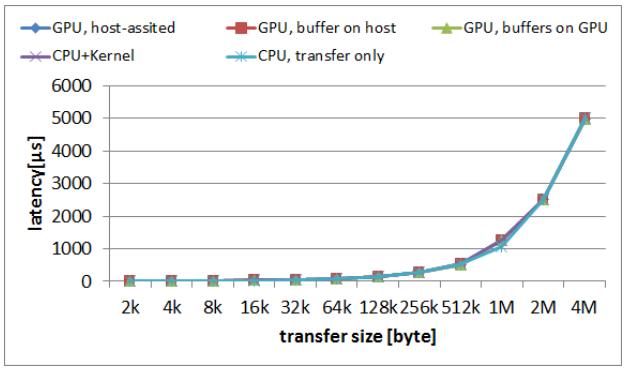
We use latency and bandwidth test to analyze the basic performance. We also implemented a barrier as synchronization primitive. Since Infiniband also support atomic operations on shared memory regions, we also test the performance of these operations on GPU memory segments. As a more complex example, we run a basic Jacobi benchmark that is based on stencil operations and requires boarder exchanges.

A. Latency

Our first test is a simple *pingpong benchmark* to measure the latency of a GPU initiated data transfer. For this, we start one kernel on each GPU, which performs the polling and communication for a given number of iterations. We use the Infiniband `rdma_write` operation to directly write to the memory of the remote GPU. We measure the runtime of this kernel and divide it by the number of the iterations. It the

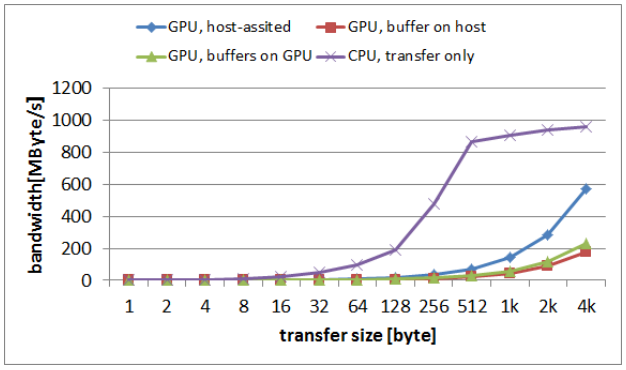


(a) small data sizes

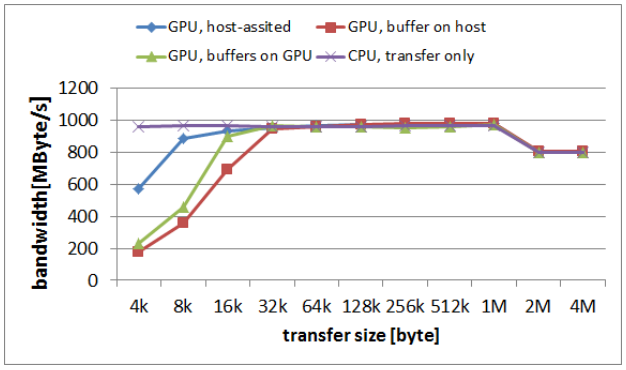


(b) large data sizes

Fig. 4. Latency of GPU to GPU data transfer



(a) small data sizes



(b) large data sizes

Fig. 5. Bandwidth of GPU to GPU data transfer

number of iterations is large enough, the kernel launching and synchronization overhead can be neglected. We compare our results with a hybrid version, were the host solely controls the network device. To take the effect of context switches into account, we use a CUDA kernel to poll on the data. If the data has arrived, the polling kernel is completed and the host starts the data transfer.

For comparison purposes, we also measure the pure data transfer latency, which neglects the overhead of finishing and restarting the polling kernel. The results are shown in Figure 4a for small data transfer sizes and in Figure 4b for lager data transfer sizes.

Unexpectedly, sourcing and sinking of network traffic directly from the GPU do poorly, especially for small message sizes. It only makes a small difference if the queue buffers are allocated on GPU or on host memory. The host assisted version is performing best for all message sizes, but only slightly better than the hybrid version taking the context switches into account. However, the pure data transfer latency, measured on host, is significantly better than the all versions taking the GPU into account.

For large messages the latency doesn't differ between all approaches, as shown in Figure 4b.

The results also show the benefits of the GPUDirect RDMA-technology compared to previous approaches, were host copies where required. Due to the overhead of host-memory copies, the minimal latency for a host initiated GPU-to-GPU data transfer were around 30 μs [5].

B. Bandwidth

Our second test is the bandwidth test. For this, multiple remote write requests are posted to one queue and synchronized, if the queue is full. Again, only one GPU kernel is started to perform this work. We compare the bandwidth with a host initiated data transfer between the GPUs. Since no GPU kernel is started, this benchmark doesn't take the context switches into account. The results are shown in Figure 5a and 5b.

For small messages, the bandwidth of a host initiated data transfer is much higher than the bandwidth of a GPU initiated data transfer. Only for messages larger than 32 kByte, a GPU initiated data transfers reaches the same bandwidth like a host initiated data transfer.

The maximal bandwidth is around 980 MBytes/s, what is much lower than the expected hardware bandwidth, which is around 3 GByte/s for a host to host data transfer using Infiniband QDR. For data sizes larger than 1 MByte the bandwidth even falls down to 800 MBytes/s. This results are in particular disappointing, if we compare them with

the results in [5], [8] and [12], where a bandwidth between 2.6 GB/s and 3 GB/s are reached for a GPU-to-GPU data transfer using Infiniband QDR. However, the low bandwidths is caused by the insufficient support of PCIe device to device communication in common Intel chipsets, which limits the bandwidth of direct device to device data transfers.

A higher bandwidth currently only can be reached, if the data are buffered in host memory. However, this limitation may be eliminated in next generation chipsets.

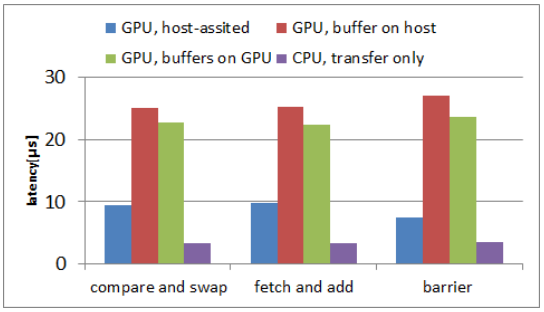


Fig. 6. Performance of atomic operations and barrier

C. Atomic Operations and Barrier

The performance of Infiniband atomic operations on GPU memory are shown in Figure 6. The host version doesn't take context switches into account, but the atomic operations are performed on GPU memory. Again, GPU initiated atomic operations are doing poor. Of these, the host-assisted version is performing best, since it is more than three times faster than the GPU-only versions.

In Figure 6 also the performance results of a barrier are shown. However, since we only use two GPUs for our basic tests, this result doesnt say anything about scaling. Still, the GPU initiated versions a performing much worse than the CPU initiated version.

D. Stencil Code

As a more application level benchmark, we implemented a 3-D stencil code. 2-D thread blocks process the grid from the bottom to the top. The domain is sliced along the z-direction between the GPUs, so the boundary values are on the top and bottom of the grid processed from different blocks.

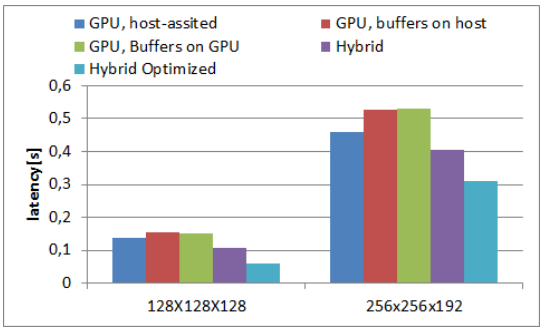


Fig. 7. Performance of a stencil code with different problem sizes

To guarantee data consistency, the last block processing the grid, sources the network traffic. We use CUDA atomic operation to determine this last block. We used a flag mechanism, to synchronize the sending and the receiving side. The flag is checked, once the remote data are needed. This allows an overlapping of communication and computation on the GPU. The results are shown in Figure 7 for two problem sizes.

We compare our results with hybrid version, using GPI2 for GPUs [12] as a communication layer. We implemented two versions of this hybrid version. The simple one only starts one kernel, synchronizes this kernel and then performs the communication. The second one overlaps communication and computation by starting two kernels, one for the upper part and one for the lower part. The calculation of the upper part is overlapped with the transfer of the bottom boundaries and the other way around.

The hybrid versions using GPI, both the optimized and the simple version, are performing much better than the versions sourcing and sinking network traffic directly from the GPU. The optimized hybrid version is two times faster than the versions, where the GPUs control the network device.

VI. Result Discussion

Our results show, that controlling the network device from the GPU currently doesn't provide any performance improvements but in some cases deterioration.

To find the cause for this behavior, we measure the latency of sourcing a communication request on the GPU and the CPU, without the message transfer latency. In other words, we measure the runtime of the function *ibv_post_send* on both, host and GPU. The results are shown in Table I.

TABLE I
Runtime of *ibv ibv_post_send* on different platforms

GPU host-assited	GPU buffers on GPU	GPU buffers on Host	CPU
6.6μs	9.6μs	10.5μs	0.1μs

It takes more than 100 times longer to source network traffic on the GPU than on the host. The host-assisted version is only slightly better. The work request generation on the GPU outperforms the actual data transfer latency for small message, while it can be neglected on the CPU.

We assume two reasons for this behavior: First of all, the work request generation can only be done by a single thread, while GPUs are only optimized for multi-threaded work and the single thread performance of a GPU is very low. Besides, GPUs are optimizes for floating point operations, and not for work request generation.

Second, the work request generation requires several accesses to the global device memory or the host memory, which are routed through the PCIe-bus. Since a single thread performs these accesses, they cant be coalesced. In contrast to CPUs, GPUs are not optimized for low latency memory accesses. Normally, long latency memory accesses are hided

by scheduling another wrap of threads [15], but this doesnt work for single-threaded work like work request generation.

For the poor performance of the stencil benchmark further causes have to be added.

While one GPU thread communicates with the network device, all other threads of this wrap block and cannot continue the calculation. Since GPU threads are non-preemptable, and it's not guaranteed when a specific thread is scheduled, it is also not possible to offload the communication work to one specific thread or block.

Furthermore, if the CPU is used for communication, this work is outsourced to the CPU, which also allows a better overlapping of communication and computation and reduces the work for the GPU.

VII. Conclusion

We showed in this work, that the device drivers and user space libraries of Infiniband network cards and GPUs can be modified to allow the GPU to control the network device and independently source and sink network traffic. However, our performance results show that this approach does not improve but deteriorate the performance of simple applications. It's even faster for the GPU to initiate the CPU to control the communication than doing this itself. This is caused by the overhead of work request generation on GPUs. While CPUs are highly optimized for this single-threaded work, the GPU performs poorly.

However, communication in GPU accelerated clusters remains a challenging problem. For future exascale systems, power efficient communication strategies must be found. We see two basic options to allow efficient communication directly from the GPU. Either a hybrid solution, where the GPU is equipped with a low power CPU, which is optimized for single threaded work like communication and work flow control. Or, another kind of network hardware is required, which allows an easier sourcing and sinking of communication requests and is more in line with the high-parallelized GPU model.

Our future work will look more closely into both directions. First of all, a power analysis is required to verify if direct sourcing and sinking of network traffic from the GPU can help to save power. We also will have a closer look on other RDMA capable hardware, which may be better, suitable for GPUs to initiate data transfer. We also will have a closer look at the new CUDA-dynamic parallelism feature that allows starting computation kernels directly from the GPU. Maybe this feature will help to develop more efficient GPU-only solutions.

Acknowledgment

We gratefully acknowledge the generous support of this research effort by Nvidia Corporation.

References

[1] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. Rome, Italy: IEEE, 2009, pp. 1–12.

[2] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending MPI to accelerators," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ser. ASBD '11. New York, NY, USA: ACM, 2011, pp. 19–23.

[3] L. Oden and H. Fröning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. Indianapolis, USA: IEEE, 2013.

[4] (2014) TSUBAME 2.5 - cluster platform sl390s g7, xeon x5670 6c 2.930ghz, infiniband QDR, nvidia k20x. [Online]. Available: http://top500.org/system/178249

[5] A. J. Pena and S. R. Alam, "Evaluation of inter-and intra-node data transfer efficiencies between gpu devices and their impact on scalable applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 144–151.

[6] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for infiniband clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 257–266, 2011.

[7] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. Shanghai, China: IEEE, 2012, pp. 1848–1857.

[8] 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 *Proc. Intl Conf. Parallel Processing*, Lyon, France, 2013.

[9] A. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur, "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*. Liververpool, United Kingdom: IEEE, 2012, pp. 647–654.

[10] 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*. Boston, Massachusetts USA: IEEE, 2013, pp. 1001–1012.

[11] R. Machado and C. Lojewski, "The fraunhofer virtual machine: a communication library and runtime system based on the rdma model," *Computer Science-Research and Development*, vol. 23, no. 3-4, pp. 125–132, 2009.

[12] L. Oden, "GPI2 for GPUs: A PGAS framework for efficientcommunication in hybrid clusters," in *Parallel Computing - ParCo2013, International Conference on, in press*. Munich, Germany: IOS, 2013.

[13] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, "Unified parallel C for GPU clusters: Language extensions and compiler implementation," in *Languages and Compilers for Parallel Computing*. Springer, 2011, pp. 151–165.

[14] M. Si, Y. Ishikawa, and M. Tatagi, "Direct MPI library for Intel Xeon Phi co-processors," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. Boston, Massachusetts: IEEE, 2013, pp. 816–824.

[15] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[16] (2014) The top 500 website. [Online]. Available: http://www.top500.org/

[17] G. F. Pfister, "An introduction to the infiniband architecture," *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001.

[18] (2014, jan) Developing a linux kernel module using rdma for gpudirect. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[19] (2014, jan) Mellanox ofed gpudirect rdma beta. [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family=116

[20] (2014, jan) Infiniband: An introduction + simple IB verbs program with RDMA write. [Online]. Available: http://www.cloudcomp.ch/2013/11/infiniband-an-introduction-simple-ib-verbs-program-with-rdma-write

[21] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for gpus," *arXiv preprint arXiv:1110.4623*, 2011.

[22] D. Grunewald, "BQCD with GPI: A case study," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 2012, pp. 388–394.

[23] J. A. Stuart, M. Cox, and J. D. Owens, "GPU-to-CPU callbacks," in *Euro-Par 2010 Parallel Processing Workshops*. Ischia, Italy: Springer, 2011, pp. 365–372.

[24] D. Grünewald and C. Simmendinger, "The GASPI API specification and its implementation GPI 2.0," in *7th International Conference on PGAS Programming Models*, 2013, p. 243.