GGAS: Global GPU Address Spaces for Efficient Communication in Heterogeneous Clusters

Lena Oden[1], Holger Fröning[2]
[1] Fraunhofer Institute & University of Heidelberg, Germany, [2] University of Heidelberg, Germany

Abstract— **Modern GPUs are powerful high-core-count processors, which are no longer used solely for graphics applications, but are also employed to accelerate computationally intensive general-purpose tasks. For utmost performance, GPUs are distributed throughout the cluster to process parallel programs. In fact, many recent high-performance systems in the TOP500 list are heterogeneous architectures. Despite being highly effective processing units, GPUs on different hosts are incapable of communicating without assistance from a CPU. As a result, communication between distributed GPUs suffers from unnecessary overhead, introduced by switching control flow from GPUs to CPUs and vice versa. Most communication libraries even require intermediate copies from GPU memory to host memory. This overhead in particular penalizes small data movements and synchronization operations, reduces efficiency and limits scalability. In this work we introduce global address spaces to facilitate direct communication between distributed GPUs without CPU involvement. Avoiding context switches and unnecessary copying dramatically reduces communication overhead. We evaluate our approach using a variety of workloads including low-level latency and bandwidth benchmarks, basic synchronization primitives like barriers, and a stencil computation as an example application. We see performance benefits of up to 2x for basic benchmarks and up to 1.67x for stencil computations.**

Keywords-parallel processing, hybrid computing clusters, GPU communication, bulk-synchronous execution

I. INTRODUCTION

Various technological limitations have led to a stagnating performance of single-thread CPUs, and only by introducing multiple cores the CPU vendors were able to maintain Moore's law. Opposed to this, the performance of GPUs has increased dramatically in the recent years and led to an adoption of a variety of non-graphical applications to GPUs, emphasized by popular programming paradigms like NVidia's *CUDA* or *OpenCL* by Khronos or directive-based approaches like *OpenACC*. Even Intel had to confess that GPUs are faster than CPUs for a pretty broad range of applications [1]. Along with this interest in GPU Computing, the high performance computing community rapidly adopted GPUs for their purposes, employing them in clusters as accelerators, trying to satisfy more of the computational needs of their performance-hungry applications.

GPUs are primarily designed to perform graphical computations, and the huge consumer market allows GPUs to be highly cost-effective. Considering this specialization, it's no surprise that throughput-oriented applications can benefit most from GPUs. Furthermore, GPUs only excel if they can perform their calculations *in-core*, otherwise performance is significantly limited by data movement over the PCIe interface, which is several orders of magnitude slower than the GPU's special memory. This is a severe limitation, in particular taken into account that GPUs suffer from too few memory anyway, which is typically only in the range of 4 - 6GBs.

Thus, even using GPUs, the computational requirements of many applications still cannot be satisfied and data-intensive applications (*Big Data*) are pushing these needs even further. A tightly-coupled cluster of GPUs can help to overcome this situation. For such a communication-centric architecture, minimal costs for communication and synchronization are required. Then, a GPU cluster can be a viable way to overcome the limitations of in-core computing, and use resource aggregation to keep the majority of data in-core. However, GPUs are peripheral slave devices and thus incapable of sourcing or sinking network traffic. Usually, a communication layer running on the CPUs is used as communication assistant. For most use cases, this is an unnecessary indirection that limits performance and scalability.

In this work we introduce *Global GPU Address Spaces* (GGAS), a communication model for distributed GPUs that differs from previous work in several aspects:

1. GGAS maintains the GPUs bulk-synchronous, massively parallel programming model by relying on thread-collective communication.
2. GGAS allows confining the control flow to the GPU domain, bypassing the CPUs for all computation and communication tasks and avoiding context switches that are costly in terms of energy and time.
3. Opposed to communication layers based on message passing, GGAS minimizes branch divergence[1], as communication is performed by all threads in a block collaboratively.
4. GGAS is a direct, zero-copy communication model that moves data without intermediate copies between

[1] GPUs can only maximize sustained performance if a single control flow is maintained for all threads within a so called warp (typically 32 threads). Otherwise, the so called branch divergence leads to substantial performance losses.

distributed GPU memories, again contributing to the minimization of time and energy.

Note that this work maintains the commodity aspect of GPUs, as all required hardware changes are constraint to the network device, and do not affect the GPU. The network device is responsible to intercept local accesses targeting the global address space and to forward them to remote locations.

In the remainder of this work we start with a background on GPU computing and available GPU communication techniques in section 2, before we present our idea of Global GPU Address Spaces in section 3. In section 4, we describe the technical details and implications of our approach. Section 5 is dedicated to our evaluation methodology and describes our example workloads. In section 6 we discuss the obtained results and compare to other communication techniques. Section 7 presents related work, before we summarize and conclude in the last section.

II. Background

A GPU is a powerful high-core-count device with multiple *Shared Multiprocessors* (SMs) that can execute thousands of threads concurrently. Each SM is essentially composed by a large number of computing cores, and a shared scratchpad memory. Threads are organized in blocks, but the scheduler of a GPU doesn't handle each single thread or block; instead threads are organized in warps (typically 32 threads) and 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; otherwise the branch divergence will result in performance losses as the scheduler is not able to handle such unaligned control flows efficiently. Also, enough threads have to be ready to maintain full utilization when long-latency events occur. Thus, typically at least one order of magnitude more threads are scheduled than can execute concurrently. Also, memory accesses are only efficient if multiple accesses from different threads can be coalesced. Thus, enough threads have to access non-conflicting memory locations concurrently, to offer the memory controller enough possibilities for coalescing.

CUDA is a parallel computing platform and programming model created by NVidia, which provides a virtual instruction set to use NVidia GPUs for computation. CUDA allows running a parallel kernel on the GPU, but the CPU has to launch this kernel. A more detailed description of GPUs and CUDA can be found in the excellent book by Kirk and Hwu [2]. Although we use CUDA in this work, all principles are also applicable to other GPU programming languages, including OpenCL.

III. The GGAS Model

The idea of a cluster-wide *Global GPU Address Space* (GGAS) is to allow efficient and fast communication between multiple GPUs in heterogeneous clusters. The common method for GPU-to-GPU communication is a hybrid programming model, using a GPU programming language like CUDA, OpenCL, or OpenACC in combination with a message passing library like MPI. The GGAS programming model distinguishes in two main points from this approach:

1. GGAS maintains the bulk-synchronous, massively parallel programming model of GPUs without increasing complexity by introducing message passing paradigms. GGAS is a natural extension to the GPU programming model rather than a new or hybrid programming model.
2. GGAS allows keeping the control flow for both communication and computation tasks on the GPU. Then, the CPU is no longer required to control the data transfer between GPUs and thus can completely be bypassed. This property is in particular useful in combination with *Dynamic Parallelism*, allowing launching new computing or communication kernels from the GPU domain.

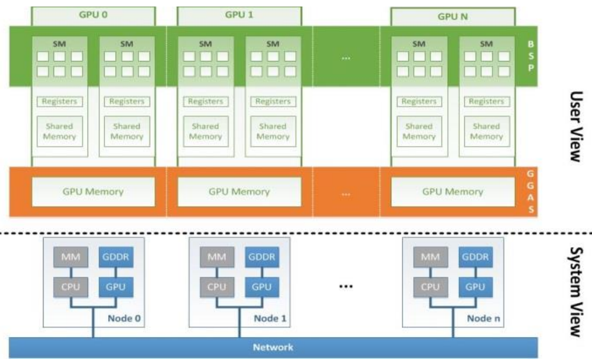

Figure 1 System and user view to a GGAS Cluster

A. *The GGAS Communication Model*

In Figure 1 the system and the user view of a GGAS cluster are shown. The bottom part shows the structure of an example GPU cluster. The cluster consists of multiple nodes, equipped with one GPU and connected with a high performance interconnect. Without GGAS, each GPU in the system is a discrete system, which only has access to its own, local device memory[2]. For access to the memory of a remote GPU, special communication functions are required. In contrast to this, the upper figure depicts how GGAS transforms this system view into a simplified user view. While resources like the SMs and the shared memory are still local to one GPU, the distributed device memories of all GPUs in the system are transformed into one global GPU address space. Each thread in the system, independent of the GPU it is running on, can access any part of this GGAS space. This allows access to remote GPU memory in the same way like to the local device memory, still with an increased latency.

Below, an example of a CUDA device function using GGAS is shown. This function writes a specific value to the

[2] To avoid confusions with the similar names of the global device memory and the global address space, we will use the term device memory for the global device memory of a single GPU. The term global address space is used for the memory region composed by shared device memories of multiple, distributed GPUs at cluster level.

GPU device memory of a remote GPU and is usually called collectively by all threads on the GPU simultaneously.

```
__device__ remote_write ( double val,
                          int GPU, int index )
{
    double* ptr = __ggas_get_ptr_of_node ( GPU );
    ptr [ index ] = val;
}
```

B. *Comparing the Message Passing and GGAS Paradigms*

The usual way to utilize a heterogeneous cluster is a hybrid programming model. The GPU is used for computation while the CPU controls the communication process. Although modern RDMA-capable hardware like Infiniband is able to transfer the data independently, still the CPU has to create send and receive requests and guarantee data consistency by synchronizing data transfer and computation.

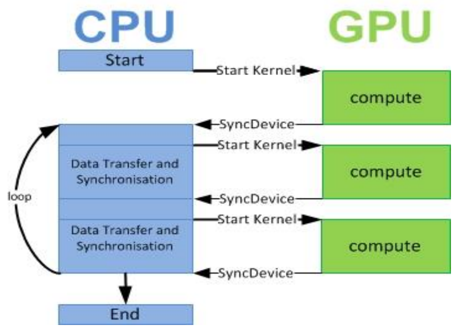

Figure 2 Control Flow for Message Passing

In Figure 2 this work flow of an iterative multi-GPU program is shown. The control flow reverts from the GPU to CPU domain any iteration to initiate communication. Although CUDA-streams and RDMA-capable hardware allow good overlapping of communication and computation, context switching between GPU and CPU causes latency issues. Especially for small messages, this can easily surpass the raw data transfer latency.

We measured the time of starting and synchronizing a simple CUDA kernel. In TABLE I the launch and synchronization times for different GPUs are shown. Each kernel is started with 32 blocks of 32 threads.

TABLE I KERNEL LAUNCH TIME

GPU	Tesla K20	Tesla K10	Quadro FX 5800	Quadro 2000
Time (us)	13.5	13.4	13.78	9.4

In contrast to this, the latencies for different message sizes transferred over Infiniband are shown in 0, demonstrating the overhead of context switches.

TABLE II DATA TRANFER TIME OVER INFINIBAND

Size (byte)	2	16	1k	4k	32k	64k
Time(us)	1.33	1.36	3.50	4.95	13.57	23.39

Also, the message passing paradigm requires a least one CPU thread to orchestrate communication. On a multi-GPU node, usually one thread or process for each GPU is used. These threads are often in a polling state, waiting for notifications of CUDA kernels or communication requests. This requires extra CPU cycles and increases power consumption.

In contrast to this, the control flow for a GPU program using GGAS is shown in Figure 3. The GPU can source and sink data transfers autonomously, so the control flow can be confined to the GPU domain. Synchronization primitives like barriers are directly implemented on the GPU.

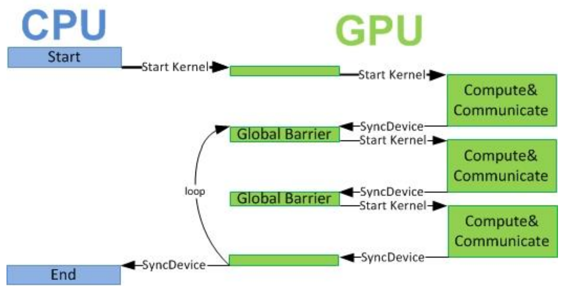

Figure 3 Control Flow for GGAS

An alternative way to avoid context switches is to create a message passing library directly running on the GPU; still this contradicts the CUDA programming paradigm. For a maximized sustained performance, threads executing within a warp have to minimize, better avoid, branch divergences. This is hardly possible using a model based on message passing.

IV. TECHNICAL IMPLEMENTATION

In this section, the technical implementation and the hardware requirements for GGAS are described.

A. *Requirements to the GPU*

To allow access to the GPU's device memory, the memory must be visible to the host system.. For NVidia GPUs, this was enabled with the new *GPU Direct RDMA* Technology, introduced with CUDA 5 [4].

This technique allows mapping GPU memory to one of the GPUs *Base Address Registers (BARs)*. These BARs are normally used for communication between host and peripheral devices. However, from the point of view of another peer device, the physical addresses are the same and either point to host memory or to the BARs of another device.

B. *Requirements to the network device*

As described above, *GPU Direct RDMA* allows a network device to access device memory like host memory. Still, this is not sufficient to enable a direct communication from a CUDA kernel and to completely bypass the host CPU, since the data transfer still has to be initiated and controlled. In common network hardware, this is done by creating work requests and exchanging notifications with the device. Even if it is theoretically possible for the GPU to perform this work, this would require massive changes to device drivers and user-space libraries of both GPU and the network device. Also, this approach is not compatible with the massively parallel GPU thread model.

So another kind of hardware is required, which allows an easier sinking and sourcing of data transfers. A *Shared Memory Engine (SME)* like the one described in [9] can meet these claims, although it was originally designed to create shared memory regions of host memory. The basic idea of such a shared memory mapper is to map a part of the memory of one node to the physical address space of remote nodes. The physical address space, where the memory of the remote notes is mapped into, is called the global address space.

The global addresses are set up in such a way that they include a coding of the target node identifier [9]. A load or store request is then encapsulated in a network packet and transferred to the target node. A store is completed by writing the payload to the requested address. For a load, the target node sends back an appropriate response. A detailed description of an example implementation can be found in [11].

C. Extending Global Address Spaces to GPUs

To create the Global GPU Address Space, we extended this concept to GPU memory. We use the *GPU Direct RDMA* feature to map GPU memory to the physical address space of the host system. The shared memory engine is now configured to forward an incoming read or write instruction to the GPU BAR. The yellow arrows in Figure 4 show such an incoming request.

Since the Shared Memory Engine is part of a peripheral device, the physical addresses of GGAS are located within the BAR of the network device. To make the shared memory accessible from the GPU, these physical addresses must be mapped into the virtual address space of the GPU. *Unified Virtual Addressing (UVA)* allows mapping a part of the host memory to the virtual address space of the GPU.

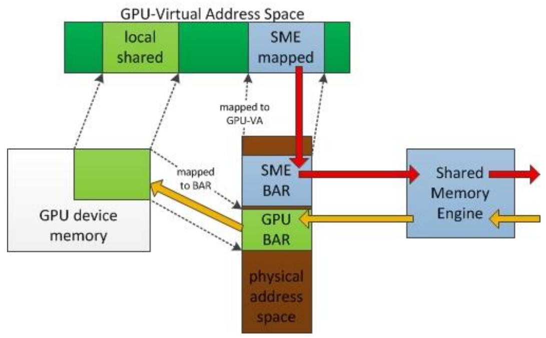

Figure 4 GGAS Mappings and Data Flows

A minor patch to the low level NVidia device driver allows extending this to physical addresses lying within a BAR of a peripheral device. A read or write instruction to a virtual address pointing to the BAR is now forwarded to the shared memory engine, which in turn forwards this request to the target node. The red arrows in Figure 4 show such outgoing requests.

V. Performance Evaluation

In this section we describe the implementation of a set of benchmarks using GGAS. We use basic latency and bandwidth tests, and a barrier as an example of synchronization primitive. The Himeno benchmark serves as a more complex example, which is based on stencil operations. Since GGAS is a new model of multi-GPU programming, we provide a more detailed description of the implementation of these benchmarks.

A. Latency Tests

Our first test evaluates latency and is based on a Ping-Pong pattern between two GPUs.

As GGAS allows respectively benefits from a collaborative use by multiple threads and the associated coalescing, we also extend this Ping-Pong test to a parallel version that starts a bundle of threads in parallel on each GPU, each one performing the tasks described above. We start up to 8192 threads, scheduled in blocks of 32 threads. Below, a code snippet for the *ping* side is shown.

```
__device__ ping ( int remote_id )
{
  int* local = __ggas_get_ptr_of_node ( ggas_id );
  int* remote = __ggas_get_ptr_of_node (remote_id);
  int ix = threadIdx.x + blockIdx.x * blockDim.x;
  volatile int tmp;
  // start collective ping by all threads
  remote [ ix ] = 1;
  // poll collectively for pong
  do {
    tmp = local [ ix ];
  } while ( !tmp );
  local [ ix ] = 0; // reset for next polling
}
```

Since GPU thread execution is non-preemptive, the possibility of lifelocks is present if more threads are scheduled than cores are available. Our experiments validate this, and for an NVidia Kepler-class K20 up to 8192 threads can be started without running into such unsafe situations. Note that this number only applies to this Ping-Pong test. The exact number of threads for a given workload depends on the overall resource usage, including shared memory and registers.

B. Bandwidth Tests

To measure the sustained bandwidth for data transfers, we implement two different tests: one is using simple read and write instructions, while the other one is relying on asynchronous *cudaMemcpy* operations, initiated directly by CUDA kernels.

1) Remote stores

Using GGAS, GPUs on different nodes simply transfer data by writing to global memory addresses. Since there is no explicit synchronization, we developed a simple protocol that uses flag-based hand-shake synchronization. Because of the low single-thread performance of a GPU, multiple threads (preferable all threads in a thread block) collaboratively communicate to foster coalescing effects and to minimize branch divergence.

2) *cudaMemcpy*

Using *Dynamic Parallelism*, an asynchronous *cudaMemcpy()* operation can be called directly by a CUDA kernel. We use this to copy the data from local to remote buffers. We start a simple kernel with only one thread that initiates a copy operation and then synchronizes. Compared to the send/receive protocol, this is a one-sided communication operation and can be compared with a *Put* operation.

C. Global Barrier Synchronization

A global address space model like the one used here requires explicit synchronization to ensure consistency. In parallel computing, a barrier is a synchronization primitive that guarantees that each thread or process reaches a specific point in its control flow before proceeding. Using GGAS, this must also guarantee that each read and write instruction to the GGAS space is completed before a thread leaves the barrier.

We use a hierarchical approach: first, all threads within a block synchronize. Second, all the thread blocks within a GPU execute a barrier. *Dynamic Parallelism* allows inter-block synchronization on a GPU [3]. For cases in which this feature is not available, Xiao and Feng [10] designed a barrier which is ideal for such GPU inter-block synchronization. Third, to synchronize distributed GPUs, an efficient and fast barrier developed for distributed shared memory architectures is used [20]. This barrier does not require atomic operations as multiple writes to a single location are avoided. It consists of two phases:

1. *Check-in phase*: Each block or thread arriving at the barrier sets a check-in flag. One master checks if the flags for all participants are set. Once all thread blocks have arrived at the barrier, the check-in phase is completed.
2. *Check-out phase*: the master sets the check-out flag for all other participants. All other participants are polling on this check-out flag for changes.

To achieve good performance, the location of the flags is important. As shown in [20], for a strong scalability remote loads should be avoided. Polling on remote locations massively increases contention and latency. We avoid remote loads by placing the check-in flags on the master GPU and the check-out flags on the respective GPUs running the slave threads. Thus, we avoid remote loads completely and instead rely solely on a push model.

D. Himeno Benchmark

We implemented the Himeno benchmark on GGAS for an application-level performance assessment. The Himeno algorithm was developed in 1996 at the RIKEN Institute in Japan, and a recent and good performing multi-GPU solution using MPI for communication is described in [12]. It focuses on the solution of 3D Poisson equation in generalized coordinates on a structured curvilinear mesh. Using finite differences, the Poisson equation is discretized in space yielding a 19-point stencil. The multi-GPU solution slices the domain along the z-direction.

We implemented a multi-GPU solution of this benchmark, using *Dynamic Parallelism* in combination with GGAS to exchange borders. During the kernel execution, border points are copied to remote buffers using write instructions to the global address space, as can be seen in the following code snippet:

```
for ( z = 1; z < ZMAX; z++ ) {
    … // do caculations
    p_new [ index ] = new_value;
    if (z == ZMAX or 1) // z is part of a border
        remote_buf [ index ] = new_value;
}
```

At the destination side, this data can be read directly from the local buffers – without further memory copies. Note, that copying data to the shared address space adds some overhead to the kernel, since every thread has to perform an extra write operation. To synchronize the data transfer, we use our barrier implementation described above.

VI. Results Discussion

Our experiments run on a test system composed of two machines with 6 core AMD Opteron 4100 series, running at 2.2GHz core frequency. Each machine is equipped with a Tesla K20 GPU. A GPU 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 the *Dynamic Parallelism* and *GPU Direct RDMA* features.

To support GGAS, we have implemented a custom network device on an FPGA that supports global address spaces. The FPGA is running only at 200MHz and provides network links with a peak bandwidth of 12.8Gbps.

We compare our results with Infiniband in combination with *GPU Direct RDMA*. Our Infiniband test system consists of two machines, each with two Intel 6-Core Xeon X5660, and directly connected using Mellanox Connect X3 Infiniband (peak bandwidth of 32Gbps). These machines also are equipped with two Kepler K20 GPUs. Since the CPUs are only used for the setup, the different CPUs have no impact on the results in this comparison.

A. Latency Results

For Infiniband it is not possible to directly sink a data transfer from the GPU, so this has to be done by the host. To measure the effect of context switches, we use a CUDA kernel to poll on the data. If the data has arrived, the polling kernel is completed and the Infiniband device is initiated to start 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 for the Ping-Pong test are shown in Figure 5. The minimal round-trip latency for GGAS is around 3.8us, so the half round-trip latency starts at 1.9us. This latency keeps constant up to 1024 parallel threads, which translates to a payload size of 4kB since every thread writes a 4 byte floating point value. Using Infiniband, the minimal round-trip latency is 6.8us and only stays constant up to a payload size of 1kB. If the time for the additional context switch is included, the minimal latency is about 20us.

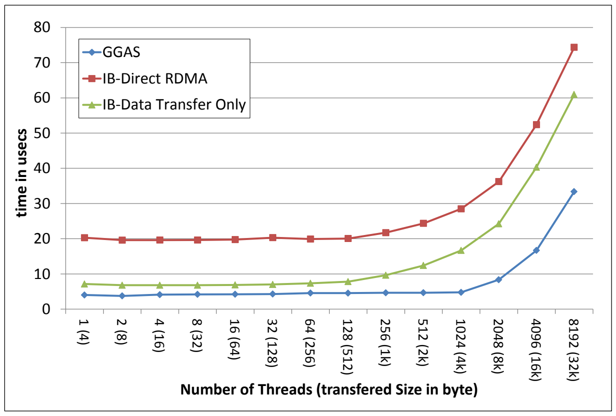

Figure 5 Ping-Pong Full Round-Trip Latency Results

These results show that especially for small data transfers GGAS significantly outperforms Infiniband. Note, that we use the IB Verbs library directly (with an additional patch to use *GPU Direct RDMA*). Using a message passing library like MPI would only add further overhead to the data transfer.

B. *Bandwidth Results*

In Figure 6 the results of the bandwidth measurement are shown. We compare GGAS again with the Infiniband IB-RDMA implementation, but also with MPI. The maximal bandwidth is around 1GB/s for both GGAS and Infiniband using *GPU Direct RDMA*. For Infiniband, this is much lower than expected, since the peak bandwidth for a data transfer between the host memories of two nodes using Infiniband is usually more than 3.2GB/s, while using MPI and host-buffered copies we achieve about 2.1GB/s. To verify our results, we repeated these measurements with Infiniband on different machines, but didn't achieve better results. This could be affiliated to the bad support of device-to-device communication by the PCI-Express system on most modern chipsets, more precisely to issues with the read functionality.

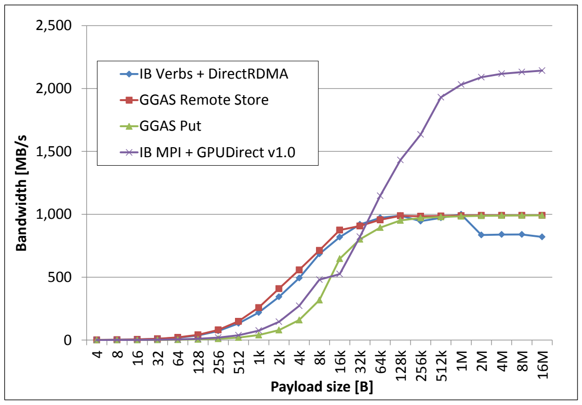

Figure 6 Results of the Bandwidth Measurements

Using GGAS remote stores for data transfer shows almost the same bandwidth like Infiniband and is slightly better for payloads up to 16kB. Surprisingly, the *cudaMemcpy()* operation performs worse even for intermediate payload sizes. However, remote stores require many threads to perform these operations and thus extra cycles. The *cudaMemcpyAsync()* operation allows a better overlap between communication and computation, since the data transfer is done by the DMA engines. The preferable communication method is depending on the workload.

C. *Barrier*

In TABLE III the results for the barrier synchronization are shown. We put the results into context by comparing to an *MPI_Barrier()* call that is executed using Infiniband, and to a *cudaDeviceSynchronize()* call that obviously only synchronizes the threads within a single GPU.

TABLE III BARRIER LATENCY

Method	CUDA+MPI	GGAS	Single GPU
Barrier Time	12.12us	5.34us	1.44us

A host-initiated *cudaDeviceSynchronize()* call in combination with an *MPI_Barrier()* call takes as twice as long as the GGAS barrier. Comparing the GGAS barrier to the single-GPU barrier shows that GGAS synchronization only adds about 3.9us to the device synchronization time, which directly translates to the Ping-Pong latency from Figure 6. Note that the number of threads per GPU has very little influence on barrier latency, so we skipped these details here.

Unfortunately, as this is the very first work on GGAS our test system only consists of two nodes, so no scalability experiments were possible. However, for two reasons we expect good scalability: first, the synchronization algorithm for local inter-block synchronization shows a very strong scaling for an increasing number of blocks [10]. Second, the same algorithm was analyzed in [20] for up to 1k CPU cores, and results demonstrated a very strong scalability.

D. *Himeno Benchmark*

To outline the strength and weakness of the GGAS model, we run the Himeno benchmark with three different problem sizes. Since our experimental system consists of two nodes, we were only able to simulate scaling tests by varying the ratio between calculation and communication. The algorithm divides the domain along the slowest direction (*z*), so adding more GPUs would reduce the problem size only along this direction, while the amount of data that has to be exchanged stays constant. Thus, we ran our benchmark for three different problem sizes of the x- and y-direction, and varying size of the z-direction.

We compare the results of the GGAS implementation to the MPI version of the benchmark [12]. Both experiments are run on our two test systems. We used MVAPICH2 v.1.7, which doesn't support *GPU Direct RDMA* yet, but shows very good scaling by using streams and overlapping communication and computation. In addition, the bandwidth is currently still better than using *GPU Direct RDMA*, for reasons described before.

In the following Figures 7-9 we compare execution times for different setups, using grid sizes of 64x64, 128x128 respectively 256x256 (x and y). For small problem sizes of 64x64 points, GGAS outperforms MPI significantly, especially for small z-sizes. For medium problem sizes of 128x128 points, GGAS is still faster than MPI, but the difference in execution time decreases. For large problem sizes (256x256 points), the GGAS and MPI implementations perform similarly, with varying but small advantages for either one depending on the z-size.

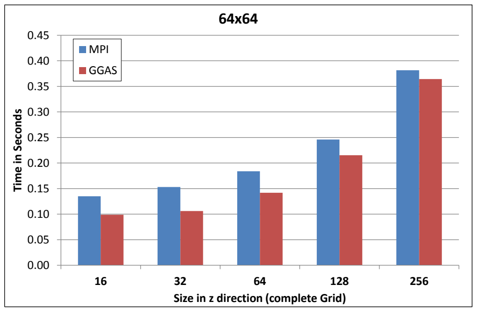

Figure 7 Himeno Benchmark with a grid size of 64x64 points (x and y)

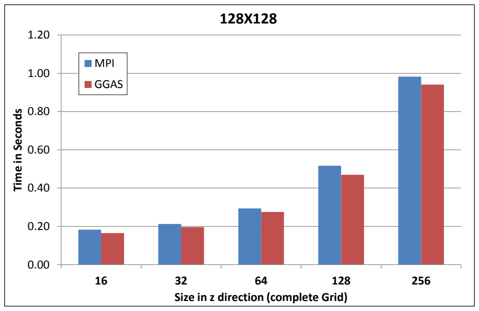

Figure 8 Himeno Benchmark with a grid size of 128x128 points (x and y)

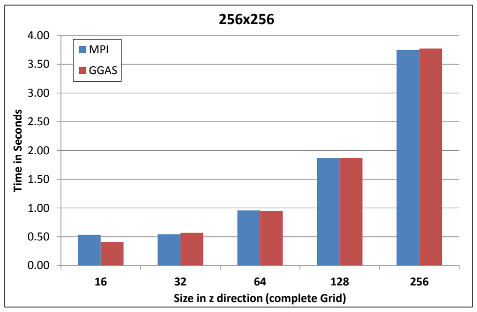

Figure 9 Himeno Benchmark with a grid size of 256x256 points (x and y)

The MPI version uses streams and asynchronous communication for overlapping of computation and communication. For smaller problem sizes, particularly for smaller z-sizes, the data transfer latency is too high to be completely overlapped. Here, GGAS profits from its very low latency for small data transfers.

For larger grid sizes along the x and y direction, more points have to transferred. This introduces also more overhead to the GPU, since more remote store instructions are required. This could be avoided by using a *cudaMemcpyAsync* operation instead. Also, for large bulk transfers the CPU could assists as off-load engine for communication tasks, which although it is beneficial for execution time increases power consumption.

E. *Discussion*

The current GGAS implementation is running on bandwidth-limited FPGAs, which yield only a peak transfer bandwidth of 12.8 Gbps. Opposed to this, MPI executes on ASIC-based Infiniband that provides transfer rates of up 32 Gbps for host to host copies, which has a huge impact on bulk transfer times.

Also, our experiments are performance-oriented with the goal to show that GGAS is either beneficial for execution time or only has little impact. What we cannot put into numbers here is that GGAS is completely in-line with the bulk-synchronous, massively parallel GPU programming model and does not require a hybrid programming approach like it is required for message-passing communication layers.

VII. RELATED WORK

Communication between GPUs is one of the main bottlenecks in GPU-accelerated high performance computing, so various work address this issue.

In [13] an FPGA-based interconnect using GPU-Direct peer-to-peer was introduced. However, while the data is directly transferred between network device and GPU, the MPI communication is still controlled by the host. The most common approach to utilize a hybrid cluster is using MPI with CUDA or other accelerator languages. In [14] Wang et al. introduce MVAPICH2-GPU, an MPI-Version for Infiniband that can use pointers to GPU memory for send and receive operations. Internally, data is transferred from GPU to host, then from source host to target host over Infiniband, and finally from host to the designated GPU. The data transfer is pipelined, and the features of *GPU Direct 1.0* are used. In [15] Potluri et al. optimize intra-node communication in MVAPICH2-GPU by using the features of *GPU Direct 2.0*. However, this is only applicable to intra-node communication. In [16] this work was optimized using *GPU-Direct RDMA*.

Another framework for programming of hybrid clusters is MPI-ACC, which is introduced in [17] by Aji et al. Its main focus is to be portable, so it not only supports CUDA but also OpenCL. Compared to this work here, it again relies on the CPU to orchestrate data movements.

DCGN [5] (Distributed Computing on GPU Networks) is 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. Note that the authors of this paper clearly state that support for direct communication among GPUs without CPU involvement is desirable.

A concept of distributed texture memory across multiple GPU nodes is introduced in [18]. It allows programs to run across multiple GPUs within a common, distributed and consistent address space. In contrast to GGAS, the underling communication and memory management is handled by the CPUs.

The approach of sourcing and sinking network traffic from an accelerator device was introduced in [19] for the Intel Xeon Phi, using Infiniband. However, the Xeon Phi has a Linux runtime-system running, which allows porting of device drivers. This approach is currently not applicable for GPUs.

Summarized, best to our knowledge no previous work allows maintaining the thread-collective nature of GPUs, instead all require assistance by the CPU for communication among distributed GPUs.

VIII. CONCLUSION

We have introduced a new approach to enable a direct communication among distributed GPUs, maintaining their thread-collective nature. Our approach allows bypassing CPUs completely for all communication tasks, confining the control flow to the GPU domain. Best to our knowledge, this is the first time that this has been achieved for GPUs. Our approach relies on global address spaces, in particular on one of their associated characteristics: transparency.

Although the first results presented here are based on only two distributed GPUs, the impact of our approach is clearly visible: compared to message exchange as state-of-the-art, it allows reducing end-to-end latencies by up to 50%, speeds up global barrier synchronization by 1.8x, and stencil computations by up to 1.67x. Note that these numbers are based on a comparison of a frequency-limited FPGA prototype with a fully-flavored Infiniband network. This speed-up comes with an increased utilization of the GPUs, which is in particular important for energy consumption. GPUs are powerful devices, but they are also power-hungry and only a maximized utilization allows for high energy efficiencies.

We'd also like to highlight that the new *Dynamic Parallelism* feature allows for the very first time a GPU to dispatch work inside the GPU as needed and also completely independently of the host CPU. Thus, combining *Dynamic Parallelism* and GGAS, all the computation and communication tasks can now be constraint to the GPU domain, rendering CPUs virtually unnecessary, except for boot-up purposes. Or, viewed from a different angle, CPUs are completely free to perform other tasks, increasing the overall energy efficiency of the system.

GGAS can improve all algorithms with irregular, small, but frequent communication patterns, since it allows low-latency, one-sided communication. In addition, the massively thread-parallel programming model of GPUs is maintained and the complexity of hybrid programming models is avoided. Future work will include other synchronization primitives, scalability optimizations and analyses, and applying these methods to other computing domains, like data warehousing, decision support and graph algorithms.

IX. ACKNOWLEDGEMENT

We gratefully acknowledge the generous support of this research effort by NVidia Corporation and Xilinx, Inc.

X. REFERENCES

[1] Lee, V.W., et al. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 451-460.

[2] Kirk, D., and Hwu, W. 2010. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann.

[3] NVidia Corp. CUDA *Dynamic Parallelism Programming Guide.* Avialable online at: http://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf

[4] NVidia Corp. *Developing a Linux Kernel Module using RDMA for GPUDirect.* Available online at http://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[5] Stuart, J.A., Owens, J.D. 2009. Message passing on data-parallel architectures. *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp.1-12, 23-29 May, 2009.

[6] Shainer, G., et al. 2011. The development of Mellanox NVIDIA GPUDirect over Infiniband—a new model for GPU to GPU communications. *Comput. Sci. Res. Dev. (2011) 26*: 267-273.

[7] Mellanox Corp. *NVIDIA GPUDirect technology - accelerating GPU-based systems*. Available online at http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf

[8] NVidia Corp. *Peer to peer and Unified Virtual Addressing*. Available online at http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf

[9] Scott, S. L. 1996. Synchronization and communication in the T3E multiprocessor. *7th International Conference on Architectural Support For Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, US.

[10] Xiao, S., and Feng, W. 2009. Inter-block GPU communication via fast barrier synchronization. *Technical Report TR-09-19*, Dept. of Computer Science, Virginia Tech.

[11] Fröning, H., and Litz, H. 2010. Efficient Hardware Support for the Partitioned Global Address Space. *International Symposium on Parallel & Distributed Processing Workshops (IPDPSW),* Atlanta, GA, US.

[12] Phillips, E.H., and Fatica, M. 2010. Implementing the Himeno benchmark with CUDA on GPU clusters. *International Symposium on Parallel & Distributed Processing (IPDPS),* April 19-23, 2010.

[13] Ammendola, R., et al. 2013. GPU Peer-to-Peer Techniques Applied to a Cluster Interconnect. *International Symposium on Parallel & Distributed Processing Workshops (IPDPSW)*, 2013.

[14] Wang, H., et al. 2011. MVAPICH2-GPU: Optimized GPU to GPU Communicationfor InfiniBand Clusters. *International Supercomputing Conference (ISC)*, June 2011.

[15] Potluri, S., et al. 2012. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. *International Parallel and Distributed Processing Symposium Workshops (IPDPSW),* May 21-25, 2012.

[16] D.K.Panda. 2013. MVAPICH2: A High Performance MPI Library for NVIDIA GPU Clusters with InfiniBand, *GPU Technology Conference*, San Jose, 2013.

[17] Aji, A. M., et al. 2012. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator Based Systems. *International Conference on High Performance Computing and Communications (HPCC).*

[18] Moerschell A., and Owens, J.D. 2006. Distributed texture memory in a multi-GPU environment. *ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics hardware* (GH '06). New York, NY, USA.

[19] Si, M., Ishikawa, Y., Tatagi, M. 2013. Direct MPI Library for Intel Xeon Phi Co-Processors. *International Symposium on Parallel & Distributed Processing Workshops (IPDPSW),* 2013.

[20] Fröning, H, et al. 2011. Highly Scalable Barriers for Future High-Performance Computing Clusters. *IEEE International Conference on High Performance Computing (HiPC 2011),* Bangalore, India.