Energy-Efficient Collective Reduce and Allreduce Operations on Distributed GPUs

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

Benjamin Klenk† and Holger Fröning†
†University of Heidelberg
Institute of Computer Engineering
Heidelberg, Germany
{klenk,froening}@uni-hd.de

***Abstract*—GPUs gain high popularity in High Performance Computing, due to their massive parallelism and high performance per Watt. Despite their popularity, data transfer between multiple GPUs in a cluster remains a problem. Most communication models require the CPU to control the data flow; also intermediate staging copies to host memory are often inevitable. These two facts lead to higher CPU and memory utilization. As a result, overall performance decreases and power consumption increases.**

Collective operations like reduce and allreduce are very common in scientific simulations and also very sensitive to performance. Due to their massive parallelism, GPUs are very suitable for such operations, but they only excel in performance if they can process the problem in-core. Global GPU Address Spaces (GGAS) enable a direct GPU-to-GPU communication for heterogeneous clusters, which is completely in-line with the GPUs thread-collective execution model and does not require CPU assistance or staging copies in host memory. As we will see, GGAS helps to process collective operations among distributed GPUs in-core.

In this paper, we introduce the implementation and optimization of collective reduce and allreduce operations using GGAS as a communication model. Compared to message passing, we get a speedup of 1.7x for small data sizes. A detailed analysis based on power measurements of CPU, host memory and GPU reveals that GGAS as communication model not only saves cycles, also the power and energy consumption is reduced dramatically. For instance, for an allreduce operation half of the energy can be saved by the reduced the power consumption in combination with the lower run time.

***Keywords*-GPUs, Power, Energy, Collective Operations, Data Transfer, Global Address Space**

I. Introduction

GPUs are powerful high-core-count devices, which are now not longer exclusively used for graphics processing but are also omnipresent in High Performance Computing (HPC). Their performance has increased dramatically in the recent years, and the introduction of GPU Computing language extensions like CUDA or OpenCL made their computational power accessible to users who are not familiar with the typical graphic-specific aspects. This has led to an adoption of a variety of non-graphical applications to GPUs.

However, GPUs only excel if they can perform their calculations in-core. This severe limitation is even amplified by two facts: first, special memory as a scarce resource is significantly limited in terms of capacity, currently not exceeding 12 GB. Second, an increasing interest in data-intensive applications (*Big Data*) is demanding for a huge amount of in-memory storage as close to the processors as possible, resulting in increased PCIe traffic due to main memory spilling.

A tightly-coupled cluster of GPUs can help to overcome this situation. However, for such a communication-centric architecture, minimal costs for communication and synchronization are mandatory to maintain scalability, performance and energy efficiency. In previous work [1] we have introduced GPU Global Address Spaces (GGAS) as a communication model that is perfectly in-line with the GPUs thread-collective execution model. GGAS not only allows increasing performance for a variety of workloads, it also avoids hybrid programming models like MPI+CUDA and complete bypasses the CPU for inter-GPU communication tasks.

Workload analysis shows that high performance applications like scientific simulations, iterative solvers and more spent a large fraction of their run time performing collective operations. It is assumed that this fraction will increase in the future due to an anticipated continuation of the core count increase for HPC systems. For instance, [2] and [3] show that MPI-based scientific simulations spend over 40% of their run time for reduction operations. Iterative solvers such as Conjugate Gradient, GMRES, and Newton, which are important components of many scientific simulations, highly rely on reductions [4].

In principle, GPUs are highly suitable for performing reduction calculations due to their thread-collective, massively parallel execution model, but they can only operate efficiently on their special memory. Additionally, for tasks like controlling network devices their execution model yields poor performance. As collective operations often result in high message counts, depending on the number of participating processes, the latter problem is even amplified. Thus, there is a need for an efficient support for collective reductions on GPUs clusters, in terms of time, energy and productivity.

An increasing number of HPC systems are becoming limited by energy consumption, for technical, economical

and ecological reasons. In this sense, the required energy to complete a certain operation might replace time as a key metric. Accelerators like GPUs are employed in HPC systems to improve both run time and power efficiency. An Intel Xeon E5-2687W Processor (8 cores, 3.4 GHz, AVX), for example, achieves about 216 GFLOPS at a thermal design power (TDP) of about 150 W, resulting in 1.44 GFLOPS/W. An NVIDIA K20 GPU is specified with a TDP of 250 W and a single precision peak performance of 3.52 TFLOPS resulting in 14.08 GFLOPS/W. Therefore moving tasks from CPU to GPU can improve performance as well as power efficiency. Instead of comparing theoretical CPU and GPU power efficiencies, we here go one step further: we compare our GGAS reduce operations against the current state-of-the-art (based on MPI) regarding actual power and energy consumption. As we will see, our analysis shows that a communication model tailored for the thread-collective execution model of GPUs can dramatically improve the execution, both in terms of energy and time.

In this paper, we advance previous work by the following contributions:

- An implementation and optimization of Reduce and Allreduce operations on GPU clusters using the GGAS communication model in combination with performance analysis in terms of run time.
- A comparison to an MPI implementation as state-of-the-art to assess performance improvements when employing a communication model tailored for GPUs.
- A detailed power and energy analysis of both GGAS and MPI implementations for allreduce operations, separating power and energy data for components including CPU, main memory, and GPU.

The reminder of this paper is structured as follows: The next section gives an overview about related work. In section 3 an short introduction into GPUs and a Global GPU Address Space (GGAS) is given. Section 4 describes different implementation strategies for collective reduce and allreduce operations on GPUs. In section 5, we evaluate the performance and compare to other communication techniques. In section 6, we analyze the power and energy consumption and compare them with MPI, 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 most common approach to utilize a hybrid cluster is using MPI with CUDA or other accelerator languages. In [5] Wang et al. introduce MVAPICH2-GPU, an MPI-Version for Infiniband that can use pointers to GPU memory. This work is optimized in [6] for intra-node communication, using GPUDirect peer-to-peer technology, and in [7] to use GPUDirect RDMA technology. In [8] optimization for the global all-to-all operation in GPU clusters is introduced and in [9] for non-continuous data transfers, using the GPU to pack and unpack data.

Another framework for GPU support is MPI-ACC, which is introduced in [10] by Aji et al. Its main focus is to be portable, so it not only supports CUDA but also OpenCL.

In [11] the OpenShmem extension to GPUs is introduced, which allows creating of shared memory segments on GPU memory. A similar approach is GPI2 for GPUs [12] [13], which describes the GPU extension to the GASPI standard. In all this work, the communication among distributed GPUs is handled by the host respectively the CPUs, resulting in performance and/or productivity losses. The main reasons include the context switches between CPU and GPU domain, and staging copies from/to main memory. Also, even though all these communication libraries support (all-)reduce communication functions, neither implementations nor results are presented.

DCGN [14] (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. Besides, the allreduce and reduce operations are not implemented. However, it is noteworthy that the authors clearly demand for a direct GPU communication technique.

In general, the optimization of allreduce and reduce operations has been a topic of various MPI based research. To name a few, in [4], [15] and [3], blocking and nonblocking allreduce and reduce operations are optimized. Also, since GPUs are highly suitable to perform parallel reductions, the in-core reduction on a single GPU has been highly optimized, like described in [16]. or [17]. These previous work can be leveraged here to optimize multi-GPU reductions.

III. Background

In the following we provide details about GPUs and CUDA and give a short description about Global GPU Address Spaces (GGAS), as it will be needed in the following sections.

A. GPUs and CUDA

A GPU is a powerful high-core-count device with multiple Streaming Multiprocessors (SMs) that can execute thousands of threads concurrently. 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 be hidden easily. 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

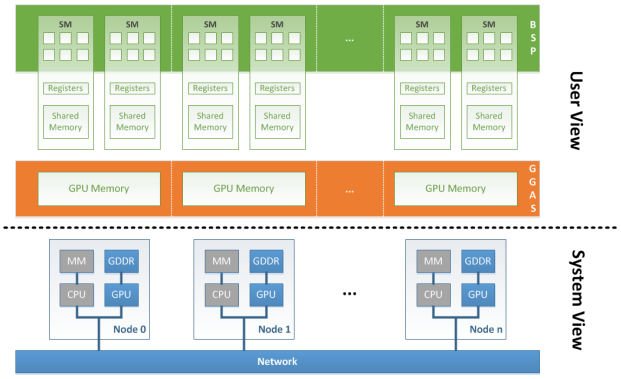


Figure 1. System and user view on GGAS cluster

scheduler is not able to handle such unaligned control flows efficiently.

CUDA is a parallel computing platform and programming model created by Nvidia, which provides a virtual instruction set to use Nvidia GPUs for computation. A more detailed description of GPUs and CUDA can be found in the excellent book by Kirk and Hwu [18]. Although we use CUDA in this work, all principles are also applicable to other GPU programming languages, including OpenCL.

A GPU has a separate, high throughput memory system that is independent from the host system. This memory is controlled from the host, not directly but special communication functions are required.

However, the new GPUDirect RDMA technology makes GPU memory accessible also for network devices, which allows a direct data transfer between multiple GPUs. We use this technique for efficient data transfer in GPU clusters and avoid copies to host memory.

B. Global GPU Address Spaces

In the bottom part of Fig. 1 the system view of a typical GPU cluster is shown. The cluster consists of multiple nodes, equipped with one GPU and connected with a high performance interconnect. Each GPU in the system has its own, local device memory[1]. To access the memory of a remote GPU, special communication functions are required. Most multi-GPU programming models are hybrid, using a GPU programming language like CUDA or OpenCL and a communication library like MPI for data transfer between GPUs. In these approaches, the data transfer is initiated and controlled by the CPU.

The GGAS model [1] distinguishes between these approaches. In the upper part of Fig. 1 the simplified user view of a GGAS GPU cluster is shown. The distributed device memory of the GPUs is transformed to one global GPU address space (GGAS), while other resources like SMs and shared memory are still local to one GPU. By this, every thread in the system can directly read and write data of a remote GPU like data on local host memory, still with more latency. Below, a short code example is shown to demonstrate the simplicity of the GGAS communication model. This function writes a specific value to the device memory of a remote GPU and is usually called collectively by all threads on the GPU simultaneously.

Listing 1. Simple GGAS remote write example

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

Summarized, GGAS provides the following benefits to hybrid programming models.

1) GGAS maintains the bulk-synchronous, massively parallel programming model of GPUs without increasing complexity by introducing message passing paradigms.
2) GGAS allows keeping the control flow for both communication and computation tasks on the GPU. Tthe CPU is no longer required to control the data transfer between GPUs and can completely be bypassed.

In common network hardware, network traffic is handled by creating work requests and exchanging notifications with the device. Even if it is theoretically possible for the GPU, this approach is not compatible with the massively parallel GPU model. Therefore, GGAS relies on a small modification in the network device which allows an easier sourcing and sinking of network traffic. A Shared Memory Engine (SME) like the one described in [19] can meet these claims. The basic idea of this hardware is the forwarding of local memory requests to a remote memory region. A more detailed description of this specialization can be found in [20].

IV. Implementing and Optimization of the Reduce Operation

Reduce and Allreduce operations are among the most commonly used global operations in high performance computing. Most parallel programming models, like MPI or PGAS based approaches like UPC, openShmem or GPI, provide implementations for this operation.

For a reduce operation in MPI, every process submits an input vector to the operation. The function performs a global reduce operation across the input vectors of all other processes in a specified group and returns the vector with the combined values back to the root process. For an allreduce operation, the result is returned to all processes. For a multi

[1]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.

threaded program, only one thread should call the reduce operation (Listing 2).

For a GPU, the concept of processes does not exist. Since GGAS is a communication model for inter-GPU communication, reduce and allreduce operations in GGAS are performed between multiple GPUs. Since GGAS maintains the massively parallel programming model of GPUs, the reduction operation is not only performed by a single thread, but by as many threads in parallel as possible, hence it should be called by all threads running on the GPU (Listing 3).

Listing 2. Allreduce on host

```
if(my_thread_id==0)
  MPI_Allreduce(...); //only one thread
```

Listing 3. Allreduce on GPU in GGAS

```
idx = threadIdx.x+blockDim.x*blockIdx.x;
GGAS_Allreduce(...); //all threads
```

A. Reduction with remote read operations

A global address space allows direct reading of remote GPU memory, so in the naïve implementation the data is directly read from the shared area, like it is shown in Listing 4 for the global sum operation. GPUs have to be synchronized to guarantee, that the data in the shared area is valid.

For larger message sizes, the reduction operation is performed by multiple threads in parallel to use the massive parallelism of GPUs.

Listing 4. Reduce with remote read

```
idx = threadIdx.x+blockDim.x*blockIdx.x;
if(idx <message_size) {
  result[idx] = 0;
  __ggas_barrier();
  for(i=0; i<ggas_nodes; i++)
   result[idx]+=pointer_to_gpu_i[i][idx];
}
```

For a reduce operation, only the root GPU executes the reduction, but allreduce is executed by all GPUs in parallel. This adds some network traffic to the application, since every GPU has to read the values from all other GPUs. On the other hand, data copies and additional synchronization between the GPUs can be avoided.

The main disadvantage of this implementation is that it requires remote reads. In Fig. 2, the bandwidths of remote read and write operations in GGAS are shown. The bandwidth of remote read operation is much lower than the bandwidth of remote write operations. This is due to two main reasons: first, the GPU can only handle a limited number of remote read requests. If the number of remote read requests is too large, it takes longer to complete a request. Second, a remote read requires a peer-to-peer read access between the GPU and the network device trough the PCIe bus, which is only poorly supported on common chip sets [7] [13]. Hence, this

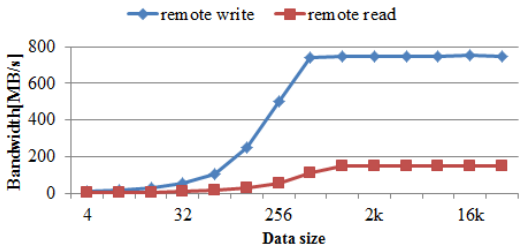


Figure 2. GGAS Bandwidth of remote reads and writes

implementation is only suitable for small message sizes. For larger message sizes, another implementation, which avoids remote read operations, is required.

B. Reduction with remote writes

To avoid remote reads, a version that relies on remote write operations was developed. This can be divided in two, respectively three steps.

1) All reduction data is transferred to one root GPU. This is referred as the *gather phase* (Fig. 3a), since the root GPU gathers the messages of all other GPUs.
2) The root GPU performs the reduction operation on its local device memory, this is referred as the *reduction* phase.
3) Last, for an *allreduce* operation, the root GPU copies the result back to all other GPUs. This is referred as the *broadcast phase* (Fig. 3b).

This approach requires at least two synchronization steps to avoid race conditions. The first one is required to guarantee that all GPUs have transferred their input data to the reduction buffer of the root GPU. The second one is required after the broadcast operation. However, although GGAS allows fast synchronization between the GPUs [1], every synchronization requires additional data transfers and adds overhead to the application. Especially for small sizes, this synchronization overhead may surpass the data transfer latency. Therefore, we implemented a version in which *all* GPUs transfer their input data to *all* other GPUs in an *allgather* manner (Fig.3c).

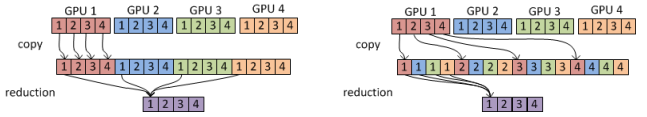
This allows all GPUs to perform the reduction operation on their local memory, so both the broadcasting operation and the second synchronization can be omitted. Still, the *allgather* operation in the first step increases the data transfer.

C. Work sharing: data distribution over multiple GPUs

For larger message sizes and a larger number of involved GPUs, the implementations described above may not scale. Either the same work performed several times on different GPUs or a single GPU performs all the reduction work while all other GPUs idle. Moreover, the gather, broadcast and allgather operations cause a lot of network traffic between all GPUs, which also can slow down the operation.

(a) Gather (b) Broadcast (c) All-Gather (d) All-to-All

Figure 3. Data Distribution and collection



(a) transfer in blocks, reduction from nonconsecutive addresses

(b) Transfer to nonconsevutive addresses, reduction in blocks

Figure 4. Data transfer strategies

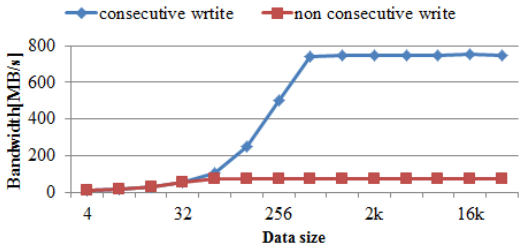


Figure 5. GGAS Bandwidth for consecutive and non-consecutive writes

To achieve a better scaling for larger messages and number of GPUs, the work is distributed among the GPUs. We implemented two work distribution strategies, which also could be combined for larger data and a larger number of GPUs.

1) Tree-based hierarchical design: For larger cluster sizes, a hierarchical design is recommended, using trees to collect and distribute the data. However, several work has be done in optimizing this trees for MPI, eg. [15] [4], and this is not the scope of this work. However, is worth to mention that these structures can also be used for GGAS and GPU clusters.

2) Work distribution: For larger data sizes, the reduction work can be distributed between the GPUs. The reduction vectors are split up into smaller blocks and then are evenly distributed between all GPUs. By analogy to MPI, this operation is called all-to-all (Fig. 3d)

Every GPU executes the reduction operation on its part of the data. For a reduce operation, the result is transferred back to the root GPU (*gather*); for an allreduce operation, the result is transferred back to all GPUs in an *allgather* manner.

D. Data transfer: optimization

The most time consuming part of the reduce and allreduce operation this is the data transfer.

All data transfer in GGAS is based on remote write and read operations, so every access to a remote memory address causes a network transfer. Therefore, two ways appear to be useful for transferring data. In the first case, the data of one GPU is written to consecutive addresses on the remote GPU memory (Fig.4a). Doing this, the reduction kernel has to read the data from nonconsecutive addresses to calculate the reduction.

In the second case, the values for one reduction operation are written to consecutive addresses (Fig. 4b). This simplifies the reduction kernel, since the GPU can read the data of one reduction operation from consecutive addresses. For both cases, the number of remote writes is the same.

To find the optimal data transfer method, we measured the bandwidth of thread-collective write operations to consecutive and nonconsecutive addresses of remote memory. For the nonconsecutive case, only every second value is written (Listing 5 and Listing 6).

Listing 5. Write value to consecutive addresses

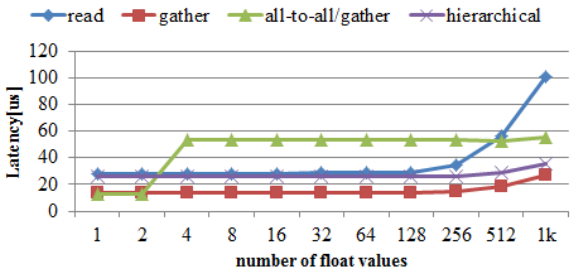
```
tid = threadIdx.x+blockDim.x*blockIdx.x
remote[tid]= local[tid]
```

Listing 6. Write value to nonconsecutive addresses

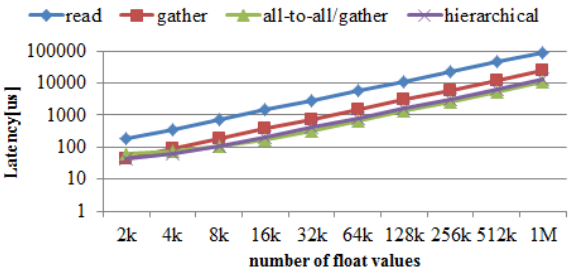
```
tid = threadIdx.x+blockDim.x*blockIdx.x
remote[tid*2]= local[tid]
```

The results of the bandwidth measurements are shown in Fig. 5. Note that GGAS is currently using an FPGA, which is running only at 156MHz and therefore provides a peak bandwidth of 9.98 Gbps. The bandwidth of consecutive writes is about ten times higher than the bandwidth of nonconsecutive writes to remote memory. This is due to the *coalescing* abilities of the GPU memory controller. The GGAS targeting network hardware allows counting of requests to the global address space. We use this feature to count the number of incoming requests for consecutive and nonconsecutive writes. In Table I the number of written values and the number of incoming requests to the hardware for both cases are shown.

For nonconsecutive writes, the number of incoming requests matches the number of the written values, while for the consecutive accesses, the GPU memory controller coalesces the request before forwarding them to the hardware. Thereby, the amount of requests to the hardware is reduced and the bandwidth is increased. So to reach optimal bandwidth, the thread collective writes should target consecutive addresses.

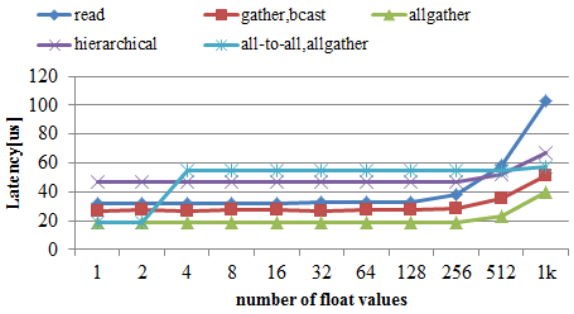


(a) small data sizes

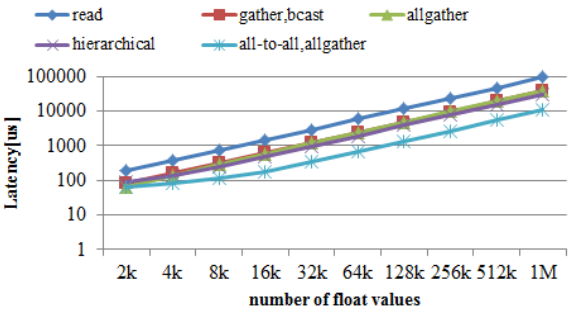


(b) large data sizes

Figure 6. Reduce Latency



(a) small data sizes



(b) large data sizes

Figure 7. Allreduce Latency

Table I
INCOMING REQUESTS FOR CONSECUTIVE AND NON-CONSECUTIVE REMOTE WRITES

Written Values	1	16	64	256	1024	4096
consecutive	1	1	4	16	64	256
non-consecutive	1	16	64	256	1024	4096

V. Performance Analysis

In this section we analyze the performance of our implementations of the allreduce and reduce operations. We run the tests on four nodes, each equipped with one K20c GPU. A GPU comes with 13 SMs, each with 64 double precision cores; or 832 cores in total. The GPUs are equipped with 5 GB GDDR5 device memory. To support GGAS, we have implemented a custom network device on an FPGA that supports global address spaces. The FPGA is running only at 156 MHz and provides network links with a peak bandwidth of 9.98 Gbps.

A. Reduce performance results

The results of the different implementations of the reduce operation are shown in Fig. 6. The naïve implementation, which uses remote read operations, performs worst for all data sizes. For small sizes, the gather version, in which all GPUs transfer the input data to the root GPU, performs best. For larger message sizes, the tree version and the distributed version, using the all-to-all operation to split the message, are performing best. Note that the message is only split, if the number of values is equal or greater than the number of GPUs. Therefore, for one or two values, the distributed version shows the same behavior like the gather version.

We only were able to run the test on a cluster with four GPUs. Therefore, we used a two-level tree for the hierarchical implementation, which adds overhead to application. For a larger number of GPUs, the tree-based approach may outperform the other approaches also for smaller messages.

B. Allreduce perfomance resluts

The results of the allreduce implementations are shown in Fig. 7. For small data sizes, the allgather-version performs best, while for larger messages the all-to-all version, where the input vector is distributed over all GPUs, performs better. For smaller data sizes, the distribution and the synchronization overhead exceeds the advantages of the work distribution. Another interesting observation is that for larger messages it only makes a small difference, if the data is transferred to one GPU and the result is broadcasted to all other GPUs, or if the complete input data is transferred to

all GPUs, which then all perform the reductions separately. However, the read version is performing worse for all data sizes, so remote read operations should not be used for allreduce and reduce operations.

Again, we are running the test on a cluster of only four GPUs. So for the hierarchical version the same assumptions apply like for the reduction operation.

C. Comparison with MPI

We compare the reduce and allreduce operations with MPI, using OpenMPI 1.6.1. The MPI library uses the same custom network device on the FPGA, but other functional units.

Although the GPUDirect RDMA technology allows the network hardware to directly read and write GPU memory, this technique cannot be used for allreduce operations without further ado, since the device memory is still not directly accessible from host. Either, the data has to be copied to the host to perform the reduction operation or the data stays on the GPUs and a reduction kernel is started to perform the reduction. We implemented the first method.

For GGAS, we used a hybrid version that uses the *allgather-method* for small messages and the *all-to-all/allgather* method for lager messages. The results are shown in Fig. 8. In MPI, the complete communication is handled by the host and no GPU kernel is started, while for GGAS, a GPU kernel is started which performs the complete reduce or allreduce operation in core. The host is not required at all.

GGAS outperforms MPI for small sizes, due neither copies to the host nor communication with the host is required. For larger messages, GGAS and MPI don't differ strongly, still GGAS is a little bit better. However, this corresponds to our previous results [1] that show that GGAS is performing best for small and medium size data transfers.

The *global sum* benchmark takes the context switches between GPU and host into account, which are required in a hybrid programming model. This benchmark determines the global sum of an array of values, which is distributed over the GPUs. First, every GPU calculates the sum of its local values, then we use the allreduce operation to determine the global sum. This operation is often required in numeric algorithms to determine a residual.

The results are shown in Fig. 9. For small data sizes for both, MPI and GGAS, the data transfer is the most time consuming part of the distributed reduction operation. However GGAS outperforms MPI. Up to a local data size of 4 kByte, the data operation on GGAS takes less than 30 μs, while in MPI around 50 μs are required. The gap between GGAS and MPI here is larger then in the pure allreduce operations, due to the context switches between host and GPU.

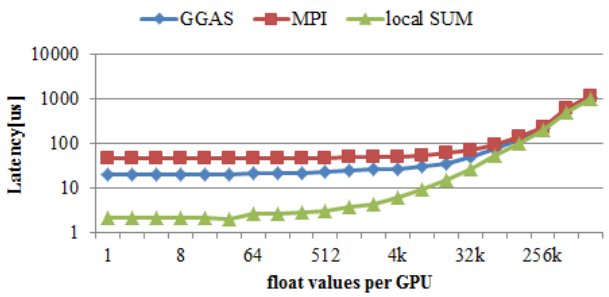


Figure 9. Calculation of a global sum

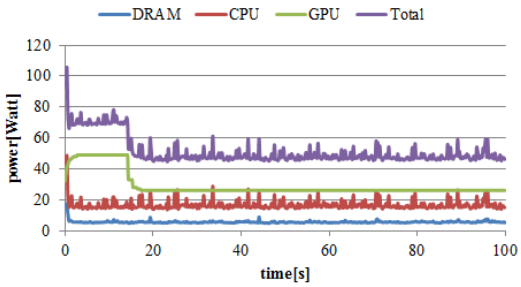


Figure 10. Idle power consumption

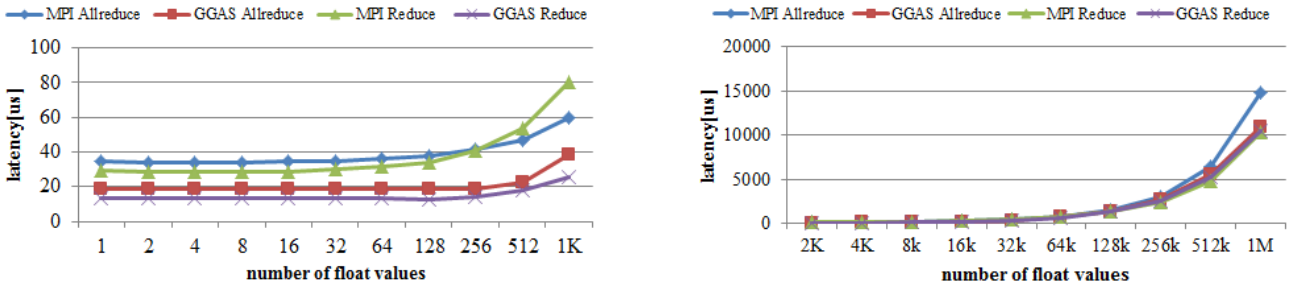
VI. Power Analysis

Power respectively energy consumption has become a crucial metric of today's HPC systems. Accelerators like GPUs typically offer an improved power efficiency (GFLOPs/Watt) compared to CPUs, at least based on theoretical peak performance. In this section, we will assess the power and energy consumption for (all-)reduce operations performed on GPUs. In particular, we compare our GGAS-based allreduce operation against an implementation based on MPI, which we assume to be state-of-the-art. Using power monitoring, we can separately measure the power consumption of components including CPU, main memory, and GPU.

A. Methodology of power measurement

Malony et al. [21] composed different approaches of measuring power on heterogeneous parallel systems. While measuring power using external devices is costly, software approaches are less accurate but no additional hardware is needed.

Nvidia offers utilities to measure power accurately using the Nvidia Management Library (NVML) whereby power is estimated with milliwatt precision within a range of ± 5 W [22]. The tool *nvidia-smi* gives information about power consumption, temperature and memory occupation but the sample rate is rather low. Another approach is to use performance counters and power models to estimate power consumption with the result that not only the total power can be measured but rather the cause of the power consumption



(a) small data sizes

(b) large data sizes

Figure 8. Reduce and Allreduce Latency, MPI vs. GGAS

can be identified. However, this needs a accurate power model of the GPU [23].

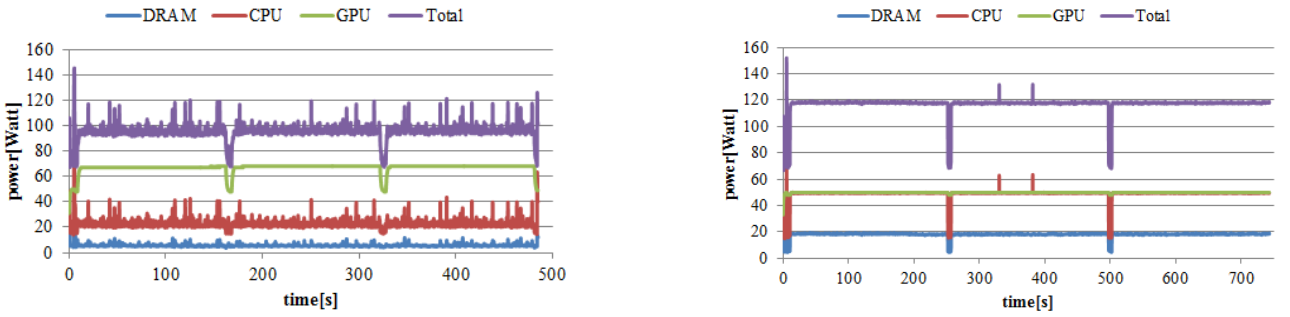
CPU and DRAM power can be estimated by reading the Running Average Power Limit (RAPL) registers provided by recent Intel CPUs. The RAPL registers are part of the Model-specific Registers (MSR) of the CPU. According to Intel the registers are updated every millisecond and power information is gathered from power sensors of the CPU introduced by Intel's Sandy Bridge architecture. The *power_gadget* tool from Intel provides a RAPL library that can be used to get power information but also to set power limits in order to cap power consumption of the CPU.

For this work a tool was created that polls the power consumption of GPU, CPU and DRAM while an application is running by using NVML resp. RAPL to obtain GPU resp. CPU power consumption data. While the NVML library provides direct information about the power consumption with milliwatt precision, the RAPL registers need to be read at least two times because RAPL provides information about energy consumption and therefore the difference between two accesses needs to be divided by the time between these two accesses. However, the power measurement also needs some power but the fraction is low compared against the power consumed by the application.

It is worth mentioning that we didn't use the CUDA primitive *cudaDeviceSynchronize* for GGAS because it would busy-wait for the GPU to finish its tasks. Instead a simple check and sleep approach was used to avoid the CPU to be busy and power consuming while the CUDA kernel is running. Since the host is not required using GGAS, this approach barely impacts the runtime of the application.

B. Results of the power consumed by reduce operations

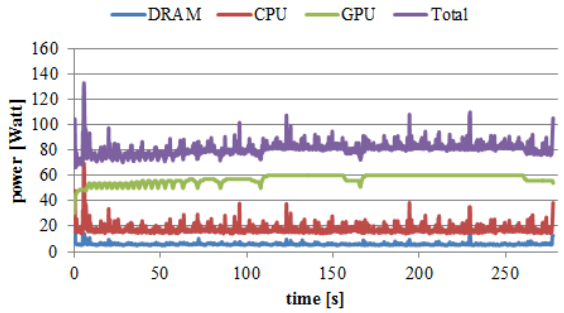
The results of the power measurement of the allreduce operation with 16,384 elements are shown in Fig. 11. Only the power consumption of one node is shown because the application load is balanced. The number of elements were chosen to ensure the runtime to be long enough in order to measure the power accurately. Besides CPU and GPU power, the DRAM and total power consumption are shown. Between thousands of iterations a break of about ten seconds is made to highlight the difference between application and idle power. As can be seen, the total power that is consumed by GGAS is about 100 Watt, whereby about 70 Watt is consumed by the GPU and about 30 Watt by the host system. For MPI, the total consumed power is about 20 Watt more than for GGAS. Actually the GPU needs only 50 Watt but CPU and DRAM consume about 40 Watt more than the



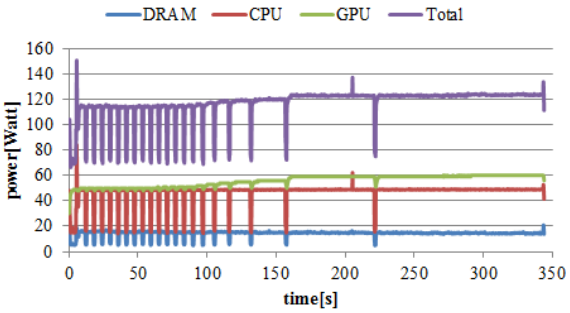
(a) GGAS

(b) MPI

Figure 11. Allreduce power consumption

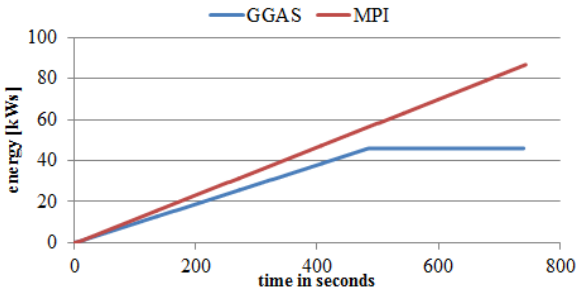


(a) GGAS

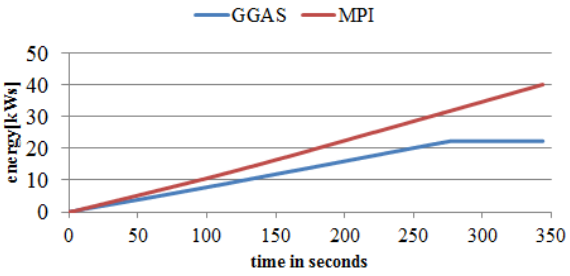


(b) MPI

Figure 12. Global sum power consumption



(a) Allreduce



(b) Global sum

Figure 13. Energy consumption

GGAS implementation. We associate this to communication that has to be handled by the host system, on the one hand the data has to be copied back and forth between host and GPU and on the other hand also between the network controller and host system. The CPU and DRAM graph show some jitter when GGAS is used. This is due to the CPU being idle. In Fig. 10 the idle power of the system is shown. While the CPU is idle some background noise can be observed. However, the idle power of the GPU drops to about 30 Watt after 15 seconds. We assume this is due to some initialization of the NVML library because this behavior is also observed when the NVidia tool *nvidia-smi* is used.

Besides the allreduce operation the global sum benchmark, which was explained in section V, was also assessed regarding power consumption. The results are depicted in Fig. 12. As can be seen, GGAS also needs less power due to the CPU being in idle mode while using MPI the CPU is still required to perform communication tasks. This leads to total savings of about 40 Watt if GGAS is used.

In terms of energy the GGAS version performs also better. Regarding the allreduce operation, besides less power consumption also the runtime is about 200 seconds shorter. The total energy consumption as the integral of the power over time is shown in Fig. 13. At the end of the MPI application, it has consumed about twice as much energy as the GGAS implementation. The energy consumed by the global sum benchmark is also shown. As can be seen, the course of the graph is simalr. As a result GGAS outperforms MPI regarding performance as well as energy consumption.

VII. Conclusion

In this work we proposed different methods to perform reduce and allreduce operations on distributed GPUs, using a Global GPU Address Space that maintains the massive parallelism of GPUs. We show that depending on the data size, different methods for data transfer and work distribution are preferable. We also learned that the coalescing abilities of the GPU memory controller can massively improve the performance of thread-collective communication.

Our approach, a Global GPU address space, allows bypassing the CPU completely, keeping the control flow on the GPU domain. Communication and reduction can be directly performed on the GPU. Therefore, the latency for allreduce and reduce operations on GPU memory can be reduced to a minimum. Compared to message passing, we get a speedup of 1.8 for small message sizes.

Also, the power consumption is reduced by 20 Watt, which, in combination with a runtime reduction results in an energy consumption of 50% of message passing. Since

the CPU is not required for the communication and synchronization, it can be sent to sleep, consuming less power without any performance misses. In our future work, we will apply these methods to other computing domains, like data warehousing, decision support and graph algorithms.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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*, 2013.

[2] R. Rabenseifner, "Automatic profiling of MPI applications with hardware performance counters," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 1999, pp. 35–42.

[3] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations inMPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[4] M. G. Venkata, P. Shamis, R. Sampath, R. L. Graham, and J. S. L. Ladd, "Optimizing blocking and nonblocking reduction operations for multi core clusters: Hierarchical design and implementation," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, 2013.

[5] 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.

[6] 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*. IEEE, 2012, pp. 1848–1857.

[7] 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*, 2013.

[8] A. K. Singh, S. Potluri, H. Wang, K. Kandalla, S. Sur, and D. K. Panda, "MPI alltoall personalized exchange on GPGPU clusters: Design alternatives and benefit," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 420–427.

[9] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, "Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 308–316.

[10] 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)*. IEEE, 2012, pp. 647–654.

[11] 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.

[12] 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.

[13] L. Oden, "GPI2 for GPUs: A PGAS framework for efficient-communication in hybrid clusters," in *Parallel Computing - ParCo2013, International Conference on*. IOS, in press.

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

[15] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," in *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*. IEEE, 2007, pp. 1–10.

[16] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *NVIDIA Developer Technology*, vol. 2, 2007.

[17] M. Harris, "Mapping computational concepts to gpus," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 50.

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

[19] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *ACM SIGPLAN Notices*, vol. 31, no. 9. ACM, 1996, pp. 26–36.

[20] H. Fröning and H. Litz, "Efficient hardware support for the partitioned global address space," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–6.

[21] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with GPUs," in *Parallel Processing (ICPP), 2011 International Conference on*, 2011.

[22] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson, "Power aware computing on GPUs," in *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, 2012.

[23] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.