Energy-efficient Computing on Distributed GPUs using Dynamic Parallelism and GPU-controlled Communication

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

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

***Abstract*—GPUs are widely used in high performance computing, due to their high computational power and high performance per Watt. Still, one of the main bottlenecks of GPU-accelerated cluster computing is the data transfer between distributed GPUs. This not only affects performance, but also power consumption. The most common way to utilize a GPU cluster is a hybrid model, in which the GPU is used to accelerate the computation while the CPU is responsible for the communication. This approach always requires an dedicated CPU thread, which consumes additional CPU cycles and therefore increases the power consumption of the complete application.**

In recent work we have shown that the GPU is able to control the communication independently of the CPU. Still, there are several problems with GPU-controlled communication. The main problem is intra-GPU synchronization, since GPU blocks are non-preemptive. Therefore, the use of communication requests within a GPU can easily result in a deadlock. In this work we show how Dynamic Parallelism solves this problem. GPU-controlled communication in combination with Dynamic Parallelism allows keeping the control flow of multi-GPU applications on the GPU and bypassing the CPU completely. Although the performance of applications using GPU-controlled communication is still slightly worse than the performance of hybrid applications, we will show that performance per Watt increases by up to 10% while still using commodity hardware.

I. Introduction

During the last years, graphic processing units have gained high popularity in high performance computing. Programming languages like CUDA, OpenCL or directive-based approaches like OpenACC make the features of GPUs available for developers that are not familiar with the classic graphical aspects.

Therefore, GPUs are deployed in an increasing number of HPC systems, especially since energy efficiency becomes more and more important due to technical, economical and ecological reasons. In particular, the first 15 systems of the Green500 from June 2014 are all accelerated with NVIDIA Kepler K20 GPUs [1]. For example, an Intel Xeon E5-2687W Processor (8 cores, 3.4 GHz, AVX) 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 $GFLOPS/W$.

New features like CUDA Dynamic Parallelism help the GPU to become more independent of the CPU by allowing the GPU start and stop compute kernels without context switches to the host. By this the CPU can be relieved from this work, helping to save power for GPU-centric applications.

GPUs are powerful, scalable many-core processors, but they excel in performance only if they can operate on data that is held *in-core*. Still, GPU memory is a scarce resource and also due to this, GPUs are deployed in clusters. However, communication and data transfer is one of the main bottlenecks of GPU-accelerated computing. Since we are now facing an area in which communication and data transfers dominate power consumption [2], it is necessary not only to optimize the computation of GPU-accelerated applications with regard to energy and time, it is even more important to optimize communication aspects.

Applications that are running on distributed GPUs normally use a hybrid-programming model, in which computational tasks are accelerated by GPUs, while data transfer between the GPUs is controlled by the CPUs. This approach requires frequent context switches between CPU and GPU, and for the whole execution time a dedicated CPU thread is required to orchestrate GPU computations and GPU-related communication. This CPU thread requires additional power and therefore increases the energy consumption of the complete application, preventing the CPU from entering sleep states.

In recent work [3] we introduced a framework that allows GPUs to source and sink communication requests to Infiniband hardware and thereby completely to bypass the CPU. So far, this approach does not bring any performance benefits, but losses. This is caused by the work request generation on GPUs, which shows a much higher overhead compared to CPUs. Still, this technique allows to keep the control flow of an multi-GPU application on the GPU, avoiding context switches between CPU and GPU and relieving the CPU from communication work.

Thus, GPU-controlled communication can help to reduce

the power consumption of GPU-centric distributed applications by consolidation computation and communication tasks on the GPU. In this work, we will focus on how *Dynamic Parallelism* can further improve the effectiveness of this approach. In particular, we address the issue of intra-GPU synchronization, which is up to now a severe bottleneck for GPU-controlled communication.

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

- A discussion of different intra-GPU synchronization methods for GPU-controlled communication
- A discussion of the benefits of Dynamic Parallelism for GPU-controlled communication
- A detailed performance and energy analysis of GPU-controlled and CPU-controlled communication methods for distributed GPUs

The rest of the paper is organized as follows. The next section gives a short overview about related work. Section three provides some background information on GPUs, communication in GPU clusters and power measurement. In section four, the benefits of Dynamic Parallelism are presented, while in section five the general work flow of a hybrid-multi GPU application is discussed. In section six different concepts of GPU-controlled communication using Dynamic Parallelism are presented and in section seven and eight, a detailed analyzed of performance and energy consumption of these concepts are presented, before we conclude in section nine.

II. Related work

A lot of work exists in the area of energy-aware heterogeneous processing, however, most of this work concentrates on computation and disregards communication and data transfer for distributed systems.

In [4], a global GPU address space [5] is used to perform reduce and allreduce collectives completely on the GPU. The global GPU address space allows the GPU to control the communication independently of the host CPU, resulting in an increased energy efficiency for these collective operations. In principle, a similar approach is used here. However, the work in [4] concentrates on the implementation of reduce and allreduce operations and requires a special hardware unit to enable inter-GPU communication. The approach in this work is based on commodity network hardware instead, in particular Infiniband.

Paul et al. contribute an excellent work optimizing energy management for heterogeneous processors and HPC workloads [6], but only investigate intra-node optimizations. In [7], the impact of clustered GPUs on energy efficiency is explored, but communication models are not part of this analysis.

GPU-controlled communication was first topic in [8] from Owens et. al. They introduced a message-passing interface for GPUs, in which communication requests are forwarded from the GPU to the host CPU. In particular, they highlight the need for CPU bypassing and mention the problem of deadlocks with regard to GPU-controlled communication.

In our previous work [3], we introduced an Infiniband Verbs implementation for GPUs, which forms the basis of this work. This framework allows GPUs to source and sink communication request to Infiniband hardware while complete bypassing the host CPU. In [9], different Put/Get APIs for GPUs where analyzed in terms of performance but not power consumption.

Optimizing the communication in heterogeneous clusters with the scope of performance was also topic in various related work, however they all rely on CPU-controlled communication. The most common approach is using MPI in combination with CUDA or another accelerator language. In [10], Wang et al. introduce MVAPICH2-GPU, an MPI-Version for Infiniband that can handle pointers to GPU memory, which is optimized in [11] by using GPUDirect RDMA technology. In [12], the OpenSHMEM extension for GPUs is introduced. A similar approach is GPI2 for GPUs [13], which describes the extension of the GASPI standard to support GPUs.

Dynamic Parallelism is also an important topic in current research. In [14], the impact on clustering algorithms is examined. They show that it can positively impact performance of the divisive hierarchical clustering algorithm, while the performance of K-means is slowed down. In [15], [16], the implementation of different clustering and graph algorithms on GPUs with Dynamic Parallelism are described. In [17], an alternative to Dynamic Parallelism for nested parallelism is introduced, due to the low performance of Dynamic Parallelism for short loops.

III. Background

This section provides background information on GPU computing, GPU communication models and power measurement, as it will be needed in the following sections.

A. GPU Computing

The architecture of a modern GPU is optimized for highly data-parallel execution, with thousands of threads operating in parallel. A GPU is composed of multiple Streaming Multiprocessors (SMs), who themselves are composed of a large number of computing cores. Compared to complex modern CPU cores, these cores have a very simple structure and are optimized for a high throughput of floating point operations.

The threads of a GPU are organized in blocks. These thread blocks are tightly bounded to the SMs, but the GPU scheduler does not schedule the threads in blocks or with single-thread granularity. Instead, a block is divided in so-called warps, which have the size of 32 threads on modern GPUs. All the threads of one warp are scheduled at the same time and follow the same instruction stream. A divergent branch within one wrap results in a sequential execution of the code, since all other threads simply block for such non-coherent branches in the code. Therefore, different branching within the threads of a warp result in significant performance losses.

1) Dynamic parallelism: The feature of *Dynamic Parallelism* [18] was introduced with CUDA 5 for Nvidia GPUs with compute capability 3.5 and higher. Dynamic Parallelism

allows the launching of new compute kernels directly from the GPU. Before, only the host CPU was allowed to start new kernels on the GPU. Its main objective is to lower the overhead of starting and synchronizing kernels. However, several studies have shown that starting and synchronization of these kernels on the GPU still has got a large overhead, which is referred as the *Dynamic Parallelism overhead* [17]. Therefore, it not necessarily helps to improve the performance, especially if it is used to run multiple small kernels concurrently on the GPU. However, for our work the main benefit is that it provides an easy way for inter-block synchronization on the GPU, which was previously not supported and required special synchronization primitives, like for example inter-block barriers, which are described in [19]. Such an inter-block barrier introduces eventually unnecessary synchronization and thus overhead, and also can lead to deadlocks.

B. Communication and data transfer between distributed GPUs

This section gives a short overview about communication and data transfer methods for distributed GPUs and classifies the methods used in this work. When analyzing such communication methods, we have to distinguish between *data transfer* and *communication control*.

1) Data transfer: can either been direct or staged. A direct data transfer means that data is directly transferred between the memories of two GPUs, while for a staged data transfer the data is buffered in host memory. For a staged data transfer, the sending GPU first copies the data to the host memory, where network device reads the data and transfers it to the host memory of the remote node. Here, the receiving GPU copies the data to its own device memory. GPUDirect RDMA [20] allows a direct data transfer between distributed GPUs, since it enables RDMA capable network devices to directly read and write GPU device memory.

A direct data transfer is ideal for short messages. However, for larger messages some issues with current PCIe implementations limit the performance, and therefore often staged copies are used [11], [13].

2) The communication control: describes which unit *controls* the data transfer, therefore sources and sinks communication requests and synchronizes with the remote side. For heterogeneous computing nodes like relevant for this work, communication control can be either performed by GPUs or CPUs.

The most common approach is to use a *hybrid model*, in which the CPU controls the communication while the GPU is only used to accelerate compute-intensive tasks. Good examples for this are CUDA-aware MPI versions, so that MPI operations can use GPU pointers as source and destination [11], [21]. A similar but one-sided approach is GPI2 for GPUs [13]. Both use direct data transfers for small messages and a staged data transfer for large messages to bypass the previously mentioned PCIe performance issue. However, such hybrid approaches result in frequent context switches between host and GPU, and require a dedicated host thread to control communication.

To avoid both drawbacks, *GPU-controlled communication* is essential. In our recent work we enabled the GPU to source communication requests to Infiniband Hardware by completely bypassing the host CPU [3]. To allow this, first an Infiniband communication environment is set up by the CPU. Parts of this environment are then mapped into the GPU address space. This mapping then allows GPU threads to access the resources required to issue work request to Infiniband hardware and to ensure completion of these work requests.

However, we showed that this approach currently does not bring any performance benefits compared to a hybrid approach. We identified two main reasons: first, the inter-block synchronization on the GPU, and second the massive overhead for generating work requests by GPU threads. In this work here, we analyze how Dynamic Parallelism can help to address the first of this problems.

C. Power measurement

In this work, we use a software approach to measure the power consumption, using the integrated power measurement facilities of the used hardware components. Most recent Intel CPUs support Running Average Power Limit (RAPL) registers [22], which report power consumption of CPU and DRAM. The counterpart for Nvidia GPUs is the Nvidia Management library (NVML) [23], which reports the power drawn by the GPU add-in card. Both techniques use power modeling to estimate the current power consumption of the respective hardware components, and are widely accepted as accurate. The advantage of this methodology is that it a relative fine grain instrumentation with a sampling period of around 200 ms is possible, and that single components like CPU, DRAM and GPU can be characterized.

An alternative method would be the use of external power meters, either specialized for individual components like the add-in cards or for power outlets. The main advantage of this method is the accuracy. However, using the power meters for outlets do not allow a break down to the individual components, while instrumenting each component with a individual power-meter is either not practical or obstructive.

Note that the software approach leaves other components like the network device uncovered. However, communication patterns to not differ for the different communication approaches used here, and network power consumption is typically independent of the actual traffic, as it is dominated by (de-)serializers that employ serial link coding and embedded clocking.

IV. Dynamic Parallelism and power saving

Before we look at the combination of Dynamic Parallelism and GPU-controlled communication, we first analyze if and how an application on a single node can benefit from Dynamic Parallelism. Relieving a CPU from work (i.e., GPU on-loading) can lead to more CPU idling and thus allows the CPU to enter low-power states, saving energy.

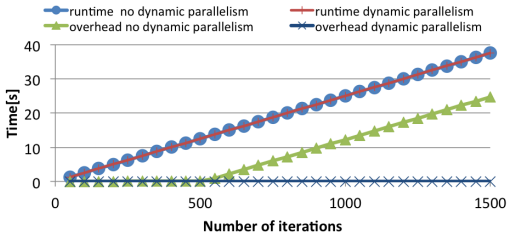


Fig. 1. Run time and CPU overhead for the Himeno benchmark

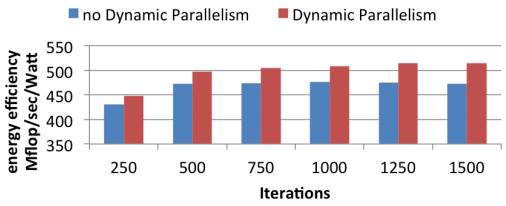


Fig. 2. Energy efficiency of the Himeno benchmark on a single GPU for different number of iterations, using host kernels and dynamic parallelism

We use the Himeno benchmark to illustrate how Dynamic Parallelism can be used to achieve this. The implementation of a good performing single- and multi-GPU version of this benchmark can be found in [24], however we first focus on the single-GPU version. We run the Himeno benchmark with a grid size of $512x512x256$ points and vary the number of iterations. For every iteration, two CUDA kernels are added to the stream, one for the computation of the grid and one to compute the residuum. The first version of this benchmark does not use Dynamic Parallelism; the host CPU adds all compute kernels to a single stream. Then, this stream is synchronized with *cudaStreamSycnhronize*.

In the second version, the host CPU only starts a single kernel (with a only one thread block), the *master kernel*. The master kernels then starts the compute kernels for every iteration using Dynamic Parallelism. The host CPU synchronizes with this *master kernel* using *cudaStreamSynchronize*. We measure the complete execution time of the benchmark, therefore the time from issuing the kernel(s) until *cudaStreamSynchronize* completes. In addition, we measure the time that the hosts spends in adding kernels to the stream, reported as *CPU overhead*. During this time, the CPU cannot be used for other tasks, while for

$$t_{idle} = t_{execution} - t_{overhead}$$

the CPU is it potentially available for other tasks or can enter a sleep state.

The results are shown in Figure 1. The execution time of the benchmarks differs so little that the graphs are hardly distinguishable. For a small number of iterations, the CPU overhead is negligible for both cases. However, for more then 500 iterations the CPU overhead without Dynamic Parallelism starts increasing linearly while with Dynamic Parallelism is remains barely noticeable.

The reasons for this is probably the queue size of the streams. Since the CPU overhead rises for more then 500 iterations and two kernels are added to the stream for every iteration, it seems that this queue is limited to about 1000 entries.

Figure 2 reports the impact of employing Dynamic Parallelism on power consumption. We use a synchronization method as shown in Listing 1 for both cases with a decreased polling rate, as frequent polling can prohibit such sleep states. Since the execution time of the iterated kernels is large enough, there are no performance penalties. The results in Figure 2 show that using Dynamic Parallelism the energy efficiency is significantly larger. The difference increases with the number of iterations. Therefore, Dynamic Parallelism seems to be a feasible method to improve energy efficiency for GPU-centric applications.

Listing 1. Allowing the CPU to sleep while waiting for completion

```
while( cudaStreamQuery(stream) == cudaErrorNotReady)
            usleep(5000);
```

V. Distributed Computing with Multiple GPUs

In the previous section we have shown that Dynamic Parallelism allows keeping the control flow of an application on the GPU, thereby reducing the overhead for the CPU significantly and saving power for GPU-centric applications.

While this is a power optimization for GPUs within a single node, we now focus on distributed GPUs, in particular the mandatory communication among them. Common practice is a hybrid programming model, in which the CPU controls the communication. Using techniques like GPUDirect RDMA, data can be directly transferred between GPU memories, however such techniques still rely on the CPU to control the communication. In this case CUDA Dynamic Parallelism is of little use since frequent context switches between GPU and CPU are required anyway.

In Figure 3 the control flow of a hybrid multi-GPU implementation of the Himeno benchmark is shown. The communication is controlled by the CPU and the GPU is only used for computation. In the multi-GPU version, like described in [24], the three-dimensional domain is sliced along the z-direction. Each GPU has to exchange the top and the bottom border with its direct neighbors.

To overlap communication and computation, two kernels are used: *kernel_top* and *kernel_bottom*. We used CUDA *events* to synchronize these kernels with the host process. The events are inserted into the stream right after the compute kernel. Another method would be the use of two streams, as described in [24]. This method is mandatory if no CUDA-aware communication library is used and explicit data transfer between GPU and host is required, to overlap this data transfer with the computation. However, since we are using a CUDA-aware communication library, event synchronization turned out to be most efficient.

In every iteration, a host thread first starts the data transfer of the bottom boundary (*send_top*) and then the compute kernel for the top part of the grid (*top_kernel_start*). The data transfer itself is handled by the network device and overlapped with

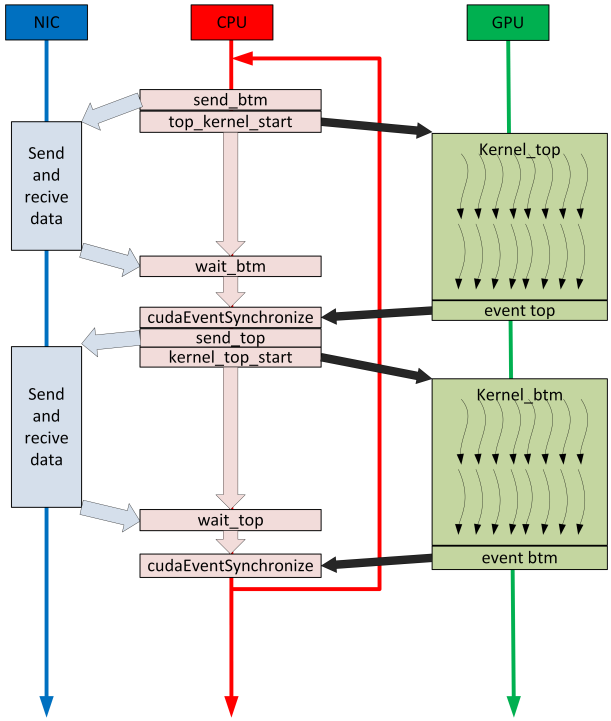


Fig. 3. Control flow of a hybrid application

the computation on the GPU. Theoretically, now the CPU can be used for other work or set to sleep, while the compute kernel is running on the GPU and the network device handles the data transfer. Still, before the next compute kernel can be started, the host CPU has to ensure that the remote boundaries are updated (*wait_btm*). Similarly, before the CPU starts the data transfer of the top boundaries (*send_top*), it has to ensure that the top kernel has completed. Therefore, usually a host thread is delegated only to control the flow between network device and GPU for such multi-GPU applications.

The communication can be handled by MPI or another CUDA-aware communication library. If MPI is used, the *wait* function corresponds to a MPI receive function. Here we use GPI2 for GPUs, since we also use a one-sided communication scheme on the GPU. The benefit of this communication scheme is that it adds less overhead to the communication and therefore provides a very thin communication layer.

To synchronize the one-sided communication, we used GPI2 *weak synchronization* [25], which uses a flag mechanism to inform the remote side of the completion of a remote write. Then, the wait routine corresponds to polling on this notification flag.

However, independent of the underlying communication framework, as long as the communication is controlled by the CPU, Dynamic Parallelism cannot be used to handle multiple iterations on the GPU and to release the CPU from synchronization tasks. Instead, a CPU thread is required to control the flow between network device and GPU. The only way to avoid this is to move communication tasks to the GPU.

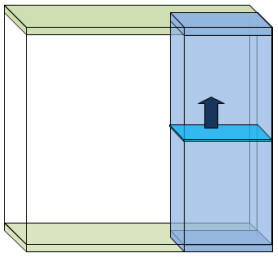


Fig. 4. 3-D Stencil, thread blocks process columns of the domain, boundaries are on top and bottom

VI. GPU-controlled Communication

In our previous work [3], we have shown that is possible to enable the GPU to source and sink communication requests to the network device by completely bypassing the host CPU. However, apart from performance, there are several problems with this approach which are related to the GPU programming model.

Normally, a GPU kernel is started with more threads and blocks then can be scheduled concurrently on the GPU. But once a thread block is scheduled, it cannot be preempted until it has completed. Also, it is not predictable which block is scheduled at which point in time. This behavior, combined with GPU-controlled communication, can easily result in a deadlock: on both GPUs thread blocks waits for remote data, but the thread blocks that actually transfer the data cannot be scheduled since the waiting blocks are blocking the GPU resources.

This problem is best illustrated by describing the implementation of a simple benchmark. We again use the Himeno benchmark and the GPU implementation as described in [24]. In every iteration all points of the grid are updated using the neighboring points. On a single GPU, the 3-D domain is processed by two-dimensional thread blocks. These thread block walk through the domain from the bottom to the top of the 3-D grid, as shown in Figure 4.

In the simplest case for multiple GPUs, the complete domain is sliced along the z-direction and then distributed between the GPUs. Then the boundary values, which have to be exchanged, are on the top and bottom of the grid, like shown in Figure 4 in green. That implies that all GPU thread blocks update a part of these boundaries – and that all thread blocks require data from a remote GPU. Furthermore, all thread blocks also require data points that are processed by the neighboring thread blocks, since the stencil computation depends on neighboring points from all six directions. For this, inter-block synchronization is mandatory but the CUDA execution model only defines such a synchronization semantic when kernels are finished. This is the the crux of the matter for GPU-controlled communication.

Normally, this inter-block synchronization is provided by starting a new kernel for every iteration and adding these kernels to the same stream. Since thread blocks are non-preemptive, starting a kernel with more blocks then can be run concurrently on the GPU can result in a deadlock, if

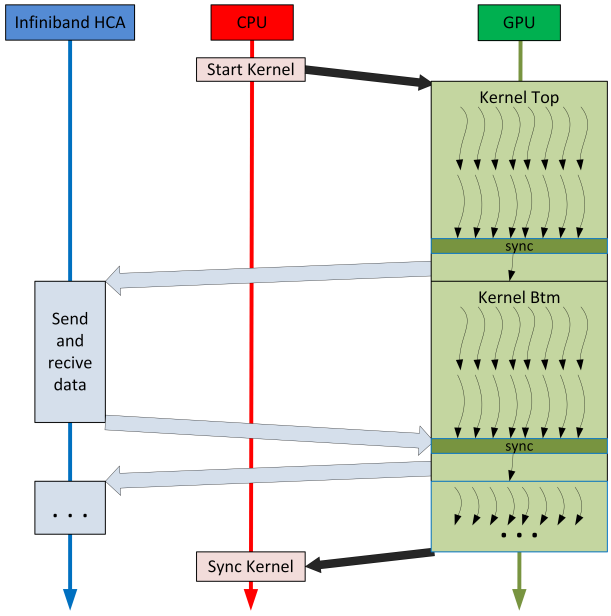


Fig. 5. Control flow of an application with communication control on the GPU, inter-block synchronization within a CUDA kernel

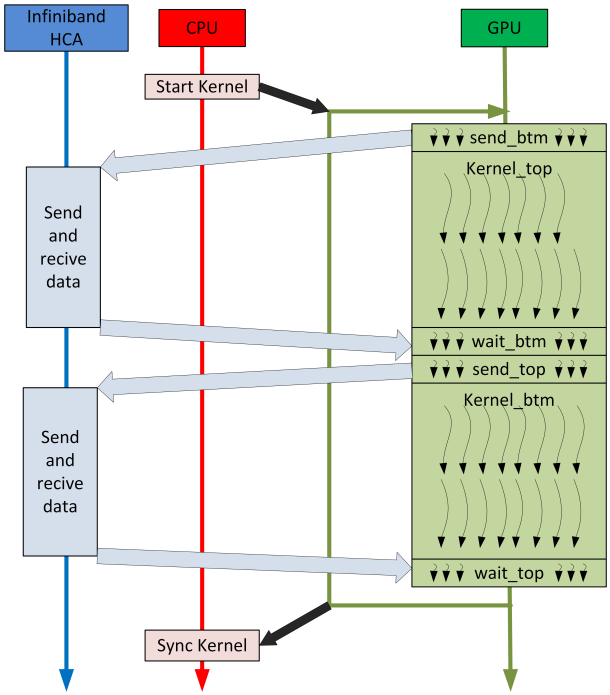


Fig. 6. Control flow of an application with communication control on the GPU, using stream synchronization

another method of inter-block synchronization is used. In the following, three possible solutions for this problem are presented and the benefits of using Dynamic Parallelism with these methods are explored.

A. *In-Kernel synchronization*

In the first approach, a communication request is sourced and sinked directly by the compute kernel. The control flow of such an application is shown in Figure 5. Using this approach, the required inter-block synchronization within the compute kernel can result in a deadlock.

One possibility to avoid deadlocks is to start only as many threads on the GPU as can be scheduled concurrently . This technique is also called *persistent threads*. Then, for intra-GPU synchronization a barrier as described in [19] can be used. All blocks on one GPU are first synchronized using an inter-block barrier. After this, one block can start the data transfer of the complete boundary before the next iteration is started. However, in [26] was shown that the use of persistent threads on GPUs results in performance losses in many cases. In particular, the CUDA execution model relies on parallel slackness to hide long-latency events, and persistent threads dramatically reduce the amount of parallel slackness. Therefore this is often not a suitable solution.

Another possible solution is to organize the communication in a way that ever block is responsible for its part of the boundary. This solution results in many small data transfers, since every block starts a data transfer. Due to the two-dimensional structure of the blocks, the data of one block is also non-contiguous in memory, resulting in a large number of small data transfers or in additional work due to data packing and unpacking. Therefore, a solution is required that allows starting of as many thread blocks as required, but still provides an intra-GPU synchronization to avoid many small data transfers.

We propose to use an atomic counter as a solution to this problem. For every iteration, a new kernel is started to guarantee inter-block synchronization. To overlap communication and computation, it is also possible to use more then one kernel, for example one for the top and one for the bottom part of the grid like shown in Figure 5. The kernels are started with as many thread blocks as required to process the grid.

The GPU schedules the thread blocks of one kernel until all have completed. Only then a new kernel in the same stream is started. When the last block has finished the computation, all points in the grid are updated and the boundaries can be transferred. Therefore, this last block should start the data transfer. Since it is not predictable which block is scheduled when, we use an atomic counter to determine the last thread block. Every iteration, each block increments the counter using an atomic operation. The last block starts the communication and resets the counter.

To synchronize the distributed GPUs, a flag-based notification mechanism similar to the *weak synchronization* for CPU-controlled communication is used. Immediately after a remote write operation, which transfers the boundaries, a notification is written to the remote GPU to signal the availability of new data. So, at the beginning of every iteration all blocks have to wait for this remote notification before the computation is continued. This polling on a notification can efficiently be done by all threads in parallel.

B. Stream synchronization

The second approach uses stream synchronization to synchronize communication and computation on the GPU. In Figure 6 the control flow for the stencil code using this approach is shown. Here, the communication functions *send_top/btm* are added to a stream as independent kernels, so now these functions are implemented as CUDA kernels and not as device functions. The same applies for the synchronization functions (*wait_top/btm*).

To overlap data transfer and computation, two pure compute kernels (*kernel_top* and *kernel_btm*) are used. For every iteration, at first a communication kernel (*send_btm*) is added to the stream. This kernel starts the data transfer of the bottom boundary, which is then handled by the network device. Next, the compute kernel for the top part of the grid is added to the stream (*kernel_top*). The execution of this kernel is overlapped with data transfer of the bottom boundaries. The next kernel that is added to the stream is again a communication kernel (*wait_btm*), which waits until the bottom boundaries are updated. As we use a flag mechanism, in this kernel a thread polls for a notification flag. The same procedure is now repeated with the data transfer of the top boundaries (*send_top* and *wait_top*) and the computation of the bottom part of the grid (*kernel_btm*). Since the order of the kernels in one stream is maintained, the data transfer is not started until the local borders are updated; similarly a new kernel is not started until the remote borders are updated.

C. Synchronization with Dynamic Parallelism

The previously described approaches actually do not require Dynamic Parallelism, since all kernel can theoretical also be submitted from the host. However, to keep the overhead on the host as small as possible, we in addition use Dynamic Parallelism for these methods. Then, the host CPU starts a single kernel (*control block*), which then starts the communication and compute kernels on the GPU. This method, however, requires Dynamic Parallelism and cannot be realized without some of the features.

Basically, the idea is to transfer the hybrid approach to the GPU, using dynamic parallelism. However, the hybrid approach described in the previous section cannot be directly transferred to the GPU, since neither *cudaEventSynchronoize* nor *cudaStreamSynchronized* are supported on the GPU. Instead, we use *cudaDeviceSynchronize* to synchronize the compute kernels. The work flow of this approach is shown in Figure 7. Again, two pure compute kernels are used. In contrast to the previous approach, the communication functions are not implemented as CUDA kernels but as device functions, which are called by the *control kernel*.

The CPU only starts a kernel with a single thread block. For each iteration, this kernel first starts the data transfer of the top boundaries by submitting a work request to the network hardware (*send_btm*). Then, the compute kernel for the top part of the grid is started (*kernel_top*) and overlapped with the data transfer, handled by the network device. The *control block* now waits until the compute kernel is completed by

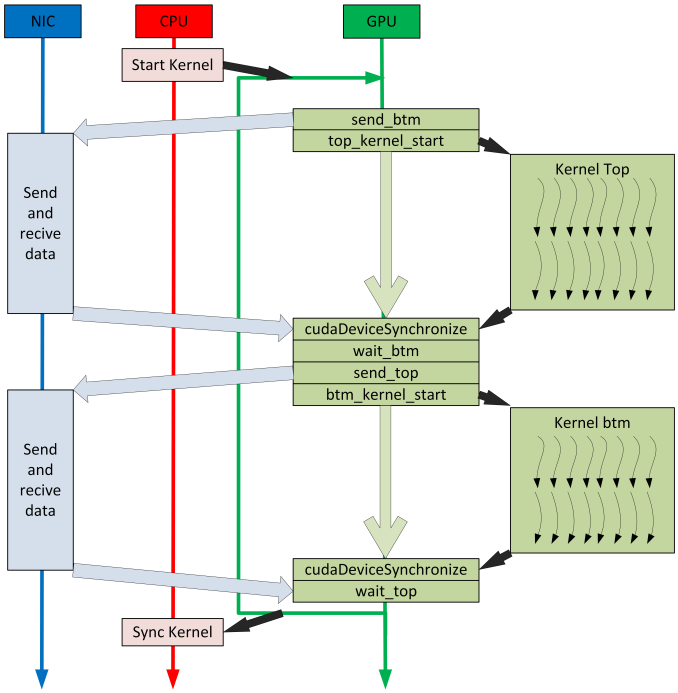


Fig. 7. Control flow of an application with communication control on the GPU, using CUDA Dynamic Parallelism and device synchronization

using *cudaDeviceSynchronize* and until the bottom boundaries are updated by the remote GPU (*wait_btm*). Now, the same procedure is repeated for the data transfer of the top boundary (*send_top* and *wait_top*) and the computation of the bottom part of the grid (*kernel_btm*).

VII. Performance Results

In this section we discuss and analyze the performance results of the previously described approaches. In particular, we compare GPU-controlled communication with a hybrid solution, in which the CPU controls the communication while the GPU performs the computation.

Since enabling the GPU to control the Infiniband device requires changes in the device drivers of both GPU and network device [3], we were not able to run our tests on more then two nodes. However, the scope of this work is to analyze the principle benefits of Dynamic Parallelism for GPU-controlled communication, therefore we believe this setup to be sufficient.

The test system consists of two nodes with each two Intel 4-core Xeon E5-2609 CPUs, directly connected with a Mellanox ConnectX-3 FDR Infiniband network. Each node is equipped with a single Nvidia K20c GPU, each with 5120 *MBytes*of global memory. We use CUDA 6 and device drivers version 331.6, which are patched to allow the registration of MMIO addresses [3]. For Infiniband, the Mellanox OFED-2.1-1.0.6 in combination with the Mellanox GPUDirect RDMA driver [27] to allow direct data transfer between distributed GPUs is used.

We run the Himeno benchmark for different input sizes. In Table I the properties for the different sizes are summarized.

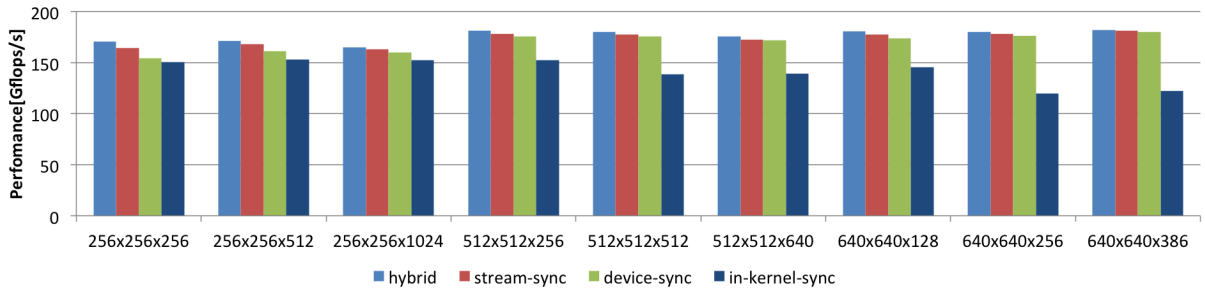


Fig. 8. Performance of the Himeno Benchmark for different problem sizes and communication methods

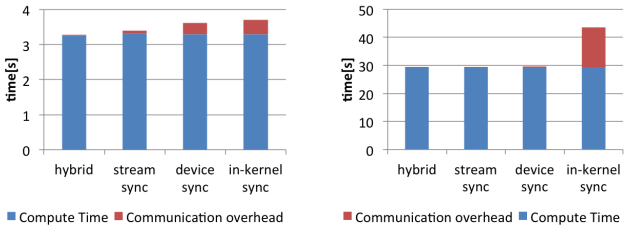
TABLE I
PROPERTIES OF THE HIMENO BENCHMARK FOR DIFFERENT PROBLEM SIZES AND AND AN EXECUTION ON TWO GPUS

Problem size	required Memory per GPU (Byte)	FLOPs per GPU per Iteration	Border size (Byte)
$256 \times 256 \times 256$	482 M	280 M	256 k
$256 \times 256 \times 256$	954 M	561 M	256 k
$256 \times 256 \times 1024$	1.90 G	1.78 G	256 k
$512 \times 512 \times 256$	1.93 G	1.13 G	1 M
$512 \times 512 \times 512$	3.81 G	2.26 G	1 M
$512 \times 512 \times 640$	4.75 G	2.83 G	1 M
$640 \times 640 \times 128$	1.54 G	0.89 G	1.5 M
$640 \times 640 \times 256$	3.01 G	1.77 G	1.5 M
$640 \times 640 \times 384$	4.48 G	2.56 G	1.5 M

The benchmark is started with a block size of 64×4 threads. These sizes are selected in such a way that each of the three problem sizes results the same amount of data transfer, while the number of data points to calculate differs. By this, we can vary the ratio between communication and computation.

The results are shown in Figure 8. The hybrid version, in which the CPU controls the communication, is always performing best, especially for small problem sizes. However, this corresponds to the results from our previous work [3]. This kind of workload allows to exploit overlap to hide the communication overhead, which is clearly done for host-controlled communication. For GPU-controlled communication, such an exploitation is not (yet) possible, resulting in an increased execution time for the GPU. Another reason is that the communication overhead on the CPU negligible small, since CPUs are optimized to execute sequential workload, which is required to source and sink communication requests to Infiniband hardware. However, for larger problem sizes, the performance of the GPU-controlled versions become more and more closer to the hybrid version, if stream or device synchronization is used. Only the version using in-kernel synchronization is performing bad for all problem sizes. The stream synchronization is performing best for all problem sizes using GPU-controlled communication.

The reason for this becomes clear if we look at the communication overhead for different configurations and methods. In Figure 9 the execution time for two problem sizes is shown. The complete execution time is separated into computation time and communication overhead.



(a) $256 \times 256 \times 256$ (b) $640 \times 640 \times 384$

Fig. 9. Communication overhead for the different problem sizes

For the hybrid approach the overhead is very small for both problem sizes, actually it is in the area of measurement uncertainty. The data transfer can completely be overlapped with the computation, resulting in the best performance.

For GPU-controlled communication using stream-synchronization or device synchronization, the communication overhead for the small problem size is large in relation to the computation time. Therefore, the communication overhead has got a relatively large influence to the performance. The overhead using the stream synchronization is the smallest. However, for larger problem sizes, the communication overhead stays constant while the computation time rises, making the overhead less relevant for larger problem sizes.

This is different for the in-kernel synchronization. Here, the communication overhead rises for larger problem sizes. We assume two reasons for this: first, for the larger problem size more thread blocks are started, resulting in an increasing overhead for inter-block synchronization. Second, if more thread blocks are started, a thread block advancing slower (e.g., the thread block that starts the communication) can slow down all the other thread blocks, so the effect of in-kernel synchronization is strengthened.

VIII. POWER ANALYSIS

In this section we analyze the power consumption and energy efficiency of the previously described approaches.

For the hybrid version, the CPU is set to sleep while waiting for the completion of a data transfer and the particular computation kernel. However, to avoid performance losses, the sleep period can set to maximal 1000 μs.

For the approaches using GPU-controlled communication,

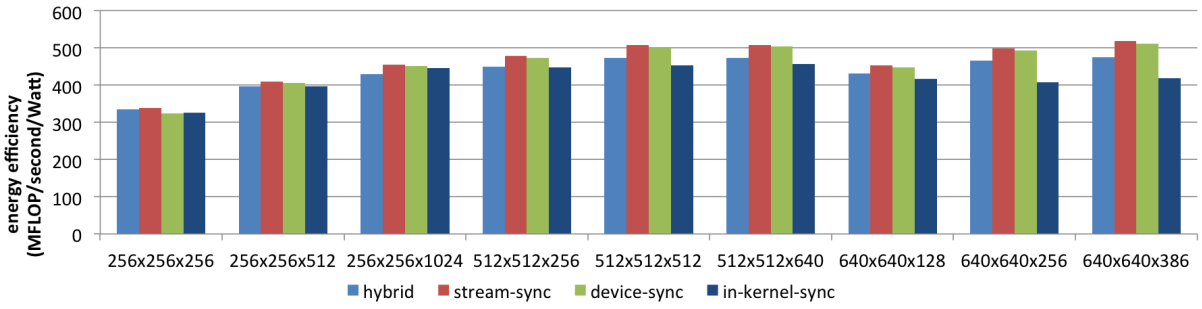


Fig. 10. Energy efficiency of the Himeno benchmark for different problem sizes and communication methods

the CPU is set to sleep right after starting the master kernel. Since the runtime of this master kernel is much longer, here a sleep period of 5000 μs can be used without any performance losses.

The benchmarks are run with 1000 iterations and the power consumption of GPU, CPU and DRAM is measured during the complete execution time. The energy efficiency of the different methods for an execution on two GPUs, expressed in $MFLOP/s/Watt$ (or $MFLOP/Joule$), is shown in Figure 10.

The results show that stream synchronization, allowing the GPU to autonomously control the communication, is the most energy-efficient method for all problem sizes, although the performance for small problem sizes is recognizable worse then for the hybrid version. However, for larger problem sizes, the energy efficiency compared to hybrid approach is significantly better. For instance, for a problem size of $640 \times 640 \times 386$, the energy efficiency of the stream synchronization is about 10% better than the hybrid version. The main reason for this is the execution time of a single iteration, therefore the relation between communication and computation.

In Figure 11 the power consumption of the different components over time for a problem size of $640 \times 640 \times 384$ is shown. The figures show the course of the power for a single machine. While the GPU's power consumption is identical for the hybrid, stream- and device-synchronization methods, the values for CPU and DRAM differ. For the hybrid version, a CPU thread is required to control the communication. Although the communication is handled by the network device, the CPU cannot enter a sleep state for a longer period and therefore it consumes about 50 $Watt$. If the GPU controls the communication, the CPU can be set to sleep for the complete execution time of the kernel. Therefore, it then only consumes between 24 and 29 $Watt$.

The DRAM power consumption can hardly be recognized in Figure 11, however it also differs for the hybrid and the GPU-controlled versions. During execution time, for the hybrid version DRAM consumes about 3.9 $Watt$, while the GPU-controlled versions require only about 2.2 $Watt$.

Another interesting observation is that using in-kernel synchronization the GPU consumes considerable less power compared to all other methods. The reason for this may be that the in-kernel synchronization slows down the complete GPU and therefore not all SMs can be fully utilized. However, while this leads to a lower power consumption, the increased execution time outweighs this benefit for almost all cases. Only for a very small problem size the in-kernel synchronization method shows benefits (see Figure 10).

IX. Conclusion

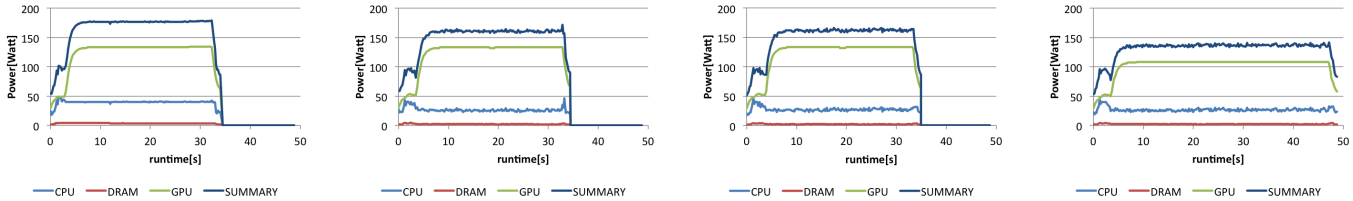
We have shown that GPU-controlled communication results in performance losses compared to a hybrid solution, in which the CPU controls the communication while computational tasks are off-loaded to the GPU. However, the smaller the communication overhead in contrast to the computational work is, the smaller is this effect.

For GPU-controlled communication, the intra-GPU synchronization between the individual thread blocks is the most important factor for performance. The best results are reached if the communication functions are added as kernels to the same stream as the compute kernels. Although the performance of GPU-controlled communication still is unconvincing, the power saving that can achieved with this method is very promising. For large problem sizes, up to 10% of the energy can be saved by moving communication tasks to the GPU. Alternatively, one can argue that the CPU is available for other tasks.

In particular, we showed that energy savings are possible by relieving the CPU from controlling the work flow between the GPU and the network device. This could only be achieved by CUDA Dynamic Parallelism, which allows the GPU to start and stop compute and communication kernels independent from the CPU.

Especially for small problem sizes, the communication overhead still carries a lot of weight. The main reason for this is that a complex communication protocol like Infiniband verbs is not very suitable for the high degree of parallelism of GPUs. We have shown this in detail in our previous work [3], [9]. In summary, power savings are possible at the cost of increased execution time, which however is a viable way for a higher energy efficiency.

As even higher energy savings would be possible if the GPUs could handle communication more efficient, we plead for specialized communication models and methods for future GPU-accelerated systems, which allow an easier sourcing and



(a) Hybrid (b) Stream synchronization (c) Device synchronization (d) In-kernel synchronization

Fig. 11. Power over time for the Himeno benchmark using a problem size of $640 \times 640 \times 384$ and different communication methods

sinking of communication requests than the Verbs protocol. It seems that the Verbs interface is a poor match to the GPU execution model. Considering upcoming technologies like Nvlink, which helps to overcome the PCIe bottleneck, such an approach will be pioneering for future energy-efficient, GPU-centric high performance systems.

In our future work, we plan to explore other GPU-controlled communication methods to find a suitable, fast and energy-efficient model that is most suitable for GPU architectures. Also, we hope to extend our experiments to larger clusters and to a broader spectrum of applications with distinct communication characteristics. We think that future energy-efficient systems require specialized processors like GPUs to improve the performance-per-Watt metric. However, communication models and methods have to match these specialized architectures, as otherwise gains in energy efficiency can be diminished or even be neutralized.

REFERENCES

[1] (2014, August) Green500. [Online]. Available: http://www.green500.org/

[2] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR'10)*. Springer-Verlag, 2011, pp. 1–25.

[3] L. Oden, H. Fröning, and F.-J. Pfreundt, "Infiniband-verbs on GPU: A case study of controlling an infiniband network device from the GPU," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International*. IEEE, 2014, in press.

[4] L. Oden, B. Klenk, and H. Fröning, "Energy-efficient collective reduce and allreduce operations on distributed gpus," in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Chicago, IL, US, May 2014.

[5] L. Oden and H. Fröning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *IEEE Cluster*, 2013.

[6] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili, "Coordinated energy management in heterogeneous processors," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. ACM, 2013.

[7] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. Stone, and J. Phillips, "Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters," in *Green Computing Conference, 2010 International*, 2010.

[8] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *International Symposium on Parallel & Distributed Processing. IPDPS2009*, 2009.

[9] B. Klenk, L. Oden, and H. Fröning, "Analyzing put/get apis for thread-collaborative processors," in *Parallel Processing Workshops (ICPPW), 2014 43st International Conference on*. IEEE, 2014.

[10] 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.

[11] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. Panda, "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, 2013.

[12] 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.

[13] L. Oden, "GPI2 for GPUs: A PGAS framework for efficient communication in hybrid clusters," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing, M. Bader and A. Bode, Eds., vol. 25. IOS Press, March 2014, pp. 461 – 470.

[14] J. DiMarco and M. Taufer, "Performance impact of dynamic parallelism on different clustering algorithms," in *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2013, pp. 87 520E–87 520E.

[15] J. Dong, F. Wang, and B. Yuan, "Accelerating BIRCH for clustering large scale streaming data using CUDA dynamic parallelism," in *Intelligent Data Engineering and Automated Learning–IDEAL 2013*. Springer, 2013, pp. 409–416.

[16] F. Wang, J. Dong, and B. Yuan, "Graph-based substructure pattern mining using CUDA dynamic parallelism," in *Intelligent Data Engineering and Automated Learning–IDEAL 2013*. Springer, 2013, pp. 342–349.

[17] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in gpgpu applications," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014, pp. 93–106.

[18] S. Jones. (2014, march) Introduction to dynamic parallelism. [Online]. Available: http://on-demand.gputechconf.com/gtc/2012/presentations/S0338-GTC2012-CUDA-Programming-Model.pdf

[19] S. Xiao and W.-c. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[20] Nvidia. (2014, jun) GPUDirect RDMA; CUDA toolkit documentation. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[21] J. Kraus. (2014, march) An introduction to CUDA-aware MPI. [Online]. Available: https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/

[22] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10. New York, NY, USA: ACM, 2010.

[23] [Online]. Available: developer.nvidia.com/nvidia-management-library-nvml

[24] E. H. Phillips and M. Fatica, "Implementing the himeno benchmark with cuda on gpu clusters," in *Parallel and Distributed Processing (IPDPS)*, 2010.

[25] 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.

[26] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–14.

[27] (2014, August) Mellanox ofed gpudirect rdma. [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family=116