J. Parallel Distrib. Comput. 114 (2018) 28–45

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

GPUDirect Async: Exploring GPU synchronous communication techniques for InfiniBand clusters

E. Agostini [*], D. Rossetti, S. Potluri

NVIDIA, Santa Clara, CA, United States

HIGHLIGHTS

- GPUDirect Async: new technology which enables the GPU to directly trigger and sync network transfers.
- LibMP: a simple message passing library to demonstrate the use of GPUDirect Async in applications.
- Two different Async communication models: Stream Asynchronous model and Kernel-Initiated model.
- Performance models introduced to help interpreting results on domain decomposed numerical applications.
- Representative benchmarks of different scientific domains to demonstrate the benefits of GPUDirect Async.

ARTICLE INFO

Article history:
Received 16 February 2017
Received in revised form 24 October 2017
Accepted 12 December 2017
Available online 20 December 2017

Keywords:
GPUDirect Async
CUDA 8.0
InfiniBand
Asynchronous communication models

ABSTRACT

NVIDIA GPUDirect is a family of technologies aimed at optimizing data movement among GPUs (P2P) or among GPUs and third-party devices (RDMA). GPUDirect Async, introduced in CUDA 8.0, is a new addition which allows direct synchronization between GPU and third party devices. For example, Async allows an NVIDIA GPU to directly trigger and poll for completion of communication operations queued to an InfiniBand Connect-IB network adapter, with no involvement of CPU in the critical communication path of GPU applications. In this paper we describe the motivations and the building blocks of GPUDirect Async. After an initial analysis with a micro-benchmark, by means of a performance model, we show the potential benefits of using two different asynchronous communication models supported by this new technology in two MPI multi-GPU applications: HPGMG-FV, a proxy for real-world geometric multi-grid applications and CoMD-CUDA, a proxy for Classical Molecular Dynamics codes. We also report a test case in which the use of GPUDirect Async does not provide any advantage, that is an implementation of the Breadth First Search algorithm for large scale graphs.

1. Introduction

NVIDIA GPUDirect technologies [16] allow peer GPUs, network adapters and other devices to directly read from and write to GPU device memory. This eliminates additional copies to host memory, reducing latencies and lowering CPU overhead. This results in significant improvements in data transfer times for applications running on NVIDIA Tesla, GeForce and Quadro GPUs [29]. The first GPUDirect version was introduced in 2010 along with CUDA 3.1, to accelerate the communication with third party PCIe network and storage device drivers via shared pinned host memory. In 2011, starting from CUDA 4.0, GPUDirect Peer-to-Peer (P2P) allowed direct access and transfers between GPUs on the same PCIe root port. Around that time, some *CUDA-aware*[1] MPI middlewares added support for GPUDirect P2P to accelerate intra-node GPU to GPU communications.

Finally with CUDA 5.0, NVIDIA released the GPUDirect RDMA, enabling the direct PCIe data path between GPUs and third-party peripheral devices, like network interface controllers (NICs). Since MLNX OFED 2.1, Mellanox [25] have been supporting GPUDirect RDMA on ConnectX-3 and later Host Channel Adapters (HCAs). Similarly Chelsio added support for GPUDirect RDMA to their OFED software stack [6]. More recently Broadcom announced the same too.

[1] Refers to the ability for the user to pass GPU memory pointer to MPI communication functions. By doing so, MPI middlewares have the opportunity to improve performance, by deploying communication protocols optimized for GPUs, e.g. pipelined data staging, as well as usability, i.e. explicit device-to-host or host-to-device memory copies beforeafter communications are no more necessary. For a tutorial on *CUDA-aware* MPI, see [11].

[*] Corresponding author.
E-mail address: eagostini@nvidia.com (E. Agostini).

https://doi.org/10.1016/j.jpdc.2017.12.007
0743-7315/© 2017 Elsevier Inc. All rights reserved.

Async [31,1] is a recent member of the GPUDirect family of technologies which has been initially announced at the Supercomputing 2015 conference [30] and the required software APIs introduced in CUDA 8.0 [32]. While GPUDirect Async is generic in that it can be applied in principle to different realms – e.g. network communications, storage I/O, etc. – in this paper we focus on its use in combination with Mellanox InfiniBand Host Channel Adapters (HCAs). With MOFED 3.4, Mellanox has released Peer-direct Async, a set of Verbs extensions complementary to GPUDirect Async.

Traditionally in GPU-accelerated parallel applications, the CPU works as the orchestrators between GPU compute and network communication tasks, for example waiting for a (set of) GPU compute task(s) – CUDA kernels in NVIDIA parlor – to complete (`cudaStreamSynchronize`) before issuing a related communication onto the NIC, or conversely waiting for communication completions (`MPI_Wait`) before issuing GPU compute tasks. Both GPUDirect P2P and RDMA are all about optimizing data movement respectively among GPUs and with third-party devices, in the previous example the NIC. GPUDirect Async instead helps offloading the control path onto the GPU, by enabling the GPU to both trigger communication transfers and synchronize on notifications, directly over the PCIe bus without the use of agents running on the host CPU. That helps getting rid of the CPU from the critical path in applications, potentially improving performance by allowing for more computation–communication overlap, by freeing CPU core cycles which could be spent elsewhere, or conversely by potentially sustaining performance in conjunction with low-performance CPUs.

In the following we describe two separate ways of leveraging GPUDirect Async, referred to as asynchronous communication models (Section 4), distinguished by the particular GPU engine which interacts with the NIC HW. In the first model, *Stream Asynchronous* or SA, we introduce *on-a-stream* point-to-point and one-sided communication primitives, which blend communications and computations in the same way GPU task synchronizations are expressed in CUDA, that is through the concept of CUDA streams. In other words, communications are naturally executed respecting the order in which they are submitted on the CUDA stream, correctly mixing together with CUDA asynchronous memory copies and CUDA kernels. We distinguish multiple phases: (a) the CPU prepares the communication primitives – including buffer pointers and sizes, network addresses, etc. – and posts the associated descriptors onto the NIC command queues; (b) the meta-data required to activate those descriptors are collected and converted into CUDA Memory Operations (MemOps); (c) the CPU submits those MemOps on the user CUDA stream; (d) sometimes later the CUDA stream executes those MemOps with the effect of triggering the communications prepared in phase (a). The second model, *Kernel-Initiated* or KI, is a variation of the first, where phases (a) and (b) remain the same as in the previous case, followed by phase (c) where the CPU passes the meta-data to a CUDA kernel; (d) later, probably after some form of inter-thread synchronization which is specific to the particular computation, the CUDA kernel (as opposed to the CUDA stream) uses the meta-data to either trigger the communications or to wait on their completions. Note that in both cases, phase (a) is executed on the CPU by the same software stack which handles regular communications, i.e. in our case by the Mellanox user-space `libmlx5` driver. That code, full of bitwise operations and branches, is NIC HW dependent (each NIC vendor has its own HW interface), hardly parallelizable so therefore it is convenient for it to be kept on the CPU which is optimized for low latency.

The rest of the paper is organized in the following way: other papers have already explored moving parts or most of the low-level communication stack onto the GPU programmable cores (SMs), mostly at the experimental level; those are briefly recalled in Section 2. GPUDirect Async and its implementation to InfiniBand Verbs is described in Section 3. There we depict the software stack, including LibMP, a simple message passing library that we developed to quickly enable GPUDirect Async in our benchmarks. In Section 4, for both Async communication models we introduce simplified performance models in order to help in interpreting the experimental results. In Sections 5 and 6, we show performance results respectively for micro-benchmarks and for a small suite of applications representative of a few scientific domains. Finally we draw our conclusions 7.

2. Related works

Since the introduction of GPUs as general purpose accelerators, many papers have studied ways to optimize the communication data path between GPUs and NICs, see for example [2,4], custom FPGA-based NICs natively implementing both the NVIDIA P2P and the RDMA protocols in HW, and similarly [27], where the authors experimented with GPUDirect RDMA using the EXTOLL interconnect.

So far instead, only a few papers have explored ways to offload the communication control path onto the GPU. In Table 1, we schematize those papers, based on where the different communication phases (preparation of communication descriptors, triggering) are carried out on (CPU, GPU stream, GPU SMs), the location of the NIC control structures (host memory, GPU memory), the benchmark applications and the location of the related data buffers (GPUDirect RDMA is used to communicate data buffers located on GPU memory).

S. Kim et al. [23] describe a native GPU networking layer that provides a BSD-like socket abstraction and high-level networking APIs to GPU programs. While those APIs can be invoked directly by CUDA threads, they are actually performed by a proxy agent running on the CPU.

Lena Oden et al. [28] explored different approaches to generating and posting InfiniBand send and receive operations from within CUDA kernels. In one of the experiments they implemented a GPU-side subset of IB Verbs, modifying both the open source part of the NVIDIA kernel-mode driver and the Mellanox user-space driver to map some key InfiniBand HCA resources (e.g. memory queues and HW doorbell registers) onto the GPU. Those GPU Verbs APIs use a critical section to serialize access to the IB QP at the single CUDA thread-block granularity. They showed unsatisfactory results leading the authors to conclude that the GPU-native design is inferior to traditional CPU-controlled network transfers.

F. Daud et al. [12] implements a GPU-side subset of the Global address space Programming Interface (GPI) library, enabling high-performance RDMA communications directly from GPU code, i.e. the CPU is completely bypassed and the CUDA kernel threads both prepare and trigger the NIC commands relative to the communications. Similarly to [28], they map InfiniBand resources onto the GPU. They also experiment with having some of those resources (QPs, CQs) backed by GPU memory instead of host memory. In the last two papers, they re-implement part of the communication stack on the GPU side. Besides, they had to hack the GPU and/or HCA drivers to allow the GPU to access the NIC doorbell and to place the control structure on GPU memory. This approach presents two drawbacks: the GPU-side stack uses more GPU resources (e.g. registers), potentially reducing the occupancy and consequently the performance of the computation kernels, where the communication functions are used. Besides to the best of our knowledge they are affected by a GPU memory consistency issues [19] associated to using receive data buffers, updated via GPUDirect RDMA, from inside persistent CUDA kernels. GPUDirect Async officially introduces a mechanism to fence incoming traffic directed towards GPU memory buffers. This mechanism is exposed as a new FLUSH

Table 1
Comparing the different approaches cited vs. those proposed in this paper (last two rows).

Paper	comm descriptors creation (WQEs)	comm trigger (control path)	IB control structures location (QP, CQ)	Data buffers location	Benchmarking
Oden et al. [28]	GPU SMs	CUDA kernels (SMs)	GPU or host	GPU	Micro-benchmarks
Kim et al. [23]	CPU	proxy on CPU	Host	GPU	Synth. workloads
Daud et al. [12]	GPU SMs	CUDA kernels (SMs)	GPU or host	GPU	Micro-benchmarks synth. workloads
Venkatesh et al. [34]	CPU	CPU assisted GPU stream	Host	GPU	
Async SA here & [1]	CPU	GPU stream	Host	Host	Micro-benchmarks HPC mini-apps
Async KI here & [1]	CPU	CUDA kernels (SMs)	Host	Host	Micro-benchmarks HPC mini-apps

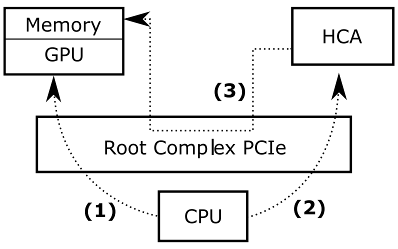

Fig. 1. GPUDirect RDMA compute-and-send workflow.

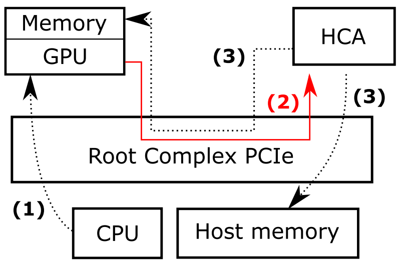

Fig. 2. GPUDirect Async compute-and-send workflow.

MemOP which can be queued after the wait on a communication completion notification and before releasing a pre-launched GPU kernel

In [34] the Ohio State University team, in collaboration with some of the authors, presented early results of using GPUDirect Async technology in MVAPICH2 (MPI-GDS). The scope of that paper was to explore protocol designs which take advantage of GPUDirect Async while at the same time respecting the demands of the MPI specification. In that respect, this paper constitutes a premise of that other work. More specifically MPI-GDS offers MPI point-to-point primitives synchronous to CUDA streams. MPI tag matching and the rendezvous protocol are supported and implemented using an hybrid approach, where the CPU actively progresses part of the protocol at the cost of additional overhead. In this paper instead we measure the potential performance of GPUDirect Async alone, irrespective of the use of other GPUDirect technologies. Besides we also explore one-sided communication primitives and CUDA kernel initiated communications. Finally there they focus on micro-benchmarks while here we present benchmarks on applications.

Compared to our previous work [1], this paper introduces several improvements:

- A more detailed description of the Async technology and its software stack.
- A general performance model capturing the communication pattern of domain decomposed multi-GPU applications.
- The performance model is applied to GPUDirect Async models in order to clarify the requirements needed by each asynchronous model to reach a gain in performance.
- New micro-benchmarks and a negative test case are discussed here, clarifying some GPUDirect Async limitations.

3. GPUDirect Async

It is common for scientific applications to alternate between compute and communication phases. Transition from compute to communication in a multi-node GPU application involves: launching a compute kernel on a GPU, waiting for it to complete and then sending the data over the network.

The workflow when using a GPUDirect RDMA-enabled InfiniBand HCA, is illustrated in Fig. 1:

1. CPU queues some *computation tasks* to the GPU (kernel launch) and synchronizes waiting for its completion.
2. CPU queues *communication tasks* to the InfiniBand HCA.
3. HCA fetches data directly from GPU memory, thanks to GPUDirect RDMA.
4. HCA injects the associated messages through the network.
5. CPU synchronizes with the HCA by waiting for a completion (not shown in the figure).

Note that when GPUDirect RDMA is not used, suitable GPU-to-host copies will be included in the *communication tasks*.

GPUDirect Async removes dependency on the CPU by enabling the GPU to trigger communication on the HCA and the HCA to unblock CUDA tasks. The CPU needs only to prepare and enqueue both the compute and communication tasks. The compute-and-send workflow in the presence of GPUDirect Async is shown in Fig. 2:

1. CPU prepares and queues both *computation* and *communication tasks* to the GPU.
2. GPU completes computation tasks and directly triggers the pending communications on the HCA.
3. HCA fetches data directly from host memory, or from GPU memory when GPUDirect RDMA is used.
4. HCA sends data.

In this latter case, the CPU workload changes. For example, after having prepared and queued all the necessary tasks onto the GPU, the CPU can go back and do other useful work.

We note here that GPUDirect Async is independent of GPUDirect RDMA, so we can experiment with it in isolation, i.e. using the first without enabling the second. In [3] it has been noted that GPUDirect RDMA performance heavily depends on the PCIe fabric, i.e. the type and number of PCIe bridges and switches which connects the NIC to the GPU, as well as on the particular GPU architecture [32] used. So for example, for large message sizes in the rendezvous protocol, pipelined staging through host memory on the sender side is more efficient than using GPUDirect RDMA. Doing the same thing here, as in MPI-GDS [34], would require the CPU to progress the communication, defying our objective of

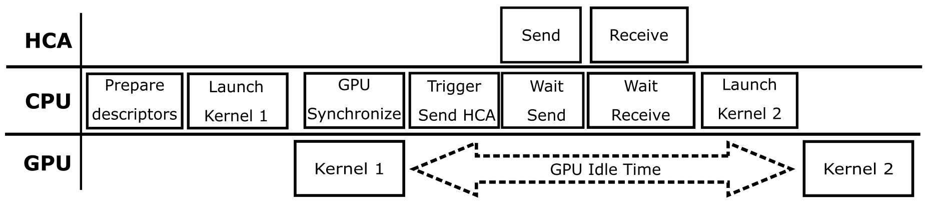

Fig. 3. Communication period MPI multi-GPU application timeline.

benchmarking GPUDirect Async in isolation. Considering that in one of our benchmarking platforms GPUDirect RDMA is inefficient, we decided to not use it.

3.1. Motivations

Fig. 3 depicts the timeline of a typical multi-GPU application, comprising both computation and communication phases. The CPU iteratively schedules work onto the GPU, waits for GPU kernel completion, triggers communications on the HCA and finally polls the HCA for the completion of communications. The CPU must be constantly running at peak performance most of the times just to ensure responsiveness, that is that the different phases are scheduled as quickly as possible.

Generally, when the applications are strong scaled, due to geometrical and/or physical properties, the length of both the GPU computations and the network communication reduces on each compute node. Additionally, the rate of reduction is different from computation and communications, e.g. the former scaling with the volume while the latter with the surface as with the domain decomposition approach. So therefore, even when the algorithm allows for them to be overlapped, it is increasingly difficult to hide communication behind the computation. Beyond a certain point, the application does not scale anymore. The onset of the non-scaling regime can be anticipated – i.e. the application stops scaling at a smaller number of GPUs – if the overheads incurred by the CPU when launching computation and communication tasks become of the same order as the time necessary to execute them respectively on the GPU and the NIC. In these cases, launching a GPU computation can take up to tens of microseconds, which can be about the same time it takes to execute that very same task, or to exchange a few kilobytes of data over the network. Similarly some applications, as HPGMG which is introduced later, may go through phases – coarse grain levels – where it is more convenient to move computations back to the CPU to avoid the overhead of launching work on the GPU.

By leveraging GPUDirect Async instead, a whole parallel computation phase can be offloaded onto a CUDA stream. That in turns allows to overlap – thereby paying the cost of – the work submission at iteration i while at the same time iteration $i-1$ is being orchestrated by the GPU, effectively removing the CPU from the critical path. As shown qualitatively in Fig. 4, there are times when the CPU becomes idle for potentially extended period of times, during which it can do useful work. When that is not possible, the CPU can be allowed to go down into deeper sleep states, thereby lowering the application power profile.

3.2. Implementation

Currently, support for GPUDirect Async requires extended IB Verbs APIs contained in MLNX OFED 4.0 and is limited to the latest generation Mellanox InfiniBand Host Channel Adapters (HCAs) [22], those supported by the `libmlx5` user-space provider library.

Traditionally, the CPU issues communication operations onto the IB HCA by filling in data structures (Work Requests or WQEs) onto either the send or the receive memory queue, and subsequently updating some kind of *doorbell register*, both associated to a specific Queue Pair (QP). The doorbell update is needed to inform the HCA about the new requests ready to be processed. In the particular case of recent Mellanox HCAs, two distinct doorbell updates are required when triggering send operations: one to a 32-bits word (DBREC) in host memory, while the other to a HW register located at a specific offset into one of the HCA PCIe resources (Base Address Register or BAR). When kernel by-pass is used, the user-space process directly updates the DB by using an uncached memory-mapped I/O (MMIO) mapping of the HCA BAR page holding that registers. When a request is completed – i.e. data have been sent or received – the HCA adds a new CQE (Completion Queue Entry) respectively into the send or the receive Completion Queue (CQ) associated to that QP at creation time. The application needs to poll the corresponding CQ in order to detect whether a request has been completed (Fig. 5).

While the CPU is still in charge of preparing the commands, GPUDirect Async requires the GPU to directly access to the HCA doorbell registers and to the CQs (which reside on the host memory in our case), using a combination of two CUDA driver functions: *cuMemHostRegister()* to page-lock an existing host memory range and maps it into the GPU's address space; and *cuMemHostGetDevicePointer()* to retrieve the corresponding device pointer.

In particular, the *CU_MEMHOSTREGISTER_IOMEMORY* flag when registering an MMIO address range belonging to a third-party PCIe device (the InfiniBand HCA in our case). The later corresponds to the creation of a so called *GPU peer mapping*, that is a GPU mapping to peer PCIe device. Note that in the current implementation, the whole MMIO range must be physically contiguous and marked cache inhibited for the CPU.

Because of HW limitations in NVIDIA GPUs prior to the *Pascal* architecture, a special Mellanox HCA firmware is required to let the HCA PCIe resource (BAR) to be placed in the appropriate address range.

Once the doorbell registers and the CQs are mapped on the GPU, it is possible to access them on either (a) CUDA streams or (b) from CUDA kernel threads. We refer to the former as the Stream Asynchronous (SA) communication model – see Section 4.2 – and to the latter as the Kernel-Initiated (KI) communication model — Section 4.3.

In the SA model, we make extensive use of CUDA Memory Operations APIs, described below, to either wait (poll) the CQEs or write (ring) the doorbell registers.

1. *cuStreamWaitValue32(stream, value, address, condition)*: enqueues a synchronization of the CUDA stream on the given memory location. Work ordered after the operation will block until the given condition (EQual, Greater-or-EQual, AND) is met. That for example allows to block the CUDA stream to until a particular completion event (CQE) is signaled by the NIC.
2. *cuStreamWriteValue32(stream, value, address)*: writes the passed value into the memory identified by the device address. This API is used to *ring* the QP doorbell register.

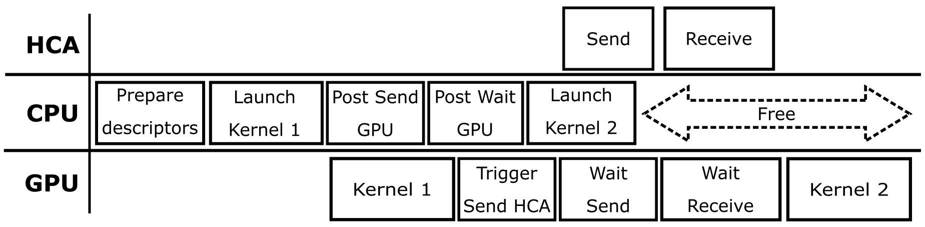

Fig. 4. Communication period multi-GPU application timeline with GPUDirect Async.

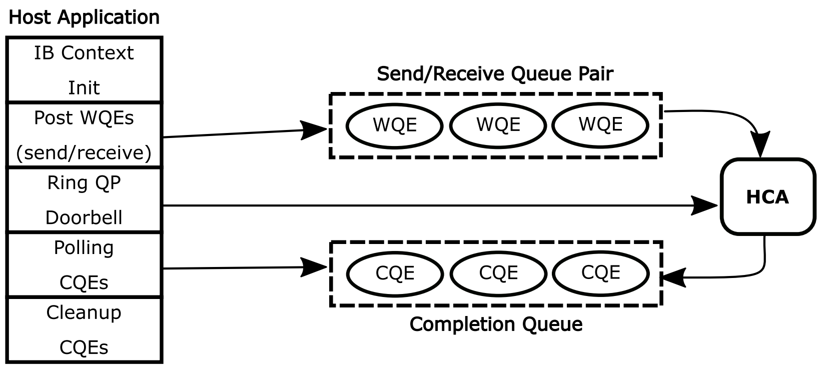

Fig. 5. InfiniBand HCA send/receive requests processing.

3. *cuStreamBatchMemOp(stream, count, mem_ops[])*: a batch version of the previous functions, taking as input a vector of memory operations (wait or write).

When GPUDirect Async is used – see interaction diagram in Fig. 6 – the CPU is still needed to:

- allocate and register communication buffers (device or host pinned memory);
- map the HCA specific data structures onto the GPU, as explained above;
- prepare and post WQEs on the QPs
- prepare the send and receive requests descriptors and convert them into a sequence of Memory Operations;
- poll on the CQEs, once successfully read by the CUDA stream.

On the contrary, GPU has more tasks to do:

- triggering prepared WQEs by ringing the doorbells;
- waiting for the CQE related to a send or receive WQE, polling on the CQ.

Given the small set of features offered by the CUDA Memory Operation APIs, i.e. the CUDA stream can only block on a memory location, we cannot implement a full blown CQE parser and dispatcher there, as on the CPU. Hence, if we want to literally keep the CPU off the critical path, the GPU stream needs the CQEs and the send/receive operations to be strictly associated, i.e. the completion of the i-th WQE will be placed in the i-th available CQE. This is for example the case if at QP creation time, distinct send and receive CQs are used, as well as giving up on Shared Receive Queues (SRQs).

Note that error handling and recovery is still done by the CPU. When polling the CQEs, the CPU may observe completions with errors. In that case, it is responsible to abort all outstanding work for both the GPU and the HCA.

3.3. Software stack

In order to take advantage of the GPUDirect Async technology, we implemented or modified libraries at different levels of the software stack shown in Fig. 7.

3.3.1. libibverbs

`libibverbs` implements the OpenFabrics InfiniBand Verbs API specification. In version 4.0, Mellanox has introduced the new Peer-Direct Async APIs (e.g. see the `peer_ops.h` header) targeting the NVIDIA GPUDirect Async technology.

3.3.2. libmlx5

It is the vendor-specific low-level provider library managing recent Mellanox InfiniBand HCAs. It allows user-space processes to access directly Mellanox HCA hardware with low latency and low overhead (kernel by-pass).

3.3.3. LibGDSync

Developed by the authors, it conceptually implements GPUDirect Async support on InfiniBand Verbs, by bridging the gap between the CUDA and the Verbs APIs. It consists of a set of low-level APIs which are still very similar to IB Verbs though operating on CUDA streams.

LibGDSync is responsible for the creation of Verbs objects, i.e. queue pairs (QPs), completion queues (CQs), structures respecting the constraints of GPUDirect Async, to register host memory when needed, to post send instructions and completion waiting directly on the GPU stream. Functions like *gds_stream_queue_send* or *gds_stream_wait_cq*, internally use the CUDA Stream MemOp APIs as described in the previous Section 3.2.

3.3.4. LibMP

Implemented by the authors, it is a lightweight messaging library built on top of LibGDSync APIs, developed as a technology demonstrator to easily deploy the GPUDirect Async technology in applications. Once the MPI environment is initialized (i.e. communicator, ranks, topology, etc.), it is possible to replace the standard MPI communication primitives with the respective LibMP ones, e.g. *mp_isend_on_stream()* instead of *MPI_Isend()*, *mp_wait_on_stream()* instead of *MPI_Wait()*, etc. LibMP features and design tradeoffs are:

- Point-to-Point communication primitives using the send /receive semantic of IB verbs: receive buffers are consumed in the order they are posted on the particular QP.

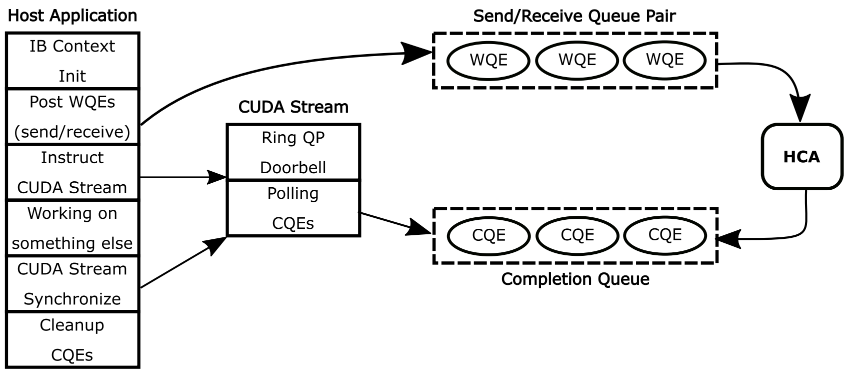

Fig. 6. InfiniBand HCA send/receive requests processing with GPUDirect Async.

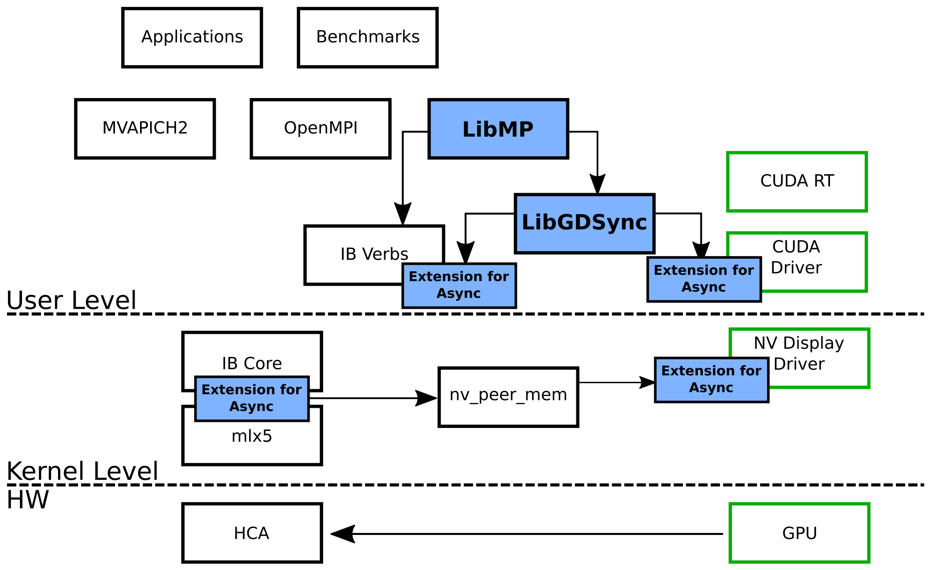

Fig. 7. GPUDirect Async software stack.

- One-Sided asynchronous communications, e.g. put and get on remote memory addresses.
- No support for MPI-style tag matching.
- No collective communication primitives.

As previously stated, each QP has its own CQ. The depth of both the WQs and the CQs can be set at run-time; in our benchmarks we used a default depth of 512 entries. In our experiments the WQs, CQs and DBREC were residing on host memory; in a future version we plan to enable the use of GPU memory for CQs and DBREC.

The parameters for the communication primitives (i.e. destination/source peer ranks, message size, buffer pointers) are used when the CPU post the WQEs, before collecting the descriptors and turning them into CUDA API calls. Hence they must be known at the time of WQE posting and cannot be for example the result of a GPU computation, which can add complexity in some applications as shown below. While in principle it is possible to change some of those parameters directly modifying the WQEs from within the GPU work, e.g. prior to triggering them, that would pose well-known challenges as discussed in [12,28].

3.3.5. System requirements

Async requirements are:

- Mellanox Connect-IB or later HCA, possibly with a special firmware version.
- MLNX OFED 4.0 for Peer-Direct Async Verbs APIs.
- CUDA 8.0 for Stream Memory Operations APIs described in Section 3.2
- NVIDIA display driver version 384 or newer.
- LibGDSync library, available on [17].
- A special NVIDIA kernel driver registry key is required to enable GPU peer mappings.
- The nvidia_peer_memory kernel module.
- The GDRcopy library [15].

In algorithm 1 we present the typical structure of a GPUDirect Async application, using LibMP functions, where two processes exchange some data using the *Stream Asynchronous* model, mixing communication and computation tasks.

4. GPUDirect Async models

As described in previous sections, LibMP presents two different execution models: the *Stream Asynchronous* model (SA), where communications are asynchronous with respect to the host and synchronous with respect to the CUDA stream and the *Kernel-Initiated* model (KI), where communications are triggered by CUDA threads within a kernel. In this section, with the help of abstract performance models, we compare the behavior of our Async models with respect to the standard MPI communication model. We consider an execution flow which is typical of GPU-accelerated MPI

Algorithm 1 LibMP example C-pseudocode

```
numRanks=2, Nreq=1;
                                        ▷ Initialize MPI and CUDA environment
initialize_MPI_environment();
cuda_init();
myRank = get_MPI_rank();
...
                                        ▷ Initialize LibMP environment
mp_init(MPI_COMM_WORLD, !myRank, numRanks);
...
                                        ▷ Create mp requests descriptors
mp_request_t * sreq, rreq;
host_memory_alloc_request(sreq, Nreq);
host_memory_alloc_request(rreq, Nreq);
...
                                        ▷ Allocate send/receive buffers
memory_alloc_buffer(sendBuffer, sizeS);
memory_alloc_buffer(recvBuffer, sizeR);
...
                                        ▷ Register related memory regions
mp_reg_t sreg, rreg;
mp_register(sendBuffer, sizeS, &sreg);
mp_register(recvBuffer, sizeR, &rreg);
...
                                        ▷ Post a Receive WQE
mp_irecv(recvBuffer, sizeR, !myRank, &rreg, &rreq));

                                        ▷ Start a CUDA kernel to prepare send buffers
launch_cuda_kernel(sendBuffer, ....., stream);

                                        ▷ Trigger HCA for Send WQE
mp_isend_on_stream(sendBuffer, sizeS, !myRank, &sreg, &sreq,
stream);

                                        ▷ Wait (poll) for Receive CQE
mp_wait_on_stream(&rreq, stream);

                                        ▷ Start a CUDA kernel to work on received data
launch_cuda_kernel(recvBuffer, ....., stream);
...
                                        ▷ Cleanup CQEs
mp_wait(&rreq);
mp_wait(&sreq);
...
                                        ▷ Synchronize and cleanup
cudaDeviceSynchronize();
mp_deregister(&rreg);
mp_deregister(&sreg);
cleanup_MPI_environment();
```

applications, where each MPI rank alternates between computations and communication with other peers. Later, in Section 6, we will be using our performance models to explore the conditions when we expect GPUDirect Async to improve over the MPI model.

4.1. CPU synchronous model

As an example of regular multi-GPU MPI applications, we consider the *kernel* of a D-dimensional iterative stencil computations parallelized using the domain decomposition approach. Three independent phases can be identified:

1. **Compute and Send**: for X times, launch (LA_i time) some CUDA tasks (running for A_i time) like kernels or memory transfers, execute some operations on the host, like a synchronization with the CUDA stream (TH time), then send the computed data (S_i time).
2. **Interior Compute**: for Y times, execute some operations on the host (TH time) and launch (LB_j time) some CUDA tasks (B_j time) working on inner data elements, i.e. not dependent upon data coming from neighboring nodes.
3. **Receive and Compute**: for Z times, wait to receive something from the other processes (W_k), execute some operation on the host (TH time) and launch (LC_k time) CUDA tasks (C_k time) working on received data.

Considering R iterations of the above pattern, schematized in Fig. 8, Eqs. (1) represents the time spent respectively on the CPU ($TCPU_S$), the GPU ($TGPU_S$) and by the whole applications (T_S). T_{idle} is the GPU idle time spent while waiting for CPU work.

$$\begin{aligned} TCPU_S &= R \times [\sum_{i=1}^{X}(LA_i + TH_{sync} + S_i) + \sum_{j=1}^{Y}(LB_j) + \\ &\quad \sum_{k=1}^{Z}(W_k + TH_{sync} + LC_k) + TH_{sync}] \\ TGPU_S &= R \times [\sum_{i=1}^{X} A_i + \sum_{j=1}^{Y} B_j + \sum_{k=1}^{Z} C_k] + T_{idle} \\ T_S &= \max(TCPU_S, TGPU_S). \end{aligned} \tag{1}$$

The total time T_S will be equal to the CPU time, because the CPU is always busy, i.e. at worst waiting for the completion of GPU tasks, represented by the TH_{sync} parameter:

$$TGPU_S \le TCPU_S \rightarrow T_S = TCPU_S.$$

In the following sections we examine the case of the LibMP communication models, giving some examples of their application.

4.2. Stream synchronous, CPU asynchronous model (SA)

As described previously, in this model communications are enqueued into a CUDA stream along with other CUDA tasks, like kernels, memory transfers, etc. Usually this model is relatively easy to use because it requires very few changes to the MPI application (i.e. modifying *MPI_Isend* with *mp_isend_on_stream*, neglecting CUDA synchronization primitives). Computation and communication tasks are executed asynchronously with respect to the host code but synchronized to the CUDA streams.

The class of applications introduced in Section 4.1 can be leveraged with the SA model if it is possible to change the original algorithm in order to be coherent with the following Formula (2) (represented by Fig. 9):

$$\begin{aligned} TCPU_{SA} &= R \times [\sum_{i=1}^{X}(LA_i + LS_i) + \sum_{j=1}^{Y} LB_j + \sum_{k=1}^{Z}(LW_k + LC_k)] \\ TGPU_{SA} &= R \times [\sum_{i=1}^{X}(A_i + S_i) + \sum_{j=1}^{Y} B_j + \sum_{k=1}^{Z}(W_k + C_k)] \\ T_{SA} &= \max(TCPU_{SA}, TGPU_{SA}) \end{aligned} \tag{2}$$

where LS_i and LW_k are respectively the time spent by CPU to enqueue the send or to wait for receive operations on the CUDA stream. In this model, the T_{idle} time can be considered negligible because, due to the asynchronous behavior, the CPU enqueues a lot of sequential tasks on the CUDA stream without waiting for their completion.

To ensure an asynchronous behavior, during communication periods it is required that:

- all the CUDA synchronization primitives must be removed
- all the non-asynchronous CUDA primitives must be replaced with the respective CUDA asynchronous primitives

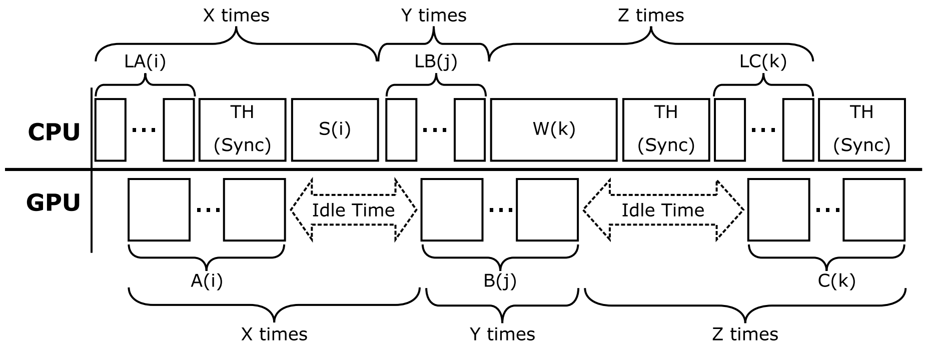

Fig. 8. Typical timeline of an iterative multi-GPU domain-decomposed MPI application.

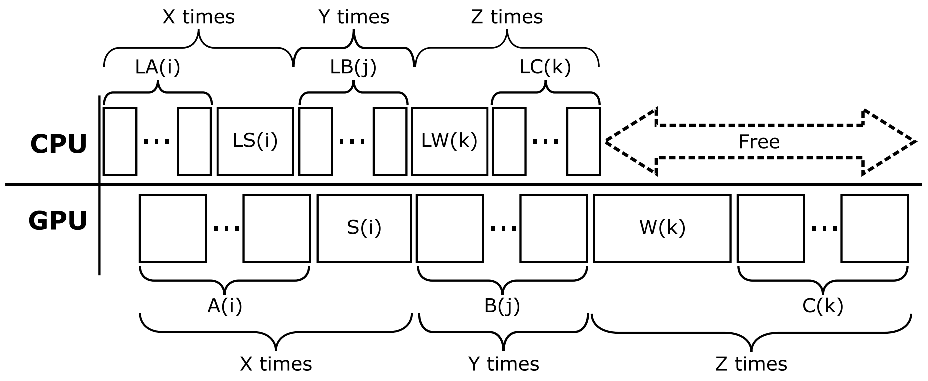

Fig. 9. General communication pattern of a multi-GPU with SA model application, Formula (2) representation.

- communication parameters must be known at the time of posting (for example send or receive buffers size, destination ranks, pointers, etc.)
- all the MPI functions must be replaced by LibMP functions.

An apparent side effect is that the CPU has less work to do because host code does neither synchronizations nor communications, thus it is not relevant in an asynchronous context; for this reason we can consider negligible the *TH* parameter in Formula (2).

An algorithm that is coherent with Formula (2) represents an improvement with respect to a synchronous version if the following 3 conditions are verified.

4.2.1. C1 condition: asynchrony

In Formula (2) the total execution time is equal to the GPU time if:

$$TCPU_{SA} < TGPU_{SA} \rightarrow T_{SA} = TGPU_{SA}.$$

That is the time required by the CPU to enqueue tasks on the CUDA stream (launch time) must be less than the time spent by GPU to execute those tasks (C1 condition):
$(C1): \ \sum(LA + LS + LB + LW + LC) < \sum(A + S + B + W + C)$

Without this condition, the asynchrony cannot happen, because the CPU launch time is greater than the GPU execution time.

4.2.2. C2 condition: time gain

The SA model (Formula (2)) is faster than the synchronous model (Formula (1)) if:

$$T_{SA} < T_S \rightarrow TGPU_{SA} < TCPU_S.$$

This is always verified with the stronger condition:

$$TGPU_{SA} < TGPU_S$$

because $TGPU_S \leq TCPU_S$. If GPU computation tasks require about the same time in both models:

$$[\sum(A + B + C)]_{SA} \leq [\sum(A + B + C)]_S.$$

Then, we obtain a simple condition:
$(C2): [\sum(TS) + \sum(TW)]_{SA} \leq [T_{idle}]_S$

which means that the SA model is faster if the sum of the communication times (send *TS* and wait *TW*) is less than the GPU T_{idle} time (the idle time spent by GPU waiting for CPU work) in the synchronous model. Intuitively the effectiveness of SA model relies on the relative magnitude of CUDA stream synchronous communications in the SA model with respect to CPU initiated communications plus GPU synchronizations in the S model.

4.2.3. C3 condition: fragmented computations

The larger the number of sub-tasks R, X, Y and Z, the more the execution becomes asynchronous (due to C1).
$(C3): \ R > 0, Y \geq 0, \max(X, Z) > 0$

In Section 6 we apply those conditions to several MPI + CUDA applications.

4.3. Kernel-Initiated model (KI)

The Streaming Multiprocessors (SMs), which are in charge of executing the CUDA kernels, can directly issue communication primitives to send messages or wait for receive completion. Having the HCA doorbell registers and CQs mapped in the GPU, a CUDA thread can use simple value assignments and comparisons inside the code to ring the doorbell or poll the CQs, respectively. In the KI model we used the *kernel fusion technique*, where in a single CUDA kernel both communication (send or wait) and computation tasks (type A, B or C) are fused together. This approach can lead to GPU memory consistency problems in case it is combined with GPUDirect RDMA [19]. In order to avoid this issue, in our

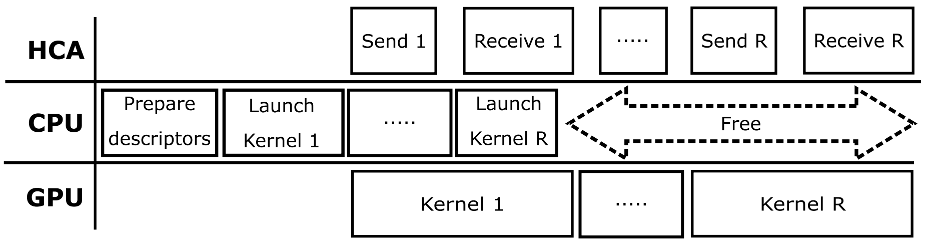

Fig. 10. Kernel-Initiated communication pattern timeline.

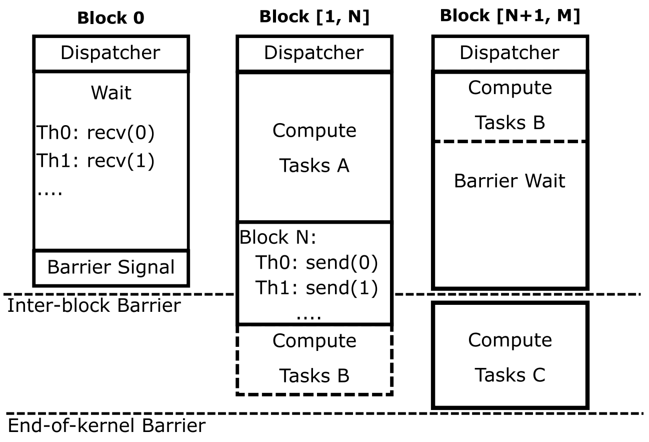

Fig. 11. KI kernel, CUDA blocks tasks.

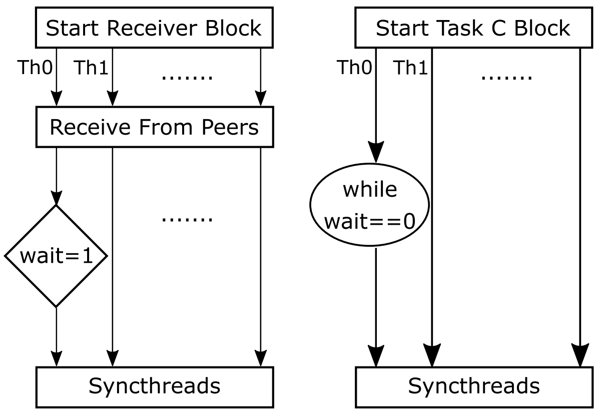

Fig. 12. Inter-block barrier.

benchmarks (Section 6) we use host memory for communication buffers. A typical timeline for the Kernel-Initiated model is shown in Fig. 10.

As in the SA model, the CPU prepares the communication descriptors and later those communications are triggered directly by threads in the CUDA kernels KI (using descriptors) instead of being triggered by the CUDA streams as in the SA model. The complexity is moved to the CUDA kernels KI, which requires at least $N+M+1$ blocks, where N is the number of blocks required to compute type A tasks before the send operation, M is the number of blocks required by type C tasks working on received data plus 1 block, used to poll the CQs as shown in Fig. 11. As in SA model, tasks B represent other (possible) CUDA tasks not related to communications, that can be performed by either kind of blocks (as shown in the Figure) depending on the particular algorithm.

In the KI model the kernel fusion technique is combined with a dynamic dispatcher which uses atomic operations to pick each thread block[2] and to assign it to the right task according to the following rules:

- In order to avoid dead-locks, the *receive* operation must not prevent the *send* operation from starting or progressing: there must be always, at least, one block waiting for receiving and an other block executing the send.
- Receiving is time critical in our experience, so the first *receiver* block is used to wait for incoming messages. In particular each thread *polls* on the receive CQ associated to each remote node.
- The blocks from the second up to the $N+1$-th are assigned to the A group of operations, whereas the remaining M blocks assigned to the C group of operations wait for the *receiver* block to signal that all incoming data have been received.
- An *inter-block barrier* scheme [36] is used to synchronize the *receiver* and the *taskC* blocks, as explained in Fig. 12, where the thread 0 of each *taskC* block waits for the thread 0 of the *receiver* block to set a global memory variable to 1, whereas the remaining threads move to the *__syncthreads()* barrier. To prevent a waste of CUDA blocks waiting for the *receiver*, *TaskA*, *TaskB* and *send* should always be executed before *TaskC*.
- When this happens (after the receive completion), all threads 0 in the *taskC* blocks will reach the matching *__syncthreads()* barrier and then start to unpack the received data.

For the same class of applications introduced in Sections 4.1 and 4.2, the KI model execution time can be estimated as:

$$
\begin{aligned}
TCPU_{KI} &= R \times LK \\
TGPU_{KI} &= R \times \{\sum_{i=1}^{X}(A_i+S_i)+\sum_{k=1}^{Z}(W_k+C_k)+\sum_{j=1}^{Y}B_j \; - O_{KI}\} \\
O_{KI} &= Overlap([\sum_{i=1}^{X}(A_i+S_i)], [\sum_{k=1}^{Z}(W_k+C_k)], [\sum_{j=1}^{Y}B_j], \\
&\quad \#SM, dispatcher, communications\ pattern, \ldots) \\
T_{KI} &= \max(TCPU_{KI}, TGPU_{KI})
\end{aligned}
\tag{3}
$$

where A_i+S_i is the time spent by *sender* blocks to execute tasks A plus send and W_k+C_k is the time spent to wait data and execute tasks C; B_j represents other (possible) tasks not related to communications (like working on internal structures), that can be performed by any type of blocks (A blocks, C blocks or other blocks).

Empirically, we measured that $TCPU_{KI}$ is always negligible, therefore we can consider $T_{KI} = TGPU_{KI}$.

When running on a GPU, multiple blocks of a CUDA kernel can be executed concurrently, therefore the execution of tasks $A+S$,

[2] There is no guarantee about the order in which blocks are scheduled by the GPU HW.

tasks $W + C$ and tasks B can overlap. To represent this magnitude, we defined *Overlap* that is a non-trivial function representing the overlap time among all the tasks considering several input parameters like the number of GPU SMs, the task dispatching algorithm, the particular communication patterns, computation times of tasks A, B and C, etc.

The gain of the KI model with respect to the SA model can be described as:

$$T_{KI} = T_{SA} - O_{KI}.$$

The best scenario is when all tasks fit in the logical number of CUDA blocks available on the GPU and tasks overlap for the most part of the execution:

$$\sum(A+S) + \sum(W+C) + \sum(B) \simeq 2 \times O_{KI}.$$

On the contrary, the worst scenario is when each of the tasks require an high number of blocks, decreasing the importance of the times overlap:

$$\sum(A+S) + \sum(W+C) + \sum(B) \gg 2 \times O_{KI}$$

which means that A, B and C tasks will be executed almost sequentially, like in case of the SA model, and the overlap in Formula (3) does not represent a real improvement in respect to Formula (2).

In Section 6, we explain when it is more convenient to use the KI model or the SA model.

5. Micro-benchmarks

In this section we discuss a few micro-benchmarks in the three variations, i.e. standard MPI, SA and KI model. We start with the ping-pong latency benchmark, based on point-to-point (send–receive) communications. Then we take a quick diversion to analyze the performance of waiting in the SA model. Finally we consider a variations of the first benchmark but using one-sided primitives. These tests are designed to expose both the communication latency and the CUDA kernel launch overhead. Here the size of the messages and the duration of the kernels are treated as independent parameters, even though most of the times in real applications those are closely related, e.g. in 3*D* stencil applications communications are $O(L^2)$ while computations are $O(L^3)$.

For all the micro-benchmarks we run 1000 warm-up iterations and take 1000 samples. The test environment consists of two Supermicro servers, each hosting a single Tesla K40m GPUs (using boosted clock at 875 MHz) and a Mellanox Connect-IB card (56 Gbps bandwidth). The MPI implementation is OpenMPI v1.10.7.

5.1. Ping-pong latency benchmark

Algorithm 2 depicts a simple ping-pong test between two MPI processes, *rank0* and *rank1*, exchanging some data placed in host memory and optionally executing a constant-time CUDA kernel in order to simulate a GPU computation consuming the data received or preparing the data to be sent.

Algorithm 2 Ping-pong latency with kernel compute

```
1: procedure PINGPONGKERNEL(myRank, msg_size, calc_size, use_kernel, communication_type)
2:     for i=0 to NUM_ITERS do
3:         post_recv(msg_size);
4:     end for
5:     for i=0 to NUM_ITERS do
6:         if myRank == rank0 then
7:             wait_recv(rank1);
8:             if use_kernel == 1 then
9:                 calc_kernel<<< ... >>>(calc_size);
10:                if communication_type != SA then
11:                    cudaDeviceSynchronize();
12:                end if
13:            end if
14:            send(rank1, buf, msg_size);
15:        else
16:            send(rank0, buf, msg_size);
17:            wait_recv(rank0);
18:            if use_kernel == 1 then
19:                calc_kernel<<< ... >>>(calc_size);
20:                if communication_type != SA then
21:                    cudaDeviceSynchronize();
22:                end if
23:            end if
24:        end if
25:    end for
26: end procedure
```

In Fig. 13, we considered the MPI, SA model and KI model cases without the CUDA kernel, to get the baseline ping-pong latency.

Because there is no computation involved in this case, i.e. no GPU kernels are launched, and the data buffers are in CPU memory, the MPI results are in line with the expectations, i.e. in the order of the microsecond for the half round-trip latency — note that full round-trip latency is plotted in the figure. We note that traditional CPU driven communications are faster for small message sizes. This is partly due to overheads in the GPU communications path,[3] which shows up as a constant per-operation overhead of roughly 2.5 μs. Latency grows linearly as the message size increases, but the latency of the SA model (red circles) is more irregular than both the MPI (blue squares) and the KI model (green triangles), with piece-wise constant periods interleaved with sudden peaks; to clarify this behavior, we need to explain how communications are carried out by the CUDA Stream.

5.2. Notification waiting in the SA model

As described in Section 3.2, in case of the SA model, multiple *cuStreamWriteValue32()* MemOps are used to trigger the send operation while *cuStreamWaitValue32()* is used to stall the CUDA stream while waiting for both receive and send completions. During the wait, the GPU front-end unit is responsible for polling the CQE associated to each communication operation. The polling frequency depends on the GPU internal stream scheduling. We define *Wait Reaction Time* (WRT) as the time between the update of the CQE, by the HCA in our case, and the GPU observing the updated value.

To estimate the WRT on our K40m GPUs, we wrote a standalone micro-benchmark, *poll-latency*, where a pinned host buffer is updated by the CPU, simulating the CQE update by the HCA, and polled using CUDA MemOps APIs on a single stream, as shown in the interaction diagram in Fig. 14. Specifically, in a loop, the CPU calls *cuStreamWaitValue32(stream, i, ptr1, GEQ)* and *cuStreamWriteValue32(stream, i, ptr2)*, then sleeps for Δt microseconds – to let the associated commands be fetched by the GPU front-end unit, – sets `*ptr1=i` and then measure the time spent before observing `*ptr2==i`.

In Fig. 15 we plot WRT $=$ WRT(Δt) for the Tesla K40m, where it appears to be a sawtooth wave function of the sleep time with a period of 12 μs. Repeating the test with multiple active CUDA stream N_{AS}, if actively polling, shows one additional microsecond per stream:

$$\mathrm{WRT}(\Delta t, N_{AS}) = \mathrm{WRT}_{min} + F(\Delta t) + N_{AS}. \quad (4)$$

[3] The expensive GPU fence operations, required when injecting work on the HCA, as explained in 3.2 as well as the use of non-inlined HCA commands.

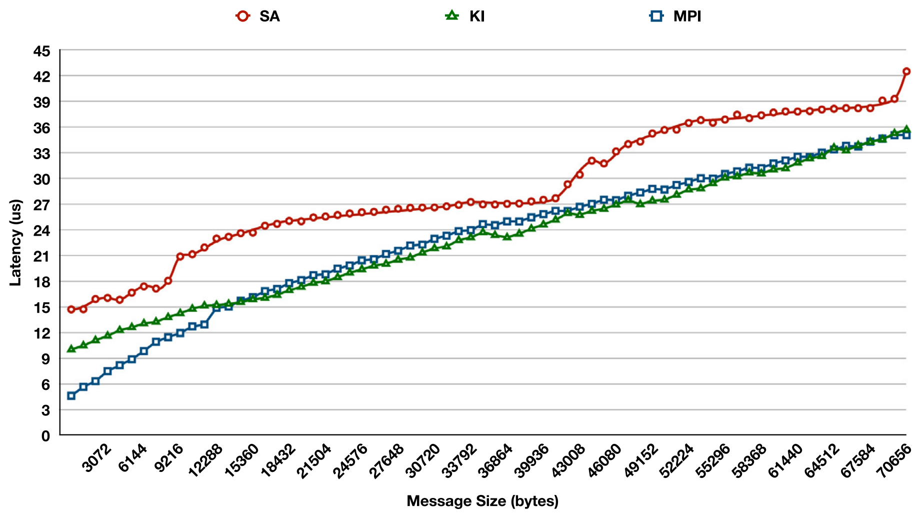

Fig. 13. Ping-Pong microbenchmark with MPI, SA model and KI model, communications only. Full round-trip time is plotted. 1000 iterations, excluding warmup. K40m GPU, Mellanox Connect-IB HCA, FDR IB switch. Open MPI default eager limit is 12 kB.

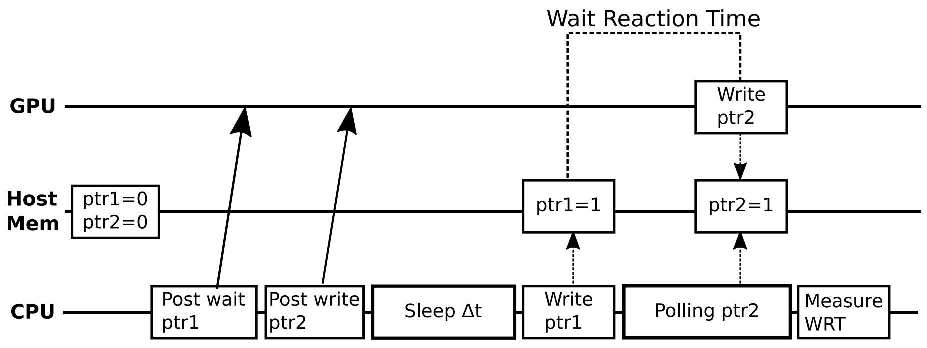

Fig. 14. The poll-latency micro-benchmark, used to measure the GPU Stream Wait Reaction Time (WRT).

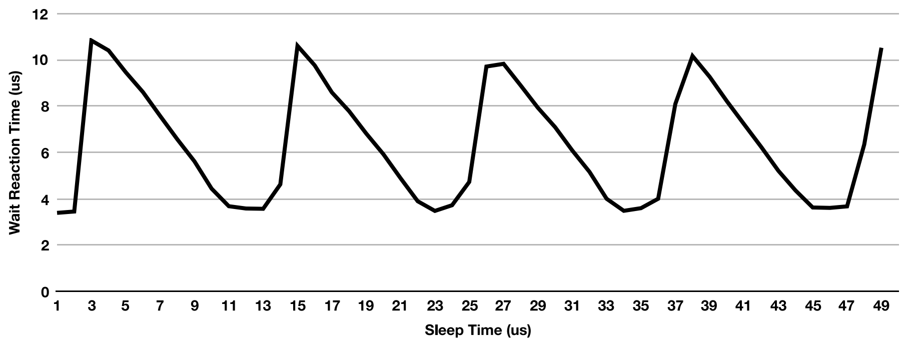

Fig. 15. Wait Reaction Time curve , GPU Telsa K40m, single active CUDA stream.

This is compatible with the GPU front-end unit circulating among the active streams, each polling taking one microsecond, with a pause of ten microseconds at the end of every cycle.

To qualitatively show how WRT influences the SA model latency in the ping-pong benchmark, we measured the IB Verbs send latency at varying message size, see red line in Fig. 16, using the standard *ib_send_lat* test (Mellanox Perftest 5.6 package [26]) on a couple of Supermicro servers. The blue line is our previous measurement of WRT plotted on a time scale rather than the message size, using the equivalence 1 µs $\simeq$ 6 kB, i.e. assuming 6 GB/s for the bandwidth of the Connect-IB used in this experiment. The yellow line in Fig. 16 is the sum of the blue (WRT) and the red (send latency) line and is qualitatively similar to the SA model ping-pong latency in Fig. 13. We conclude that the piece-wise constant behavior of the SA model in Fig. 18 is related to the polling pattern of the GPU front-end unit.

5.2.1. *Polling on newer GPU architectures*

Our experiments show that WRT varies across different GPU architectures. Running the *poll-latency* test on a GPU with Maxwell architecture shows to similar results as in Fig. 15 while on the newest Pascal architecture we obtained better results, as shown in Fig. 17.

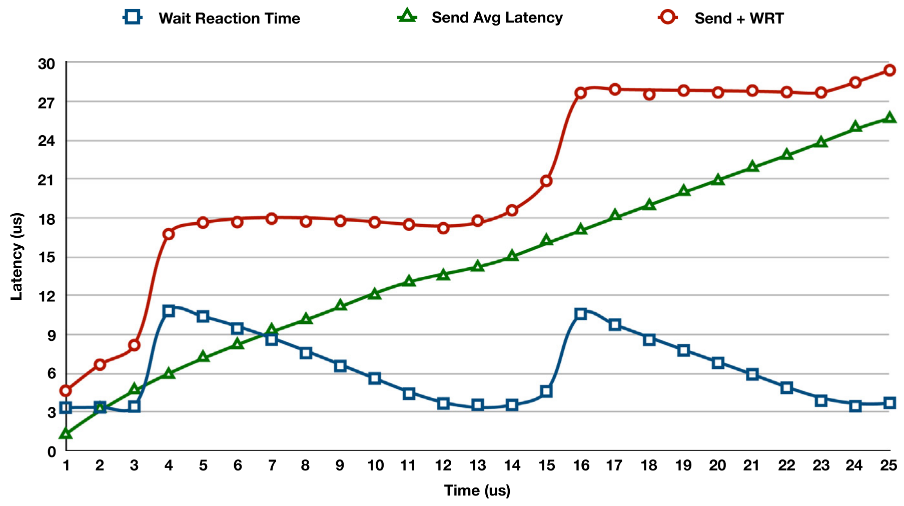

Fig. 16. Ping-pong qualitative dependence from WRT.

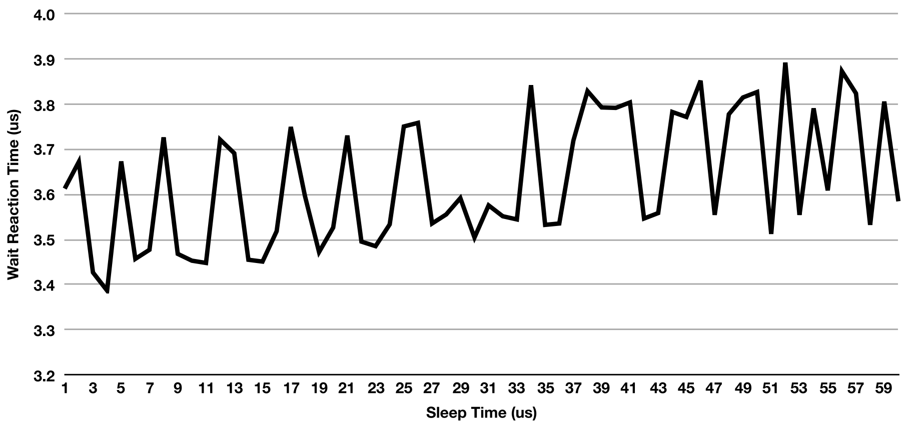

Fig. 17. Wait Reaction Time curve trend, GPU Tesla P100, Pascal architecture.

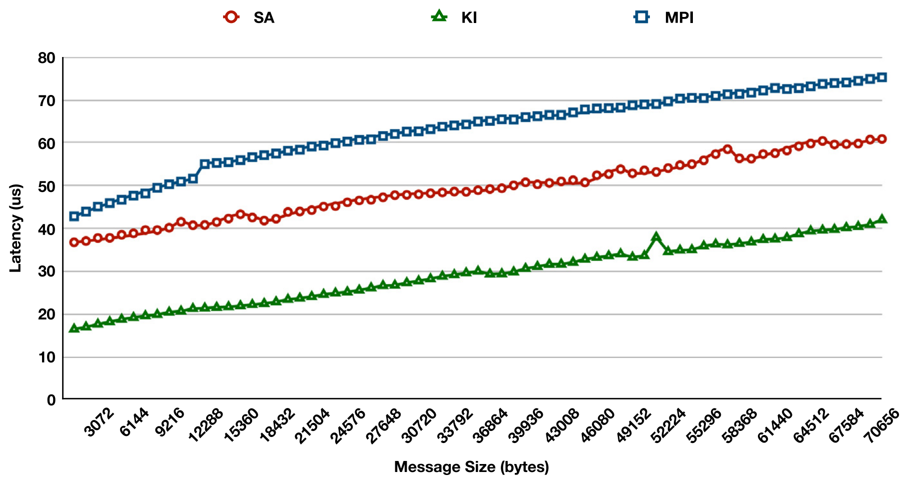

Fig. 18. Ping-Pong micro-benchmark with MPI, SA model and KI model, communications and computation, Round-Trip Time latency.

5.3. *Ping-pong latency with GPU compute*

In Fig. 18 we plot the round-trip latency when enabling a $\sim 5\ \mu s$ CUDA kernel computation, in the three cases considered.

The overall performance depends on both the GPU computation and the communication:

- MPI: as visible in the profiler timeline – Fig. 19 – *rank0* waits for the receive completion, launches the CUDA kernel, synchronizes with the CUDA stream and finally sends data.
- SA model: in Fig. 20, all communications and CUDA kernels are posted on the GPU Stream and executed about 3ms later. CPU and GPU work completely overlap.

- KI model: here we fuse together the polling on the receive completion, the constant time compute and the trigger for the send into a new single kernel. That fused kernel is launched multiple times in a loop. From the profiler point of view, there is only a sequence of kernels on the CUDA Stream.

We note that the KI model is the best performing. Besides the piece-wise constant behavior noticed for the SA model is not present in this case.

6. Applications benchmarks

GPUDirect Async is a new experimental technology, therefore we need to study its behavior into real MPI applications in order to verify if and when GPUDirect Async can improve performance. We choose several different multi-GPU MPI applications (with the constraint of a single GPU for a single process) having the communication periods similar to the one described in Section 4, replacing the synchronous communication calls with the Async calls, testing both SA and KI models. For our benchmarks we used the Wilkes cluster [35] in Cambridge University (UK). The system consists of 128 Dell T620 servers, 256 NVIDIA Tesla K20c GPUs interconnected by 256 Mellanox Connect IB cards, having the CUDA 8.0 Toolkit, and OpenMPI version 1.10.3. At the time of benchmarks, LibGDSync was on top of OFED 3.4 with an additional patch (recently included in OFED 4.0) to enable GPUDirect Async in case of IB Verbs.

6.1. HPGMG-FV CUDA

HPGMG [20] is an HPC benchmarking effort developed by Lawrence Berkeley National Laboratory. It is a geometric multigrid solver for elliptic problems using Finite Volume (FV) [13] and Full Multigrid (FMG) [14] methods where the solution to a hard problem (a continuous problem represented by a finer grid of elements by means of discretization) is expressed as a solution to an easier problem (coarser grid of element). In case of multi-process execution, the workload is fairly distributed among processes: in order to improve the parallelization, each problem level is divided into several same-size boxes. HPGMG-FV takes as input the number and the $log_2(size)$ of the finest level boxes and calculates the size of all the other (smaller) levels. In HPGMG-FV CUDA version (improved by NVIDIA [33]) finer levels are processed by GPU while coarser levels by the CPU (according to a threshold on the level's number of elements). Considering a multi-process execution of HPGMG-FV CUDA the main and most used communication function, in case of GPU finer levels, follows a 2D Stencil pattern that is similar to the one described in Section 4.1:

1. **Pack**: a single CUDA kernel which stores its result data into the send buffers (type A task)
2. **Send**: Synchronize with the CUDA stream and send result data
3. **Interior computation**: a single CUDA kernel working on internal structures (type B task)
4. **Receive**: Receive data from other processes
5. **Unpack**: a single CUDA kernel computation on received data (type C task)

In Algorithm 3 there is the MPI pseudo-code of this communication function.

Communication pattern is schematized in Fig. 21 and it appears on the CUDA Visual Profiler as in Fig. 22: between CUDA kernels, the GPU is unloaded waiting for new tasks from the CPU (green IDLE areas).

We applied both SA and KI models, modifying the HPGMG-FV CUDA algorithm according to the models described in Sections 4.2 and 4.3.

Algorithm 3 HPGMG-FV, communication function, MPI version

```
procedure EXCHANGELEVELBOUNDARIESMPI(...)
    cudaDeviceSynchronize();
    for $peer \in \{0, \dots, numPeers - 1\}$ do
        MPI_Irecv(recvBuf, ..., recvReqs[p]);
    end for
    pack_kernel<<< ..., stream >>>(sendBuf, ...);
    cudaDeviceSynchronize();
                                        ▷ Overlap between send and local kernel
    for $peer \in \{0, \dots, numPeers - 1\}$ do
        MPI_Isend(sendBuf, ..., sendReqs[p]);
    end for
    interior_kernel<<< ..., stream >>>(...);
    MPI_Waitall(recvReqs, ...);
    unpack_kernel<<< ..., stream >>>(recvBuf, ...);
    MPI_Waitall(sendReqs, ...);
end procedure
```

6.1.1. SA model

The host code between CUDA kernels and communications consists of a simple *cudaDeviceSynchronize()* after the *Pack* kernel that can be easily removed (negligible TH value); in addition, communication parameters are known at posting time. Therefore the SA model can be applied to HPGMG-FV and, considering Formula (2), the constant values are (C3 condition):

$R > 0, X = Z = Y = 1.$

We modified the *exchangeLevelBoundariesMPI* function in Algorithm 3 with the *exchangeLevelBoundariesSA* in Algorithm 4 . In case of InfiniBand communications, if a send is posted but the corresponding receive is not ready yet, there are some delays during communications. For this reason here we used the one-sided asynchronous call *mp_iput_on_stream* to ensure that each asynchronous send has the corresponding receive buffer posted by the opposite peer.

Algorithm 4 HPGMG-FV, communication function, SA model version

```
procedure EXCHANGELEVELBOUNDARIESSA(...)
    for $peer \in \{0, \dots, numPeers - 1\}$ do
        mp_irecv(recvBuf, peer, ..., recvReqs[p]);
        mp_iput_on_stream(remoteAck[peer], ..., stream);
    end for

    pack_kernel<<< ..., stream >>>(sendBuf, ...);

    for $peer \in \{0, \dots, numPeers - 1\}$ do
        mp_wait_word_on_stream(localAck[peer], ..., stream);
        mp_isend_on_stream(sendBuf, ..., sendReqs[p], stream);
    end for

    interior_kernel<<< ..., stream >>>(...);
    mp_wait_all_on_stream(recvReqs, ..., stream);
    unpack_kernel<<< ..., stream >>>(recvBuf, ...);
    mp_wait_all_on_stream(sendReqs, ..., stream);
end procedure
```

The SA model communication pattern is represented in Fig. 23 and it appears on the CUDA Visual Profiler as in Fig. 24: the CPU posts tasks but they are executed several microseconds later.

Considering a 4 process execution having 8 boxes with $log_2(size) = 4$ only the finest level runs on the GPU; in that case using the NVIDIA Visual profiler we found that moving from MPI to SA model led to a decrease of about 64% of the $TCPU_{SA}$ work time (due to $TH \rightarrow 0$) with respect to $TCPU_S$ and that the $TGPU_{SA}$ computation was 49% longer than the $TCPU_{SA}$ to complete its tasks (condition C1 is satisfied); furthermore we measured a decrease of about 28% of the $TGPU_{SA}$ time (condition C2 is satisfied).

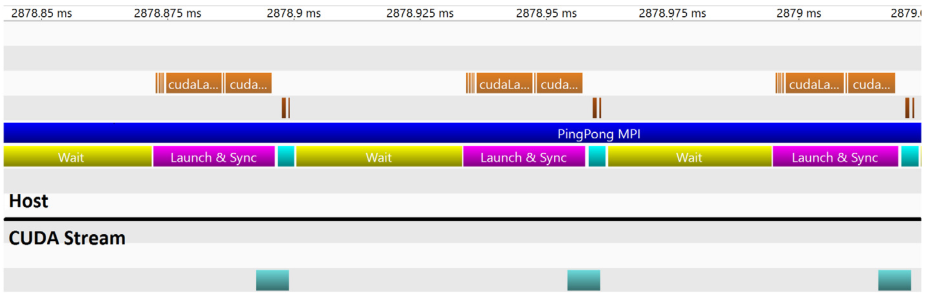

Fig. 19. Rank 0, Ping-Pong CUDA Visual Profiler with MPI plus CUDA kernel.

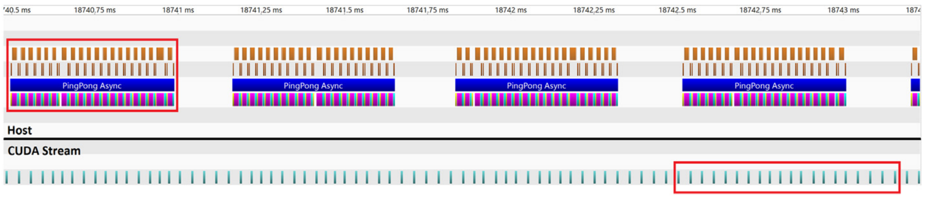

Fig. 20. Rank 0, Ping-Pong CUDA Visual Profiler with Async plus CUDA kernel.

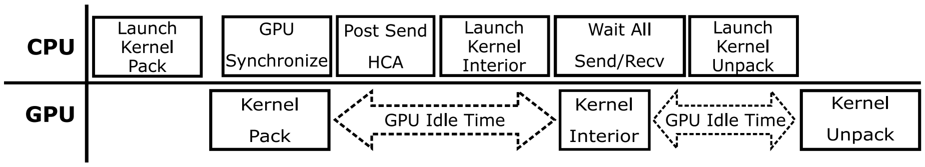

Fig. 21. HPGMG-FV, communication function timeline, MPI version.

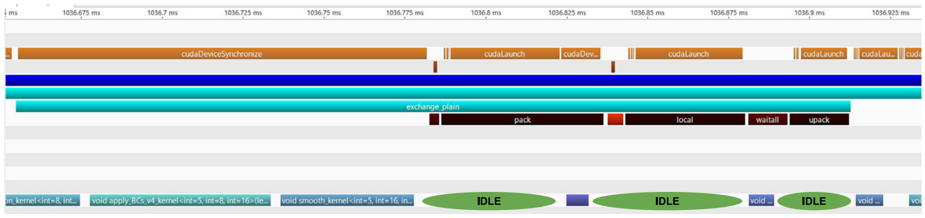
cudaDeviceSynchronize
cudaLaunch
cudaDev...
exchange_plain
pack
local
waitall
upack
IDLE
void apply_BCs_v4_kernel<int=5, int=8, int=16>(le...
void smooth_kernel<int=5, int=16, in...

Fig. 22. HPGMG-FV, communication function on CUDA Visual Profiler, MPI version.

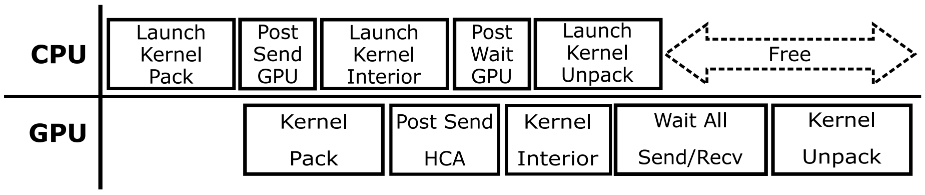

Fig. 23. HPGMG-FV, SA model, communication pattern timeline.

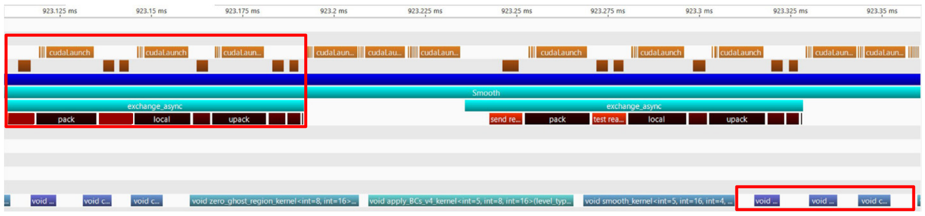

Fig. 24. HPGMG-FV, SA model, CUDA Visual Profiler.

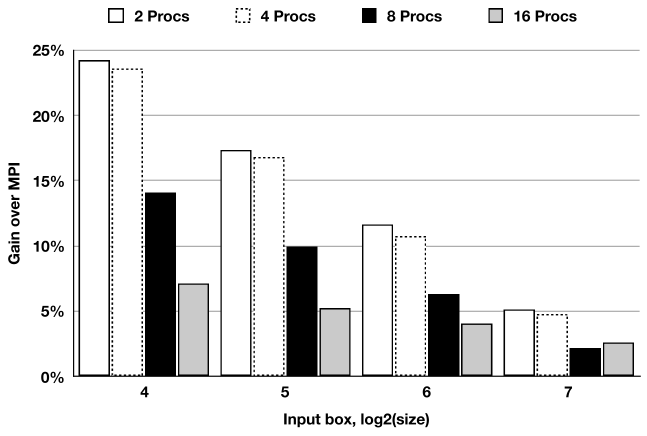

Fig. 25. HPGMG-FV SA implementation time gain with respect to MPI, GPU levels only, up to 16 processes, weak-scaling.

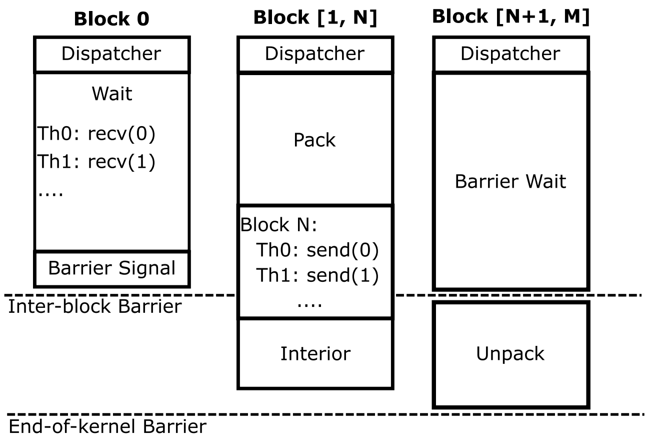

Fig. 26. HPGMG-FV communication pattern, KI model version.

In Fig. 25 the Y axis shows the performance increase of the GPU levels in case of this SA implementation compared with the standard MPI version in the Wilkes cluster using up to 16 processes. The maximum gain (about 24%) is reached in case of $\log_2(size) = 4$ with 2 processes, while performance gain decreases when increasing the size of the input level for two reasons:

- Message size grows with the box size, therefore communication overhead becomes less important. For large sizes, all communication methods should converge to the same performance level
- HPGMG-FV has weak-scaling, because the number of boxes per rank is always the same for each input size
- Increasing the size and amount of levels, increases the CUDA kernels workload (i.e. computation time) decreasing the GPU idle time; thus replacing a small amount of GPU idle time with communications on the CUDA Stream should not result into a significant performance improvement (C2 condition). Furthermore, if communication time is greater than idle time, there could be a negative gain.

6.1.2. KI model

According to previous observations, *Pack* kernel is a type A task, *Unpack* kernel is a type C task and *Interior* kernel is a type B task. Therefore we modified each communication period using a single kernel organizing CUDA blocks as in Fig. 26.[4]

The type B task is computed by type A blocks after the send operation in order to overlap with the Wait and Unpack operations.

Considering the previous 4 processes execution with 8 boxes and $\log_2(size) = 4$, in SA model the sum of CUDA blocks required by *Pack*, *Interior* and *Unpack* kernels is 193, each one having 64 threads. Each thread needs almost 37 registers and no shared memory is required. The Wilkes cluster has Tesla K20 GPUs, with 13 SMs and 2048 maximum threads per SM, which means that all the 193 logical CUDA blocks can be executed concurrently and all tasks can overlap for the most part of the time, leading to the best KI model scenario (Section 4.3). Observing with the NVIDIA Visual profiler, in case of KI model implementation we got a reduction of $TCPU_{KI}$ and $TGPU_{KI}$ time as summarized in Table 2.

We plot in Fig. 27 the KI model implementation performance gain. The maximum gain is 26% in case of 2 processes with $\log_2(size) = 4$ box size and, generally speaking, the performance are better than the SA model performance.

[4] "Receive" is the combination of receive ack put before wait while "Send" is the combination of the receive ack wait before the send, as in SA model.

Table 2
HPGMG-FV KI model time reduction.

Compared to	CPU time	GPU time
MPI	77%	32%
Async	37%	4%

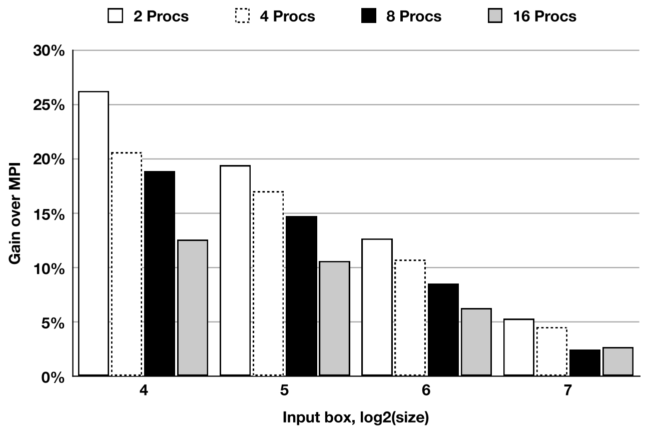

Fig. 27. HPGMG-FV KI implementation time gain with respect to MPI, GPU levels only, up to 16 processes.

6.2. CoMD-CUDA

CoMD is one of several proxy-apps developed as part of the *ExMatEx* project [9] and is a reference implementation of typical classical molecular dynamics algorithms.

In particular, it considers materials where the inter-atomic potentials are short range and the simulation requires the evaluation of all forces between atom pairs within the cutoff distance; that is, considering a multi-node distributed execution, the atoms on each node are assigned to cubic link cells and each atom only needs to test all the other atoms within its own cell and the 26 neighboring link cells in order to guarantee that it has found all possible interacting atoms. Only two different potentials are implemented: the Lennard-Jones (LJ) and the Embedded Atom Method (EAM). NVIDIA enhanced the original CoMD implementation with CUDA (see [7] and [9]). Considering the EAM potential, there are two different communication functions: force exchange and atoms exchange, where CoMD-CUDA repeats for 3 times (x, y, and z dimensions) the following steps: load data in 2 different buffers (one for atom's moments and one for atom's positions) by means

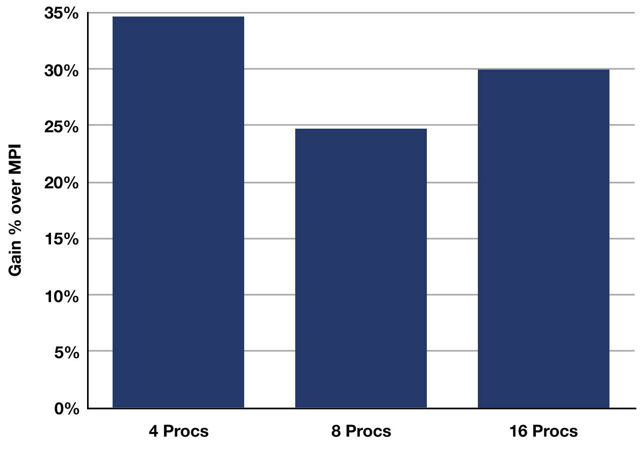

Fig. 28. CoMD-CUDA time gain, SA model on MPI, communication periods only, Wilkes cluster, weak-scaling.

of 2 CUDA kernels (tasks A), synchronize with the GPU and send data, execute 2 CUDA kernels (tasks B), receive data and process them using 2 CUDA kernels (tasks C).

6.2.1. SA model

Applying the SA model, the values of constants in Formula (2) are (C3 condition):

$$\begin{cases} R = 3 & \rightarrow \quad \text{low number of sequential communication periods} \\ X = 2 & \rightarrow \quad \text{2 CUDA kernels and 2 send} \\ Y = 2 & \rightarrow \quad \text{2 CUDA kernels between send an receive} \\ Z = 2 & \rightarrow \quad \text{2 CUDA kernels and 2 receive.} \end{cases} \tag{5}$$

We considered executions with 4, 8 and 16 processes having a distributed set of 2.048.000 atoms. The performance gain of SA on MPI in Fig. 28 refers to communication periods only.

Removing all the synchronizations with the CUDA stream and transforming some mandatory host code into CUDA kernel code (halving the computation time) we obtained a negligible *TH* in order to satisfy the C1 condition: considering the 4 processes run, there is a difference of about 27 ms between the CPU launch of the GPU tasks and their actual execution on the CUDA stream. This behavior led to a negligible T_{idle} time. $TGPU_{SA}$ (considering the case of 4 processes) is about 40% lower than $TGPU_S$ (measured with the NVIDIA profiler), satisfying the condition C2.

The overall CoMD-CUDA performance is dominated by the CUDA kernels computing the force between the atom pairs, whereas communications play a marginal role (about 7% of total time in case of 16 processes). For this reason, the great improvement of the SA model (between 25% and 35% time gain) in communications does not lead to a remarkable improvement of CoMD-CUDA total time (just about 5%), according to C2 time gain condition. The aim of this experiment is to provide another example of the advantage in using GPU Direct Async communications.

6.2.2. KI model

The communication pattern of CoMD-CUDA is similar to the HPGMG-FV one, therefore is easy to move communications into a single CUDA kernel as described before. Considering the benchmarks in the SA model, the problem resides on the number of blocks required by the CUDA kernel KI: N (task A) and M (task C) are very high number (an average of about 850 blocks), leading to the worst scenario for the KI model: the new KI kernel will require at least 1700 CUDA blocks, each one having 128 threads. Considering a Tesla K20 with a maximum of 2048 threads per SM, to have a whole parallel execution you should require more than 100 SM (the value of the *Overlap* function is very small). For this reason, our KI model implementation of CoMD-CUDA gave us similar results as the SA model.

6.3. BFS

M. Bisson et al. in [5] implemented a parallel distributed Breadth First Search algorithm on multi-GPU systems for large graphs, represented by the adjacency matrix and partitioned by means of a 2D decomposition. The graph is partitioned among the computing nodes by assigning to each one a subset of the original vertices and edges sets. The search is performed in parallel, starting from the processor owning the root vertex. During each step of the main loop, processors handling one or more frontier vertices follow the edges connected to them to identify unvisited neighbors. The reached vertices are then exchanged in order to notify their owners and a new iteration begins. The search stops when the connected component containing the root vertex has been completely visited. They adopted a technique to reduce the size of exchanged messages based on a fixed-size *bitmap*, when messages size exceeds a pre-defined threshold.

Each rank during an iteration in the main loop executes the following steps:

1. exchange vertices in the frontier with MPI point-to-point primitives;
2. by means of CUDA kernels, CUDA primitives and CUB functions [10], get neighbor vertices that have not been visited yet and put them into the next send buffer;
3. exchange vertices in the frontier with MPI point-to-point primitives;
4. by means of CUDA kernels, CUDA primitives and CUB functions, update the frontier;
5. evaluate the number of remaining vertices to be visited by means of the collective *MPI_Allreduce* function. If the number is 0, exit from the loop.

We faced several issues in porting the MPI version to the corresponding SA version.

6.3.1. Main loop iterations number

The number of iterations in the main loop is unknown (*R* parameter in performance model formulas, Section 4), because it stops only when step 5 is satisfied. Thus, even substituting the *MPI_Allreduce* with several point-to-point LibMP functions, there is the need of a synchronization at the end of each iteration in order to check the exit condition. This issue reduces a lot the asynchrony (C1 condition), having $R = 1, X = 1, Z = 1$ (C3 condition).

6.3.2. Computation parameters

Steps 2 and 4 use a number of CUDA tasks, like kernels and synchronous primitives, to compute at run-time some values useful to start next computation tasks. For the reasons explained in Section 3, this represents an issue for Async. In Algorithm 5 we reproduce an example of parameters *pA* and *pB* computed at run-time.

Algorithm 5 BFS synchronous pseudo-code

```
1: int pA, pB;
2: ....
3: cudaKernel1<<<...>>>(deviceBuffer, ...);
4: cudaMemcpy(& pA, deviceBuffer[bufferLenght], ...);
5: cudaKernel2<<<pA, ...>>>(pA, ...);
6: cub::DeviceScan::ExclusiveSum(deviceBuffer, pA, ...);
7: cudaMemcpy(& pB, deviceBuffer[pA], ...);
8: cudaKernel3<<<pB, ...>>>(pA, pB, ....);
```

To overcome this issue we needed to oversize the number of items in the CUB calls to the total number elements of the *deviceBuffer* and to fix the CUDA kernels grid size (according to TeslaK20

Algorithm 6 BFS asynchronous pseudo-code

```
1:  int * pA, * pB;
2:  allocateGPUMemory(pA, 1, sizeof(pA));
3:  allocateGPUMemory(pB, 1, sizeof(pB));
4:  ....
5:  cudaKernel1<<<...>>>(deviceBuffer, ...);
6:  cudaMemcpyAsync(pA, deviceBuffer[bufferLenght], ...);
7:  cudaKernel2<<<MAX_BLOCKS, MAX_THREADS>>>(pA, ...);
8:  cub::DeviceScan::ExclusiveSum(deviceBuffer, bufferLenght, ...);
9:  cudaKernel4<<< 1, 1 >>>(deviceBuffer, pA, pB);
10: cudaKernel3<<<MAX_BLOCKS, MAX_THREADS>>>(pA, pB, ....);
11: ....
12: function CUDAKERNEL4(deviceBuffer, pA, pB)
13:     pB[0] = deviceBuffer[pA[0]];
14: end function
15:
```

hardware), increasing the computation overhead as described in Algorithm 6:

The final *cudaMemcpy()* (row 7, Algorithm 5) has been transformed in *cudaKernel4()* (row 9 Algorithm 6).

6.3.3. Communication parameters

It is possible to use Async only when the fixed-length *bitmap* is used during communications, otherwise the size of the communication buffers is evaluated at run-time and is unknown at communication posting time (LibMP requirement, Section 3.3).

6.3.4. SA model benchmark

Considering all those issues, the final SA implementation had no improvements, i.e. $TGPU_{SA} = TGPU_S$. That is, even if $[\sum(TS) + \sum(TW)]_{SA} \leq [T_{idle}]_S$, the C2 condition hypothesis about the times required by computation tasks is not verified, because we measured (by means of the NVIDIA Visual Profiler) an increase of about 14% of time for the most important CUDA kernel (due to a fixed and non-optimized grid size) whereas CUB calls required about 3× the time required by the synchronous version (due to the oversizing of items), thus:

$$[\sum(TS) + \sum(TW)]_{SA} > [T_{idle}]_S.$$

7. Conclusion and future work

GPUDirect Async is a technology enabling direct control paths between GPUs and 3rd party devices. It has been initially released with CUDA 8.0. InfiniBand network support for Async comes in the form of a new set of experimental OFED verbs (Mellanox OFED 4.0 in [18]), a mid-layer abstraction library LibGDSync [17], and a sample message-passing library LibMP [24]. All the applications presented in this paper are available on GitHub: HPGMG-FV CUDA Async in [21] and CoMD-CUDA Async in [8]. In summary we note that:

- GPUDirect Async allows for a new communication model for multi-GPU accelerated applications.
- GPUDirect Async is not necessarily faster, for example when the GPU idle time is greater than communication time.
- The more the CPU can post several consecutive asynchronous communication periods, the greater the potential performance gain.

GPUDirect Async major drawbacks are:

- Communication parameters must be known before posting on the GPU Stream.
- If the GPU is overloaded or if the idle time is smaller than communication time, performance may actually decrease.

We remark here that GPUDirect Async is still under development. We are evaluating its efficacy in other domains, for example in combination with CUDA Multi Process Services (MPS), when multiple MPI processes share a single GPU. New optimizations are being explored, such as allocating InfiniBand elements (i.e. CQs) on the GPU memory, offloaded collectives. We plan to analyze the interaction of GPUDirect Async with GPUDirect RDMA, explaining how to overcome issues like memory consistency [19]. Up to now we used only Mellanox hardware, therefore we will explore new type of connections. Moreover we would like to test asynchronous applications using a higher number of nodes to better evaluate their scalability.

Given the design choices mentioned in Section 3.2, we expect to eventually be faced with scalability problems, e.g. the GPU having to poll $O(N)$ CQs. In a sense we already faced the limitations of polling in the SA model. We are currently working on improved designs to relax the current constraints. We also plan to explore combining GPUDirect Async with advanced HW features of modern NICs, such as HW tag matching and MPI protocol offloading.

Acknowledgments

The authors would like to thank Filippo Spiga, for the opportunity to experiment on the Wilkes cluster, and Massimo Bernaschi of the National Research Council of Italy, for reviewing the paper. We also thank Nikolay Sakharnykh for useful discussions and David Fontaine for major contributions to the CUDA MemOp APIs.

References

[1] E. Agostini, D. Rossetti, S. Potluri, Offloading communication control logic in GPU accelerated applications, in: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Madrid, Spain, May 14-17, 2017.
[2] R. Ammendola, et al., APEnet+: a 3D Torus network optimized for GPU-based HPC Systems, J. Phys. Conf. Ser. 396 (2012) IOP Publishing.
[3] R. Ammendola, et al., 2013. GPU peer-to-peer techniques applied to a cluster interconnect, in: CASS 2013 workshop at 27th IEEE International Parallel & Distributed Processing Symposium, IPDPS.
[4] R. Ammendola, et al., NaNet: a flexible and configurable low-latency NIC for real-time trigger systems based on GPUs, J. Instrum. 9 (2014).
[5] M. Bisson, M. Bernaschi, E. Mastrostefano, Parallel distributed breadth first search on the kepler architecture, IEEE Trans. Parallel Distrib. Syst. 27 (7) (2016).
[6] Chelsio GDR, http://www.chelsio.com/gpudirect-rdma.
[7] CoMD https://github.com/ECP-copa/CoMD.
[8] CoMD-CUDA Async on GitHub. https://github.com/e-ago/CoMD-CUDA-Async.
[9] CoMD-CUDA Code. https://github.com/NVIDIA/CoMD-CUDA.
[10] CUB CUDA. https://nvlabs.github.io/cub.
[11] CUDA-Aware MPI. https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi.
[12] F. Daoud, A. Watad, M. Silberstein, GPUrdma: GPU-side library for high performance networking from GPU kernels, in: Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, Article No. 6.
[13] Finite Volume method. https://en.wikipedia.org/wiki/Finite_volume_method.
[14] Full MultiGrid method. https://en.wikipedia.org/wiki/Multigrid_method.
[15] GDRcopy. https://github.com/NVIDIA/gdrcopy.
[16] GPUDirect family, https://developer.nvidia.com/gpudirect.
[17] GPUDirect LibGDSync. http://github.com/gpudirect/libgdsync.
[18] GPUDirect libmlx5. https://github.com/gpudirect/libmlx5.
[19] GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/#design-considerations.
[20] HPGMG, https://hpgmg.org.
[21] HPGMG-FV CUDA Async on GitHub. https://github.com/e-ago/hpgmg-cuda-async.
[22] InfiniBand Standard. http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[23] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, GPUnet: Networking Abstractions for GPU Programs, in: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, October, 2014.
[24] LibMP on GitHub. https://github.com/gpudirect/libmp.
[25] Mellanox GDR, http://www.mellanox.com/page/products_dyn?product_family=116.
[26] Mellanox Perftest. https://community.mellanox.com/docs/DOC-2802.
[27] L. Oden, H. Froning, GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters, in: Cluster Computing, CLUSTER, 2013 IEEE International Conference on.
[28] L. Oden, H. Fröning, F. Pfreundt, Infiniband-Verbs on GPU: A case study of controlling an Infiniband network device from the GPU, in: Parallel & Distributed Processing Symposium Workshops, IPDPSW, 2014 IEEE International.
[29] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, D.K. Panda, Efficient Inter-node MPI communication using GPUDirect RDMA for infiniband clusters with NVIDIA GPUs, in: Proceedings of the 42nd International Conference Parallel Processing, ICPP, 2013.
[30] D. Rossetti, Mellanox booth talk at Supercomputing 2016. https://www.youtube.com/watch?v=eO2tTVo8ALE.
[31] D. Rossetti, E. Agostini, How to enable NVIDIA CUDA stream synchronous communications using GPUDirect, http://on-demand.gputechconf.com/gtc/2017/presentation/s7128-davide-rossetti-how-to-enable.pdf.
[32] D. Rossetti, S. Potluri, D. Fontaine, State of GPUDirect technologies, http://on-demand.gputechconf.com/gtc/2016/presentation/s6264-davide-rossetti-GPUDirect.pdf.
[33] N. Sakharnykh, High-Performance Geometric Multi-Grid with GPU Acceleration. https://devblogs.nvidia.com/parallelforall/high-performance-geometric-multi-grid-gpu-acceleration.
[34] A. Venkatesh, K. Hamidouche, S. Potluri, D. Rossetti, C.H. Chu, D.K. Panda, MPI-GDS: High performance MPI designs with GPUDirect-aSync for CPU–GPU control flow decoupling, in: International Conference on Parallel Processing, August 2017.
[35] Wilkes cluster Cambridge, UK. www.hpc.cam.ac.uk.
[36] S. Xiao, W. Feng, Inter-Block GPU Communication via Fast Barrier Synchronization.

E. Agostini received her Ph.D. in Computer Science from the University of Rome "La Sapienza" in collaboration with the National Research Council of Italy and is currently a Software Engineer at NVIDIA Corp. Her main scientific interests are parallel computing, GPGPUs, HPC, network protocols and cryptanalysis.

D. Rossetti has a degree in Theoretical Physics from Sapienza Rome University and is currently a senior engineer at NVIDIA Corp. His main research activities are in the fields of design and development of parallel computing and high-speed networking architectures optimized for numerical simulations, while his interests span different areas such as HPC, computer graphics, operating systems, I/O technologies, GPGPUs, embedded systems, digital design, real-time systems, etc.

S. Potluri received his Ph.D. in Computer Science and Engineering from The Ohio State University and is currently a Senior Software Engineer at NVIDIA Corp. His research interests include high-performance interconnects, heterogeneous architectures, parallel programming models and high-end computing applications. His current focus is on designing runtime and network solutions that enable high performance and scalable communication on clusters with NVIDIA GPUs