Optimizing communications in multi-grid methods using GPUDirect Async

Elena Agostini
University "La Sapienza"
Rome, Italy
Email: elena.ago@gmail.com

Davide Rossetti
NVIDIA
Santa Clara, CA, USA
Email: drossetti@nvidia.com

Nikolay Sakharnykh
NVIDIA
Santa Clara, CA, USA
Email: nsakharnykh@nvidia.com

***Abstract*—HPGMG is an HPC benchmarking effort based on geometric multi-grid methods. A multi-grid solver works on levels of different size so the amount of computation and communication varies a lot within the same application. NVIDIA developed a CUDA version of HPGMG-FV (Finite Volume) where, according to a threshold, finer (higher) levels are computed on GPU and coarser (lower) levels are computed on CPU: this hybrid scheme takes the advantage of each processor's strengths.**

During the HPGMG-FV execution there are three different types of communication periods: exchange boundaries (among a variable number for hosts), interpolation (lower level to higher level) and restriction (higher level to lower level), all of them using MPI. In this paper we show how by leveraging the GPUDirect Async technology it is possible to increase the speed of the communication phases by as much as 13%. GPUDirect Async is a new technology introduced by NVIDIA in CUDA 8.0 which allows mutual direct control between the GPU and the interconnect device, a Mellanox Infiniband HCA in our case.

I. Introduction

Linear solvers are probably the most common tool in scientific computing applications and can be divided in two basic classes: *direct* and *iterative*. *Multi-grid* methods are iterative methods that can deliver linear complexity by solving elliptic PDEs $Ax = b$ using a hierarchical approach, i.e. the solution to an hard problem (finer grid of elements) is expressed as solution to an easier problem (coarser grid of element). There are two different types of *Multi-grid* methods: algebric multi-grid (*AMG*), where operator A is a sparse matrix, and geometric multi-grid (*GMG*), where operator A is a stencil. While AMG method is a more general approach using an arbitrary graph, GMG method is more efficient on structured problems, since is can take advantage of the additional information from the geometric representation of the problem.

An example of GMG is HPGMG [3], an HPC benchmarking effort developed by Lawrence Berkeley National Laboratory. In particular, HPGMG-FV solves variable-coefficient elliptic problems on isotropic Cartesian grids using the finite volume method (FV) [5] and Full Multigrid (FMG) [6]. It supports high-order discretizations, and is extremely fast and accurate; in case of multi-process execution, the workload is fairly distributed among processes because, in order to improve the parallelization, each problem level is divided into several same-size boxes.

NVIDIA improved the HPGMG-FV implementation [4] using a hybrid solution (involving both CPU and GPU), achieving a great enhancement in performances. In Section II we explain the most important changes carried out by NVIDIA useful to understand the rest of the paper.

Considering the NVIDIA hybrid implementation, in case of multi-GPU (multi-process) execution, communications play an important role in the algorithm. Section IV, focus on the communication enhancement obtained by the use of GPUDirect Async, a new technology introduced by NVIDIA in CUDA 8.0 (see Section III)

II. HPGMG-FV with CUDA

At the core of every multi-grid solver there is a V-cycle (Figure 1) computation pattern: starting from the finest structured grid, where a smoothing operation is applied to reduce high-frequency errors, the residual is calculated and propagated to the coarser grid. This process is repeated until the bottom level is reached, at which point the coarsest problem is solved directly, and then the solution is propagated back to the finest grid. The V-cycle is mainly dominated by the smoothing (GSRB smoother in our case), and residual operations at the top levels. These are usually 3D stencil operations on a structured grid.

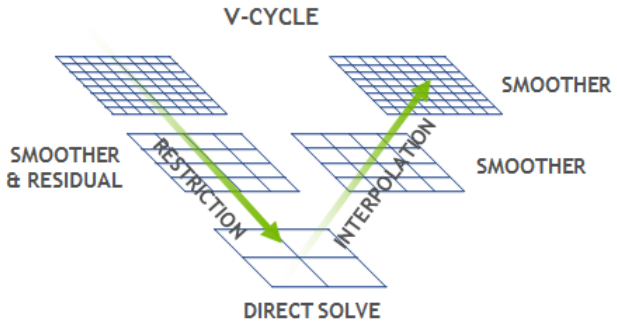


Fig. 1: V-Cycle

HPGMG-FV implements an F-cycle, which starts at the bottom of the multi-grid hierarchy and performs multiple V-cycles, gradually adding finer levels. HPGMG-FV takes as input the amount and the $log_2(size)$ of the finest level

boxes, calculating the level total size; then it obtains the size of all the other (smaller) levels. The F-cycle is considered a state-of-the-art in multi-grid methods and converges much faster than a conventional V-cycle.
NVIDIA analyzed the difference among levels in an F-Cycle: Top (fine) levels have lots of grid points and can run efficiently on throughput-oriented parallel architectures like GPUs, while bottom (coarse) levels will be latency limited on a GPU because there is not enough work to make efficient use of all the parallel cores. During an F-Cycle, coarse levels are visited progressively more often than the fine levels therefore their cost is significant in an F-Cycle. For those reasons, coarse grids are better suited for latency-optimized processors like CPUs. Thus, for optimal performance, an hybrid scheme is required to guarantee that each level is executed on the suitable architecture: if the size of a level is over a certain threshold (empirically set to 10000 elements), then it runs on GPU, otherwise on CPU (Figure 2).

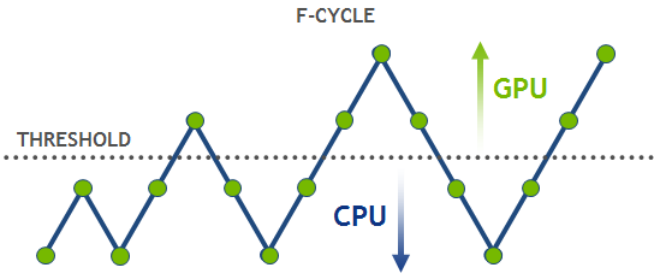


Fig. 2: F-Cycle with CPU-GPU threshold

To enable GPU accelleration, the simplest way was to add corresponding GPU kernels for low-level stencil operators and update memory allocations using *cudaMallocManaged()* instead of *malloc()*, in order to use Unified Memory [7] for memory used by GPU only, and use host pinned memory if it must be used by both CPU and GPU (i.e. communication buffers).

In Figure 3 there is a simplified operations timeline in case of coarse level (CPU) to fine level (GPU) and again coarse level (CPU).

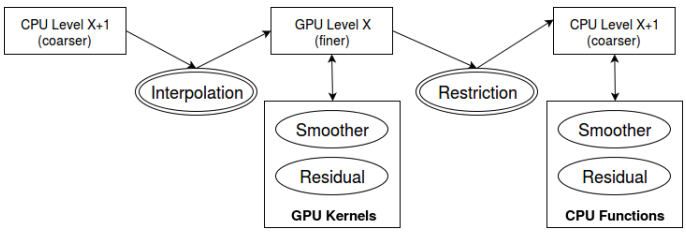


Fig. 3: F-Cycle: moving from a coarse level to a finer level and then go back to the coarse level

In Figure 4 we report the enhancement obtained by NVIDIA using the hybrid solution in a benchmark on the ORNL Titan supercomputer [8]. For further details, please refer to [4]

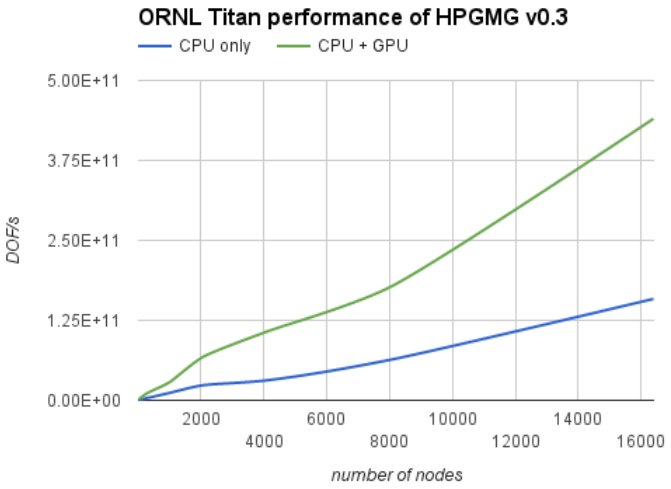


Fig. 4: Performance of GPU-accelerated HPGMG on the ORNL Titan supercomputer. Results obtained by Sam Williams from Lawrence Berkeley National Lab.

A. Communication periods

As described in Section II, in case of *Multi-grid* methods the smoother is a stencil. According to a stencil-like code, in case of multi-GPU execution the smoother must exchange the boundary ghost regions with the other processes (*intra-level* communication). On the contrary, (Figure 3) restriction and interpolation play a role in case of moving from a level to another (*inter-level* communication). *inter-level* exchanges play a bigger role at scale where we do consolidation of N to N processes (so we have lots of data moved around)

1) Exchange Boundaries: This function implements the boundary region exchange doing a {pack}, {send}, {interior_compute}, {receive}, {unpack} sequence among processes working on the same level. See Algorithm 1 for the pseudo-code.

Algorithm 1 Exchange Boundaries function

```
1: for i = 1 to PROCESSES do
2:     cudaMallocHost(sendBuffers[i])
3:     cudaMallocHost(receiveBuffers[i])
4: end for
5: ...
6: function EXCHANGEBOUNDARIES()
7:     for i = 1 to PROCESSES do
8:         MPI_Irecv(receiveBuffers[i], &reqs_recv[i])
9:     end for
10:    cuda_pack(sendBuffers)
11:    cudaDeviceSynchronize()
12:    for i = 1 to PROCESSES do
13:        MPI_Isend(sendBuffers[i])
14:    end for
15:    cuda_interior_compute(localBuffers)
16:    MPI_Waitall(reqs_recv)
17:    cuda_unpack(receiveBuffers)
18: end function
```

A *cudaDeviceSynchronize()* is required between the CUDA

kernel pack operation and the *MPI_Isend()* to guarantee correctly updated *sendBuffers*. The *exchangeBoundaries()* is the most used communication function during an HPGMG-FV execution.

2) Restriction: This function occurs when moving from a finer level to a coarser level; in case of GPU-to-CPU level, it is quite similar to *exchangeBoundaries()*: GPU kernels works on *sendBuffers*, then a *cudaDeviceSynchronize()* is needed before the *MPI_Isend()*. Moreover, if the coarsest level is on CPU, an additional *cudaDeviceSynchronize()* is required because GPU kernels are launched asynchronously and we need to guarantee completion before we start CPU tasks.

3) Interpolation: It occurs when moving from a coarser level to a finer level. It requires a *cudaDeviceSynchronize()* before the *MPI_Isend()* but it doesn't needs to synchronize if the coarsest level is on CPU because CPU tasks are synchronous and they will be completed before we launch GPU kernels.

Although communications are not the most expensive part of the algorithm, profiling the GPU levels execution we noticed that:

- the CPU launched a lot of CUDA kernels for residual and smoothing and each kernel launch required a lot of time, leaving sometime the GPU idle, waiting for an other kernel
- the high number of *cudaDeviceSynchronize()* slowed the performances

To hide the kernel launches and remove as many *cudaDeviceSynchronize()* as possible, we used GPUDirect Async (see Section III) to improve performances of all the communications periods.

In Figure 5 there is a simplified timeline to clarify how *exchangeBoundaries()*, *restriction()* and *interpolation()* are used during a GPU level processing.

Fig. 5: Synchronous *exchangeBoundaries()* timeline

III. GPUDirect Async

GPUDirect [2] is a family of technologies aimed at optimizing data movement respectively among GPUs (P2P) and with third-party devices (RDMA). In particular, GPUDirect RDMA reduces latency and improves bus utilization by enabling a direct data path over the PCI Express (PCIe) bus between the GPU and a third-party device, in our case a network interface controller (NIC).

GPUDirect Async is a new GPUDirect technology introduced by NVIDIA in CUDA 8.0; it allows mutual direct control between the GPU and the third party device, a recent generation Infiniband HCA in our case.

Although an in-depth explaination of the GPUDirect Async implementation is beyond the scope of this paper, in the following we will breafly describe how it works and how it can be leveraged in HPGMG-FV. With Async the GPU is able to trigger communications on HCA, while at the same time HCA is able to unblock CUDA tasks; the CPU is only needed to prepare and queue both the compute and communication tasks on GPU. More specifically:

- The CPU allocates communication buffers (device or host pinned memory)
- It registers some HCA specific data structures, like command queues (IB Verbs QPs) and completion queues (IB Verbs CQs) (*cuMemHostRegister()*)
- Prepares the send/receive requests descriptors
- Converts those descriptors into a sequence of basic operations synchronized to a GPU stream. Examples of those operations are:
  - Triggering a prepared communication sequence, i.e. by ringing the HCA doorbell.
  - Triggering a pre-launched CUDA kernel.
  - Waiting (polling) for communication task completion (IB Verbs CQ entries, or CQEs).

After having prepared and queued all the necessary tasks, for example the CPU can go back and do other useful work. That way a whole multi-host parallel computation can be offloaded onto a CUDA stream. Still when necessary, the CPU query the CUDA stream for the status of the outstanding operations. On the other hand, the same mechanism is expected to be useful to efficiently scale in combination with low performance CPUs, removing their preparation work from the critical path.

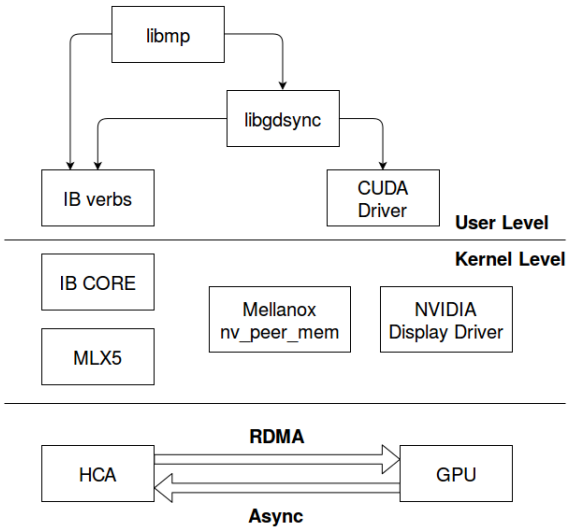


Fig. 6: GPUDirect Async Software stack

The GPUDirect software stack (Figure 6) is composed by:

1) libmlx5: (Vendor/device specific) It's the bottom level. *libmlx5* is a low-level device driver for Mellanox Connect-IB InfiniBand host channel adapters (HCAs). This allows userspace processes to access Mellanox HCA hardware directly with low latency and low overhead. The standard

implementation has been extended for needs of GPUDirect Async.

2) libibverbs: (Verbs APIs) *libibverbs* library implements the OpenFabrics Infiniband Verbs API. The standard implementation has been extended with new Verbs specific to GPUDirect Async.

3) LibGDSync: (NVIDIA open-source) Developed by NVIDIA, it consist of a set of hybrid APIs where both IB Verbs and CUDA GPU stream are merged. It is responsible to create IB tracking data 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.

4) LibMP: (NVIDIA open-source) It is at the top level and is a messaging library (similar to MPI) developed as a tecnology demostrator to easily deploy the GPUDirect Async technology on MPI applications. It leverages LibGDSync APIs and offer basic communications like *mp_isend_on_stream()*, *mp_wait_on_stream()*, *mp_iput_on_stream()*, etc..

In addition to the CUDA stream synchronous communication mode previously described, in this paper we are going to explore a *kernel-initiated* model, where the Simultaneous Multiprocessors (SMs) which are in charge of executing the CUDA kernels can directly issue communication primitives, i.e. sending messages and waiting for completions.

In the following sections, we will show that kernel-initiated mode can be faster than Stream Async mode, i.e. by allowing kernel fusion techniques thereby exposing more concurrency to the highly parallel GPU HW units. The downside of it is that it is more complicated to use, mainly because the programmer needs to manually schedule different sub-tasks (send, receive or compute) to separate CUDA kernel thread blocks respecting the constraints of the algorithm.

IV. Asynchronous communications

We tried both the asynchronous modes in HPGMG-FV, to evaluate performances considering execution times of GPU levels only.

A. Stream Async mode

See Algorithm 2 for the *exchangeBoundariesAsync()* pseudo-code.

The *cudaDeviceSynchronize()* between *cuda_pack()* and *mp_isend_on_stream()* function is no more required. The GPU stream will:

1) Write on *sendBuffers* using the *cuda_pack()* kernel
2) Post the send requests
3) Execute the *cuda_interior_compute()* kernel
4) Wait the receive completion
5) Read the received data (*receiveBuffers*) with the *cuda_unpack()* kernel

Similar considerations can be done for *restriction()* and *interpolation()* functions.

Stream Async is used only if the level is a GPU level: this means that only a *cudaDeviceSynchronize()* is needed during the *restriction()* function from the last GPU (higher) level to the first CPU (lower) level. During GPU levels, CPU can launch all the CUDA kernels without waiting, hiding the kernel launches times (Figure 7).

Algorithm 2 Exchange Boundaries Stream Async function

```
1: for i = 1 to PROCESSES do
2:     cudaMallocHost(sendBuffers[i])
3:     cudaMallocHost(receiveBuffers[i])
4: end for
5: ...
6: function EXCHANGEBOUNDARIESSTREAMASYNC(STREAM)
7:     for i = 1 to PROCESSES do
8:         mp_irecv(receiveBuffers[i], &receiveDescriptors[i])
9:     end for
10:    cuda_pack(sendBuffers, stream)
11:    for i = 1 to PROCESSES do
12:        mp_isend_on_stream(sendBuffers[i],
   &sendDescriptors[i], stream)
13:    end for
14:    cuda_interior_compute(localBuffers)
15:    mp_wait_all_on_stream(receiveDescriptors)
16:    cuda_unpack(receiveBuffers)
17:    mp_wait_all_on_stream(sendDescriptors)
18: end function
```

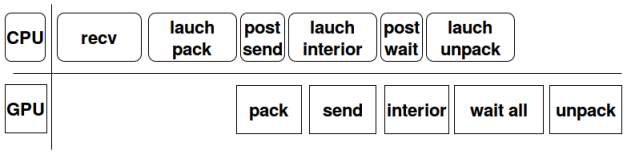


Fig. 7: *exchangeBoundariesStreamAsync()* timeline

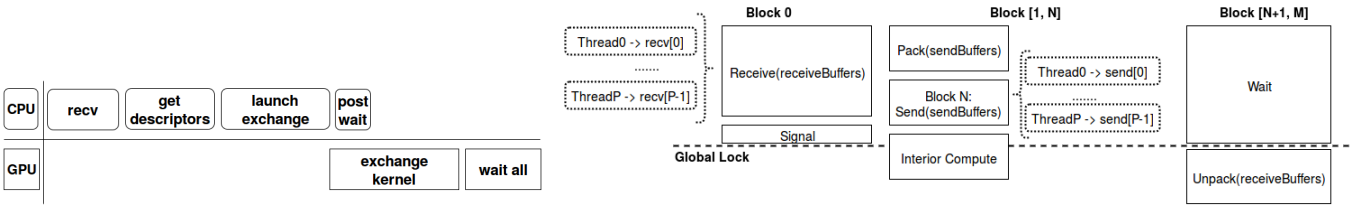
B. kernel-initiated mode

The algorithm from the CPU point of view is extremely simple (Figure 8a): it must prepare send/receive descriptors and launch a single CUDA kernel in which GPU will overlap, as much as possible, all the *exchangeBoundaries()* operations (see Algorithm 3).

Algorithm 3 Exchange Boundaries kernel-initiated function

```
1: for i = 1 to PROCESSES do
2:     cudaMallocHost(sendBuffers[i])
3:     cudaMallocHost(receiveBuffers[i])
4: end for
5: ...
6: function EXCHANGEBOUNDARIESKERNELINITIATED()
7:     for i = 1 to PROCESSES do
8:         mp_irecv(receiveBuffers[i], &receiveDescriptors[i])
9:     end for
10:    for i = 1 to PROCESSES do
11:        mp_isend_prepare(&sendDescriptors[i])
12:    end for
13:    cuda_compute_exchange_kernel(receiveDescriptors, sendDe-
   scriptors)
14:    mp_wait_all_on_stream(sendDescriptors)
15: end function
```

The most difficult part lies in the *cuda_compute_exchange_kernel()*. According to previous



(a) *exchangeBoundariesKernelInitiated()* timeline

(b) cuda_compute_exchange_kernel()

Fig. 8: kernel-initiated mode

observations about stencils, we can distringuish three different groups of independent operations: [pack, send], [interior compute], [receive, unpack]; then we can execute them in parallel using different CUDA kernel blocks. Basically, the *cuda_compute_exchange_kernel()* needs N+M+1 blocks in a mono-dimensional grid, where N is the number of blocks required by the *cuda_pack()* and *cuda_interior_compute()* and M is the number of blocks required by *cuda_unpack()* plus 1 block, used to receive data as explained in Figure 8b.

The receive is the most expensive one, then the first incoming kernel block waits to receive all data (each block's thread receives from a single peer process); all the blocks from the second to the N-th work on the [pack, send] group of operations plus the [interior compute]. Finally the remaining M blocks wait for the the completion of the first block receive and then they will unpack received data, finishing the *exchangeBoundaries()* execution.

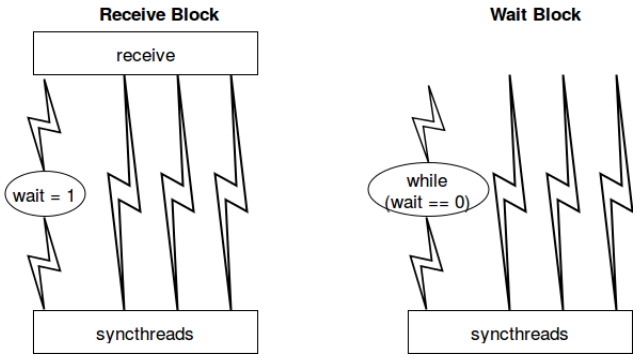


Fig. 9: Global Lock among kernel blocks

To force the last blocks to wait for the receive completion, we used a global lock as explained in Figure 9. All threads in wait blocks move to the *__syncthreads()* except for the first one: the Thread0 of all the wait blocks is waiting for the Thread0 of the receive block to set a global memory variable to 1. When this happens (after the receive completion), all the Thread0 in the wait blocks will reach the *__syncthreads()* and then they start to unpack received data.

When using kernel-initiated mode, it is very important that the receive (or wait for receive) operation is not preventing the send, otherwise you will have a deadlock. For example, having a GPU with 15 SM, if the first one is waiting on the receive and the others 14 are waiting for the receive completion, the [pack, send] operations will never occur.

V. Benchmarks and Results

As described in Section II, HPGMG-FV takes as input the size and number of the boxes in the highest level; during our benchmarks we used 8 boxes varying the size in [4,5,6,7]. According to Section II, the threshold size used during NVIDIA tests was 10000; considering 4 as minimun log_2 size for boxes, this means that the 3 smallest levels are always executed on CPU and all the others on GPU.
We tried different threshold values (i.e. changing the number of GPU levels for each execution) during our asynchronous benchmarks but we found that the best values is always 10000.

A. First Benchmark: NVIDIA servers

For the first benchmark we used two NVIDIA servers, having both a GPU Tesla K40m with the clock boost to 875 MHz, OS RHEL 6.6 using an NVIDIA internal display driver. In Figure 10 there is the time gain of Stream Async mode and kernel-initiated mode on the MPI mode considering GPU levels only having 2 processes: the more the boxes size increases, the more performance gain decreases, because the messages size grows with the box size increasing. Moreover, kernel-initiated mode is always a better solution than Stream Async mode.

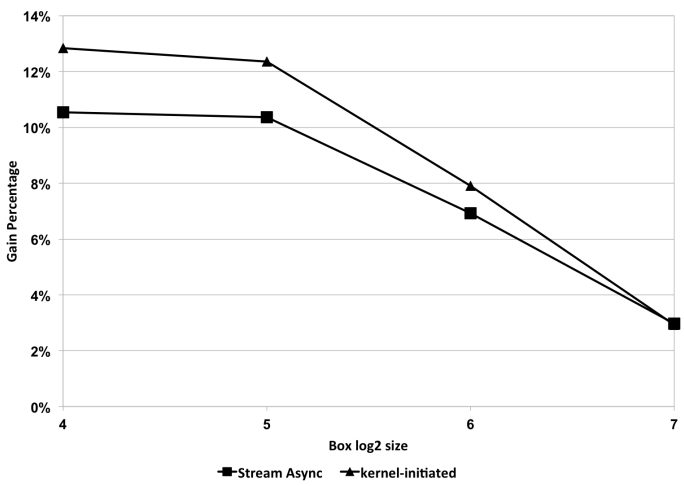


Fig. 10: 2 processes, Asynchronous time gain, NVIDIA servers

B. Second Benchmark: Wilkes HPC

For this second benchmark, we used the Wilkes HPC (University of Cambridge, UK)[9]. The system consists of 128 Dell T620 servers, 256 NVIDIA Tesla K20c GPUs interconnected by 256 Mellanox Connect IB cards. We couldn't use all the 256 GPUs because we needed to install our GPUDirect Async libraries which requires some particular settings; for this reason during benchmark we used up to 16 different nodes (each one having a single GPU) using the 361.62 driver available along with the CUDA 8.0 RC package. To make a comparison with the first benchmark, we started using only 2 Wilkes nodes (see Figure 11).

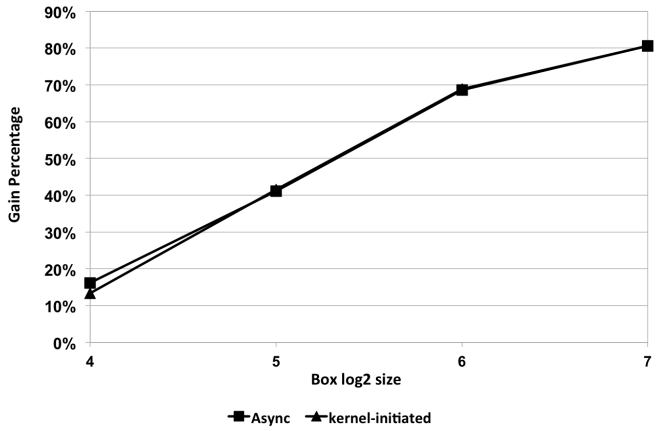


Fig. 11: 2 processes, Asynchronous time gain, Wilkes HPC

Although gain seems incredible (80% in case of box size 7), the reason relies on MPI execution; we noticed that the Unified Memory with display driver 361.62 led to a lot of CPU page faults; moreover *cudaDeviceSynchronize()* and CUDA kernel launches employed much more time than the same MPI execution in the first benchmark. We got the same result installing the 361.62 display driver on the NVIDIA servers.

On the contrary, execution times of the asynchronous versions were similar to the first benchmark; therefore we can say that asynchronous versions do not rely on Unified Memory performance while the original synchronous MPI code suffered from extensive CPU page faults.

Recently NVIDIA updated the display driver in CUDA 8.0 RC to the 361.77 version; we are waiting for the driver update on Wilkes HPC to perform the same benchmark again.

VI. Conclusion

In this paper we presented the first application of the NVIDIA GPUDirect Async technology on the HPGMG-FV multi-GPU implementation. In particular, we developed an asynchronous version of a stencil operator, that is highly used in the context of scientific and engineering applications. Althought communications aren't the most relevant part in the HPGMG-FV algorithm, we reached a time gain of about 13%. Unfortunately for the moment, we had some problem during large-scale benchmarks related to the display driver released by NVIDIA along with the new CUDA 8.0 RC Toolkit. The next step is to perform all benchmarks again up to 64 nodes on the Wilkes HPC using the most updated (and recently released) display driver 361.77.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.
[2] GPUDirect family: https://developer.nvidia.com/gpudirect
[3] HPGMG https://hpgmg.org
[4] N. Sakharnykh *High-Performance Geometric Multi-Grid with GPU Acceleration*. https://devblogs.nvidia.com/parallelforall/high-performance-geometric-multi-grid-gpu-acceleration
[5] *Finite Volume method*. https://en.wikipedia.org/wiki/Finite_volume_method
[6] *Full MultiGrid method*. https://en.wikipedia.org/wiki/Multigrid_method
[7] *Unified Memory*. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6
[8] *ORNL Titan supercomputer*. https://www.olcf.ornl.gov/titan
[9] *Wilkes HPC Cambridge, UK*. www.hpc.cam.ac.uk