OPTIMIZING COMMUNICATIONS IN MULTI-GRID METHODS USING GPUDIRECT ASYNC *

ELENA AGOSTINI †

In collaboration with: D. Rossetti Collaborator NVIDIA, N. Sakharnykh Collaborator NVIDIA, M. Bernaschi PhD Supervisor

Abstract. HPGMG is a benchmark for HPC geometric multi-grid methods. A multi-grid solver works on a grid having a fixed total size but different resolutions during the execution so that the amount of computation and communication varies significantly 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 the GPU whereas coarser (lower) levels on the CPU: that hybrid scheme exploits at their best the strengths of both platforms. During the HPGMG-FV execution, there are three main communication phases: boundaries exchange (among a variable number of cluster nodes), interpolation (lower to higher level) and restriction (higher to lower level). By default communications use the Message Passing Interface (MPI).

GPUDirect Async is a new technology introduced by NVIDIA in CUDA 8.0 to support mutual direct control between the GPU and the interconnect device, a Mellanox Infiniband HCA in our case.

In this paper, we propose two optimized communication schemes involving the GPUDirect Async technology, where the burden of the communication is off-loaded onto the GPU itself. When running a properly modified version of HPGMG-FV, where Async technology is used for only those aforementioned communication phases, the communication overhead reduces up to 26% .

Key words. HPGMG-FV, Multi-Grid, GPU, Asynchronous Communication, GPUDirect Async

1. Introduction. *Multi-grid* methods are iterative methods that can deliver linear complexity by solving elliptic PDEs using a hierarchical approach, i.e. the solution to a 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[1]. While AMG method is a more general approach using an arbitrary graph, GMG method is more efficient on structured problems, since it can take advantage of the additional information from the geometric representation of the problem.

HPGMG is an example of GMG developed by Lawrence Berkeley National Laboratory [4] that is often employed also as a HPC benchmark. In particular, HPGMG-FV solves variable-coefficient elliptic problems on isotropic Cartesian grids using the finite volume method (FV) [6]. It supports high-order discretizations, and it is extremely fast and accurate; 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, calculating the level total size; then it obtains the size of all the other (coarse grain) levels. Every distributed level is computed with smoothing and residual operations;

*Project in collaboration with the CUDA team at NVIDIA.

†University of Rome "La Sapienza", (agostini@di.uniroma1.it, `http://www.uniroma1.it/`).

[1]Stencil codes are a class of iterative kernels which update array (generally a 2- or 3-dimensional regular grid) elements (cells) according to some fixed pattern, called stencil. The state of a cell in the next time step can be deduced from its own previous state and the states of the cells in its neighborhood.

in particular the smoother is a stencil code and the tasks of each process are:

- Pack: compute the boundaries of the box and copy them inside send buffers;
- Send: distribute the boundaries to the other processes (*intra-level* communication);
- Interior Compute: apply the stencil operator to the inner elements of the box;
- Receive: receive boundaries from the other processes (*intra-level* communication)
- Unpack: copy (and compute) received data along box boundaries

HPGMG-FV implements an F-cycle (Figure 2) which starts at the bottom of the multi-grid hierarchy and performs multiple V-cycles (Figure 1), gradually adding finer levels. During a V-cycle, the grid is smoothed and a residual is computed and propagated to the coarser grid. At the coarsest level, a direct solver is applied, and the solution is then iteratively interpolated to finer grids.

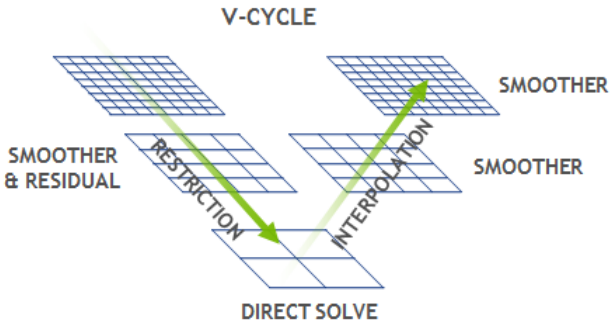

FIG. 1. *V-Cycle*

The F-cycle is considered state-of-the-art in multi-grid methods and converges much faster than a conventional V-cycle.

2. HPGMG-FV and GPUs. NVIDIA improved the HPGMG-FV implementation [5] using a hybrid solution that guarantees that each level in the F-Cycle is executed on the most suitable architecture: if the size of a level is over a certain threshold (empirically set to 10000 elements), it is processed by the GPU, otherwise by the CPU (Figure 2).

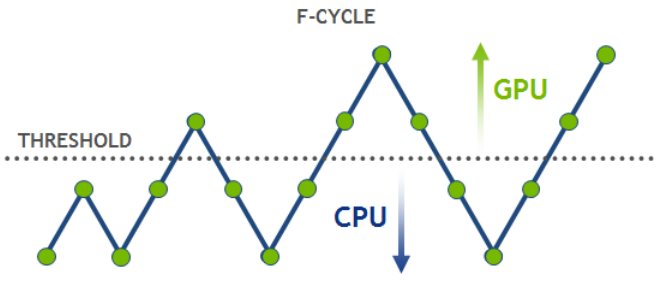

FIG. 2. *F-Cycle with CPU-GPU threshold*

To leverage GPU acceleration for higher levels, the simplest way was to use device memory for compute-only buffers and host pinned memory for communication buffers.

Considering a distributed HPGMG-FV execution, we can distinguish between two different communication phases (referring to Figure 1):

1. **Intra-level communication**: exchange boundaries (boundary region exchange of the same level between proccesses)
2. **Inter-level communication**: restriction (moving from a finer level to a coarser level) and interpolation (moving from a coarser level to a finer level)

The most frequent communication is the *exchangeBoundaries()* executed by the smoother's stencil code on a distributed level, that works according to Algorithm 1.

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
```

cudaDeviceSynchronize() is required between the CUDA kernel pack operation and the *MPI_Isend()* to guarantee the correctness of updated *sendBuffers* (Figure 3). The *exchangeBoundaries()* is the most used communication function during a HPGMG-FV execution.

FIG. 3. *Synchronous exchangeBoundaries() timeline*

Similarly, the other two *inter-level* communication functions, i.e. restriction and interpolation, require a *cudaDeviceSynchronize()* before the *MPI_Isend()*. Restriction needs an additional *cudaDeviceSynchronize()* when moving from a GPU level to a CPU level.

Figure 4 shows a simplified operations timeline in case of a hybrid solution when moving from a coarse level (CPU) to a fine level (GPU) and coming back to the coarse level (CPU).

Starting from that NVIDIA hybrid implementation, we have enhanced the performance of the communication phases, which play an important role during distributed executions, using a new feature named GPUDirect Async, introduced by NVIDIA

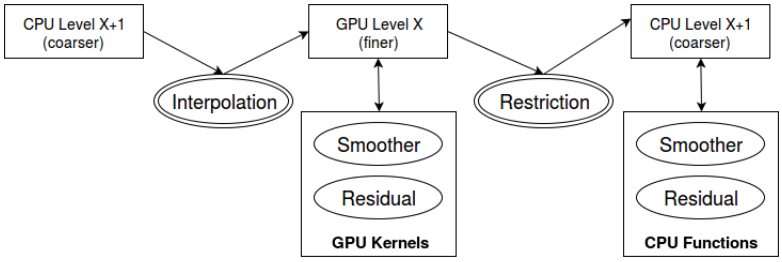

FIG. 4. *F-Cycle: moving from a coarse level to a finer level and then going back to the coarse level. Restriction and interpolation play a role in case of moving from a level to another (*inter-level *communication)*

along with CUDA 8.0. The aim of the experiment is to demonstrate how this new technology can reduce the communication overhead in real-world distributed applications having high complexity like HPGMG-FV.

3. GPUDirect Async. GPUDirect [1] is a family of technologies aimed at optimizing data movement respectively among GPUs (P2P) and with third-party devices (RDMA). GPUDirect Async is a new GPUDirect technology introduced by NVIDIA in CUDA 8.0; it allows mutual direct control between the GPU and third party devices like Infiniband in case of communications. The CPU can enqueue on a CUDA stream both computation (i.e. *classic* CUDA kernels) and communication tasks and go back to do useful work waiting for GPU tasks completion only when needed. Although an in-depth explaination of the GPUDirect Async implementation is beyond the scope of the present work, in the following we will briefly describe how it works and how it can be useful to improve HPGMG-FV performance.

Considering a typical multi-GPU application in case of distributed execution, a common pattern is:

- Work on some data using one or more CUDA kernels
- Store data results in one or more send buffers
- Send data
- Wait for data from the other processes
- Launch one or more CUDA kernels to work on received data

In Figure 5 there is a timeline of this sequence in case of Infiniband as communication technology. By leveraging GPUDirect Async, a whole parallel computation phase can be offloaded onto a CUDA stream, removing the CPU from the critical path, as described in Figure 6.

GPUDirect Async has two different execution models:

- **Stream Async model (SA)**: The one described in this Section, in which communications are enqueued into a CUDA stream;
- **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 completion.

On the top of the GPUDirect Async stack software there is LibMP, an NVIDIA open-source messaging library (similar to MPI) developed as a technology demostrator to easily deploy the GPUDirect Async technology on MPI applications. LibMP API enables the use of asynchronous communication in both SA and KI models and

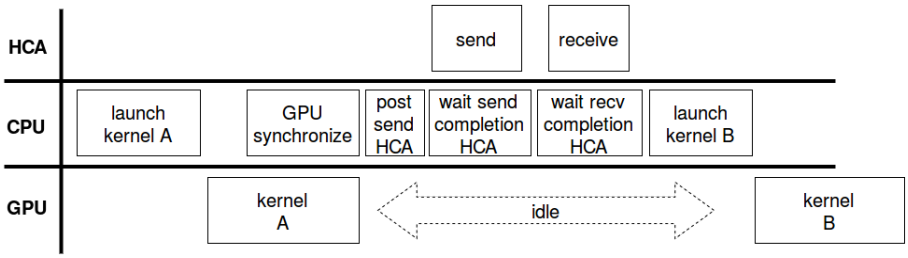

FIG. 5. *Common communication pattern in Multi-GPU application timeline*

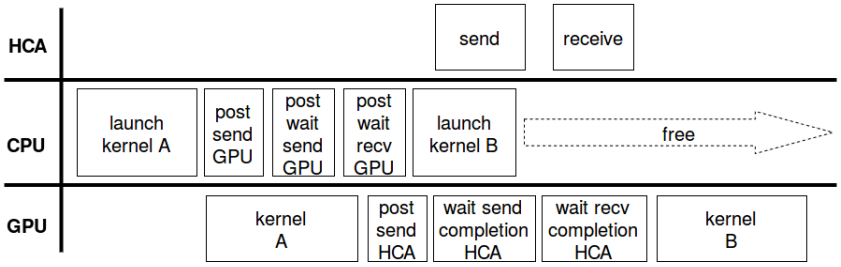

FIG. 6. *Communication pattern in Multi-GPU application timeline leveraged with GPUDirect Async*

presents also a third model that is totally synchronous for performance comparison purposes. The library needs an already-initialized MPI environment (i.e. communicator, ranks, topology, etc..); then it is possible to use LibMP asynchronous communication functions (like *mp_isend_on_stream()*, *mp_wait_on_stream()*, etc..) instead of standard MPI functions (*MPI_Isend()*, *MPI_Wait()*, etc..). HPGMG-FV communication pattern is quite similar to the one in Figure 5, therefore we improved communications with GPUDirect Async.

For the following benchmarks, we used two different environments. The first one (E1) is composed by two standard 2U Xeon based servers, each one with a Mellanox dual-port FDR Connect-Infiniband HCA and a single Tesla K40m (boost clocks set to 875 MHz), running RHEL 6.6 and a pre-release version of the GPU display driver with CUDA 8.0 RC and OpenMPI version 1.10.3.

The second one (E2) is the Wilkes HPC cluster at the Cambridge University, UK [9]. The system consists of 128 Dell T620 servers, 256 NVIDIA Tesla K20c GPUs interconnected by 256 Mellanox Connect Infiniband cards, having the CUDA 8.0 Toolkit, display driver 367.44 and OpenMPI version 1.10.3. We had a reservation of 16 nodes to benchmark our HPGMG-FV Async solution.

4. HPGMG-FV Async. We used both SA and KI models trying to reduce, during communications, the number of *cudaDeviceSynchronize()*, which slow down performance, and to hide kernel launches latency. We implemented also a synchronous version with the LibMP functions to make a comparison with the similar synchronous MPI functions. For our benchmarks we considered the *Gauss-Seidel Red-Black* (GSRB) smoother (all the smoothers used by HPGMG-FV have the same number of communication periods) using a fixed number of 8 boxes for each process, varying the $\log_2(size)$ parameter between 4 and 7.

In Figure 7 there is a first benchmark with 2 processes on the E1 environment, considering the times of GPU levels only. Both LibMP synchronous and asynchronous models are faster than the original MPI implementation and, in particular, Kernel-Initiated model offers a time gain up to 13% .

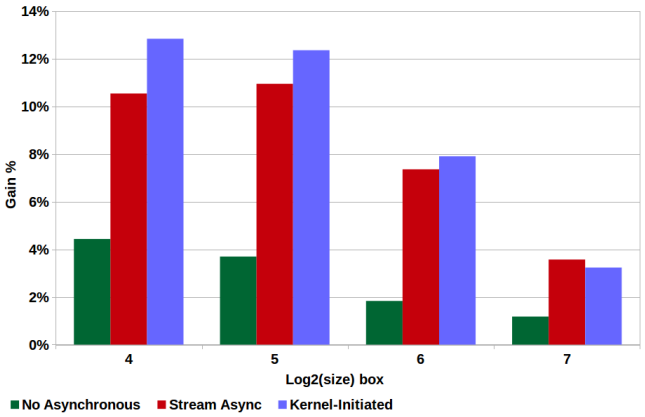

FIG. 7. *HPGMG-FV time gain LibMP models on MPI, GPU levels only, E1 environment, 2 processes*

4.1. Stream Asynchronous model implementation. Algorithm 2 shows the pseudo-code of the *exchangeBoundariesAsync()*.

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, stream)
15:    mp_wait_all_on_stream(receiveDescriptors, stream)
16:    cuda_unpack(receiveBuffers, stream)
17:    mp_wait_all_on_stream(sendDescriptors, stream)
18: end function
```

The *cudaDeviceSynchronize()* between the pack and the send operations is no more required. The CUDA stream is responsible for:

1. Packing data in the *sendBuffers* by means of the *cuda_pack()* kernel
2. Triggering the send operations (send WQEs have been previously posted in the QP by the CPU)
3. Executing the *cuda_interior_compute()* kernel

4. Waiting for the receive completion
5. Reading the received data (*receiveBuffers*) with the *cuda_unpack()* kernel

Considering a similar implementation for restriction and interpolation, since the asynchronous communications are used only in case of GPU levels, a single *cudaDeviceSynchronize()* is required during restriction when moving from a GPU level to a CPU level. During GPU levels, the CPU can launch all the CUDA kernels without waiting, hiding the kernel launch times (timeline in Figure 8).

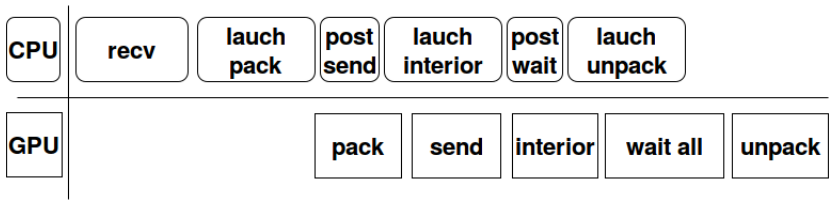

FIG. 8. *exchangeBoundariesStreamAsync() timeline*

In Figure 9 the y axis shows the performance improvement of the GPU levels using the Stream Async implementation with respect to the standard MPI mode in the E2 environment (Wilkes HPC) using up to 16 processes. The maximum gain (about 24%) is reached in case of $\log_2(size) = 4$ with 2 processes. The bigger is the box size, the more the performance gain decreases, because the message size grows with the box size, therefore comunication overhead becomes less important.

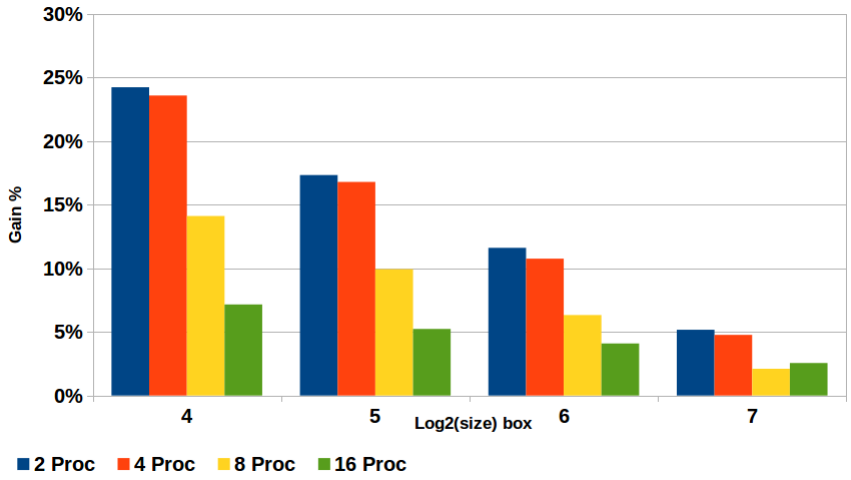

FIG. 9. *HPGMG-FV time gain Stream Async on MPI, GPU levels only, E2 environment, up to 16 processes*

4.2. Kernel-Initiated model implementation. The algorithm from the CPU point of view is extremely simple (Algorithm 3 and timeline in Figure 10): 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.

The complexity is moved to *cuda_compute_exchange_kernel()* which in a sense employs both tasks (different blocks deal with different tasks) and data (threads in the same block cooperatively work on the same task) parallelism in the same kernel.

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(stream)
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_send_prepare(&sendDescriptors[i])
12:    end for
13:    cuda_compute_exchange_kernel(receiveDescriptors, sendDescriptors, stream)
14:    mp_wait_all_on_stream(sendDescriptors, stream)
15: end function
```

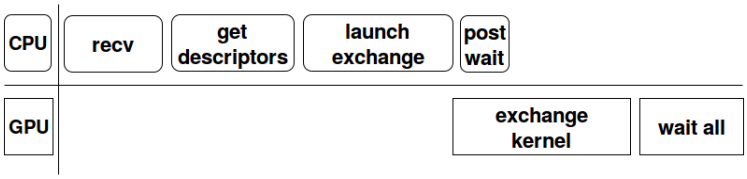

FIG. 10. *exchangeBoundariesKernelInitiated() timeline*

According to previous observations about the HPGMG-FV smoother, we can distringuish three different groups of independent operations: [pack, send], [interior compute], [receive, unpack]. The idea is to assign these tasks to the different blocks of a single CUDA kernel. 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 shown in Figure 11.

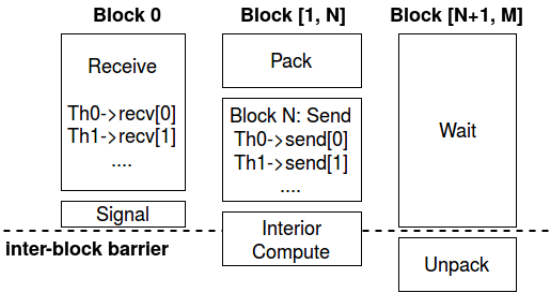

FIG. 11. *cuda_compute_exchange_kernel()*

We use atomic memory operations to pick each thread block [2] and to assign it to the right task preventing dead-locks.

Receiving is a time critical task, so the first *receiver* thread block is used to wait

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

for incoming messages. In particular each thread *polls* on the receive completion queue associated to each remote node. All the *sender* blocks from the second to the $N+1$-th are assigned to the [pack, send] group of operations plus the [interior compute]. Finally, the remaining M *unpacker* blocks wait for the *receiver* block to signal that all incoming data have been received, before unpacking them.

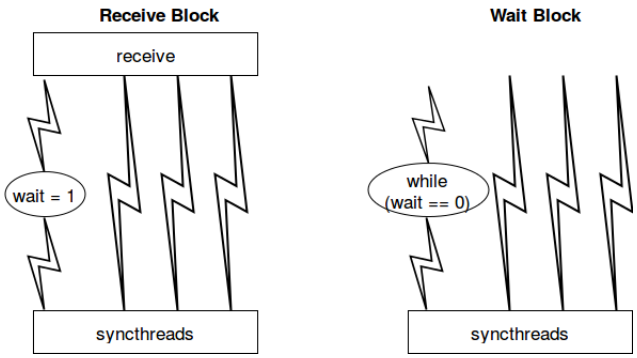

FIG. 12. *inter-block barrier*

An inter-block barrier scheme is used to synchronize the *receiver* and the *unpacker* blocks, as explained in Figure 12. Thread 0 of each *unpacker* block waits for thread 0 of the *receiver* block to set a global memory variable equal to 1, whereas the remaining threads move to the *__syncthreads()* barrier. When that happens (after the receive completion), all threads 0 in the *unpacker* blocks will reach the matching *__syncthreads()* barrier and then start to unpack the received data.

When using Kernel-Initiated model, to avoid dead-locks the *receiver* task must not prevent the *sender* task from starting or progressing: there must be always at least one block waiting for receiving and an other block executing the send.

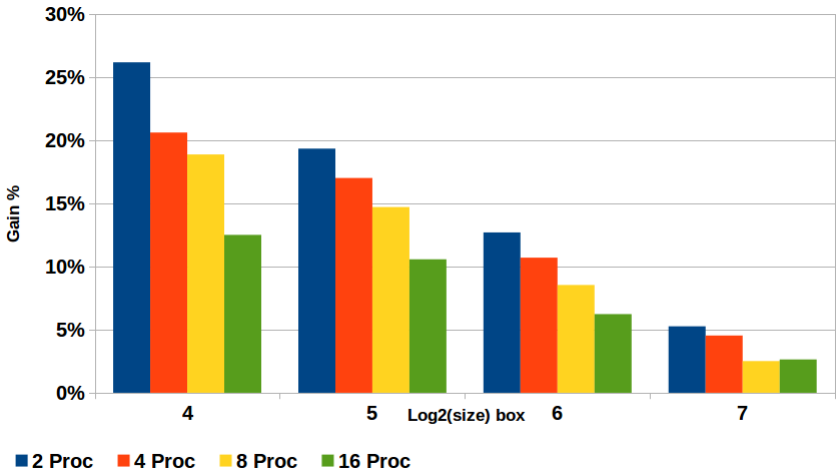

FIG. 13. *HPGMG-FV time gain Kernel-Initiated on MPI, GPU levels only, E2 environment, up to 16 processes*

We performed the same tests as in the previous Section (on the E2 environment using up to 16 processes, Figure 13) to evaluate Kernel-Initiated implementation

performance. The maximum gain is 26% in case of 2 processes with $\log_2(size) = 4$ box size and, generally speaking, the performance with Kernel-Initiated model is always better than Stream Async model performance.

5. Conclusions. We presented the performance of a GPU accelerated multi-node implementation of the HPGMG-FV benchmark using the recently introduced NVIDIA GPUDirect Async technology, demonstrating how the use of asynchronous communications, according to different execution models, can be useful for this particular class of applications. GPUDirect Async has been released with CUDA 8.0 whereas Infiniband network support for Async comes in the form of a new set of experimental OFED verbs (Mellanox OFED 3.4 plus updates in [11]) and a mid layer abstraction library [12] as BSD. In the near future, we are going to test asychronous HPGMG-FV, and other similar proxy applications, using a higher number of nodes to better evaluate the scalability of the GPUDirect technology..

REFERENCES

[1] *GPUDirect family* https://developer.nvidia.com/gpudirect
[2] *Mellanox GDR* http://www.mellanox.com/page/products_dyn?product_family=116
[3] *IB Verbs RDMA programming guide.* http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf
[4] HPGMG https://hpgmg.org
[5] N. Sakharnykh *High-Performance Geometric Multi-Grid with GPU Acceleration.* https://devblogs.nvidia.com/parallelforall/high-performance-geometric-multi-grid-gpu-acceleration
[6] *Finite Volume method.* https://en.wikipedia.org/wiki/Finite_volume_method
[7] *Full MultiGrid method.* https://en.wikipedia.org/wiki/Multigrid_method
[8] *ORNL Titan supercomputer.* https://www.olcf.ornl.gov/titan
[9] *Wilkes HPC Cambridge, UK.* www.hpc.cam.ac.uk
[10] *ExMaTex.* http://www.exmatex.org/comd.html
[11] *GPUDirect libmlx5.* https://github.com/gpudirect/libmlx5
[12] *GPUDirect libgdsync.* https://github.com/gpudirect/libgdsync