Computing and Informatics, Vol. 39, 2020, 510–536, doi: 10.31577/cai_2020_3_510

INVESTIGATION OF PARALLEL DATA PROCESSING USING HYBRID HIGH PERFORMANCE CPU + GPU SYSTEMS AND CUDA STREAMS

Paweł Czarnul

Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology, Narutowicza 11/12, 80-233
Gdansk, Poland
e-mail: pczarnul@eti.pg.edu.pl

Abstract. The paper investigates parallel data processing in a hybrid CPU + GPU(s) system using multiple CUDA streams for overlapping communication and computations. This is crucial for efficient processing of data, in particular incoming data stream processing that would naturally be forwarded using multiple CUDA streams to GPUs. Performance is evaluated for various compute time to host-device communication time ratios, numbers of CUDA streams, for various numbers of threads managing computations on GPUs. Tests also reveal benefits of using CUDA MPS for overlapping communication and computations when using multiple processes. Furthermore, using standard memory allocation on a GPU and Unified Memory versions are compared, the latter including programmer added prefetching. Performance of a hybrid CPU+GPU version as well as scaling across multiple GPUs are demonstrated showing good speed-ups of the approach. Finally, the performance per power consumption of selected configurations are presented for various numbers of streams and various relative performances of GPUs and CPUs.

Keywords: GPGPU, overlapping computations and communication, MPS, Unified Memory, performance, power consumption

Mathematics Subject Classification 2010: 68M20, 65Y05, 68N15

1 INTRODUCTION

In today's high performance computing (HPC) systems, several computing devices are typically used – multi- and many-core CPUs, GPUs, FPGAs. All have their advantages and disadvantages depending on particular types of codes and applications [9]. Most of HPC systems nowadays feature either traditional multicore CPU + accelerator (GPU, Intel Xeon Phi x100) or manycore CPUs (such as Intel Xeon Phi x200 or Sunway manycore CPUs in the Sunway TaihuLight cluster). Selected application examples of applications running on such systems include data encryption and decryption algorithms [32], pattern matching for deep packet inspection [26], RNA secondary structure prediction [27], parallel implementation for a DVB-RCS2 receiver [46], parallelization of large vector similarity computations [14, 11], stitching large scale optical microscopy images [4], etc. For this reason, efficient management of computations among these processors is a key to achieving high throughput, especially for incoming data streams that must be processed under time constraints. GPGPU has become very popular for processing large data sets in the Single Instruction Multiple Threads fashion. As long as processing in threads does not result in too much divergence, one can achieve very high processing throughput. Especially important is also fast delivering of input data from host memory to GPU memory and results back from GPU memory to the host. This can be achieved through overlapping communication and GPU and CPU computations by using multiple streams. This topic is investigated in this paper in detail, in terms of performance for various numbers of streams, threads managing computations in a CPU + GPU setting, using the standard GPU memory management and Unified Memory [34] approaches. Furthermore, as today's HPC is not only about performance, power consumption is also considered, in the context of performance to power consumption ratio of various configurations.

The approach adopted in this paper includes analysis based on a custom built benchmark, described in Section 3, that assumes input data that is composed of multiple data chunks which are fed into CUDA streams to GPUs or processed on multicore host CPUs. The benchmark allows for various compute time to host-device communication time ratios, numbers of streams and threads managing computations and communication and thus, depending on the values of parameters, can be regarded as a template representative of many real world applications.

The objective of this work is to assess performance and selected performance/power characteristics of parallel processing of a data stream which is passed for computations to either GPUs using CUDA streams or to GPUs and CPU cores in a hybrid CPU + GPU approach. The contribution includes assessment of preferred numbers of streams for various GPU architectures, preferred application architecture in terms of the number of host GPU management and computing threads, assessment of performance differences between standard memory management, Unified Memory and Unified Memory with prefetching, all for various compute to communication ratios. Additionally, performance per power consumption is evaluated for selected configurations. Furthermore, scaling from 1 to 4 NVIDIA Tesla V100 GPUs

of DGX Station installed at the Faculty of ETI, Gdansk University of Technology, is presented.

The outline of the paper is as follows. Section 2 presents the existing related work and contributions of this paper in that context, Section 3 the processing model and design of the benchmark used for experiments, Section 4 tests and results including testbed systems, impact of multiple streams on performance using various numbers of threads managing computations, launching computations from multiple processes with and without MPS, performance with and without Unified Memory, scalability of hybrid CPU + GPU code, scaling across multiple GPUs and performance-power consumption ratios for hybrid configurations. Finally Section 5 presents conclusions and future work.

2 RELATED WORK

2.1 Mechanisms for Data Management in Selected GPU-Aware Parallel Programming APIs

Overlapping computation on the GPU, CPU as well as CPU-GPU and GPU-CPU communication is a well known technique that allows to minimize execution time of an application using GPUs [14, 31, 13, 25]. This approach can be used for both batch processing if the data is already available when the application starts or is incoming to a node in possibly many data streams.

In CUDA, kernel functions are executed in parallel on a GPU by a grid which is composed of thread blocks each of which consists of a number of threads. Blocks within a grid and threads within a block can be lined up in 1, 2 or 3 dimensions. Various operations (out of host-to-device communication, device-to-host communication, kernel execution) submitted to two different streams can potentially be overlapped in H2D, compute and D2H queues. Thus, a larger number of streams can potentially allow better overlapping (so-called n-way in the case of n streams [41]) if there is potential for that in the application and if the GPU and the driver support that. Potentially kernels can also be executed in parallel, depending on their requirements and the GPU. Unified Memory allows allocation and access to data from the host and device sides and page migration, transparent to the user. The contribution of the paper is how a particular configuration (with a given number of streams) for a given GPU (GPUs of various architectures were used) benefits which is otherwise very difficult to predict given these factors.

It should be noted that OpenCL offers a similar programming model to CUDA but targeting systems with both GPUs as well as CPUs [13, 15]. Specifically, a kernel can be executed on a compute device by a structure called NDRange that consists of work groups which in turn consist of work items. Both work groups within the NDRange and work items within a work group can be lined up in 1, 2 or 3 dimensions. A kernel is executed by work items in parallel within a context that is associated with one or more devices. Input and output data are managed through memory objects. Overlapping can be achieved using command queues, similarly to

using streams in CUDA. OpenCL version 2.0+ allows to use Shared Virtual Memory which allows codes running on the host and a device to share data. Various modes including coarse-grained or fine-grained with the possibility of accessing locations concurrently if SVM atomic operations are supported. Another high level API allowing to use GPUs in a way similar to OpenMP is OpenACC [13, 15]. OpenACC allows to use directives for instructing parallelization of code regions, specifically loops as well as scoping of data and synchronization. Data related directives allow to specify allocation, releasing memory and rely on the concept of reference counters to data.

Assessment of benefits and the performance of Unified Memory was done previously in [22], but it was only for batch type input data for applications such as verification of Goldbach's conjecture, 2D heat transfer analysis and adaptive numerical integration. That research was then extended with evaluation of not only the basic Unified Memory code against the standard approach but also Unified Memory with prefetching [23]. Results were presented for four applications: Sobel and image rotation filters as well as stream image processing and computational fluid dynamic simulation. Tests were performed on Pascal and Volta architecture GPUs, specifically NVIDIA GTX 1080 and NVIDIA V100 cards. Furthermore, evaluation of Unified Memory oversubscription over the standard manual management approach was provided, generally showing slight benefits of the latter, if implemented efficiently. In those contexts, the contribution of this paper is assessment of impact of the number of streams with Unified Memory, assessment of NVIDIA MPS's performance and consideration of power consumption with the number of streams in parallel processing with CUDA.

2.2 Selected Works on Efficiency of Using Multiple Streams Using GPUs

There are studies in the literature on efficiency of using multiple streams using GPUs. For instance, paper [20] investigates the impact of using various numbers of streams on the performance of such an application with a theoretical formula for the best number of streams. It was considered in terms of the number of iterations of a loop within a kernel. Tests were performed for GTX 280 and GTX 480 cards which are not widely used anymore. GPU architectures have also changed considerably since then. In paper [12], the author analyzed and compared the performance of processing on a GPU using 1, 2 and 4 streams for modern GPUs: mobile NVIDIA GeForce 940MX, desktop GTX 1060, server Tesla K20m and Tesla V100. Tests were performed for various compute time to host-device communication time ratios proving large benefits of using 2 or 4 streams for overlapping communication and computations and showing relative performances of the tested GPUs. Compared to [20] this paper contributes by analysis on newer GPUs, consideration of Unified Memory approach and performance to power consumption analysis. Compared to [12] this paper brings testing using more streams, multi-threaded and single-threaded applications, MPS as well as performance to power consumption

considerations. Apart from multiple streams, concurrent kernel execution is also possible on GPUs. Paper [45] investigates approaches such as context switching, manual context funneling and automatic CUDA context funneling but tests were performed on older CUDA 4 and earlier versions and demonstrated that automatic CUDA context funneling (sharing a context among process threads) is very efficient. Work [29] proposes a detailed computation-bound single kernel performance model for understanding the resource scheduling system with CUDA streams and focuses on multi-kernel concurrency. Similarly, paper [8] investigates conditions needed for concurrent execution of kernels simultaneously.

2.3 Using Multiple Streams for Various Applications

Deployment of multi-stream processing for GPU based systems for particular applications has been analyzed in the literature. In paper [43] authors focus on performance improvement through more effective overlapping of communication and computations using OpenMP as well as multiple CUDA threads. Many threads control each GPU and the authors have launched 4 CUDA streams for each pair of neighboring GPUs to overlap communication and computation of inner domain points. A 3D stencil use case was used to demonstrate benefits over previous solutions. Tests were performed on Kepler and Fermi cards. Compared to that work, this paper considers a model with independent input data chunks rather than geometric Single Program Multiple Data paradigm [13], considers more streams, Unified Memory and power consumption for a more recent Pascal card. In paper [19] authors focus on improvement of performance of Sparse matrix-vector multiplication (SpMV) code using many GPUs installed within a node. Optimization is performed using multiple OpenMP threads that control particular GPUs as well as multiple CUDA streams for overlapping. Benefits of such improved approach using 2 GPUs are shown against a naive 1 GPU system implementation for a variety of sparse matrices. Compared to that approach, this paper considers hybrid CPU + GPU processing, investigates multiple streams, Unified Memory and performance to power consumption ratios. Paper [35] proposes a multi-stream implementation of stereo disparity estimation and anaglyph video frame generation using GPUs. Specifically, multiple threads are started using Pthreads, each of which manages a certain number of streams. Performance is presented for a thread count between 1 and 8 and the number of streams between 1 and 8 showing considerable speed-ups of the solution with 100 frames per second for $1\,024 \times 1\,024$ color images. GeForce GTX780 cards where used for experiments. Paper [38] contributes by proposal of a parallel CPU + GPU code for image formation in scanning transmission electron microscopy. Similarly to this work, an algorithm for parallelization using multicore CPUs and GPUs are provided, with assessment of benefits from using multiple CUDA streams. In that context, this paper contributes by analysis of various numbers of streams, Unified Memory and performance to power consumption ratios for similar computations. Utilization of CUDA streams for parallel implementation of a genetic algorithm is presented in paper [39]. Data stream processing accelerated

using GPUs in the context of DBMSes is discussed in [36] for data representation better matching the GPU architecture. Similarly, this paper contributes by consideration of various stream and thread CPU + GPU configurations, Unified Memory and performance to power consumption ratios.

2.4 Selected Frameworks and Environments for Processing Data Using GPUs

Paper [24] provides analysis of programming environments for processing large amounts of data efficiently. Specifically, the work investigates programmability vs. performance such that programs can increase their performance at the cost of decreasing programmability. Java and Stream API, C/C++ and OpenMP, C/C++ and CUDA (with and without CUDA streams) are compared. Power-aware computations for data processing is also an important research topic considered today [16]. There exist frameworks that provide higher than OpenMP, CUDA and MPI programming abstractions to processing data streams using GPUs, good performance and relatively easy-to-use programming models. Available solutions for data streaming include, in particular, Spark [47], Storm [28, 21], Storm working in a geographically distributed and highly variable environment [6], FastFlow [2], extension of FastFlow for a network of multi-core workstations [1], Flink [5, 18], PiCo [33], Thrill [3]. Paper [48] describes GStream that is a scalable framework suited for a cluster of GPUs with GStream API over CUDA, Pthreads and MPI. It is demonstrated for benchmarks such as FIR, MM, FFT, IS and LAMMPS that it offers very good speed-ups, only slightly worse that raw CUDA. For this and the following high level approaches, the contributions of this paper can be used for improvement of performance of lower level building blocks and mapping computations onto GPUs and CPUs as well as optimization of CPU-GPU communication. Another general data processing platform utilizing GPUs is G-Storm [7] which can be used for various applications and data types and provides a high level programming approach. It handles data transfers and resource allocation automatically. If data is to be further used on the same GPU in subsequent operations, it will not be copied back and forth between the host and the GPU. G-Storm very much relies on CUDA MPS that allows to create a single context that can be used from many processes on the host. It should be noted that this paper evaluates gains from MPS and shows benefits of multi-threaded and CUDA multi-stream approach for even better performance and such can be used to improve existing systems. Paper [40] proposes an efficient real-time system for processing large amounts of high frequency data such as video and text. The approach integrates Hadoop for parallel processing, Spark for the real-time component and GPUs for processing. Matrix type data is processed on GPUs similarly to MapReduce. The authors conclude that the proposed solution is faster than CPU MapReduce. Such a system could also benefit from low level optimization between host and GPUs presented in this paper. Work [44] proposes a CPU + GPU system for processing a large number of incoming data streams with hard real-time constraints. A scheduler running on the CPU side distributes

streams among CPUs and GPUs for high utilization of the system in order to meet the constraints. The solution was evaluated using an AES-CBC encryption kernel on thousands of streams proving over 80 % more data processing rate than a single GPU system. Paper [42] presents KernelHive that can be used to optimize scheduling and execution of processing using a stream of multiple independent data chunks on hybrid CPU + GPU systems. Efficient multithreaded data stream processing in a workflow management system called BeesyCluster, either within a high performance workstation or even spanning multiple clusters, is presented in paper [10]. In that context, the contribution of this paper is optimization of internal building blocks for efficient GPU management and consideration of power consumption as well.

3 PROCESSING MODEL AND DESIGN OF BENCHMARK

This section presents the custom-developed application benchmark that is representative of various applications run on GPUs or in a hybrid CPU + GPU environment. Many variables have been considered and can be changed in the proposed processing model and as such were used for subsequent tests. Design of the benchmark application is shown in Figure 1. It is assumed that the application processes a sequence of input data packets such that two data packets serve as input to a processing function that produces output data. This general assumption corresponds to many real life applications, depending on relative sizes of output and input data, e.g. multiplication, addition or other operations on matrices that are important computational steps in various artificial intelligence applications such as deep neural network training. Parallelization involves the following elements and ideas:

1. At a high level of parallelism, OpenMP threads are spawned – one thread per each GPU and additionally one thread managing computations on a multi-core CPU(s). These threads fetch input data from memory in a critical section and pass for computations either to a GPU or the CPU(s). This scheme, working in a loop, effectively supports dynamic load balancing among compute devices.
2. Nested OpenMP parallelism is used for parallelization with many threads on the CPU(s).
3. Input data can be stored in regular RAM from which it can be sent to GPU's global memory explicitly or stored in previously allocated space in Unified Memory. In the latter case, prefetching can be turned on for enabling overlapping computations with host-device communication. In the case of the Unified Memory based version, streams are still used for maximum concurrency of operations [34].

The benchmark allows to set various modes and parameters and correspondingly allows to mimic behavior of various applications following the assumed processing pattern:

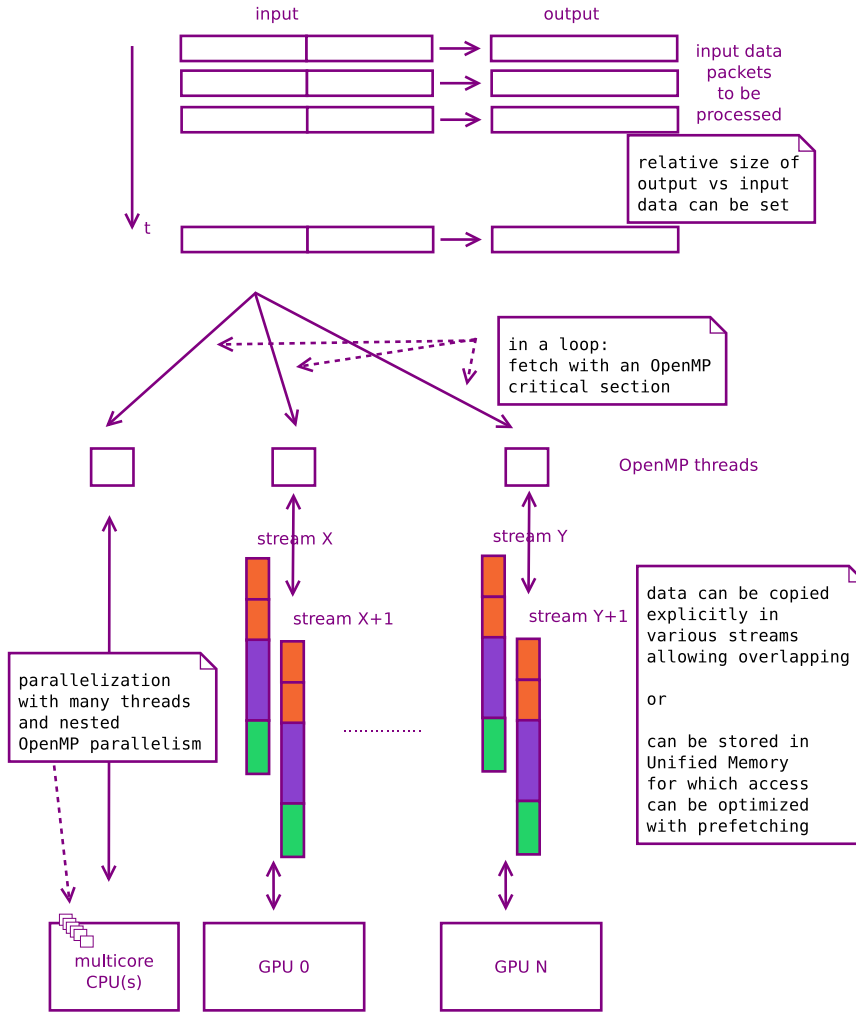

Figure 1. Proposed processing framework

1. memory mode – several modes are possible:

   (a) allocation of host memory *std* using `cudaHostAlloc()` with flag `cudaHostAllocPortable` that does allow subsequent overlapping computations and communication in various streams,

   (b) allocation of memory *UM* using Unified Memory (UM) by calling `cudaMallocManaged()` that allows to use the same pointer from both host threads to write input data, from a kernel to read input data and write output results as well as from the host to read output.

   (c) allocation of memory *UMprefetch* using Unified Memory with data prefetching through `cudaMemPrefetchAsync()` for streams to be used in subsequent steps,

2. compute time to host-device communication time ratio that corresponds to the computational time on a given input data chunk divided by the communication time of this data chunk (CPU-GPU-CPU),
3. output-input ratio that denotes the ratio of the size of output data to the size of input data,
4. stream count – the number of streams per one GPU used,
5. host thread count – the number of threads among which computations are scheduled on CPU(s) cores,
6. GPU count ⟨ids of GPUs⟩ – the number and ids of GPU(s) to be used for computations.

In each experiment, unless otherwise noted, data chunk was 256 KB in size and 1.6 GBs of data was processed. In the test we assumed 1 024 threads per block and the total number of threads was 262 144. In the GPU kernel function, a thread fetches its unique index in a grid and processes data from two input arrays into a result stored in its own location (depending on its index) in an output array. Specifically, it computes averages of selected vector elements of the two input arrays and computes a distance between the averages which is added to the final output. All arrays are stored in global memory and the kernel uses 3 variables as temporary indices and one variable as a loop counter. Compute time to host-device communication time ratio is configured with a proper number of iterations of the aforementioned loop.

4 EXPERIMENTS AND TESTS

4.1 Testbed Systems

For experiments, we used the benchmark described in Section 3 run on three modern multicore CPU(s) + GPUs workstations. Specifications of the systems are listed in Table 1. Testbeds 1 and 2 feature 2 Intel Xeon CPUs + 2 NVIDIA GPUs, of various generations while testbed 3 an Intel Xeon CPU + 4 NVIDIA Tesla V100 cards used for testing scaling across multiple GPUs.

For each particular configuration, unless otherwise noted, 10 tests were performed and the average value is presented.

4.2 Impact of Multiple Streams on Performance

The purpose of the following experiments is to determine the impact of using multiple streams for overlapping computations and communication and finally execution time of a GPU enabled application.

Testbed	1	2	3
CPUs	2 × Intel Xeon CPU E5-2620v4 @ 2.10 GHz	2 × Intel(R) Xeon(R) CPU E5-2640 @ 2.50 GHz	Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20 GHz
CPUs – total number of physical/logical cores	16/32	12/24	20/40
System memory size (RAM) [GB]	128	64	256
GPUs	2 × NVIDIA GTX 1070 (Pascal)	2 × NVIDIA Tesla K20m (Kepler)	4 × NVIDIA Tesla V100 (Volta)
GPUs – total number of CUDA cores	2 × 2 048	2 × 2 496	4 × 5 120
GPU Compute capability	6.1	3.5	7.0
GPU memory size [MB]	2 × 8 192	2 × 5 120	4 × 16 384
Operating system	Ubuntu Linux version 4.15.0-36-generic	CentOS Linux version 3.10.0-862.9.1.el7.x86_64	Ubuntu Linux version 4.4.0-83-generic
Compiler/version	CUDA compilation tools, release 9.1, V9.1.85, gcc 7.3.0	CUDA compilation tools, release 9.1, V9.1.85, gcc 4.8.5	CUDA compilation tools, release 9.0, V9.0.176, gcc 5.4.0

Table 1. Testbed configurations

The following tests have been performed for several values of compute time to host-device communication time ratio, for several GPU cards and for the number of streams between 1 and 32. Additionally, two different ways of launching computations on a GPU are presented and compared:

A: One thread per GPU managing computations through one or more streams. In this case, the thread launches CPU-GPU communication, kernel and GPU-CPU communication asynchronously through streams one after another.

B: As many threads as the number of streams are launched per GPU, each of which launches CPU-GPU communication, kernel, GPU-CPU communication in a separate stream. Threads need to synchronize while fetching new input data packets.

Figure 2 presents the results for these versions for particular numbers of threads and streams used for testbed 1 while Figure 3 does so for testbed 2. It can be seen that, in general, best results were obtained using one dedicated host thread per GPU launching communication and computations to multiple streams with 2 streams for testbed 1. For testbed 2 the same implementation offers best results with 2+ streams with small differences between the number of streams larger than 2–4. At the same

time, we can see very small deviations between runs (10 measured) for testbed 2 (default affinity values are presented). For testbed 1, we can observe larger deviations for configurations with multiple host threads launching operations on the GPU (we present threads/close affinity values). These differences might stem from various operating system settings and a compiler version as the CUDA versions were the same.

4.3 Launching Computations from Multiple Processes Using MPS

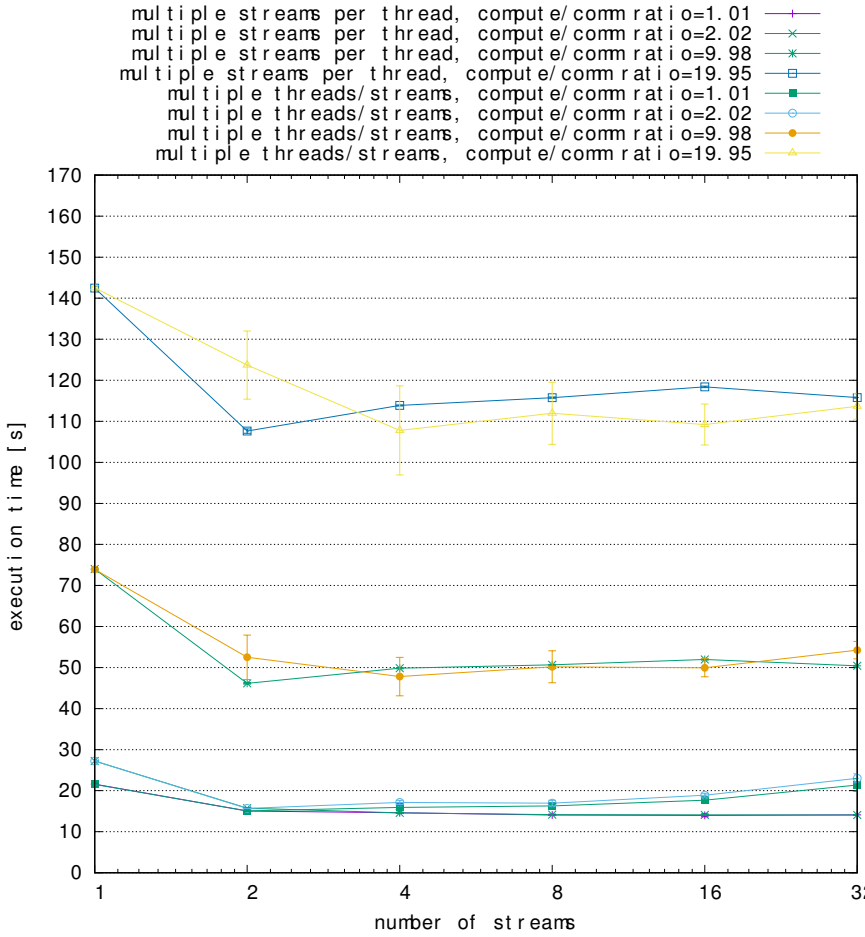

Figure 2. Comparison of implementations with various numbers of threads and streams on a GPU, testbed 1, bars represent standard deviation

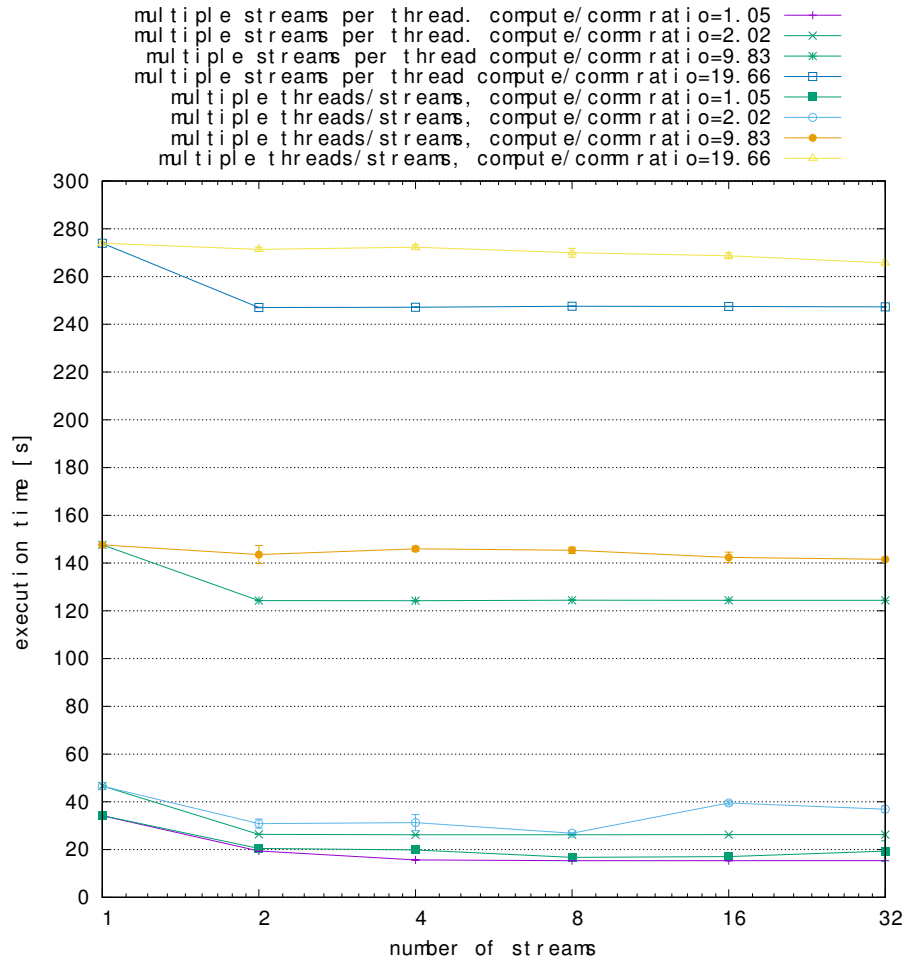

Figure 3. Comparison of implementations with various numbers of threads and streams on a GPU, testbed 2, bars represent standard deviation

In case there is no dedicated parallel application available for parallel processing of incoming data streams to a computer node, it is probable that several processes working in parallel will try to submit the work for processing on a GPU that will be shared in such a case. This may lead to inefficiency of the processing. One solution would involve writing a dedicated multi-stream application as analyzed in this paper. An alternative approach has been made available by NVIDIA through Multi Process Service (MPS) that tries to overlap CPU-GPU communication and processing on a GPU from various contexts. It does not require code modifications which is a considerable advantage. Details of its usage can be found in [13]. Figures 4

and 5 present the results of using the MPS enabled configuration vs the standard configuration for testbed 1 and testbed 2, respectively. Five tests were performed for each configuration and the average value is presented. The results really indicate that the solution improves execution time visibly, except for smaller compute time to host-device communication time ratio for testbed 1. In these tests, two different processes were launched in parallel on the number of data chunks half the sizes of the cases shown in Figures 2 and 3. It should be noted that the best results obtained for 2 streams shown in Figure 2 still offer better execution times while the ones shown in Figure 3 show practically the same or marginally a better performance compared to the one with MPS.

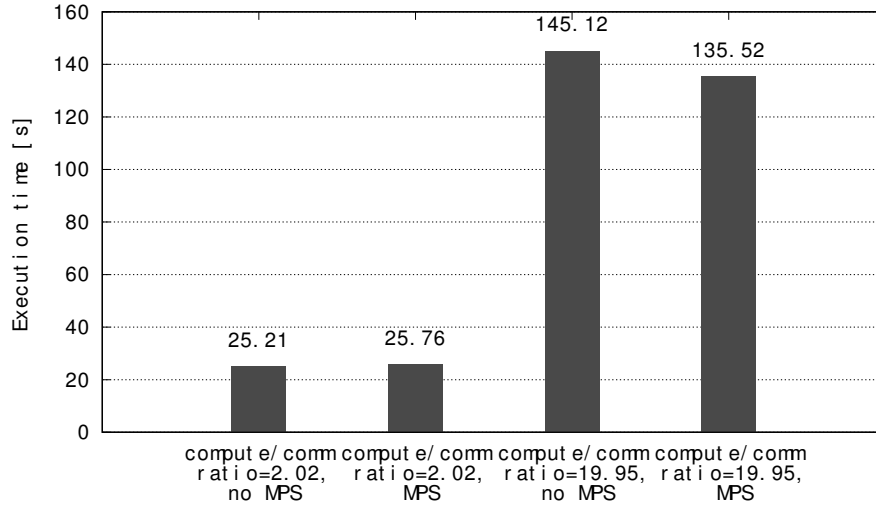

Figure 4. Comparison of performance with and without NVIDIA MPS, testbed 1

4.4 Performance with Unified Memory

Since the latest cards and CUDA versions offer the benefit of easier programming with Unified Memory, this experiment is to show the performance of Unified Memory based implementation compared to previous best cases. The test involves setting input data on the host and launching a kernel that processes data packets on the GPU. Subsequently, results are read from the host side in order to find the maximum of results and display to the user.

The basic UM enabled version was further optimized using data prefetching (we denote this version by UMprefetch). Specifically, the data packet to be processed in a subsequent step in a given stream is prefetched using a call to function `cudaMemPrefetchAsync(...)` on the two input buffers.

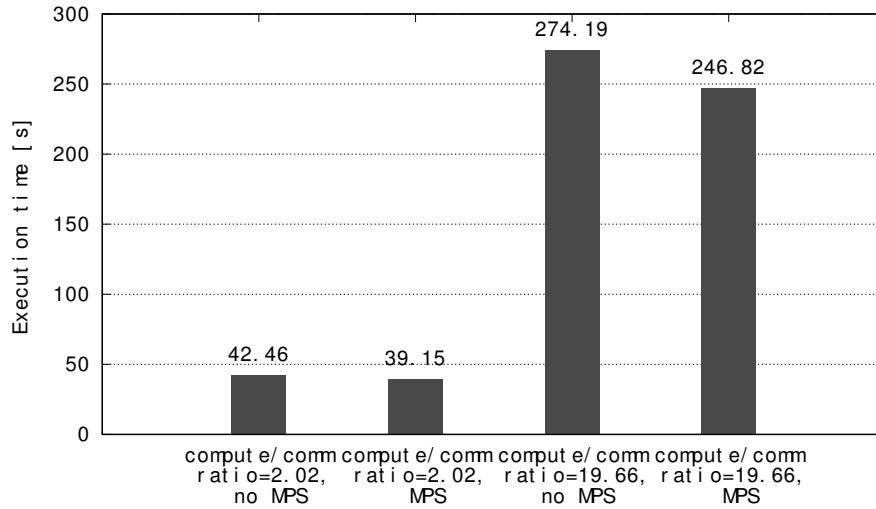

Figure 5. Comparison of performance with and without NVIDIA MPS, testbed 2

Figure 6 presents comparison between std, UM and UMprefetch versions for 1 GPU on testbed 1 while Figure 7 presents comparison between std, UM and UMprefetch versions for 2 GPUs on testbed 1, data size proportionally smaller than in the previous tests. It can be seen that prefetching really improves the performance but still the standard memory optimized multi-stream version offers the best performance. This is in line with some previous works comparing performance of Unified Memory to standard based versions showing generally similar or worse performance in [22], [30] and [37] in return for an easier programming model. This paper confirms it for various compute time to host-device communication time ratios, numbers of streams and 1 and 2 GPUs.

4.5 Scalability of Hybrid CPU + GPU Code

The purpose of the following experiments (using standard memory management) is to show scalability of the hybrid parallel code on the two testbeds with 1 GPU, 2 GPUs as well as host threads engaged for computations, for various GPU/CPU performance ratios. The latter can vary depending on an application. In this case, 2 streams per GPU were used for testbed 1 and 4 streams per GPU for testbed 2. The same thread affinities and binding as in Section 4.2 were used.

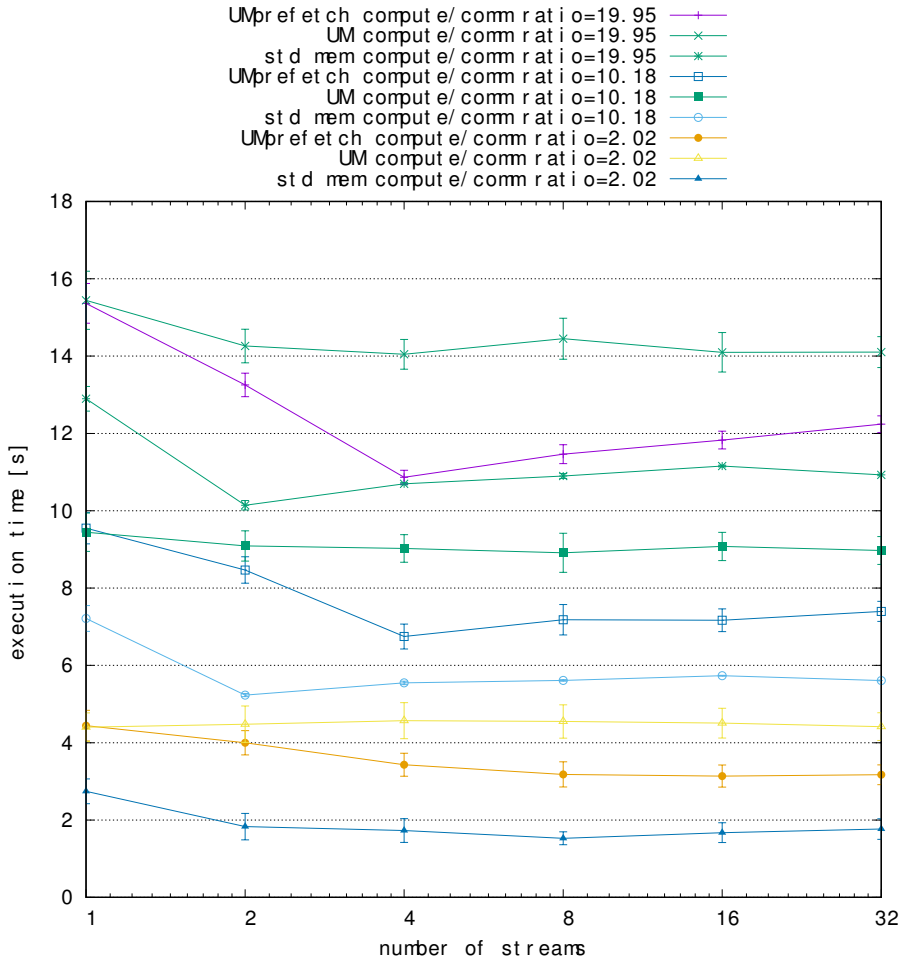

Figure 6. Comparison of standard memory (std), Unified Memory (UM) and optimized Unified Memory (UMprefetch) implementations – testbed 1, 1 GPU, bars represent standard deviation

The results presented in Figure 8 for testbed 1 and in Figure 9 for testbed 2 allow to assess GPU/CPU performances for which adding host threads for computations brings visible savings in execution times. It can be noticed that 2 GPUs configurations achieve relatively better performance than proportional scaling from 1 GPU configurations, apparently due to using one of the GPUs for display as well. Scaling from 1 to 2 GPUs is clearly visible. Increasing the number of host threads decreases application execution time at rates very much depending on GPU to CPU performances, with practically no gains when using 2 GPUs and GPU/CPU per-

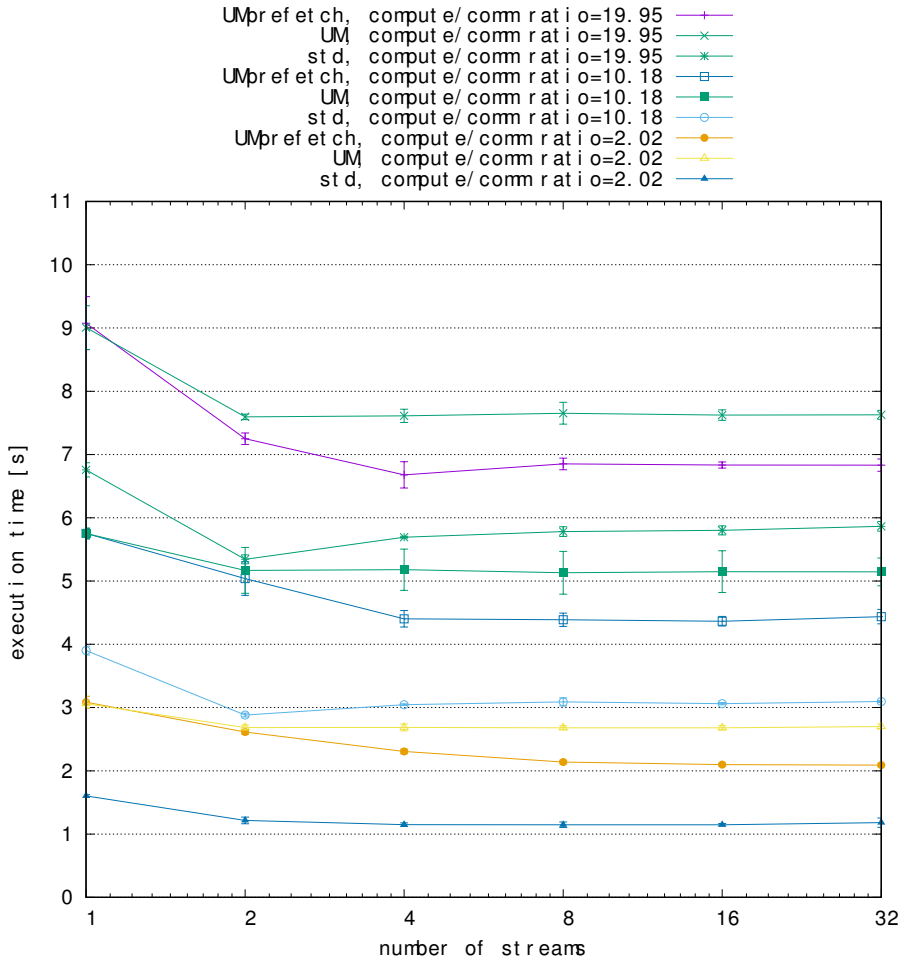

Figure 7. Comparison of standard memory (std), Unified Memory (UM) and optimized Unified Memory (UMprefetch) implementations – testbed 1, 2 GPUs, bars represent standard deviation

formance ratio around 30 for testbed 1. It should be kept in mind that in case some CPU cores are used for computations, still as many threads as the number of GPUs are used for management of computations on the GPUs. Furthermore, the threads managing computations on the GPUs and the CPUs fetch next data packets synchronizing on an OpenMP critical section which also decreases potential speed-ups.

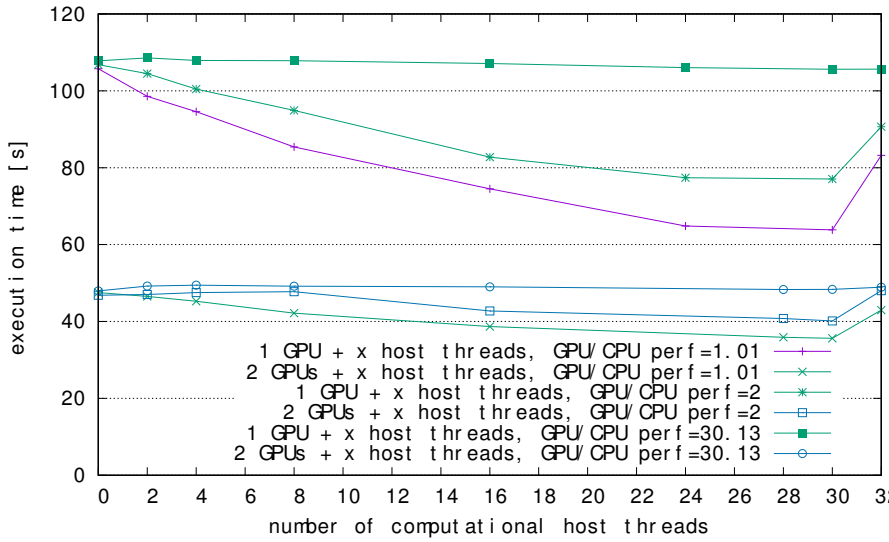

Figure 8. Performance of a hybrid GPU+CPU implementation for various GPU/CPU performances and numbers of host threads, testbed 1

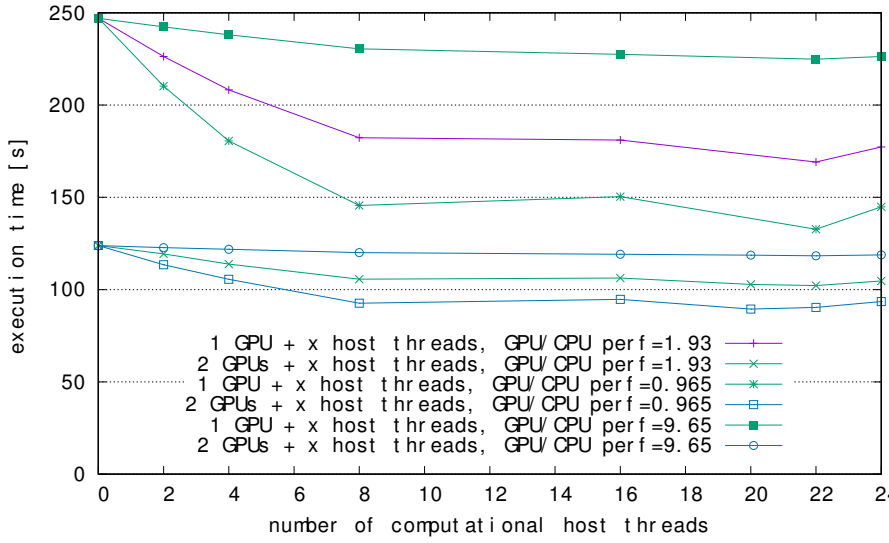

Figure 9. Performance of a hybrid GPU+CPU implementation for various GPU/CPU performances and numbers of host threads, testbed 2

4.6 Performance-Power Consumption Ratio

In today's high performance computing systems, power consumption has become an important topic. It is considered in designs of future clusters for which the total power consumption is suggested not to exceed 20 MW for 1 Exaflop/s [17]. In this context, we analyze the performance to power consumption for the various configurations analyzed in this paper, specifically for:

1. various numbers of streams involved when using 1 GPU,
2. GPU + CPU configurations with various numbers of host threads involved in computations.

GPU performance was calculated as the inverse of the sum of data chunk CPU-GPU communication, processing and GPU-CPU result transfer times. CPU performance was calculated as the inverse of data chunk processing time on the CPU(s). Average power consumption of various configurations was measured using a hardware meter within a 10 minute period for each configuration. A bash script was used to run a particular configuration. Figure 10 shows normalized performance computed as inverse of execution time divided by average power consumption through application run for 1 GPU and various numbers of streams. Normalization of performance was done by dividing results by quotients of compute time to host-device communication time ratios of various configurations. It can be seen that normalized performance per power consumption has its maxima depending on compute time to host-device communication time ratio. It is interesting to note that for 2+ numbers of streams the best normalized ratios are observed for compute time to host-device communication time ratio 9.98 and lower for the other ratios.

Furthermore, the performance by power consumption is shown for 1 and 2 GPU configurations with addition of various numbers of host threads used for computations using testbed 1. The results for the GPU/CPU performance ratio of around 30 are shown in Figure 11. It can be seen that, while execution times slightly decrease, as shown in Figure 8 before, the performance-power consumption ratio goes down due to too little improvement of execution times thanks to CPU compared to its power consumption. Had the computational power been better compared to GPUs, the ratio would have been better for higher numbers of host threads. Such a simulation was performed and its results are shown in Figure 12 for a smaller GPU/CPU relative performance ratio. It can be seen from the tests that for GPU/CPU performance equal to 2 using more host threads offers benefits in terms of performance/power consumption.

4.7 Scaling Across Multiple GPUs

The following experiments demonstrate how the code scales across GPUs in testbed 3 with 4 NVIDIA Tesla V100 Volta series GPU cards. Firstly, Figure 13 presents how the numbers of streams affect performance for the largest configuration shown in

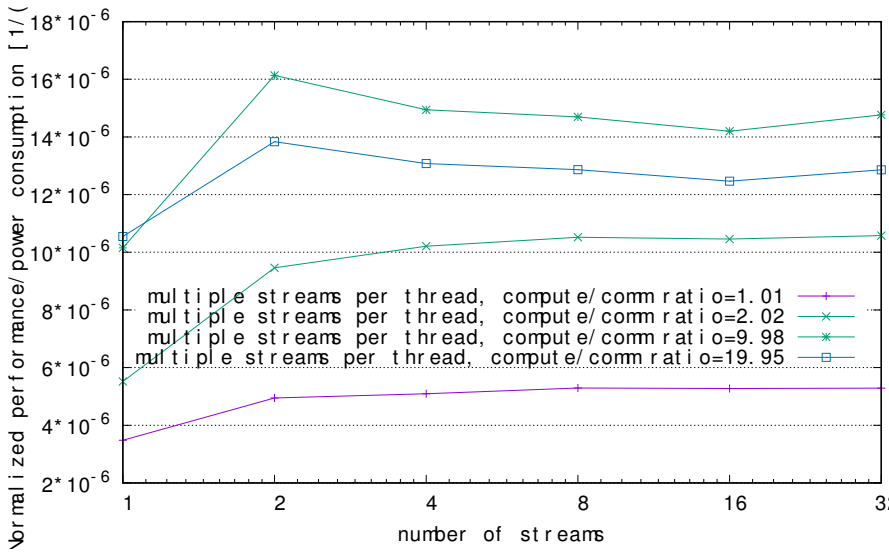

Figure 10. Normalized performance by power consumption for 1 GPU and various numbers of streams, testbed 1

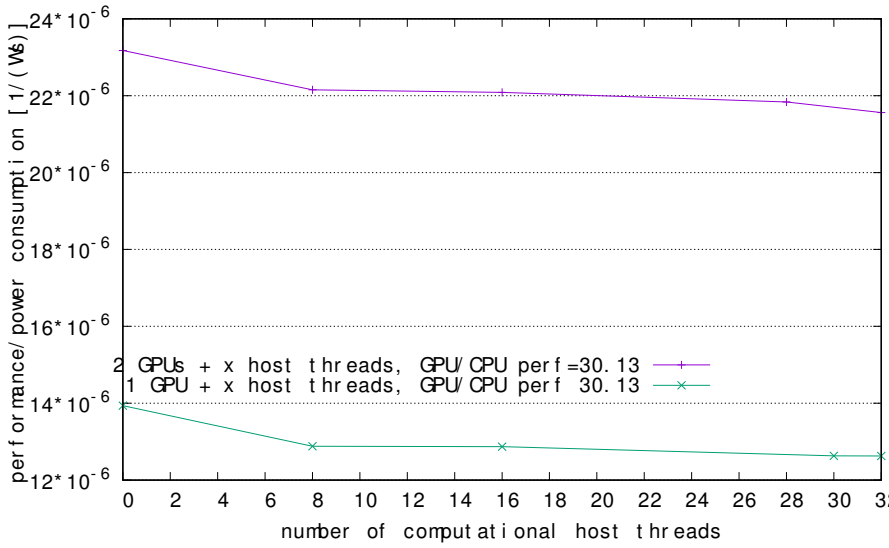

Figure 11. Performance by power consumption for 1 and 2 GPU configurations and various numbers of host threads, GPU/CPU performance = 30.13, testbed 1

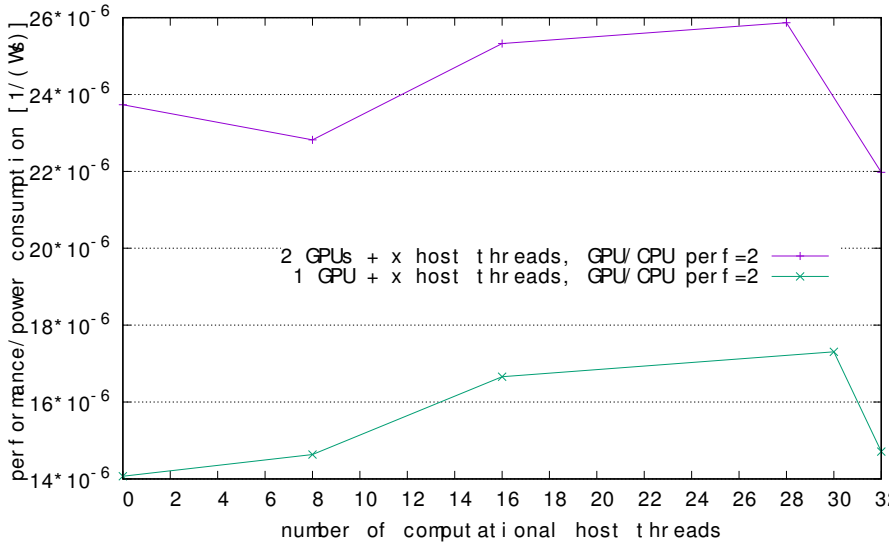

Figure 12. Performance by power consumption for 1 and 2 GPU configurations and various numbers of host threads, GPU/CPU performance = 2, testbed 1

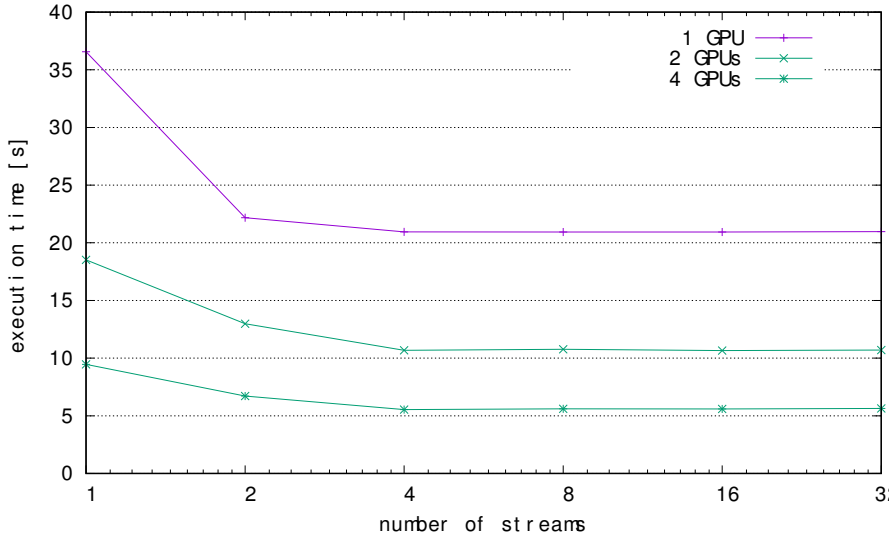

Figure 13. Execution time vs number of streams for various numbers of GPUs, largest case from Figure 2, testbed 3

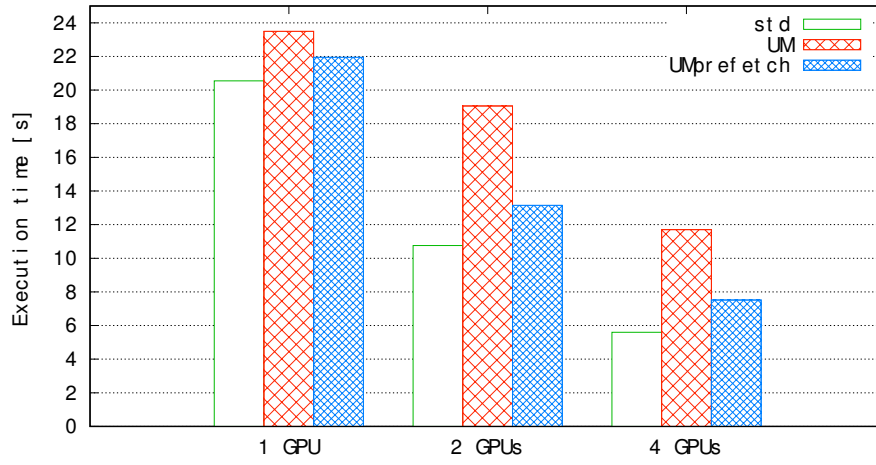

Figure 14. Execution time vs memory management solutions for various numbers of GPUs, largest case from Figure 2, 8 streams per GPU, testbed 3

Figure 2. Then, assuming 8 streams per GPU which (the configuration which already gives small execution times on the flat parts of the chart), execution times are shown for the standard memory management, UM and UMprefetch as in the previous cases. It can be seen in Figure 14 that, again, the UM version offers visible overhead over the standard memory management version. UMprefetch, thanks to manual prefetching, offers performance half-way between these two versions for 1 GPU. For 2 and 4 GPUs, it is worse than the standard memory management version by about 30 % of the difference between the other two versions.

5 CONCLUSIONS AND FUTURE WORK

In the paper we analyzed the performance and performance to power consumption ratio of multi-stream data processing on modern multicore CPU+ GPU systems. Using a benchmark that allows to set up various compute time to host-device communication time ratios, number of streams, number of threads managing computations it was possible to assess performance of various configurations on modern testbeds with Intel Xeon CPUs and NVIDIA Tesla K20m, GTX 1070 Pascal and Tesla V100 Volta series cards. The benefits of using a properly implemented multi-stream code were shown compared to GPU computations managed by various threads or processes for various numbers of streams. Furthermore, the benefits of such code compared to standard Unified Memory and Unified Memory with prefetching were shown showing performance gains at the cost of increased programming

effort. Additionally, the gains from using NVIDIA Multi Process Service have been presented for multi-process configurations. The performance to power consumption ratios have been shown for various numbers of streams and compute time to host-device communication time ratios as well as for hybrid CPU + GPU configurations for various numbers of computational threads on the host and relative GPU and CPU performances. Scalability of the code was presented between 1 and 4 GPUs using NVIDIA Tesla V100 cards.

The results can be generalized as follows. For the considered data stream processing application and various compute to communication ratios, using multiple streams, at least 2 offered visible benefits, with best results using one dedicated host thread per GPU launching communication and computations to multiple streams. Some configurations result in best execution times for 2, 4, 8 or even 16 streams but we can note that benefits over 4 streams, if any, are very small. Secondly, we confirmed that using NVIDIA MPS gives visible benefits especially for larger compute to communication ratio. Furthermore, for various compute to communication ratios we confirmed that Unified Memory brings visible overhead over the standard memory management implementation while a Unified Memory version with manual prefetching ranks between the two. For CPU + GPU codes, increasing the number of computational host threads up to the number of available logical processors decreases application execution time at rates very much depending on GPU to CPU performances with considerable gains with CPU performance in the same order as the one of the GPU. It has been shown that the observed performance per power consumption varies with the number of streams, GPU to CPU performance ratio and the number of computational host threads.

These results can be used as guidelines for best performance implementations for various applications as the tests are of generic nature and, depending on values of particular aforementioned parameters, are representative of many applications. Specifically, obtained results can be used for implementation of building blocks for data stream frameworks using multi-core CPUs and GPUs, especially multi CUDA stream communication optimization.

Future work includes extending the scope of the conducted tests performed on systems with NVIDIA Tesla V100, specifically regarding various CPU + GPU configurations, tests for various compute/communication ratios, as well as extending tests to larger V100 based systems such as NVIDIA DGX-1 featuring 8 V100 GPUs. More experiments with thread affinities will be conducted, with research of their impact for particular codes. Additionally, we plan to incorporate the outcomes of this work into higher level frameworks such as KernelHive [42] and possibly others and investigate the impact of Unified Memory oversubscription compared to the traditional implementation model.

Acknowledgments

The research in the paper has been partially supported by the Statutory Funds of Electronics, Telecommunications and Informatics Faculty, Gdansk University of

Technology, Poland. Additionally, the author would like to express his gratitude to Aleksandra Preiss from Gdansk University of Technology.

REFERENCES

[1] Aldinucci, M.—Campa, S.—Danelutto, M.—Kilpatrick, P.—Torquati, M.: Targeting Distributed Systems in Fastflow. In: Caragiannis, I. et al. (Eds.): Euro-Par 2012: Parallel Processing Workshops. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7640, 2013, pp. 47–56, doi: 10.1007/978-3-642-36949-0_7.

[2] Aldinucci, M.—Danelutto, M.—Kilpatrick, P.—Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore. Chapter 13. In: Pllana, S., Xhafa, F. (Eds.): Programming Multi-Core and Many-Core Computing Systems. Wiley-Blackwell, 2017, pp. 261–280, doi: 10.1002/9781119332015.ch13.

[3] Bingmann, T.—Axtmann, M.—Jöbstl, E.—Lamm, S.—Nguyen, H. C.—Noe, A.—Schlag, S.—Stumpp, M.—Sturm, T.—Sanders, P.: Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++. 2016 IEEE International Conference on Big Data (Big Data), 2016, pp. 172–183, doi: 10.1109/BigData.2016.7840603.

[4] Blattner, T.—Keyrouz, W.—Chalfoun, J.—Stivalet, B.—Brady, M.—Zhou, S.: A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images. 2014 43rd International Conference on Parallel Processing, 2014, pp. 1–9, doi: 10.1109/ICPP.2014.9.

[5] Carbone, P.—Katsifodimos, A.—Ewen, S.—Markl, V.—Haridi, S.—Tzoumas, K.: Apache Flink™: Stream and Batch Processing in a Single Engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol. 38, 2015, No. 4, pp. 28–38.

[6] Cardellini, V.—Grassi, V.—Presti, F. L.—Nardelli, M.: On QoS-Aware Scheduling of Data Stream Applications over Fog Computing Infrastructures. 2015 IEEE Symposium on Computers and Communication (ISCC), 2015, pp. 271–276, doi: 10.1109/ISCC.2015.7405527.

[7] Chen, Z.—Xu, J.—Tang, J.—Kwiat, K. A.—Kamhoua, C. A.—Wang, C.: GPU-Accelerated High-Throughput Online Stream Data Processing. IEEE Transactions on Big Data, Vol. 4, 2018, No. 2, pp. 191–202, doi: 10.1109/TBDATA.2016.2616116.

[8] Cruz, R.—Drummond, L.—Clua, E.—Bentes, C.: Analyzing and Estimating the Performance of Concurrent Kernels Execution on GPUs. XVIII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2017), 2017, pp. 136–147.

[9] Cullinan, C.—Wyant, C.—Frattesi, T.: Computing Performance Benchmarks among CPU, GPU, and FPGA. Worcester Polytechnic Institute, E-project-030212-123508, 2012. https://web.wpi.edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking_Final.pdf.

[10] Czarnul, P.: A Model, Design, and Implementation of an Efficient Multithreaded Workflow Execution Engine with Data Streaming, Caching, and Storage Con-

straints. The Journal of Supercomputing, Vol. 63, 2013, No. 3, pp. 919–945, doi: 10.1007/s11227-012-0837-z.

[11] Czarnul, P.: Benchmarking Performance of a Hybrid Intel Xeon/Xeon Phi System for Parallel Computation of Similarity Measures Between Large Vectors. International Journal of Parallel Programming, Vol. 45, 2017, No. 5, pp. 1091–1107, doi: 10.1007/s10766-016-0455-0.

[12] Czarnul, P.: Benchmarking Overlapping Communication and Computations with Multiple Streams for Modern GPUs. In: Ganzha, M., Maciaszek, L., Paprzycki, M. (Eds.): Communication Papers of the 2018 Federated Conference on Computer Science and Information Systems (FedCSIS 2018), Poznań, Poland, 2018. Annals of Computer Science and Information Systems, Vol. 17, 2018, pp. 105–110, doi: 10.15439/2018F17.

[13] Czarnul, P.: Parallel Programming for Modern High Performance Computing Systems. 1st Edition. Chapman and Hall/CRC, Taylor & Francis, 2018. ISBN: 978-1138305953.

[14] Czarnul, P.: Parallelization of Large Vector Similarity Computations in a Hybrid CPU + GPU Environment. The Journal of Supercomputing, Vol. 74, 2018, No. 2, pp. 768–786, doi: 10.1007/s11227-017-2159-7.

[15] Czarnul, P.—Proficz, J.—Drypczewski, K.: Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High Performance Computing Systems. Scientific Programming, Vol. 2020, 2020, Art. No. 4176794, 19 pp., doi: 10.1155/2020/4176794.

[16] Danelutto, M.—De Sensi, D.—Torquati, M.: A Power-Aware, Self-Adaptive Macro Data Flow Framework. Parallel Processing Letters, Vol. 27, 2017, No. 1, Art. No. 1740004, doi: 10.1142/S0129626417400047.

[17] Dongarra, J.: Challenges for Exascale Computing. PARA 2010: State of the Art in Scientific and Parallel Computing, Reykjavík, Iceland, June 2010. `http://www.netlib.org/utk/people/JackDongarra/SLIDES/para-06102.pdf`.

[18] Friedman, E.—Tzoumas, K.: Introduction to Apache Flink: Stream Processing for Real Time and Beyond. 1st Edition. O'Reilly Media, Inc., 2016.

[19] Guo, P.—Zhang, C.: Performance Optimization for SpMV on Multi-GPU Systems Using Threads and Multiple Streams. 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2016, pp. 67–72, doi: 10.1109/SBAC-PADW.2016.20.

[20] Gómez-Luna, J.—González-Linares, J. M.—Benavides, J. I.—Guil, N.: Performance Models for Asynchronous Data Transfers on Consumer Graphics Processing Units. Journal of Parallel and Distributed Computing, Vol. 72, 2012, No. 9, pp. 1117–1126, doi: 10.1016/j.jpdc.2011.07.011.

[21] Jain, A.: Mastering Apache Storm: Real-Time Big Data Streaming Using Kafka, Hbase and Redis. Packt Publishing, 2017.

[22] Jarząbek, Ł.—Czarnul, P.: Performance Evaluation of Unified Memory and Dynamic Parallelism for Selected Parallel CUDA Applications. The Journal of Supercomputing, Vol. 73, 2017, No. 12, pp. 5378–5401, doi: 10.1007/s11227-017-2091-x.

[23] Knap, M.—Czarnul, P.: Performance Evaluation of Unified Memory with Prefetching and Oversubscription for Selected Parallel CUDA Applications on NVIDIA Pascal and Volta GPUs. The Journal of Supercomputing, Vol. 75, 2019, No. 11, pp. 7625–7645, doi: 10.1007/s11227-019-02966-8.

[24] Ko, B.—Han, S.—Park, Y.—Jeon, M.—Lee, B.: A Comparative Study of Programming Environments Exploiting Heterogeneous Systems. IEEE Access, Vol. 5, 2017, pp. 10081–10092, doi: 10.1109/ACCESS.2017.2708738.

[25] Kreutz, J.: CUDA Streams, Events and Asynchronous Memory Copies. April 2017. GPU Programming@Jülich Supercomputing Centre, https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/cuda/09-cuda-streams-events.pdf?__blob=publicationFile.

[26] Lee, C.-L.—Lin, Y.-S.—Chen, Y.-C.: A Hybrid CPU/GPU Pattern-Matching Algorithm for Deep Packet Inspection. PLoS ONE, Vol. 10, 2015, No. 10, Art. No. e0139301, 22 pp., doi: 10.1371/journal.pone.0139301.

[27] Lei, G.—Dou, Y.—Wan, W.—Xia, F.—Li, R.—Ma, M.—Zou, D.: CPU-GPU Hybrid Accelerating the Zuker Algorithm for RNA Secondary Structure Prediction Applications. BMC Genomics, Vol. 13, 2012, No. S-1, Art. No. S14, doi: 10.1186/1471-2164-13-S1-S14.

[28] Leibiusky, J.—Eisbruch, G.—Simonassi, D.: Getting Started with Storm. O'Reilly Media, Inc., 2012.

[29] Li, H.—Yu, D.—Kumar, A.—Tu, Y.-C.: Performance Modeling in CUDA Streams – A Means for High-Throughput Data Processing. 2014 IEEE International Conference on Big Data (Big Data), 2014, pp. 301–310, doi: 10.1109/BigData.2014.7004245.

[30] Li, W.—Jin, G.—Cui, X.—See, S.: An Evaluation of Unified Memory Technology on NVIDIA GPUs. 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015, pp. 1092–1098, doi: 10.1109/CCGrid.2015.105.

[31] Luitjens, J.: CUDA Streams: Best Practices and Common Pitfalls. nVidia, GPU Technology Conference, 2014. http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[32] Marks, M.—Jantura, J.—Niewiadomska-Szynkiewicz, E.—Strzelczyk, P.—Gozdz, K.: Heterogeneous GPU & CPU Cluster for High Performance Computing in Cryptography. Computer Science, Vol. 13, 2012, No. 2, pp. 63–79, doi: 10.7494/csci.2012.13.2.63.

[33] Misale, C.—Drocco, M.—Tremblay, G.—Aldinucci, M.: PiCo: A Novel Approach to Stream Data Analytics. In: Heras, D. B. et al. (Eds.): Euro-Par 2017: Parallel Processing Workshops. Springer, Cham, Lecture Notes in Computer Science, Vol. 10659, 2018, pp. 118–128, doi: 10.1007/978-3-319-75178-8_10.

[34] nVidia. CUDA Toolkit v10.0.130 Programming Guide. October 2018, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[35] Picos, K.—Díaz-Ramírez, V. H.—Tapia, J. J.: Real-Time 3D Video Processing Using Multi-Stream GPU Parallel Computing. Research in Computing Science, Vol. 80, 2014, pp. 87–95.

[36] Pinnecke, M.—Broneske, D.—Saake, G.: Toward GPU Accelerated Data Stream Processing. In: Saake, G., Broneske, D., Dorok, S., Meister, A. (Eds.): Proceedings of the 27th GI-Workshop Grundlagen von Datenbanken (GvD 2015), Gommern, Germany, May 26–29, 2015, CEUR Workshop Proceedings, CEUR-WS.org, Vol. 1366, 2015, pp. 78–83.

[37] Pirjan, A.—Petrosanu, D.-M.: Improving Parallel Programming in the Compute Unified Device Architecture Using the Unified Memory Feature. Journal of Information Systems and Operations Management, Vol. 8, 2014, No. 2, pp. 352–362.

[38] Pryor Jr., A.—Ophus, C.—Miao, J.: A Streaming Multi-GPU Implementation of Image Simulation Algorithms for Scanning Transmission Electron Microscopy. Advanced Structural and Chemical Imaging, Vol. 3, 2017, No. 1, Art. No. 15, doi: 10.1186/s40679-017-0048-z.

[39] Radford, D.—Calvert, D.: A Comparative Analysis of the Performance of Scalable Parallel Patterns Applied to Genetic Algorithms and Configured for NVIDIA GPUs. Procedia Computer Science, Vol. 114, 2017, pp. 65–72, doi: 10.1016/j.procs.2017.09.009.

[40] Rathore, M. M.—Son, H.—Ahmad, A.—Paul, A.—Jeon, G.: Real-Time Big Data Stream Processing Using GPU with Spark Over Hadoop Ecosystem. International Journal of Parallel Programming, Vol. 46, 2018, No. 3, pp. 630–646, doi: 10.1007/s10766-017-0513-2.

[41] Rennich, S.: CUDA C/C++. Streams and Concurrency, 2011, NVIDIA, `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`, accessed on July 19, 2017.

[42] Rościszewski, P.—Czarnul, P.—Lewandowski, R.—Schally-Kacprzak, M.: KernelHive: A New Workflow-Based Framework for Multilevel High Performance Computing Using Clusters and Workstations with CPUs and GPUs. Concurrency and Computation: Practice and Experience, Vol. 28, 2016, No. 9, pp. 2586–2607, doi: 10.1002/cpe.3719.

[43] Sourouri, M.—Gillberg, T.—Baden, S. B.—Cai, X.: Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads. 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2014, pp. 981–986, doi: 10.1109/PADSW.2014.7097919.

[44] Verner, U.—Schuster, A.—Silberstein, M.—Mendelson, A.: Scheduling Processing of Real-Time Data Streams on Heterogeneous Multi-GPU Systems. Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12), 2012, Art. No. 8, 12 pp., doi: 10.1145/2367589.2367596.

[45] Wang, L.—Huang, M.—El-Ghazawi, T.: Exploiting Concurrent Kernel Execution on Graphic Processing Units. 2011 International Conference on High Performance Computing and Simulation, 2011, pp. 24–32, doi: 10.1109/HPCSim.2011.5999803.

[46] Wang, Y.—Wang, F.—Li, R.—Dou, Y.: An Efficient CPU-GPU Hybrid Parallel Implementation for DVB-RCS2 Receiver. Concurrency and Computation: Practice and Experience, Vol. 30, 2018, No. 19, Art. No. e4529, 14 pp., doi: 10.1002/cpe.4529.

[47] Zaharia, M.—Xin, R. S.—Wendell, P.—Das, T.—Armbrust, M.—Dave, A.—Meng, X.—Rosen, J.—Venkataraman, S.—Franklin, M. J.—

Ghodsi, A.—Gonzalez, J.—Shenker, S.—Stoica, I.: Apache Spark: A Unified Engine for Big Data Processing. Communications of the ACM, Vol. 59, 2016, No. 11, pp. 56–65, doi: 10.1145/2934664.

[48] Zhang, Y.—Mueller, F.: GStream: A General-Purpose Data Streaming Framework on GPU Clusters. 2011 International Conference on Parallel Processing (ICPP), 2011, pp. 245–254, doi: 10.1109/ICPP.2011.22.

Paweł Czarnul received his Ph.D. in computer science in 2003 and D.Sc. in computer science in 2015, both from the Gdansk University of Technology, Poland. His research interests include: high performance computing, distributed information systems and processing, artificial intelligence. He is author of over 80 publications in the area of parallel and distributed processing, including book entitled Parallel Programming for Modern High Performance Computing Systems, Chapman and Hall/CRC, 2018. He is currently Head of Computer Architecture Department and Vice Dean of the Faculty of ETI, Gdansk University of Technology, Poland.