Computing and Informatics, Vol. 35, 2016, 282–298

SELECTED ISSUES ON HISTOGRAMING ON GPUS

Janusz Chwastowski, Krzysztof Korcyl

Institute of Nuclear Physics
Polish Academy of Sciences
Radzikowskiego 152
32-342 Cracow, Poland
e-mail: {janusz.chwastowski, krzysztof.korcyl}@ifj.edu.pl

Joanna Płażek, Piotr Poznański

Institute of Teleinformatics
Cracow University of Technology
Warszawska 24
31-155 Cracow, Poland
e-mail: {plazek, ppoznanski}@pk.edu.pl

Abstract. The contemporary large scale measuring systems in the real-time environment make extensive use of histogramming as a tool for the experimental data quality monitoring. The processing of a large number of data channels requires a suitable computing power where the graphical processors seem to be well suited. Histogramming operations run on the central and graphics processing units are discussed. Results of the performance measurements including various configurations of the allocation of the histograms in various parts of the memory of used devices are presented.

Keywords: Histogramming, big data processing, graphic processing unit

Mathematics Subject Classification 2010: 68-U01

1 INTRODUCTION

Histogramming is a very popular way to present the data. Due to its usual simplicity of interpretation and linearity of operations it has a very broad range of applications. It is also commonly used by the High Energy Physics community to present not only results of the advanced analyses (see for example [1, 2, 3], and references therein) but also the details of the on-line status of an experiment in a way that is easy to interpret, and hence to control. In particular, it is widely applied to test the quality of the collected experimental data as well as the experiment running conditions and its performance.

This paper summarises results of the tests concerning the histogramming operations. These tests made an extensive use of the Graphics Processing Units (GPUs). They were performed assuming different configurations of the allocation of the histograms in different parts of the physical memory of the used devices. In the following, results of the measurements targeted at the feasibility studies of the application of the GPUs for the data histogramming purposes necessary in the monitoring tasks are presented.

The paper is organised as follows. In the second section the aim of the present investigations is formulated. A short overview of the Graphic Processing Unit architecture is given in Section 3. The fourth section is devoted to the discussion of the used set-up configurations and the obtained results. The fifth section discusses planned investigations and the sixth one concludes and summarises this paper.

2 AIM OF THE MEASUREMENTS

The modern computer architectures of CPUs and GPUs explore parallelism what allows for an increase of data processing rates in large scale systems with hundreds of thousands of independent data channels. The representative samples are the large real time data acquisition systems where the scale of the architecture requires a dedicated system to monitor quality of the collected data. Contemporary measuring systems operating in the real-time environment make an extensive use of the information quantisation. In the simplest case, whole information is confined within a bit showing active/passive state of an information (data) channel. In more complex cases, the ADC (Analog to Digital Converter) and TDC (Time to Digital Converter) converters are used. The size of the data delivered by such devices reflects their resolution and is typically 8-16 bits long. In the following the channels generating 8-bit data will be considered.

The development of electronics introduces a possibility to construct systems of enormous scale consisting of hundreds of thousands of channels that are sampled with rates of the order of tens of MHz. It has to be stressed that a large number of channels folded with high throughput requires the multiport, multilevel networks and leads to the complicated topologies and architectures of the data acquisition systems. In such conditions an automated monitoring of the equipment performance is simply a must. To this end one typically uses dedicated computers, which

in the simplest case perform histogramming of the monitored quantities and typically compare the obtained results with the reference histograms. Such reference histograms are usually created during the measurements performed in the period of stable and carefully controlled running conditions of the controlled system. In a properly designed system all the information channels should be monitored. However, it should be noted that it is not required that all the input data have to be used for the monitoring purposes. It is natural that a proper, stochastic sampling of the input data streams delivers sufficient information on the system performance and quality of data being collected. As a good example one may take the data acquisition system of ATLAS [4], one of the high energy experiments running at the Large Hadron Collider (LHC). It assembles the data generated by about 140 millions of channels and forms event packets with an average size of 1.5 MB, depending on a number of channels which recorded signal above some threshold. Information from such packets, apart from the physics analysis, can also be used in the data quality monitoring system to verify whether each channel is alive and produces appropriate data.

The generalised architecture of a system targeted at the data quality monitoring is presented in Figure 1. Basically, the system consists of the sensors (information channels) attached to the system via the sensor dedicated interface cards, the collection nodes, a network and the monitoring nodes. The data collection nodes read out information delivered by the sensors using the interface cards. Then, the data are selected and packed into the packets which in turn are sent to the monitoring nodes through the network. The monitoring nodes process the incoming data, performing all the requested operations, on the best effort basis. Eventually, a statistically significant picture of activity of all channels of the monitored system is built. The computing power needed to carry out such a task could be delivered in the form of classical CPUs or currently much more attractive, by the General Purpose GPUs (GPGPUs). Earlier attempts to evaluate the applicability of the classical CPU resources for the monitoring purposes showed that due to a large scale of the considered system some of the external resources located in the remote processing farms may prove to be necessary to perform the desired task [5]. Moreover, it has been demonstrated that the access to such remote farms could be granted within the framework of the interactive grid project [6] providing that the network throughput to the grid farms has enough capacity to handle the demanding traffic.

Even though the histogramming is a simple operation, in the case of a large number of sequentially analysed channels it requires high performance computing. Processing parallelisation at the input data level seems to strongly enhance the monitoring efficiency. The SIMD (Single Instruction Multiple Data) architectures perform the same instruction on the streams of data in a single cycle. Since the histogramming for each monitored channel requires updating of a content of a histogram bin then the use of SIMD architecture should lead to the increase of the histogramming speed by a factor reflecting the number of channels being monitored in parallel. The SIMD architectures, implemented as SIMT (Single Instruc-

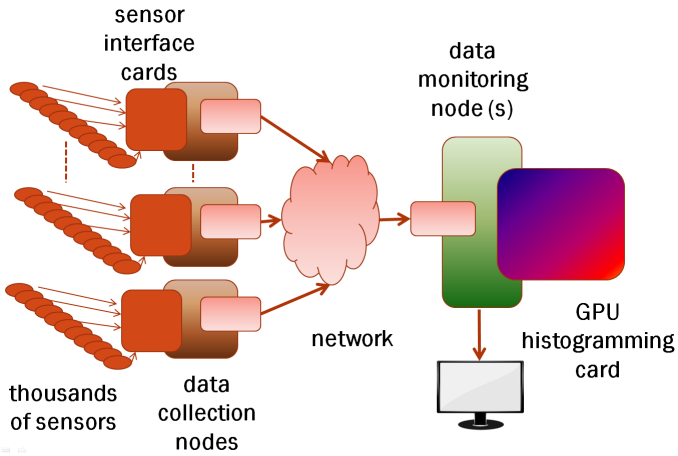


Figure 1. A sketch of a large scale data quality monitoring system

tion Multiple Threads) in the CUDA[1] context, are successfully used in graphics processors which aim is to perform large scale, complex calculations on large volume data samples. Therefore, it is expected that their application in the large, real-time data acquisition systems will lead to a substantial increase of the performance.

Histogramming with GPU cards was subject of a number of former research. Podlozhnyuk presents in [7] two implementations using the histograms of different size: one uses the 64-bin and the other one the 256-bin histograms. Both these implementations are included in the CUDA software development kit. In [8] Shams and Kennedy discuss the algorithm which can be used on large volume data-sets and for thousands of bins. Their methods target the data-mining applications, reading 32-bit values on input in contrast to Podlozhnyuk's 8-bit input values. Another CUDA implementation can be found in [9] where the computations of the original and a new histogram distribution is presented. Authors of [10] introduce two novel histogramming methods targeted at GPUs. Both methods outperform the existing ones and increase the performance predictability. The authors explore and discuss the algorithmic design choices for both methods, and they identify and evaluate performance limitations.

In this paper the performance increase of the monitoring system running on a cluster node equipped with GPU cards was estimated. The performed measurements and analyses addressed the following:

- what the performance gain of running the histogramming on a GPU comparing to the CPU is

[1] The acronym CUDA stands for Compute Unified Device Architecture.

- what computation model (memory model, etc.) is the most suited one for this kind of calculations
- what maximal data processing bandwidth can be achieved.

The stress was put on the experiments local to a monitoring node. This means that the data source was located in that node and that the performed measurements considered only the computing GPU/CPU and the memory data flow factors.

3 GPU ARCHITECTURE OVERVIEW

Architecture of a typical CUDA-capable GPU is organized into an array of highly threaded streaming multiprocessors (SMs). Each SM has a number of streaming processors (SPs) that share the control logic and instruction cache. Naturally, a general purpose GPU application includes both the CPU and the GPU codes. Serial code is executed in a host (CPU) thread and the parallel one, known as the kernel, is executed in many device (GPU) threads. Each kernel is executed on one device. Multiple kernels can be concurrently run on a device. The computation on a GPU is distributed onto a grid of blocks of threads. Each block contains the same number of threads and is identified within a grid by the two-dimensional block-ID. In turn, each thread within a block can be identified by its ID for an easy indexing of the data being processed. A block is organised as a three-dimensional array of threads. The block and grid dimensions, which are collectively known as the execution configuration, can be set at the run-time and are typically based on the size and dimensions of the data to be processed [11].

Each block is executed by one streaming multiprocessor. Several concurrent blocks can reside on a single SM depending on the blocks' memory requirements and the SM's memory resources. The blocks can be executed in any order, concurrently or sequentially. The threads within a block are grouped into warps. At any time a multi-processor executes a single warp. All threads (typically 32) of a warp execute the same instruction but operate on different data (in case of conditional statements all the threads in a warp execute all branches. However, those threads which do not follow the branch execute an equivalent of a null operation).

CUDA threads may access, during their execution, the data allocated in various memory spaces. Each thread has a private local memory. Each thread block has the shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There exist also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces.

The device's DRAM, the global memory, is un-cached. Access to the global memory has a high latency (of the order of 400–600 clock cycles), which makes the reading from and writing to the global memory particularly expensive. The throughput of the global memory access depends on the access pattern. When certain requirements are met by the threads in a warp, the access to the global memory by multiple threads can be combined into a single transaction for contiguous

memory locations. This is known as the memory coalescing. Non-coalesced memory accesses can severely affect the performance of an application and should be avoided where possible [12]. A coalescing global memory access is perhaps the single most important consideration in the CUDA code optimisation. It may even be worthwhile to reorganise the data prior to the execution of a kernel in order to ensure the coalesced access.

Local memory is called so not because of its physical location but because its scope is local to a thread. In fact, the local memory is off-chip. Local memory is used only to hold automatic variables. Access time to the local memory is comparable with that to the global one. The read-only texture memory is cached. Therefore, a texture fetch costs one device memory read only on a cache miss. The texture cache is optimized for 2D spatial locality, so the threads of the same warp that read the texture addresses that are close together will achieve the best performance.

The constant memory space is cached. As a result, a read operation from the constant memory costs one memory read from the device memory only on a cache miss. Accesses to different addresses by threads within a warp are serialized, thus the cost scales linearly with the number of unique addresses read by all the threads within a warp. Such a constant cache is the most efficient when the threads in the same warp access only a few distinct locations. If all the threads access the same location then the constant memory can be as fast as a register.

Shared memory is located on the chip. So, it has much higher bandwidth and lower latency than the local and global memory (provided there are no bank conflicts between the threads). Access time to the shared memory is comparable with the register access time [11]. To achieve a high memory bandwidth for concurrent accesses, the shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously. Therefore, any memory load or store of n addresses that span n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank. However, if the multiple addresses of a memory request map to the same memory bank then the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary. As a consequence the effective bandwidth is decreased by a factor equal to the number of separate memory requests. To minimise the bank conflicts, it is important to understand how the memory addresses are mapped onto the memory banks and how to optimally schedule the memory requests [12].

A typical CUDA implementation consists of the following stages [13]:

1. memory allocation on the device;
2. data transfer from the host to the device;
3. device memory initialisation if required;
4. execution configuration determination;
5. kernel(s) execution. The result is stored in the device memory;
6. data transfer from the device to the host.

One of the keys to a good performance is to keep the streaming multiprocessors on the device as busy as possible. The efficiency of the application can be improved if one minimises the data transfer between the host and the device. The data should be kept on the device as long as possible. Programs that run multiple kernels on the same data should favour leaving of the data on the device between the kernel calls rather than transferring intermediate results to the host and then sending them back to the device for subsequent calculations. Thus, the 5^{th} step outlined above can be run several times without a need to transfer the data between the device and the host.

The other way to improve the performance is to use the pinned_RAM (or page-locked) memory. The pinned_RAM memory has an important property: the operating system guarantees that it will never swap this memory out to a disk, which ensures its residence in the physical memory. Knowing the physical address of a buffer, the GPU can then use the direct memory access (DMA) to copy the data to or from the host [14]. However, the pinned_RAM memory should not be overused. Its excessive use can reduce the overall system performance because the pinned_RAM memory is a scarce resource. Furthermore, by reducing the amount of physical memory available to the operating system for paging, reserving too much pinned_RAM memory reduces the overall system performance [12].

The use of the pinned_RAM memory has several benefits [11]:

- for some devices the transfers between the pinned_RAM host memory and the device memory can be performed concurrently with kernel execution;
- on some devices, the pinned_RAM host memory can be mapped onto the address space of the device, thus eliminating the need to perform a copy operation to or from the device memory;
- on systems with a front-side bus, the bandwidth of the transfer between the host memory and the device memory is higher if the host memory is allocated as the pinned_RAM and even higher if, in addition, it is allocated as the write-combining.

4 EXPERIMENT AND RESULTS

To quantify the increase of histogramming performance a number of measurements recording the data processing rate or time in configurations aimed to explore possible advantages of locating raw data chunks and histograms within different types of memory available in a computer system were carried out. The raw data acquired from external sensors – typically some measuring devices, for example a set of the 8-bit ADCs' readouts – arrive to the computing node equipped with the graphics card via the network interface. Next, they are copied to the main computer memory (RAM) where they can be kept or alternatively can be further transferred to the graphics card.

The former option requires that the memory pages, which were allocated in order to store the input data cannot be swapped out and become inaccessible when

the threads running on the graphics card issue a request to read in these data. The advantages and disadvantages of pinned_RAM were previously discussed (see Section 3 for details).

The other option is to subsequently transfer the input data from RAM to the common global memory of the graphics card to which all the GPU running threads have the access.

A sketch of the memory configuration considered in the discussed tests is presented in Figure 2.

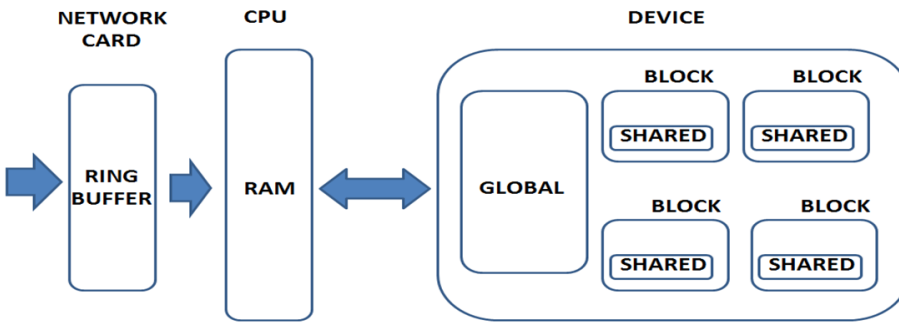


Figure 2. Memory banks involved in storing the input data and histograms

In the performed tests the data from each input channel had the values within the range of 0 to 255 thus fitting into a single byte. Moreover, it was assumed that all the input channels are sampled at the same moment and that the set of values obtained in this way forms an event. The data in the event are organised such that a value connected to a given channel has a constant address within an event. These addresses are consecutive. In other words every channel can be addressed using its offset defined with respect to the beginning of the event. The input data packet consists of a number of events with the same structure and with the same number of input channels.

For the above outlined data organisation the software architecture was designed to perform the histogramming task. Each channel present in the input data stream and undergoing the histogramming procedure was assigned a single thread. The number of active channels was stored in the packet header. During the header analysis it was decoded by the processing node to activate the required number of threads. The number of active channels was also sent to the threads. This allowed a proper, according to the input data size, configuration of the threads and unambiguous association of a thread and the monitored data channel. An additional assumption was that the histogram bin occupies four bytes of the memory (a 32 bit word), giving a possibility to store up to $2^{32} - 1$ occurrences of a measurement of a given value without transferring any intermediate results to the host memory. The above implies that 1 kB of memory is required to store a histogram consisting of 256 bins.

The histograms built during the input data analysis can be located either in the GPU global memory next to the input data or in the shared memory which is assigned to each block of threads created at the graphics card. As discussed earlier (see Section 3) the advantage of the latter solution manifests in the much shorter memory access time. Therefore, in order to avoid extensive cycle losses the histograms were stored in the GPU shared memory.

To further minimise the clock cycles loss (leading to increase of the execution time), now induced by an access of a thread to the shared memory, the resulting histograms were located using the rules required to profit from the bank organisation. Usually, there are 32 banks allowing the 32 threads forming a warp to access the memory in parallel (concurrently) if each thread accesses a location belonging to a different memory bank.

In the carried out tests the histogramming performance was measured for both cases:

- with the threads building histograms in separate memory banks (banked),
- when the banking structure was not used and the histograms were located in a continuous address space of the shared memory (non-banked).

As the testbed a computing node of the Zeus cluster owned by Academic Computer Centre CYFRONET AGH, Kraków, Poland was used. The Zeus cluster node is the HP ProLiant SL390 server equipped with two 6-cores Xeon E5645 CPUs (which sums up to 12-core CPU placed on a single motherboard) running at 2.4 GHz, the L2-cache size is 12 MB and there is 99 GB of RAM. In addition, the node houses also the Tesla M-class M2090 GPU Computing Modules. The graphics card contains eight devices running at 1.3 MHz, each having 5.6 GB of global memory. In each device there are sixteen multiprocessors, processing blocks of threads with 32 threads per warp and 48 kB of shared memory per block. The maximum number of threads per block is 1 024 and the maximum number of threads per multiprocessor is 1 536.

The requirement that each thread accesses the data from a different memory bank limits the number of histograms in the shared memory to 32 (Figure 3). Such organisation uses only 32 kB of shared memory leaving 16 kB for other purposes – such as storing information useful to determine basic statistical quantities. This space can also be used to keep the overflows and underflows for channels producing values from outside of the 0–255 range (possible only in the case of analysis of longer than a byte channel readings). Organisation with histograms in continuous memory allows the allocation of 48 histograms in 48 kB of shared memory.

In the first runs the time needed to histogram 100 events of input data as function of the number of the monitored channels was measured. Basic idea behind this measurement was the exploration of the banking organisation of the GPU shared memory and that of the way the CUDA scheduler submits large number of threads on limited hardware resources. Four tests were performed. During the tests four input data sets of randomly generated values were used. These sets were stored at

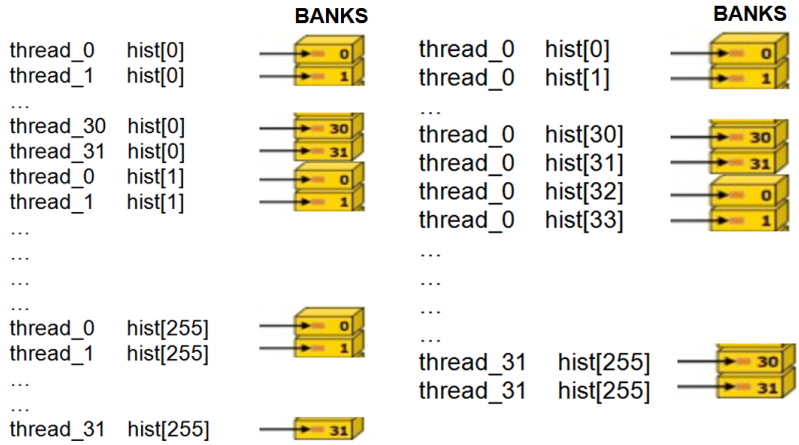


Figure 3. Layout of the histogram placement in the banking (left) and non-banking (right) configurations (see text)

the CPU RAM. Two of the generated sets were further modified in order to explore possible memory access conflicts in the case when the banking organisation of the shared memory was not used. In these selected data samples all odd numbered input channels in all 100 events were set to the same value – in the present case they were zeroed.

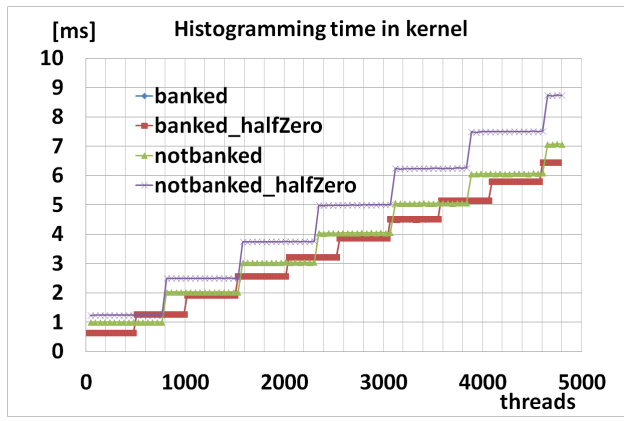


Figure 4. The step-like structure reflects operation of the CUDA scheduler assigning blocks of threads (32 threads for banked and 48 for notbanked cases) to 16 multiprocessors. The height of steps corresponds to processing time which increases when threads are serialised due to memory conflicts (notbanked) comparing to the lack of conflicts case (banked).

The measured histogramming times are presented in Figure 4. They were calculated using the number of clock ticks recorded by each thread during the kernel processing. The measured time includes the zeroing of the shared memory used to store the histograms, the histogramming of the monitored channels of 100 events and copying resulting histograms to the GPU global memory where the CPU program can access them and copy to the host computer RAM for further processing – for example for displaying of the obtained results.

The processing time increases with increasing number of threads (increasing number of the monitored data channels) for both types of the used memory configurations and four data sets. All curves exhibit a characteristic step-like shape. These steps reflect the way the CUDA scheduler assigns the threads for execution. The length of the steps reflects the internal organisation of the shared memory of the GPU card.

The scheduler, having 16 multiprocessors at its disposal, keeps assigning increasing number of blocks to free multiprocessors. When all the 16 multiprocessors will become busy with data processing the scheduler will queue any new request until at least one of the multiprocessors finishes its work and becomes idle. As all the multiprocessors run the same kernel they will finish at the same clock cycle. From this moment on the scheduler has again 16 multiprocessors at its disposal and another 16 requests can be assigned for processing at the same moment. Since in the case of the same kernel execution the processing time on multiprocessors is constant then the steps are of the same height.

The length of the steps reflects internal organization of shared memory for storing histograms. In the tests using the memory banking one multiprocessor is assigned one block containing 32 threads. Hence, 16 multiprocessors running in parallel can process $32 \cdot 16 = 512$ threads at the same time. When they finish another batch of 512 threads can start their execution. The processing time for the case with banking is the same for random data and for half data zeroed as when using the banking any memory conflicts are encountered.

The lack of the dependence of the histograming time on the data contents allows predicting the total processing time and retrieval of the results in a certain moment of time. This feature can be used in a large scale system to plan the load balancing and to avoid an extensive queuing which can grow proportionally to the processing time related to the contents of the processed data.

In the case of the continuous memory allocation the number of threads per block is set to 48 to use all the available space in the block shared memory. The scheduler can start 16 free multiprocessors processing in parallel $16 \cdot 48 = 768$ threads (what is reflected by *wider steps* – see Figure 4). On the other hand the processing time is longer (larger height of each step) when compared to the banked architecture case. This is a consequence of the presence of conflict situations observed for randomly generated data. In such a case more than one thread tries to modify the histogram bin located in the same bank – such requests have to be serialized. This is confirmed in the case when half of the data is set to zero inducing in such a way much pronounced influence of the conflicts on the measured processing time – see Figure 4.

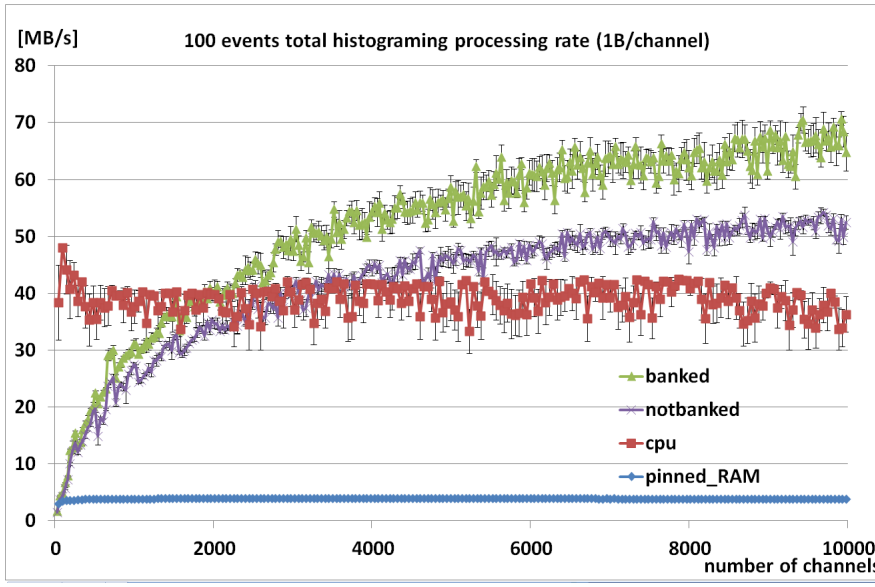


Figure 5. The processing rates for three cases of histograms located in the GPU memory (banked, notbanked, pinned_RAM) show a large memory transfer overhead for a small number of channels (small amount of data transfers). Increasing number of channels (the data size) improves the rates until the saturation sets in – see text. The initially constant CPU processing rate drops slightly for larger amount of the data sent for processing.

The comparison of the data processing rates for the three cases of the discussed memory configurations is presented in Figure 5. The rate is calculated based on the total time needed to send the data to the graphics card, to execute the histogramming kernel and to retrieve the resulting histograms from the device (GPU) global memory to RAM. For comparison, the results of the measurement performed for the histogramming task running on the CPU are also shown (marked with purple curve). In this case the test data were located in the computing node RAM next to the histograms.

The superiority of the CPU run histogramming process for a relatively small number of channels is related to the lack of data copying processes necessary when the GPU is involved. The initial low performance for the GPU cases increases with increasing number of the monitored channels and eventually tends to saturate. This saturation is observed because the data transfer overhead between the host node RAM and the GPU global memory becomes negligible when compared to the processing time of a large number of channels. Since in both, the banked and notbanked, cases the data transferred to the global GPU memory are used then one

may conclude from Figure 5 that the banking allows reaching considerably higher performances.

Indeed, the banked and notbanked configurations start to be more efficient than the CPU one if the number of channels exceeds about 2 000 and 3 500, respectively. A possibility to run in parallel 786 threads and that of an instant start of the processing of a new batch of events helps to cross the 45 MB/s boundary which seems to limit the CPU performance in the discussed case.

A stable but very poor performance seen for the pinned_RAM case (Figure 5) when the input data are located in the CPU RAM confirms earlier studies.

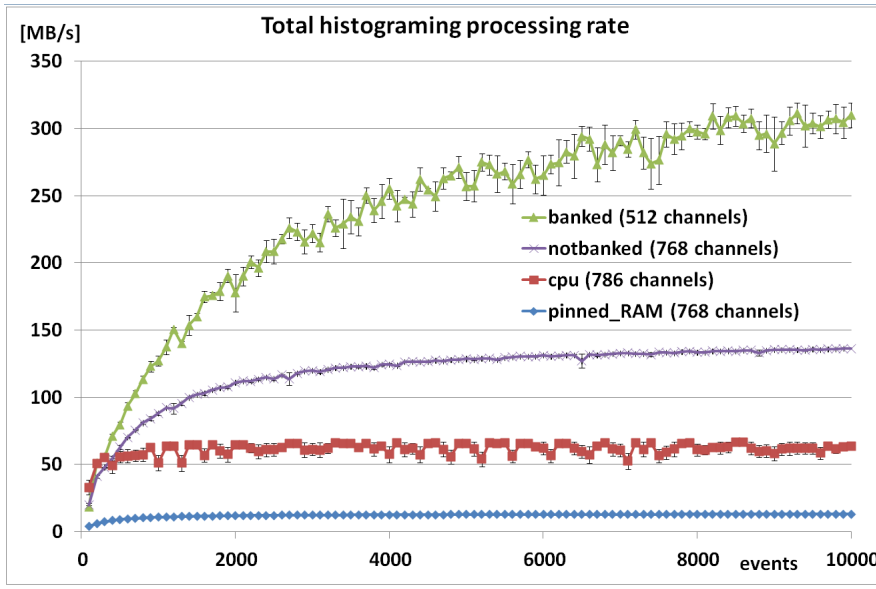


Figure 6. Increasing load on the GPU threads by sending larger number of events for processing helps to reach higher throughput. With the number of channels needed to activate 16 multiprocessors, the banked case doubles performance of the notbanked having only 2/3 of the number of channels for the notbanked case (for a large number of events).

The four placements of the histograms (memory configurations) were used in tests with variable number of events in the packet. Figure 6 shows results of the measurements. The observed trends agree qualitatively with the expectations. After an initial increase the processing rate tends to saturate with increasing number of events in a packet.

Increasing the number of events in a packet leads to a larger occupation of the GPU with histogramming operations. This reduces the relative contribution of the data transfers between the CPU and the device global memory as discussed above. Again, the poorest performance is measured in the pinned_RAM case. The his-

togramming task executed by the CPU improves when compared to the 100-events long packets (confront Figure 5). A large number of events demanding a long activity of the GPU processors allows the *banked* organisation to outperform other architectures. Its throughput almost doubles the one observed in the *not-banked* case where the memory access conflicts imply the limit of about 140 MB/s. The measurements were taken for the number of channels corresponding to full occupation of 16 multiprocessors of a single device.

5 SUMMARY AND OUTLOOK

Feasibility studies and performance testing of the histogramming were carried out using a single GPU device. The aim of our measurements was to identify differences and applicability of different memory organizations of the GPU for channel monitoring in a large scale systems. We did not attempt to find the best technology (like multi-core CPU, FPGA) for histogramming. The investigations were performed using different memory configurations (the histograms were allocated in various parts of the physical memory) as well as different 8 bit physics data scenarios. As a reference we compared our measurements to the results obtained in the case of running the same calculations on a CPU without any optimizations.

Earlier studies indicated that in case of manipulating large amounts of data and relatively simple GPU calculations the main bottleneck becomes the RAM to GPU memory bandwidth. Therefore our results, presented in this note, reflect total histogramming time which includes tranferring data from the CPU RAM to the global memory of the GPU, running the histogramming kernel, collecting separate histograms in the GPU global memory and their transfer to the CPU RAM (where applicable).

For different memory configurations all the obtained dependencies show a similar step-wise behaviour, which reflects the maximum number of threads that can be run on a single multi processor of the used GPU. It also turned out that the histogramming was the slowest in the so-called *notbanked_halfzero* case (confront Section 4 for details). For a relatively low number of the histogrammed channels the best results were obtained in the case where the operations were run on a CPU. If the number of monitored channels exceeded about 2 500 then the largest throughput was seen in the case of the *banked* memory configuration.

In case of the histograming performed by the CPU the low performance resulted from the data organization used in our tests. As the main focus of our studies was on large scale systems, the CPU memory was filled with blocks of measurements taken on a given sample for different channels. Therefore the caching mechanism of the CPU was not usable as the consecutive samples from the same channels were separated by thousands of bytes exceeding the size of the L1 cache. Analysis of adjacent samples from different channels required reloading corresponding histograms into L1 cache and the long access time to RAM degraded the CPU performance.

The processing rate as the function of a number of events in an input data packet showed characteristic behavior. After an initial increase the rate tends to flatten off with increasing number of events per packet. The best results were obtained in the *banked* memory configuration case.

Since the establishing of guidelines for building a universal architecture for the physics experiments data monitoring is our ultimate goal, we plan to perform further research in a number of areas.

First area concerns the input data scenarios. We plan to test processing of various data formats or various data types. Among them are values larger than 8 bits where we envisage more complex histogram structures with underflow and overflow entries.

A thorough benchmarking is the second area of our planned investigations. We expect to profit very much from it by identifying the bottlenecks and eventually delivering a proposal of a single computing node hardware architecture. We plan to test both the whole data flow and the processing chain on a processing unit consisting of a network device, multi-core CPU and the GPU device(s). We already anticipate that the network device with 1 Gbps throughput will be one of the main limiting factors. Therefore, we would like to test different network standards, among them 10 Gigabit Ethernet and others. Provided that it will be possible to provide much more data via the network device that GPU will be able to process, we would like to benchmark a hybrid GPU+CPU processing on the computing node. Such a scenario seems to be very promising from the cost-effectiveness perspective. A cheap node, consisting of a GPU and CPU, processing the data may perform equally well as a node equipped with a high-end GPU when its CPU is idle.

The clustering of the processing nodes is another field of our interest. Here, main stress will be put on the load balancing related issues. In addition to the performance issues of the cluster architectures, the problems connected to the cluster configuration as well as those related to the reconfiguration will serve as a subject of our future research.

Obtaining results in all the aforementioned areas will allow to formulate the architectural guidelines for building a network of the processing (monitoring) nodes. The guidelines should encompass the aspects of a single node architecture as well as those of the federation of such nodes and network infrastructure. Data processing requirements and its cost are the crucial factors to be considered.

Acknowledgments

We gratefully acknowledge support of the Academic Computer Centre CYFRONET AGH, Cracow, Poland. Calculations performed within this paper were carried out using the ZEUS cluster of CYFRONET. Access to the ZEUS infrastructure was granted thanks to the PLGrid PLUS project.

We thank most heartily Prof. P. Malecki and Prof. M. Turała for many stimulating and enlightening discussions.

REFERENCES

[1] ATLAS Collaboration—Aad, G. et al.: Observation of a New Particle in the Search for the Standard Model Higgs Boson with the ATLAS Detector at the LHC. Physics Letters B, Vol. 716, 2012, pp. 1–29.

[2] CMS Collaboration—Chatrchyan, S. et al.: Observation of a New Boson at a Mass of 125 GeV with the CMS Experiment at the LHC. Physics Letters B, Vol. 716, 2012, pp. 30–61.

[3] Olszewski, A.—Wolter, M.: A Needle in the Haystack: Higgs Boson Searches in the ATLAS Experiment. Computing and Informatics, Vol. 32, 2013, No. 6, pp. 1256–1271.

[4] Beck, H. P. et al.: The Base-Line DataFlow System of the ATLAS Trigger and DAQ. IEEE Transactions on Nuclear Science, Vol. 51, 2004, No. 3, pp. 470–475.

[5] Korcyl, K.—Szymocha, T.—Funika, W.—Kitowski, J.—Słota, R.—Balos, K.—Dutka, L.—Guzy, K.—Kryza, T.—Pieczykolan, J.: The ATLAS Experiment On-Line Monitoring and Filtering as an Example of Real-Time Application. Computer Science, Vol. 9, 2008, pp. 77–86.

[6] Funika, W.—Korcyl, K.—Pieczykolan, J.—Skital, L.—Balos, K.—Słota, R.—Guzy, K.—Dutka, L.—Kitowski, J.—Zielińskim, K.: Adapting a HEP Application for Running on the Grid. Computing and Informatics, Vol. 28, 2009, No. 3, pp. 353–367.

[7] Podlozhnyuk, V.: Histogram Calculation in CUDA. Technical Report. nVidia, 2007.

[8] Shams, R.—Kennedy, A.: Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices. Proceedings of International Conference on Signal Processing and Communications Systems (ICSPCS), Gold Coast, Australia, 2007.

[9] Yang, Z.—Zhu, Y.—Pu, Y.: Parallel Image Processing Based on CUDA. Proceedings of the International Conference on Computer Science and Software Engineering, IEEE Computer Society Washington, DC, USA, 2008.

[10] Nugteren, C.—van den Braak, G.-J.—Corporaal, H.—Mesman, B.: High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-Offs. Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4), Newport Beach, CA, USA, 2011.

[11] CUDA C Programming Guide 3.2. nVidia, 2010.

[12] CUDA C Best Practices Guide 3.2. nVidia, 2010.

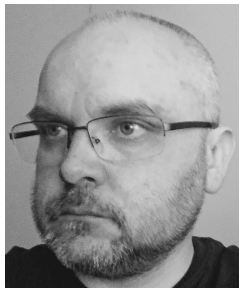
[13] Shams, R.—Sadeghi, P.—Kennedy, R.—Hartley, R.: Parallel Computation of Mutual Information on the GPU with Application to Real-Time Registration of 3D Medical Images. Computer Methods and Programs in Biomedicine, Elsevier, 2010.

[14] Sanders, J.—Kandrot, E.: CUDA by Example. Addison-Wesley, 2011.

Janusz Chwastowski graduated from the Department of Physics, Jagiellonian University in Cracow. His Ph.D. and habilitation were concluded in experimental particle physics. Since 1983 he has been working in the Institute of Nuclear Physics PAN, Cracow. He continues his research on particle physics as a member of experiments at DESY, Hamburg, Germany, and Brookhaven National Laboratory, USA, and CERN, Switzerland. He focuses on the experimental data and phenomenological analyses and the experimental apparatus, physics simulations and distributed computing. In 2006, he joined the Institute of Teleinformatics at the Cracow University of Technology where he is working in the Distributed Systems and Parallel Computing Group.

Krzysztof Korcyl is Adjunct in the Institute of Teleinformatics of Cracow University of Technology, Poland, and in the Institute of Nuclear Physics PAN, Cracow, Poland, where he received his habilitation in physics. He worked for the third level trigger of Delphi experiment at LEP and subsequently for TDAQ system of ATLAS experiment at LHC at CERN, Geneva. His research interests include modeling of large scale real time systems and applicability of FPGA and GPGPU technologies for improving performance in data acquisition and filtering systems.

Joanna Płażek received her M.Sc. (in 1984) and Ph.D. (in 2000) degrees in computer science from the Faculty of Electrical Engineering, Automatics, Computer Science and Engineering in Biomedicine, AGH University of Science and Technology in Cracow. She is Assistant Professor in the Institute of Teleinformatics of Cracow University of Technology. Her research interests include mathematical modeling and computer simulation of flows, distributed systems and parallel computing.



Piotr Poznański is Adjunct in the Institute of Teleinformatics of Cracow University of Technology, Poland. From 2000 to 2004, he worked at CERN, Switzerland as a member of the European Data Grid Project (EDG). He was a task leader and a member of the EDG Architecture Task Force. He is co-author of Quattor, a toolkit for managing large scale systems. In years 2005–2011, he worked for Motorola Kraków, Poland as a Senior Engineer designing solutions for radio trunking systems and leading groups of developers. He received his Ph.D. in computer science from AGH University of Science and Technology, Poland. His research interests include large systems and software engineering.