7 A Scalable Software Framework for Stateful Stream Data Processing on Multiple GPUs and Applications

Farhoosh Alghabi, Ulrich Schipper and Andreas Kolb

University of Siegen,

Institute for Vision and Graphics

Hölderlinstr. 3, 57076 Siegen, Germany

{farhoosh.alghabi,ulrich.schipper,andreas.kolb}@uni-siegen.de

Abstract During the past few years the increase of computational power has been realized using more processors with multiple cores and specific processing units like Graphics Processing Units (GPUs). Also, the introduction of programming languages such as CUDA and OpenCL makes it easy, even for non-graphics programmers, to exploit the computational power of massively parallel processors available in current GPUs. Although CUDA and OpenCL relieve programmers from considering many low-level details of parallel programming on multiple cores on a single GPU, the same support at a higher level of parallelization for multiple GPUs is still under research. In particular, fundamental issues of memory management and synchronization must be dealt with directly by the programmer. In this chapter we introduce concepts for CUDA-based frameworks which are designed for stateful stream data processing for graph-like arrangements of processing modules on two or more GPUs in a single compute node. We evaluate these concepts and further elaborate on the approach of our choice. Our approach relieves the programmer from error-prone chores of memory management and synchronization. The chapter presents detailed evaluation results which demonstrate the scalability of the proposed framework. To demonstrate the usability of our framework, we utilize it for demanding on-line processing in the areas of crystallographic structure detection and video decryption.

Keywords: GPGPU, Software Framework, Multi-GPU, Stream Data Processing

7.1 Introduction

Although the idea of parallel processing has been around for some decades, the interest to this field and its applications in various scientific and engineering areas has grown significantly in the past few years. There are two reasons that have played a major role in this growth. One reason is the advancements in hardware industry which have enabled processor manufacturers to put more processing cores on a single die, thereby moving the parallel programming from expensive mainframes and clusters to desktop computers. This fact is verified by noticing the widespread use of multi-core CPUs and many-core GPUs in almost any PC around the world. The second reason is the introduction of languages, libraries and tools that ease the task of parallel programming for these processors. Particularly, we can mention CUDA and OpenCL which both target GPUs and unleash the huge computational power even to programmers not familiar with computer graphics.

Both OpenCL and CUDA offer general-purpose application programmers with great support for parallel programming. This is accomplished by providing concepts and features that easily map to real-world problems which are parallel in nature, thus enabling efficient exploitation of computational power delivered by the numerous cores of a single GPU with minimal effort. Although these features work well for cases where there is only one GPU available in the compute node, they are not so easily extensible to the cases where multiple GPUs exist in a single node. Thus, it remains the task of programmer to take care of any details in order to provide the same degree of scalability at this new level of parallelization (***multi-GPU, single-node parallelization***) as the one available across the cores on a single GPU.

This chapter specifically addresses the concept and realization of a CUDA-based framework for multi-GPU, single-node parallelization problems, where GPU-scalability is a major concern. The framework has been designed with easy use by application programmers in mind. As a consequence, transparency is an important property of the proposed framework, mainly with regard to memory management and synchronization. Actually the most important programming task left to the application programmer is writing CUDA kernels responsible for processing of data as if they would run on a single-GPU node.

We assume the data is provided as sequences of homogeneous data sets $D^i(t_j)$ (*frames* at time t_j), where i indicates the last processing module M_i with which the data has been processed. We address a specific class of stream processing [1] problems, which can be characterized as follows (see also Fig. 7.1):

Module-Based Stream Processing: We assume data to be loaded to the framework via one or several *source modules* and to be processed by several *processing modules*. The resulting data is exported via *sink modules*.

A module M_i can be seen as a CUDA kernel which processes stream data, i.e. transforming input data $D^{i-1}(t_j)$ to output data $D^i(t_j)$ that is fed into subsequent module(s).

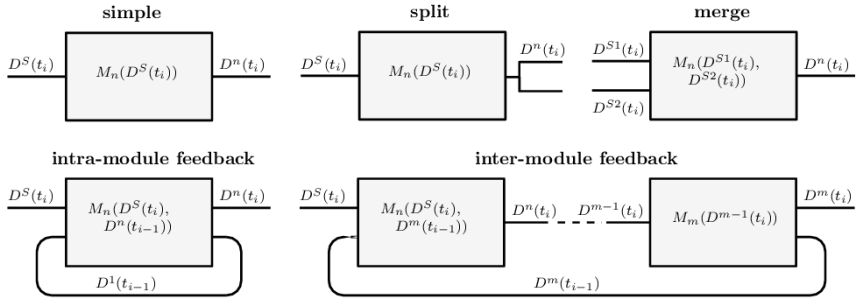

Fig. 7.1. Processing modules and their arrangement, including stream splits (top-middle), merges (top-right), intra-module feedback (bottom-left) and the optional inter-module feedback (bottom right)

Graph-Based Layout: The stream data is transferred between modules, which can be arranged like a graph, including stream splits and stream merges.

Stateful Processing of Data: It means that previous data or processing results are required for processing newly arrived data. This is realized using *intra-module feedback*; here, the processing in module M_i of frame t_j also depends on the prior result of *the same* module, i.e. on $D^i(t_{j-1})$.

Inter-Module Feedback (Optional): Inter-module feedback improves on the intra-module feedback by letting two distinct modules be connected via feedback.

As a result, the addressed problem class is more general than standard pipeline-processing and thus it has a wider range of applications. Stateless problems, nevertheless, can still be subject to automatic multi-GPU processing.

In order to give an impression of how useful our framework is, we briefly discuss two applications from different domains here. One application lies within the scope of information security. To protect against unauthorized access to information, various cryptographic and steganographic algorithms have been developed. Not surprisingly, videos form an important subclass of data which are required to be protected against unauthorized access. The rapid growth in size of videos (due to increasing resolution, colour depth, frame rate, ...), however, has made the task of applying complex methods computationally quite expensive. One scenario shows the on-line application of cryptographic and steganographic methods.

The second application lies within the field of crystallography. One common practice in the community is to study the structure of crystals by examining a sample using x-ray imaging. Here, a crystal sample is radiated by an x-ray beam and the scattered radiation pattern is detected by an energy-dispersive pn-type charge coupled device (pnCCD) sensor. This camera generates images with 384 * 384 pixels and 2 bytes of information per pixel at currently 400 frames per second, yielding an overall data rate of beyond 112 MB/s. The overall goal of these kinds

of experiments is to have near real-time data analysis in order to be able to directly detect improper adjustments of the setup or wrong experimental parameters. Furthermore, in the near future, these experimental setups should be applied for continuous analysis of large sample sets. In a separate section, we show that how successfully our framework is used to address this problem.

The remainder of this chapter is organized as follows: Sec. 7.2 gives an overview of works done in the area of multi-GPU as well as stream data processing. Sec. 7.3 describes the parallelization concepts and implementation details for the framework. Sec. 7.4 presents some experimental evaluations. In sec. 7.5, as mentioned, two real-world applications where our framework has been utilized are elaborated. Finally, Sec. 7.6 concludes the chapter with a brief discussion.

7.2 Related Work

As stated in the introduction we focus on multi-GPU, single-node parallelization for stream data processing. Consequently, in the following we first mention works mainly characterized by running on multi-GPU systems and then those which deal with stream data processing.

In [2] Enmyren and Kessler propose a skeleton programming library for systems with multiple CPU cores and GPUs. This is accomplished by use of CUDA and OpenCL as the backends for code running on GPU and OpenMP for CPU code. The operations supported by their library follow MapReduce model and are in the form of a C++ template library. [3] proposes an approach for high-performance scientific computing on single- and multi-GPU systems. An important feature of the prototype implemented in the paper is the separation of algorithm description from mapping to the hardware which is achieved through the definition of a domain-specific language. The language is defined in close collaboration with experts of the domain for which the framework is intended. In [4] Chen et al. propose a task-based queue scheme for systems with one or multiple GPUs. The main goal of the scheme is dynamic load balancing which is achieved by breaking down the computations into fine-grained tasks and then dynamically assigning them to GPUs. Note that in the case of single-GPU systems this reduces to assignment of tasks to CUDA cores available on a GPU which is reported to outperform the CUDA scheduler in case of unbalanced workload. Chen et al. further develop on this work to support GPUs on different nodes in a cluster [5]. They also improve their scheme for dynamic load balancing on individual nodes with multiple GPUs. As an interesting application, Stuart et al. [6] have proposed a multi-GPU design for volume rendering. In their implementation, parallel volume rendering has been fit into MapReduce model and run on a cluster of nodes equipped with GPUs. As a rather innovative work [7] presents a performance prediction model for multi-GPU systems, which gives an estimate of the expected

performance improvement when moving from a single-GPU to a multi-GPU system, based on the performance results on a single-GPU system.

In summary, all of the above-mentioned approaches either focus on a problem domain that does not include the problem domain addressed in this chapter, or they use a different hardware setup, e.g. CPU-clusters, for which the concepts can not be directly applied to our hardware setup.

Considering related works mainly characterized by stream data processing, [8] presents a framework for processing of multiple data streams on heterogeneous systems where both CPUs and GPUs are used as processors. The paper proposes a method for assignment of streams to CPUs and GPUs such that hard real-time constraints of stream data processing are satisfied. Yamagiwa et al. [9] elaborate on their efforts for porting an already existing framework for stream data processing on single GPU from previous GPU generations to present ones. To this end, they use CUDA. This, in addition to the use of OpenGL and DirectX for GPUs of old generations, leads to the development of a framework capable of running on different generations of GPUs. Teodoro et al. [10] introduce a stream data processing framework capable of exploiting the computational power of both CPUs and GPUs. A significant point with their framework is a mechanism for determining on which type of processor (CPU or GPU) the processing should be done (provided that the code for both types of processors are given). The framework uses CUDA as computational backend on GPUs. Houzet et al. [11] present a programming model which can be used for stream data processing on multi-GPU systems. The innovation of this work is its use of system design language SystemC which is used as a high-level language for description of the desired processing, thereby hiding many low level details from users. Zhang and Mueller propose a scalable stream data processing framework which runs on GPU clusters and is based on CUDA [12]. It makes extensive use of template-based generic programming techniques in C++ to offer programmability and uses MPI for inter-node communication. As the last work in this section, Vogelgesang et al. [13] have developed a GPU-based image processing framework which supports CPU usage as well. Similar to [10] their framework chooses between CPU and GPU codes provided that both codes exist. The framework supports processing on a cluster of nodes and uses OpneCL as computational backend.

All of the mentioned stream data processing approaches lack support for either multi-GPU or the problem domain addressed in this chapter (i.e. stateful stream data processing).

7.3 The Framework

In this section, we first describe possible parallelization concepts for the addressed problem domain (Sec. 7.3.1). The evaluation of these concepts in Sec. 7.4.1 forms the basis for final implementation, which is described in Sec. 7.3.3.

Remember that our framework assumes that all or majority of processing is done on GPUs, thus a processing module can be safely considered as a user-defined CUDA kernel in most cases. The *processing graph* is a collection of modules which describe the flowchart of processing done on data, including stream source and stream sink modules (see Fig. 7.1)

7.3.1 Basic Concepts

An important design question while developing the framework is how to distribute the computational load over several GPUs and, as a consequence, how the synchronization and the data management is organized.

Since in our treatment of the framework the computational load is decomposed into modules, this question reduces well to that of how to assign different modules to GPUs. We consider two completely different approaches, i.e.

Distributed Graph: In this first concept, the processing graph is divided into *N* sub-graphs, where *N* is the number of GPUs, and modules within each sub-graph are strictly assigned to a separate GPU.

Multiple Graph Instantiation: In this concept, on the other hand, one instance of each module or more precisely one instance of the whole processing graph runs on each GPU.

Tab. 7.1 summarizes their main characteristics. Note that there are two variants of the Distributed Graph approach (see Sec. 7.3.2).

Table 7.1. Characteristics of the different concepts

	Multi-threaded Distributed Graph	Single-threaded Distributed Graph	Multiple Graph Instantiation
Architecture	- One Instance - Modules Distributed over GPUs - One CUDA Stream per Module	- One Instance - Modules distributed Over GPUs - Two CUDA Streams per GPU	- Multiple Instances - One Instance per GPU - One or More CUDA Stream(s) per GPU Instance
Synchronization	CPU-thread Synchronization	CUDA Stream Barrier	CPU-thread Synchronization
Memory Transfers	Source, Sink, GPU-borders	Source, Sink, GPU-borders	Source, Sink, Feedback
Load Distribution	Module Distribution	Module Distribution	Built-in
Inter-module Feedback	Not Supported	Container Modules	Main Memory

In order to select one of the concepts for final implementation, we have implemented preliminary versions of both concepts. These preliminary versions are fully functional in terms of data management, synchronization and process-

control. Based on the preliminary implementation, the performance of the concepts has been evaluated (see Sec. 7.4.1). The essence of the evaluation is, that the Multiple Graph Instantiation approach outperforms the two variants of Distributed Graph in almost all test cases except when the number of intra-module feedbacks is large enough. Thus, we made the choice to fully implement the Multiple Graph Instantiation approach. Consequently, the technical description of the Distributed Graph concept is less detailed than the one for Multiple Graph Instantiation.

7.3.2 Distributed Graph

There are two variants of the Distributed Graph approach. The major difference between these two variants is the number of CPU threads used for controlling the modules, which strongly influences the synchronization method to be applied. In multi-threaded variant each module is controlled by a separate CPU thread (see Fig. 7.2). The module stores its result in a small output ring-buffer. If a module is idle, it polls the output buffer of the predecessor for new data to process. If this is the case, new data is copied to an input buffer (DeviceToDevice copy) and processed. If no new data is available, it yields its time slice. If a module has a successor, that is located on a different GPU, the output ring-buffer is mirrored to the host memory (DeviceToHost copy). On the other hand, if a module has a predecessor that resides on a different GPU, it copies the data from main memory to its GPU memory (HostToDevice copy). The modules are synchronized via the access to the output ring-buffer. In single-threaded variant all modules are controlled by the same CPU thread (see Fig. 7.3), which calls all CUDA functions (kernel launches and memory transfers) asynchronously. Two CUDA streams are used for each GPU, one for data transfer and the other for kernel calls, thus partially hiding data transfer time by overlapping kernel launch and memory transfer. Before the next frame is processed, the CUDA streams are synchronized by a barrier.

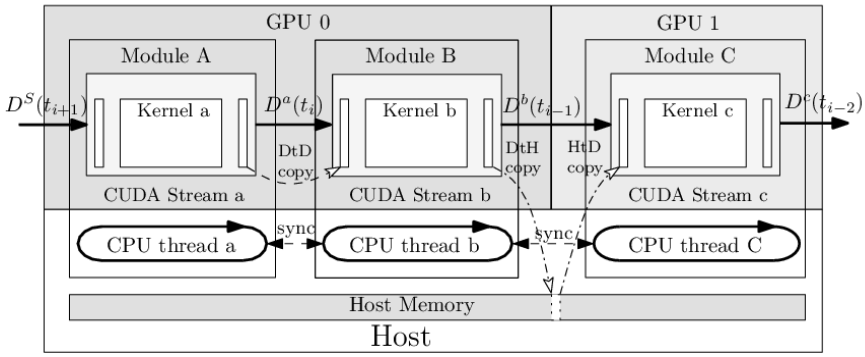

Fig. 7.2. Distributed Graph, Multi-threaded Variant: Each module runs in a separate CPU thread. The processing is synchronized via access to the ring-buffer. Data transfers across GPU borders are managed via the main memory.

This approach requires a manual decomposition of the complete processing graph into *N* sub-graphs to be distributed to the *N* GPUs. The load distribution is a direct result of this decomposition and thus a difficult task left to the user.

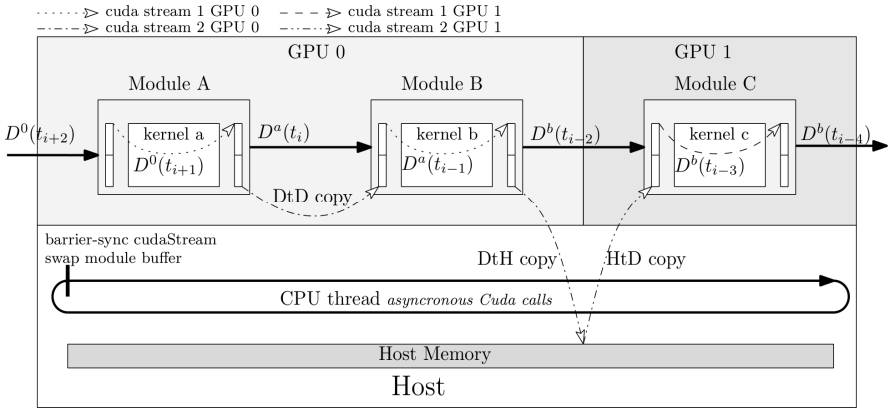

Fig. 7.3. Distributed Graph, Single-threaded Variant: In this concept, all modules are triggered within a single CPU thread using asynchronous CUDA calls. Two CUDA streams for each GPU are used to partially hide data transfer time. A CUDA stream barrier is used to synchronize after each process iteration.

7.3.3 Multiple Graph Instantiation

At the very heart of the proposed framework lies a simple idea: processing all the input stream(s) data at a specific time step t_i by a single GPU (see Fig. 7.4). Precisely speaking, for $N > 1$ GPUs, numbered from 0 to N - 1, the data from all input streams at time step $t \geq 0$ is processed by GPU $t \bmod N$. This has an immediate consequence of nearly perfect load distribution over GPUs in case of data-independent processing.

Although the basic idea behind the proposed framework is quite simple, there are still a few other considerations which affect the framework design in a significant way. The two most important considerations are synchronization and main memory management which are largely influenced by the stateful processing requirement of the framework, i.e. the realization of the intra- and inter-module feedback functionalities. For the Multiple Graph Instantiation approach, feedback data is transferred first from the memory of one GPU to the main memory of the system and then from there to the memory of another GPU. This leads to two memory transfer operations between host and device with additional synchronization requirements, whereas in the Distributed Graph concept this data remains on the same GPU.

Besides the two aforementioned considerations, there are still a few less important ones which are specifically taken care of to exploit useful features of GPUs offered by CUDA. Notably, GPU memory management and concurrent CUDA kernel launches and memory copies are among these. These last two points together with synchronization and main memory management are separately considered in the following four subsections.

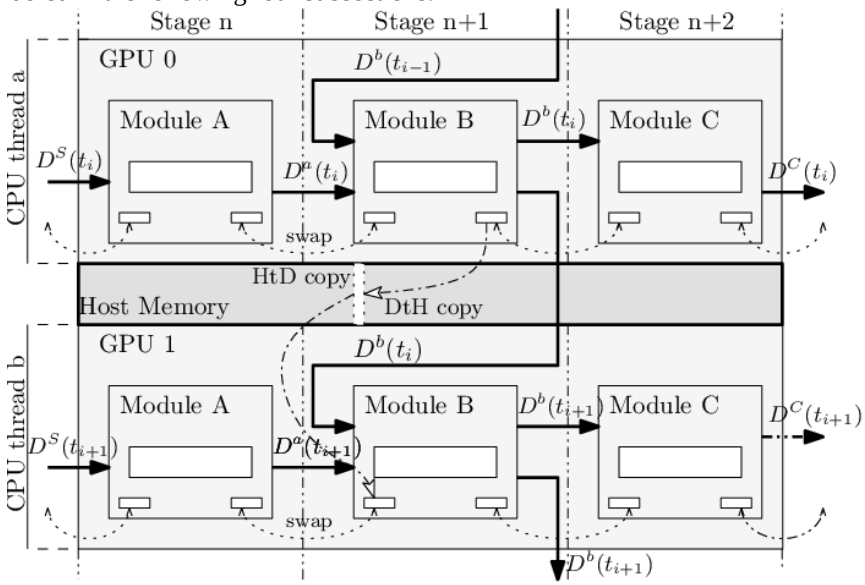

Fig. 7.4. Multiple Graph Instantiation: The whole processing graph is executed on each GPU (here, only 2 GPUs are shown). Data transfers for inter- and intra-module feedbacks are handled via main memory. The input and output buffers are swapped during stage changes to save GPU memory (see Sec. 7.3.3).

7.3.3.1 Synchronization

Considering the basic idea of the framework, there should be a mechanism which ensures us that the GPUs both read the inputs from sources and write the outputs into the sinks in correct order. In order to realize this behaviour, the framework launches as many CPU threads as GPUs where each CPU thread is in full charge of a GPU. This, in turn, lets the framework control the order of accesses to input as well as output streams by different GPUs through the use of synchronization objects defined at CPU thread level. The same mechanism is used to let each GPU access the processing results of input(s) at previous time step, thereby enabling the stateful processing property of the framework.

7.3.3.2 Main Memory Management

Main memory can be regarded as the major gateway of the framework for communication with the outside world. Actually, it is the place where inputs represented by sources are read from by GPUs and also it is the place where outputs represented by sinks are written into by GPUs. In addition to these two functionalities, the main memory also serves another important purpose: providing a place for exchange of data between GPUs. This latter point combined with previously-mentioned synchronization mechanism which is used to synchronize accesses to common main memory areas between two GPUs, realize the stateful processing capability of the framework.

7.3.3.3 GPU Memory Management

Although a straightforward way for GPU memory management is to allocate memory for inputs and outputs of all modules in the processing graph, the framework employs another strategy for this. The motivation for this has been better utilization of precious GPU memory. To implement this strategy, the framework introduces the concept of stage. A stage is defined as composed of modules whose inputs are produced in previous stage(s). Such a definition is a recursive one and the only requirement is to define the first stage. To complete our definition, the first stage is considered to be composed of only sources.

Now that we have organized all the modules in the processing graph into stages, GPU memory management can be described as allocation of two separate areas on GPU memory. From one of the GPU memory areas the inputs for all modules in the current stage are read and into the other the outputs of all modules of the current stage are written. The roles of the two GPU memory areas are swapped when finishing current stage and starting a new one. This way the output area of current stage becomes the input area of the new stage, thus ensuring the desired behaviour. This swap process is repeated whenever a stage is complete and a new one begins. Note that this GPU memory management strategy is done for each GPU separately and the two GPU memory areas are allocated on global memory of GPUs. This latter point ensures the data are persistent between two consecutive stages.

7.3.3.4 Concurrent Kernel Launches and Memory Copies

A useful concept introduced in CUDA is that of CUDA streams. An immediate consequence of this concept is the possibility of concurrent kernel launches as well as concurrent kernel execution and copies between main and GPU memories. With the aim of increasing performance, the framework is designed to exploit this

valuable feature as well. For this purpose, the framework provides the user with some CUDA streams on which to launch kernels.

7.4 Experimental Evaluation

In accordance with how the effort for development of the framework is divided into two main phases (see Sec. 7.3), the evaluations carried out are well categorized into two major groups, i.e. those aimed at the selection of a concept for final implementation (Sec. 7.4.1) and those to depict the scalability of the final implementation (Sec. 7.4.2). Note that the system used for running all the experiments in this chapter is equipped with 4 Tesla C2050 GPUs each having 448 CUDA cores and connected via a separate PCI-Express 2.0 x16 interface. The system also has two Intel Xeon E5630 2.53 GHz Quad-Core CPUs with 24GB of RAM. Finally the system runs Windows Server 2008 R2 as the operating system.

7.4.1 Comparison of Preliminary Implementations

The evaluation of the preliminary implementations is based on three different processing graphs. The stream data for all experiments consists of 10.000 data frames of 384 * 384 2-byte data elements, adding up to some 2.75 GB. Furthermore, we vary the amount of computation performed in each module. Therefore, we use two different CUDA kernels, one *light kernel*, inducing relatively little computational effort, and one *heavy kernel* with high computational costs. Then the average time measurement is reported. As the last point, in Distributed Graph experiments the distribution of modules among GPUs is done manually in order to get the best load balance for each processing graph.

The first processing graph examined is a *serial processing graph*, in which the processing modules are connected sequentially and their number varies from 1 to 10. Fig. 7.5 shows the result for this experiment. This experiment is ideal for parallelization, since the least amount of data transfer is required, i.e. no feedback, splitting or merging. The Multiple Graph Instantiation completely outperforms the two variants of Distributed Graph in both light and heavy kernels. There is, however, an interesting observation: for the heavy kernel, the Multiple Graph Instantiation implementation performs almost linear, whereas this is almost constant in light kernel version. This effect is due to the fact that the computation done in heavy kernel is large enough to constitute most of the measured time whereas in light kernel version other operations such as host (CPU) to device (GPU) and device to host memory transfer times dominate the computation time in kernels, leading to an almost constant performance. Note that these two types of memory

transfer operations are performed exactly the same number of times regardless of the number of processing modules in the serial processing graph.

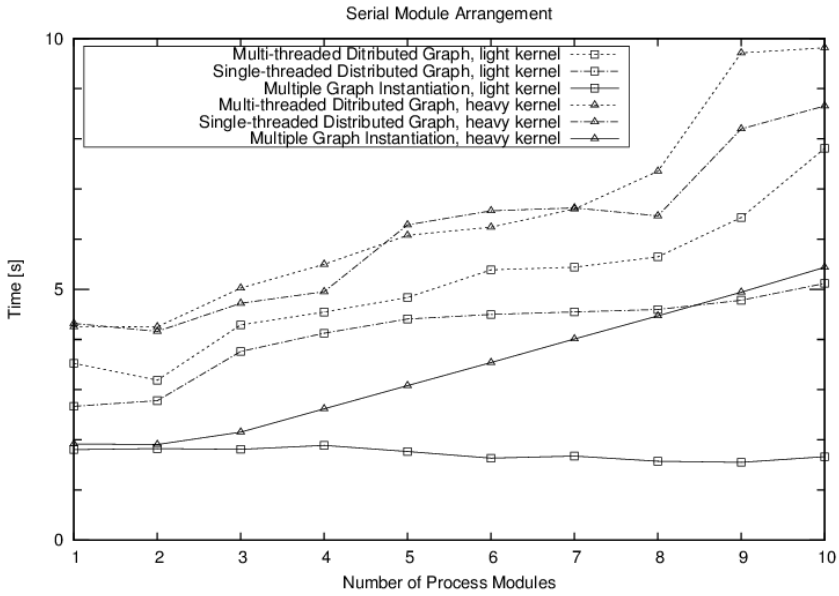

Fig. 7.5. Serial processing graph experiment performed with 1-10 processing modules consisting of either light or heavy kernels for all three concepts

The next experiment is conducted using a *parallel processing graph*, where the processing modules are arranged in a purely parallel fashion and their count varies between 1 and 10. The results of experiments are shown in Fig. 7.6. The Multiple Graph Instantiation concept again outperforms the two variants of Distributed Graph. Once again, the same effect as the one in Fig. 7.5 can be seen for light and heavy kernel modules used in the Multiple Graph Instantiation. This can well be explained by the same line of reasoning as the one stated for serial processing graph.

In the last processing graph, we use a more complex arrangement consisting of 23 processing modules (see Fig. 7.7). In this processing graph some of the processing modules have an intra-module feedback the number of which ranges between 0 and 23. As can be seen in Fig. 7.8, the Multiple Graph Instantiation performs better than multi-threaded Distributed Graph. However, for a large number of intra-module feedback, the single-threaded Distributed Graph outperforms the Multiple Graph Instantiation. This effect is a direct result from the data transfer required for feedback, i.e. the intra-module feedback implementation in the Multiple Graph Instantiation is more expensive than its Single-threaded Distributed Graph counterpart (see Sec. 7.3.3).

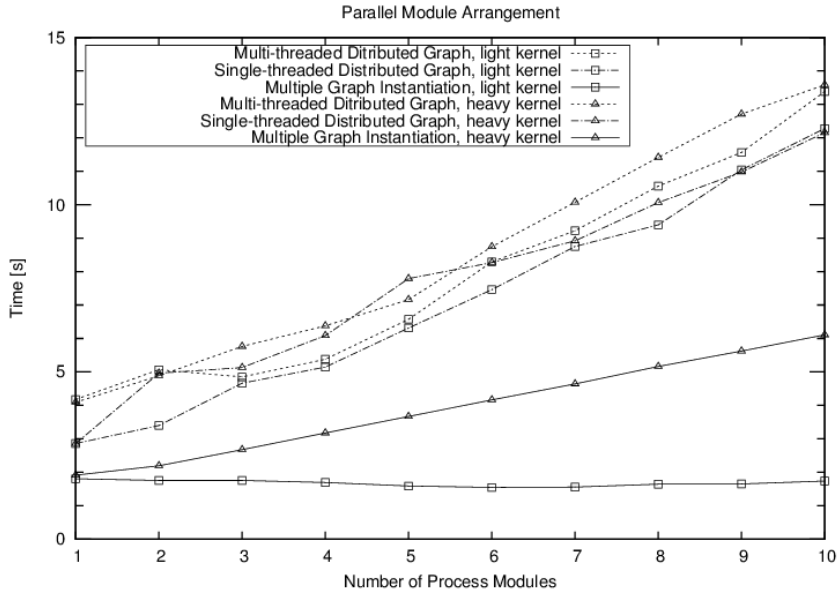

Fig. 7.6. Parallel processing graph experiment performed with 1-10 modules consisting of either light or heavy kernels for all three concepts

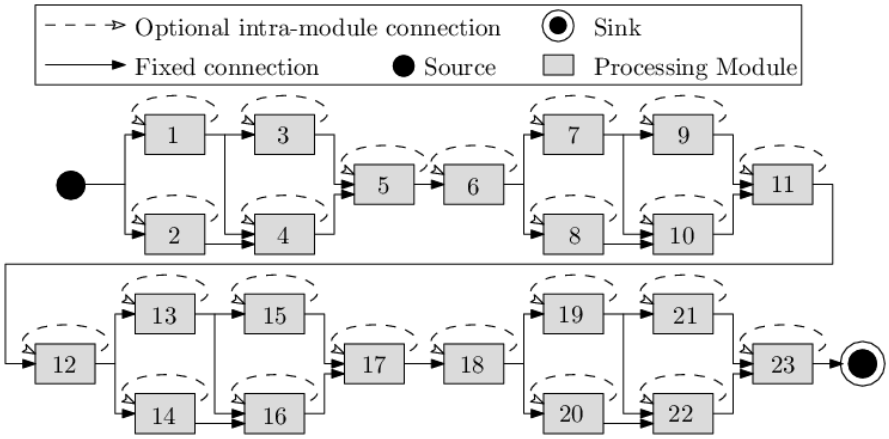

Fig. 7.7. Complex processing graph used in Sec. 7.4.1

7.4.2 Scalability and Feedback

The Multiple Graph Instantiation approach by default supports the optional functionality of inter-module feedback. To be precise, the implementation does not make any difference between intra- and inter-module feedbacks. We conducted some experiments regarding this feature in order to evaluate the effect of feedbacks on the scalability in terms of the number of GPUs. Therefore, we generated a processing graph, consisting of a linear sequence of modules with an additional inter-module feedback (see Fig. 7.9). For the evaluation we vary the computational load of modules bridged by the feedback and the ones outside the bridged subgraph.

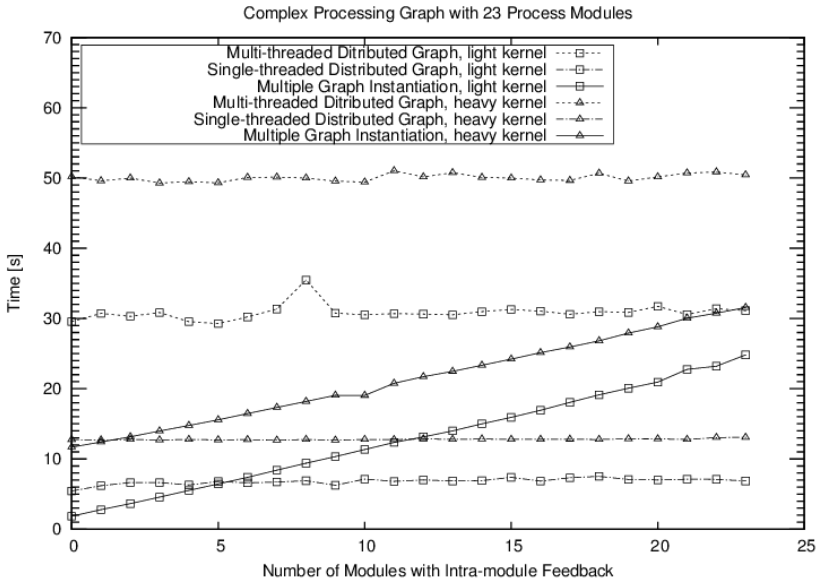

Fig. 7.8. Complex processing graph experiment performed with 0-23 intra-module feedback(s) using either light or heavy kernels for all three concepts

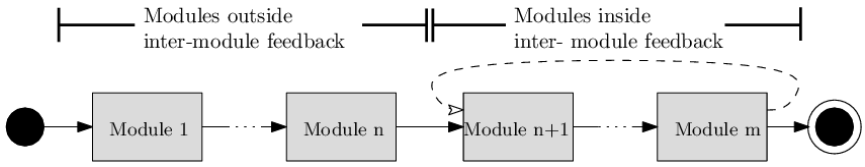

Fig. 7.9. Processing graph used for examining the effect of feedback on scalability (see Sec. 7.4.2)

The results regarding the scalability are shown in Fig. 7.10. As expected, inter-module feedback reduces the performance of our framework. Naturally, bridging the whole graph completely, i.e. having no computational load outside the bridged

sub-graph, completely destroys the GPU parallelism, since the first processing module can process $D^0(t_{j+1})$ only after the last module N has generated its result $D^N(t_j)$. The rate of performance degradation is related to the proportion of the time spent within the feedback sub-graph and that spent outside the bridged sub-graph.

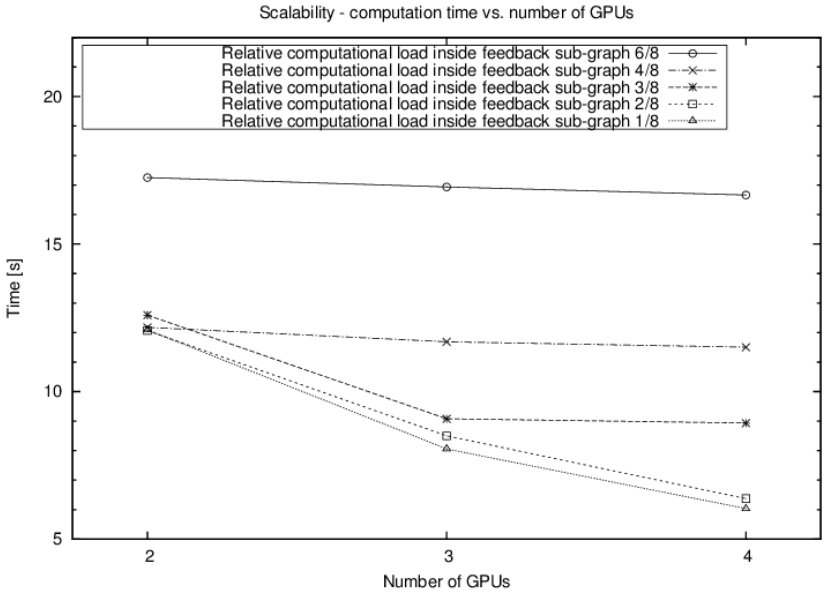

Fig. 7.10. Scalability and feedback in terms of the proportion of the computational load inside and outside the feedback sub-graph

7.5 Applications

In this section we present two different applications which have been successfully addressed by our framework. The first application deals with information security whereas the second one is in the field of crystallography.

7.5.1 Information Security using Crypto- and Steganography

Cryptography and steganography form two major groups of methods within the scope of information security. While cryptography is more concerned with hiding the content of a message, steganographic methods try to hide the message itself. To better clarify the difference between the two, one can consider the case of a simple piece of meaningful text communicated between sender and receiver. In

case of cryptography, one would encrypt the meaningful text such that each letter is replaced by another thus leading to an unmeaningful text. In steganography, however, the meaningful text (referred to as cover or cover text) could be written in such a way that the secret message is formed from the first letter of each word. As can be seen in this simple case, the advantage of steganography over cryptography is that it doesn't attract the attention of those who accidentally access the text, whereas the encrypted text would raise suspicion that there is a secret message hidden in the unmeaningful text. Therefore, cryptographic methods only protect the content of a secret message while steganography deals with protection of both secret message and communicating parties.

In this section our framework is exploited to deal with an application where both cryptographic and steganographic methods are involved. The goal is to extract a sequence of secret hidden images from an encrypted cover video. The video is encrypted based on method of [14]. In our implementation it is assumed that each video frame in the memory is divided into chunks of 8 bytes and corresponding chunks in consecutive frames form a separate sequence of plaintext blocks. Furthermore, in each video frame a secret image is hidden using least significant bit which is a steganographic transform whereby secret information are written into least significant bits of image pixels thus causing hard-to-perceive degradations in visual quality of cover image (The interested reader is referred to [15] for a survey of this and other image steganograohic methods). Based on these assumptions, our framework first decrypts a video frame by applying method of [14] to obtain the cover image and then extracts the hidden image by applying the reverse steganographic transform to the cover image (All processing for this experiment is done on GPUs). Note that in this implementation the decryption result of each video frame is affected by that of previous one thus requiring feedback as shown in Fig. 7.11. Note that H and μ are decryption parameters as defined in [14] and the task of two modules H image and decrypted frame extraction is to separate these two pieces of data which are combined at the output of frame decryption module and provide them on their outputs.

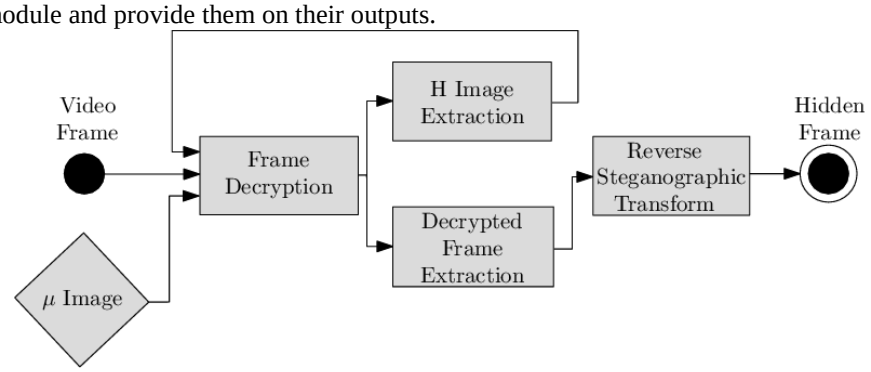

Fig. 7.11. Processing graph used to extract hidden image sequence (video) from encrypted cover video

The experiment is done using HD Videos of size 1920 * 1080, 24 bpp as cover video. The result would be a video (image sequence) of size 1920 * 135, 24 bpp. This is because from each byte in the input video only the least significant bit is preserved thus reducing the size to one eighth. The timing results for 2, 3 and 4 GPUs are shown in Fig. 7.12. Also shown in the figure are timing results for CPU implementation of the same algorithm using 1, 2, 3 and 4 CPU threads to provide the reader with a ground to compare with. Considering the typical frame rate of 1080p HD videos which is between 24 and 60 frames per second, one can easily see that the 4-threaded CPU implementation can only handle frame rates near the lower bound of this range whereas the two-GPU implementation supports frame rates well beyond its upper bound.

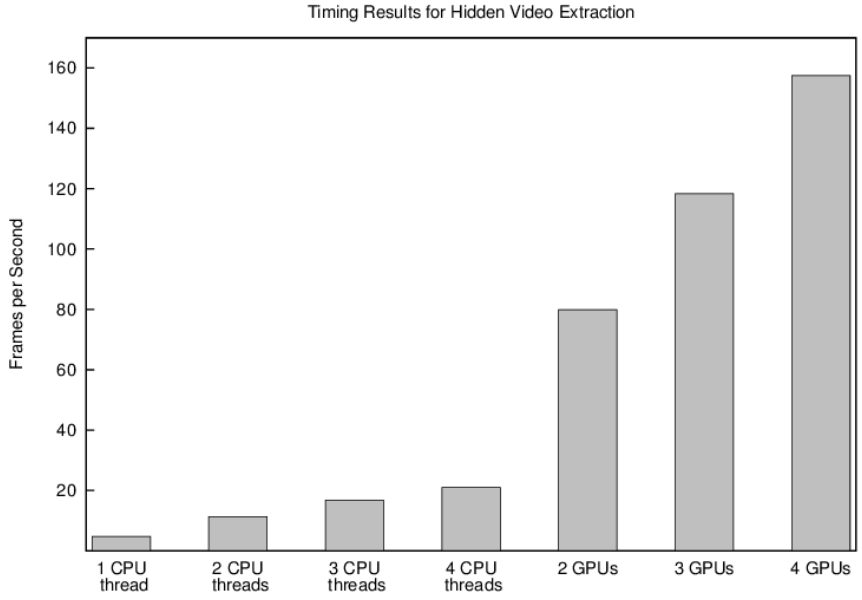

Fig. 7.12. Timing results of hidden video extraction from encrypted HD video using different number of CPU threads and GPUs

7.5.2 Crystallography using a pnCCD Camera

Considerable amounts of information about crystals are collected through examination by x-ray. There are different types of x-ray sensors which record the result of these examinations. One such sensor is an energy-dispersive CCD with fast read-out called pnCCD camera (see Sec. 7.1). The specifications of this camera were mentioned in introduction. Getting familiar with the operation of the camera,

however, needs some basic knowledge of the domain. When x-ray beam is scattered by crystal sample, scattered x-ray photons hit the camera image plane. Depending on the position of incident photons onto the image plane, a number of pixels are illuminated thus producing non-zero pixel values. Pixels illuminated by a single photon are collectively called an event. Events can be consisting of 1, 2, 3 or 4 non-zero pixels (the so-called single, double, triple and quadruple events, respectively). Fig. 7.13 shows valid patterns for double, triple and quadruple events. However, it may happen that in an image we have invalid patterns. These patterns are caused by two or more photons whose event patterns interfere and make a cluster of events. A solution to this problem is to increase the frame rate such that the probability of occurrence of interfering patterns decreases. That is why, pnCCDs support such high frame rates as 400 frames per second. Determining valid events in each frame forms the basis for many other crystallographic experiments which rely on analysis of events.

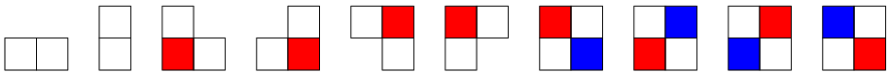

Fig. 7.13. Valid double, triple and quadruple events: Red and blue pixels show the highest and lowest pixel values in an event, respectively.

We have developed kernels for extraction of valid events from pnCCD frames [16] which is based on [17]. The whole processing can be split into two major steps of frame correction and valid event extraction. As Fig. 7.14 shows, first an offset map is subtracted pixelwise from the raw pnCCD frame. During common mode correction the median value for each row of the image is computed and then subtracted from all pixel values of the corresponding row. The processing continues by 'Zero' pixel elimination whereby all pixels whose values are less than corresponding pixel values in a noise map image multiplied by a constant factor are discarded. In gain correction the pixel values in each column are multiplied by a gain factor. In CTE correction for each column the pixel values are multiplied by a CTE factor raised to the power of the pixel's row index. Now, we have corrected frames which are then used to extract valid single, double, triple and quadruple events. Fig. 7.15 shows the performance and scalability of our framework while working with different number of GPUs (2 to 4) and different frame sizes (Note that all processing modules in the processing graph run on GPU). To better show the usefulness of GPUs for event extraction, we have implemented a single-threaded CPU version of the mentioned algorithm. The CPU version processes 92 frames of size 384 * 384 per second whereas this number is 1756 when 2 GPUs are used thus leaving a lot of computational power for further processing of events (Note that event extraction is only a first processing step in many crystallographic applications).

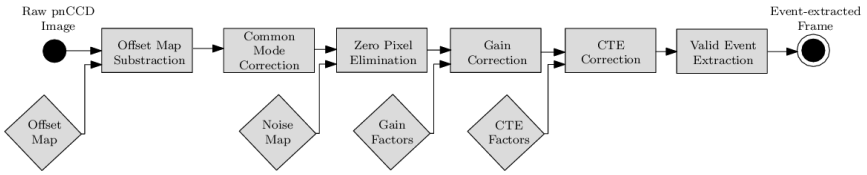

Fig. 7.14. Processing done on each raw pnCCD image to extract valid single, double, triple and quadruple events

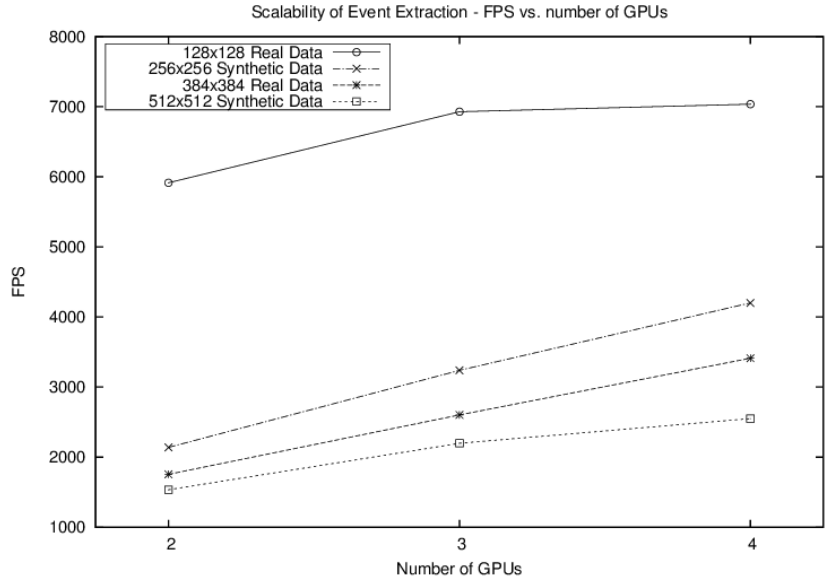

Fig. 7.15. Timing results of valid event extraction for various number of GPUs and frame sizes

7.6 Conclusion

In this chapter, we presented a scalable CUDA-based framework for stateful stream data processing on multiple GPUs in a single node. As described, the framework is designed to be both easy to use and flexible from the user part. The ease of use is achieved by transparent implementation of the framework with regard to synchronization and memory management. This, however, does not limit the flexibility of the framework in the sense that the user still has unlimited freedom to define the CUDA kernels for processing modules as desired.

Still the most important feature of the framework is scalability. For that, the chapter also presents a number of experiments for stateful processing of stream data and examines the effect of feedback in processing graphs on the scalability of

the framework with regard to GPUs. Furthermore, the practicality and usefulness of the framework for real-world tasks is demonstrated by two different application scenarios.

Acknowledgments This research was partially funded by the German Ministry for Research and Education (BMBF) under grant No. 05k10PSB.

References

[1] Macedonia, M.: The GPU enters computing's mainstream. IEEE Computer 36(10), 106-108 (2003)

[2] Enmyren, J., Kessler, C.: Skepu: A multi-backend skeleton programming library for multi-GPU systems. In: Proc. Int. ACM Workshop High-level parallel programming and applications. pp. 5-14 (2010)

[3] Meyer, B., Plessl, C., Forstner, J.: Transformation of scientific algorithms to parallel computing code: Single GPU and mpi multi GPU backends with subdomain support. In: Proc. Symp. Application Accelerators in High-Performance Computing (SAAHPC). pp. 60-63 (2011)

[4] Chen, L., Villa, O., Krishnamoorthy, S., Gao, G.: Dynamic load balancing on single- and multi-GPU systems. In: Proc. Parallel & Distributed Processing (IPDPS) (2010), dOI: 10.1109/IPDPS.2010.5470413

[5] Chen, L., Villa, O., Gao, G.: Exploring fine-grained task-based execution on multi-GPU systems. In: Proc. IEEE Int. Conf. on Cluster Computing. pp. 386-394 (2011)

[6] Stuart, J.A., Chen, C.K., Ma, K.L., Owens, J.D.: Multi-GPU volume rendering using MapReduce. In: Proc. Int. ACM Symp. High Performance Distributed Computing. pp. 841-848 (2010)

[7] Schaa, D., Kaeli, D.: Exploring the multiple-GPU design space. In: Proc. Int. IEEE Symp. Parallel and Distributed Processing (2009)

[8] Verner, U., Schuster, A., Silberstein, M.: Processing data streams with hard real-time constraints on heterogeneous systems. In: Proc. International Conference on Supercomputing. pp. 120-129 (2011)

[9] Yamagiwa, S., Arai, M., Wada, K.: Efficient handling of stream buffers in GPU stream-based computing platform. In: Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing. pp. 286-291 (2011)

[10] Teodoro, G., Sachetto, R., Sertel, O., Gurcan, M., Meira, W., Catalyurek, U., Ferreira, R.: Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In: Proc. Int. IEEE Conf. on Cluster (2009)

[11] Houzet, D., Huet, S., Rahman, A.: Syscellc: A data-flow programming model on multi-GPU. In: Proc. Int. Conf. on Computational Science. pp. 1035-1044 (2010)

[12] Zhang, Y., Mueller, F.: Gstream: A general-purpose data streaming framework on GPU clusters. In: Proc. Int. Conf. on Parallel Processing. pp. 245-254, (2011)

[13] Vogelgesang, M., Chilingaryan, S., dos Santos Rolo, T., Kopmann, A.: Ufo: A scalable GPU-based image processing framework for on-line monitoring. Proc. IEEE 14th Int. Conf. on High Performance Computing and Communications pp. 824-829 (2012)

[14] Wang, X., Bao, X.: A novel block cryptosystem based on the coupled chaotic map lattice. Nonlinear Dynamics 72, 707-715 (2013)

[15] Cheddad, A., Condell, J., Curran, K., Kevitt, P.M.: Digital image steganography: Survey and analysis of current methods. Signal Processing 90, 727-752 (2010)

[16] Alghabi, F., Schipper, U., Kolb, A.: Real-time processing of pnCCD images using GPUs. In: 14th Int. Workshop on Radiation Imaging Detectors (2012)

[17] Andritschke, R., Hartner, G., Hartmann, R., Meidinger, N., Strüder, L.: Data analysis for characterizing pnCCDs. In Proc. of Nuclear Science Symposium, pp. 2166-2172 (2008)