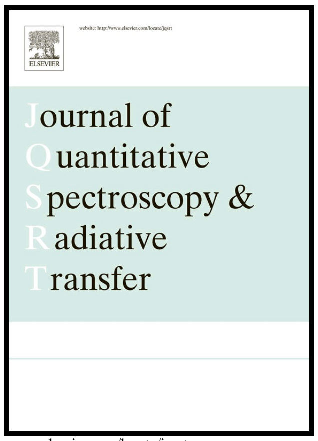


Fast data preprocessing with Graphics Processing Units for inverse problem solving in light-scattering measurements

G. Derkachov, T. Jakubczyk, D. Jakubczyk, J. Archer, M. Woźniak

PII: S0022-4073(16)30427-7
DOI: http://dx.doi.org/10.1016/j.jqsrt.2017.01.008
Reference: JQSRT5555

To appear in: *Journal of Quantitative Spectroscopy and Radiative Transfer*

Received date: 19 July 2016
Revised date: 5 January 2017
Accepted date: 6 January 2017

Fast data preprocessing with Graphics Processing Units for inverse problem solving in light-scattering measurements.

G. Derkachov[a], T. Jakubczyk[b], D. Jakubczyk[a,*], J. Archer[a], M. Woźniak[a]

[a]*Institute of Physics, Polish Academy of Sciences, Aleja Lotników 32/46, PL-02668 Warsaw, Poland*
[b]*Institute of Control and Computation Engineering, Warsaw University of Technology, ul. Nowowiejska 15/19, PL-00665 Warsaw, Poland*

Abstract

Utilising Compute Unified Device Architecture (CUDA) platform for Graphics Processing Units (GPUs) enables significant reduction of computation time at a moderate cost, by means of parallel computing. In the paper [Jakubczyk et al., *Opto-Electron. Rev.*, 2016] we reported using GPU for Mie scattering inverse problem solving (up to 800-fold speed-up). Here we report the development of two subroutines utilising GPU at data preprocessing stages for the inversion procedure: (i) A subroutine, based on ray tracing, for finding spherical aberration correction function. (ii) A subroutine performing the conversion of an image to a 1D distribution of light intensity versus azimuth angle (i.e. scattering diagram), fed from a movie-reading CPU subroutine running in parallel. All subroutines are incorporated in PikeReader application, which we make available on GitHub repository. PikeReader returns a sequence of intensity distributions versus a common azimuth angle vector, corresponding to the recorded movie. We obtained

[*]jakub@ifpan.edu.pl

an overall $\sim$ 400-fold speed-up of calculations at data preprocessing stages using CUDA codes running on GPU in comparison to single thread MATLAB-only code running on CPU.

Keywords: optical particle characterisation, numerical aberration correction, image processing, Compute Unified Device Architecture (CUDA), Graphics Processing Unit (GPU)

---

1. Introduction

Interferometric techniques, which are used for optical particle characterisation require solving an inverse problem ([1]). Most methodological efforts are aimed at performing the inversion procedure with high accuracy and in short time (see e.g.: [2, 3, 4, 5, 6, 7]). In the work [8] we described a successful implementation of inversion algorithm (Mie Scattering Lookup Table Method) on graphics processing units (GPUs; see e.g. [9, 10]), making use of parallel computing (see e.g. [11, 12]).

In this work we describe GPU implementation of algorithms for **data preprocessing**, a stage scarcely described in literature. Since the amount of scattering data requiring processing at this stage may easily become huge, it is a common practice to pre-reduce it either by using a linear camera or by selecting a narrow region of interest in the field of view. In case of imperfect imaging, this may slightly compromise the final inversion accuracy. Since our further investigations depend on high particle characterisation accuracy, we try to make the most of the data in the field of view, which results in extensive computations.

We perform data preprocessing in two steps: (i) finding the aberration correction function, common for all recorded images, as well as the droplet position versus the centre of the trap, and (ii) conversion of all images in a movie to a sequence of 1D distributions of light intensity versus a common 700-element azimuth angle vector (i.e. scattering diagrams). The second step includes pixel decoding, demosaicing and aberration correction, as well as pixel sorting versus azimuth angle, walking average of light intensity and reduction to 700 points. Both subroutines use GPU capabilities, while the image-conversion subroutine is fed from a movie-reading subroutine running in parallel on CPU. The overall speed-up of calculations was $\sim$ 400-fold in comparison to single-thread MATLAB code. The application comprising both subroutines is called PikeReader and can be freely downloaded from GitHub repository [13].

2. Motivation and design of the optical system for observation of Mie scattering images

Our primary research interest focuses on evaporation dynamics of single, free, micrometre-sized droplets. Single droplets ranging from $\sim$ 25 μm to $\sim$ 500 nm in radius can be levitated in our electrodynamic trap (see e.g. [14]). In order to infer the details of the thermodynamic process (see e.g. [15, 16, 17]) the fine details of the temporal evolution of droplet radius are analysed. As a primary tool for measuring the droplet radius, we use elastic (static) light scattering with Mie Scattering Lookup Table Method [14]. For droplets of pure liquids of few micrometres radius we reach an accuracy of $\pm$10 nm.

The observation of light scattered by a micrometre-sized droplet, confined in a centimetre sized trap (figure 1: top), enclosed in a decimetre sized chamber (figure 1: middle) requires a dedicated optical system. Since a wide-angle observation is highly preferable in view of the inversion procedure accuracy, a non-paraxial optical system is required (see figure 2). Usually, minimising aberrations in such a system requires a complex lens configuration or/and aspheric lenses. Available good quality objectives are comparatively large, which, in turn, scales up the whole setup and the chamber in particular. To avoid this and having in mind temperature and atmosphere composition gradient issues associated with larger vessels, we decided to use two simple lenses (figures 1 and 2) and to correct aberrations at the post-processing stage (see next section). The focal point of the entrance lens coincides with the droplet and there is a confocal aperture after the exit lens. The polarisers are half-way between the lenses. Since they contribute to the optical distance between the lenses, it is accounted for in the calculations.

Previously, in 4 side-port trap/chamber system we used symmetrical bi-convex lenses of 18.705 mm curvature radius and 12.6 mm diameter. The distance between apexes of the lenses was 42.57 mm. There was an additional (calibrating) circular aperture between the viewing port of the trap and the lens. The body of the objective was made of plexiglass with 16 mm entrance outer diameter.

The newly developed setup has 8 optical side-ports (see figure 1), which constricts the lens system even further. In order to retain a wide field of view, we got rid of the circular entrance aperture and put the entrance lens closer to the droplet (compare figure 2). This, however, resulted in larger optical aberrations. To minimise this, plano-convex lenses with apexes pointing inward were used. The distance between apexes was 37.4 mm. The lenses were anti-reflection coated and their curvature radius and diameter were 10.3 mm and 12.4 mm respectively. The body of the objective was made of aluminum, which allowed reduction of the entrance outer diameter to 13 mm.

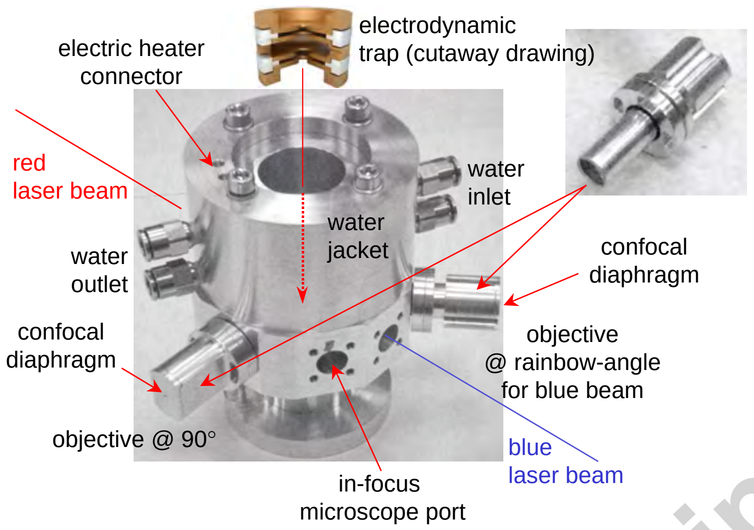


Figure 1: Middle: partially assembled new version of climatic chamber with main optical system in place. Top: cutaway drawing of the electrodynamic trap. The location of the trap in the chamber indicated with (red) solid-and-dashed arrow. Top-right corner: an assembled optical system for Mie scattering imaging.

3. Aberration correction in post-processing

It is well known that the information present in an image suffering aberrations is not lost. It can fully be retrieved in numerical post-processing, as long as the parameters of the lens system are known. In our experiments we record the temporal evolution of scattering patterns with aberration in the form of movies. The aberration correction procedure consists of two main steps: (i) finding the angle and intensity correction with analytical or numerical ray tracing, and (ii) applying the correction to each image in a movie. The intensity distribution generated by the transmitted rays is proportional to the spatial transmittance function of the lens system. Simultaneously, it provides a pixel-to-scattering angle mapping. Then, the raw scattering images must be divided by this distribution and each CCD pixel must be assigned the scattering angle value.

In our previous works, in order to assign the correct scattering angle value (both: azimuth and elevation) to each pixel, analytical ray tracing formulas were used. To simplify the problem, a circular entrance aperture was introduced between the trap port and the first lens. Then, under the assumption of axial symmetry of the droplet-lens system, the formulas involved only simple planar trigonometry. Neither the modification of light intensity distribution due to aberration was accounted for, nor Fresnel coefficients were applied at ray refractions. However, since the scattered light, being a modulated spherical wave, was observed with a flat sensor, the cosine law and the inverse-square law of illumination were applied for each sensor pixel. The factor of $(\cos\theta)/r^2$ multiplying the corrected image intensity was introduced, where θ and r are respectively the incidence angle and the length of the ray from the droplet to the pixel. Since, according to Mie theory, the scattered light intensity changes insignificantly for the elevation angles used ($\pm 5^\circ$), all the sensor pixels can be ordered versus azimuth angle assigned to them only. Accordingly, the procedure yielded the azimuth angle correction and a sufficient intensity correction function. These were applied to each scattering image, yielding scattering diagrams for inversion procedure. In order to correct the chromatic aberration, the procedure was repeated for both used light wavelengths. The eventual displacement of the droplet was accounted once for the whole movie in the inversion procedure.

In the presented work, we aimed for two major improvements of the aberration correction algorithm: (i) the relaxation of the axial symmetry assumption to account for a possible off-centre position of the droplet, as well as for the limitations of the trap/chamber mechanical accuracy and (ii) accounting not only for the angular distortion but also for the light intensity modification intrinsic to spherical aberration. Thus, assigning the true scattering angle value and intensity correction to each pixel required an exact ray tracing in 3D (figure 2). Here we outline the basic steps of the algorithm.

3.1. Finding of aberration correction function

3.1.1. The ray tracing algorithm concept

Since in our experiment the size of the droplet is negligible in comparison to the size of the lens, the droplet can be considered as a point light source placed at a given point. The origin of a coordinate system is set at the centre of the trap and the droplet may be located off-centre. Rays are generated from the droplet position evenly in the sense of the sphere surface density, over the azimuth and elevation angle. The angular ranges are slightly larger than the ones defined by the viewing port of the trap. Ray tracing is performed in a standard manner and rays missing the lenses or blocked by the viewing port or the confocal aperture are eliminated. For each refracting surface we find the point of intersection of the ray with the lens surface by solving a line-refracting surface set of equations. Then we apply Snell's law in vector form to find the refracted ray direction. For each ray, with unitary initial intensity, a unique azimuth-elevation angle pair is attributed. The azimuth and elevation is averaged over all rays hitting a pixel. The rays hitting each pixel are counted and their final intensities are summed. The number of generated rays is much larger than the number of pixels in order to achieve smooth intensity distribution. The re-distribution of transmitted light intensity is accounted for in two ways: (i) The intensity of each ray at each refracting surface is modified by the $(\cos\theta)/r^2$ factor, without considering Fresnel coefficients. Since the lenses were anti-reflection coated ($\sim$ 1% reflection), this simple method was found quite sufficient. (ii) An average intensity is calculated for each pixel. It depends on both the modified intensity of each ray as well as on the number of rays hitting the pixel, which is modified due to aberration. Depending on the CCD sensor type, some

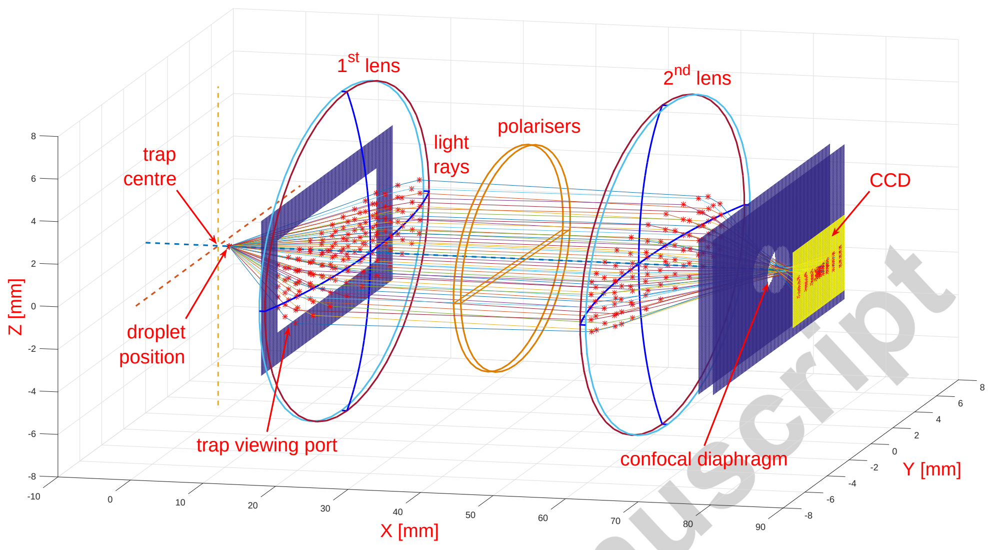


Figure 2: Non-paraxial ray tracing through the optical elements of the Mie scattering imaging system. The x-axis scale is ~ 10 times compressed versus y- and z-axis. In consequence, the curvature of the viewing port is not visible. The CCD cover glass is not shown.

additional steps in the ray tracing algorithm may have to be implemented. The CCD cover glass does not modify the angular light distribution and can usually be neglected. However, the angular dependence of CCD quantum efficiency must be considered.

3.1.2. Finding the position of droplet and detector

In order to find the aberration correction function, the real position of the droplet versus the trap centre and the position of the CCD sensor must be found first. The true position of the droplet is also necessary for further data preprocessing. The scattered light missing the viewing port is blocked and the shadow defined by the port edges is casted onto the sensor. A theoretical projection of the edges onto the CCD, obtained with ray tracing, is added to the displayed image and used as a viewfinder (see figure 3). The real position of the droplet and the CCD can be found by adjusting their position in the ray tracing subroutine and fitting the viewfinder to the real shadow. Summing (all) frames, blurs the scattering pattern and raises the visibility of the shadow. Additionally there is a possibility of projecting virtual interference fringes and comparing them with the fringes in freely selected frames. The ray tracing does not consider the diffraction at the viewing port edges, however the diffraction is usually plainly seen for summed frames and the expected position of the edges can be well estimated. In order to draw the viewfinder, less than 80 rays are necessary. After changing the ray tracing parameters, it can be redrawn instantly. Significantly more time (several minutes) is consumed by manual droplet and CCD position adjustment.

3.1.3. Speed and accuracy

The new aberration correction subroutine yielded very promising results. It was flexible in the sense of the lens system and apart from finding the aberration correction function it allowed determining the droplet position already at the data preprocessing stage. Unfortunately, when coded in MATLAB, it turned out to be very time consuming ($\sim$ 2 hours for 9×10^6 rays). However, it can easily be noticed that ray tracing is ideal for parallel computing, as different calculation threads hardly interact. Based on a previous work [8], we decided to make use of GPU and rewrote the subroutine accordingly on CUDA platform. The code was also generally optimised in the process. The project turned out very successful by making the correction of the optical aberrations nearly instantaneous ($<$ 0.3 s for

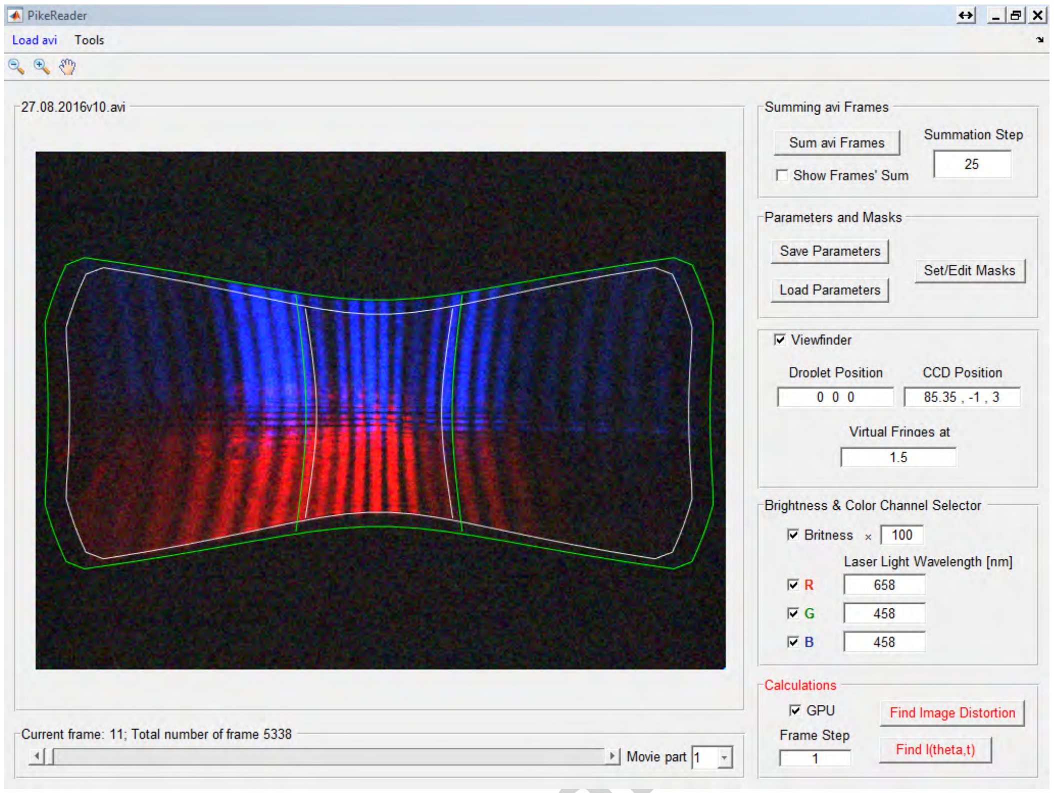
9×10^6 rays @ GTX 580 and < 0.8 s for 625×10^6 rays @ GTX Titan Black).

The accuracy of new data preprocessing subroutines can hardly be estimated directly, since the previous subroutines corresponded to different hardware: trap, optical system and camera. However, it could be estimated on the basis of achieved accuracies of droplet radius found with the same inversion subroutine. Since there was no visible gain in the accuracy of radius finding (± 10 nm for a several micron-sized droplet [14]), we conclude that the accuracy of the new preprocessing subroutines compared to the previous one is the same. The new subroutines running on GPU use single precision floating point arithmetics. However, the accuracy can be increased by using double precision at the expense of speed.

3.1.4. Implementation of ray tracing algorithm in CUDA

The ray tracing algorithm was implemented as a MEX subroutine, called `RayTracingCUDA` (see `RayTracingCUDA` subroutine in figure 4). MEX subroutine is generally versatile and easy for using with our data processing software written in MATLAB (figure 4). In PikeReader application it is invoked with `Find Image Distortion` button. However, we would like to underline that the subroutine is self-standing. Ray tracing is an indispensable tool for designing optical systems and for computer generated graphics. The structure of the subroutine is as follows:

- Ray tracing data is loaded from MATLAB memory into GPU memory.
- The ray tracing subroutine, also called - the ray tracing kernel, described above, is executed on GPU in massively multiple threads (512 for GTX 580). When one set of threads is executed, the next set is preloaded. Each thread has a number defining a single ray. For best performance, and in order to



27.08.2016v10.avi

Summation Step: 25

☐ Show Frames' Sum

☑ Viewfinder

Droplet Position: 0 0 0

CCD Position: 85.35 , -1 , 3

Virtual Fringes at: 1.5

☑ Britness × 100

Laser Light Wavelength [nm]

☑ R	658
☑ G	458
☑ B	458

☑ GPU

Frame Step: 1

Current frame: 11; Total number of frame 5338

Movie part: 1

Figure 3: The Graphics User Interface of PikeReader application. Raw, uncorrected image is displayed. The buttons invoking `Find I(Theta,t)` and `Find Image Distortion` subroutines are emphasised with red lettering.

avoid collisions, different threads shouldn't write to the same memory address.

- In order to find the values of cells (pixels) of the intensity-correction array, adding values from different threads is necessary. The result of unsynchronised additions from simultaneous threads into a variable at a single address may be indeterministic. To avoid indeterministic results, the so called atomic addition is used. The atomic function is a special subroutine which guarantees undivided access to a variable at a specific address. In our case, due to the nature of the problem, the synchronisation operation isn't very costly, since only few threads need to access the same cell of the image array.

- Finally, the results are copied back from GPU memory into MATLAB memory.

3.2. Movie processing

Movie processing is done frame by frame. For each frame, first the (averaged) background for each colour is subtracted and the correction function is applied. Then, the scattered light intensity (for both colours/polarisations) is smoothed versus azimuthal angles, which entails also averaging over elevation angles. Finally, the angular intensity distribution is scaled to a 700-element vector. It corresponds to a 700-element azimuth angle vector, which is common for all frames. The results of the preprocessing for the whole movie are stored in two arrays named I_{pp} and I_{ss}.

In our previous code, movie processing was done with standard MATLAB functions and was also rather time-consuming - $\sim$ 1 hour per gigabyte of movie - which made it several hours for an average movie. We decided to speed it up by writing our own functions in C/C++ and using parallelization. All compute-intensive calculations were written in CUDA C/C++. C++ code was compiled to MATLAB MEX-file (IntensCalc.mexw64), to interface with the rest of PikeReader MATLAB application. The movie-processing subroutine is named `IntensCalc` and is the main part of PikeReader application. It is invoked with `Find I(Theta,t)` button in the PikeReader GUI. Movie-processing subroutine rewritten in CUDA

C++ is significantly faster then its predecessor and reduces computation time from hours to seconds. The speed-up was estimated to be $\sim$ 400-fold.

3.2.1. Implementation of movie processing algorithm

A very low execution time of `IntensCalc` was achieved by identifying two major bottlenecks. (i) Reading movie from a hard drive turned out to be the first major bottleneck. Storage device access times can not be directly surpassed programmatically, so it is optimal to read from the hard drive continuously. Accordingly, we decided to create a separate, hard drive-reading-only thread, running in parallel on CPU. (ii) Secondly, we observed that, rather obviously, processing every pixel of every frame on CPU is very time consuming. We decided that these calculations should be performed in parallel on CUDA capable GPU, one frame at a time. The `IntensCalc` subroutine structure is visualised in figure 4. Below, it is described in some detail, and the names of relevant variables are also given.

Input parameters and data. The subroutine requires some pre-prepared data from PikeReader:

- movie file name and number of frames (`NumFrames`), selected colour channels with appropriate wavelength and other minor items passed from PikeReader GUI, all included in handles variable, which contains most of PikeReader environment

- masks, corresponding to colour channels, selecting regions of interest for scattering signals (`ipR`, `ipG`, `ipB`), as well as respective backgrounds averaged over separately selected regions.

- image correction arrays (both angular and intensity, pre-calculated with

`RayTracingCUDA`). Intensity correction arrays for each colour have corresponding mask applied (`ICR_N`, `ICG_N`, `ICB_N`).

- vectors of indices of selected pixels, sorted versus azimuth angle (`I_S_R`, `I_S_G`, `I_S_B`). The azimuth angles are assigned to pixels also by `RayTracingCUDA` subroutine.

Movie-reading thread. The thread reading a movie from the hard drive was made fast and simple. It just reads a movie in optimal 64 kB blocks and copies it to 640 kB buffer which is slightly bigger than one frame (614400 B). 640 kB buffers are organised by a cyclic buffer of pointers. The cyclic buffer is encapsulated in a class usually referred to as a monitor, which allows easy and safe thread synchronisation. Cyclic buffer solves the problem of Random Access Memory (RAM) being usually smaller than a movie recorded in experiment (tens of GB). The reading thread writes to the cyclic buffer, while the frame-processing thread (described in the next section) reads data from cyclic buffer, performs extensive computations and discards used data from cyclic buffer. In order to guarantee smooth data flow, each 640 kB buffer in the cyclic buffer has 4 states: *ready empty*, *ready full*, *used for writing*, *used for reading*, and the cyclic buffer has four corresponding methods: `claimForWrite`, `writeEnd`, `claimForRead`, `readEnd`. Additionally, a larger size of the cyclic buffer helps with balancing uneven reading and calculation paces, originating from operations overheads or using resources by other processes.

Frame-control thread. The movies are encoded in avi ARW6 (16-bit, raw) format. To our best knowledge, the Decodec for this dedicated format has not been published, so the movie format had to be analysed with a Hex editor. It was found, that due to Pike camera hardware and driver peculiarities, the positions of frames in avi container are a function of such camera parameters as shutter and exposition times. In consequence, the frame data position versus the frame header can vary even within the movie. We found that there may exist additional garbage data between a frame header and the actual frame data as well as between actual data and the JUNK section, and before the next frame header. In order to find frame data, the frame header (00db614400) must be found first. Then, after jumping by the frame size, the header of the next frame can be searched. However, the JUNK section also must be spotted. Usually, the frame data be may be found adjacent to the left of the JUNK section (if it is present) or else adjacent to the left of the next frame header. In rare cases (once per movie) JUNK section was found within the frame data. Since proper handling of such cases significantly complicates the algorithm, we decided to handle them manually at later stages of data processing.

Frame processing with GPU. The Frame-control thread retrieves frames from 640 kB buffers from the cyclic buffer, following the above procedure. It is highly probable that at least one full frame will be found in two consecutive 640 kB buffers. When a frame is successfully retrieved from the cyclic buffer, it is copied to GPU global memory, and a sequence of CUDA image-processing kernels (GPU subroutines which execute the same algorithm for every given set of data in parallel) is called.

Pixel-value kernel First, each pixel value is calculated. The 14-bit pixel value, yielded by analog-digital converter of the camera, is stored in 2 bytes: the first is the older 8-bit part, while the second byte contains 6 younger bits in reversed order. In order to calculate the pixel value, the first byte must be rightshifted by 2 and the second byte must be 6-bit reversed (using a look-up table was found the fastest method) and added to the first one.

Demosaic kernel Each (raw) frame is decomposed (with a demosaic kernel) into three (RGB) colours (arrays). Later operations are performed on the selected colour channels in sequence.

Distortion-correction kernel The appropriate (averaged) background value is subtracted from each selected colour channel (array) and the result is divided by the intensity-correction array with the corresponding mask applied. In this way, the amount of data is reduced by removing the irrelevant pixels and each frame intensity distribution is corrected.

Moving-average kernel All pixels (indices) in each colour channel are sorted versus the azimuth angle and the resulting angular intensity distributions (vectors) are smoothed with the moving average algorithm. Atomic addition doesn't really block threads, so this step of the moving average algorithm is quite fast. However, in order to save computation time and resources, the division operations should be limited and an additional helper kernel is called just for performing divisions.

Rescaling kernel The last kernel rescales the angular intensity distribution vectors (for each colour channel) down to a fixed length of 700. Finally, these vectors are copied from GPU memory to the corresponding addresses in MATLAB memory.

4. Conclusions

Harnessing of graphics processing unit (GPU) via Compute Unified Device Architecture (CUDA) platform has enabled a very significant reduction of computation time at a moderate cost. We have rewritten the Mie Scattering Lookup Table Method (MSLTM) codes from MATLAB to CUDA C/C++. We had already reported the CUDA code for inverse problem solving making use of GPU [8]. In case of the inverse problem solving we had obtained up to 800-fold speed-up in comparison to the single-thread MATLAB code running on CPU, while in case of new data preprocessing for inversion procedure code we obtained an overall speed-up of $\sim$ 400-fold. In all cases, the time used for calculations was reduced from hours to seconds. Some of the computational stages have become instantaneous and the data preprocessing time has been practically reduced to the time needed for preparation of required parameters. We aim to automatise some of the manual steps in the preprocessing stages to further reduce the preprocessing time. All code updates will instantly be made available on GitHub repository. In general, short data processing time should open way to precise on-line particle characterisation.

Acknowledgment: The authors acknowledge financial support from the National Science Centre, Poland, grants number 2014/13/D/ST3/01882 and 2014/13/B/ST3/04414 and a hardware grant from NVIDIA Corporation.

[1] N. Riefler, T. Wriedt, Intercomparison of inversion algorithms for particle-sizing using Mie scattering, Particle & Particle Systems Characterization 25 (3) (2008) 216–230. doi:10.1002/ppsc.200700039.

[2] F. Onofri, A. Lenoble, H. Bultynck, P. Guéring, High-resolution laser

diffractometry for the on-line sizing of small transparent fibres, Optics Communications 234 (1-6) (2004) 183–191. doi:10.1016/j.optcom.2004.02.026.

[3] F. Onofri, K. Ren, M. Sentis, Q. Gaubert, C. Pelcé, Experimental validation of the vectorial complex ray model on the inter-caustics scattering of oblate droplets, Optics Express 23 (12) (2015) 15768–15773. doi:10.1364/OE.23.015768.

[4] H. He, W. Li, X. Zhang, M. Xia, K. Yang, Light scattering by a spheroidal bubble with geometrical optics approximation, Journal of Quantitative Spectroscopy and Radiative Transfer 113 (12) (2012) 1467–1475. doi:10.1016/j.jqsrt.2012.03.011.

[5] A. Lugovtsov, A. Priezzhev, S. Nikitin, Light scattering by arbitrarily oriented optically soft spheroidal particles: Calculation in geometric optics approximation, Journal of Quantitative Spectroscopy and Radiative Transfer 106 (1-3) (2007) 285–296. doi:10.1016/j.jqsrt.2007.01.041.

[6] H. Yu, F. Xu, C. Tropea, Optical caustics associated with the primary rainbow of oblate droplets: simulation and application in non-sphericity measurement, Optics Express 21 (22) (2013) 25761–25771. doi:10.1364/OE.21.025761.

[7] M. Sentis, F. Onofri, L. Méès, S. Radev, Scattering of light by large bubbles: Coupling of geometrical and physical optics approximations, Journal of Quantitative Spectroscopy and Radiative Transfer 170 (2016) 8–18. doi:10.1016/j.jqsrt.2015.10.007.

[8] D. Jakubczyk, S. Migacz, G. Derkachov, M. Woźniak, J. Archer, K. Kolwas, Optical diagnostics of a single evaporating droplet using fast parallel computing on graphics processing units, Opto-Electron. Rev. 24 (3) (2016) 42–50. doi:10.1515/oere-2016-0019.

[9] J. Nickolls, W. Dally, The GPU computing era, Micro, IEEE 30 (2) (2010) 56–69.

[10] M. Daga, A. Aji, W. Feng, On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing, in: 2011 Symposium on Application Accelerators in High-Performance Computing, IEEE, 2011, pp. 141–149.

[11] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelick, The landscape of parallel computing research: A view from Berkeley, Tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006).

[12] S. Cook, CUDA programming: a developer's guide to parallel computing with GPUs, Newnes, 2012.

[13] https://github.com/sigrond/RayTracingCUDA_TJ (2016).

[14] D. Jakubczyk, G. Derkachov, M. Kolwas, K. Kolwas, Combining weighting and scatterometry: Application to a levitated droplet of suspension, Journal of Quantitative Spectroscopy and Radiative Transfer 12 (0) (2013) 99 – 104. doi:10.1016/j.jqsrt.2012.11.010.

[15] R. Hołyst, M. Litniewski, D. Jakubczyk, K. Kolwas, M. Kolwas, K. Kowalski, S. Migacz, S. Palesa, M. Zientara, Evaporation of freely suspended sin-

gle droplets: experimental, theoretical and computational simulations, Rep. Prog. Phys. 76 (2013) 034601(19pp). doi:10.1088/0034-4885/76/3/034601.

[16] G. Derkachov, D. Jakubczyk, M. Woźniak, J. Archer, M. Kolwas, High-precision temperature determination of evaporating light-absorbing and non-light-absorbing droplets, J. Phys. Chem. B 118 (2014) 12566–12574. doi:10.1021/jp508823z.

[17] R. Hołyst, M. Litniewski, D. Jakubczyk, M. Zientara, M. Woźniak, Nanoscale transport of energy and mass flux during evaporation of liquid droplets into inert gas: computer simulations and experiments, Soft Matter 9 (32) (2013) 7766–7774. doi:10.1039/c3sm50997d.

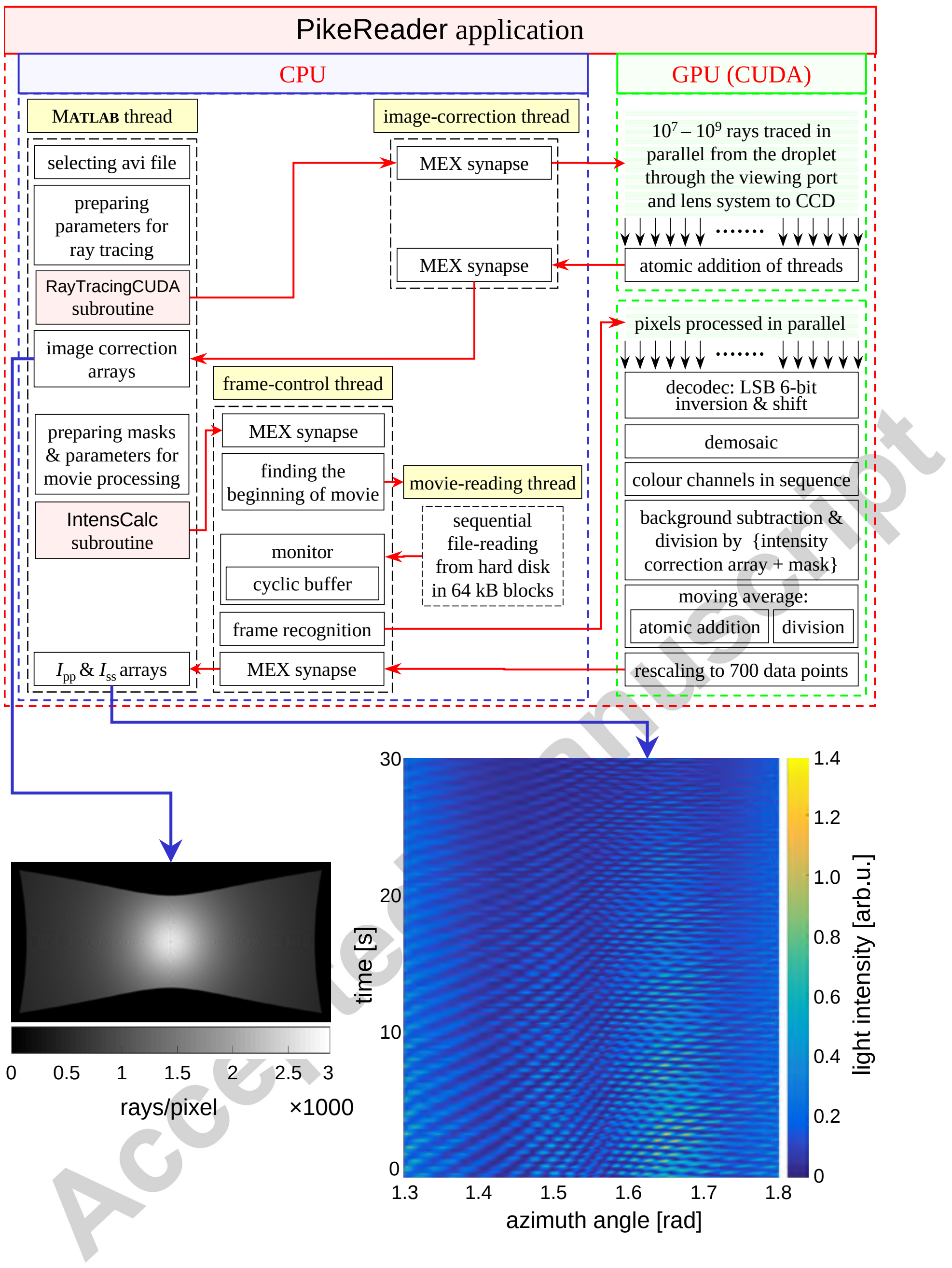

PikeReader application
CPU
GPU (CUDA)
MATLAB thread
selecting avi file
preparing parameters for ray tracing
RayTracingCUDA subroutine
image correction arrays
preparing masks & parameters for movie processing
IntensCalc subroutine
I_{pp} & I_{ss} arrays
image-correction thread
MEX synapse
MEX synapse
$10^7 – 10^9$ rays traced in parallel from the droplet through the viewing port and lens system to CCD
....
atomic addition of threads
pixels processed in parallel
decodec: LSB 6-bit inversion & shift
demosaic
colour channels in sequence
background subtraction & division by {intensity correction array + mask}
moving average:
atomic addition
division
rescaling to 700 data points
frame-control thread
MEX synapse
finding the beginning of movie
monitor
cyclic buffer
frame recognition
MEX synapse
movie-reading thread
sequential file-reading from hard disk in 64 kB blocks
0
0.5
1
1.5
2
2.5
3
rays/pixel
×1000
30
20
10
0
time [s]
1.3
1.4
1.5
1.6
1.7
1.8
azimuth angle [rad]
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
light intensity [arb.u.]

Figure 4: Top: the outline of PikeReader application. Bottom-left: the raw result of the ray tracing subroutine in one of the colour channels. Bottom-right: visualisation of a sample sequence of corrected scattering diagrams (I_{pp} array).