We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Chapter 22

Image Processing with MATLAB and GPU

Antonios Georgantzoglou, Joakim da Silva and Rajesh Jena

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/58300

1. Introduction

MATLAB® (The MathWorks, Natick, MA, USA) is a software package for numerical computing that can be used in various scientific disciplines such as mathematics, physics, electronics, engineering and biology. More than 40 toolboxes are available in the current release (R2013b released in September 2013), which include numerous built-in functions enhanced by access to a high-level programming language.

Since images can be represented by 2D or 3D matrices and the MATLAB processing engine relies on matrix representation of all entities, MATLAB is particularly suitable for implementation and testing of image processing workflows. The Image Processing Toolbox™ (IPT) includes all the necessary tools for general-purpose image processing incorporating more than 300 functions which have been optimised to offer good accuracy and high speed of processing. Moreover, the built-in Parallel Computing Toolbox™ (PCT) has recently been expanded and now supports graphics processing unit (GPU) acceleration for some functions of the IPT. However, for many image processing applications we still need to write our own code, either in MATLAB or, in the case of GPU-accelerated applications requiring specific control over GPU resources, in CUDA (Nvidia Corporation, Santa Clara, CA, USA).

In this chapter, the first part is dedicated to some essential tools of the IPT that can be used in image analysis and assessment as well as in extraction of useful information for further processing and assessment. These include retrieving information about digital images, image adjustment and processing as well as feature extraction and video handling. The second part is dedicated to GPU acceleration of image processing techniques either by using the built-in PCT functions or through writing our own functions. Each section is accompanied by MATLAB example code. The functions and code provided in this chapter are adopted from the MATLAB documentation [1], [2] unless otherwise stated.

2. Image processing on CPU

2.1. Basic image concepts

2.1.1. Pixel representation

A digital image is a visual representation of a scene that can be obtained using a digital optical device. It is composed of a number of picture elements, pixels, and it can be either two-dimensional (2D) or three-dimensional (3D). Different bit-depth images can be found but the most common ones in scientific image processing are the 1-bit binary images (pixel values 0 or 1), the 8-bit grey-scale images (pixel range 0-255) and the 16-bit colour images (pixel range 0-65535) [3]. Figure 1 shows the grey-scale variation, from black to white, for an 8-bit image.

Figure 1. Variation of grey-scale intensities for an 8-bit image.

2.1.2. MATLAB pixel convention

MATLAB uses one-based indexing, where the first pixel along any dimension has index1, whereas many other platforms are zero-based and consider the first index to be 0. By convention, counting of pixel indices starts from the top-left corner of an image with the first and second indices increasing down and towards the right, respectively. Figure 2 visualises the way that MATLAB indexes a 512×512 pixel image. This information is particularly important when the user intends to apply a transformation to a specific pixel or a neighbourhood of pixels.

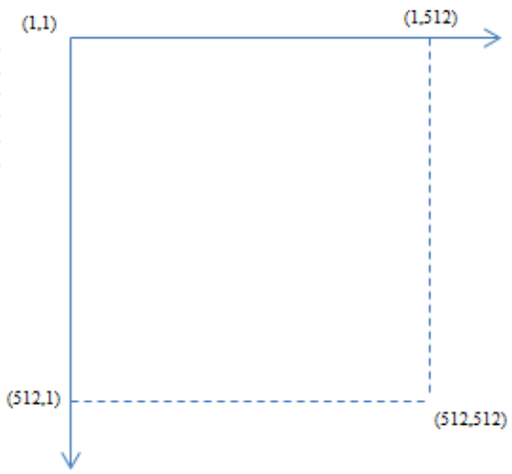

Figure 2. MATLAB's pixel indexing convention.

http://dx.doi.org/10.5772/58300

2.1.3. Image formats

Various image formats are supported by MATLAB including the most commonly used ones, such as JPEG, TIFF, BMP, GIF and PNG. Images can be read, processed and then saved in a format other than their initial one. Various parameters such as the resolution, the bit-depth, the compression level or the colour-space can be adjusted according to the user's preferences.

2.1.4. Digital image processing

Digital image processing refers to the modification of a digital image using a computer in order to emphasise relevant information. This can be achieved by both revealing latent details and suppressing unwanted noise. The process is usually designed according to the desired final outcome which can include simple image enhancement; object detection, segmentation or tracking; parameter estimation; or condition classification. Moreover, when dealing with images intended for inspection by people, the structure and function of the human visual system may be a critical factor in designing any such technique as this determines what can be perceived as an easily distinguishable feature.

2.2. Image pre-processing

Image pre-processing is a procedure that gives initial information about the digital condition of a candidate image. In order to receive such information, we need to load the image on the software platform and examine its type and pixel values.

2.2.1. Image input and output

Just as any other data set in MATLAB, images are represented by a variable. If we consider an image file with name 'image' and format 'tiff', using the function *imread*, the image can be loaded as a 2D or 3D matrix, depending on the image type. Image visualisation is achieved using the function *imshow*; to produce a new figure, a call to *imshow* has to be preceded by a call to *figure*. If, instead, *imshow* is used on its own, the new image replaces the last one in the last open figure window. Saving an image can be achieved using the function *imwrite* where the desired image (i.e. variable), the format and the final name have to be specified. Although the name of the saved image can be chosen freely, when building a large pipeline, it is suggested that the name of each resulting image be representative of the processing step in order to maintain process coherence. The following example represents how this sequence can be achieved.

```
% Image import as variable
im=imread('image.tif');
% Image visualisation
figure; imshow(im);
% Application of image processing
im_proc=...
% Export processed image as file
imwrite(im_proc, 'im_processed.tif', 'tif');
```

The function *imshow* can also be accompanied by a two-element vector *[low high]* with which the user specifies the display range of the grey-scale image. If this vector is left empty *[]*, the minimum and maximum grey-scale values of the image are displayed as black and white pixels, respectively, with the intermediate values as various shades of grey.

```
% Image visualisation with automatically and manually selected
% intensity ranges, respectively
figure; imshow(im, []);
figure; imshow(im, [low high]);
```

2.2.2. Image type conversions

Image type conversion is a useful tool as the user can convert the input image into any desired type. A special and often useful conversion is the transformation of an unsigned integer into a double-precision representation. Any image processing algorithm may thus result in more accurate outcomes since this conversion increases the dynamic range of intensities. The range of the resulting image is 0.0 to 1.0 with MATLAB maintaining up to 15 decimal digits. The following commands are examples of image conversions.

```
% Conversion to double-precision and 8-bit unsigned integer, respectively
im_double=im2double(im);
im_ui8=im2uint8(im);
```

2.2.3. Pixel information

A histogram is a useful intensity representation as it reveals the pixels' intensity distribution. It can be obtained using the function *imhist*. This information can, for example, be used for selecting an appropriate threshold value. Apart from this, a profile of intensities can also reveal information about local intensity variations which can be used to model small details. The function *improfile* can either be applied on pre-selected pixels or as a command prompt function for the user in order to manually select the desired area. Example of code for such processes follows [1], [3].

```
% Histogram presentation: output is the number of pixels (pixel_count)
% distributed at each grey-scale intensity (grey_levels)
[pixel_count grey_levels]=imhist(im);
% Visualisation of histogram with manual limit in grey levels
figure; bar(pixel_count, 'r');
set(gca, 'XLim', [0 grey_levels(end)]);
% Normalisation of histogram
pixel_count_norm=pixel_count / numel(im);
figure; bar(pixel_count_norm, 'b');
set(gca, 'XLim', [0 grey_levels(end)]);
% Profiling of specific pixels
x=[1 205 150 35];
y=[105 230 25 15];
figure; improfile(im, x, y);
```

2.2.4. Contrast adjustment

One of the main pre-processing techniques is contrast adjustment since this process can enhance desired features whilst suppressing other, unwanted ones. MATLAB has various tools for varying the image contrast. The function *imcontrast* supplies a manual adjustment tool through which the user can experiment and find the optimal contrast. The resulting parameters can then be adopted, saved and applied to a stack of images taken under the same conditions. The function *imadjust* can be used to specify an intensity range when the user knows the optimal values or has found them using the *imcontrast* tool. The same function also provides input for the gamma factor of the power-law non-linear contrast adjustment. Besides, a custom-made logarithmic transformation can be applied [3].

```
% Contrast adjustment
im_adj=imadjust(im, [low_in hign_in], [low_out high_out], gamma);
% Example of contrast adjustment
im_adj=imadjust(im, [0.2 0.8], [0.1 0.9], 1.0);
```

Figure 3 presents an original grey-scale image and its contrast adjusted counterpart using the parameters specified in the previous example.

Figure 3. Original image (left, 1296×1936 pixels) and contrast adjusted outcome (right).

```
% Custom logarithmic transformation
im_log=a*log(b+im);
im_log=a+b*log(c – im);
```

Parameters *a*, *b* and *c* can be defined and adjusted by the user meaning any such custom-made logarithmic transformation can be introduced according to specific needs.

Other techniques that can affect contrast are histogram-based ones. A histogram represents an image's grey-level intensity distribution or probability density function. Such knowledge can assist in further processing by helping the user choose the right tools [4]. Histogram stretching can be performed through the *imadjust* function while histogram equalisation can be performed through the function *histeq*. Adaptive histogram equalization can also be applied using

the function *adapthisteq* which considers small image neighbourhoods instead of the whole image as an input.

```
% Histogram equalisation and adaptive equalisation, respectively
im_eq=histeq(im);
im_adeq=adapthisteq(im, 'NumTiles', NumTilesValue);
```

The parameter *NumTilesValue* takes the form of a vector that specifies the number of tiles in each direction. Other parameters can also be specified in the *adapthisteq* function such as the dynamic range of the output data or the histogram shape. Figure 4 shows examples of histogram equalisation and adaptive histogram equalisation, respectively.

Figure 4. Histogram equalisation transformation (left) and adaptive histogram equalization transformation (right) of the original image in Figure 3. Notice the enhanced details in the right-hand image.

2.2.5. Arithmetic operations

Arithmetic operations refer to addition, subtraction, multiplication and division of two images or an image and a constant. Images subject to arithmetic operations need to have the same dimensions and grey-scale representation. The resulting image will have the same dimensions as the input. When a constant value is added or subtracted (instead of a second image), this constant is added or subtracted to each pixel's intensity, increasing or reducing the image luminosity. Most often, such operations are used for detail enhancement or suppression of unnecessary information.

```
% Addition, subtraction, multiplication and division of two images,
% respectively
im_add=imadd(im1, im2);
im_sub=imsubtract(im1, im2);
im_mult=immultiply(im1, im2);
im_div=imdivide(im1, im2);
```

In the code above, the second input parameter (*im2*) can be replaced by a scalar constant.

http://dx.doi.org/10.5772/58300

2.2.6. Miscellaneous transformations

Other useful image transformations include cropping, resizing and rotation. Cropping can be used if the user is interested only in one particular part of the input image. The user can define a specific region of interest and apply any transformation only to this part. Resizing can be applied in order either to expand or reduce the image size. Image size reduction can be especially useful in speeding up a process in case of larger images or large data sets. Rotation can be particularly useful when an image includes features of a particular directionality. The user can specify the applied interpolation method out of nearest neighbour, bilinear and bicubic. Inversion of grey-scale intensities can be useful when the interesting objects have intensities lower than the background. The following functions perform these processes.

```
% Image cropping, resizing, rotation and inversion, respectively
im_crop=imcrop(im, [x y size_x size_y]);
im_res=imresize(im, scale);
im_rot=imrotate(im, angle, method);
im_com=imcomplement(im);
```

2.3. Image processing

2.3.1. Thresholding

Thresholding is one of the most important concepts in image processing as it finds application in almost all projects. Thresholding can be manual or automatic, global or local. In manual mode, the user defines a threshold value, usually depending on the conception of the image (several trials may be needed). In automatic mode, a more detailed understanding of the image is required in order to select the correct method. The IPT provides the function *graythresh* which is based on Otsu's method and the bimodal character of an image [5]. This global threshold will create a black-and-white image where pixels with intensities above this threshold will become white (value 1) and pixels with intensities below this threshold will become black (value 0).

This method can be easily extended to multi-thresholding by using the IPT function *multithresh*. Using this function, the user specifies a suitable number of threshold levels (k) for the image. If this parameter is not supplied, it has the same functionality as the original *graythresh* function. The IPT can visualise the result of the *multithresh* function by using the *imquantize* function. The latter labels the various areas of the image according to the number of thresholds previously specified. The labelled image can then be transformed into an RGB image, preserving the type (e.g. uint8) of the original input. The following code can be used in these processes.

```
% Single threshold application and binarisation, respectively
thresh=graythresh(im);
im_bin=im2bw(im, thresh);
% Multiple threshold application and visualisation of thresholded
% areas as colours of image, respectively
```

```
mult_th=multithresh(im, k);
im_th=imquantize(im, mult_th);
im_rgb=label2rgb(im_th);
```

Figure 5 provides an example of single- and multi-threshold application on the original image of Figure 3.

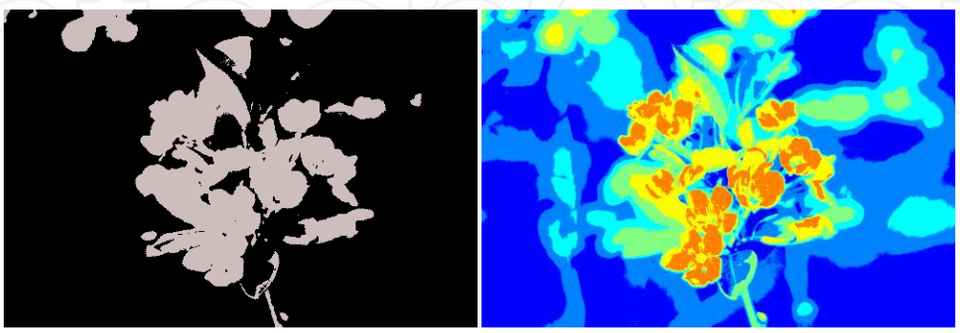

Figure 5. Single-thresholded image (left) and multi-thresholded image using 5 threshold values (right).

2.3.2. Edge detection

Edge detection is an essential part of image processing as it usually emphasises objects' outline or internal structure. An edge is a representation of discontinuity in an image and may characterise a surface that separates two objects or an object from the image background [4]. Boundaries can be characterized by single pixels or connected sequences of pixels. Such a feature can assist in further object recognition and, thus, edge detection is applied in many image processing sequences. The outcome of edge detection is a binary image with edges presented by white pixels.

2.3.3. First-order edge detection operators

The IPT includes the standard first-order edge detectors such as the Roberts, Sobel and Prewitt. Roberts edge detector relies on 2×2 masks whereas the other two rely on 3×3 masks. An optional *threshold* can be specified in order to define the minimum gradient magnitude. Useful code for such detectors follows.

```
% First-order edge detection
im_bin=edge(im, 'roberts', threshold);
im_bin=edge(im, 'sobel', threshold);
im_bin=edge(im, 'prewitt', threshold);
```

Other first-order edge-detectors can also be designed. Examples are the Kirsch and the Robinson masks which are not included in the IPT but are easy to design. They are examples of directional edge detectors which scan the image from different directions in order to detect edges with various orientations. A single kernel is used which, through rotations from 0° to

http://dx.doi.org/10.5772/58300

315° in steps of 45°, creates eight different masks. The image is convolved with each mask and the pixels in the final image are assigned the highest edge detection magnitude obtained from any of the masks [4] The following code presents these two edge-detectors, respectively [6], [4].

```
% Kirsch edge detector
K(:,:,1)=[-5 3 3; -5 0 3; -5 3 3];
K(:,:,2)=[-5 -5 3; -5 0 3; 3 3 3];
K(:,:,3)=[-5 -5 -5; 3 0 0; 3 3 3];
K(:,:,4)=[3 -5 -5; 3 0 -5; 3 3 3];
K(:,:,5)=[3 3 -5; 3 0-5; 3 3 -5];
K(:,:,6)=[3 3 3; 3 0 -5; 3 -5 -5];
K(:,:,7)=[3 3 3; 3 0 3; -5 -5 -5];
K(:,:,8)=[3 3 3; -5 0 3; -5 -5 3];
% Robinson edge detector
R(:,:,1)=[-1 0 1; -2 0 2; -1 0 1];
R(:,:,2)=[0 1 2; -1 0 1; -2 -2 0];
R(:,:,3)=[1 2 1; 0 0 0; -1 -2 -1];
R(:,:,4)=[2 1 0; 1 0 -1; 0 -1 -2];
R(:,:,5)=[1 0 -1; 2 0 -2; 1 0 -1];
R(:,:,6)=[0 -1 -2; 1 0 -1; 2 1 0];
R(:,:,7)=[-1 -2 -1; 0 0 0; 1 2 1];
R(:,:,8)=[-2 -1 0; -1 0 1; 0 1 2];
```

The point detector, another example of an edge detector, detects bright points based on the intensity difference between a central pixel and its neighbours. A point detector can be specified by the following code [7].

```
% Point edge detector
P=[-1 -1 -1; -1 8 -1; -1 -1 -1];
```

2.3.4. Second-order edge detection operators

In addition to first-order edge detectors, second-order edge detectors can find wide application. Such detectors are for example the Canny, zero-cross and Laplacian-of-Gaussian (LoG; also called Marr-Hildreth). The Canny method uses the derivative of a Gaussian filter for finding the gradient of the original image after which it relies on local maxima of the resulting image. [3] The zero-cross method searches for zero crossings after an arbitrary filter has been applied. Finally, the LoG method searches for zero crossings after the LoG transformation has been applied. Useful code for such detectors follows.

```
% Second-order edge detection
im_bin=edge(im, 'log', threshold, sigma);
im_bin=edge(im, 'canny', threshold, sigma);
im_bin=edge(im, 'zero-cross', threshold, filter);
```

In this case, *threshold* refers to the strength of an edge; *sigma* refers to the standard deviation of the Gaussian filter while *filter* refers to any filter that the user applies prior to the edge

detection. In LoG and Canny methods, threshold and sigma can be left unspecified but in the case of the zero-cross method the user has to define a filter. Figure 6 presents the resulting images after application of 'Roberts' and 'Canny' edge detectors, respectively.

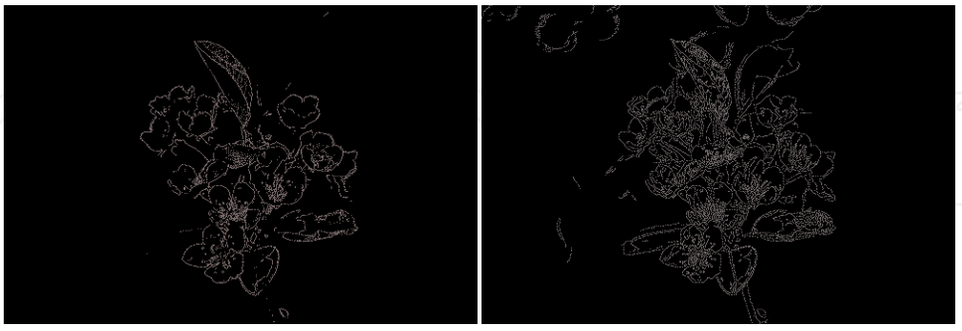

Figure 6. Application of 'Roberts' edge detector (left) and 'Canny' edge detector with a threshold value of 0.05 (right).

2.3.5. Image filtering

Spatial filtering is one of the most important processes in image processing as it can extract and process specific frequencies from an image while other frequencies can be removed or transformed. Usually, filtering is used for image enhancement or noise removal. IPT includes the standard tools needed for image filtering. The function *fspecial* can be used for filter design. Mean, average, Gaussian, Laplacian, Laplacian-of-Gaussian, motion, Prewitt-edge and Sobel-edge filters can be introduced. The designed filter is applied to the image by using the function *imfilter*. Typical examples of code follow.

```
% Filter design
filt_av=fspecial('average', hsize);
filt_gaus=fspecial('gaussian', hsize, sigma);
```

The parameter *hsize* is a vector that represents the number of rows and columns of the neighbourhood that is used when applying the filter. The parameter *sigma* is the standard deviation of the applied Gaussian filter.

```
% Filter application
im_filt_av=imfilter(im, filt_av);
im_filt_gaus=imfilter(im, filt_gaus);
```

Edge detectors can also be applied by filtering the image with the edge operator. An example follows with the application of the previously mentioned point edge detector.

```
% Filter with point edge detector
im_filt_p=imfilter(im, P);
```

Apart from user designed filters, IPT includes filters that can be directly applied to the image. Such examples are the median filter (*medfilt2*), the Wiener filter (*wiener2*) or the 2D order-statistics filter (*ordfilt2*).

```
% Filter application
im_filt_med=medfilt2(im, neighbourhood);
im_filt_ord=ordfilt2(im, order, domain);
im_filt_win=wiener2(im);
```

The *neighbourhood* in the *medfilt2* function specifies the dimensions of the area in which the median value of the pixel will be found. The *ordfilt2* function is a generalised version of the median filter. A neighbourhood is defined by the non-zero pixels of *domain*, and each pixel in the image is replaced by the *order*-th smallest of its neighbours within this domain [1]. An example could be the following command, where each pixel is replaced by the 6th smallest value found in its 3×3 neighbourhood.

```
% Order-statistics filter example
im_filt_ord=ordfilt2(im, 6, ones(3));
```

Figure 7 shows examples of Gaussian and order statistics filtered images.

Figure 7. Images filtered using a Gaussian filter (left – hsize [9 9] and sigma 1) and a 6th order statistics filter (right).

2.3.6. Morphological image processing

Morphological image processing refers to the extraction of descriptors which describe objects of interest and, thus, their morphology determines the tools that are used [8]. Structuring elements are used to probe the image [3]. The function *bwmorph* performs all morphological operations with the addition of suitable parameters. Since the processing time of this function may increase significantly with image complexity, it is supported by the PCT for increased speed of processing. Morphological processing includes dilation, erosion, opening, closing, top-hat and bottom-hat transformation, hit-or-miss transformation as well as other processes that perform pixel-specific changes.

```
% Morphological processing
im_bin=bwmorph(im, operation, n);
```

The parameter *operation* accounts for the type of morphological operator while *n* is the number of times that this process should be repeated. If *n* is not defined, the process is applied only once. Processes such as dilation and erosion can also be applied using individual functions when a custom-made structuring element is to be used. Examples of individual processes follow.

```
% Definition of flat and non-flat structuring element, respectively
se=strel('disk', 5);
se=strel('ball', 10, 5);
% Dilation, erosion and top-hat transformation
im_mor=imdilate(im, se);
im_mor=imerode(im, se);
im_mor=imtophat(im, se);
```

Figure 8 presents examples of dilation and top-hat transformation with a 'disk' structuring element of radius 10.

Figure 8. Dilated image (left) and a top-hat transformed image using a 'disk' structuring element of radius 10 (right).

The distance transform is usually applied to binary images and represents the distance between a white pixel and its closest zero pixel. Pixels in the new image obtain higher values with larger distance from a zero pixel. This transformation can act as a segmentation function and it is often used in the segmentation of overlapping disks [1], [8].

```
% Distance transform
im_dist=bwdist(im_bin);
```

2.3.7. Colour image processing

Colour images are subject to processing in many scientific fields, as different colours can represent different features. The most commonly used colour representation is RGB (Red-Green-Blue). Transformation of RGB images into grey-scale intensity or extraction of a specific colour can be done using the following code:

http://dx.doi.org/10.5772/58300

```
% RGB image to grey-scale intensity image
im_grey=im_rgb=rgb2gray(im);
% Extraction of each colour
im_r=im_rgb(:,:,1);
im_g=im_rgb(:,:,2);
im_b=im_rgb(:,:,3);
```

2.4. Feature extraction

Feature extraction is the process through which recognised objects are assessed according to some geometrical criteria. The first step of this process is to 'label' the objects of a binary image *im_bin* using the function *bwlabel*. The resulting image is referred to as labelled image (*im_lab*). The function scans the image from top to bottom and from left to right attributing a number to each pixel indicating to which object it belongs. Additionally, the IPT has the function *regionprops* which measures features of labelled objects such as area, equivalent diameter, eccentricity, perimeter, and major and minor axis lengths. This function operates on labelled images which contains *n* labelled objects. A full list of the features that can be calculated using *regionprops* can be found in the MATLAB IPT documentation [1].

Apart from the standard features that are included in the IPT, custom-defined features can be measured either by using already calculated features or by introducing completely new measurements. An example could be the measurement of an object's standard geometric characteristics as well as the thinness ratio and compactness (or irregularity) using a *for* loop for assessment of all *n* objects. Since the user may have to handle numerous measurements for many objects, it is usually useful to pre-allocate memory in order to reduce the processing time. The following code can be used to label image objects, measure the features and store them as a table of variables [1], [3].

```
% Labelling and measurement of geometrical features
[im_lab, n] = bwlabel(im_bin);
% Measurement of geometrical features
stats = regionprops(im_lab, 'Area', 'Perimeter', 'Eccentricity', ...
 'MinorAxisLength', 'MajorAxisLength', 'EquivDiameter');
% Memory pre-allocation for new features
[stats(1:n).Roundness] = deal(0);
[stats(1:n).Compactness] = deal(0);
% Measurement of new features Thinesss Ratio and Compactness
for k = 1:n
    [stats(k).ThinessRatio] = ...
        4*pi*[stats(k).Area]/([stats(k).Perimeter].^2);
    [stats(k).Compactness] = 1 / [stats(k).ThinessRatio];
end
```

Measured features are stored in structure arrays. Usually, further processing of features requires transforming structure arrays into matrices. MATLAB cannot perform such transformations without the application of an intermediate step: the structure arrays have first to be transformed into cell arrays which in turn can be converted into matrices.

```
% Conversion of a structure array into a cell array and then into a matrix
cell_features=struct2cell(stats);
features=cell2mat(cell_features');
save('features.mat', 'features');
```

Notice that the transpose of the cell array *cell_features* has been used in order to allocate features to matrix columns rather than rows. For performance reasons it is usually best to orient the data in such a way that they are processed column by column rather than row by row; in this case we are expecting to go through the data feature by feature rather than image by image.

2.5. Processing series of images

In many cases, handling of multiple images can be a laborious task unless an automated process can be established. Assuming we have a batch of 100 images that we would like to process, using a *for* loop and defining the path to the image directory, we can load, process and save the images one at a time. After saving the first image, the next one in the directory is automatically loaded, processed and saved. The procedure continues until the last image has been saved. The following code performs this operation.

```
% Find the images in the current directory with the expected name
% The symbol * indicates that the name 'image' can be followed by any
% additional characters
filelist = dir('image*.tif');
% Find the number of files to be processed
numImages = length(filelist);
% Loop to read, process and save each image
for k = 1:numImages
    myfilename=filelist(k).name;
    im = imread(myfilename);
    im_proc = ... (processing)
    imwrite(im_proc, ['image', num2str(k), '_new.tif'], 'tif');
end
```

2.6. Video handling

2.6.1. Video to frames

An interesting application for image processing is handling video data. In this case, the video file has to be divided into single frames. The function *VideoReader* can be used in order to input the file as a variable. For *n* frames, each frame is then saved as a separate image in any format. The following code reads a video (called *movie*) to a MATLAB structure and saves the frames one by one into 'tiff' format. [9]

```
% Initialisation and frames characteristics
video = VideoReader('movie.avi');
n = video.NumberOfFrames;
height = video.Height;
```

```
width = video.Width;
% Preallocate movie structure
movie(1:n) = struct('cdata', zeros(height, width, 3, 'uint8'));
% Reading video and saving frames
for k = 1:n
    movie(k).cdata = read(video, k);
    imwrite(movie(k).cdata, ...
         strcat('frame_', (strcat(int2str(k), '.tif'))));
end
```

2.6.2. Frames to video

Since every frame is stored as a single image it can be processed accordingly, either one by one or in batch mode. A possible next step in this process can be to combine the processed images into a single video again. If a new movie (called *movie_new*) is to be created from frames (called *frame#*), then the following code supplies the backbone for such a process [9].

```
% Specify video name and frame rate
video = VideoWriter('movie_new.avi');
video.FrameRate = 1;
% Open video recorder to add frames
open(video);
% Find the images in the current directory with the expected name
% The symbol * indicates that the name 'frame' can be followed by any
% additional characters
filelist = dir('frame*.tif');

% List the files and find the number of frames to be added
fileNames = {filelist.name}';
numImages = length(filelist);
% Loop over all images to read, and write to movie file
for k=1:numImages
    myfilemane = strcat('frame', num2str(k), '.tif');
    frame = imread(myfilemane);
    writeVideo(video, frame);
end
% Close video file recorder and play the new video
close(video);
implay('movie_new.avi');
```

3. Image processing on GPU in MATLAB

Large amounts of image data are produced in many technical and experimental situations, in particular where images are repeatedly acquired over time or when dealing with images of higher dimensionality than two. Time-lapse imaging and video recording can be mentioned as examples of the former, whereas the latter can be represented by any of the many tomographic imaging modalities present. 4D computed tomography (CT), where 3D CT images are

acquired at regular intervals to monitor internal patient motion, is an example of an application pertaining to both categories. It is often desirable or even critical to speed up the analysis and processing of such large image data sets, especially for applications running in or near real-time. Due to the inherently parallel nature of many image processing algorithms, they are well suited for implementation on a graphics processing unit (GPU), and consequently we can expect a substantial speedup from such an implementation over code running on a CPU. However, despite the fact that GPUs are nowadays ubiquitous in desktop computers, only 34 out of the several hundred functions of the IPT are GPU-enabled by the PCT in the current MATLAB release (2013b). In this sub-chapter we will explore the possibilities available for someone either wanting to harness the computing power of the GPU directly from MATLAB or to incorporate external GPU code into MATLAB programs. The focus will be on image processing applications, but the techniques presented can with little or no effort be adapted to other applications.

In the first part of this part we look at how to use the built-in, GPU-enabled image processing functions of the PCT. Following this, we explain how pixel-wise manipulations can be carried out using the GPU-enabled version of `arrayfun` and how we can write our own image processing functions making use of over one hundred elementary MATLAB functions that have been implemented to run on GPUs. In the second part of this section, we show how the PCT can be used to call kernel functions written in the CUDA programming language directly from MATLAB. This allows us to make use of existing kernel functions in our MATLAB applications. Further, for those with knowledge of CUDA, it makes more of the GPU's potential available from MATLAB and also provides an easy way of testing kernel functions under development. The third and final part is dedicated to more advanced users who might want to make use of one of the CUDA libraries provided by NVIDIA, who prefer to write their code in a language different from CUDA, who lack access to the Parallel Computing Toolbox, or who have access to an existing GPU code library that they would like to call from MATLAB. We look at how CUDA code can be compiled directly into MEX functions using the PCT, followed by a description of how GPU code written in either CUDA or OpenCL can be accessed from MATLAB by compiling it into a library and creating a MEX wrapper function around it. Finally, we show how the code for a MEX wrapper function can be built directly in our external compiler and, for example, included in an existing Visual Studio (Microsoft Corporation, Redmond, WA, USA) solution so that this is done automatically when building the solution.

3.1. *gpuArray* and built-in GPU-enabled functions

For the examples in this part to work, we need a computer equipped with an NVIDIA GPU of CUDA compute capability 1.3 or greater which is properly set up and recognised by the PCT [10]. The functions `gpuDeviceCount` and `gpuDevice` can be used to identify and select a GPU as described in the PCT documentation [11].

To be able to process an image on the GPU, the corresponding data first have to be copied from main CPU memory over the PCI bus to GPU memory. In MATLAB, data on the GPU are accessed through objects of type `gpuArray`. The command

http://dx.doi.org/10.5772/58300

```
imGpu=gpuArray(im);
```

creates a new `gpuArray` object called `imGpu` and assigns to it a copy of the image data in `im`. `imGpu` will be of the same type as `im` (e.g. `double`, `single`, `int32`, etc.), which might affect the performance of the GPU computation as discussed below. Let us for now assume that `im` is a 3072×3072 array of single precision floating point numbers (`single`). Correspondingly, when we have executed all our GPU calculations, we call the function `gather` to retrieve the result. For example,

```
result=gather(resultGpu);
```

copies the data in the `gpuArray` object `resultGpu` back to `result` in CPU memory. In general, copying data over the PCI bus is relatively slow, meaning that when working with large data sets we should try to avoid unnecessary copies between CPU and GPU memory. When possible, it might therefore be faster to create filters, masks or intermediates directly on the GPU. To do this, `gpuArray` has several static constructors corresponding to standard MATLAB functions, currently `eye`, `false`, `inf`, `nan`, `ones`, `true`, `zeros`, `linspace`, `logspace`, `rand`, `randi` and `randn`, to pre-allocate and initialise GPU memory. These can be invoked by calling `gpuArray.`*`constructor`* where *`constructor`* is replaced by the function call. For example,

```
noiseGpu=gpuArray.randn(3072, 'single');
```

creates a 3072 × 3072 array of normally distributed pseudorandom numbers with zero mean and standard deviation one. As with the corresponding standard MATLAB functions, the last function argument specifies the array element type (in this case `single`), and if omitted it defaults to `double`. While this is normally not a problem when working on modern CPUs, it is worth bearing in mind that NVIDIAs consumer GPUs are often several times faster at processing single precision floating point numbers (`single`), compared to double precision (`double`) or integers (`int32`). This means that where double precision is not crucial, it is a good habit to declare arrays on the GPU as single precision. As an alternative, the first seven static constructors listed above can be called through their corresponding MATLAB function by appending the argument list with `'gpuArray'`. E.g.

```
zerosGpu=zeros(3072, 'int32', 'gpuArray');
```

creates a `gpuArray` object containing 3072×3072 32-bit integers (`int32`) initialised to zero. When calling these functions, an alternative to explicitly specifying the type is using the `'like'` argument. This creates an array of the same type as the argument following the `'like'` argument, i.e.

```
onesGpu=ones(3072, 'like', zerosGpu);
```

creates a gpuArray object of int32 values initialised to one, whereas

```
onesWsp=ones(3072, 'like', im);
```

creates a standard MATLAB array of single values initialised to one. This can be useful when creating new variables in functions that are meant to run both on the CPU and the GPU where we have no *a priori* knowledge of the input type. For a gpuArray object to be able to hold complex numbers, this has to be explicitly specified upon construction, either by using the 'complex' argument when creating it directly on the GPU or by explicit casting when copying non-complex data, e.g. gpuArray(complex(im)). To inspect the properties of a GPU array we can use the standard MATLAB functions such as size, length, isreal etc. In addition to these, the function classUnderlying returns the class underlying the GPU array (since class will just return gpuArray) while existsOnGPU returns true if the argument is a GPU array that exists on the GPU and is accessible.

Once our image data are in GPU memory, we have two options for manipulating them: either we can use the sub-set of the built-in MATLAB functions (including some functions available in toolboxes) that run on the GPU, or we can write our own functions using only element-wise operations and launch them through arrayfun or bsxfun. In the first case, we use normal MATLAB syntax with the knowledge that any of these functions are automatically executed on the GPU as long as at least one argument is of type gpuArray. Using imGpu and randNoiseGpu defined above we can create a new, noisy image on the GPU by typing:

```
noisyImGpu=imGpu+0.2+0.3*noiseGpu;
```

A list of the GPU-enabled MATLAB functions available on the current system, together with all static constructors of gpuArray, can be displayed by typing methods('gpuArray'). For MATLAB 2013b, the list comprises around 200 standard functions plus any additional functions in installed toolboxes [2]. For example, using the GPU-enabled function imnoise from the IPT, the same result as above can be obtained through:

```
noisyImGpu2=imnoise(imGpu, 'gaussian', 0.2, 0.09);
```

(where a variance of 0.09 equals a standard deviation of 0.3). Another, potentially more useful, GPU-enabled function from the IPT is imfilter. Using imGpu from earlier

```
sobelFilter=fspecial('sobel');
filteredImGpu=imfilter(imGpu, sobelFilter);
```

filters the image using a horizontal Sobel filter. Note that sobelFilter is an ordinary 2D MATLAB array, [1 2 1; 0 0 0; −1 −2 −1], but since imGpu is a GPU array, the GPU-

http://dx.doi.org/10.5772/58300

enabled version of `imfilter` is automatically called and the output, `filteredImGpu`, will be a GPU array.

The second option for manipulating GPU arrays directly from MATLAB is by calling our own functions through the built-in `bsxfun` or `arrayfun` functions. As before, if any of the input arguments to the function is a GPU array, the calculation will automatically be carried out on the selected GPU. Thus, a third way of producing a noisy version of `imGpu` would be to first create the function `addAndOffset` that performs an element-wise addition of two images and adds a constant offset:

```
function result=addAndOffset(im1, im2, offset)
result=im1+im2+offset; end
```

and then calling

```
noisyImGpu3=arrayfun(@addAndOffset, imGpu, 0.3*noiseGpu, 0.2);
```

The benefit of writing functions and launching them through `bsxfun` or `arrayfun` compared to calling MATLAB functions directly on GPU arrays is a potential speedup for large functions. This is because in the former case, the whole function can automatically be compiled to a GPU function, called CUDA kernel, resulting in a single launch of GPU code (although the overhead for compiling the function will be added to the first call). In contrast, in the latter case, each operation has to be launched in sequence using the precompiled kernel functions available in MATLAB. However, when running on a GPU, `arrayfun` and `bsxfun` are limited to element-wise operations. In a typical image processing application, this means that each pixel is unaware of its neighbours, which limits the use to functions where pixels are processed independently of one another. As a result, many image filters cannot be implemented in this way, in which case we are left either to use built-in functions as described earlier, or to write our own kernel functions as described in the next part. Further, since we are constrained to element-wise manipulations, the number of built-in functions at our disposal inside our function is somewhat limited. For a complete list of the available built-in functions, as well as some further limitations when using `bsxfun` and `arrayfun` with GPU arrays, see the PCT documentation [12].

Before moving on to the next part we should stress that since GPUs are built to process large amounts of data in parallel, there is no guarantee that running code on a GPU instead of a CPU will always result in a speedup. Although image processing algorithms provide good candidates for substantial speedups, this characteristic of the GPU means that vectorisation of code and simultaneous processing of large amounts of data (i.e. avoiding loops wherever possible) becomes even more crucial than in ordinary MATLAB programs. Further, GPU memory latency and bandwidth often limit the performance of GPU code. This can be alleviated by ensuring that, as far as possible, data that are operated on at the same time are stored nearby in memory. Since arrays are stored in a sequential column-major order in MATLAB, this means avoiding random memory-access patterns where possible and organising our data so that we

mostly operate on columns rather than on rows. Finally, when evaluating the performance of our GPU code we should use the function `gputimeit`. It is called in the same manner as the regular MATLAB function `timeit`, i.e. it takes as argument a function, which itself does not take any arguments, and times it, but is guaranteed to return the accurate execution time for GPU code (which `timeit` is not). If this is not feasible, the code section we want to time can be sandwiched between a `tic` and a `toc`, as long as we add a call to `wait(gpuDevice)` just before the `toc`. This ensures that the time is measured only after execution on the currently selected GPU has finished. (Otherwise MATLAB continues to execute ensuing lines of GPU-independent code, like `toc`, asynchronously without waiting for the GPU calculation to finish). Since the MATLAB profiler only measures CPU time, we need to employ a similar trick to get accurate GPU timings when profiling: if we sandwich the lines or blocks of GPU code we want to time between two calls to `wait(gpuDevice)`, the execution time reported for the desired line or block plus the time taken by the second call to `wait` gives the correct timing.

3.2. Calling CUDA kernel functions from MATLAB

By using the techniques described in the previous part we can use MATLAB to perform many of our standard image processing routines on the GPU. However, it does not allow us to test our own CUDA-implemented algorithms or use existing ones in our MATLAB programs, nor does it provide a means to explicitly control GPU resources, such as global and shared memory. In this part we demonstrate how this can be remedied by creating a `CUDAKernel` object from a kernel function written in CUDA. Instructions on how to write CUDA code is well beyond the scope of this chapter, and therefore this part assumes some previous basic knowledge of this language.

A `CUDAKernel` object required to launch a CUDA kernel function from MATLAB is constructed by calling the static constructor `parallel.gpu.CUDAKernel`. The constructor takes three MATLAB string arguments: the path to a `.ptx` file containing the kernel code, the interface of the kernel function, and the name of the desired entry point. For the first argument, a `.ptx` file with the same name as the source file, containing the corresponding code compiled into pseudo-assembly language, is automatically generated when using the NVIDIA CUDA Compiler (NVCC) with the flag `--ptx` to compile a `.cu` file (if it contains one or more kernel functions). (Note that if using an integrated development environment, you might have to instruct it not to delete `.ptx` files when finishing the build; for example Visual Studio 2010 requires that the flag `--keep` is used.) The second argument can be either the `.cu` file corresponding to the `.ptx` file specified in the first argument, from which the argument list of the desired kernel function can be derived, or the explicit argument list itself. The latter is useful when the `.cu` file is not available, or when the argument list contains standard data types that have been renamed through the use of the `typedef` keyword. The third argument specifies which kernel function in the `.ptx` file to use, and although NVCC mangles function names similar to a C++ compiler, the names in the `.ptx` file are guaranteed to contain the original function name. MATLAB uses substring matching when searching for the entry point and it is therefore often enough to provide the name of the original kernel function (see exceptions

below). Let us assume that we have access to `myFilters.cu` containing several kernel functions named `myFilter1`, `myFilter2`, etc., and its corresponding `myFilters.ptx`. Then

```
gpuFilter1=parallel.gpu.CUDAKernel('myFilters.ptx', myFilters.cu',
 'myFilter1');
```

creates a `CUDAKernel` object called `gpuFilter1` that can be used to launch the CUDA kernel function `myFilter1` from MATLAB. If we further assume that `myFilter1` is declared as

```
__global__ void myFilter1(const float *imIn, float *imOut, float parameter)
```

the second argument above, `'myFilters.cu'`, could equally be replaced by the string `'const float *, float *, float'`. In some cases, care has to be taken when specifying the entry point. For example, if `myFilter2` is a templated function instantiated for more than one template argument, each instance of the resulting function will have a name containing the string `'myFilter2'`. Likewise, if another kernel function called `myFilter1_v2` is declared in the same `.cu` file, specifying `'myFilter1'` as the third argument of `parallel.gpu.CUDAKernel` becomes ambiguous. In these cases, we should provide the mangled function names, which are given during compilation with NVCC in verbose mode, i.e. with-`-ptxas-options=-v` specified. The full mangled name of the kernel used by a `CUDAKernel` object is stored in the object property `EntryPoint`, and can be obtained by e.g. `gpuFilter1.EntryPoint`.

Once a `CUDAKernel` object has been created, we need to specify its launch parameters which is done through the `ThreadBlockSize`, `GridSize` and `SharedMemorySize` object properties. Thus,

```
gpuFilter1.ThreadBlockSize=[32 8 1];
gpuFilter1.GridSize=[96 384 1];
gpuFilter1.SharedMemorySize=4*32*8;
```

sets the block size to 32×8 threads, the grid size to 96×384 blocks and the shared memory size per block to 4×32×8=1024 bytes, enough to hold 256 `single` or `int32` values, or 128 `double` values. In total this will launch 3072×3072 threads, one per pixel of our sample image. A fourth, read-only property called `MaxThreadsPerBlock` holds the upper limit for the total number of threads per block that we can use for the kernel function. If the kernel function is dependent on constant GPU memory, this can be set by calling `setConstantMemeory`, taking as the first parameter the `CUDAKerner` object, as the second parameter the name of the constant memory symbol and as the third parameter a MATLAB array containing the desired data. For example, we can set the constant memory declared as `__constant__ float myConstData` [128] in `myFilters.cu` and needed in `myFilter1`

by calling:

```
setConstantMemory(myFilter1, myConstData, sqrt(single(0:127)));
```

To execute our kernel function we call `feval` with our `GPUKernel` object as the first parameter, followed by the input parameters for our kernel. For input parameters corresponding to arguments passed by value in the CUDA kernel (here: scalars), MATLAB scalars are normally used (although single element GPU arrays also work), whereas for pointer type arguments, either GPU arrays or MATLAB arrays can be used. In the latter case, these are automatically copied to the GPU. A list of supported data types for the kernel arguments can be found in the PCT documentation [13]. In general these are the C/C++ standard types (along with their corresponding pointers) that have MATLAB counterparts, with the addition of `float2` and `double2` that map to MATLAB's complex `single` and `double` types, respectively. `CUDAKernel` objects have three additional, read-only properties related to input and output. `NumRHSArguments` and `MaxNumLHSArguments` respectively hold the expected number of input arguments and the maximum number of output arguments that the objects accepts, and `ArgumentTypes` holds a cell of strings naming the expected MATLAB input types. Each type is prefixed with either `in` or `inout` to signal whether it is input only (corresponding to an argument passed by value or a constant pointer in CUDA) or combined input/output (corresponding to a non-constant pointer in CUDA).

To function in a MATLAB context, a call to a CUDA kernel through a `GPUKernel` object must support output variables. Therefore, pointer arguments that are not declared `const` in the kernel declaration are seen as output variables and, if there is more than one, they are numbered according to their order of appearance in the declaration. This means that calling `gpuFilter1` above produces one output, whereas `gpuFilter2` created from

```
__global__ void myFilter2(const float *imIn, int *imInterm, float *imOut)
```

produces two outputs. With this information we can now call the `myFilter1` kernel function through

```
gpuRes1=gpuArray.zeros(3072, 'single');
gpuRes1=feval(gpuFilter1, gpuIm, gpuRes1, sqrt(2));
```

Similarly, we can call `myFilter2` through

```
gpuInterm=gpuArray.zeros(3072, 'int32');
gpuRes2=gpuArray.zeros(3072, 'single');
[gpuInterm, gpuRes2]=feval(gpuFilter2, gpuIm, gpuInterm, gpuRes2);
```

The output from a `CUDAKernel` object is always a GPU array, which means that if the corresponding input is a MATLAB array it will be created automatically. Consider the kernel

```
__global__ void myFilter3(float *imInOut, float parameter)
```

with its corresponding `CUDAKernel` object `gpuFilter3`. Since `im` from our previous examples a MATLAB array, calling

```
gpuRes3=feval(gpuFilter3, im, 3.14);
```

automatically copies `im` to the GPU and creates a new `gpuArray` object called `gpuRes3` to hold the result.

3.3. MEX functions and GPU programming

In the previous part we saw how, by running our own CUDA kernels directly from MATLAB, we can overcome some of the limitations present when working only with built-in MATLAB functionality. However, we are still (in release 2013b) limited to using kernel functions that take only standard type arguments, and we can access neither external libraries, such as the NVIDIA Performance Primitives, the NVIDIA CUDA Fast Fourier Transform or the NVIDIA CUDA Random Number Generation libraries, nor the GPU's texture memory with its spatial optimised layout and hardware interpolation features. Further, we need to have an NVIDIA GPU, be writing our code in CUDA and have access to the PCT to use the GPU in our MATLAB code. In this section we look at how we can use MEX functions to partly or fully circumnavigate these limitations. This again assumes previous experience of GPU programming and some knowledge of how to write and use MEX functions. A good overview of the latter can be found in the MATLAB documentation [14].

The first option, which removes the technical restrictions but still requires access to the PCT (running on a 64-bit platform) and a GPU from NVIDIA, is to write MEX functions directly containing CUDA code. The CUDA code itself is written exactly as it would be in any other application, and can either be included in the file containing the MEX function entry point or in a separate file. Although this process is well documented in the PCT documentation [15] and through the PCT examples [16], we briefly describe it here for consistency. The main advantage of this approach over the one described later is that it enables us to write MEX functions that use GPU arrays as input and output through the underlying C/C++ object `mxGPUArray`. As all MEX input and output, GPU arrays are passed to MEX functions as pointers to `mxArray` objects. The first step is therefore to call either `mxGPUCreateFromMxArray` or `mxGPUCopyFromMxArray` on the pointer to the `mxArray` containing the GPU data, in order to obtain a pointer to an `mxGPUArray`. In the former case, the `mxGPUArray` becomes read-only, whereas in the latter case the data is copied so that the returned `mxGPUArray` can be modified. (`mxGPUCreateFromMxArray` and `mxGPUCopyFromMxArray` also accept pointers to `mxArray` objects containing CPU data, in which case the data is copied to the GPU regardless of the function used.) We can now obtain a raw pointer to device memory that can be passed to CUDA kernels by calling `mxGPUGetDataReadOnly` (in the case of read-only data) or `mxGPUGetData` (otherwise) and explicitly casting the returned pointer to the correct type. Information about the number of dimensions, dimension sizes, total number of

elements, type and complexity of the underlying data of an `mxGPUArray` object can be further obtained through the functions `mxGPUGetNumberOfDimensions`, `mxGPUGetDimensions`, `mxGPUGetNumberOfElements`, `mxGPUGetClassID`, and `mxGPUGetComplexity`. We can also create a new `mxGPUArray` object through `mxGPUCreateGPUArray`, which allocates and, if we want, initialises memory on the GPU for our return data. With this we are in a position where we can treat input from MATLAB just as any other data on the GPU and perform our desired calculations. Once we are ready to return our results to MATLAB we call either `mxGPUCreateMxArrayOnCPU` or `mxGPUCreateMxArrayOnGPU` to obtain a pointer to an `mxArray` object that can be returned to MATLAB through `plhs`. The former copies the data back to the CPU making the MEX function return a standard MATLAB array whereas in the latter case the data stays on the GPU and a GPU array is returned. Finally, we should call `mxGPUDestroyGPUArray` on any `mxGPUArray` objects that we have created. This deletes them from the CPU and, unless they are referenced by another `mxArray` object (as will be the case if they are returned through `plhs`), frees the corresponding GPU memory. Note that there are several other functions in the `mxGPU` family to examine, duplicate, create and copy `mxGPUArray` objects, and in particular for working with complex arrays, that work in a similar way and which are described in the PCT documentation [17].

For the above to work, we need to include `mxGPUArray.h`, in addition to `mex.h`, in our source file. The source file has to have the extension `.cu` and it should contain a call to the function `mxInitGPU` before launching any GPU code in order to initialise the MATLAB GPU library. Provided that the environment variable `MW_NVCC_PATH` is set to the NVCC folder path and that a copy of the PCT version of `mexopts.bat` or `mexopts.sh` (*`matlabroot`*`\toolbox\distcomp\gpu\extern\src\mex\`*`xxx`*`64\`) is located in the same folder as the source code, we can compile our CUDA containing MEX functions from MATLAB in the usual way using the mex command. If using external libraries, these also have to be provided, which can normally be done by passing the full library path, including file name, to the mex command after the `.c`, `.cpp` or `.cu` file path.

A bare-bone MEX function calling the kernel function `myFilter1` from earlier, which takes into account the considerations above but is stripped of any boundary or argument checking, follows:

```
#include "mex.h"
#include "gpu\mxGPUArray.h"

__global__ void myFilter1(const float *imIn, float *imOut, float parameter)
{
    // Kernel code
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray const *prhs[]) {
    mxGPUArray const *gpuArrayIn;
    mxGPUArray *gpuArrayOut;
    float const *devIn;
```

```
    float *devOut;

     mxInitGPU();
    gpuArrayIn = mxGPUCreateFromMxArray(prhs[0]);
    devIn = (float const *)(mxGPUGetDataReadOnly(gpuArrayIn));
    gpuArrayOut =
        mxGPUCreateGPUArray(mxGPUGetNumberOfDimensions(gpuArrayIn),
        mxGPUGetDimensions(gpuArrayIn), mxGPUGetClassID(gpuArrayIn),
        mxGPUGetComplexity(gpuArrayIn), MX_GPU_DO_NOT_INITIALIZE);
    devOut = (float *)(mxGPUGetData(gpuArrayOut));

     const mwSize *dim = mxGPUGetDimensions(gpuArrayIn);
    int height = (int)(dim[0]);
    int width = (int)(dim[1]);
    dim3 block(32, 8, 1);
    dim3 grid((height+block.x-1)/block.x, (width+block.y-1)/block.y, 1);

    float param = *(float *)(mxGetData(prhs[1]));

    myFilter1<<<grid, block>>> (devIn, devOut, param);

    plhs[0] = mxGPUCreateMxArrayOnGPU(gpuArrayOut);
    mxGPUDestroyGPUArray(gpuArrayIn);
    mxGPUDestroyGPUArray(gpuArrayOut);
}
```

After compilation, assuming the above code is found in `mexFilter1.cu`, we can call the MEX function like this:

```
gpuRes3=mexFilter1(imGpu, single(2.72));
```

Note the explicit type conversion necessary to convert the second argument to `single` to match the input type. Note also that, depending on the compiler and system settings, in order to have `mwSize` correctly identified as a 64-bit type, we might need to use the `-largeArraysDims` flag when compiling using the `mex` command.

The rest of this part will be dedicated to describing how we can call GPU code from MATLAB even if we do not have access to the PCT or write our code in CUDA. Since without the PCT, the `mex` command is not set up to compile anything except standard C/C++ code, we have two alternatives to achieve what we want. The only drawback with these, compared to when using `mxGPUArray` from the PCT is that it is harder to have data in GPU memory persist when returning to MATLAB between calls to MEX functions. (This can still be achieved by explicitly casting a GPU memory pointer to an integer type of correct length which is returned to MATLAB. The integer is then passed to the next MEX function which casts it back to the correct pointer type. However, the problem with this approach lies in eliminating the risk of memory leaks; although solutions for this exist, they are beyond the scope of this chapter.) This means that unless we go through any extra effort, we are left either to perform all our GPU calculations

from the same MEX function before returning control to MATLAB or suffer the penalty associated with copying our data back and forth between each call. In many cases, however, this may provide less of a limitation than one might initially think, especially when using MATLAB to test GPU code under development or using MATLAB as a front-end to existing code libraries.

The first alternative is to compile our GPU code, regardless of in which language it is written, into a static library using our external compiler, and then to call this library from a MEX function that we compile using the `mex` command. Since our MEX function cannot call GPU functions directly, a small C/C++ wrapper function has to be written around our GPU code. A wrapper for the `myFilter1` CUDA kernel, which we can place either in the same file as the CUDA code or in a separate `.cu` file, could look like this (again, error checking has been omitted for brevity):

```
void callMyFilter1(const float *imIn, float *imOut, float param, int height,
int width)
{
    float *devIn;
    cudaMalloc((void**)&devIn, height*width*sizeof(float));
    cudaMemcpy(devIn, imIn, height*width*sizeof(float),
        cudaMemcpyHostToDevice);
    float *devOut;
    cudaMalloc((void**)&devOut, height*width*sizeof(float));

    dim3 block(32, 8, 1);
    dim3 grid((height+block.x-1)/block.x, (width+block.y-1)/block.y, 1);

    myFilter1<<<grid, block>>> (devIn, devOut, param);

    cudaMemcpy(imOut, devOut, height*width*sizeof(float),
        cudaMemcpyDeviceToHost);
    cudaFree(devIn);
    cudaFree(devOut);
}
```

Once the static library, normally a `.lib` file under Windows or a `.a` file under Linux and Mac OS, has been created, we can write a MEX function calling the wrapper:

```
#include "mex.h"

void callMyFilter1(const float *, float *, float, int, int);

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray const *prhs[]) {
    int height = mxGetM(prhs[0]);
    int width = mxGetN(prhs[0]);
    const float *imIn = (float *)(mxGetData(prhs[0]));
    float param = (float)(mxGetScalar(prhs[1]);
    plhs[0] = mxCreateNumericMatrix(height, width, mxSINGLE_CLASS, mxREAL);
```

```
    float *imOut = (float *)(mxGetData(plhs[0]));
    callMyFilter1(imIn, imOut, param, height, width);
}
```

Note that it is normally better, especially if using a pre-existing library, to use `#include` to include the corresponding header files rather than, as in the example above, to manually enter the function prototype. We should now be able to compile this wrapper using the `mex` command, remembering to pass the path to our own library as well as to the necessary GPU runtime library. For CUDA, this is `cudart.lib` on Windows, `libcuda.so` on Linux, or `libcuda.dylib` on Mac OS. Thus, assuming that the code above is found in `myFilter1Mex.cpp` and that the library is called `myLibrary.lib`, on Windows the call would look like:

```
mex myFilter1Mex.cpp 'library_path\myLibrary.lib' 'cuda_path\cudart.lib'
```

The second alternative is to build the MEX function directly in our external compiler, without going through the `mex` command. By doing this, the whole process can be carried out in one step and, if we wish, we are free to keep all our code in a single file. The main advantage of this approach occurs if we are developing or using an existing GPU library which we would like to call from MATLAB, for example for testing purposes. In such a case we can add the MEX file to our normal build routine so that every time we rebuild our library we automatically get the MEX function to call it from MATLAB.

A MEX function is a dynamically linked shared library which exports a single entry point called `mexFunction`. Hence, by mimicking the steps carried out by the `mex` command (found in `mexopts.bat` or `mexopts.sh`) when calling the external compiler, we can replicate this from outside MATLAB. We can view the exact steps carried out on a particular system by calling the `mex` command in verbose mode, i.e. using the `-v` flag. Detailed information on how to build MEX functions for Windows and UNIX-like systems (Linux and Mac OS) can be also found in the MATLAB documentation [18]. Here we look at a minimal example from a Windows-centric point of view, although the procedure should be similar on other systems. First, we need to specify that we want to build a dynamically linked shared library, called a dynamic-link library on Windows, and give it the correct file extension, e.g. `.mexw64`; second, we have to provide the compiler with the path to our MEX header files, normally `mex.h`; third, we have to pass the libraries needed to build our MEX function, called `libmx.lib`, `libmex.lib` and `libmat.lib` on Windows, to the linker together with their path; and finally, as mentioned above, we need to export the function named `mexFunction`. In Visual Studio 2010, all of the above steps are done in the Project Properties page of the project in question. Under `Configuration properties-> General`, we set the `Target Extension` to `.mexw64` and the `Configutation Type` to `Dynamic Library (.dll)`. For the header files we add `$(MATLAB_ROOT)\extern\include;` to `Include Directories` under `Configuration Properties-> VC++Directories`. For the libraries, we add `libmx.lib;libmex.lib;libmat.lib;` to `Additional Dependencies` under `Con-`

figuration Properties-> Linker-> Input and $(MATLAB_ROOT)\extern\lib\win64\microsoft; to Additional Library Directories under Configuration properties-> Linker-> General. Finally, we add /export:mexFunction to Additional Options under Configuration Properties-> Linker-> Command Line. In the above steps it is assumed that the variable MATLAB_ROOT is set to the path of the MATLAB root, otherwise we have to change $(MATLAB_ROOT) to, for example, C:\Program Files\MATLAB\2013b. The code for the MEX example, finally, would look something like this:

```
#include <cuda.h>
#include "mex.h"

__global__ void myFilter1(const float *imIn, float *imOut, float parameter)
{
    // Kernel code
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray const *prhs[])
{
    int height = mxGetM(prhs[0]);
    int width = mxGetN(prhs[0]);
    const float *imIn = (float *)(mxGetData(prhs[0]));
    float param = (float)(mxGetScalar(prhs[1]));
    plhs[0] = mxCreateNumericMatrix(height, width, mxSINGLE_CLASS, mxREAL);
    float *imOut = (float *)(mxGetData(plhs[0]));

    float *devIn;
    cudaMalloc((void**)&devIn, height*width*sizeof(float));
    cudaMemcpy(devIn, imIn, height*width*sizeof(float),
        cudaMemcpyHostToDevice);
    float *devOut;
    cudaMalloc((void**)&devOut, height*width*sizeof(float));

    dim3 block(32, 8, 1);
    dim3 grid((height+block.x-1)/block.x, (width+block.y-1)/block.y, 1);

    myFilter1<<<grid, block>>> (devIn, devOut, param);

    cudaMemcpy(imOut, devOut, height*width*sizeof(float),
        cudaMemcpyDeviceToHost);
    cudaFree(devIn);
    cudaFree(devOut);
}
```

In the case of an existing code library, we can add a new component (such as a new project to an existing solution in Visual Studio) containing a MEX function calling our desired GPU functions so that it is built directly with the original library without any additional steps.

4. Conclusion

In conclusion, MATLAB is a useful tool for prototyping, developing and testing image processing algorithms and pipelines. It provides the user with the option of either using the functions of the IPT or leveraging the capabilities of a high-level programming language combined with many built-in standard functions to create their own algorithms. When high performance or processing of large amounts of data is required, the computational power of a GPU can be exploited. At this point, there are three main options for image processing on GPU in MATLAB: i) we can stick entirely to MATALB code, making use of the built-in, GPU-enabled functions in the IPT and the PCT as well as our own GPU functions built from element-wise operations only; ii) we can use the framework provided by the PCT to call our own CUDA kernel functions directly from MATLAB; iii) we can write a MEX function in C/C++ that can be called from MATLAB and which in turn calls the GPU code of our choice.

Acknowledgements

The authors would like to acknowledge the European Commission FP7 ENTERVISION programme, grant agreement no.: 264552

Author details

Antonios Georgantzoglou*, Joakim da Silva and Rajesh Jena

*Address all correspondence to: ag718@cam.ac.uk

Department of Oncology, University of Cambridge, Cambridge, UK

References

[1] The MathWorks, Inc., "Image Processing Toolbox Documentation," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/images/. [Accessed 25 October 2013].

[2] The MathWorks, Inc., "Parallel Computing Toolbox Documentation," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/distcomp/. [Accessed 25 October 2013].

[3] O. Marques, Practical Image and Video Processing, Hoboken, NJ: John Wiley & Sons, Inc., 2011.

[4] S. Sridhar, Digital Image Processing, New Delhi: Oxford University Press, 2011.

[5] N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics,* Vols. SMC-9, no. 1, pp. 62-66, 1979.

[6] W. Burger and M. J. Burge, Principles of Digital Image Processing: Fundamental Techniques, London: Springer, 2009.

[7] K. Najarian and R. Splinter, Biomedical Signal and Image Processing, Boca Raton, FL; London: CRC Press / Taylor & Francis Group, 2006.

[8] C. Solomon and T. Breckon, Fundamentals of Digital Image Processing: A Practical Approach with Examples in MATLAB, Chichester: John Wiley & Sons, Ltd, 2011.

[9] The MathWorks, Inc., "Data and File Management > Data Import and Export > Audio and Video," The MathWorks, Inc., 2014. [Online]. Available: http://www.mathworks.co.uk/help/matlab/audio-and-video.html. [Accessed 25 January 2014].

[10] The MathWorks, Inc., "Discovery > MATLAB GPU Compurting > MATLAB GPU Computing Support for NVIDIA CUDA-Enabled GPUs," The MathWorks, Inc., 2014. [Online]. Available: http://www.mathworks.co.uk/discovery/matlab-gpu.html. [Accessed 25 January 2014].

[11] The MathWorks, Inc., "Parallel Computing Toolbox Documentation (Parallel Computing Toolbox > GPU Computing > Identify and Select a GPU Device)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/distcomp/. [Accessed 25 October 2013].

[12] The MathWorks, Inc., "Parallel Computing Toolbox Documentation (Parallel Computing Toolbox > GPU Computing > Run Element-wise MATLAB Code on a GPU)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/distcomp/. [Accessed 25 October 2013].

[13] The MathWorks, Inc., "Parallel Computing Toolbox Documentation (Parallel Computing Toolbox > GPU Computing > Run CUDA or PTX Code on GPU)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/distcomp/. [Accessed 25 October 2013].

[14] The MathWorks, Inc., "MATLAB Documentation (MATLAB > Advanced Software Development > External Programming Language Interfaces > Application Programming Interfaces to MATLAB > Use and Create MEX-Files)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/matlab/. [Accessed 25 October 2013].

[15] The MathWorks, Inc., "Parallel Computing Toolbox Documentation (Parallel Computing Toolbox > GPU Computing > Run MEX-Functions Containing CUDA Code)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/distcomp/. [Accessed 25 October 2013].

[16] The MathWorks, Inc., "Parallel Computing Toolbox Documentation (Parallel Computing Toolbox > Parallel Computing Toolbox Examples)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/distcomp/. [Accessed 25 October 2013].

[17] The MathWorks, Inc., "Parallel Computing Toolbox Documentation (Parallel Computing Toolbox > GPU Computing > C Functions)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/distcomp/. [Accessed 25 October 2013].

[18] The MathWorks, Inc., "MATLAB Documentation (MATLAB > Advanced Software Development > External Programming Language Interfaces > Application Programming Interfaces to MATLAB > Use and Create MEX-Files > Build C/C++ MEX-Files > Custom Building MEX-Files)," The MathWorks, Inc., 2013. [Online]. Available: http://www.mathworks.co.uk/help/matlab/. [Accessed 25 October 2013].

IntechOpen