A GPU Accelerated Local Polynomial Approximation Algorithm for Efficient Denoising of MR Images

Artur Klepaczko

Abstract This paper presents a parallelized implementation of the Local Polynomial Approximation algorithm targetted at CUDA-enabled GPU hardware. Although the application area of LPA in the image processing domain is very wide, here the focus is put on magnetic resonance image denoising. In this case, LPA serves as a pre-processing step in the method based on Shape-Adaptive Discrete Cosine Transform. It is shown, how the designed efficient implementation of LPA substantially reduces the execution time of SA-DCT.

1 Introduction

The problem of noise removal from MR images has been extensively studied and variety of effective solutions were already proposed. It is an important problem while noise-free images can largely improve correctness of medical diagnosis based both on qualitative assessment as well as automatic computer-aided pattern recognition tools (segmentation, classification). Typically, the signal-to-noise ratio in real images results from the trade-off between acquisition speed, resolution, scanner field strength. The sources of noise in MR are multifold, including thermal phenomena, inductive losses, or sampling frequency.

A common assumption underlying majority of approaches derives from the observation that noise signal in magnitude MR images can be modeled by the Rician distribution [9]. Furthermore, it can be shown, that in the case of high SNR, distribution of noise approaches the Gaussian model. Therefore, the most straightforward techniques applied to MR image denoising are based on Gaussian or Wiener filters. Another approach involves restoration based

Lodz University of Technology, Institute of Electronics, ul. Wolczanska 211/215, 90-924 Lodz aklepaczko@p.lodz.pl

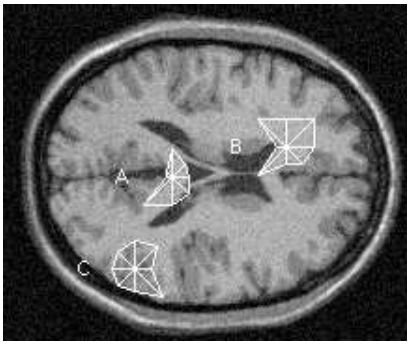

Fig. 1 Example pixels neighborhood masks determined by the LPA algorithm. Masks are adjusted to true local image contents while noise-associated intensity fluctuations are tolerated.

on non-linear optimization of an image modeled as a Markov random field [4]. However, these methods usually fail leading to over-smoothed and blurred images, corrupted also by edge-related artifacts.

Therefore, the more sophisticated techniques either apply Rician model without its simplification (e.g. joint geometric-, radiometric-, and median-metric filter [2], Median Absolute Deviation estimator or Nonlocal Maximum Likelihood [5, 3]) or—although assume the noise to be normally distributed—operate on voxels neighborhoods adapted to fine details of visualized structures, thus making noise removal more robust to blurring and over-smoothing problems. In this paper we focus on one such method, namely the Shape-Adaptive Discrete Cosine Transform (SA-DCT)[6]. As a first step, SA-DCT determines homogeneous local neighborhoods of every image voxel using Local Polynomial Approximation algorithm. The neighborhoods are finely adjusted to shape and size of local image contents (cf. Fig. 1). However, determination of local neighborhoods for every voxel involves significant computational overhead. Application of LPA to high-resolution 3D MR images occurs inefficient. Thus, in this paper parallelized GPU-accelerated implementation of LPA is proposed, so that noise can be removed from MR data not only effectively, but also efficiently.

2 Local Polynomial Approximation

The LPA algorithm is a technique of non-parametric regression adopted in various image processing applications[7, 1]. Using low order polynomial function, LPA models a non-linear relationship between an independent variable X and a dependent variable Y. Data are fitted to a modeled polynomial function within a sliding window positioned at subsequent observations (X, Y) –

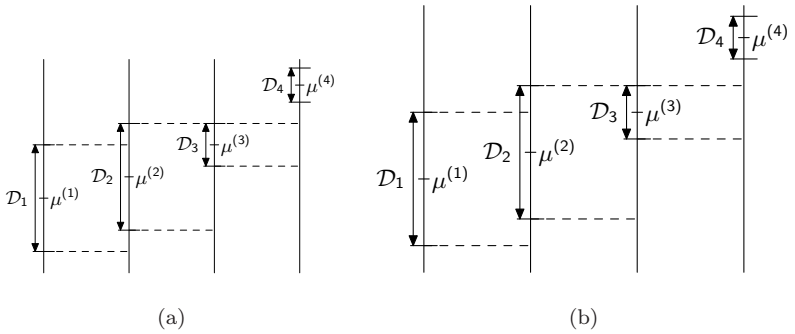

Fig. 2 Distinct directions $\theta_i, i = 1..8$ used in LPA filtering (a) and the illustration of the Intersection of Confidence Intervals rule for $h_{\max} = 3$ (b).

e.g. measured values of a sampled signal. In a window, a signal is convolved with a kernel function of a known form. This enables estimating values of Y in the neighborhood of a given data point X. Window size h is a key parameter of the method. It is defined as a number of data samples beyond which it becomes impossible to estimate signal Y basing on values measured in the proximal neighborhood of X.

In this study the 3D image is considered as a stack of flat slices and thus a 2D variant of LPA algorithm is applied. Then, each pixel neighborhood is filtered in 8 distinct directions, as illustrated in Fig. 2a. For a given pixel X, the filtered value μ is calculated as

$$\mu^{(h)} = \sum_{j=1}^{h} g_j^{(h)} I(X + (j-1)\theta_i), \tag{1}$$

where $g^{(h)}$ is a discrete convolution kernel of scale h (window size), $g_j^{(h)}$ with $j = 1, \ldots, h$ denote kernel weights which sum to unity and decrease with the increasing distance from a center pixel X. The exact procedure of weights generation is described in [7]. I is a 2D matrix of image intensity values.

Adjusting the window size to local image contents is performed using the *intersection of confidence intervals* (ICI) rule. The idea is to test several values of scale h, i.e. $h \in \{h_1 \ldots, h_k\}$ and $h_1 < h_2 < \ldots < h_k$ and for each of them evalute (1) as well as local standard deviation value

$$\sigma_{\mu^{(h)}} = \sigma \|g^{(h)}\|, \tag{2}$$

where σ is the global standard deviation determined for the whole image. Then for each direction θ_i and scale h one calculates confidence intervals

$$\mathcal{D}_h = [\mu^{(h)} - \Gamma\sigma_{\mu^{(h)}}, \mu^{(h)} + \Gamma\sigma_{\mu^{(h)}}], \tag{3}$$

in which $\Gamma > 0$ denotes a global parameter that allows controlling noise tolerance. The lower Γ, the stronger requirement for local homogeneity is, and thus fewer pixels are included in the resulting neighborhood volumes. The ICI rule states that for each direction one should choose a maximum value of h that ensures nonempty intersection of all previous confidence intervals, i.e. (cf. Fig. 2b)

$$h_{\max,i} = \max_{h \in \{h_1,\ldots,h_k\}} \{h : (\mathcal{D}_1 \cap \mathcal{D}_2 \cap \cdots \cap \mathcal{D}_h) \neq \emptyset\}. \tag{4}$$

In this study it is arbitrarily set $h \in \{1, 2, 3, 5, 7, 8\}$, hence the upper bound for the window size in any direction amounts to 8 pixels. On completion, pixels determined by relations $X + h_{\max,i}\theta_i$ constitute a set of a hull vertices whose interior determines a locally adapted volume of interest of X.

3 GPU-based implementation of LPA

One of the most important features of the CUDA technology is the separation of the code layer from the GPU hardware, allowing seamless execution of the same binaries on different devices. However there are some hardware-related details specific to various generations of Nvidia's products and thus it should be noted, that this study targets the *Fermi architecture* (GF116 in particular). Below the designed implementation is presented in 3 variations, starting from the most strightforward solution. Then, additional optimization mechanisms are introduced to improve ultimate efficiency. The presentation is preceded by recalling basic configuration issues common to any CUDA project.

3.1 The execution configuration

The CUDA programming model assumes problem decomposition into a series of threads, each executing the same operation—called a *kernel*—on different portions of data. CUDA threads have to be organized in blocks, and then blocks of threads build up a grid. The total number of threads has to comply with the problem size, i.e. every data element (an image pixel) is assigned its own corresponding thread. The grid organization into blocks of threads is referred to as *execution configuration* and it is important to properly adjust it in order to achieve the maximal utilization of GPU hardware resources. All CUDA-enabled processors are composed of the Streaming Multiprocessors (SM). In the Fermi architectures a SM can be assigned at most 8 blocks or 1536 threads at a time. Therefore, too large (e.g. 32×32 threads) or too small (e.g. 8×8) blocks results in poorer utilization of computational logic

and potential loss in performance. Percentage of threads actually assigned to a SM in relation to maximum possible number of threads per SM is called the *occupancy ratio*.

Beside execution configuration, there are other factors which may cause occupancy to be lower than optimal 100%. These relate mainly to memory resources. In CUDA there are four major types of memory. The fastest accessible are per-thread registers. They are used mainly for storing scalar variables declared in a kernel scope. In GF116 there is a limit of 32K of 32-bit registers per multiprocessor. Thus, if a SM is assigned a total number of 1536 threads, no more than 21 registers are available for a thread. It may however be beneficial to speed up kernel execution by extensive usage of registers at the cost of occupancy. Secondly, each thread can access its own local memory area (of 512 KB size), which is however a long-latency, off-chip storage. The third memory option is 48KB of shared memory to be distributed among blocks in a SM. Similarly to registers, it is zero-overhead memory but its over-utilization by a block may result in degraded occupancy. Eventually, global memory is a large capacity data storage, available to all threads in a grid throughout the whole application lifetime. However, it is again, an off-chip long latency storage.

Taking into account the above considerations, the following configuration is used in the proposed design. The block size is set to 16×16. This gives 256 threads per block and allows 6 blocks to simultaneously reside in SM. The grid size is adjusted to match the processed image. For an image of size 256×256 pixels, the grid dimension will be 16×16. In this configuration, the amount of shared memory per block is limited to 8KB (if 100% occupancy is to be maintained). The number of registers per kernel is kept equal to 21 (it can be fixed at compile time), however it could be tuned to improve final performance.

3.2 The basic kernel

The first step of LPA is to estimate global standard deviation σ of noise in an input image. This can be accomplished relatively fast in a single CPU thread. Then, the CPU part of the program (the *host*) transfers image data to the the GPU global memory and executes kernel in a grid of threads, where every thread evaluates equations (1) to (4) in its own dedicated pixel X. Thus, the kernel has to begin with identifying its pixel coordinates and proceeds as shown in Fig. 3. Note, that in this basic implementation the thread must read global memory in step 4 at least 9 times—to fetch its pixel and the nearest neighbors intensity values. In the worst case, where the ICI rule holds for the most distal pixels in all 8 directions, the number of global memory transactions in step 4 reaches value of 65 which can cause significant computational overhead.

```
1. Determine pixel coordiantes based on thread and block ID
   numbers and dimensions.
2. For each direction θ_i, i = 1,..,8
3.       For each scale h_j ∈ h_1,..h_k
4.               Calculate μ^h according to (1)
5.               Calculate σ_{μ^(h)} according to (2)
6.               Calculate D_{h_j} according to (3)
7.               If the ICI rule (eq. (4)) holds for D_{h_j}
8.                       go to step 4.
9.               else break
10.      Store scale h_j^i for direction θ_i and go to step 2.
```

Fig. 3 Pseudo code of the basic CUDA implementation of the LPA algorithm

3.3 Shared memory

Shared memory can be used to reduce the extensive traffic between GPU and off-chip global memory, in which many transfers concern the same data. Note that adjacent pixel neighborhoods investigate partially the same image region. For example, two pixels which differ only on horizontal coordinate by one require—in direction $\theta_1 = 0°$—analysis of an image row whose size is 10 pixels long, 8 of which must be read by both GPU threads in the basic kernel (see Fig. 4)a. For the whole 16×16 threads block a common image region is 32×32 pixels large. This region can be efficiently loaded in 4 stages into shared memory space (cf. Fig. 4b).

In every stage, a subregion of size 16×16 pixels is retrieved. The subregions are shifted relative to an image region associated to a thread-block. For example, in the case of subregion I, a thread reads a pixel intensity which is located 8 points to the left and 8 points above this thread dedicated pixel. After requests for the last subregion data are sent to global memory, there must be a synchronization barrier set so that transfers scheduled by all threads in a block complete before computations begin. Finally, a kernel proceeds from step 2 in the algorithm listed in Fig. 3. This time however, data requests induced in step 4 refer to shared and not global memory. In this way, there are only 4 instead of maximally 65 global memory reads.

Note, that the amount of shared memory that must be allocated for the region of size 32×32 is 4kB if the image pixels are described by 32-bit floating point numbers (the usual data type of MRI images). This volume fits the limit of 8kB per block determined by the execution configuration.

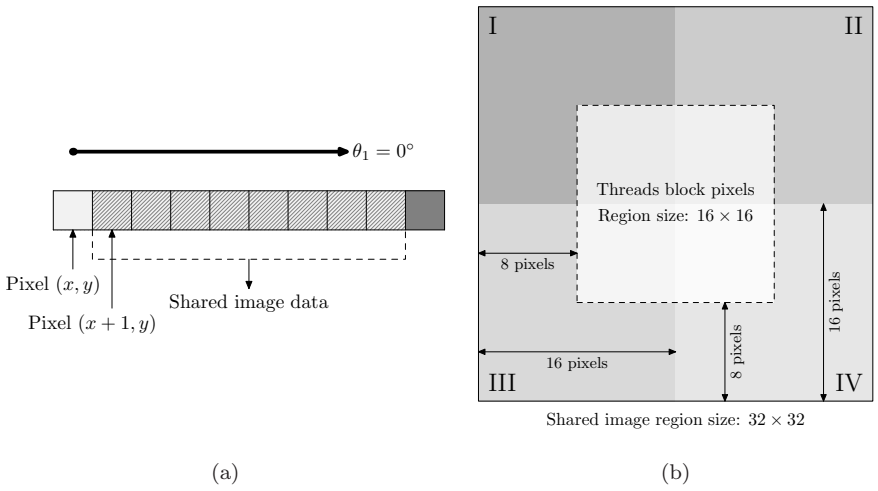

Fig. 4 Shared image data for two adjacent pixels in direction $\theta_1 = 0°$ (a) and for the whole thread block (b).

3.4 Increased thread responsibility

Additional mechanism allowing to improve kernel performance is to re-use the data once they are loaded from global memory. One can notice that an image region loaded into shared memory as shown in Fig. 4 already contains a substantial portion of information needed to calculate LPA masks for one of the neighboring thread blocks. Thus, to make profit of data re-use, every thread can be made responsible for two image pixels. This requires allocation of 6kB of shared memory space covering an image region of size 32×48 pixels. Hence, there are only 2 additional global memory reads (50% more) for a thread while the number of LPA masks calculated in a kernel doubles. The execution time of a single thread grows, but the number of threads that neeeds to be invoked reduces by half and this leads to considerable performance boost as reported in the next section.

4 Experiments

Efficiency of the proposed implementation was tested in a series of experiments performed on 10 2D simulated brain MR magnitude images[8]. Width and height of 2D slices—originally 181×217 pixels—were zero-padded to match the size of 256×256 pixels. One half of images was degraded by additive Gaussian noise of zero mean and variable standard deviation, i.e.

$\sigma = 0.01, 0.03, 0.05, 0.07$ and 0.15. The other half was corrupted with the Rician noise modeled as

$$p(m|X) = \frac{m}{\sigma^2} \exp^{-(m^2+X^2)/2\sigma^2} I_0 \left(\frac{Xm}{\sigma^2} \right) \tag{5}$$

where m denotes corrupted image pixel, X is the noise-free intensity of the pixel, and σ is the standard deviation of the underlying normally-distributed noise, which—in real conditions—adds to raw complex MR data. After Fourier transform of k-space, these data become Rician-distributed. Thus, in order to obtain noisy image from simulated brain phantom the following equation was applied to every image pixel

$$m = \sqrt{A^2 + B^2}, \tag{6}$$

where $A \sim N\big(X \cdot \cos(a), s^2\big)$ and $B \sim N\big(X \cdot \sin(a), s^2\big)$ are independent normal distributions (any real a). The parameter s can be treated as *noise level*, which in the conducted experiments was set to $s = \{1, 2, 3, 4, 5\}$. Measurements presented below are the average estimates obtained for all tested images. Fig. 5 presents example 2D images corrupted with Gaussian and Rician noise along with a sample result of noise-removal procedure accomplished using the designed implementation of LPA algorithm as part of the SA-DCT method.

The GPU code was run on the GF116-compliant GeForce GTX 560M chip. Since the graphics processor used in the experiments is targeted at mobile devices, also the CPU chip chosen for tests was a mobile variant of the Intel Core i7 (i7-2630QM). Time records viewed in Table 1 were measured using the CUDA Event API.

5 Results discussion and conclusions

Analysis of the obtained results shows superior performance of the GPU accelerated implementation of LPA over an analogous program run on CPU. In the latter case though, the time was measured for a single CPU thread. However, even if the score was divided by a factor of 8 (theoretical number of threads which can be simultaneously executed on i7 processor), GPU code

Table 1 Execution times of LPA kernels [ms] under various implementation designs

	CPU	Basic	Shared	Shared×2
Host-to-device data transfer	N/A		0.81	
LPA kernel	721.0	2.34	2.03	1.73
Device-to-host data transfer	N/A		4.53	

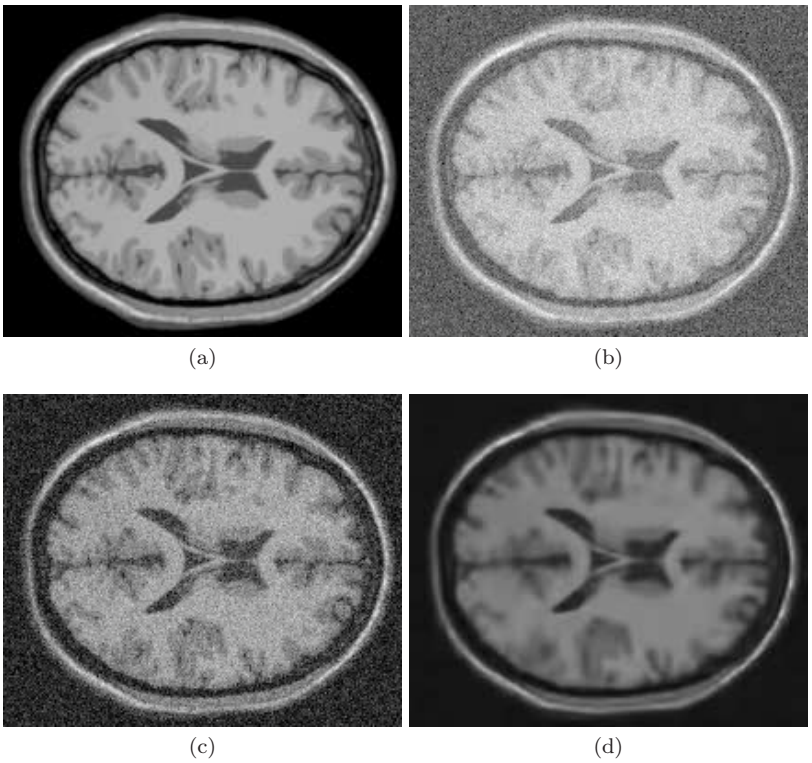

Fig. 5 Example 2D slice of synthetic brain image: (a) noise-free, (b) corrupted with Gaussian noise ($\sigma = 0.15$), (c) corrupted with Rician noise ($s = 5$), (d) denoised image (b).

runs on average and depending on the implementation variant 38 to 52 times faster. These ratios scale down to 11.7 and 12.7 if host-to-device and device-to-host data transfers are taken into account.

Moreover, it can be noticed how usage of shared memory speeds up computations. Execution time in the implementation variant described in Sect. 3.3 is 20% lower than in the case of the basic kernel. Increased responsibility of a kernel (this variant is denoted 'Shared×2' in Table 1) leads to even higher performance. Eventually, to test how the number of registers used by a thread affects the overall efficiency, the program (version 'Shared×2') was compiled using variable option `maxreg` in the CUDA `nvcc` compiler. Recall that excessive usage of registers—although may speed up a kernel execution—degrades the occupancy. However, as shown in Table 2, despite lower occupancy the optimum for the designed implementation is 29 registers per thread. Increasing the number of registers from 21 to 29 leads to observable improved efficiency. This trend halts only after the occupancy drops below 70%.

Table 2 Execution times of LPA kernels vs. number of registers used by a thread

No. of registers	21	23	25	27	29	31
Execution time [ms]	1.73	1.68	1.65	1.63	1.62	1.65
Occupancy [%]	100	92	85	79	73	69

To conclude, it must be underlined that the designed GPU-based implementation of the LPA algorithm performs very efficiently. Accomplished within a timeframe reduced to miliseconds, generation of LPA masks no longer entails any significant computational load to SA-DCT-based noise removal method.

Acknowledgements

This paper was supported by the Polish National Science Centre grant no. N N519 650940.

References

1. Bergmann, Ø., Christiansen, O., Lie, J., Lundervold, A. (2009) J. Digital Imaging 22(3):297–308
2. Chang, H.H. (2011) Rician noise removal in MR images using an adaptive trilateral filter. In: Proc. Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on, 467–471
3. Coupé, P., Manjón, J., Gedamu, E., Arnold, D., Robles, M., Collins, D.L. (2010) Medical Image Analysis 14(4):483–493
4. Garnier, S.J., Bilbro, G.L., Snyder, W.E., Gault, J.W. (1994) J. Digit. Imaging, 7(4):183–188
5. He, L., Greenshields, I.R. (2009) IEEE Trans. Medical Imaging 28(2):165–172
6. Katkovnik, V., Egiazarian, K., Astola, J. (2002) J. Math. Imaging and Vision 16:223–235
7. Katkovnik, V., Egiazarian, K., Astola, J. (2006) Local Approximation Techniques in Signal and Image Processing. SPIE Press
8. Kwan, R.S., Evans, A., Pike, G. (1999) IEEE Trans. Medical Imaging 11:1085–97
9. Sijbers, J., Dekker, J.d., Audekerke, J.V., Verhoye, M., Dyck, D.V. (1998) Magnetic Resonance Imaging 16(1):87–90