Accelerating Colonic Polyp Detection Using Commodity Graphics Hardware

David Williams, Valeriu Codreanu and Jos B.T.M. Roerdink
JBI Institute for Mathematics and Computer Science,
University of Groningen,
Groningen, The Netherlands
{d.p.williams, v.b.codreanu, j.b.t.m.roerdink}@rug.nl

Po Yang, Baoquan Liu and Feng Dong
Department of Computer Science and Technology,
University of Bedfordshire,
Bedfordshire, United Kingdom
{po.yang, baoquan.liu, feng.dong}@beds.ac.uk

Alessandro Chiarini
Super Computing Solutions,
Bologna, Italy
a.chiarini@scsitaly.com

***Abstract*—We present a parallel implementation of an algorithm for the detection of colonic polyps from CT data sets. This implementation is designed specifically to take advantage of the computational power available on modern *Graphics Processing Units* (GPUs), which significantly reduces the execution time to streamline the workflow of clinicians examining the data. We provide details about the changes which were made to the existing algorithm to suit the new target hardware, and perform tests which demonstrate that the results are a very close match to the reference implementation while being computed in a fraction of the time.**

Computer aided diagnosis, GPU computing, Colon screening;

I. Introduction

The increasing prevalence of Computer Aided Diagnosis (CAD) in the modern clinical environment can be attributed to the development of new and innovative algorithms for data analysis as well as the increasing availability of powerful computing hardware on which such algorithms can be run. With regards to the second point, the last few years have seen a rise in the use of *Graphics Processing Units* (GPUs) which expose a parallel programming model well suited to tasks such as image processing, computer vision and medical visualization. In this paper we investigate the application of such hardware to the process of identifying colonic polyps from abdominal CT data sets.

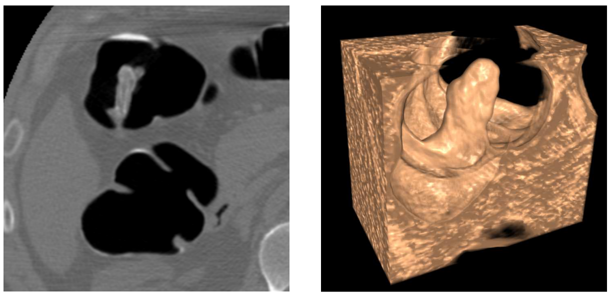

Figure 1. It is difficult to identify colon polyps from 2D slice data because the intersection of slices with the colon wall creates many false positives (a). This problem is avoided in 3D where it is easier to identify polyps by their distinct shape (b).

The identification of such polyps has been an active research area for a number of years, due to the difficulty in visually identifying colon polyps from 2D image slices (see Fig. 1(a)) and also the potential to prevent development of colon cancer through early diagnosis and intervention. This research has resulted in a number of approaches for automatically identifying potential colon polyps [1, 2, 3, 4] and visualizations which allow easy confirmation of the findings [5, 6].

Wide-spread adoption of such techniques is dependent on a quick and efficient workflow, which in turn requires fast processing of the input data. Existing systems take several minutes to perform this processing and this can be a significant portion of the time a clinician will typically spend analyzing a scan. Ideally, the automatic identification of colon polyps would take just a few seconds for a typical workstation and this will ease integration into CAD packages.

GPU hardware is already widely used for the visualization of medical data and so its extension to image analysis is a natural progression for algorithms which can be effectively mapped to its architecture. Not all algorithms fall under this category as they need to exhibit a high degree of parallelism to effectively utilize the GPU, but in optimal cases a speed–up of two or three orders of magnitude can be obtained. Fortunately, the polyp detection algorithm which we discuss shortly does indeed exhibit these characteristics and so can benefit greatly from the available hardware.

Our system for identifying colon polyps is described by [4] and is shown in Fig. 2. This system consists of a preprocessing step to remove noise, a number of stages for generating polyp candidates, and a Bayesian classifier for determining which candidates correspond to true polyps. Performance analysis has

The research leading to these results has received funding from the European Union's Seventh Framework Programme managed by REA-Research Executive Agency http://ec.europa.eu/research/rea (FP7/2007-2013) under grant agreement no. 286545

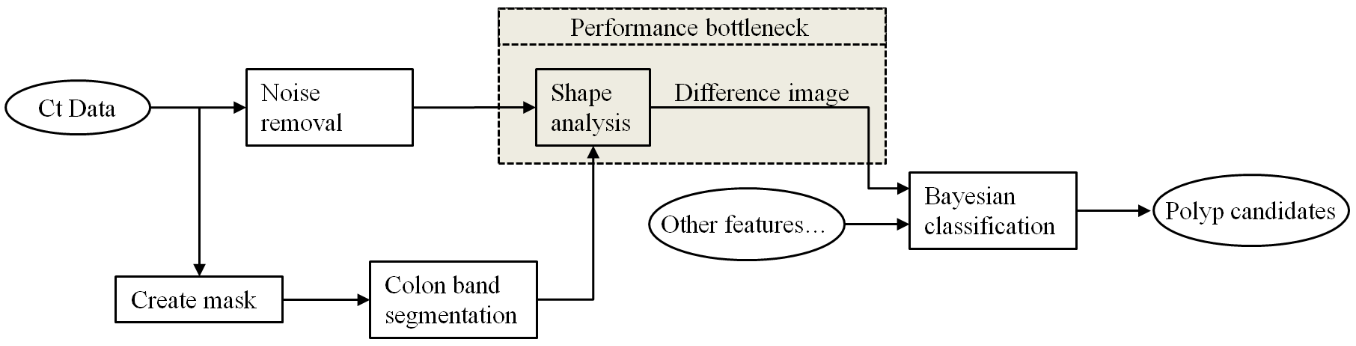

Figure 2. Flow diagram of the polyp detection system presented in [4]. Shape analysis has been identified as the performance bottleneck due to its high computational complexity. This is the part of the pipeline which has the most potential to benefit from GPU hardware.

revealed that calculation of shape information is the bottleneck which limits the overall speed of the system. It is here that we apply the computational power of GPU hardware by accessing it through the *Open Computing Language* (OpenCL) [7] programming model.

The remainder of this paper discusses previous work in the areas of computer aided detection of polyps and the application of GPU hardware to medical imaging. Section III provides an overview of the algorithm we are attempting to accelerate, and Section IV discusses how the algorithm was modified in order to take advantage of the GPU. Section V provides measurements of the performance and accuracy implications of this work. Lastly we provide a discussion of our findings and ideas for future research.

II. Related Work

Detection of colon polyps has long been an area of interest for the medical research community, with numerous approaches being developed over the years. The distinctive shape of colonic polyps provides the criterion which is most commonly used to identify them.

In [1] a two-stage approach is used which begins by applying a robust sphere fitting to locations on the surface of the colon wall. This is able to immediately identify approximately 90% of the colonic wall as being normal, and the second stage then uses more precise polyp models based on spherical harmonic decompositions to classify the remaining 10% of the surface.

An approach based on curvature analysis is presented by Summers et al. [2]. The curvature is computed from triangulated isosurfaces which are extracted from the data at three different threshold levels, and a set of additional filters are then used to eliminate false positives. The authors claim a sensitivity of 71% for polyps larger than 10mm.

Wijk et al. [3] also adopt the use of curvature analysis but compute parameters directly from the image data rather than relying on a surface extraction step for a particular isovalue. By performing a large number of iterations, each consisting of curvature analysis followed by polyp flattening, they are able to develop a system that works across a range of polyp sizes rather than being optimized for a single size. The work of Wijk et al. forms one component of the system described by Ye et al. [4] and is the basis for the work presented here.

While GPU hardware has not to our knowledge been applied to polyp detection in CT images it has been applied in a number of other areas related to medical imaging. Schiwietz et al. demonstrated how the reconstruction of cone bean CT data could be parallelized for implementation on commodity graphics hardware [8] and obtained a four times speed up over the fastest CPU approaches. This has been followed by work on other modalities [9, 10] with significant speedups also being reported.

GPUs have also been used to accelerate medical image processing tasks such as segmentation [11, 12], registration [13] and image analysis [14]. In all cases the tasks exhibit a high degree of parallelism which allows for an effective mapping to graphics hardware.

III. Algorithm Details

The algorithm which is used for shape analysis is described in detail by Wijk et al. [3] and we refer the reader to that paper for a full explanation of its operation. However, we provide a short overview here to provide the background necessary to understand why the algorithm can be effectively mapped to the GPU.

As shown previously by Fig. 2, the input to the algorithm is an abdominal CT scan that has been passed through a noise removal filter, and also a *colon band segmentation* volume which indicates which voxels of the data set are within a 1cm proximity of the colon wall. Shape analysis then consists of two main steps which are repeated a large number of times:

TABLE I. The positive second principal curvature of polyps can be used to seperate them from other structures.

Structure	**First principal curvature (k_1)**	**Second principal curvature (k_2)**
Colon wall	Negative	Zero
Folds in colon	Positive	Zero
Polyps	Positive	Positive

1) The principal curvatures are calculated for each voxel in order to determine the type of structure to which each voxel belongs. Table 1 shows that a positive second principal curvature indicates that a voxel may correspond to the surface of a polyp while a second principal curvature of zero means it is more likely to belong to another structure such as the colon wall or a fold.

2) The intensity of each voxel is next reduced by an amount proportional to its second principal curvature. This serves to gradually erode the polyps such that they are no longer present after a number of iterations.

This process is repeated until all potential polyps have been eroded, at which point a difference image can be computed between the original and eroded data sets. Regions of high difference then correspond to polyp candidates which are passed on to the classification stage of the pipeline.

IV. GPU Version

Modern GPUs provide a *Single Instruction Multiple Data* (SIMD) architecture which is well suited to calculations exhibiting a high degree of parallelism and which have minimal dependencies between tasks. Many image processing algorithms fall into this category and the curvature analysis outlined previously is no exception to this. The algorithm is appropriate for implementation on the GPU because:

1) Within any given iteration the results for a voxel do not depend on the results of any other voxels. Note however that dependencies do exist between successive iterations as the flattening must be performed before the curvature can be recalculated.

2) Memory accesses are highly localized. The curvature analysis requires the examination of a voxels' immediate neighbors but does not have to look any further than that. This property of the algorithm is well aligned with the ideal memory access pattern of a GPU, in which limited amounts of data can be cached for fast local access but global access is considerably slower.

3) The high computational complexity of curvature analysis means that the runtime of a *CPU* implementation is dominated by arithmetic operations rather than memory accesses. By contrast, *GPUs* excel at such number crunching tasks.

Our GPU implementation makes use of a pair of 3D images which are stored on the GPU for the duration of the execution. One of these images serves as a source and the other as a destination, with all the relevant data being loaded into the source image at the start. Curvature analysis and flattening are performed in a single OpenCL kernel that is launched by the CPU for each iteration of the algorithm. The source and destination are then swapped and the procedure is repeated the required number of times.

Experimental results have shown that all our data sets converge within 50 iterations, and so we use this fixed value to avoid pulling data back to the CPU to evaluate the finish criterion. Note that it would be possible to evaluate this criterion on the GPU such that the CPU would have no work to do for the duration of the algorithm, but this prevents the GPU handling rendering tasks which can impair responsiveness on single GPU systems.

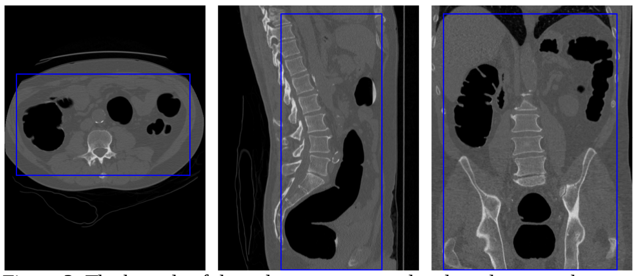
Figure 3. The bounds of the colon are computed and used to crop the extents of the volume. This allows the data to be reduced to 40% of its previous size.

Execution of the algorithm is split across a number of work groups as per the standard OpenCL execution model. In general these work groups benefit from being as large as possible, but the actual size is limited by the capabilities of the underlying hardware.

Although it was generally straightforward to implement the shape analysis on the GPU there are a few specifics which needed more careful consideration. These are discussed in the following subsections.

A. Memory Management

The large size of CT data sets is one of the primary challenges to implementing processing algorithms on the GPU. Our test datasets are all at a resolution of 512x512 pixels per slice, with between 395 and 472 slices in the complete dataset. A 16-bit integral data type is sufficient for representing the possible range of Hounsfield units in the input data but is not sufficient for storing intermediate results as the erosion is performed. For this we need to represent fractional values using the floating point types supported by graphics hardware. Full 32-bit floating point values (as defined by IEEE 754 [15]) are supported by the GPU and match the type being used by our reference CPU implementation. With this type the average size of our data sets is 433Mb, and storing two of these (source and destination) will limit the range of hardware on which the algorithm can be run.

There are a number of approaches which can be used to combat this, with one popular choice being the use of tiling by breaking the volume into pieces which are each small enough to fit in GPU memory. This solution is certainly applicable to our scenario but has drawbacks in terms of the additional complexity of managing the tiles as well as the overhead of accessing the neighbors of a voxel (needed for curvature analysis) which may actually lie in a different tile. Duplication of voxels along a tile border can help here but at the expense of further increased memory usage.

In our case it was possible to use the much simpler approach of cropping the extents of the volume data according to the size of the colon defined by the colon band segmentation computed previously. On average this reduced the memory consumption of the volume data to about 40% of its previous value (see Fig. 3). In addition to this, we allow the user to specify the precision at which data should be stored on the

GPU. Full 32-bit floating point precision is desirable to match the CPU reference implementation where possible, but a 16-bit floating point implementation is also available for environments in which memory is constrained.

The OpenCL API provides two approaches to storing volume data in compute device memory. The first approach is via *buffer objects* which, from a behavioral point of view, are analogous to arrays in traditional CPU programming. When set up appropriately these buffer objects can be read and written by both the CPU and the GPU and so provide an ideal basis for implementation of our algorithm.

The second storage approach provided by OpenCL is *image objects*. These provide similar functionality to buffer objects but are tailored specifically to take advantage of GPU texture hardware. Such hardware is widely used in the graphics pipeline, and OpenCL exposes this functionality to allow applications to benefit from texture caching, hardware interpolation of samples, and simplified handling of image boundary conditions.

Unfortunately, while OpenCL allows full read and write access to 1D and 2D image objects, the standard limits access to 3D image objects to *read only* which makes it impossible to implement the exact approach we described earlier. The '*cl_khr_3d_image_writes*' extension to the standard can be used to overcome this limitation but is not supported on all hardware [7]. We make use of this extension where available and otherwise fall back on using buffer objects for our volume data.

B. *Effective usage of the colon segmentation band*

Our CPU reference implementation makes use of a colon segmentation band derived from the input dataset. This specifies which voxels in the dataset are actually part of the colon wall, and therefore allows the expensive curvature and flattening calculations to be limited to these voxels. This is done by a simple conditional test at the start of the curvature calculation code.

However, conditional logic has a somewhat different behavior on GPU hardware due to its highly parallel architecture. The main constraint is that all *work items* which comprise a particular *work group* should follow the same execution path, and in the case of divergent behavior the execution time for the work group is dominated by the slowest work item.

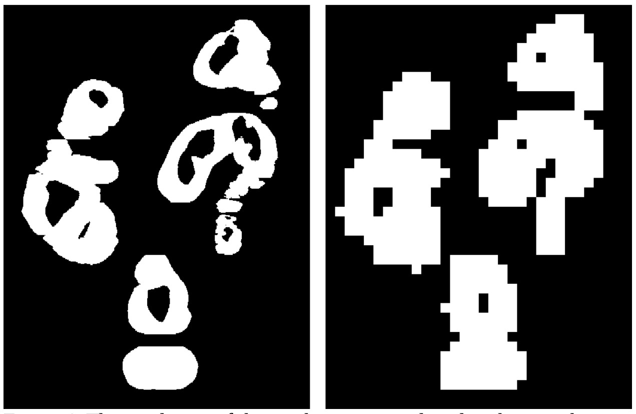
Figure 4. The resolution of the mask image is reduced in this case by a factor of 16. Note that for visualization purposes the image on the right has then been scaled up by a factor of 16, but a significant reduction in memory is obtained.

Given this behavior it is no longer beneficial to store a colon segmentation band at the same resolution as the underlying data, and the resolution can instead be reduced such that there is only one sample for each work group. This has the additional benefit of decreasing the storage requirement for the segmentation data which is desirable in our memory constrained scenario.

The original segmentation is a binary image, and each voxel of the low resolution version is computed by finding the maximum of all corresponding voxels in the high resolution image. This ensures that the low resolution segmentation covers at least the voxels which are set in the original mask. This is illustrated by Fig. 4.

C. *Precision*

Our CPU reference implementation of the shape analysis algorithm makes use of 32-bit floating point values for storage of both the CT volume data and intermediate images, and 64-bit floating point values for some intermediate calculations. It is desirable to match the output of our GPU implementation to this reference but some practical considerations make this difficult.

Firstly, the high memory usage described previously means that it is not always possible to store the data at full precision. Most recent graphics cards will indeed have the required memory for this, but we have also implemented a fallback which allows lower-end systems to store the data as 16-bit half precision values instead. This is typically slower, as the GPU performs calculations only on 32-bit floats and so data has to be converted as it is loaded to and from memory.

Another consideration is that most graphics hardware does not support operations on 64-bit floating point types, and even when these types are supported the performance is typically several times lower. This situation is expected to improve in the future through the availability of the OpenCL '*cl_khr_fp64*' extension [7], but for now our implementation is limited to 32-bits for intermediate values.

The impact of these precision concessions is measured in our tests in the following section.

V. RESULTS

Performance and accuracy of our implementations have been tested on two separate machines to analyze the benefits on both high-end and low-end hardware. Our high-end hardware is a desktop PC containing an Intel Quad-Core i7-2600 CPU running at 3.40GHz and equipped with an Nvidia GeForce GTX 590. Note that the GTX 590 is a dual GPU board but we are only making use of a single GPU for our tests. Our low-end hardware is a small laptop containing an AMD Dual-Core C-60 processor and an integrated Radeon HD 6290 graphics card.

We have two implementations of our algorithm to test. The first is the existing CPU implementation which runs on a single

TABLE II. INTEL QUAD-CORE I7-2600 CPU @ 3.40GHZ + NVIDIA GEFORCE GTX 590

	Full precision			**Half precision**		
	Time (s)	***Error (RMS)***	***Error (max)***	***Time (s)***	***Error (RMS)***	***Error (max)***
Reference	58.60	0.0	0.0	60.90	1.710×10^{-4}	0. 145
Multi-core	13.82	2.257×10^{-4}	0.290	31.45	2.585×10^{-4}	0.290
GPU+buffers	1.34	2.298×10^{-4}	0.290	1.22	2.563×10^{-4}	0.290
GPU+images	-	-	-	-	-	-

TABLE III. AMD DUAL-CORE C-60 CPU + INTEGRATED RADEON HD 6290

	Full precision			**Half precision**		
	Time (s)	***Error (RMS)***	***Error (max)***	***Time (s)***	***Error (RMS)***	***Error (max)***
Reference	295.01	0.0	0.0	325.13	1.710×10^{-4}	0.145
Multi-core	352.10	2.313×10^{-4}	0.290	517.24	2.592×10^{-4}	0.290
GPU+buffers	-	-	-	50.32	2.595×10^{-4}	0.290
GPU+images	47.70	2.306×10^{-4}	0.290	47.89	4.313×10^{-4}	0.290

CPU core and produces results which have been verified across a large number of data sets. This provides our reference implementation, and we expect our new GPU implementation to exceed this in performance whilst giving an output that is as close as possible to the reference.

The second is our new OpenCL implementation as described in Section IV. OpenCL is a programming model for general parallel computing hardware and so our test can be run on both the GPU and the multi-core CPU present in each machine.

We also have some additional variations on the algorithm. Section IV discussed the merits of utilizing *image objects* instead of *buffer objects* when the required OpenCL extension is available. The NVidia GPU does not support this extension, but curiously our low-end AMD GPU does support it despite being inferior to the NVidia GPU in almost every other way. This provides us with an opportunity to test the impact of this particular feature.

Lastly, we also test our algorithm using both full precision (32-bit) floating point storage and half-precision (16-bit) floating point storage. In both cases we compute the root mean square and maximum error metrics from our reference implementation. The results of all these tests are given in Tables 2 and 3.

Execution times for all variations of the algorithm are given in Tables 2 and 3 for both high-end and low-end test configurations. Note that times are inclusive of OpenCL data transfer where applicable. Timings for ‘GPU+Images’ are not provided for the NVidia hardware because it did not support the required OpenCL extension, and timings for full precision ‘GPU+buffers’ are not provided for AMD hardware as memory limitations prevented the algorithm from running.

VI. DISCUSSION

There are a number of properties of these results which are worth discussing further.

Firstly, it can be observed that the reference implementation produces consistent results across both test configurations. This is the expected behavior but worth verifying as it helps validate the results of the other comparisons. The full-precision calculations can be seen to execute slightly faster than the half-precision calculations which is due to the lack of native half-precision support in modern CPUs. In both cases the desktop configuration outperformed the laptop by a factor of about five.

Next, we can observe that the GPU implementation is always faster than both the reference and the multi-core CPU implementation. This is again expected as the promised performance gains were the main motivation behind bringing this algorithm to the GPU. The greatest performance increase (about 50x) is seen on the NVidia hardware when working with the half precision data, while the laptop gives a more modest increase of approximately 6.5x. In both cases the results justify our efforts to parallelize the algorithm for the GPU.

Results for the OpenCL implementation running on multi-core CPUs were more mixed. The full precision desktop test did indeed give the expected 4x performance boost for a quad-core processor, but when operating at half precision it only gave a 2x performance increase. As mentioned earlier, half precision is not natively supported by CPUs and it seems the OpenCL emulation is not particularly efficient in this regard. More surprisingly, the multi-core OpenCL implementation was actually slower than the reference implementation on the low-end AMD hardware. The reasons for this are not entirely apparent, but further research has revealed that other benchmarks [16] have obtained similar results.

There is a general trend for the full-precision implementations to be faster than their half-precision counterparts due to native hardware support for the IEEE 754 standard. The exception to this is the NVidia GPU result which shows a 10% performance increase for the half-precision version. This is most likely due to the high processing power of this GPU causing memory access to become a bottleneck rather than arithmetic operations, at which point the smaller size of the half-precision type brings cache benefits which outweigh the additional processing time.

The benefits of using image objects rather than buffer objects are visible in the half-precision results on the laptop, but at only a 6% performance increase, which was not as great as we had hoped. However, the image objects did bring an additional surprise benefit for full-precision, as the buffer objects were not usable in this scenario due to memory constraints. This limitation appears to arise from the integrated nature of this GPU, which has only a limited amount of dedicated memory and relies on memory shared with the CPU for most of its operations.

Looking at the error metrics we can observe the expected behavior of RMS error being slightly higher for the half-precision calculations than for the full-precision calculations. The maximum error of 0.29 is only 0.028% of the entire 0-1040 output range, while the RMS error is significantly lower than this.

VII. Future Work

The performance benefits demonstrated in this paper open the door to a number of further opportunities. One avenue of investigation is to sacrifice some of the newly obtained performance and spend time improving the behavior of the algorithm, for example by applying a more robust noise reduction algorithm [17, 18] prior to processing the data. It is possible that such improvements will offset the precision differences seen earlier.

Another opportunity is to improve the curvature analysis by implementing a larger kernel which looks beyond the immediate neighbors of a given voxel [19]. It may be possible to utilize the image interpolation hardware present in GPUs to reduce the number of image samples which are required for this [20] and doing so would provide an additional benefit to using OpenCL image objects for the data storage.

It may also be interesting to investigate executing the algorithm across multiple GPUs to obtain even greater speedups. The challenge here is determining the best approach to splitting the memory and ensuring that results are synchronized between the GPUs involved.

Lastly, the work presented here will be used for the evaluation of our research on *automatic* parallelization of algorithms for the GPU. This aims to simplify the process of utilizing GPU hardware and our results so far will serve as a reference implementation for this.

References

[1] G. Kiss, J. van Cleynenbreugel, S. Drisis, D. Bielen, G. Marchal, and P. Suetens, “Computer-aided detection for low-dose CT colonography,” Proceeding of MICCAI’05, pp. 859–67, 2005.

[2] R. M. Summers, C. D. Johnson, L. M. Pusanik, J. D. Malley, A. M. Youssef, and J. E. Reed, “Automated polyp detection at CT colonography: Feasibility assessment in a human population,” Radiology, vol. 219, no. 1, pp. 51–9, 2001.

[3] C. van Wijk, V. F. van Ravesteijn, F. M. Vos, and L. J. van Vliet, “Detection and segmentation of colonic polyps on implicit isosurfaces by second principal curvature flow”, IEEE Transactions on Medical Imaging, pp. 688–696, 2010.

[4] Y. Xujiong and G. Slabaugh, “A model-driven Bayesian method for polyp detection and false positive suppression in CT colonography Computer–Aided detection”, Machine Learning in Computer-Aided Diagnosis: Medical Imaging Intelligence and Analysis, pp. 220–237, 2012.

[5] L. Hong, A. Kaufman, Y. Wei, A. Viswambharen, M. Wax and Z. Liang, "3D virtual colonoscopy", in Proceedings of Biomedical Visualization, pp. 26-32, 1995.

[6] D. Williams, S. Grimm, E. Coto, A. Roudsari and H. Hatzakis, "Volumetric Curved Planar Reformation for Virtual Endoscopy", IEEE Transactions on Visualization and Computer Graphics, Vol. 14, No. 1, pp. 109–119, 2008.

[7] Khronos OpenCL Working Group, “The OpenCL Specification”, Ver 1.1, Rev. 44.

[8] T. Schiwietz, S. Bose, J. Maltz, R. Westermann, "A Fast And High-Quality Cone Beam Reconstruction Pipeline Using The GPU", In Proceedings of SPIE Medical Imaging, Vol. 6510, pp. 1279–1290, 2007.

[9] W. Xu, F. Xu, M. Jones, B. Keszthelyi, J. Sedat, D. Agard and K. Mueller, "High-Performance Iterative Electron Tomography Reconstruction with Long-Object Compensation using Graphics Processing Units (GPUs)", Journal of Structural Biology, pp. 142-153, 2010.

[10] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, B. P. Sutton and Z. P. Liang, "Accelerating advanced MRI reconstructions on GPUs", Journal of Parallel and Distributed Computing, pp. 1307–1318, 2008.

[11] M. Rumpf and R. Strzodka, “Level set segmentation in graphics hardware,” in Proceedings of IEEE International Conference on Image Processing, Vol. 3, pp. 1103–1106, 2001.

[12] A. Sherbondy, M. Houston, and S. Napel, “Fast volume segmentation with simultaneous visualization using programmable graphics hardware” in Proceedings of the IEEE Visualization, pp. 171–176, 2003.

[13] R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley, “A survey of medical image registration on multicore and the GPU,” IEEE Signal Processing Magazine 27(2), pp. 50–60, 2010.

[14] S. Kockara, T. Halic and C. Bayrak, "Real-time Minute Change Detection on GPU for Cellular and Remote Sensor Imaging", International Conference on Advanced Information Networking and Applications Workshops, pp. 13–18, 2009.

[15] IEEE Computer Society, "IEEE Standard for Floating-Point Arithmetic", 2008

[16] SiSoftware Limited, “Benchmarking : OpenCL CPU Performance (OpenCL vs native/Java/.Net)”, accessed 21 October 2012, http://www.sisoftware.net/?d=qa&f=gpgpu_cpu_perf

[17] S. Manay, A. Yezzi, "Anti-geometric diffusion for adaptive thresholding and segmentation, in Proceedings of IEEE International conference on Image Processing, Vol 2, pp 829-832, 2001

[18] M.K. Kalra et al, “Detection and Characterization of Lesions on Low-Radiation-Dose Abdominal CT Images Postprocessed with Noise Reduction Filters”, Radiology, Vol. 232, Nr. 3, pp. 791-797, 2004

[19] G. Kindlmann, R. Whitaker, T. Tasdizen and T. Moller, “Curvature-Based Transfer Functions for Direct Volume Rendering: Methods and Applications”, in Proceedings of the IEEE Visualization, pp 67-74, 2003

[20] C. Sigg, M. Hadwiger, “Fast third-order texture filtering”, in GPU Gems 2, Chapter 20, pp. 313-329, 2005