GPU-accelerated Convex Multi-phase Image Segmentation

Subrahmanyam Gorthi[1], Arnaud Le Carvennec[2], Hadrien Copponnex[1],
Xavier Bresson[3], Jean-Philippe Thiran[1]
[1]Signal Processing Laboratory (LTS5),
Ecole Polytechnique Fédérale de Lausanne(EPFL), Switzerland.
[2]ENS Cachan-antenne de Bretagne, France.
[3]Department of Computer Science, City University of Hong Kong, Hong Kong.
subrahmanyam.gorthi@epfl.ch

Abstract

Image segmentation is a key area of research in computer vision. Recent advances facilitated reformulation of the non-convex multi-phase segmentation problem as a convex optimization problem (see for example [2, 4, 9, 10, 13, 16]). Recently, [3] proposed a new convex relaxation approach for a class of vector-valued minimization problems, and this approach is directly applicable to the widely used classical Mumford-Shah segmentation model [11]. While the approach in [3] provides the much deserved convexification, it achieves this at the expense of an increased computational complexity due to the increased dimensionality of the reformulated problem; however, the algorithm proposed in [3] can indeed profit from a parallelized implementation. In this paper, we present a GPU-based implementation of the convex formulation for Mumford-Shah piecewise constant multi-phase image segmentation algorithm proposed in [3]. The main goal of this paper is to provide insights into the way the algorithm has been parallelized in order to obtain good speedup. We present multi-phase segmentation results both on synthetic and real images. The speedup of GPU-based implementation is evaluated on three different GPUs. For sufficiently large images, the speedup achieved on GTX 285 GPU is around 40, compared to an optimized CPU-implementation. The speedups obtained from GPU-based implementation are quite satisfactory. We also made our CUDA code available online[1].

1. Introduction

Image segmentation is one of the fundamental problems in computer vision. It aims at partitioning a given image into meaningful regions. There exist many approaches and algorithms to solve the segmentation problem. Many of the existing algorithms, however, lead to non-convex optimization problems. It means that the solution to which those algorithms converge is dependent on the initialization of the segmentation; thus, convergence to a global solution is not guaranteed in such cases.

There are significant developments in recent years in reformulating the original non-convex multi-phase segmentation problem as a convex optimization problem [2, 3, 4, 5, 9, 10, 13, 16]. The method proposed in [3], which is based on [7, 13], achieved the goal of convexifying the multi-phase region-based segmentation problem, by reformulating the original problem in a higher dimensional space to remove the non-convexity. The method proposed by [13] is analogous in the continuous setting to the approach of [8] that was proposed for the discrete Markov random field (MRF) setting. [3, 7] generalized the approach of [13] to vector-valued problems, i.e., where the unknown to be computed can be vector-valued rather than scalar-valued.

As we mentioned, while the approach in [3] provides a convex formulation for region-based multi-phase segmentation, the computational complexity is increased due to the increased dimensionality. For example, with the new formulation in [3], for an N-dimensional image segmentation, the dimensionality increases by N. However, many computations of this algorithm can indeed be parallelized, and thus, can profit from the GPU-based implementation.

In this paper, we present and analyze GPU-based implementation of the algorithm in [3] for image segmentation. In particular, we deal with region-based, piecewise constant, multi-phase segmentation model based on the classical Mumford-Shah model [11]. The rest of the paper is organized as follows: In Section 2, we briefly describe convex algorithm of [3]. In Section 3, we present the details of parallel implementation of this algorithm on GPU, with CUDA. In Section 4, we present segmentation results on synthetic and real 2-D images, and evaluate the speedups

[1]The GPU-based CUDA code can be downloaded from http://lts5srv2.epfl.ch/~gorthi/software.html

on different GPUs; we also perform a study of the effect of size of the image on the speedup, and other relevant studies. Finally, conclusions are presented in Section 5.

2. Convex multi-phase segmentation algorithm

The segmentation method used here is based on the Mumford-Shah model [11]. This model approximates the image to be segmented with piecewise smooth regions, and the functional that it minimizes is given by:

$$\inf_{f,K} \int_{\Omega} (f(x) - I(x))^2 dx + \upsilon \int_{\Omega \setminus K} |\nabla f(x)|^2 dx + \mu |K|,$$

where I is the input image to be segmented, f is the output segmented image, K is a closed edge set ($K \subset \Omega$) that partitions the image into different phases, and $|K|$ is the length of the boundary of K; υ, μ are weighting parameters ($\upsilon \geq 0$, $\mu \geq 0$) for the last two terms of the above functional. With the further assumption (as in [6]) that the image to be segmented has piecewise constant regions (i.e.,$\nabla f(x) = 0$), the above equation simplifies to:

$$\inf_{c_i,K} \sum_i \int_{\Omega_i} (c_i - I(x))^2 dx + \mu |K|,$$

where c_i is the mean value of the intensity for the ith phase, and Ω_i is the region in the output image corresponding to the ith phase (with $\Omega = \cup_i \Omega_i$).

It is often assumed that the number of phases is fixed ($= n$), and the corresponding optimal mean intensity values c_i are known apriori. This reduces the above equation to:

$$\inf_{\Omega_0,\cdots,\Omega_{n-1}} \sum_{i=0}^{n-1} \int_{\Omega_i} (c_i - I(x))^2 dx + \frac{\mu}{2} |\partial \Omega_i|.$$

With the level set representation proposed in [15], two level set functions are sufficient to represent four phases; extension to even higher number of phases can be done with just those two level set functions, by using the four color theorem, as shown in [15]. Hence, without loss of generality, we consider here a four-phase segmentation problem. The above equation can be rewritten as follows, in terms of the two level set functions (denoted by $u_1(x)$ and $u_2(x)$):

$$\min_{\vec{u}:\Omega \to \{0,1\}^2} \left\{ \int_{\Omega} \rho(x, u_1(x), u_2(x)) dx + \sum_{i=1}^{2} |\nabla u_i| dx \right\},$$

where

$$\begin{aligned}
&\rho(x, u_1(x), u_2(x)) = (c_0 - I(x))^2 \, u_1(x) \, u_2(x) \\
&+ (c_1 - I(x))^2 \, u_1(x) \, (1 - u_2(x)) \\
&+ (c_2 - I(x))^2 \, (1 - u_1(x)) \, u_2(x) \\
&+ (c_3 - I(x))^2 \, (1 - u_1(x)) \, (1 - u_2(x)).
\end{aligned}$$

The method proposed in [3] reformulated the above non-convex functional, in a higher dimensional space, to an equivalent convex relaxation method. Since the goal of this paper is to present the details of the GPU-based implementation, we mention here only the final equations; we refer the readers to [3] for the details of the reformulation. The final set of equations that are to be solved iteratively are given by:

$$\begin{aligned}
(\vec{p})^{n+1} &= \Pi_{D^h}((\vec{p})^n + \tau_p A(\overline{\phi^h})^n) \\
(\phi^h)^{n+1} &= \Pi_{C^h}((\phi^h)^n - \tau_\phi A^* (\vec{p})^{n+1}) \\
(\overline{\phi^h})^{n+1} &= 2\,(\phi^h)^{n+1} - (\phi^h)^n, \qquad (1)
\end{aligned}$$

where A is a linear gradient operator, A^* is the adjoint of A, and Π_{D^h} is the projection given by:

$$\begin{aligned}
\Pi_{D^h}(\vec{p}) = (&\frac{p_1}{\max(|(p_1,\cdots,p_m)|,1)}, \\
&\cdots, \frac{p_m}{\max(|(p_1,\cdots,p_m)|,1)}, \ \min(p_\gamma, \rho)).
\end{aligned}$$

$\Pi_{C^h}(\phi^h)$ is truncation of ϕ^h to [0, 1] interval. τ_p, τ_ϕ are constants related to the spatial step sizes, and ϕ^h is the reformulated version of the function to be minimized, defined in [3].

Since the reformulated algorithm is convex, the final solution does not depend on the initialization. Hence, we can start with any arbitrary initialization for each phase. We use the following convergence condition for the iterative process:

$$||(\phi^h)^k - (\phi^h)^{k-1}||_{\ell 2} < \epsilon,$$

where k is the number of iterations, and ϵ is the convergence parameter. From our experiments on a set of images, we noticed that $\epsilon = 0.001$ is good enough for obtaining stable segmentation results.

3. Implementation on GPU with CUDA

In order to perform a GPU-based implementation of any algorithm, the primary step is to identify the portions of the algorithm that can be processed in parallel, and then mapping the algorithm to CUDA (or other similar) programming environment. Each of the data-parallel portion can be run on separate CUDA kernels. Based on the dependencies that exist among various kernels (as we shall see in more detail shortly), a synchronization step may be required before switching from one kernel to the other. In this Section, we first mention the tasks that we parallelize in the algorithm of [3], and followed by the important details on the memory related optimizations; the memory optimizations are found to be critical in improving the speedup of the GPU-based implementation.

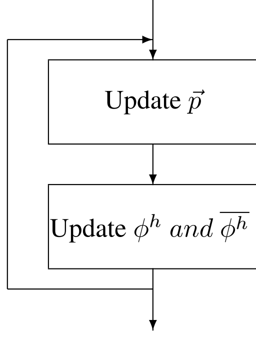

Figure 1: Control flow of the kernels for the CUDA implementation on the GPU.

Regarding the dependencies that exist among the variables: $\vec{p}$, ϕ^h, $\overline{\phi^h}$ at a given iteration, we make the following observations from equation-set (1). Notice that, at iteration number $(n+1)$, the computation of $\vec{p}$ depends only on ϕ^h, $\overline{\phi^h}$ that are already computed during the previous iteration (n); also note that $(\vec{p})^{n+1}$ value for each pixel can be calculated in parallel, since it's computation does not depend on $(\vec{p})^{n+1}$ values of neighboring pixels.

Similarly, note that the computation of $(\phi^h)^{n+1}$ at a given pixel, requires $(\vec{p})^{n+1}$ values at the same pixel, as well as at the pixels in it's neighborhood. So, when performing the computations at each pixel in parallel, before starting the computation of $(\phi^h)^{n+1}$ at a pixel, we need to make sure that the computation of $(\vec{p})^{n+1}$ is completed for all it's neighboring pixels also; thus, an overall synchronization step is essential for $\vec{p}$, before starting the computation of ϕ^h.

Regarding $(\overline{\phi^h})^{n+1}$ value at each pixel, note that it's computation requires only $(\phi^h)^{n+1}$ values at that pixel, and thus, does not depend at all on any other neighborhood pixel values. So, even if each pixel is processed in parallel, as long as ϕ^h and $\overline{\phi^h}$ are computed in sequence for individual pixels, there is no need for any overall synchronization step between ϕ^h and $\overline{\phi^h}$ (unlike the previous case between $\vec{p}$ and ϕ^h).

Based on the above mentioned observations, the entire algorithm is divided into two CUDA *kernels*[2], as illustrated in Figure 1. The first kernel updates $\vec{p}$ values, and then, after a synchronization step, the second kernel updates both ϕ^h and $\overline{\phi^h}$ values; the implementation iterates through these two kernels until the convergence criteria (mentioned in the preceding Section) is achieved.

We implemented the above mentioned algorithm with NVIDIA's GPGPU (General Purpose computing on Graphics Processing Unit) programming unit, CUDA, version-3.1 [1]. We noticed that the most critical task for the GPU-based implementation of the current algorithm is memory access. We now present in detail, the main memory related optimizations that we implemented, which resulted in significantly improving the memory access timings.

The first optimization that we do is related to accessing the global memory space on the GPU. It is important to follow the right access pattern in order to get the maximum memory bandwidth, especially given how costly accesses to device memory are. According to CUDA specifications [1], if a certain set of access requirements (mentioned below) are met, the global memory access by all threads of a *half-warp*[3] is coalesced into a single memory transaction; it means that, when those specific requirements are met, the global memory access transactions by all the 16 parallel threads (i.e., half-warp) can be coalesced into just a single memory transaction (instead of the original 16 transactions), and this enhances the throughput significantly. The exact conditions to be met for this purpose are as follows:

(i) All the 16 words accessed by the half-warp must lie in the same segment of the GPU's global memory, and they must access the words in sequence (i.e., k^{th} thread of a half-warp should access only the kth word in the memory segment); however, not all threads need to participate.

(ii) The starting memory address of the segment should be a multiple of 16.

(iii) The size of the words accessed by the threads must be same, and should be of 4, 8, or 16 bytes.

- (a) If the size of the word is 4-bytes, all 16 words must lie in the same 64-byte contiguous segment.
- (b) If the size is 8-bytes, all 16 words must lie in the same 128-byte contiguous segment.
- (c) If the size is 16-bytes, the first 8 words must lie in the same 128-byte segment, and the last 8 words in any of the following 128-byte segment.

We now illustrate the above conditions in more detail, with the help of some examples presented in Figure 2. The left side figure shows two examples of memory access patterns that result in just a single memory transaction instead of usual 16 transactions; note that the data type used in them is 32-bit float, and hence, the size of the words accessed by the threads is 4-bytes (condition-(iii)); the memory location of the segment accessed by the half-warp started at the address: 112, which is a multiple of 16 (condition-(ii)). In the example 2(a), each kth thread is accessing k^{th} word

[2] A CUDA *kernel* is a function, that, when called, is executed N times in parallel by N different threads, as opposed to only once like regular C-functions [1].

[3] The process of managing, scheduling and executing threads in a group of 32 parallel threads is called warps; a half-warp is either the first or second half of a warp [1].

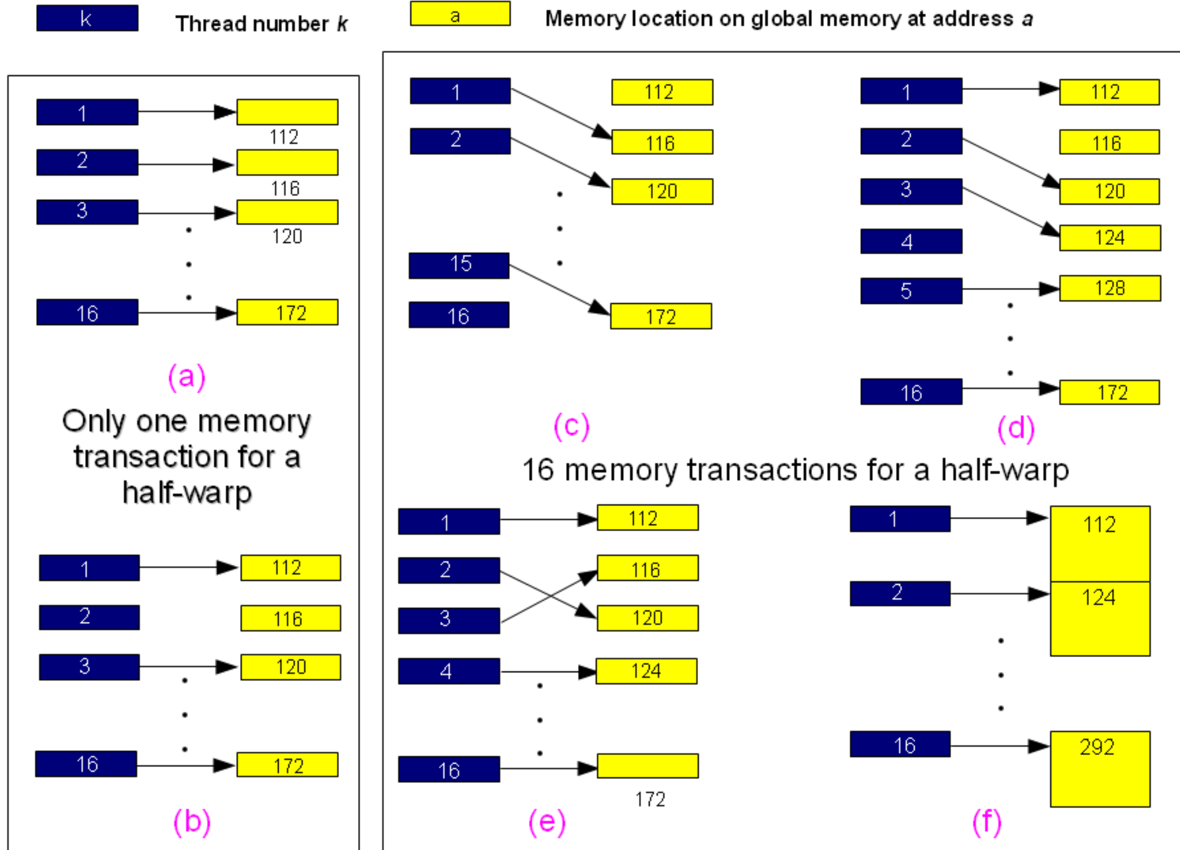

Figure 2: The left side figure illustrates two types of memory access patterns that effectively result in a single memory transaction, and thereby, significantly improves the overall memory access time. On the contrary, the right side figure illustrates various memory access patterns that cannot profit from such single memory transaction.

in the memory, while in the example 2(b), although some threads do not participate, rest of the threads still satisfy the condition-(i) by accessing only their respective k^{th} words.

On the contrary, the sub-figures in the right side of Figure 2 (c–f) show examples of memory access patterns that fail to satisfy one of the three conditions mentioned above, and thereby, could not profit from such single memory transaction[4]. It can be noted that, in the pattern shown Figure 2(c), the starting memory address is not a multiple of 16, and thus violates the condition-(ii); Figure 2(d) and 2(e) show patterns that violate the condition-(i) regarding the kth thread accessing the kth word; finally, Figure 2(e) shows a pattern for which the size of the words accessed is not 4, or 8, or 16 bytes, and thus violates the condition-(iii).

In our current implementation, in order to profit from the above mentioned single memory transaction for each half-warp, we do the following: The input 2-D image is represented as a 1-D array by concatenating the pixels column-wise, and each column of the image is zero-padded with an appropriate memory space such that every column starts at an address that is a multiple of 16; when the original column width is not a multiple of 16, the new column width after padding (denoted by N_y^{pad}) is given by:

$$N_y^{\mathrm{pad}} = \left(\mathrm{floor}\left(\frac{N_y}{16}\right) + 1\right) 16,$$

where N_y is the number of pixels in each column. Regarding the dimensions of the grid of thread blocks, we have chosen a dimension of 1 for rows, while the number of columns is chosen empirically with the constraint that the number of columns is a multiple of 32, and at least equal to the number of cores of the GPU; this constraint is based on the recommendations given in CUDA documentation.

It is easy to notice that with the above mentioned configurations, each half-warp of threads *always* contains contiguous pixels from a single column; further, since this con-

[4] We would like to make a note to the readers that the coalesced memory transactions described here are based on version-3.1 of CUDA. In the recently released versions, there are slight modifications to it. In particular, coalesced memory transactions can be carried out for a warp instead of a half-warp, and the conditions to be satisfied for a single memory transaction are also made more flexible.

figuration also satisfies the conditions mentioned earlier, the access to each of these columns in the relevant variables can be made with just a single memory transaction.

To further improve the memory access time, we use the fact that some components of $\overrightarrow{p}$ are used at the same time, for the same kind of memory operations (reading or writing). Notice that for 2-D images, $\overrightarrow{p}$ in equation (1) has three terms, and the operations on the first two terms of $\overrightarrow{p}$ are similar. Because of this reason, instead of representing these two terms separately, we represent them using a built-in vector type of CUDA, called Float2, which is a structure of floats; with this kind of representation, the number of coalesced memory transactions are halved compared to representing the two terms separately. Note that instead of just two terms, although all the three terms of $\overrightarrow{p}$ could be represented as a single built-in structure of CUDA, called Float3, it will not however lead to any improvements; this because, the size of Float3 structure is 12 bytes, and hence, this will not satisfy the above mentioned condition-(iii), which is essential to benefit from coalesced memory transactions. Finally, we also use texture memory wherever applicable, since the memory read operations are faster with texture-memory than with the global memory of the GPU.

4. Results

We first present segmentation results obtained from the convex multi-phase segmentation algorithm, followed by a detailed evaluation of the GPU-based implementation. We present here results for 4-phase image segmentation; as mentioned earlier, this can be extended to any higher number of phases using approaches like four color theorem [15].

The segmentation results are presented for a synthetic image and a real image. The synthetic image used here is similar to the image from the work of [15] with noise. The second image is an axial slice extracted from the Magnetic Resonance (MR) image of the human brain. Each pixel of the MR image is segmented into one of the four regions, viz., (i) gray matter (GM), (ii) white matter (WM), (iii) cerebrospinal fluid (CSF) or (iv) background. For these two images, the mean intensity values of each region are known in prior. In case these values are unknown, a standard approach of alternating the estimation of mean intensity values and region estimates can be used [6, 15].

The input images to be segmented are shown in the first column of Figure 3. The middle column shows initializations made for the respective images. For an easy visualization, each phase is represented in different color. The last column shows the output segmentations. The segmentation results are found to be independent of the initialization.

For evaluating the improvement in the computational time from the serial implementation to the GPU-based implementation, we use the quantitative metric: "Speedup",

Table 1: The details of the CPU and GPUs used in the evaluation are described in this Table.

Model	Number of cores used
CPU (Work Station)	
Intel Xeon E5520	1
GPUs (NVIDIA)	
(1) GT 9800	112
(2) GTX 260	216
(3) GTX 285	240

and it is defined as follows:

$$\text{Speedup} = \frac{\text{Time taken on CPU}}{\text{Time taken on GPU}}.$$

Note that the time taken for GPU also includes the data transfer times for copying data from CPU to GPU memory, as well as from GPU to CPU memory. The serial implementation is run on one of the 4 CPU cores, dedicated exclusively for this task. Notice that, in general, if the speedup achieved is around $2-3$ times only, then it is not considered worth enough to go for a GPU-based implementation. The details of CPU and GPUs that we use for the evaluation are presented in Table 1. We first study the effect of size of the input image to be segmented on the speedup. For this purpose, we fixed the number of iterations to 300. Figure 4 shows the plots of time-taken (in seconds) versus "size of the image". We notice that the time taken for both serial implementation on CPU, and parallel implementation on GPUs scale approximately linearly with the image size. The corresponding speedups are presented in Figure 5. We notice that for sufficiently large images (approximately 1024×1024 size or more), the speedup achieved with GTX 285 is close to 40, and this is quite satisfactory. Out of the 3 GPUs, GTX 285 provided the maximum speedup followed by GTX 260 and then, GT 9800.

Finally, we study the effect of "number of iterations" on the speedup, for a given image size. For this experiment, we fixed the image size to 512×512 pixels. Figure 6 shows the plots of time-taken (in seconds) versus "number of iterations". Here also, for both serial and GPU implementations, the time taken is scaling approximately linearly with number of iterations. Figure 7 shows corresponding speedups. The speedups are again around $35-40$ for GTX 285, for iterations more than 400.

Recently, in [14], GPU-based parallel implementation of multi-phase segmentation is done for another convex algorithm proposed in [12]. We notice that, on images of size 625×391 pixels, for 750 iterations and four phases, on GTX 280, they reported an average segmentation time of 1008.72 ms. When we run our algorithm on GTX 285, with similar

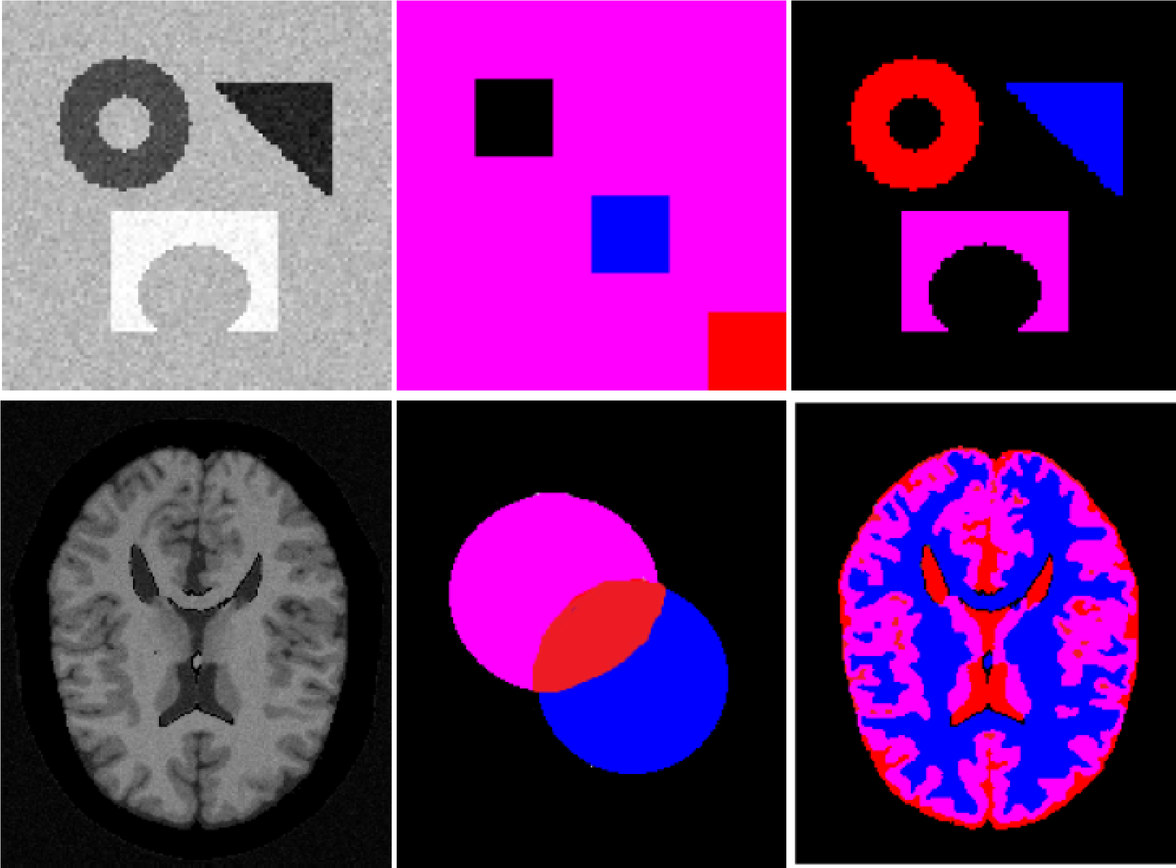

Figure 3: Segmentation results from the convex multi-phase segmentation algorithm. First column shows the input images to be segmented. The middle column shows initialization made; for an easy visualization, each phase is shown in different color. The last column shows the segmentation results.

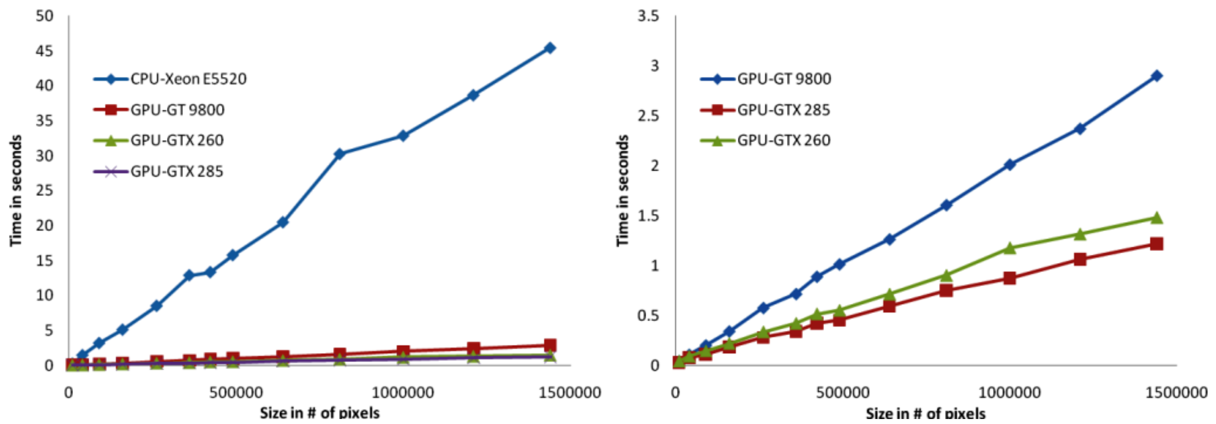

Figure 4: Time-taken (in seconds) versus "size of the image" (in terms of number of pixels) for the serial implementation on CPU, and parallel implementation using CUDA on 3 GPUs. The second figure shows separately the plots only for GPUs from the first figure.

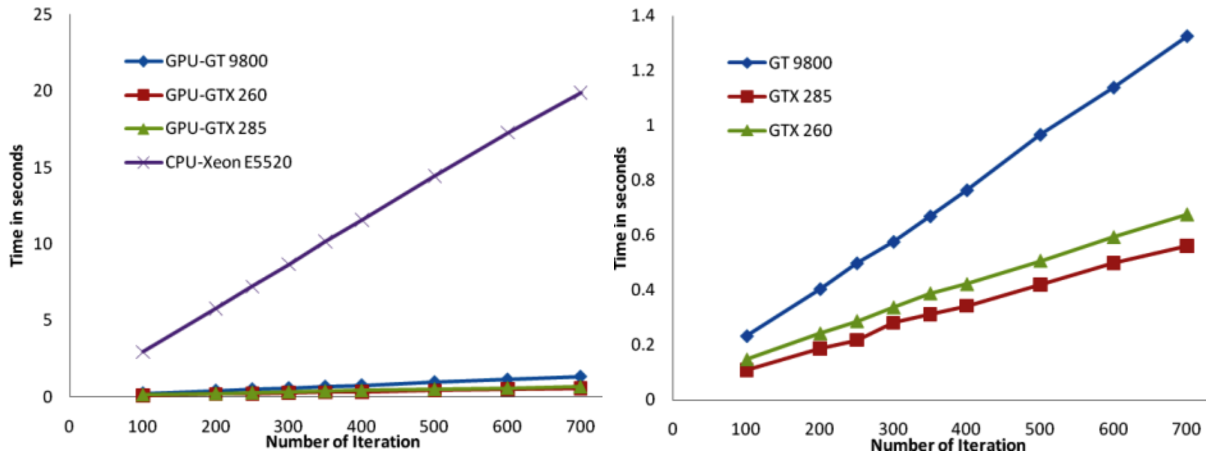

Figure 6: Time-taken (in seconds) versus "number of iterations" for the serial implementation on CPU, and parallel implementation using CUDA on 3 GPUs. The second figure is a zoomed version of the plots for GPUs from the first figure.

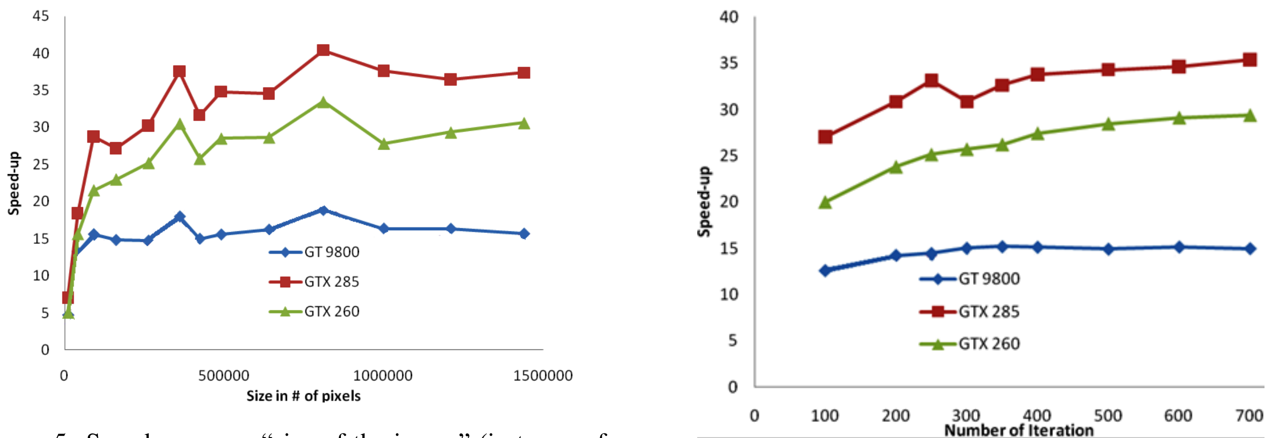

Figure 5: Speedup versus "size of the image" (in terms of number of pixels) for the 3 GPUs (for fixed number of iterations).

Figure 7: Speedup versus "number of iterations" for the 3 GPUs (for a fixed image size).

parameter values and image size, it took, on an average, 598 ms, and the corresponding speedup is 35.58. But, no definite conclusions can be drawn from these figures, since the comparison is not thorough, and is not done on the same data set, and no quantitative accuracy measures are computed. However, we want to reemphasize that, the main goal of this paper is to provide insights into the GPU-based implementation of the convex multi-phase algorithm of [3], and evaluate the improvements in the computational performance, compared to it's original serial implementation.

5. Conclusions

In this paper, we have presented a GPU-based implementation of the convex multi-phase algorithm for Mumford-Shah segmentation model, proposed in [3]. The main contribution of this paper is to provide insights into parallelizing the convex segmentation algorithm of [3]. We have presented a detailed description on the tasks that can be parallelized, and various optimization approaches for speeding up the memory access. We made our CUDA code available online at: http://lts5srv2.epfl.ch/~gorthi/software.html.

The speedups achieved with CUDA implementation are quite satisfactory. As mentioned earlier, [3] provides convex formulation for a class of vector-valued problems, but, at the cost of increased computational complexity due to increased dimensionality compared to the original non-convex model [11]. Hence, GPU-based implementation is particularly useful in this context as it can compensate for the increased computational complexity.

Among the 3 GPUs we have evaluated, the latest one is GTX 285 (available from the beginning of 2009), and it

provided the best speedups (up to 40) out of the 3 GPUs. With the rapid developments in GPU technologies nowadays, new GPUs can be expected to provide even more significant speedups.

One of the computationally expensive task in the CUDA implementation of the current algorithm on GPU is memory access. In the recent versions of GPU, a new memory structure: “surfaces” has been introduced. The speedup may be further improved by profiting from such structures for reducing the memory access time.

Acknowledgments

This work is supported by the Swiss National Science Foundation under grant 205321-124797. X. Bresson would like to thank the NVIDIA Academic Partnership Program for its support.

References

[1] *NVIDIA CUDA Version 3.1*. http://developer.nvidia.com/cuda-toolkit-31-downloads. 3

[2] E. Bae, J. Yuan, and X. Tai. Global minimization for continuous multiphase partitioning problems using a dual approach. *International journal of computer vision*, pages 1–18, 2009. 1

[3] E. Brown, T. Chan, and X. Bresson. A convex relaxation method for a class of vector-valued minimization problems with applications to Mumford-Shah segmentation. Technical report, UCLA, 2010. ftp://ftp.math.ucla.edu/pub/camreport/cam10-43.pdf. 1, 2, 7

[4] A. Chambolle, D. Cremers, and T. Pock. A convex approach for computing minimal partitions. Technical Report 649, CMAP, Ecole Polytechnique, France, 2008. 1

[5] T. F. Chan, S. Esedoglu, and M. Nikolova. Algorithms for finding global minimizers of image segmentation and denoising models. *SIAM Journal on Applied Mathematics*, 66(5):1632–1648, 2006. 1

[6] T. F. Chan and L. A. Vese. Active contours without edges. *IEEE Transactions on Image Processing*, 10(2):266–277, 2001. 2, 5

[7] T. Goldstein, X. Bresson, and O. Stanley. Global minimization of markov random fields with applications to optical flow. Technical report, UCLA, 2009. ftp://ftp.math.ucla.edu/pub/camreport/cam09-77.pdf. 1

[8] H. Ishikawa. Exact optimization for markov random fields with convex priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1333–1336, 2003. 1

[9] J. Lellmann, J. Kappes, J. Yuan, F. Becker, and C. Schnorr. Convex multi-class image labeling by simplex-constrained total variation. In *Proceedings of the Second International Conference on Scale Space and Variational Methods in Computer Vision*, pages 150–162. Springer-Verlag, 2009. 1

[10] J. Lellmann and C. Schnörr. Continuous multiclass labeling approaches and algorithms. *CoRR*, abs/1102.5448, 2011. 1

[11] D. Mumford and J. Shah. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics*, 42(5):577–685, 1989. 1, 2, 7

[12] T. Pock, A. Chambolle, D. Cremers, and H. Bischof. A convex relaxation approach for computing minimal partitions. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 810 – 817, June 2009. 5

[13] T. Pock, T. Schoenemann, G. Graber, H. Bischof, and D. Cremers. A convex formulation of continuous multi-label problems. In *European Conference on Computer Vision (ECCV)*, pages 792–805. Springer, 2008. 1

[14] J. Santner. *Interactive Multi-Label Segmentation*. PhD thesis, Graz University of Technology, October 2010. 5

[15] L. A. Vese and T. F. Chan. A multiphase level set framework for image segmentation using the mumford and shah model. *International Journal of Computer Vision*, 50(3):271–293, 2002. 2, 5

[16] C. Zach, D. Gallup, J. Frahm, and M. Niethammer. Fast global labeling for real-time stereo using multiple plane sweeps. In *Proceedings of Vision, modeling, and visualization (VMV) Workshop*, Konstanz, Germany, October 2008. 1