Fast Parallel Vessel Segmentation

Nitin Satpute[a], Rafael Palomar[b], Rabia Naseem[c], Orestis Zachariadis[a], Juan Gómez-Luna[d], Faouzi Alaya Cheikh[c], Joaquín Olivares[a]

[a]*Department of Electronic and Computer Engineering, Universidad de Córdoba, Spain*
[b]*The Intervention Centre, Oslo University Hospital, Norway*
[c]*Norwegian Colour and Visual Computing Lab, Norwegian University of Science and Technology, Norway*
[d]*Department of Computer Science, ETH Zurich, Switzerland*

Abstract

Accurate and fast assessment of vessel segmentation from liver slices remain challenging and important tasks for the clinicians. The algorithms from the literature are not fast and accurate for vessel segmentation. We propose fast parallel gradient based seeded region growing approach for vessel segmentation. Seeded region growing is tedious when the inter-connectivity between the image elements is unavoidable. Parallelizing region growing algorithms are essential towards achieving real time performance for the overall process of accurate vessel segmentation. The parallel implementation of seeded region growing for vessel segmentation is iterative and hence time consuming process. Seeded region growing is implemented as kernel termination and relaunch on GPU due to its iterative mechanism. The iterative process in region growing is time consuming due to intermediate memory transfers between CPU and GPU. We propose persistent and grid-stride loop based parallel approach for region growing. We analyze static region of interest of tiles on GPU for the acceleration of seeded region growing. The proposed parallel approach provide fast accurate 2D vessel segmentation. We implement parallel gradient based seeded region growing for vessel segmentation. The proposed parallel seeded region growing for vessel segmentation is 1.9x faster compared to the kernel termination and relaunch.

Keywords: Seeded Region Growing, GPU, Kernel Termination and Relaunch (KTRL), Persistent, Grid-stride loop

1. Introduction

In medical imaging, vessel segmentation from liver slices is one of the challenging tasks. Seeded region growing (SRG) is a widely used approach for semi automatic vessel segmentation [1]. Delibasis et. al. have proposed a tool based on a modified version of SRG algorithm, combined with a priori knowledge of the required shape [2]. SRG starts with a set of pixels called seeds and grows a uniform, connected region from each seed. Key steps to SRG are to define seed(s) and a classifying criterion that relies on the image properties and user interaction [3]. SRG starts from a seed and finds the similar neighboring points based on the threshold criteria using 4 or 8 connectivity. The region is grown if the threshold criteria is satisfied. Similar neighbors are new seed points for the next iteration. This process is repeated until the region can not be grown further. In practice, it demands high computational cost to the large amount of dependent data to be processed in SRG especially in the medical image analysis and still requires efficient solutions [4].

SRG is an iterative process. SRG is invoked continuously until region can not be grown further. Iterative process in SRG, when implemented on GPU requires terminating kernel and relaunching from CPU (Kernel Termination and Relaunch (KTRL)) and data transfers between CPU and GPU [3]. So our main objective is to reduce these data transfers using different inter block GPU synchronization (IBS) methods resulting in an efficient parallel implementation of SRG. IBS provides flexibility to move all the computations on GPU by providing visibility to updated intermediate data without any intervention from CPU.

In this paper, we propose persistent, grid-stride loop and IBS based GPU approach for SRG to avoid intermediate memory transfers between CPU and GPU. This also reduces processing over unnecessary image voxels providing significant speedup. Persistent thread block (PT) approach is basically dependent on number of active thread blocks and grid-stride loop becomes essential when the number of threads in the grid are not enough to process the image voxels independently [5, 6].

We implement parallel image gradient using grid-stride loop. We propose gradient and shared memory based fast parallel SRG implemented entirely on GPU without any

intermediate transfers between CPU and GPU. This is inspired by parallel processing on static region of interest (RoI) of tiles on GPU. We compare the proposed persistent based parallel SRG with KTRL for accurate vessel segmentation. The gradient based fast parallel SRG for 2D vessel segmentation is 1.9x faster compared to the state-of-the-art.

The rest of the paper is structured as follows. Section 2 briefs relevant works and state-of-the-art with respect to SRG. Section 3 explains GPU approaches (KTRL and Static) for SRG implementation using persistence and grid-stride loop. The application of parallel SRG to vessel segmentation is discussed in the Section 4. Performance results and comparison of persistent and grid-stride loop based parallel SRG for vessel segmentation are mentioned in the Section 5. Section 6 concludes summarizing the main conclusions of this paper and indicating future directions.

2. Background and Motivation

The SRG plays vital role in medical image segmentation. Smistad et. al. [7] and [8] have discussed about parallel SRG for image segmentation. The reference implementation is shown in the Figure 1. Medical image dataset is cropped before processing. Then the CPU allocates the memory equivalent to the cropped size to copy the data to the cropped image on the GPU. Further SRG is performed for image segmentation. This is the simplistic representation of the work by Smistad et. al. [8]. We have not considered pre-processing stage in this work assuming the images are pre-processed. Smistad et. al. [8] have proposed non-PT (non persistent thread) approach for SRG based vessel segmentation.

Smistad et al. in [3] have proposed parallel region growing with double buffering algorithm based on the parallel breadth first search algorithm by Harish and Narayanan [9]. They have suggested a dynamic queue for SRG and mentioned that changing the number of threads (due to border expansion of the region) typically involves restarting the kernel, and this requires reading all the values from global memory again. But they have not recommended probable solution for this problem. Smistad et al. in [10] have presented a data parallel version of the SRG based Inverse Gradient Flow Tracking Segmentation algorithm using KTRL. Zhang et al. [11] have implemented bidirectional region growing where they have

used a dynamic queue (stack). Jiang et al. [12] have proposed improved branch based region growing vessel segmentation algorithm using stack.

GPU based implementation of SRG needs a dynamic queue (stack). CPUs provide hardware support for stacks but GPUs do not [6]. Any queuing system has a large number of pieces of work to do and a fixed number of workers corresponding to the fixed number of computing units. Pieces are then assigned dynamically to the workers. The problem is deciding the maximum number of pieces of work in the queuing system. If decided, persistent blocks iterate through these pieces of work in the queuing system.

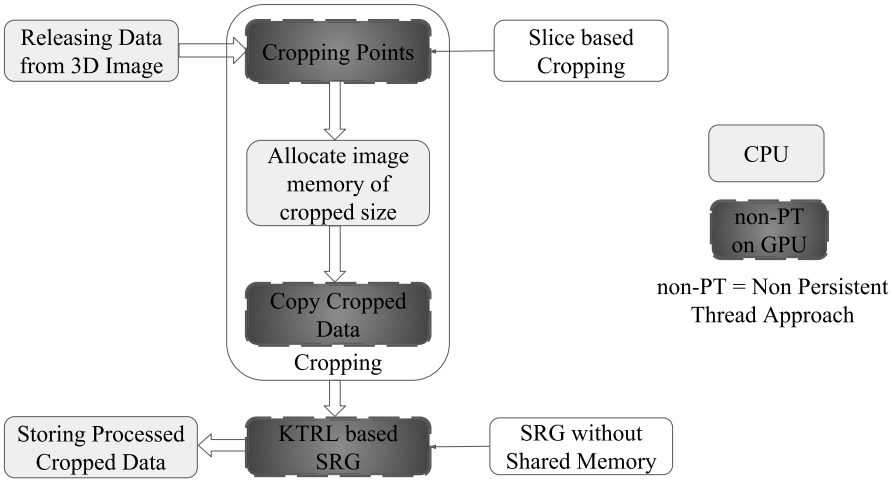


Figure 1: Reference Approach derived from Smistad et. al. [8]

GPU implementation of a stack requires continuous changes in memory allocations which in turn requires iterative GPU kernel invocation from CPU i.e. kernel termination and relaunch (KTRL) as discussed in the algorithms IVM backtracking and work stealing phase by Pessoa et al. [13]. Task-parallel run-time system, called TREES, that is designed for high performance on CPU/GPU platforms by Hechtman et al. [14] have shown the invocation of GPU kernels from CPU iteratively for updating task mask stack (TMS) in TREES execution. The loop involved while implementing data flow through the stream kernels of the rendering system (involving stack) on GPU controlled by CPU (i.e. KTRL) is proposed by Ernst et al. [15].

Nevertheless, there is an alternate GPU implementation of queuing system (stack) using dynamic kernel launching. Chen et. al. [2015] have proposed free launch based dynamic kernel launches through thread reuse technique [6]. This technique requires no hardware extensions, immediately deployable on existing GPUs. By turning subkernel launch into a programming feature independent of hardware support, free launch provides alternate approach for subkernel launch which can be used beneficially on GPUs.

KTRL includes terminating a GPU kernel and invoking it from the CPU if the region can be grown further [3, 10]. GPU kernel SRG is called from CPU. Region grows from a seed based on the threshold criteria. SRG kernel is terminated and relaunched from CPU if region is not grown completely. This process continues until region can not be grown further. The process involves transfer of data to and fro from CPU and GPU. In KTRL, SRG kernel operates on each voxel of whole image data in all the iterations. It includes redundant memory transfers and unnecessary computations over complete image. Hence the main contributions of this paper are the implementation of persistence based approaches to improve the performance of SRG by reducing unwanted computations and avoiding intermediate memory transfers between CPU and GPU. Memory on the GPU is limited and may not be enough for processing large medical datasets. However, most medical datasets contain a lot of data that is not part of the RoI.

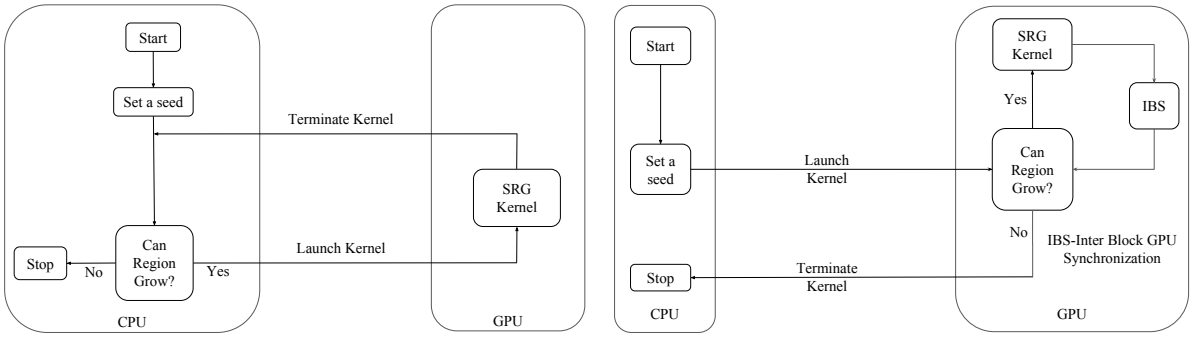
The process of KTRL which involves iterative calling of the kernel is not efficient when implemented on GPU. Hence, as an optimized solution to KTRL, we propose persistent and grid-stride loop based GPU approaches. These approaches are based on processing over static RoI of tiles and dynamic RoI of tiles. We discuss the further details in the upcoming sections.

3. Parallel SRG

GPU is a grid of block of threads. Thread is the smallest computational unit mapped on the cores and block of threads are mapped on the streaming multiprocessors (SMs). Each SM can occupy more than one block. The threads from independent blocks can access data via shared memory in the SM [16]. In order to communicate valid data between the blocks,

these persistent blocks need to be synchronized via IBS through device memory. Persistence implies maximum number thread blocks that can be active at the time of computation depending upon the GPU resources available [5, 17].

We use PT and shared memory based approaches for SRG implementations. Shared memory and grid-stride loop based SRG reduces total memory transfers and computations. Grid-stride is inspired when the grid is not large enough to occupy all the data elements [18, 19]. Rather than assuming that the thread grid is large enough to cover the entire image elements, the kernel loops over the image one grid-size at a time. The stride of the loop is the total number of threads on the grid [18]. These threads (or block of threads) iterate over the image until the process of SRG terminates.



(a) using KTRL by Smistad et. al. [8], [7] (b) using Grid-Stride Loop

Figure 2: GPU Implementations of SRG

For each thread in parallel on GPU, SRG starts from the seed thread and finds similar neighbours surrounding it Region is grown by making similar neighbouring elements as new seeds. The process of SRG is repeated until similar neighbours can not be found. Normally SRG can be implemented on the GPU as a recursive or iterative kernel calling (KTRL) as shown in Figure 2a. Kernel calling involves invocation of a grid of block of threads. The blocks are executed on streaming multiprocessors (SM) and threads are executed on cores. Park et el. [20] and Smistad et al. [3] have given brief introduction about CUDA (Compute Unified Device Architecture) architecture and GPU computing. They have detailed the information on grid, blocks, threads and memory hierarchy of CUDA architecture.

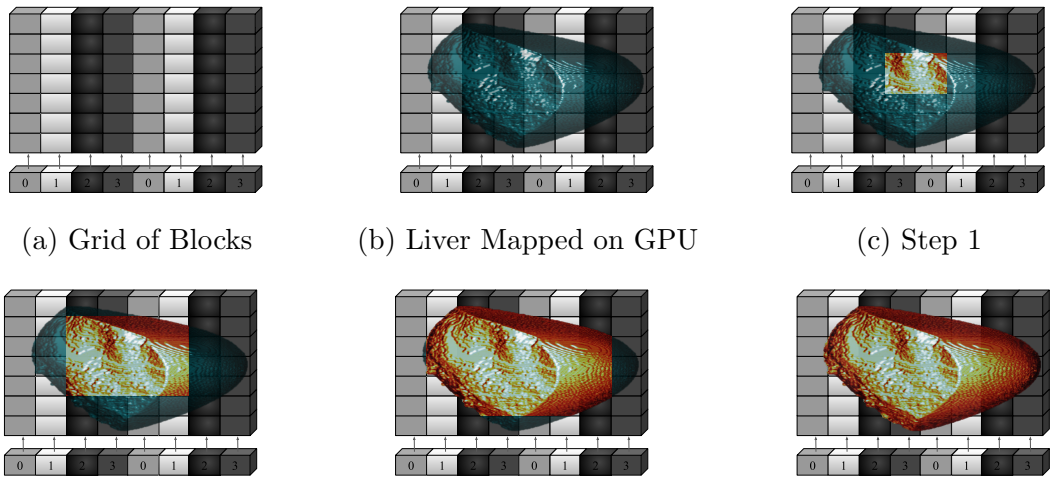
SRG can be recursive or iterative process. Recursive kernel calling can not utilize GPU cores efficiently due to hardware limitations [21]. Iterative GPU kernel call from CPU is costlier due to memory transfers between CPU and GPU and it involves all the image elements to be considered in each step of SRG. GPU implementation of SRG using KTRL is shown in the Figure 2a. It shows that, the kernel SRG is called on GPU continuously from the host CPU until the region can not be grown further. It starts from the seed, finds similar neighbours and grows the region. This process continues until the region can not be grown further. The process of the KTRL causes unnecessary image elements to be part of computations and intermediate memory transfers between CPU and GPU.

Hence in order to avoid these problems, we propose grid-stride loop through complete image based GPU approach as shown in the Figure 2b. SRG starts from the seed and the control goes to GPU. The SRG kernel is launched if the region is not grown completely. IBS is needed in order to transfer valid data in between the active thread blocks. The number of active thread blocks on SMs are limited due to resource constraints. These maximum number of active blocks are persistent blocks [5, 16, 17]. The looping i.e. grid-stride loop terminate when the region can not be grown further and control returns to the host CPU as shown in the Figure 2b. We have discussed KTRL based GPU approach for SRG implementation and its disadvantages. Now, we are going to analyze PT based GPU approaches for high performance SRG implementation. Proposed approaches exploit parallelism using persistence and IBS as detailed in the static and dynamic approaches.

3.1. Static Approach

In the proposed approach, we apply grid-stride loop through static RoI (complete image) using persistence and IBS [5, 17]. The complete liver image is mapped on the GPU as grid of block of threads as shown in the Figure 3b. CPU invokes SRG kernel on GPU. Persistent blocks iterate through complete image and grow region from the seed in each and every iteration on GPU. This iteration of persistent blocks over the tiles of the image and the grid-stride loop based SRG is shown in Figure 2b. Steps of SRG in Figures 3c - 3f show the grown region of the liver. SRG kernel terminates when the region is grown completely. We

copy the data from the device memory to the shared memory. This data is shared by all the threads inside the blocks. This is necessary to share the neighbouring elements between different voxels of the image. For each parallel thread in the block, if seed is found and is not the boundary element of the block, we calculate similar neighbouring elements. Region is grown by making similar neighbouring elements as new seeds.



(a) Grid of Blocks (b) Liver Mapped on GPU (c) Step 1

(d) Step 2 (e) Step 3 (f) Step 4

Figure 3: SRG using Persistence and Grid-Stride Loop through Complete Image

There are four persistent blocks shown in Figure 3. These four persistent blocks are iterated through liver elements. Tiles with the same color are iterated by same persistent block. In KTRL, these tiles are processed by the thread blocks randomly.

Step 1 in Figure 3c is obtained when the grid-stride loop by persistent blocks (4 in this example) is applied on the tiles over liver image. Region is grown around the seed containing similar elements. IBS is applied to communicate valid data in between the blocks for the next step of SRG as shown in Figure 2b. Persistent blocks iterate over the liver image and the region is grown again in step 2 as shown in Figure 3d. IBS is applied and valid data is communicated in between the blocks so that the region can be grown further as shown in Figures 3e and 3f. After step 4 in Figure 3f, SRG stops as region can not be grown further. Each step contain many iterations where region starts growing when persistent blocks iterate through tiles of the image. This iterative process continues until region can not be grown

further. Code snapshot of the complete process is provided in the Algorithm 1.

Algorithm 1: Grid-stride Loop through Complete Image

```
1:  unfinished=1;
2:  while unfinished==1 do
3:    unfinished=0;
4:    for int i=blockIdx.x; i <= width/(blockDim.x − 2); i=i+gridDim.x do
5:      for int j=blockIdx.y; j <= height/(blockDim.y − 2); j=j+gridDim.y do
6:        for int k=blockIdx.z; k <= depth/(blockDim.z − 2); k=k+gridDim.z do
7:          Region_Growing(arguments, unfinished);
8:        end for
9:      end for
10:   end for
11:   Inter_Block_GPU_Sync();
12: end while
```

Global variable "unfinished" is 1 if region has to be grown further else it is 0. Persistent blocks in x, y and z directions iterate through complete image. Two is subtracted from block dimensions to avoid computations around boundary voxels (from left and right in each dimensions) from shared memory as region can not be grown further in the block. After each step of SRG, when the processing on complete image is done then all the persistent blocks are globally synchronized via "Inter_Block_GPU_Sync()" barrier. This ensures that valid data is communicated for the next step of SRG. This barrier can be Atomic(), Quasi(), LockFree() or can be implemented using NVIDIA CUDA API Cooperative-groups [17, 22, 23]. We use quasi based IBS because of its efficicient implementation [22].

4. Application to 2D Vessel Segmentation

The 2D segmentation algorithm is inspired by the gradient based SRG algorithm developed by Rai and Nair [24]. We proposed the fast parallel SRG based segmentation algorithm on GPU for vessel segmentation. We discuss the two important modules i.e. image gradient and SRG for the fast parallel 2D segmentation of vessels from CT liver images.

4.1. Parallel Image Gradient

Rai and Nair [24] have presented homogeneity criterion selection and its impact on the quality of segmentation using SRG. They have used gradient based cost function. These cost functions are based on object contrast, region boundary, homogeneity of the region, and texture features like shape and color, intensities values, gradient direction and magnitude. The cost function exploits certain features of the image around the seed. Gradient based cost function requires gradient of the image, largest gradient magnitude (max_g) and minimum gradient (min_g) present in the image.

Algorithm 2: Parallel Image Gradient using Grid-stride Loop

```
1:  voxel.x = blockIdx.x * blockDim.x + threadIdx.x;
2:  voxel.y = blockIdx.y * blockDim.y + threadIdx.y;
3:  stridex = blockDim.x * gridDim.x;
4:  stridey = blockDim.y * gridDim.y;
5:  for int k=voxel.x; k < rows; k=k+stridex do
6:    for int l=voxel.y; l < cols; l=l+stridey do
7:      candidate.x = k + 1; candidate.y = l + 1;
8:      check if neighbour candidate is within image dimensions;
9:      gx = 0.5*(data[candidate.x*cols + l] - data[k*cols + l]);
10:     gy = 0.5*(data[k*cols + candidate.y] - data[k*cols + l]);
11:     g = sqrt(gx*gx + gy*gy);
12:     data_g[k*cols + l]=g;
13:     if(max_g < g) atomicMax(&max_g, g);
14:     if(min_g > g) atomicMin(&min_g, g);
15:   end for
16: end for
```

The cost functions are:

$$cost1 = g/(k * max\_g) \qquad 0 < cost1 < 1 \tag{1}$$

$$cost2 = (max\_g - g)/(max\_g - min\_g) \qquad 0 < cost2 < 1 \tag{2}$$

where g is gradient magnitude of the pixel under consideration and k is the constant

parameter which controls the region growth. The pixel under consideration is added in the growing region if it matches with the seed elements i.e. cost functions specified by Equations 1 and 2 are satisfied otherwise it is excluded from consideration.

We propose grid-stride loop based parallel image gradient method in Algorithm 2. For each pixel in parallel, we calculate its gradient magnitude (g) with respect to neighbouring element. Horizontal and vertical gradient components are given by gx and gy. The magnitude of maximum and minimum gradients are updated simultaneously. The gradient of the image is desired input for SRG based segmentation along with the seed. This is discussed in the next section.

4.2. Parallel Vessel Segmentation

We propose fast parallel vessel segmentation as shown in Figure 4. The algorithm is inspired from gradient based segmentation algorithm by Rai and Nair [24]. Figure 4 shows parallel implementation of vessel segmentation where the user selects seed(s). These seed(s) along with the image are transferred to the GPU. Device kernel calculates the image gradient in parallel as discussed in the earlier section. The IBS is necessary to reflect the updated image gradients in the device memory.

Further we apply SRG algorithm. The cost functions based on gradient are shown in Equations 1 and 2. For each pixel in parallel, the pixel under consideration invokes SRG kernel if it satisfies the cost functions. The seeds are updated after IBS and the gradient based cost functions are verified again for new pixels. This process continues until no new seeds are formed i.e. no new pixels are added to the growing region.

The kernel is terminated and the control returns to the CPU when the region is grown completely. The segmented image is transferred to the CPU. The process of segmentation stops. This GPU implementation avoids iterative call of SRG kernel from CPU. We use gradient and persistent based parallel SRG for vessel segmentation.

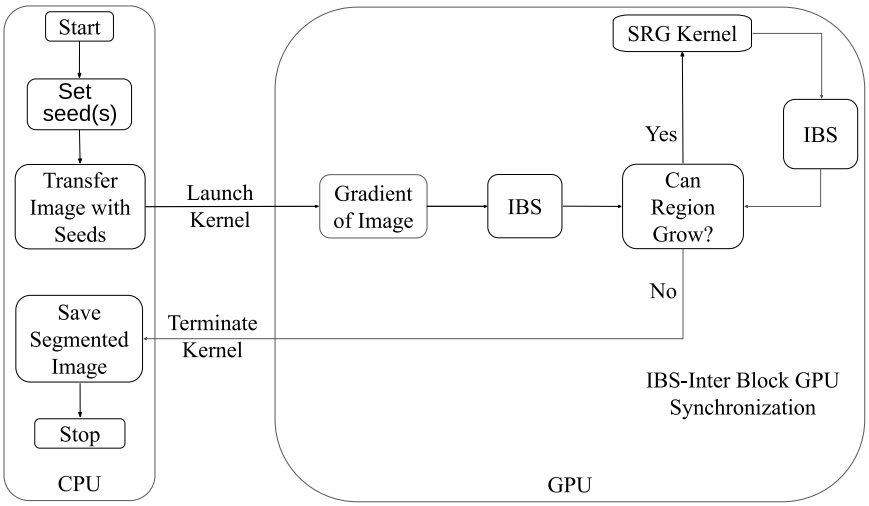


Figure 4: Proposed Parallel Vessel Segmentation

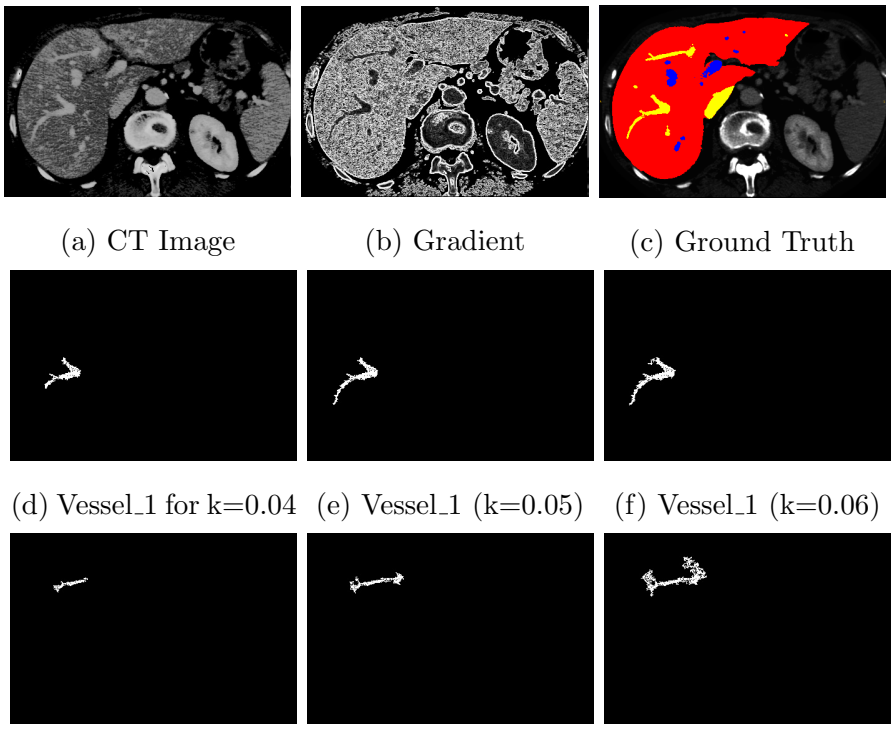
5. Performance Evaluation

We propose persistent and grid-stride based GPU approaches for fast parallel 2D vessel segmentation. The performance results are obtained from KTRL and proposed persistent based GPU approach. We compare proposed approaches with KTRL. We use Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz RAM 24 GB, NVIDIA GPU 1050 (RAM 4GB), OpenCL 1.2 (ref. [25]) and CUDA Toolkit 10.1 for the implementation.

5.1. Parallel 2D Vessel Segmentation

We propose persistent and grid-stride based GPU approaches for fast parallel 2D vessel segmentation. Variations in vessel segmentation with constant parameter k using parallel SRG is shown in Figure 5. The input to the parallel 2D SRG is CT slice of the liver as shown in Figure 5a. We calculate the gradient of input CT image as shown in Figure 5b. The ground truth for the segmentation is shown in Figure 5c.

We show the two segmented vessels with change in parameter k i.e. 0.04, 0.05, and 0.06. The first segmented vessel as shown in Figures 5d, 5e, 5f is accurate at 0.05. Similarly we



(a) CT Image (b) Gradient (c) Ground Truth

(d) Vessel_1 for k=0.04 (e) Vessel_1 (k=0.05) (f) Vessel_1 (k=0.06)

(g) Vessel_2 (k=0.04) (h) Vessel_2 (k=0.05) (i) Vessel_2 (k=0.06)

(Note: k=0.05 is providing visually better results for the first Liver slice)

Figure 5: Variations in Fast Parallel Vessel Segmentation with Constant Parameter 'k' using Parallel SRG on First Liver Slice

show the segmentation of second vessel from the same slice. The variations in segmentation w.r.t. k are shown in Figures 5g, 5h, 5i. The more accurate segmentation is obtained at 0.05.

Further we show the accuracy of the segmentation on another CT Slice as shown in Figure 6a. We calculate the gradient (Figure 6b) of the input CT image on GPU. We apply the parallel SRG using gradient based thresholding criteria giving more accurate results at k=0.05 for two vessels inside the CT slice as shown in Figures 6c and 6d. The ground truth for the segmentation is shown in Figure 6e. We analyze that the vessels are more accurately segmented when the parameter k takes the value 0.05.

The speedup obtained by proposed parallel static approach over KTRL on first two CT liver slices are shown in the Table 1. The maximum speedup for vessel segmentation by

(a) CT Image (b) Gradient (c) First Vessel (d) Second Vessel (e) Ground Truth

Figure 6: Vessel Segmentation (for k=0.05) using Parallel SRG on Second Liver Slice

Table 1: Time and Speedup for Vessel Segmentation

Data →	Vessel Segmentation	
GPU Approaches → Metrics ↓	KTRL	Static (Speedup)
Time in ms for kernel SRG - 1st Slice (k=0.05) 1st vessel	5.7	3.4 (1.67x)
Time in ms for kernel SRG - 1st Slice (k=0.05) 2nd vessel	2.1	1.5 (1.4x)
Time in ms for kernel SRG - 2nd Slice (k=0.05) 1st vessel	1.5	1 (1.5x)
Time in ms for kernel SRG - 2nd Slice (k=0.05) 2nd vessel	3.5	2.4 (1.45x)

proposed parallel static SRG is 1.67x w.r.t. KTRL on the first liver slice. But the average speedup obtained by proposed parallel static approach for all the vessels (in 6 slices tested) is 1.9x compared to KTRL. We evaluate the speedup of the vessel segmentation on parameter k=0.05 because the vessel segmentation is more accurate as shown in Figure 5.

Further we analyze the effect of parallel SRG on different slices for multiple vessel segmentation using multiple seeds as shown in Figures 7, 8, 9, and 10. The segmentation

(a) CT Image (b) Gradient (c) Two Seeds (d) Four Seeds (e) Ground Truth

Figure 7: Fast Parallel Vessel Segmentation using Parallel SRG on Third Liver Slice using multiple seeds

(a) CT Image (b) Gradient (c) Three Seeds (d) Four Seeds (e) Ground Truth

Figure 8: Fast Parallel Vessel Segmentation using Parallel SRG on Fourth Liver Slice using multiple seeds

(a) CT Image (b) Gradient (c) Single Seed (d) Four Seeds (e) Ground Truth

Figure 9: Fast Parallel Vessel Segmentation using Parallel SRG on Fifth Liver Slice using multiple seeds

(a) CT Image (b) Gradient (c) Two Seeds (d) Three Seeds (e) Ground Truth

Figure 10: Fast Parallel Vessel Segmentation using Parallel SRG on Sixth Liver Slice using multiple seeds

of the long vessel as shown in Figure 8d is slightly extended compared to the ground truth shown in Figure 8e. It can be seen from input CT image and gradient image (Figures 8a and 8b), the long vessel has extension which is not shown in the ground truth. We show the thick vessel segmentation in Figures 7d, 9c, 8c and thin vessel segmentation in Figures 7c, 8c and 8d.

Our proposed parallel implementations of SRG are not only fast but also accurate for vessel segmentation. This accuracy of the segmentation depends on the constant parameter 'k'. The clinicians get the flexibility to decide which segmentation is more accurate. The process takes very less time (few ms). Hence this reduces the overall time for segmentation for various values of parameter 'k' if the clinician wants to have more accuracy.

5.2. Discussion

In this paper, we propose persistence and grid-stride loop based SRG implementation. In order to obtain significant speedup, we need to exploit parallelism by using persistence and IBS. It involves change in the large body of SRG algorithm. We want algorithms that require as less synchronization as possible. In general if algorithm requires IBS, it is probably not going to be particularly fast. The fastest algorithms on GPUs are ones that fit nicely into the GPU programming model, where blocks are independent from each other and do not require synchronization.

But the problem arises when iterative calling of the kernel can not be avoided. It incurs memory transfers from CPU to GPU when KTRL is used for global synchronizations. Hence it has to go through synchronizations as the next step of SRG which is dependent on the current step. Terminating a kernel and relaunching incurs data transfers from CPU to GPU and vice versa. It is time consuming.

If we use IBS method along with persistence, then we can map whole algorithm on GPU with synchronization. Control comes back to CPU only if the kernel task is over. CPU launches a kernel on GPU, GPU executes it and final results are copied to CPU. No intermediate data communication occurs in the proposed approach (unlikely in KTRL).

6. Conclusion

In this paper, we discuss SRG based vessel segmentation and its parallel implementation on GPU. We propose persistence and grid-stride loop based GPU approach for SRG providing significant speedup. Normally recursion/iterative calling of a kernel is generally a bad idea on GPUs. We use persistence and grid-stride approach as an alternate implementation for

KTRL. We compare proposed GPU optimization strategy for SRG implementation. The proposed persistent and gradient based parallel SRG for 2D vessel segmentation is accurate and 1.9x faster compared to the KTRL.

Acknowledgements

The work is supported by the project High Performance soft tissue Navigation (HiPerNav). This project has received funding from the European Union Horizon 2020 research and innovation program under grant agreement No. 722068. We thank The Intervention Centre, Oslo University Hospital, Oslo, Norway for providing the CT images with ground truths for the clinical validation of vessel segmentation

References

[1] R. Palomar, F. A. Cheikh, B. Edwin, Å. Fretland, A. Beghdadi, O. J. Elle, A novel method for planning liver resections using deformable bézier surfaces and distance maps, Computer methods and programs in biomedicine 144 (2017) 135–145.

[2] K. K. Delibasis, A. Kechriniotis, I. Maglogiannis, A novel tool for segmenting 3d medical images based on generalized cylinders and active surfaces, Computer Methods and Programs in Biomedicine 111 (1) (2013) 148 – 165. doi:https://doi.org/10.1016/j.cmpb.2013.03.009.
URL http://www.sciencedirect.com/science/article/pii/S0169260713000989

[3] E. Smistad, T. L. Falch, M. Bozorgi, A. C. Elster, F. Lindseth, Medical image segmentation on gpus–a comprehensive review, Medical image analysis 20 (1) (2015) 1–18.

[4] J. Wassenberg, W. Middelmann, P. Sanders, An efficient parallel algorithm for graph-based image segmentation, in: International Conference on Computer Analysis of Images and Patterns, Springer, 2009, pp. 1003–1010.

[5] K. Gupta, J. A. Stuart, J. D. Owens, A study of persistent threads style gpu programming for gpgpu workloads, in: Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012), IEEE, 2012, pp. 1–14.

[6] G. Chen, X. Shen, Free launch: optimizing gpu dynamic kernel launches through thread reuse, in: Proceedings of the 48th International Symposium on Microarchitecture, ACM, 2015, pp. 407–419.

[7] E. Smistad, Seeded region growing, https://github.com/smistad/FAST/tree/master/source/FAST/Algorithms/ (2015).

[8] E. Smistad, A. C. Elster, F. Lindseth, Gpu accelerated segmentation and centerline extraction of tubular structures from medical images, International journal of computer assisted radiology and surgery 9 (4) (2014) 561–575.

[9] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the gpu using cuda, in: S. Aluru, M. Parashar, R. Badrinath, V. K. Prasanna (Eds.), High Performance Computing – HiPC 2007, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 197–208.

[10] E. Smistad, A. C. Elster, F. Lindseth, Gpu accelerated segmentation and centerline extraction of tubular structures from medical images, International journal of computer assisted radiology and surgery 9 (4) (2014) 561–575.

[11] X. Zhang, X. Li, Y. Feng, A medical image segmentation algorithm based on bi-directional region growing, Optik-International Journal for Light and Electron Optics 126 (20) (2015) 2398–2404.

[12] H. Jiang, B. He, D. Fang, Z. Ma, B. Yang, L. Zhang, A region growing vessel segmentation algorithm based on spectrum information, Computational and mathematical methods in medicine 2013.

[13] T. C. Pessoa, J. Gmys, N. Melab, F. H. de Carvalho Junior, D. Tuyttens, A gpu-based backtracking algorithm for permutation combinatorial problems, in: J. Carretero, J. Garcia-Blas, R. K. Ko, P. Mueller, K. Nakano (Eds.), Algorithms and Architectures for Parallel Processing, Springer International Publishing, Cham, 2016, pp. 310–324.

[14] B. A. Hechtman, A. D. Hilton, D. J. Sorin, TREES: A CPU/GPU task-parallel runtime with explicit epoch synchronization, CoRR abs/1608.00571 (2016) ,. arXiv:1608.00571.
URL `http://arxiv.org/abs/1608.00571`

[15] M. Greiner, Stack implementation on programmable graphics hardware, Vision, Modeling, and Visualization 2004: Proceedings (2004) 255.

[16] V. Vineet, P. J. Narayanan, Cuda cuts: Fast graph cuts on the gpu, in: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008, pp. 1–8. doi:10.1109/CVPRW.2008.4563095.

[17] S. Xiao, W. C. Feng, Inter-block gpu communication via fast barrier synchronization, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–12. doi:10.1109/IPDPS.2010.5470477.

[18] M. Harris, Cuda pro tip:write flexible kernels with grid-stride loops (2015).
URL `http://goo.gl/b8Vmkh`

[19] M. Sourouri, S. B. Baden, X. Cai, Panda: A compiler framework for concurrent cpu+gpu execution of 3d stencil computations on gpu-accelerated supercomputers, International Journal of Parallel Programming 45 (3) (2017) 711–729.

[20] S. Park, J. Lee, H. Lee, J. Shin, J. Seo, K. H. Lee, Y.-G. Shin, B. Kim, Parallelized seeded region

growing using cuda, Computational and mathematical methods in medicine 2014.

[21] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, C. R. Das, Controlled kernel launch for dynamic parallelism in gpus, in: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 649–660. doi:10.1109/HPCA.2017.14.

[22] Y. Komura, Y. Okabe, Gpu-based single-cluster algorithm for the simulation of the ising model, Journal of Computational Physics 231 (4) (2012) 1209–1215.

[23] T. Sorensen, H. Evrard, A. F. Donaldson, Cooperative kernels: Gpu multitasking for blocking algorithms, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 431–441.

[24] G. Rai, T. Nair, Gradient based seeded region grow method for ct angiographic image segmentation, arXiv preprint arXiv:1001.3735, (2010).

[25] J. E. Stone, D. Gohara, G. Shi, Opencl: A parallel programming standard for heterogeneous computing systems, IEEE Des. Test 12 (3) (2010) 66–73. doi:10.1109/MCSE.2010.69.
URL http://dx.doi.org/10.1109/MCSE.2010.69