A Fast Parallel Level Set Segmentation Algorithm for 3-D Images

Benjamin Trapani, *Student*, Julian Gutierrez, *Student*, David Kaeli, *Professor*
Dept. of Electrical and Computer Engineering
Northeastern University, Boston, MA

Abstract

Image segmentation is one of the key analysis tools in biomedical imaging applications. Although level set segmentation algorithms have been explored thoroughly in the past, these approaches are non-scalable due to their inherent data dependencies. Algorithms with large corresponding data dependency graphs that contain many small cycles are difficult to parallelize, prohibiting these algorithms from effectively leveraging modern highly parallel compute devices. Given that the resolution of medical imaging hardware has continued to increase each year, and CPU performance has not kept pace, there is a need to explore parallel solutions for processing medical images. Prior work described an efficient level set segmentation algorithm designed for parallel architectures for segmenting 2D images. The algorithm segments an input image into four components based on an initial curve. The prior 2-D level set segmentation algorithm is extended, providing a solution for 3-D images. The algorithm is improved by examining adjacent voxels at each step, versus visiting adjacent pixels instead. The initial curve is a user-provided sphere that is defined parametrically to reduce copy overhead to the compute device. The efficiency of the 2D algorithm is preserved in the conversion, enabling the resulting algorithm to perform on the order of ten times faster than existing GPU-accelerated 3-D level set segmentation implementations. The implementation presented in this work supports real-time segmentation of 7T MRI images, leveraging the computational power of a NVIDIA Tesla K20 GPU to reduce execution time. This image segmentation algorithm supports identification of tumors, tissue volume measurements, and surgery planning at the rate required by radiologists today.

I. INTRODUCTION

Image segmentation is the process of separating an input image into a set of disjoint components, thereby aiding analysis and simplifying further processing. Efficient image segmentation is critical to the delivery of real-time computer vision systems which require low and deterministic latency, and high throughput. One field that demands accurate and efficient image segmentation is medicine. Biomedical imaging applications use image segmentation algorithms to enable radiologists and doctors to isolate tumors, tissue classes and organs for further analysis. In a typical 8-hour workday, radiologists are required to analyze approximately one image every four seconds [1]. Performing an accurate analysis under this time constraint is difficult, especially when existing image segmentation systems are not capable of processing the image under analysis in four seconds or less.

The typical workflow for the user consists of uploading raw images into the system, viewing a raw image, selecting regions of interest in the image and running the image segmentation algorithm to gain detailed regions that highlight the desired biological features. The result of the segmentation process is a set of refined spatial regions in the image that are likely part of the regions of interest specified initially. The initial regions of interest are specified using two pieces of data: a shape drawn in the input image, and an intensity range. Both properties of the input range serve as hints to the image segmentation system about the location of the desired biological features. The image segmentation system should extract detailed connected regions such that the extracted regions' intensity ranges mostly match the corresponding input range and the shape mostly matches the corresponding input shape. Parts of the input image included in the segmented region that do not fall within the input shapes or do not fall within the input intensity ranges should only be selected because they are connected to a shape that is definitely part of the desired region. Existing implementations that segment 3-D images do so by numerically solving partial differential equations [2]. These algorithms fail to achieve real-time performance on larger inputs due to data dependencies and high computational requirements.

Previous research by Gutierrez et al. [3] details an algorithm for image segmentation that is optimized for data-parallel architectures. A 2-D implementation of the algorithm was presented, and its performance evaluated using the Compute Unified Device Architecture (CUDA) run on a NVIDIA graphics processing unit (GPU). Gutierrez et al. improved the performance by at least 5.5 times that of each previous implementation benchmarked. Their algorithm evolves an input set of curves given an intensity range by examining adjacent pixels and modifying the state of each central pixel based on the state of adjacent pixels. Extending the algorithm to three dimensions is a logical extension of the prior work, but poses implementation challenges, particularly in preserving the performance of the 2-D implementation. The performance goal guiding the development and optimization of the 3-D version is meeting or exceeding the per-pixel throughput of the 2-D algorithm, which is 2.6×10^8 pixels/s [3].

Although modern data-parallel computer architectures, such as GPUs, can provide high throughput, maximizing the performance of a GPU implementation requires tuning of the associated memory access patterns. The prior implementation [2] coalesces all global memory accesses by reordering input data on the CPU using Morton codes for each pixel, resulting in adjacent pixels in 2-D space being collocated in a one-dimensional data array on the GPU. Optimizing memory accesses in 3-D space when using a one-dimensional data array can be accomplished by reordering the input data per the 3-D Morton code at each index [4]. However, performing this reordering introduces an additional performance cost that is an extra linear term with respect to the number of input voxels. A voxel is a 3D cube analogous to a pixel in 2D space. To offset the performance cost of the initial reordering, Equation 1, seen below, must be satisfied. M_b and M_a are the relative memory bandwidths before and after the optimizations respectively. V_r and V_i are the number of voxels read from memory and the number of voxels in the input image respectively.

$$\frac{V_r}{M_b} > \frac{V_i + V_r}{M_a} \quad (1)$$

Since typical medical images can be segmented with approximately twenty full reads from global memory (see results), other optimizations are pursued due to the relative difficulty of improving global memory efficiency by more than 5%. These optimizations are further necessitated by and adapted for the order of magnitude growth in data size due to the additional image dimension and rapidly increasing medical image sizes. This paper presents a careful performance analysis of the adaptation of the algorithm to 3-D space.

II. Algorithm

A. Definitions

The level set segmentation algorithm uses a 3D array of the same shape as the input image to store the current state of each voxel in the original image. The coded state of each voxel is used by the algorithm to iteratively compute the next state until convergence. The possible voxel state codes and associated meanings are summarized in Table 1, seen below.

Table 1. The relative location, relative to true image, denoted for each voxel.

Voxel state	Location
-2	Not in image
-1	Outside border of image (might be part of image)
1	Inside image
2	Inside border of image (might not be part of image)

The following variables define the sets and accessors for these sets that are used in the level set segmentation algorithm pseudocode:

- **N(x)**: The set of voxels adjacent to x excluding those at positions diagonal to x.
- **L**: label shapes
- **V**: All voxels in the input image
- **I**: Intensities of input image
- **I(x)**: intensity of voxel x
- **R**: valid intensities per label

The level set segmentation algorithm implemented is divided into three stages. The first initializes the voxel states based on the intensity of the initial image and the position of the label curve. The second stage updates the state of each voxel based on adjacent voxel states. This stage runs iteratively until convergence. The final stage assigns final voxel codes that are consistent with the converged voxel states and the meaning of each code.

B. Stages

Stage 1.

```
K = ∅
For λ ∈ L:
  For v ∈ V:
    If I(v) ∈ R(λ):
     If v ∈ λ:
       I(v) = 1
     Else
       I(v) = -1
       K = K ∪ v
    Else:
     If v ∈ λ
       I(v) = 2
       K = K ∪ v
     Else
       I(v) = -2
```

Stage 2.

```
P = ∅
While K ≠ ∅ ∩ P ≠ I:
  P = I
  For v ∈ K:
     prevVI = ∅
     While I(v) ≠ prevVI:
        prevVI = I(v)
        If I(v) == -1 ∩ z ∈ N(v) ∩ I(z) == 1, I(v) = 1
        Else if I(v) == 2 ∩ z ∈ N(v) ∩ I(z) == -2, I(v) = -2
     If I(v) ∈ {-2, 1), K = K ∩ ~v
```

Stage 3.

```
For v ∈ V:
  If I(v) ∈ {1, 2}:
    I(v) = 2 If z ∈ N(v) ∩ I(z) ∈ {-1, -2}, else I(v) = 1
  Else:
    I(v) = -2 If z ∈ N(v) ∩ I(z) ∈ {-1, -2}, else I(v) = -1
```

III. Implementation

Four different implementations of the algorithm are built to evaluate the kernel and total execution times of a baseline implementation. The performance of the baseline implementation is compared against the performance of three optimizations. The target device for all versions is a NVIDIA Tesla K20 GPU. The Tesla K20 is a mid-range workstation GPU [5]. It is representative of the devices that medical image segmentation systems would typically run on. All implementations share a common pipeline structure with stages in the pipeline matching the stages in the algorithm pseudocode detailed in Section II. All implementations also use shared memory in an identical fashion. The label image representation, number of voxels processed per thread and alignment of global memory are adjusted for each implementation. The baseline implementation launches one thread per voxel, uses a dense representation of label curves and does not align global memory. The three optimizations process multiple voxels per thread, use parametric label definitions and align global memory.

A. Kernel Pipeline

A parent kernel is launched from the host CPU, sending a single one-dimensional block of data that is equal in size to the number of label curves supplied by the user. The parent kernel is responsible for launching three other kernels, which perform the three stages in the algorithm. Leveraging NVIDIA's dynamic parallelism technology to instantiate child kernels from the parent kernel, unnecessary memory transfers between the CPU and GPU can be eliminated during successive runs of stage 2, which has been shown to increase the performance of the algorithm [3]. Since each thread in the parent kernel operates on independent sections of the working and output sets, it is safe to evolve all label curves in parallel.

All child kernels are instantiated with a 3-D grid and block dimensions. The block size is user-specified and constant across each dimension. Assuming a block dimension size of K and a number of voxels processed per thread V, the number of blocks per grid dimension for a grid dimension with x voxels is computed according to Equation 2, seen below:

$$f(x) = 1 + \frac{(x-1)}{K*V} \quad (2)$$

K=8 is used for these experiments. A block size of 8 x 8 x 8 is used because it is the configuration with power of two side lengths that results in the number of threads per block being closest to the optimal threads per block discovered in the 2D implementation, which contains 1024 [3]. V=1 and V=2 are evaluated in these experiments. To evaluate the second iteration condition in stage 2, at least one state of the evolving curve has to have been modified in the last iteration. Parallel writes are then issued to global memory addresses shared among all threads for the given label. An array of integers of size equal to the number of label curves is allocated in global memory and initialized to 0 prior to starting each pass through the outer loop in stage 2. A flag located in shared memory is used to determine if any thread in a block was updated as part of curve evolution, during the current pass through the outer loop. If at least one thread in the current block updated the state of a voxel in the evolving curve, the thread in the lower front-left corner of the block will write a 1 to the global memory address for the current label, indicating that the current label curve needs to continue to evolve.

Filtering the threads that perform the evolution logic, shown in stage 2, is performed at a block granularity, versus a thread granularity. An array of integer flags is used to indicate whether blocks have converged. Block granularities are used because filtering at a thread granularity would have required much more memory for the flags and would have resulted in warp divergence, negating the potential performance improvement gained by reducing the number of scheduled threads that did not modify the evolving curves. Blocks converge when at least one voxel is in state -1 or 2. The array is allocated in global memory, initialized in stage 1, and updated after each execution of stage 2. A variable in shared memory, indicating whether at least one thread in the current block is in state -1 or 2, is used to update the global block indicator. The thread operating in the lower front left corner of each block updates the global block indicator stored in the shared value computed for the current block.

B. Shared Memory and Block Borders

Shared memory is used in all child kernels to minimize the number of global memory reads. Since the second and third stages in the algorithm require that each thread examines adjacent voxels, threads within a block will collectively read the same location in global memory at least nine times. Shared memory is located on the streaming multiprocessor where a block is scheduled to run, whereas global memory is located off the chip. Higher memory bandwidth can be achieved by using shared memory to avoid redundant reads to global memory [6]. A device function, called by the kernels for the second and third stages, loads data equal in size as the input image into shared memory prior to executing the algorithm defined in each sub-kernel. Each thread fetches the data, corresponding to the thread's location in 3-D space, into shared memory. If the thread is on the border of the block, the thread fetches the voxel one sequential location across the boundary nearest it. If the number of voxels processed per thread is one, the thread simply loads the voxel one unit across the border. If there are two voxels processed in each dimension, the thread loads the voxel one unit across the border if it is on the lower boundary and loads the voxel two units from its position (one unit across the upper border) otherwise. In the version which processes eight voxels per thread, each thread loads a 2 x 2 x 2 cube of voxels into shared memory. The coordinates of the thread in 3-D space, multiplied by 2, define the lower front left corner of the cube.

C. Parametric Labels

To reduce the overhead incurred by copying data to the GPU, parametrically defined label curves are evaluated. The initial 3-D implementation, based on the 2-D implementation by Gutierrez et al. [3], operates on label curves defined in the same format and resolution as the input intensity image. Although this implementation gives users the ability to define more complex labels, which have the potential to converge in fewer iterations, the enhanced flexibility comes at a significant performance cost when accounting for time spent copying data

to the GPU. An alternative parametric definition of labels is evaluated to reduce the amount of data copied to the GPU. The second iteration of the image segmentation system supports parametric label spheres. Any point inside or on the border of a given label sphere is defined to be within the label in stage 1 of the algorithm. The amount of data copied to the GPU per label is 128 bits, consisting of four 32-bit floats which define the origin and the radius. For an intensity image that is at least 3 x 3 x 3 voxels in size, the parametrically defined label sphere requires less data than the original label format. When using parametrically-defined label spheres, additional overhead is incurred in performing the square root operation to compute the distance between each voxel and the center of each label sphere. Extending the implementation to support other parametrically-defined shapes as required by different use cases is a trivial extension of the existing implementation, although performance characteristics may not be the same for other shapes.

D. Eight Voxels Per Thread

Performance of the 2-D implementation improves when 2 x 2 pixel blocks are processed by each thread [3]. If the ratio between threads and streaming multiprocessors (SMs) exceeds 1, unnecessary overhead is incurred when mapping/scheduling the resulting lightweight threads to SMs. Although optimizing the ratio between threads and SMs is system and data set specific, the method for implementing this optimization on the 3-D extension is detailed so that future users can tweak this implementation to best suit their needs. In stage 1, 2 x 2 x 2 chunks of data, which are contiguous in 3-D space, are processed on each thread. The 2 x 2 x 2 sets of elements, separated from each other by one block in each dimension, are processed in kernel stages 2 and 3. Blocking is used to avoid bank conflicts when operating on data in shared memory. When writing data back to global memory in stages 2 and 3, each thread writes a 2 x 2 x 2 chunk that is contiguous in the 3-D space.

E. Aligned Global Memory

It is possible to improve global memory performance by coalescing the reads issued by threads in each warp into one memory transaction. Sequential accesses to global memory will be coalesced via 32, 64, or 128-byte transactions, aligned on the size of each block [7]. Threads are assigned to warps sequentially along the x-dimension. To help ensure that reads issued by threads in a warp are likely to be coalesced, allocations are padded along the x-dimension such that the size of the x-dimension and the size of the xy-plane are multiples of one of the memory transaction sizes listed above. CUDA provides a function that queries the supported memory transaction sizes for the current device and allocates a padded chunk of memory given the dimensions of the 3-D image, as provided by the user [8]. The input, working and output images are padded along the x-axis, using the best value, prior to being copied to global memory. The parent kernel continues to instantiate sub-kernels across each dimension of the input image. Sub-kernels are modified to rely on an additional set of width parameters for each image, instead of using the dimensions of the grid. No additional computational overhead is introduced by aligning global memory. Three additional kernel parameters are required, resulting in 192 additional bits of data to be copied to the GPU per execution. The maximum number of wasted bytes *w* due to padded allocations in an input image can be computed using Equation 3, seen below:

$$w = 127 * h * d \quad (3)$$

As shown in Equation 3, the wasted bytes, *w*, is a function of the product of image height, *h*, and depth, *d*. 127 bytes are wasted per row of voxels if the GPU has the maximum global memory alignment of 128 bytes and the image width modulo 128 is one.

IV. Results

A. Benchmarks

All benchmarks are run on synthetic spheres that are of uniform intensity. Four implementations are evaluated, one for each optimization. Each implementation is tested on images of identical size in each dimension. The radius of the synthetic sphere is equal to one fourth of the size of a single dimension of the input image. Images of size 128^3, 256^3, 512^3 and 1024^3 voxels are evaluated. For each synthetic input sphere, label spheres of both half the radius and twice the radius of the input sphere are used as the initial curve, enabling the measurement of the difference in performance between inward and outward evolutions of the initial curve. For each unique tuple of version, image dimension and label radius, twenty benchmarks are generated and processed. The execution performance results for the largest image are summarized below in Table 2.

Table 2. Kernel execution time and total time to segment the input image, as a function of the input image size.

Version	image dim (voxels)	label radius (voxels)	avg. kernel exec time (ms)	avg. total exec time (ms)
8vx	1024	128	508.38	1440.839
8vx-aligned	1024	128	476.178	1354.368
basic	1024	128	5102.493	6298.697
param-labels	1024	128	5313.346	6293.904
8vx	1024	512	929.29	1852.648
8vx-aligned	1024	512	882.841	1742.711
basic	1024	512	5913.594	7147.313
param-labels	1024	512	6087.566	7049.704

The first four rows in Table 2 indicate both the kernel and total performance of the proposed system when the initial curve specified by a user is half the size of the target object. The second four entries represent the corresponding performance data when the initial curve fully encloses the target object and is twice as large. The kernel execution time is the time elapsed between the invocation of the parent kernel from the CPU and the completion of this kernel. The total execution time includes the time required to create GPU resources, transfer data to and from the GPU and free the allocated resources.

Since radiologists and other users of the image segmentation

system could specify the initial curve such that it is either larger or smaller than the desired region, the proposed system should efficiently identify the desired region in both cases. The kernel performance for the outward evolution of the initial label sphere when segmenting the largest synthetic sphere is summarized in Figure 1, seen below.

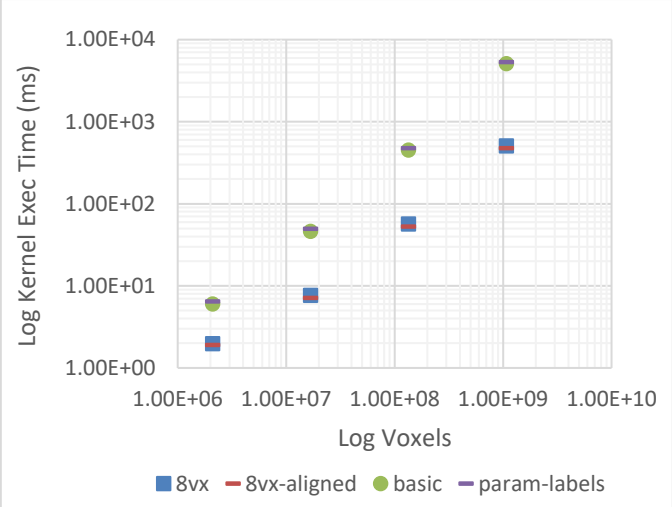

Figure 1. Average kernel execution time (log_{10}) as a function of voxel count (log_{10}) in input image for each implementation when label sphere has a radius equal to half that of the input sphere's.

The corresponding kernel performance for inward evolution when segmenting the largest synthetic sphere is summarized in Figure 2, seen below.

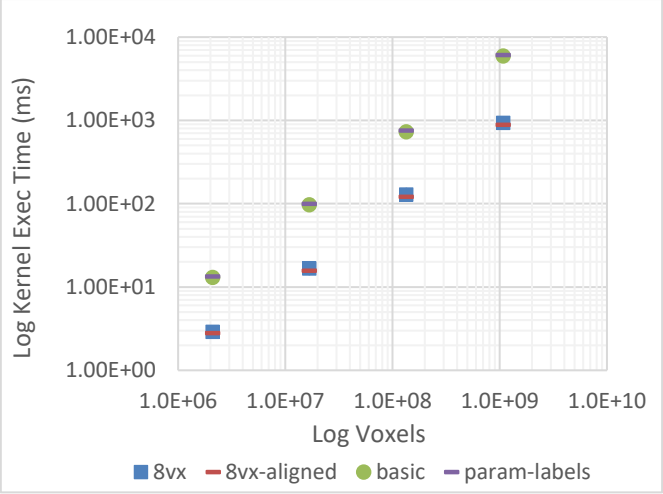

Figure 2. Average log kernel execution time as a function of log voxel count in input image for each implementation when label sphere radius is twice that of the input sphere's.

Providing the algorithm with an initial curve that is smaller than the target and entirely contained by the target sphere results in faster segmentation, versus using an initial curve that is larger than the target sphere. Segmentation using the 8vx-aligned implementation and a label sphere that is smaller than the target sphere takes 22% less time than segmentation when the initial label sphere is larger. The reason for this behavior is that the number of non-converged blocks per pass through stage 2 of the algorithm is much larger when evolving inward, as compared to evolving outward, due to the larger number of voxels on the boundary of the evolving curve. The voxels on the boundary change state frequently, due primarily to being outside of the intensity range provided, requiring each block with at least one voxel on the boundary of the evolving curve to be processed in full. The logic that examines the state of the adjacent voxels per thread in stage 2 is skipped for converged blocks, resulting in the observed performance improvement when the initial curve is smaller than the target and evolves outward. The ratio between inward and outward evolution performance is larger for the 8vx and 8vx-aligned versions because they process eight times as many voxels per block compared to the other implementations. Processing more voxels per block results in an increase in the number of voxels that are processed unnecessarily on the border of the evolving shape due to filtering converged regions at block granularity.

To compare the overall performance of each version, the total time required to allocate GPU memory, copy data to the GPU, perform the algorithm, copy the results back to the CPU and free up used GPU memory is summarized in Figure 3, seen below.

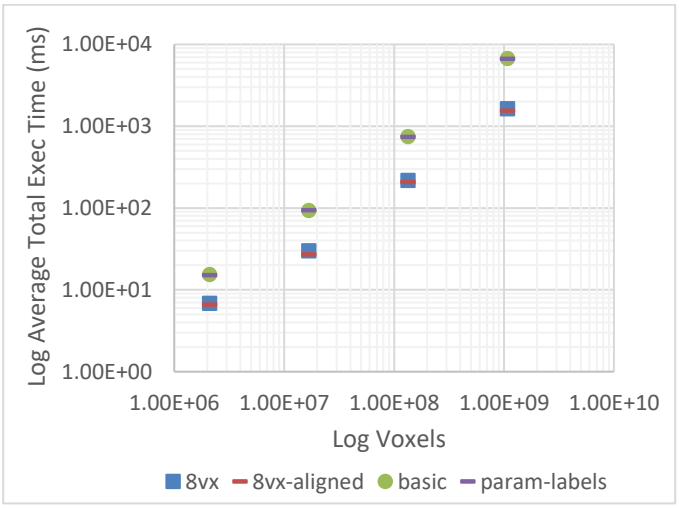

Figure 3. Total amount of time to segment synthetic spheres of a given size averaged across inward and outward total evolution times.

The basic and parametric label implementations perform comparably, with the parametric label implementation taking 0.99 times that of the basic implementation for the largest image. Although the amount of time taken to copy the data to the GPU is greatly reduced in the parametric version, the cost of computing the distance between the sphere center and each voxel causes the kernel performance to deteriorate sufficiently to virtually offset the gains achieved in the reduced copy overhead. The 8vx and 8vx-aligned versions perform comparably as well. The version using aligned memory took 0.94 times the amount of time taken by the 8vx version. The kernel execution time was improved by using aligned memory. However, the performance of the memory allocation suffers because of the additional overhead incurred when padding the

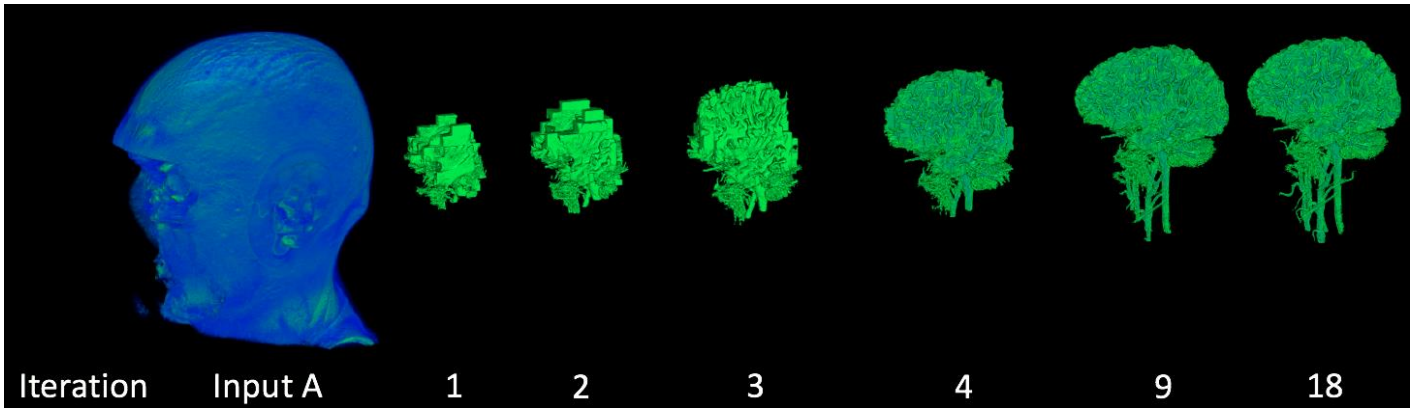

Figure 4. Evolution of spherical label curve on benchmark brain MRI image.

data to ensure alignment in global memory. Further overhead is incurred when transferring the padded data between the CPU and GPU due to the increase in space required to store the padded data. For all images benchmarked, the additional overhead resulting from padding memory is smaller than the associated kernel performance improvement.

B. Accuracy

To verify the accuracy of all implementations, the segmented image regions are checked by comparing them to the data in the original image. If the code is -2 or 1, the corresponding voxel in the original image must not be part of the sphere or must be inside the sphere, respectively. If the code is 2, the current voxel must be adjacent to at least one voxel that is inside of the sphere in the original image. It must also be located on the outermost shell of voxels in the original sphere. If the code is -1, the current voxel must be adjacent to at least one voxel that is not inside of the sphere in the original image. It must also be located on the shell of voxels one unit larger than the input sphere. The number of places that a code's meaning is inconsistent with the input image data is counted. The number of incorrectly segmented voxels, divided by the total number of voxels in the input image, is computed. The latter number is computed from the input and resulting image for each benchmark. The percentage of incorrectly segmented voxels is zero in all benchmarks, implying that the segmented results are completely correct for the synthetic inputs.

C. Performance Comparison

The most efficient existing GPU-accelerated 3-D image segmentation system is developed by Roberts et al. [2]. They benchmark their application on an MRI image of a brain of size 256^3 voxels. Their implementation takes 7 seconds to converge when run on a NVIDIA GTX 280 GPU. The implementation in this paper is tested on an MRI image of a brain with size 274 x 384 x 384 voxels using a label curve with a radius of fifty voxels located in the center of the image. The average time required to segment the MRI image is 99.56 ms. The results of the segmentation are found in Figure 4, seen above. The benchmark is obtained from an average of twenty runs using the 8vx-aligned version of the code. The operating system targeted in the implementation by Roberts et al. is Windows while the implementation in this paper targets Linux. Benchmarking the previous 3-D implementation [2] on the hardware and software stack used to obtain the benchmarks in this paper would require porting it to Linux, which is left as future work. A service that aggregates GPU benchmarks called User Benchmark suggests that a NVIDIA GTX 1080 is about 12.95 times more efficient at typical 3-D rendering tasks than a GTX 280. This data is derived from 93,288 and 511 samples for the GPUs, respectively [9]. Benchmarks for the GTX 1080 are leveraged for comparison in place of those for the Tesla K20 because benchmarks for the Tesla K20 GPU are not available on this service. The GTX 1080 provides a more recent GPU architecture, and includes a higher clock speed, higher memory bandwidth, and faster memory versus the Tesla K20 GPU [5, 10].

To compare the previous implementation [2] to the implementation presented in this paper, the benchmarks of the GTX 1080 and those of the GTX 280 are used to generate a GPU factor by which to convert the performance of the previous implementation to the equivalent performance of the same implementation when run on a Tesla K20. The number of voxels processed is equivalent to the total number of voxels in the input image. The total time is the time required to segment the image, including overhead incurred when setting up and freeing GPU resources. The adjusted result is computed according to Equation 4 seen below.

$$AdjustedResult = \frac{(VoxelsProcessed * GPUFactor)}{TotalTime} \quad (4)$$

These results are detailed in Table 3, seen below.

Table 3. Performance comparison between existing algorithms.

Version	**Voxels Processed**	**Total time (s)**	**GPU factor**	**Adjusted Result (voxels/s)**
Roberts	1.677×10^7	7.0	12.95	3.104×10^7
Proposed	4.04×10^7	0.1	1	4.04×10^8

Despite the generous handicap of assuming that a Tesla K20 is as fast as a GTX 1080, the proposed implementation is able to process voxels at a rate of 13 times the rate of the previous 3-D implementation [2]. Obtaining benchmarks of the previous implementation on Linux using a Tesla K20 would reveal an

even larger performance improvement, the size of which is to be determined in future experiments. Due to the 3-D algorithm requiring the examination of more adjacent locations per step and operating on larger data, it is expected that the 3-D algorithm would have a lower per unit throughput than that of the 2-D implementation. However, the 3-D implementation processes 4.04 x 10^8 voxels/s, while the previous 2-D implementation achieved 2.6 x 10^8 pixels/s [3]. The previous 2-D implementation was benchmarked on the same hardware and software stack leveraged in this implementation.

V. Conclusion

Previous 3-D image segmentation implementations are relatively efficient. However, the 7 seconds required to segment a typical MRI image of a brain using existing implementations is larger than the 4 seconds radiologists have to analyze a single image [1]. The implementation presented in this paper is over 13 times more efficient than the best previous implementation. The efficiency gained as a result of the system detailed in this paper will enable radiologists to segment incoming images well within the allotted time per analysis, since the proposed system only requires 0.1s to segment a typical MRI image of a brain. The performance of the proposed system also provides near-instantaneous feedback, which will enable radiologists to quickly refine the input parameters to the level set segmentation algorithm. As a result, they will obtain higher quality refined images to guide their analyses. Higher quality image segmentations in less time will increase the productivity and accuracy of radiologists. Providing radiologists with improved tools to allow efficient analysis of medical images without sacrificing their accuracy will benefit patients around the world who depend on medical imaging for diagnosis and monitoring, including patients with cancer, heart disease and osteoporosis among many others. The goal of surpassing or maintaining the performance of the 2D implementation is achieved. The throughput of the proposed system is over 1.5 times larger than the throughput achieved by the previous implementation on typical images [3]. The remaining work required to deliver this implementation as a pluggable stage in existing medical image processing pipelines consists of defining an API that client applications can use to specify labels, set initial data and read results. Future work will focus on making the system production-ready, in addition to porting the previous implementation [2] to Linux and benchmarking it.

Acknowledgment

The author would like to thank Julian Gutierrez for introducing him to general purpose GPU computing, explaining in depth his 2-D implementation of the algorithm, suggesting the best ways to optimize the 3-D implementation and for providing feedback on this paper throughout all revisions. The author would also like to thank Dr. Kaeli for editing this paper and for providing the resources required to conduct this research.

Ben Trapani received a B.S. degree in Computer Science from Northeastern University in 2018. He received an award for Outstanding Student Research in Computer Science at Northeastern University's RISE 2017 event. After graduating in the spring of 2018, he will be joining Microsoft as a software engineer on the AI and Ink team. He can be contacted at ben.trapani1995@gmail.com.

Julian Gutierrez received a B.S. degree in Electrical Engineering from the University of Costa Rica in 2012. He was a Structural Design Engineer at Intel Corp. in Heredia, Costa Rica and a lecturer at the University of Costa Rica between 2011 and 2015. He received a MSc. degree in Electrical and Computer Engineering from Northeastern University in 2017. He joined the Northeastern University Computer Architecture Research Laboratory under Dr. David Kaeli in 2015 and has been conducting research in High Performance Computing with a focus on GPU applications and heterogeneous systems. He is currently a PhD Candidate in Computer Engineering at Northeastern University. He can be contacted at gutierrez.jul@husky.neu.edu.

David Kaeli received a BS and PhD in Electrical Engineering from Rutgers University, and an MS in Computer Engineering from Syracuse University. He is the Associate Dean of Undergraduate Programs in the College of Engineering and a Full Processor on the ECE faculty at Northeastern University, Boston, MA. He is the director of the Northeastern University Computer Architecture Research Laboratory (NUCAR). Prior to joining Northeastern in 1993, Kaeli spent 12 years at IBM, the last 7 at T.J. Watson Research Center, Yorktown Heights, NY. Dr. Kaeli has published over 300 critically reviewed publications, 7 books, and 11 patents. His research spans a range of areas including microarchitecture to back-end compilers and database systems. His current research topics include hardware security, graphics processors, virtualization, heterogeneous computing and multi-layer reliability. He serves as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems, ACM Transactions on Architecture and Code Optimization, and the Journal of Parallel and Distributed Computing. Dr. Kaeli is an IEEE Fellow and a Distinguished Scientiest of the ACM. He can be contacted at kaeli@ece.neu.edu.

References

[1] McDonald RJ, Schwartz KM, Eckel LJ, et al, "The effects of Changes in Utilization and Technological Advancements of Cross-Sectional Imaging on Radiologist Workload", Acad Radiol. Published online July 22, 2015.

[2] M. Roberts, J. Packer, M. Costa Sousa and J. Ross Mitchell, "A Work-Efficient GPU Algorithm for Level Set Segmentation", Eurographics Association, Saarbrucken, Germany, 2010

[3] J. Gutierrez, F. Nina-Paravecino and D. Kaeli, "A Fast Level-Set Segmentation Algorithm for Image Processing Designed For Parallel Architectures", 2016 6th Workshop on Irregular Applications: Architecture and Algorithms, 2016.

[4] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing", International Business Machines Company New York, 1966.

[5] "Tesla K20 GPU Accelerator", *nvidia.com*, 2017. [Online]. Available: https://www.nvidia.com/content/PDF/kepler/Tesla-K20-

Passive-BD-06455-001-v05.pdf. [Accessed: 31- Aug- 2017].

[6] "Kepler Tuning Guide :: CUDA Toolkit Documentation", *Docs.nvidia.com*, 2017. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#axzz4jigtZfbq. [Accessed: 11- Jun- 2017].

[7] M. Harris, "How to Access Global Memory Efficiently in CUDA C/C++ Kernels", *Parallel For all*, 2017. [Online]. Available: https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/. [Accessed: 02- Jul- 2017].

[8] "CUDA Runtime API :: CUDA Toolkit Documentation", *Docs.nvidia.com*, 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g188300e599ded65c925e79eab2a57347. [Accessed: 11- Jun- 2017].

[9] "UserBenchmark: Nvidia GeForce GTX 280 vs 1080", *Gpu.userbenchmark.com*, 2017. [Online]. Available: http://gpu.userbenchmark.com/Compare/Nvidia-GTX-1080-vs-Nvidia-GeForce-GTX-280/3603vsm8413. [Accessed: 31- Aug- 2017].

[10] "GeForce GTX 1080 Graphics Cards from NVIDIA GeForce", *NVIDIA*, 2017. [Online]. Available: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/. [Accessed: 31- Aug- 2017].