





# Parallel Implementation of the DRLSE Algorithm

Daniel Popp Coelho<sup>(✉)</sup>  and Sérgio Shiguemi Furuie 

School of Engineering, University of São Paulo, São Paulo, Brazil  
1danielcoelho@usp.br

**Abstract.** The Distance-Regularized Level Set Evolution (DRLSE) algorithm solves many problems that plague the class of Level Set algorithms, but has a significant computational cost and is sensitive to its many parameters. Configuring these parameters is a time-intensive trial-and-error task that limits the usability of the algorithm. This is especially true in the field of Medical Imaging, where it would be otherwise highly suitable. The aim of this work is to develop a parallel implementation of the algorithm using the Compute-Unified Device Architecture (CUDA) for Graphics Processing Units (GPU), which would reduce the computational cost of the algorithm, bringing it to the interactive regime. This would lessen the burden of configuring its parameters and broaden its application. Using consumer-grade, hardware, we observed performance gains between roughly 800% and 1700% when comparing against a purely serial C++ implementation we developed, and gains between roughly 180% and 500%, when comparing against the MATLAB reference implementation of DRLSE, both depending on input image resolution.

**Keywords:** CUDA · DRLSE · Level Sets

## 1 Introduction

The field of medical imaging possesses many different imaging modalities and techniques. For this field, image segmentation allows delineating organs and internal structures, information that is used for analysis, diagnosis, treatment planning, monitoring of medical conditions and more. The quality of the segmentation, then, directly affects the quality and effectiveness of these processes, again highlighting the importance of the correct choice of segmentation algorithm [1, 2].

Level Set segmentation algorithms define a higher dimensional function on the domain of the image, and starting from a user-input initial state of this function, iterate it with some procedure, formulated in such a way that the set

---

The results published here are in part based upon data generated by the TCGA Research Network: <http://cancergenome.nih.gov/>. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Finance Code 001.

of pixels for which the Level Set Function equals zero (called the zero-level set) converges on a target region. After a set number of iterations, the zero-level set represents the contour that is the result of the segmentation [3,4]. Those algorithms are capable of segmenting images with soft and hard edges, and the generated contour is also capable of splitting and merging during the iteration process, which is ideal for segmenting biological structures with complex shapes and varying intensity levels.

The Distance-Regularized Level-Set Evolution or DRLSE is an established Level Sets algorithm that effectively solves some of the usual problems with that class of algorithms, including the problem of reinitialization [4]. It has since been applied in many different contexts since its inception. The elevated computational cost and sensitivity to input parameters are still significant disadvantages of the class, however.

The configuration of DRLSE parameters is a manual task that usually involves trial and error and specialized knowledge, and depends on the specific image used and target object to segment. The situation is made worse by the elevated computational cost of the algorithm itself, that implies in slow iteration cycles. Fortunately, like most Level Sets algorithms, the DRLSE algorithm is highly parallelizable.

Graphics Processing Units (GPUs) are pieces of hardware originally designed for acceleration and parallelization of 3D rendering tasks. Through frameworks like CUDA (Compute Unified Device Architecture) [5], it is now possible to use GPUs for acceleration of General-Purpose algorithms (GPGPU) [5,6].

The specifics of GPU architecture vary greatly between manufacturers and generations. For the purposes of this paper, suffice to say that a GPU is composed of many streaming multiprocessors (SM), which are groups of individual processor cores. When a programmer dispatches a CUDA kernel (small GPU program) for execution on the GPU, he or she specifies how many blocks and threads these kernels should execute on, which are abstractions of SM and processor cores, respectively [5,6]. The abstraction allows CUDA code to be relatively independent of the exact GPU architecture it executes on. It also allows the hardware to manage kernel execution more freely. For example, when more blocks are requested at a time than are available on the hardware, they are placed in a queue and dispatched to SM as they become free. Additionally, CUDA allows multiple blocks to be executed concurrently on the same SM under some conditions [5].

The CUDA framework will be used in this paper to develop kernels for a parallel implementation of the DRLSE algorithm, allowing not only for faster usage, but faster iteration cycles when choosing the optimal segmentation parameters.

## 2 Materials

The program was developed and tested exclusively on a computer with an AMD Ryzen 7 2700X 3.70 GHz processor, 32 GB of DDR4 RAM, Windows 10 Enterprise N (64 bit) and an NVIDIA GTX 1060 6 GB GPU (Pascal architecture) [7].

The program was always executed out of a WD Blue 1TB 7200 RPM SATA 6Gb/s 3.5 in. hard disk drive.

The algorithm was developed and built for CUDA 9.1, using CMake 3.10.1 and Microsoft Visual C++ Compiler 14 (2015, v140). We also developed a purely serial implementation of DRLSE for C++ to be used as a point of reference for the performance comparisons. Additionally, the reference MATLAB implementation of DRLSE [8] was used for performance comparisons.

All of the medical images used in this work were retrieved from [9, 10].

NVIDIA’s official profiler, nvprof was used for timing execution of all CUDA programs analyzed in this work. It is available bundled with the CUDA 9.1 SDK.

### 3 Methods

Special care was taken to guarantee the operations performed were as similar as possible between all implementations, as a way to isolate the performance impacts of parallelization. With that in mind, gradients were calculated via central differences, and second derivatives were implemented using discrete laplacian kernels, so as to mirror MATLAB’s grad and del2 functions, respectively. Tests were then performed on large datasets to guarantee the results agreed for these isolated operations, between all three implementations.

We then implemented the DRLSE algorithm according to [4] and the MATLAB reference [8]. For more details and the derivation of the following equations, the reader is directed to the original DRLSE work at [4]. For the purposes of this work, suffice to describe the main iteration equation and its components.

Assuming that  $\Omega$  describes the domain of an input image  $I$ , the Level Set function can be defined as  $\Phi : \Omega \rightarrow \mathbb{R}$ . The final zero-level set of  $\Phi$ , the result of the segmentation, is then a contour given by a set  $S$  of image pixels, described by  $S = \{x, y \in \Omega \mid \Phi_{x,y} = 0\}$ .

For the bidimensional case (2D images, where  $\Omega = \mathbb{R}^2$ ), (1) describes the main iteration equation for the DRLSE algorithm. In (1),  $\Phi_{x,y}^k$  describes the value of  $\Phi$  for pixel  $[x, y]$  at iteration number  $k$ .

$$\begin{aligned} \Phi_{x,y}^{k+1} = & \Phi_{x,y}^k + \tau \cdot [\mu \cdot \text{div}(\frac{p'(|\nabla\Phi_{x,y}^k|)}{|\nabla\Phi_{x,y}^k|} \cdot \nabla\Phi_{x,y}^k) \\ & + \lambda \cdot \delta_\varepsilon(\Phi_{x,y}^k) \cdot \text{div}(g(x, y) \cdot \frac{\nabla\Phi_{x,y}^k}{|\nabla\Phi_{x,y}^k|}) \\ & + \alpha \cdot g(x, y) \cdot \delta_\varepsilon(\Phi_{x,y}^k)] \end{aligned} \quad (1)$$

At each time step and for every pixel,  $\Phi_{x,y}^k$  is incremented by a term weighted by a time constant  $\tau$ . The value incremented (enveloped in brackets in (1)) is a combination of the regularization term (weighted by the constant  $\mu > 0$ ), the length term (weighted by the constant  $\lambda > 0$ ), and the area term (weighted by the constant  $\alpha \in \mathbb{R}$ ). These three terms are formulated in terms of  $p(s)$ ,

a double-well energy potential function, and its derivative,  $p'(s)$ , both described respectively in (2) and (3).

$$p(s) = \begin{cases} \frac{1}{(2\pi)^2}(1 - \cos(2\pi \cdot s)) & s \leq 1 \\ \frac{1}{2}(s - 1)^2 & s \geq 1 \end{cases} \quad (2)$$

$$p'(s) = \begin{cases} \frac{1}{2\pi}\sin(2\pi \cdot s) & s \leq 1 \\ s - 1 & s \geq 1 \end{cases} \quad (3)$$

The indicator image  $g(x, y) : \Omega \rightarrow \mathbb{R}$ , used in (1) and described in (4), is constructed in such a way as to have low values near the contours of interest. In our implementation and in [8], (4) uses the magnitude of the gradient of the input image  $I$  convolved with a gaussian kernel  $G_\sigma$ . The constant  $\sigma$  describes the standard deviation of  $G_\sigma$ , and is used as an additional paramater for this implementation. The reasoning behind this is that its optimal value depends on the hardness of the borders of the target object to segment.

$$g(x, y) = \frac{1}{1 + |\nabla G_\sigma \cdot I(x, y)|^2} \quad (4)$$

Finally,  $\delta_\varepsilon(s)$  describes a simple approximation of Dirac's delta function, described by (5).

$$\delta_\varepsilon(s) = \begin{cases} \frac{1}{3}(1 + \cos \frac{\pi s}{1.5}) & |s| \leq 1.5 \\ 0 & |s| > 1.5 \end{cases} \quad (5)$$

The main parameters of this DRLSE implementation, that need to be configured before each segmentation, are  $\alpha$ ,  $\mu$ ,  $\lambda$ , and  $\sigma$ . The DRLSE algorithm is sensitive to these parameters, and whereas we determined experimentally that a sensible range of potential values for  $\alpha$ ,  $\mu$ ,  $\lambda$  would be  $[-15, 15]$  (respecting the fact that  $\mu > 0$  and  $\lambda > 0$ ), it is common to find scenarios where a change of 0.1 in any of these parameters drastically alters the final segmentation result.

The edge indicator image  $g(x, y)$  can be constructed with Algorithm 1, executed on the CPU. Step 3 of Algorithm 1 involves dispatching the kernel *EdgeIndicatorKernel* to the GPU, which is described on Algorithm 2. For brevity, gaussian convolution and the gradient calculation kernel are omitted here.

Note that *edge\_grad\_image* has the dimensions of *input\_image*, however it possesses two channels,  $x$  and  $y$ , describing the gradient in the  $x$  and  $y$  directions, respectively. The *gradient kernel* described on step 4 of Algorithm 1 fills both channels of this image based on the values of *edge\_image*.

After the construction of the edge indicator image, the main DRLSE iteration procedure is executed on the CPU. That procedure, for an iteration process with 1000 iterations, is described on Algorithm 3.

The laplace kernel used at line 4 of Algorithm 3 is omitted for brevity, but involves a simple convolution with a  $3 \times 3$  laplacian kernel.

It is worth noting that *phi\_grad\_image* has the dimensions of *input\_image*, however it possesses four channels, unlike *edge\_grad\_image*. The first two channels describe the gradient of *phi\_image* in the  $x$  and  $y$  directions, respectively.

Channels 3 and 4 of *phi\_grad\_image* contain the normalized gradient in the  $x$  and  $y$  direction, respectively, so that the magnitude of the gradient equals one. In the same manner as before, the normalized gradient kernel fills in all four channels of *phi\_grad\_image* with the gradient and normalized gradient of *phi\_image*.

---

**Algorithm 1:** Edge Indicator Image  $g(x, y)$  (CPU)
 

---

**input :** *gaussian\_kernel15x15*, *input\_image*  
**output:** *edge\_image*, *edge\_grad\_image*

- 1 Execute convolution kernel with *gaussian\_kernel15x15* and *input\_image* generating *temp\_image*;
  - 2 Define *edge\_image* and *edge\_grad\_image*, both with same dimensions as *input\_image*;
  - 3 Execute *EdgeIndicatorKernel* with *temp\_image* and *edge\_image*, modifying *edge\_image*;
  - 4 Execute gradient kernel with *edge\_image* and *edge\_grad\_image*, modifying *edge\_grad\_image*;
  - 5 Return *edge\_image* and *edge\_grad\_image*;
- 

---

**Algorithm 2:** EdgeIndicatorKernel (GPU)
 

---

**input :** *temp\_image*, *edge\_image*  
**output:** Nothing

- 1 Acquire position  $[x, y]$  of the current pixel;
  - 2 Define  $plusX = temp\_image[x + 1, y]$ ;
  - 3 Define  $minX = temp\_image[x - 1, y]$ ;
  - 4 Define  $plusY = temp\_image[x, y + 1]$ ;
  - 5 Define  $minY = temp\_image[x, y - 1]$ ;
  - 6 Define  $gradX = plusX - minX$ ;
  - 7 Define  $gradY = plusY - minY$ ;
  - 8 Define  $g = 1.0f / (1.0f + 0.25f * (gradX * gradX + gradY * gradY))$ ;
  - 9 Write  $edge\_image[x, y] = g$ ;
- 

Step 5 of Algorithm 3 involves dispatching the *LevelSetKernel*, which is described on Algorithm 4. This is the main kernel of this implementation, and is larger than usual so as to minimize the overhead of dispatching multiple kernel calls. The main DRLSE iteration equation described in (1) is implemented in step 12 of Algorithm 4.

The procedures *DistRegPre* and *DiracDelta* used in steps 11 and 12 of Algorithm 4, respectively, are identical to the ones used in [8], so are omitted for brevity. It is worth noting however that *DistRegPre* receives a 4-component

value from *phi\_grad\_image* and returns a value with two components, *x*, and *y*. These *x* and *y* members of the returned results are accessed directly in step 11, for brevity.

---

**Algorithm 3:** Iterate DRLSE (CPU)
 

---

**input** : *input\_image*, *phi\_image*, *edge\_image*, *edge\_grad\_image*, *mu*,  
*lambda*, *alpha*  
**output:** *phi\_image*

- 1 Define *phi\_grad\_image* and *laplace\_image*, both with same dimensions as *input\_image*;
- 2 **for** *i* = 1 to 1000 **do**
- 3     Execute the normalized gradient kernel with *phi\_image* and *phi\_grad\_image*, modifying *phi\_grad\_image*;
- 4     Execute laplace kernel with *phi\_image* and *laplace\_image*, modifying *laplace\_image*;
- 5     Execute *LevelSetKernel* with *mu*, *lambda*, *alpha*, *phi\_image*, *edge\_image*, *edge\_grad\_image*, *phi\_grad\_image*, *laplace\_image*, modifying *phi\_image*;
- 6 **end**
- 7 Return final *phi\_image*;

---



---

**Algorithm 4:** LevelSetKernel (GPU)
 

---

**input** : *mu*, *lambda*, *alpha*, *tau*, *phi\_image*, *edge\_image*,  
*edge\_grad\_image*, *phi\_grad\_image*, *laplace\_image*  
**output:** Nothing

- 1 Acquire position [*x*,*y*] of the current pixel;
- 2 Define *phi* = *phi\_image*[*x*,*y*];
- 3 Define *phi\_grad* = *phi\_grad\_image*[*x*,*y*];
- 4 Define *edge* = *edge\_image*[*x*,*y*];
- 5 Define *edge\_grad* = *edge\_grad\_image*[*x*,*y*];
- 6 Define *plusX* = *phi\_grad\_image*[*x* + 1, *y*];
- 7 Define *minX* = *phi\_grad\_image*[*x* - 1, *y*];
- 8 Define *plusY* = *phi\_grad\_image*[*x*, *y* + 1];
- 9 Define *minY* = *phi\_grad\_image*[*x*, *y* - 1];
- 10 Define *curvature* = 0.5f \* (*plusX*.*xnorm* - *minX*.*xnorm* + *plusY*.*ynorm* - *minY*.*ynorm*);
- 11 Define *delta*=*DiracDelta*(*phi*);
- 12 Define *increment* = *tau* \* (*mu* \* *dist\_reg* + *lambda* \* *delta* \* (*edge\_grad*.*x* \* *phi\_grad*.*xnorm* + *edge\_grad*.*y* \* *phi\_grad*.*ynorm* + *edge* \* *curvature*) + *alpha* \* *delta* \* *edge*);
- 13 Write *phi*[*x*,*y*] = *phi* + *increment*;

---

## 4 Results and Discussion

### 4.1 LevelSetKernel’s Occupancy

Occupancy is a common performance metric for CUDA programs, and it roughly measures how many of the GPU’s cores are used when the kernel is dispatched [6]. 50% occupancy means that half of the processors were idle during execution. Ideally the programmer aims for 100% occupancy, as it usually means the hardware is being used to its full capacity.

We determined experimentally that the optimal number of threads per block for the LevelSetKernel on the NVIDIA GTX 1060 6GB is 64. In normal situations the GPU’s block scheduler would have been able to just run more concurrent blocks on the same SM, which would still allow for 100% occupancy.

The LevelSetKernel, however, obtains a theoretical occupancy of 62.5%, and observed occupancy of 59%, according to nvprof. This is explained by the fact that the LevelSetKernel is bottlenecked by its usage of 48 registers per thread. With 64 threads per block, the kernel uses a total of 3072 registers per block. Given the hardware limit of 65536 threads per SM, the scheduler is restricted to dispatching a maximum of 20 concurrent blocks per SM. Those 20 blocks, each with 64 threads, lead to a maximum theoretical number of 1280 concurrent threads per SM, compared to the device maximum of 2048 threads per multiprocessor. The ratio of 1280 to 2048 corresponds to the 62.5% theoretical occupancy limit.

The standard recommendation for optimizing scenarios where register usage is preventing 100% occupancy is to limit the number of registers that the compiler may allocate to each thread. Restricting the number of registers used per thread would allow the dispatching of more concurrent blocks to each SM. It would also, on the other hand, lead to the occasional eviction of some of kernel’s used data from fast registers into higher-latency memory (L1 cache). Whether the trade-off is worth it in terms of performance varies greatly with application. To investigate this experimentally, we performed test segmentations on a  $512 \times 512$  pixel input image, where the maximum number of registers per thread were varied from the ideal 48 to 32. Three kernels dispatched from Algorithm 3 were timed, as well as the total time after 1000 complete iterations. The results are displayed on Table 1.

Note that for all trials performed in this work that record execution time, the time required to allocate the required memory, and the time required to transfer data to and from the GPU, when appropriate, has been ignored. Additionally, all data describing execution time of CUDA programs has been collected only after a proper period of “warm-up”, where the CUDA driver performs just-in-time compilation of the kernels and any additional one-time setup procedures.

The data in Table 1 shows that even though occupancy does increase as predicted, the restriction of register count leads to an overall loss of performance, meaning it is not an effective method of optimizing this CUDA program. Also note the fact that the normalized gradient average execution time (NG time) and laplace gradient execution time (Laplace time) are not as affected from the

**Table 1.** Occupancy and performance as a function of register usage

mag reg <sup>a</sup>	used reg LSK <sup>a</sup>	Theor. Occ. (%)	Real Occ. (%)	LSK time <sup>b</sup> ( $\mu$ s)	NG time <sup>b</sup> ( $\mu$ s)	Laplace time <sup>b</sup> ( $\mu$ s)	1000 iter. total (ms)
48	48	62.5	59.2	89.081	37.029	25.772	180.27
46	46	62.5	59.2	96.633	40.095	27.240	179.71
44	40	75.0	70.3	102.96	38.621	23.661	181.52
42	40	75.0	70.5	104.75	38.026	26.425	183.60
40	40	75.0	70.3	111.40	39.729	24.736	189.69
38	38	75.0	70.7	110.43	40.055	26.626	191.60
36	32	100.0	90.4	130.84	39.842	26.606	206.55
34	32	100.0	90.9	128.31	40.649	26.801	208.94
32	32	100.0	91.5	129.60	40.557	26.607	204.58

<sup>a</sup> max reg describes the hard limit set on how many registers a thread could use, and used reg LSK indicates how many registers the LevelSetKernel actually used.

<sup>b</sup> LSK time indicates the average “wall clock” execution time for the LevelSetKernel after 1585000 executions. Data was retrieved from nvprov so standard deviation was inaccessible. NG time describes the analogue for the normalized gradient kernel, and Laplace time describes the analogue for the laplace kernel.

register count restriction as the *LevelSetKernel* average execution time (LSK time). This is due to the fact that the normalized gradient kernel and the laplace kernel use 22 and 20 registers per thread, respectively.

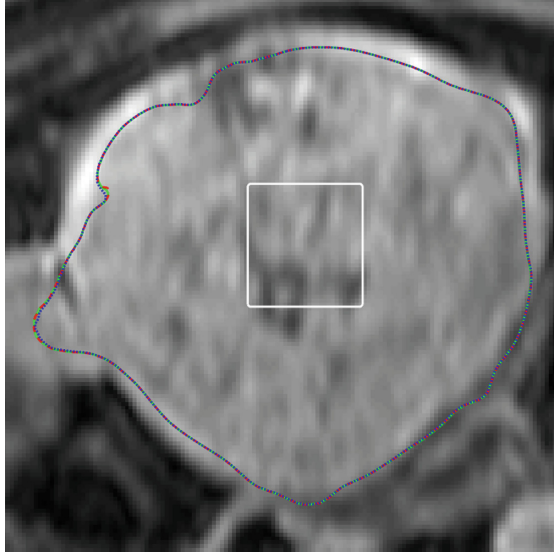
## 4.2 Implementation Verification

With the aim of ensuring that the overall DRLSE implementations in C++ and CUDA matched the reference in [8], a  $512 \times 512$  pixel medical image was segmented with the exact same parameters and initial Level Set contour, and the results were compared. The final zero-level set segmentations after 1000 iterations each are shown in Fig. 1. Visual inspection suggests a near perfect match between the results obtained from all three implementations, shown in Fig. 1.

This mismatch can be explained by two discrepancies between the implementations. The first and greater discrepancy arises from how the three programs behave on calculations that depend on neighboring pixels (such as gradients and convolution results) at the extreme borders of the images. Given that the DRLSE algorithm uses many of such operations, the influence from this discrepancy spreads from the border pixels inwards as the Level Set is iterated.

The second, and lesser source of discrepancies between the results obtained by the three implementations is that the C++ and CUDA versions exclusively use 32-bit floating point numbers, while MATLAB exclusively uses 64-bit floating-point numbers. Despite of this drawback, we intentionally chose this approach for the CUDA implementation due to smaller memory usage and memory transfer time, and the difference in performance: The NVIDIA GTX 1060 GPU provides a theoretical performance of 4.375 TFLOPS (trillion floating point operations per second) on 32-bit floating point numbers, while it provides only 0.137 TFLOPS on 64-bit floating point numbers. This difference in performance alone can lead to a theoretical performance gain of nearly 32 times.





**Fig. 1.** Final zero-level set contours produced from the different implementations. Solid green: CUDA; Dashed red: MATLAB reference; Dotted blue: C++; Solid white: Initial Level Set function for all three scenarios (Color figure online)

### 4.3 Total Memory Usage

Given that the input data is also converted to 32-bit floating point numbers, and the fact that *cudaSurfaceObject\_t* memory (main data structure we used to store image data) is aligned to 512 bytes on the NVIDIA GTX 1060 6 GB, the total memory consumption of the algorithm, in bytes, can be described by (6).

$$Total = 11 \cdot \max(width \cdot 4, 512) \cdot height \quad (6)$$

This means that for a  $512 \times 512$  pixel image, the total memory consumption of the algorithm is roughly 11.5 MB, which corresponds to approximately 0.2% of the total global memory capacity of the NVIDIA GTX 1060 6 GB.

### 4.4 Performance and Time Complexity

For all performance tests, 10 segmentations of 1000 iterations each were performed for each DRLSE implementation, for images of varying sizes. The resulting “wall clock” execution times are displayed in Table 2.

Observing the results in Table 2, the performance advantage of the CUDA implementation over both C++ and MATLAB implementations is immediately apparent. For the larger image size of  $1024 \times 1024$  pixels, the C++ program takes 1700 times as much time as the CUDA program takes to complete, while the MATLAB code takes over 500 times as long as the CUDA program. It should

**Table 2.** Performance comparisons

Image size (square)	MATLAB time (s)	C++ time (s)	CUDA time (s)	MATLAB time/ CUDA time	C++ time/ CUDA time
128	$3.313 \pm 0.048$	$14.465 \pm 0.058$	$0.018 \pm 0.001$	184.1	803.6
256	$8.655 \pm 0.052$	$57.739 \pm 0.274$	$0.044 \pm 0.005$	197.7	1312.3
512	$70.540 \pm 0.396$	$228.892 \pm 2.254$	$0.147 \pm 0.001$	479.9	1557.1
1024	$274.390 \pm 0.0487$	$924.158 \pm 5.376$	$0.547 \pm 0.001$	501.6	1690.0

be noted however, that these results are not strict performance benchmarks, as the MATLAB code has not necessarily been written with performance in mind.

It is also possible to note how the C++ program is almost perfectly linear in time complexity with respect to the number of pixels, meaning time complexity  $O(\text{width} * \text{height})$ , as one would expect of a purely single-threaded serial implementation. The same cannot be observed from the data collected from MATLAB, as it likely internally uses parallelization mechanisms such as Single-Instruction, Multiple Data (SIMD).

The data referring to the CUDA implementation suggests a time complexity close to linear. This can also be observed on the execution time ratio when compared to the purely linear C++ implementation, as image size increases. This behavior is due to how even though the SM execute the kernel purely in parallel at first (conferring some level of independence to the dimensions of the input data), the behavior changes once the hardware is saturated. Once all SM are occupied, the remaining kernel blocks to execute are placed in a queue, and executed once SM become free at a constant rate. As the image size increases, the initial independence to the dimensions of the input data becomes progressively less significant, and the time complexity approaches linearity.

## 5 Conclusions and Further Work

The CUDA implementation developed presents significant performance advantages over a purely single-threaded C++ implementation, and the reference MATLAB implementation [8]. Additionally, the CUDA program developed is capable of performing 1000 iterations in under a second, enough to segment even  $1024 \times 1024$  pixel images. This fact allows the segmentation of medical images with in an interactive regime, which is critical for the trial and error procedure involved in configuring the parameters of the DRLSE algorithm.

## References

1. Nuruozi, A., et al.: Medical image segmentation methods, algorithms and applications. IETE Techn. Rev. **31**(3), 199–213 (2014)
2. Taha, A.A., Hanbury, A.: Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool. BMC Med. Imaging **15**(1), 29 (2015)

3. Zhang, K., Zhang, L., Song, H., Zhang, D.: Reinitialization-free level set evolution via reaction diffusion. *IEEE Trans. Image Process.* **22**(1), 258–271 (2013)
4. Li, C., Xu, C., Gui, C., Fox, M.D.: Distance regularized level set evolution and its application to image segmentation. *IEEE Trans. Image Process.* **19**(12), 3243–3254 (2010)
5. NVIDIA: NVIDIA CUDA programming guide. Version: 10.1.2.243 (2019). <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Accessed 26 Jan 2020
6. Cheng, J., Grossman, M., McKercher, T.: *Professional CUDA C Programming*. Wiley, Indianapolis (2014)
7. NVIDIA: NVIDIA Tesla P100 the most advanced datacenter accelerator ever built featuring Pascal GP100, the world’s fastest GPU. Version: 01.1 (2014). <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>. Accessed 26 Jan 2020
8. Li, C.: Reference implementation for the distance regularized level set evolution (DRLSE) algorithm. <http://www.imagecomputing.org/~cml/DRLSE/>. Accessed 26 Jan 2020
9. Erickson, B.J., et al.: Radiology data from the cancer genome atlas liver hepatocellular carcinoma [TCGA-LIHC] collection. The Cancer Imaging Archive (2016). <https://doi.org/10.7937/K9/TCIA.2016.IMMQW8UQ>. Accessed 26 Jan 2020
10. Clark, K., et al.: The cancer imaging archive (TCIA): maintaining and operating a public information repository. *J. Digit. Imaging* **26**(6), 1045–1057 (2013)