Commun.Fac.Sci.Univ.Ank.Series A2-A3
Volume 58, Number 2, Pages 50-69 (2016)
DOI: 10.1501/commua1-2_0000000096
ISSN 1303-6009

ACCELERATION OF THE EDGE STRENGTH FUNCTION ON GPU USING CUDA

Hacer YALIM KELEŞ

Ankara University, Faculty of Engineering, Department of Computer Engineering, 06830, Gölbaşı, Ankara, TURKEY
E-mail: hkeles@ankara.edu.tr

(Received: October 11, 2016; Accepted: December 23, 2016)

ABSTRACT
Edge strength function (ESF) generates a family of level curves that evolves under the influence of curvature motions. This function is proved to be useful in representing images in computer vision for analysis and recognition purposes in different contexts. Computation of the ESF requires solving a partial differential equation, hence computationally costly. In this work, we present two parallel implementations of ESF that work on Graphics Processing Units (GPU) using Compute Unified Design Architecture (CUDA). Both implementations reduce the computational time significantly with respect to their serial counterpart. The implementations differ mainly in the type of memory that is utilized for accessing data; the first approach utilizes the shared memory and the second one utilizes the texture memory. We obtain between 40 to 65 times speedup in the shared memory based implementation and between 35 to 55 times in the texture memory based implementation with respect to the single threaded CPU implementation. The amount of speedup changes depending on the data size..

KEYWORDS: Edge strength function; parallelization; general purpose programming on graphics processing units; compute unified device architecture.

1. INTRODUCTION

Edge strength function (ESF) is offered by (Tari *et al.*, 1997) as an alternative tool for curve evolution. It is a modified form of the Ambrossio-Tortorelli approximation (Ambrossio and Tortorelli, 2003) to the Mumford-Shah functional (Mumford and Shah, 1989) that uses large diffusion values and interprets a smoothed distance function (Erdem and Tari, 2009). The function generates a family of level curves that evolves under the influence of curvature motions. ESF is proved to be useful for extracting essential shape characteristics and used in different computer vision tasks, such as object

recognition via axis based image representation (Aslan and Tari, 2005), image segmentation (Erdem *et al.*, 2005), part embedding (Keles *et al.*, 2012), and critical points detection (Keles and Tari, 2015). A sample image and an ESF of this image are depicted in Figure 1.

Figure 1. Top: A sample image, bottom: ESF of the image.

This research is performed to improve our preliminary shape grammar implementation, which is a multi-core evolutionary approach that solves the embedding problem for sketches (Keles, 2015). In this work, we had to use a smoothed version of the standard distance transform instead of the ESF for part embeddings, due to its high computational cost. As a follow up research, to solve this efficiency problem, we have designed two efficient parallel solutions for ESF computations. The motivation for this improvement comes from the fact that ESF is reported to be more accurate for representing intrinsic characteristics of shapes than the standard distance transform and its smoothed variants (Keles *et al.*, 2012).

In this work, we provide the details of two different parallel implementations of the ESF that run on many-core GPU architectures. We evaluate the performance of the proposed implementations with respect to our serial, single threaded CPU implementation. The rest of the paper is organized as follows. In Section 1.1, we introduce the edge strength function and its discrete domain solution in detail. Following that, an overview of the related works on GPUs is provided. Then, in Section 2, we present the two CUDA based solutions in detail. Finally, in Section 3, we compare the performances of the two parallel implementations with respect to the single threaded CPU implementation.

1.1. The Edge Strength Function

The edge strength function, which will be depicted here as v, generates an exponentially decaying, smoothed distance field. It is defined as the minimizer to the following energy functional (Aslan and Tari):

$$\frac{1}{2}\iint \rho\|\nabla v\|^2 + \frac{v^2}{\rho}dxdy \quad (1)$$

Subject to $v = 1$ on the shape boundary. Here ρ is a small number which controls the level of smoothing.

On a binary drawing, where each pixel on the drawing is represented as 1 and the rest as 0 in a bounded image domain Ω, it is computed by solving the following PDE (Tari *et al.* 1997):

$$\frac{\partial v}{\partial t} = \left(\nabla^2 - \frac{1}{\rho^2}\right)v \quad (2)$$

Where, $\frac{\partial v}{\partial n} = 0$ on $\partial\Omega$. In this equation, $\partial\Omega$ is the boundary of the image plane and n is the normal direction on the boundary.

The discrete version of the equation (2) can be written using the finite difference approximations, where central differences are used for spatial derivatives and forward differences are used for temporal derivatives. In this equation, ∇^2 is the Laplacian operator, which is approximated in the ESF computations using the 3x3 convolution kernel that is shown below:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Then, the solution becomes:

$$\frac{v_{i,j}^{k+1} - v_{i,j}^{k}}{\Delta t} = v_{i+1,j}^{k} + v_{i-1,j}^{k} + v_{i,j+1}^{k} + v_{i,j-1}^{k} - \left(4 + \frac{1}{\rho^2}\right)v_{i,j}^{k} \quad (3)$$

Where, $i, j \in \Omega$ are the spatial indexes and k is the iteration number.

At the beginning, v^0 is initialized with the original image. During the iterations, the pixel values that belong to the drawing/shape are kept constant and have their original value; i.e. 1. For convergence, Δt needs to be less than 0.25 (Tari *et al.*, 1997); in our experiments, we set Δt to 0.2.

1.2. Related Works

Data parallel computation on commodity graphics processors has started with the fixed function graphics pipeline in the early 2000's. Early works utilized graphics API functions to solve PDEs for non-linear diffusion (Rumpf and Strzodka, 2001) and levelsets (Lefohn *et al.*, 2003).

The extended capabilities of the fixed function graphics pipeline, through programmable vertex and fragment processors, enabled general purpose programming on GPUs more convenient using the shader languages like Cg (Fernando and Kilgar, 2003), Direct3D's shading language (HLSL), OpenGL shading language (GLSL). Programming a GPU using shading languages was still based on graphics pipeline, yet it was easier to utilize the computational power of GPUs for general purpose programming. This attracted many researchers in the field to optimize computationally expensive algorithms by redesigning the data structures and the data flow for these GPU architectures (Keles *et al.*, 2006; Rudomin *et al.*, 2005; Ruiz *et al.*, 2008).

CUDA provides a programming model which enables utilization of massive data parallelism through the abstractions for thread groups, shared memory and thread synchronization with a minimal set of extensions to C programming language (Owens *et al.*, 2008). This framework provides opportunities for developing massively parallel desktop applications that run on GPUs to solve performance problems for various scientific domains, such as physics (Gremse *et al.*, 2016), material sciences (Jimenez and Ortiz 2016), and applied mathematics (Reis *et al.*, 2016).

Solving partial differential equations is a computationally demanding task; therefore there are some ongoing research activities in the design and implementation of PDE solvers that run in parallel. One of the early works that solves a nonlinear diffusion model on GPUs belongs to Rumpf and Strzodka (2001). The aim in this work is smoothing images based on a modified Perona-Malik model (Perona and Malik, 1990). Their solution is based on a finite element scheme implemented using texture hardware where graphics pipeline is utilized to modify the data that is stored as a texture through texture processing operations.

Sanderson et al. (2009) present a framework that solves PDEs of advection-reaction-diffusion models on GPU. The aim of this work is to create spatio-temporal patterns that can be used for texture synthesis and the visualization of vector fields. They implemented their GPU based solver with fragment programs where the data is stored as a texture and the graphics pipeline functions are utilized to access and modify data.

In the literature, more recent works provide solutions to some reaction-diffusion problems using CUDA. Molnar *et al.* (2011) generated simulations for four different reaction-diffusion problems in 3d; they report 5-40 speedup in reference to their single threaded CPU implementation. Similarly, Holmen and Foster (2014) focused on 3d reaction-diffusion simulations to solve the diffusion of a chemically inert compound using CUDA. Their focus is on

improving the performance of single iteration; they reported 8.69x speedup in reference to their multi-threaded CPU based implementation.
Recently, D'Ambra and Filippone (2016) presented a GPU based solution to the image segmentation problem that solves the Ambrossio-Tortorelli model. Their approach is based on solving a system of two coupled elliptic PDEs by a generalized relaxation method. In the implementation, they use the GPU extensions of the Parallel Sparse Basic Linear Algebra Subprograms (PSBLAS) library.
Although there are GPU based solutions to some reaction-diffusion problems in different domains, to the best of our knowledge, a parallel implementation to the ESF, as it is presented in Tari *et al.* (1997), has not been presented in the literature, yet. The details of our solution are provided in Section 2.

2. MATERIALS AND METHODS

In this section, we discuss the design of our CUDA based solutions to the ESF computation. Two different approaches are implemented in CUDA, regarding the GPU memory utilization: (1) shared memory, (2) texture memory. In both approaches, the same CPU-CUDA implementation is used as a framework; only the kernel resource initializations and invoked kernels are changed depending on the configuration. The flow chart of the framework is depicted in Figure 2.

In this framework, the image is first copied from the host (CPU) memory to the device (GPU) memory, to be used in the ESF computations over multiple iterations. The number of iterations is configured by the user in order to evaluate the performance of the algorithms for a range of iterations. During these iterations, the evolved ESF of the image is kept in the device memory all the time. The details of the two solutions will be made clear in the upcoming sections.

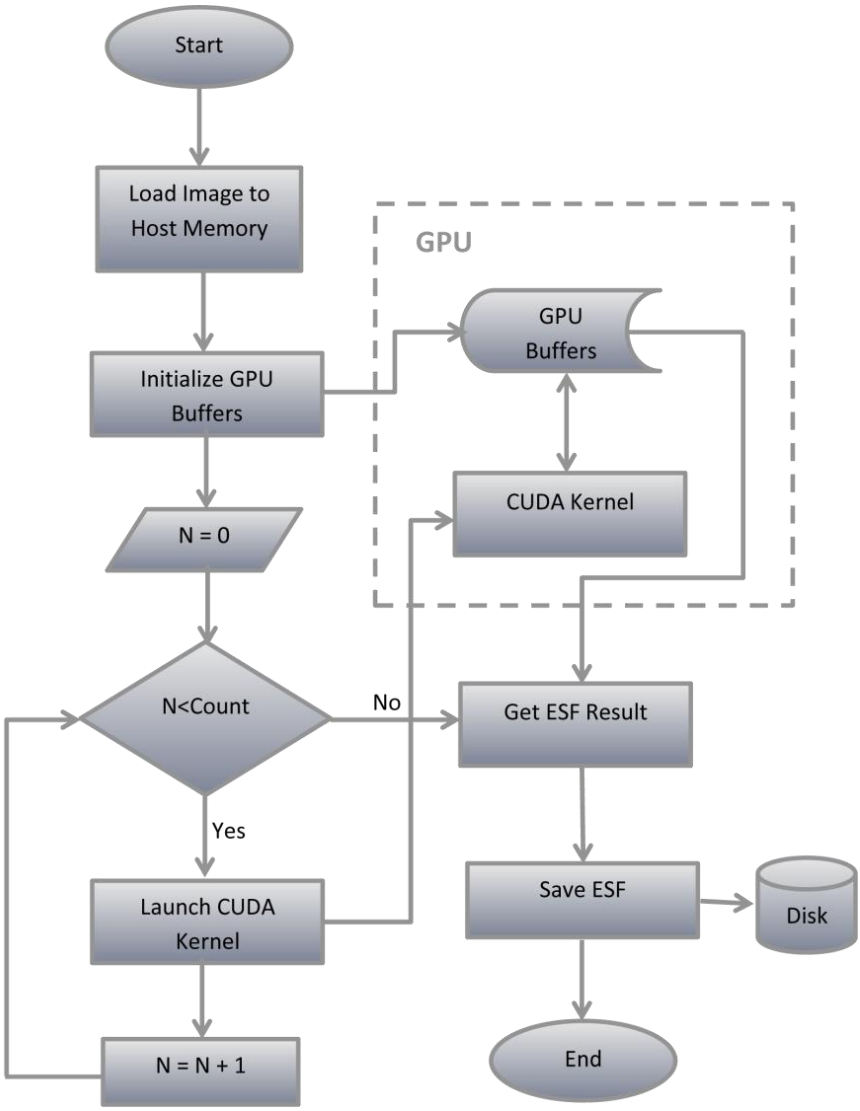

Figure 2. Flow chart of the ESF computation framework.

2.1. Shared Memory Implementation

Shared memory provides data sharing among the threads in the same block, hence it is more efficient to load and access the data that is used by multiple threads in a block into the shared memory instead of accessing it through the

global memory. In the CUDA supported GPUs, the access times of shared memory is 100x faster than the global memory; hence, efficiency increases when a thread needs to access multiple data elements in the same kernel. In the ESF computation, for convolution computation, each pixel is accessed 9 times. Even when the kernel is optimized by excluding the pixels that correspond to the zero coefficients, each pixel is read at least 5 times.
In our solution, the image is semantically represented as a set of tiles. The related image contents corresponding to each tile are copied to the shared memory. Copy operation is performed in parallel as well. We represent the two dimensional input image as a one dimensional array in the global memory of the GPU; while image tiles are represented as 2d arrays in shared memory. In our implementation, each thread copies four pixels (i.e. floating point values) from the global memory; the pixels that reside at the upper left, upper right, lower left and lower right corners of the pixel corresponding to the kernel center. When all the threads in a block complete this operation, the content of a tile is loaded into the shared memory. In order to apply convolution at each pixel in a tile, we also need to copy the apron pixels to the boundaries of the shared memory. The convolution function in the ESF requires replication of the boundary values for the apron pixels at the image boundary (Figure 3). Moreover, the apron pixels for the inner tiles need to be copied from the content of the neighboring tile boundaries (Figure 4).

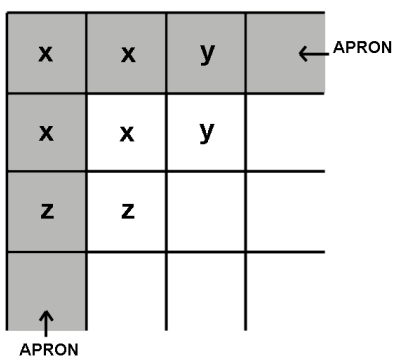

Figure 3. Apron pixels on the boundaries.

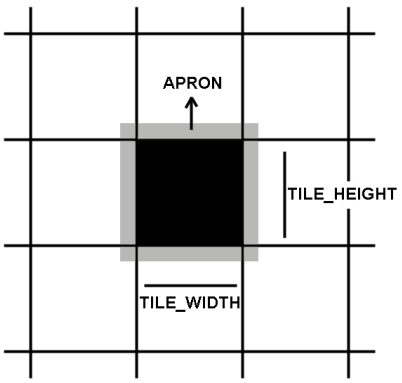

Figure 4. Apron pixels of the inner tiles.

After the thread synchronization, the updated ESF values are computed for each pixel in parallel (Algorithm 1).

As it is explained in the commented sections of Algorithm 1, each thread copies 4 pixels; namely the upper left, upper right, lower left and lower right sections, to keep all the threads active for copying the whole tile content to their proper positions in the shared memory. In order to keep pseudocode compact, we provide the algorithms that set the shared memory content in the Appendix A. The apron pixels at the image boundaries are checked explicitly in each of them and copied from the proper positions, in order to satisfy the boundary conditions so that the gradients at the image boundary are zero.

Algorithm 1. Pseudocode of the Shared Memory Kernel

```
1: procedure SharedESF (dData = ESF values, width = width of data,
height = height of data, lambda = λ, dt = Δt)

2: sData ←allocate 2D shared memory of size [TILE_H + 2][TILE_W + 2]
3: gx  ←threadIdx.x + blockIdx.x * blockDim.x
4: gy  ←threadIdx.y + blockIdx.y * blockDim.y
5: data_idx ←  gy * width + gx
6: offset ←  width
7: bx ←threadIdx.x + 1  //Kernel Radius is 1
8: by ←threadIdx.y + 1  //Kernel Radius is 1
9: if data idx < 0 OR data idx ≥ width * height then return
10: end if
11: sData[by - 1][bx - 1]←Copy upper left (Refer to Algorithm UL in
Appendix A)
12: sData[by - 1][bx + 1]←Copy upper right (Refer to Algorithm UR in
Appendix A)
13: sData[by + 1][bx - 1] ←Copy lower left (Refer to Algorithm LL in
Appendix A)
14: sData[by + 1][bx + 1] ←Copy lower right (Refer to Algorithm LR in
Appendix A)
15: __syncthreads()
16: L←sData[by][bx-1] + sData[by][bx+1] + sData[by+1][bx]+
       sData[by-1][bx] – 4 * sData [by][bx]
17: eps←1e-5
18: prev_val←sData[by][bx]
19: if prev_val ≤ 1.0f + eps AND prev_val ≥ 1.0f - eps then
20:    dData[data_idx] ←1.0f
21: else
22:    dData[data_idx] ←prev_val + (dt * (L - (lambda * prev_val)))
23: end if
24: end procedure
```

As it is explained in the commented sections of Algorithm 1, each thread copies 4 pixels; namely the upper left, upper right, lower left and lower right sections, to keep all the threads active for copying the whole tile content to their proper positions in the shared memory. The apron pixels at the image boundaries are checked explicitly in each of them and copied from the proper

positions, in order to satisfy the boundary conditions so that the gradients at the image boundary are zero.

2.2. Texture Memory Implementation

In this approach, a 2d floating point texture is utilized for the Laplacian computation. The advantage of using texture memory is that texture memory is cached on-chip and texture caches are optimized for thread operations, where memory access patterns depict spatial locality. Therefore, accessing the cached texture memory improves the kernel performances and reduces the memory traffic considerably.
In this approach, a CUDA array, which has the same width and height with the input image, is allocated and bound as the texture memory. CUDA arrays optimize the cache coherence for filtering functions, so that reading the data from the upper and lower rows and neighboring columns are efficient. Moreover, it is not necessary to consider the tile apron pixels explicitly; because this is handled automatically by *tex2D* function. Hence, the implementation with texture memory is more trivial for the ESF computation. Initially, the input image is on the host memory. It is copied from the host memory to the allocated CUDA array, which is on the device memory. For a number of iterations; (1) a CUDA Kernel is launched which first computes the Laplacian of the current image using the previously computed ESF values that are stored on the texture buffer (Algorithm 2), (2) the Laplacian and the previous values of the ESF are used to compute the updated ESF values conforming to the Equation (3).

Algorithm 2. Pseudocode of the Texture Memory Kernel

```
1: procedure textureESF (foatText : texture<float; 2> = current ESF
values, dData=updated ESF values, width = width of data, height =
height of data, lambda = λ, dt = Δt)

2: gx ← threadIdx.x + blockIdx.x * blockDim.x
3: gy ← threadIdx.y + blockIdx.y * blockDim.y
4: data_idx ← gy * width + gx
5: if data_idx < 0 OR data_idx ≥ width * height then return
6: end if
7:  L ← tex2D(foatTex, gx-1, gy) + tex2D(foatTex, gx+1, gy) +
tex2D(foatTex, gx, gy-1) + tex2D(foatTex, gx, gy+1) - 4*tex2D(foatTex,
gx, gy)
8: eps ← 1e-5
9: prev_val ← tex2D(foatTex, gx, gy)
10: if prev_val ≤ 1.0f + eps AND prev_val ≥ 1.0f - eps then
11:     dData[data_idx] ← 1.0f
```

```
12: else
13:     dData[data_idx] ← prev_val + (dt * (L - (lambda * prev_val)))
14: end if
15: end procedure
```

3. RESULTS AND DISCUSSIONS

In this section, we present the performances of the two CUDA implementations with respect to our single threaded, serial CPU implementation. We selected two images with different characteristics for performance evaluations. The first image is a Seljuk pattern from Kayseri (Anatolia) that is composed of thin, crowded line stripes. The second image is the Lena image in binary form that is composed of large segments of foreground and background parts (Figure 5).

Figure 5. Test images: on the left, Seljuk pattern from Kayseri (Anatolia); on the right, binary Lena image.

The Seljuk pattern has been used embedding parts in a shape grammar related research (Keles *et al.*, 2012). Such drawings are common in computational design field. In this work, the ESF is used extensively to transform the images to a weighted domain and the algebra defined on that domain is used to operate on the weighted representation of the images. This transformation enables efficient part searching in a given shape. In the second scenario, we utilize ESF to remove the noise and fill in the pixel gaps by smoothing that the ESF provides. The selected test image is the popular Lena image. The resultant

ESF images for both patterns are shown in Figure 6. In both of the examples, ρ is set to 64 and ESF is generated after 50 iterations.

Figure 6. On the left, ESF of the Seljuk pattern; on the right, ESF of the Lena image.

Image size is the bottleneck for the ESF computations; hence, we provide the performances of the parallel implementations for 4 different image sizes for the test images. In order to evaluate the performances, we use the *speedup* metric. Speedup is computed as the ratio of the execution time on CPU to the one that we obtain on GPU. It is depicted in Equation (4). Here, T_{GPU} is the total execution time of the ESF kernels on GPU and T_{CPU} is the total execution time of the ESF implementation on CPU. We did not include the memory allocation and deallocation times, yet we keep track of the execution times of the kernel calls for a specified number of iterations. All the experiments are performed for 50 times and their average is reported.

$$S = \frac{T_{CPU}}{T_{GPU}} \tag{4}$$

The experiments on the CUDA kernels are performed on an NVidia GeForce GTX 970 graphics card and the CPU implementation is tested on a 4GHz Intel i7 processor. In order to evaluate the speedup performances, we prepared a totally white image as a benchmark. A white image is the most saturated image for ESF computations; hence the diffusion computation is not performed for any of the pixels. The speedup obtained from this image demonstrates the level of performance improvement by merely subdividing the data into tiles on GPU. The speedup from our benchmark images in various sizes are depicted in Figure 7.

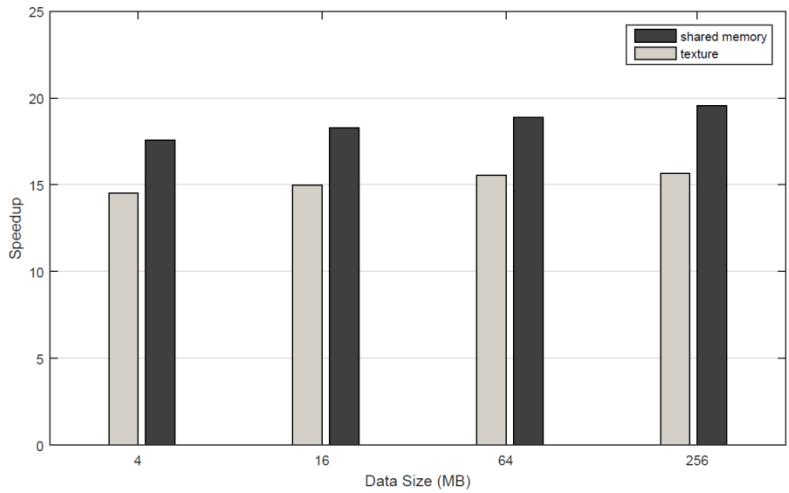

Figure 7. Speedup obtained using the benchmark images, for 200 iterations.

The speedup for the benchmark images show two things: (1) the speedup improves slightly as the data increases, (2) the shared memory implementation runs faster than the texture memory implementation. Although the memory access in cached texture memory is fast, the transfer of updated ESF values from global GPU memory to *cuda array* creates an overhead; hence, results in memory-bound execution times (Algorithm 3). The data transfer, which is performed in each iteration, is necessary since texture memory is cashed and read-only. The speedup is between 15 to 20 for the shared memory implementation and around 15 for the texture memory implementation.

Algorithm 3. Pseudocode for Texture Memory Kernel Call

```
1: procedure callTextureKernel // d array: CUDA array that is bind as
2d float texture , d image: data in global GPU memory

2: copy data from d_image to d_array. // device to device copy
3: bind d array as floatTex
4: for a number of iterations: N do
5:      call textureESF Kernel
6:      copy data from d_image to d_array. // device to device copy
7:      bind d_array as floatTex
8: end for
9: end procedure
```

3.1. Experiments with the Seljuk Pattern Images

The crowded and overlapping line stripes in Seljuk Patterns make them ideal for ESF performance tests, because in each tile there are both foreground and background pixels, there is a high branch divergence in almost every tile.

Table 1. Execution times of the Seljuk patterns for 50 iterations (in milliseconds). SM: Shared memory implementation, TM: Texture memory implementation.

Size (MB)	CPU	GPU (SM)	GPU (TM)
4	267.7	6.2	7.0
16	1076.3	23.9	25.7
64	4319.5	92.6	101.3
256	17266.4	360.8	410.3

The execution times of the Seljuk patterns in various sizes are depicted in Table 1. These values are obtained for 50 iterations. The corresponding speedup obtained with the GPU implementations are shown graphically in Figure 8.

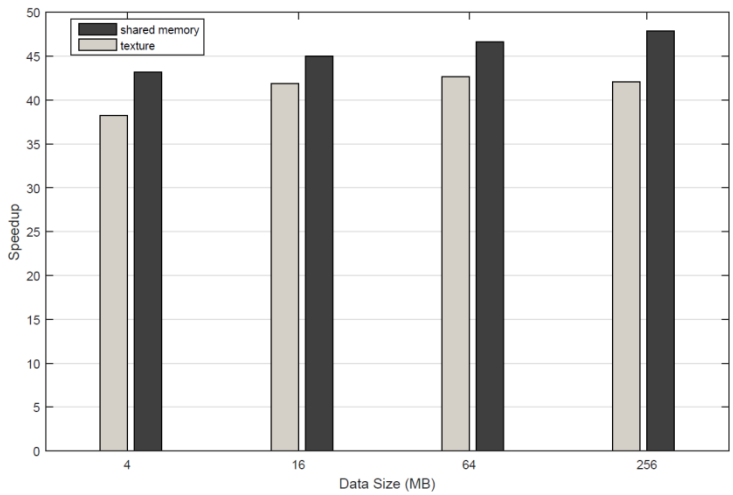

Figure 8. Speedup obtained with the Seljuk patterns, for 50 iterations.

Execution time of the ESF increases when we increase the data size, proportionately for both CPU and GPU; yet there is still a slight improvement in speedup. When the size of the data is small, i.e. 4 MB, speedup does not

increase as fast due to rather inefficient utilization of the data parallelism of the GPU. Since the ESF kernels are memory-bound, i.e. they require accessing multiple memory locations, yet do not contain dense arithmetic computations; the memory access cost becomes more dominant in the overall execution time (Table 2).

Table 2. Execution times of the Seljuk patterns for 200 iterations (in ms).

Size (MB)	CPU	GPU (SM)	GPU (TM)
4	1070.2	25.3	26.8
16	4309.2	91.5	102.3
64	17270.3	363.9	404.6
256	69295.6	1441.8	1641.3

3.2. Experiments with the Lena Images

The execution times for the ESF computation using the Lena images are depicted in Table 3. The test results show that the speedup for the Lena images are relatively higher than the ones that we get from Seljuk images (Figure 9). It is due to the decrease in branch divergence in Lena images, compared to the Seljuk pattern; because there is a relatively more coherent distribution of foreground and background pixels in the Lena images. Note that, the bigger sized Lena images are created by scaling the image. Therefore, when we scale the image, the foreground region is doubled while tile dimensions in the kernels are fixed. This enables the threads in the same block, hence in the same tile, to execute the same branch most of the time. This is not the case for the complicated line stripes in Seljuk image, since the Seljuk tiles are extended to the bigger sizes without scaling.

Table 3. Execution times of the Lena images for 200 iterations (in ms).

Size (MB)	CPU	GPU (SM)	GPU (TM)
4	1042.6	24.9	26.6
16	5016.1	91.5	103.9
64	23192.1	359.2	412.8
256	86319.7	1406.1	1667.3

The tests in Lena images also show that when the data size is increased to 256MB, the average speedup do not improve further. Although the speedup in this size is not better than 64MB, it is still higher than the other configurations. We believe that the slight decrease in speedup is due to the

change in the tile alignments when the data is doubled in size. The speedup gets better than 64MB, in this particular test image, in this size, when the number of iterations is increased to 400 or more.

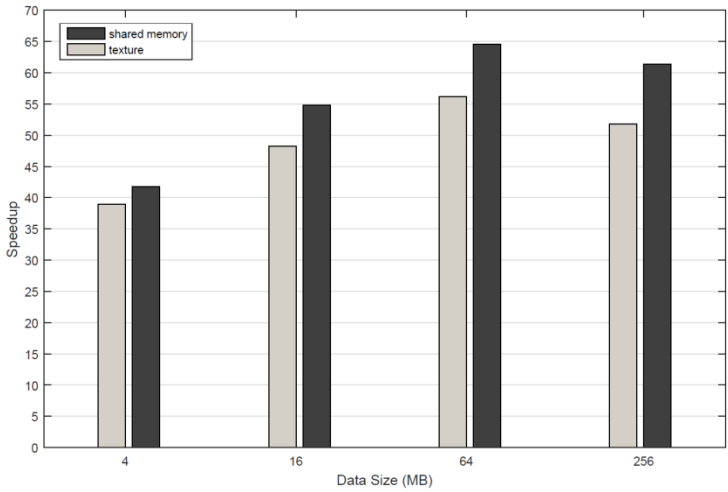

Figure 9. Speedup obtained with the Lena images, for 200 iterations.

4. CONCLUSION

In this research, we present two GPU implementations of the ESF using CUDA, namely the shared memory and the texture memory implementations; and compared their performances. The test images are selected considering the domains that the ESF is utilized more frequently, and the experiments are planned on varied sizes of these images. Our experiments show that, the speedup in the shared memory implementation is higher in all the experiments, due to the efficient block level data sharing among the treads in the same tile and faster access times of shared memory. Although the access times in the texture memory is fast with cached data, there is an overhead for moving data from the global memory to the texture memory in each iteration. Both of the GPU implementations result in significant speedups with respect to our single threaded CPU implementation; the improvement is at least 35 times in the texture memory based implementation and more than 40 times in the shared memory implementation. Speedup increases as the data size is increased.

The methods developed in this research are utilized in a shape grammar interpreter to improve computation times of part embeddings. Moreover, the

proposed CUDA based ESF solutions will also be used in an upcoming research on a novel shape segmentation algorithm.

Acknowledgements
This research has been funded by Ankara University, Scientific Research Projects Grant 15H0443009. The author would like to thank to Prof. Dr. Mine Özkar who draw the original Seljuk pattern.

Appendix A.

Algorithm 4. Setting Upper Left (UL)

```
1: procedure UL
  // dData = ESF values (global memory), sData = ESF values (shared
memory)
  // width : imagewidth; offset ← width
  // gx ← threadIdx.x + blockIdx.x* blockDim.x
  // gy ← threadIdx.y + blockIdx.y* blockDim.y
  // bx ← threadIdx.x +1, by   threadIdx.y +1
  // data_idx ← gy*width+gx
2: if gx - 1 < 0 AND gy - 1 < 0 then // image boundary: top-left pixel
3:    sData[by - 1][bx - 1] ← dData[data_idx]
4: else if gx - 1 < 0 then // image boundary: pixel before the first
column
5:    sData[by - 1][bx - 1] ← dData[data_idx - offset]
6: else if gy - 1 < 0 then // image boundary: pixel below the _rst
row
7:    sData[by - 1][bx - 1] ← dData[data_idx - 1]
8: else
9:     sData[by - 1][bx - 1] ← dData[data_idx - offset - 1]
10: end if
11: end procedure
```

Algorithm 5. Setting Upper Right (UR)

```
1: procedure UR
  // dData = ESF values (global memory), sData = ESF values (shared
memory)
  // width : imagewidth; offset ← width
  // gx ← threadIdx.x + blockIdx.x* blockDim.x
  // gy ← threadIdx.y + blockIdx.y* blockDim.y
  // bx ← threadIdx.x +1, by   threadIdx.y +1
  // data_idx ← gy*width+gx
2: if gx + 1 > width - 1 AND gy - 1 < 0 then // image boundary: top-
right pixel
3:    sData[by - 1][bx + 1] ← dData[data_idx]
4: else if gy - 1 < 0 then // image boundary: pixel below the first
row
5:    sData[by - 1][bx + 1]   dData[data_idx + 1]
6: else if gx + 1 > width - 1 then // image boundary: pixel beyond
the last column
7:    sData[by - 1][bx + 1] ← dData[data_idx - offset]
8: else
```

```
9:    sData[by - 1][bx + 1] ← dData[data_idx - offset + 1]
10: end if
11: end procedure
```

Algorithm 6. Setting Lower Left (LL)

```
1: procedure LL
  // dData = ESF values (global memory), sData = ESF values (shared
memory)
  // width : imagewidth; offset ← width
  // gx ← threadIdx.x + blockIdx.x* blockDim.x
  // gy ← threadIdx.y + blockIdx.y* blockDim.y
  // bx ← threadIdx.x +1, by   threadIdx.y +1
  // data_idx ← gy*width+gx
2: if gx - 1 < 0 AND gy + 1 > height - 1 then // image boundary:
lower-left pixel
3:    sData[by + 1][bx - 1] ← dData[data_idx]
4: else if gx - 1 < 0 then // image boundary: pixel before the _rst
column
5:    sData[by + 1][bx - 1] ← dData[data_idx + offset]
6: else if gy + 1 > height - 1 then // image boundary: pixel beyond
the last row
7:    sData[by + 1][bx - 1] ← dData[data_idx - 1]
8: else
9:    sData[by + 1][bx - 1] ← dData[data_idx + offset - 1]
10: end if
11: end procedure
```

Algorithm 7. Setting Lower Right (LR)

```
1: procedure LR
  // dData = ESF values (global memory), sData = ESF values (shared
memory)
  // width : imagewidth; offset ← width
  // gx ← threadIdx.x + blockIdx.x* blockDim.x
  // gy ← threadIdx.y + blockIdx.y* blockDim.y
  // bx ← threadIdx.x +1, by   threadIdx.y +1
  // data_idx ← gy*width+gx
2: if gx + 1 > width - 1 AND gy + 1 > height - 1 then // image
boundary: lower-right pixel
3:    sData[by + 1][bx + 1] ← dData[data_idx]
4: else if gx + 1 > width - 1 then // image boundary: pixel beyond
the last column
5:    sData[by + 1][bx + 1] ← dData[data_idx + offset]
6: else if gy + 1 > height - 1 then // image boundary: pixel beyond
the last row
7:    sData[by + 1][bx + 1] ← dData[data_idx + 1]
8: else
9:    sData[by + 1][bx + 1] ← dData[data_idx + offset + 1]
10: end if
11: end procedure
```

REFERENCES

[1] Ambrosio, L. and Tortorelli, V. (2003). On the approximation of functionals depending on jumps by elliptic functionals via Γ-convergence. *Communications on Pure Applied Mathematics*. 43(8): 999-1036; doi: 10.1002/cpa.3160430805.

[2] Aslan, C. and Tari, S. (2005). An Axis-Based Representation for Recognition, In: 10'th IEEE International Conference on Computer Vision; 17-21 Oct. 2005; Beijing, PRC, pp: 1339- 1346; doi: 10.1109/ICCV.2005.32.

[3] D'Ambra, P. and Filippone, S. (2016). A parallel generalized relaxation method for high-performance image segmentation on GPUs. *Journal of Computational and Applied Mathematics*. 293: 35-44; http://dx.doi.org/10.1016/j.cam.2015.04.035.

[4] Erdem, E., Erdem, A., Tari, S. (2005). Edge strength functions as shape priors in image segmentation. *Energy Minimization Methods in Computer Vision and Pattern Recognition*. Volume 3757 of the series Lecture Notes in Computer Science pp. 490-502; doi: 10.1007/11585978_32.

[5] Erdem, E., Tari, S. (2009). Mumford-Shah Regularizer with Contextual Feedback. Journal of Mathematical Imaging and Vision. 33: 67-84; doi: 10.1007/s10851-008-0109-y.

[6] Fernando, R., Kilgar, M. (2003). Cg: The Cg Tutorial. AddisonWesley, New York. ISBN: 9780321545398 0321545397.

[7] Gremse, F., Höfter, A., Razik, L., Kiessling, F., Naumann, U. (2016). GPU-accelerated adjoint algorithmic differentiation. *Computer Physics Communications.* 200: 300-311; http://dx.doi.org/10.1016/j.cpc.2015.10.027.

[8] Holmen, J.K., Foster, D.L. (2014). Accelerating Single Iteration Performance of CUDA-Based 3D Reaction-diffusion Simulations.

International Journal of Parallel Programming. 42: 343-363; doi: 10.1007/s10766-013-0251-z.

[9] Jimenez, F., Ortiz, C.J. (2016) A GPU-based parallel Object kinetic Monte Carlo algorithm for the evolution of defects in irradiated materials. *Computational Materials Science*. 113: 178-186; http://dx.doi.org/10.1016/j.commatsci.2015.11.011.

[10] Keles, H.Y., Es, A., Isler, V. (2006). Acceleration of direct volume rendering with programmable graphics hardware. *The Visual Computer.* 23: 15-24; doi: 10.1007/s00371-006-0084-5.

[11] Keles, H.Y., Ozkar, M., Tari, S. (2012). Weighted shapes for embedding perceived wholes. *Environment and Planning B: Planning and Design.* 39: 360-375; doi: 10.1068/b37067.

[12] Keles, H.Y. and Tari, S. (2015) A robust method for scale independent detection of curvature-based criticalities and intersections in line drawings. *Pattern Recognition*. 48: 140-155; http://dx.doi.org/10.1016/j.patcog.2014.07.005.

[13] Lefohn, A.E., Kniss, J.M., Hansen, C.D., Whitaker RT. (2003). Interactive Deformation and visualization of level set surfaces using graphics hardware. In: Proceedings of the 14th IEEE Visualization, Washington, DC, USA; pp.11-19; doi: 10.1109/VISUAL.2003.1250357.

[14] Molnar, Jr. F., Izsak, F., Meszaros, R., Lagzi, I. (2011) Simulation of reaction diffusion processes in three dimensions using CUDA. *Chemometrics and Intelligent Laboratory Systems*. 108: 76-85; http://dx.doi.org/10.1016/j.chemolab.2011.03.009.

[15] Mumford, D., Shah, J. (1989). Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure Applied Mathematics*. 42(5): 577-685; doi: 10.1002/cpa.3160420503.

[16] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C. (2008). GPU Computing, Proceedings of the IEEE. 96(5): 879-899; doi: 10.1109/JPROC.2008.917757.

[17] Perona, P. and Malik, J. (1990). Scale space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 12(7): 629-639; doi: 10.1109/34.56205.

[18] Reis, R.F., Loureiro, F.S., Lobosco, M. (2016). 3D numerical simulations on GPUs of hyperthermia with nanoparticles by a nonlinear bioheat model. *Journal of Computational and Applied Mathematics*. 295: 35-47; http://dx.doi.org/10.1016/j.cam.2015.02.047.

[19] Rudomin, I., Millan, E., Hernandez, B. (2005). Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory*. 13: 741-751; http://dx.doi.org/10.1016/j.simpat.2005.08.008.

[20] Ruiz, A., Guil, N., Ujaldon, M. (2008). Recognition of circular patterns on GPUs: Performance analysis and contributions. *Journal of Parallel and Distributed Computing*. 68: 1329-1338; http://dx.doi.org/10.1016/j.jpdc.2008.05.010.

[21] Rumpf, M. and Strzodka, R. (2001). Nonlinear diffusion in graphics hardware. In: Proceedings of EG/IEEE TCVG Symposium on Visualization, pp. 75-84; doi: 10.1007/978-3-7091-6215-6_9.

[22] Sanderson, A.R., Meyer, M.D., Kirby, R.M., Johnson, C.R. (2009). A framework for exploring numerical solutions of advection-reaction-diffusion equations using a GPU-based approach. *Computing and Visualization in Science*. 12: 155-170; doi: 10.1007/s00791-008-0086-0.

[23] Tari, Z.S., Shah, J., Pien, H. (1997). Extraction of shape skeletons from grayscale images. *Computer Vision and Image Understanding*. 66: 133-146; 10.1006/cviu.1997.0612.