Distance Transform Separable by Mathematical Morphology in GPU

Francisco de Assis Zampirolli* and Leonardo Filipe

Universidade Federal do ABC, São Paulo, Brazil
{fzampirolli,leonardo.filipe}@ufabc.edu.br
http://www.ufabc.edu.br

Abstract. The Distance Transform (DT) is one of the classical operators in image processing, and can be used in Pattern Recognition and Data Mining, and there is currently a great demand for efficient parallel implementations on graphics cards, known as GPU. This paper presents simple and effective ways to implement the DT using decompositions of erosions with structuring functions implemented on GPU. The DT is equivalent to a morphological erosion of the binary image by a specific structuring function. However, this erosion can be decomposed by a sequence of erosions using small structuring functions. Classical and efficient algorithms of the DT are implemented on CPU. New 1D and 2D algorithms are implemented on GPU, using decomposition of structuring functions, inspired by implementations of convolution filters. All the GPU implementations used in this paper are known as *brute-force*, and even then present excellent results, comparable to the best CPU algorithms, which might contribute to future applications in image processing.

Keywords: Distance Transform, Mathematical Morphology, GPU.

1 Introduction

The Distance Transform (DT) [12,2] is an important algorithm in image processing because it can be used in many other transformations, such as dilation, erosion, the shortest path between two pixels, skeleton, SKIZ (Skeleton of Influence Zone), Voronoi diagram, Delaunay triangulation, Grabriel Graph, pattern matching, image compression, etc. [4,18,11,14,5]. Besides being a basic operator in image processing, it helps in the study of other similar algorithms, such as watershed [17] and IFT (Image Floresting Transform) [6]. Thus, improving the efficiency of DT makes it possible to improve the efficiency of similar operators. The DT can also be computed by a sequence of local operations, using 3×3 or one-dimensional neighborhoods, making the algorithms simpler and faster.

DT implementations can be classified by the method it employs to raster of the pixels in the image. *Sequential algorithms* perform very well, but it is not possible to calculate the Euclidean DT (EDT) using only this type of algorithm [12,2,16]. *Parallel algorithms* for the DT can be implemented using parallel architectures,

* This research is sponsored by FAPESP (Process: 2009/14430–1) and CAPES.

J. Ruiz-Shulcloper and G. Sanniti di Baja (Eds.): CIARP 2013, Part I, LNCS 8258, pp. 41–48, 2013.
© Springer-Verlag Berlin Heidelberg 2013

and are the most intuitive form of implementation. Such parallel algorithms are also able to compute the EDT [12,2,16,8,15], but usually perform poorly on single processor architectures. It is also possible to calculate the EDT using *propagation algorithms*, which use *queue* structures to store the pixels that might have their values changed in a given iteration. Such pixels are called *boundary* pixels [16,20,9].

Mathematical Morphology is an area based on set theory. This theory is heavily applied in image processing, with the basic operators of dilation and erosion, in which the neighborhood can be defined by structuring functions [1].

There is a relationship between EDT and Mathematical Morphology, and the goal of this paper is to study this relationship using parallel programming on GPU (*Graphics Processing Unit*), exploring several implementations of EDT. Furthermore, comparisons are made with the most efficient CPU algorithms with a parallel version of the algorithm defined in Lotufo and Zampirolli [9].

Previous works have focused in accelerating the computation of EDT using GPUs, achieving good results, as in Schneider et al. [13]. However, their work still uses *DirectX*, a computer graphics API, to implement the algorithm, instead of a proper GPU computing framework. Also, their approach does not use Mathematical Morphology to define the EDT.

2 Methods

A two-dimensional *binary image* is a function f that maps the *elements* (or *pixels*) of a space E in $\{0, k\}$, where E is usually a matrix. The position of a pixel is given by their position in the array. Thus, the line x and column y of the pixel is associated with point (x, y) of the Cartesian plane. Then, any distance function defined on the Cartesian plane induces a distance function in the field of the image [3]. For a given distance function, the *Distance Transform* (DT) assigns to each pixel of an object from a binary image the smallest distance between these pixels and background pixels. Consider any two finite and non-empty sets E and K. A ψ *operator* of E and K is defined as a mapping of E in K and denoted $\psi : E \to K$ or $\psi \in K^E$. A *digital image*, or simply *image*, is defined as a function of the K^E lattice. Thus, if f is an image then $f \in K^E$. Consider E the *domain* of the image, which is *one-dimensional* if $E \subset Z$, where Z is the set of the integers, and *two-dimensional* if $E \subset Z^2$.

2.1 Decomposition of the Structuring Function

Some properties of Minkowski operators produce a method for the *decomposition of a structuring element* [19]. For example, one dilation by a 3×3 structuring element is equivalent to perform two uni-dimensional dilations, one 1×3 and one 3×1. The result of a decomposition is a *generalized Minkowski sum*, defined as $B_G = B_1 \oplus \cdots \oplus B_k$, where $\{B_1, \cdots, B_k\}$ are the elements in which B_G can be decomposed. Thus,

$$\varepsilon_{B_G}(f) = \varepsilon_{B_k}(\cdots(\varepsilon_{B_1}(f))\cdots). \tag{1}$$

These procedures of decomposition are useful for implementations of erosion and dilation [7] and will be addressed in this paper. To illustrate the following equation of erosion, an erosion algorithm is defined by $\forall x \in E$,

$$\varepsilon_b(f)(x) = \min\{f(y) \dot{-} b(y-x) : y \in B_x \cap E\}, \tag{2}$$

where b is a *structuring function* defined on B with $b : B \to Z$. If the elements of b are nonzero, b is called a *non-flat structuring function* or *non-planar structuring function*. Let $v \in Z$, we define $t \to t \dot{-} v$ in K [7]. For the Equation 2, with an input image with a domain E of dimensions $h \times w$, the algorithm performs the erosion in $\Theta(hw)$ time. By Huang and Mitchell [8], considering Equations 1 and 2, applying the erosion several times using varying structuring functions, as the one shown in Equation 3, the EDT is computed. This process is defined in Algorithm 1.

$$b_i = \begin{bmatrix} -4i+2 & -2i+1 & -4i+2 \\ -2i+1 & \mathbf{0} & -2i+1 \\ -4i+2 & -2i+1 & -4i+2 \end{bmatrix}, \tag{3}$$

where the origin, at the center, is bold and $i \in \{1, 2, \ldots\}$.

ALGORITHM 1: Euclidean DT: $g = EDT(f)$

1: Calculates the EDT of f
2: $i = 1$;
3: **while** $f \neq g$ **do**
4: b_i is defined by Equation 3;
5: $g = f$;
6: $f = ero(g, b_i\}$; by Equation 2.
7: $i++$;
8: **end while**

The convergence of Algorithm 1 occurs due to the idempotent property of the erosion when considering these particular structuring functions. In this algorithm, the structuring function changes with each iteration, at line 4. If no pixel has its value changed, then the algorithm has converged. Depending on the image, the amount of necessary erosions to achieve convergence may vary. In the worst case, an image with a single 0 value at one end of one of its diagonals, $\sqrt{h^2 + w^2}$ erosions are necessary, where h the image height and w the width. Knowing that the erosion runs in $\Theta(hw)$ time, once again considering a small neighborhood of b size, the EDT has $O(hw\sqrt{h^2 + w^2})$ complexity. Assuming $h = w$, we can simplify and say that the complexity is $O(h^3)$.

2.2 Separable Convolution Using Shared Memory Using the GPU

Graphics Processing Units (GPUs) are coprocessors specialized in generating computer graphics. For several years, their graphics pipelines only allowed the use of fixed functions to render a set of primitives, such as lines and triangles, to create computer graphics. More recently, however, to allow for more realistic

graphics, several parts of that pipeline have become programmable. With such programmable pipelines, it became possible, with small changes to GPU architectures, to use the rapidly increasing processing power within them to solve general computing problems. To allow for that *General Purpose GPU computing (GPGPU Computing)*, technologies such as the *CUDA architecture* and the *OpenCL API* were created. This work focuses in the use of the CUDA architecture, through the *CUDA C* library for *ANSI C*, on nVidia GPUs to create new implementations for the EDT.

In a GPU, consider a 16×16 block that can be stored on a block of shared memory, a much faster kind of memory. A (x, y) pixel within an image can be accessed through the following conversion of the thread and block indexes, provided as built-in variables by the CUDA API: $x = threadIdx.x + blockIdx.x * blockDim.x$ and $y = threadIdx.y + blockIdx.y * blockDim.y$, where $(threadIdx.x, threadIdx.y)$ represents a pixel within the block, indexed by the values $blockIdx.x * blockDim.x$ and $blockIdx.y * blockDim.y$. Thus, if an image processing problem can be solved by analyzing a neighborhood stored on a portion of shared memory, the (x, y) pixels are transferred from the global memory to this efficient memory. In this example, the image must be subdivided in subimages of 16×16 size. These divisions increase the complexity of implementing algorithms that rely on access to neighbor pixels. The problem becomes even worse when dealing with global problems, such as the EDT or *labeling*.

Most of that extra difficulty introduced when using the shared memory comes from the fact that, for each block of threads, there is a separate portion of this memory. Pixels on the border of a shared memory block will have neighbors stored on another block, which are inaccessible to the thread assigned to the current pixel. There might even be missing neighbors if the pixel is not only on the border of a block, but also on the border of the image. Convolution filters and morphological operators share several similarities. For example, the decomposition of strucutirng functions based on the presented Minkowski sum is similar to the problem of convolution separability. Taking these similarities into account, it is interesting to analyze existing GPU implementations for the convolution filter, in order to learn from their shared memory management and try to improve the performance of the EDT implementations.

An algorithm for the separable bi-dimensional convolution will be presented as a two-step uni-dimensional algorithm. The code for this algorithm can be found in the CUDA SDK library. The convolution filter is an image processing technique that is mostly used for pre-processing, in order to remove noise and obtain a smoother image. In addition, it can be used for edge detection in objects. As the convolution and morphological operators work with neighborhoods, the border processing on blocks need special treatment. To minimize these border operations, this border must be as small as possible. Thus, it is more efficient to have a border with a thickness of one pixel. For a convolution, the border is initialized with the 0 (zero) value. For an erosion, as the operation is performed using the neighboring minimum, this border must be initialized with the maximum value supported by the used image type. The separable convolution

implemented in [10] has two steps: In the first, the load phase, data from the global memory is transferred to shared memory. The second step performs filtering and writes the results back to the global memory. The filtering step also occurs in two stages, filtering the lines first, and then filtering the columns.

3 Results

3.1 EDT on the GPU Using Erosions and Shared Memory

In the same way as the Algorithm 1 for EDT, and the Equation 2, the EDT is implemented using successive erosions by varying structuring functions using shared memory on GPU. In this first algorithm, each thread copies a pixel and its neighbors from the global memory to their corresponding places in the space of shared memory of the thread's block. This way, each 16×16 image block is stored in a 18×18 block of shared memory (in the case of a 3×3 structuring function). The calculation of the minimum value in this pixel is done in Algorithm 2, making this algorithm inefficient. One solution would be to calculate the erosion for all pixels in a block. We have also found that an erosion in 16×16 blocks can be implemented using two 16×1 and 1×16 erosions, requiring less operations.

Algorithm 2: Erosion using shared memory: $g(x) = \varepsilon_b(f)(x)$, where $x = [x0][y0]$ and $[tx][ty]$ is the offset in a block

```
 1: tx = threadIdx.x; ty = threadIdx.y; // offset in block
 2: x0 = blockIdx.x*blockDim.x+ tx;
 3: y0 = blockIdx.y*blockDim.y+ ty; // offset in image
 4: data[18][18]; // allocates shared memory define data[x][y] of f[x0][y0]
 5: if border block[tx][ty] then
 6:     if border f[x0][y0] then
 7:         data[x][y] = MAX;
 8:     else
 9:         data[x][y] = f[x0][y0];
10:     end if
11: end if
12: data[tx+1][ty+1] = f[x0][y0];
13: g[x0][y0] = erosion in data[tx+1][ty+1];
14: ...
```

3.2 EDT on the GPU Using 1D Erosions on Shared Memory

The algorithm presented in this section computes the EDT in two steps. In the first one, the EDT is calculated for the columns using a sequential algorithm, on the CPU. On the second step, the EDT is computed for the lines, using a brute force algorithm on the GPU until convergence.

In the second part, the GPU's shared memory is used, in a similar fashion to the 1D convolution found in the SDK, `convolutionSeparable` [1]. This algorithm

[1] Source: `http://developer.nvidia.com/cuda-toolkit-sdk`

was inspired by the LZ algorithm [9]. By Equation 3, b_i can be decomposed into four structuring functions of one dimension, two vertical *North* (b_{Ni}) and *South* (b_{Si}), and two horizontal *East* (b_{Ei}) and *West* (b_{Wi}):

$$b_{Ni} = \begin{bmatrix} -2i+1 \\ \mathbf{0} \end{bmatrix}, b_{Ei} = \begin{bmatrix} \mathbf{0} & -2i+1 \end{bmatrix}, b_{Si} = \begin{bmatrix} \mathbf{0} \\ -2i+1 \end{bmatrix}, b_{Wi} = \begin{bmatrix} -2i+1 & \mathbf{0} \end{bmatrix}. \tag{4}$$

This part of the algorithm is calculated through successive erosions by the structuring functions $b_{E1}, b_{E2}, \ldots$, and $b_{W1}, b_{W2}, \ldots$ until stabilization, due to the idempotence property. The LZ implementation uses queue structures to store the pixels that could be altered in an erosion, minimizing the necessary operations. In order to efficiently use the shared memory, the algorithm to compute the successive erosions for the lines uses structuring functions that have the same dimension as the *thread* blocks. Thus, using blocks with a 16×16 dimension and considering erosions in a single dimension, the structuring function to be used will have a 1×16 dimension. Consider $b_{l_i} = [-2i+1 \quad \mathbf{0} \quad -2i+1]$ where the origin, at the center, is bold and $i \in \{1, 2, \ldots\}$. It is possible to define b_{G_1} of 1×16 dimension, as $b_{G_1} = b_{l_1} \oplus \cdots \oplus b_{l_8}$. To generalize this equation, consider $b_{G_k} = b_{l_{8(k-1)+1}} \oplus \cdots \oplus b_{l_{8(k-1)+8}}$, where $k \in \{1, 2, \ldots\}$. The b_{G_1} structuring function will be used on the first iteration of the 1D erosion on the columns. On the second iteration, the b_{G_2} structuring function, also of 1×16 dimension, will be used, and so on. Refer to Figure 1.

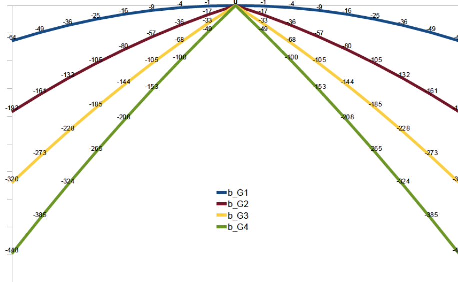


Fig. 1. Illustration of the structuring functions b_{G_1}, b_{G_2}, b_{G_3} e b_{G_4}

The second algorithm (as in Algorithm 1) computes successive erosions on the GPU, with varying structuring functions $b_{G_1}, b_{G_2}, \ldots$, using the sequential erosion as its input image f. With this 1×16 structuring function, each thread performs erosion on eight pixels, instead of a single one, as in Algorithm 2, improving the efficiency of the algorithm.

4 Conclusions

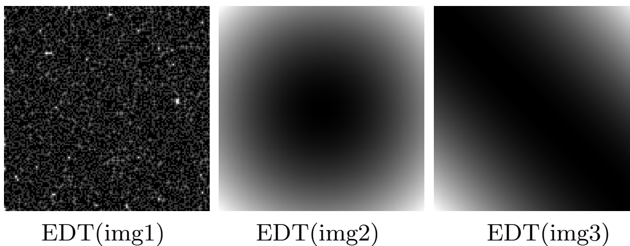
The algorithms presented in this paper were compiled and executed on the following computer: *MacBook OS X - v.10.6.5 - 1.26GHz Intel Core 2 Duo*, with *2GB RAM*, and a *NVIDIA GeForce 9400M* GPU.

The table 1 shows the performance of the algorithms presented in this paper when applied to three images, as shown in Figure 2. The classic and efficient `Eggers-CPU` algorithm runs on the CPU [5]. The `1D-LZ-CPU` algorithm runs on the CPU and was described in [9]. The `2D-GPU`, `1D-GPU` and `1D-LZ-GPU` algorithms presented in this paper compute the EDT using the GPU and its shared memory. The `2D-GPU` algorithm considers a 3×3 neighborhood. The `1D-GPU` algorithm decomposes the structuring function in 1×16 and 16×1 dimensions. The `1D-LZ-GPU` version computes the EDT for the image lines using a sequential algorithm running on the CPU, while its second part uses the GPU to compute the EDT for the columns.

Table 1. Execution times of several algorithms applied to different images (time in seconds)

	512×512			1024×1024		
	$img1$	$img2$	$img3$	$img1$	$img2$	$img3$
$Eggers - CPU$	0.051	0.018	0.022	0.201	0.095	0.095
$1D - LZ - CPU$	0.014	0.012	0.303	0.138	0.077	2.494
$2D - GPU$	0.012	0.625	0.626	0.048	4.903	4.905
$1D - GPU$	0.018	0.152	0.224	0.063	1.148	1.717
$1D - LZ - GPU$	0.013	0.083	0.088	0.057	0.614	0.654

Analyzing this table we observe good performance on the GPU implementations for the image `img1`, Figure 2. This is due to the low number of iterations (erosions) since the objects for computation of the EDT are small. As for the images `img2` and `img3`, the number of required erosions to compute the EDT is high, and these GPU implementations need improvement and/or the use of machines with increased processing power, such as the TESLA GPUs. Even so, the `1D-LZ-GPU` implementation already performs comparatively well against `Eggers-CPU`. It should also be noted that all the GPU implementations used in this paper are known as *brute-force*, and even then have results comparable to the best CPU algorithms for some kinds of images.



EDT(img1) EDT(img2) EDT(img3)

Fig. 2. EDT obtained from input images: img1, img2 and img3 (refer to the text)

References

1. Banon, G.J.F., Barrera, J.: Decomposition of mappings between complete lattices by mathematical morphology, Part I: general lattices. Signal Processing 30, 299–327 (1993)
2. Borgefors, G.: Distance transformations in digital images. Computer Vision, Graphics and Image Processing 34, 344–371 (1986)
3. Breu, H., Gil, J., Kirkpatrick, D., Werman, M.: Linear time euclidean distance transform algorithms. IEEE - TPAMI 17(5), 529–533 (1995)
4. Danielsson, P.E.: Euclidean distance mapping. Computer Graphics and Image Processing 14, 227–248 (1980)
5. Eggers, H.: Two fast euclidean distance transformations in z^2 based on sufficient propagation. In: Computer Vision and Image Understanding (1998)
6. Falcao, A.X., Stolfi, J., Lotufo, R.A.: The image foresting transform: theory, algorithms, and applications. IEEE Transactions Pattern Analysis and Machine Intelligence 26(1), 19–29 (2004)
7. Heijmans, H.J.A.M.: Morphological Image Operators. Acad. Press, Boston (1994)
8. Huang, C.T., Mitchell, O.R.: A euclidean distance transform using grayscale morphology decomposition. IEEE Transactions on Pattern Analysis and Machine Intelligence 16, 443–448 (1994)
9. Lotufo, R.A., Zampirolli, F.A.: Fast multidimensional parallel euclidean distance transform based on mathematical morphology. In: Proceedings of SIBGRAPI, pp. 100–105 (2001)
10. Podlozhnyuk, V.: Image convolution with cuda. In: nVidia (2007)
11. Ragnemalm, I.: Neighborhoods for distance transformations using ordered propagation. In: CVGIP: Image Understanding (1992)
12. Rosenfeld, A., Pfalz, J.L.: Distance functions on digital pictures. Pattern Recognition 1, 33–61 (1968)
13. Schneider, J., Kraus, M., Westermann, R.: GPU-based real-time discrete euclidean distance transforms with precise error bounds. In: International Conference on Computer Vision Theory and Applications, pp. 435–442 (2009)
14. Sharaiha, Y., Christofides, N.: Graph-theoretic approach to distance transformations. Pattern Recognition Letters (1994)
15. Shih, F.Y.C., Mitchell, O.R.: A mathematical morphology approach to euclidean distance transformation. IEEE Trans. on Image Processing 1, 197–204 (1992)
16. Vincent, L.: Morphological algorithms. Mathematical Morphology in Image Processing Edition. vol. Marcel-Dekker, ch. 8, pp. 255–288. E. Dougherty (September 1992)
17. Vincent, L., Soille, P.: Watersheds in digital spaces: an efficient algorithm based on immersion simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence 13(6), 583–598 (1991)
18. Vincent, L.: Exact euclidean distance function by chain propagations. In: IEEE Int. Computer Vision and Pattern Recog. Conf., Maui, HI, pp. 520–525 (June 1991)
19. Wang, X., Bertrand, G.: An algorithm for a generalized distance transformation based on minkowski operations. In: 9th International Conference on Pattern Recognition, vol. 2, pp. 1164–1168 (November 1988)
20. Zampirolli, F.A., Lotufo, R.A.: Classification of the distance transformation algorithms under the mathematical morphology approach. In: Proceedings of SIBGRAPI, pp. 292–299 (2000)