Published in final edited form as:

J Real Time Image Process. 2020 October 28; 17(5): 1255–1266. doi:10.1007/s11554-019-00886-7.

Real-Time Thinning Algorithms for 2D and 3D Images using GPU processors

Martin G. Wagner

Department of Medical Physics, University of Wisconsin, 1111 Highland Avenue, Madison, WI 53705

Abstract

The skeletonization of binary images is a common task in many image processing and machine learning applications. Some of these applications require very fast image processing. We propose novel techniques for efficient 2D and 3D thinning of binary images using GPU processors. The algorithms use bit-encoded binary images to process multiple points simultaneously in each thread. The simpleness of a point is determined based on Boolean algebra using only bitwise logical operators. This avoids computationally expensive decoding and encoding steps and allows for additional parallelization. The 2D algorithm is evaluated using a dataset of handwritten characters images. It required an average computation time of 3.53 ns for 32×32 pixels and 0.25 ms for 1024×1024 pixels. This is 52 to 18,380 times faster than a multi-threaded border-parallel algorithm. The 3D algorithm was evaluated based on clinical images of the human vasculature and required computation times of 0.27 ms for $128 \times 128 \times 128$ voxels and 20.32 ms for $512 \times 512 \times 512$ voxels, which is 32 to 46 times faster than the compared border-sequential algorithm using the same GPU processor. The proposed techniques enable efficient real-time 2D and 3D skeletonization of binary images, which could improve the performance of many existing machine learning applications.

Keywords

Centerline; GPU Programming; Medial Axis; Skeletonization; Thinning

1 Introduction

The skeletonization of binary images is a common task in image processing applications. The resulting skeleton, often also referred to as medial axis or medial lines, reduces the complexity of an object while retaining its topology. Arguably, one of the most common application is optical character recognition (OCR), where handwritten or printed letters are classified based on their skeletons [1,2]. In [3,4,5] skeletons are used for automatic

Terms of use and reuse: academic research for non-commercial purposes, see here for full terms. <https://www.springer.com/aam-terms-v1>

mwagner9@wisc.edu.

Publisher's Disclaimer: This Author Accepted Manuscript is a PDF file of an unedited peer-reviewed manuscript that has been accepted for publication but has not been copyedited or corrected. The official version of record that is published in the journal is kept up to date and so may therefore differ from this version.

chromosome analysis (karyotyping), to identify abnormalities in the morphology. Also for computer vision tasks such as object recognition [6, 7] and object tracking [8] skeletonization is an important tool. Furthermore, in [9] it is used to reconstruct the 3D path of interventional devices from two orthogonal projection images. Other applications include extraction of features for fingerprint analysis [10] and protein structure analysis [11]. Many of the tasks and applications that use skeletonization require real-time image processing. The focus of this work is to propose a fast and accurate thinning algorithm that is suitable for these tasks and discuss its efficient implementation on modern GPUs.

Because of the wide variety of applications that rely on skeletonization, a large number of publications have been dedicated to this topic. Most methods can be classified as thinning, contour or distance map driven algorithms or combinations of these. Contour-based approaches only consider the boundary points of an object. The skeleton is then found e.g. by identifying the opposite boundary point and calculating the center [12] or determining the centers of maximally inscribed disks [13]. In [14] the Voronoi diagram for the boundary points is constructed and subsequently pruned to determine the skeleton. Some contour based techniques allow the extraction of medial lines with subpixel accuracy [13].

The distance transform calculates for each object pixel, the distance to the closest background pixel. The skeleton points can then be determined by identifying and tracking local maxima [15, 16, 17, 18, 19] or using level sets [20]. Contour and distance transform based algorithms often represent elegant solutions to the skeletonization problem. However, it can be difficult to guarantee the connectedness of the skeleton and the preservation of the topology.

Thinning algorithms iteratively remove the outer layer of pixels of an object until only one pixel thin centerlines are remaining. For most thinning approaches, it can be proven that the topology of the object is preserved and that the skeletons of connected objects remain connected. The concept of digital topology, which provides the basis for most thinning algorithms, has been thoroughly studied in [21].

Thinning approaches can generally be divided into sequential, border-sequential, subfield-sequential and border-parallel algorithms [3]. Sequential thinning algorithms process a single pixel at a time. If a pixel along the boundary of the object is simple, i.e. its removal does not alter the topology of the binary image, it is turned into a background pixel. This can be determined e.g. using the Hilditch crossing number [4], which calculates the number of 8-connected components in a local 3×3 neighborhood. In [22] the average outward flux is calculated based on the distance map to determine the order in which pixels are removed.

In order to take advantage of multiprocessor systems parallel thinning algorithms have been proposed. Border-sequential algorithms use multiple subiterations, where only border points along certain directions are removed [3], [23,24,25]. In [23, 25] a set of 3×3 templates is used to determine whether a pixel is simple. Other approaches divide the image into multiple subfields, where in each subiteration only pixels of one subfield are processed [26, 27]. Finally, border-parallel techniques have been proposed, where all pixels can be processed at the same time [28, 29, 30, 31, 32, 33]. A downside of border-parallel thinning algorithms is

that a 3×3 neighborhood is not sufficient to determine whether a pixel can be deleted. Most algorithms [28, 29, 33] use a combination of removal and restoring templates within the local 5×5 neighborhood.

The concept of thinning can also be extended to three dimensions [26, 34, 35, 36, 37, 38]. However, due to the much larger number of possible configurations in the local neighborhood of a voxel and special structures such as “tunnels”, topology preserving thinning in 3D images is a much more complex task [38]. In [35] and [37] 6-subiteration border-parallel approaches are described, where [35] uses a set of $3 \times 3 \times 3$ templates, whereas [37] uses the number of connected components in the local neighborhood to determine if a voxel is simple. A border-parallel technique was proposed in [34] based on a set of $3 \times 3 \times 3$ templates with additional points outside of this neighborhood. Furthermore, a sequential approach is described in [36], which uses a distance map to determine in which order voxels are removed.

This work focuses on fast parallel thinning algorithms for 2D and 3D images. Despite the fact that border-parallel thinning algorithms only require a single subiteration and therefore process a larger amount of pixels with each iteration through the image, they are not necessarily faster than border-sequential algorithms. This is mainly due to the larger neighborhood that has to be considered for every pixel. Within this work, we propose fast parallel thinning algorithms for 2D and 3D images using 4-subiteration and 6-subiteration border-sequential techniques respectively. Additionally, an efficient implementation using general purpose computing on GPU processors (GPGPU) is discussed. An implementation of the proposed techniques can be found at <https://github.com/mwagner9/BEGPUThinning>.

2 General Border-Sequential 2D Thinning

This section gives a brief description of the algorithm suggested in [3], which provides the basis for the technique proposed here. Within the scope of this work the definition of concepts such as simple pixels, connectedness, topology and neighborhoods and their notation is based on [21]. Without loss of generality, it is assumed that the background pixels, herein denoted as ‘black’ or ‘0’ pixels are 4-connected, whereas object pixels (‘white’ or ‘1’) are 8-connected. A binary image is thinned by iteratively removing the border pixels of connected white components in the image. Each iteration consists of four subiterations, where only pixels that are north, south, east or west borders respectively are processed. This ensures that the parallel removal of multiple pixels does not cause skeletons to disconnect or branches to disappear. During one subiteration, each white pixel p in the image is turned into a black pixel if it satisfies all of the following three criteria.

- C2.1** p is a border pixel in the direction of the current subiteration, i.e. its 4-connected neighbor on the north, south, east or west side respectively is a black pixel.
- C2.2** p has more than one white neighbor. This avoids removing already thinned lines and isolated points.
- C2.3** p is adjacent to exactly one 8-connected white component.

Assuming the adjacent pixels in the local 3×3 neighborhood of p are labeled n_0 to n_7 in clockwise orientation starting with the top or north neighbor as illustrated in Fig. 1, C2.1 is satisfied by $n_{2s} = 0$, where s is the index of the subiteration, with $s \in \{0, 1, 2, 3\}$. The second criteria can also be easily verified using

$$\sum_{i=0}^7 n_i > 1 \quad (1)$$

For the third criteria the crossing number [3] can be used which calculates the number of 0/1 crossings when walking along the neighbors of p , where 0 corners can be skipped. This can be written as

$$\sum_{s=0}^3 (n_{2s} \wedge \bar{n}_{2(s+1)}) + (n_{2s+1} \wedge \bar{n}_{2s} \wedge \bar{n}_{2(s+1)}) = 1 \quad (2)$$

where $\wedge$ and $\bar{}$ denote the logical operations ‘and’ and ‘not’ respectively. The thinning process is continued until no white pixels can be removed within a full iteration.

3 Bit-Encoded 2D Thinning Algorithm

The proposed algorithm is a 4-subiteration border-sequential thinning approach based on the strategy described in section 2. It allows efficient and parallelized computation using bit-encoded binary images, i.e. each pixel is represented by a single bit. The bit-encoded processing reduces memory access times, which is further discussed in Section 3.2, and allows for an additional level of parallelization. Instead of decoding and processing, each pixel separately the proposed algorithm processes all pixels encoded in a 32-bit integer in parallel within a single thread. Therefore, all three conditions are verified with a single Boolean equation, which only uses the bitwise logical operators available on common processors (‘and’, ‘or’, ‘exclusive or’, ‘not’). For the north boundary subiteration a pixel can be deleted if it satisfies

$$\bar{n}_0 \wedge \{ [\bar{n}_1 \bar{n}_7 \wedge (n_2 \oplus n_3 \oplus \bar{n}_4) \wedge (n_3 \oplus n_5) \wedge (n_3 \oplus n_6)] \vee [(n_1 \oplus n_5) \wedge (n_2 \oplus n_5) \wedge (n_4 \oplus \bar{n}_5) \wedge \bar{n}_6 \bar{n}_7] \vee (n_2 n_4 \bar{n}_7) \vee (\bar{n}_1 n_4 n_6) \vee (n_2 n_4 n_6) \vee (\bar{n}_1 \bar{n}_2 \bar{n}_3 n_6 n_7) \} = 1 \quad (3)$$

where $\vee$ and $\oplus$ denote the logic operators ‘or’ and ‘exclusive or’ respectively. For better readability we used $n_i n_j$ synonymously with $n_i \wedge n_j$. The corresponding Boolean expressions for all subiterations s can be obtained by substituting all n_i by $n_{(i+2s) \bmod 8}$, where ‘mod’ is the modulo operation. We can prove that (3) satisfies conditions C2.1 through C2.3 by simply evaluating it for all 256 possible configurations as shown in appendix A. In order for a pixel to be removed it only needs to satisfy (3) for the direction corresponding to the current subiteration. The algorithm terminates when no pixel can be removed within a full iteration.

3.1 Hardware Configuration

Although the proposed algorithm could also provide considerable speed improvements using single or multiprocessor CPUs, the focus in this work was to provide an efficient GPU implementation using a *single instruction multiple threads* (SIMT) programming model [39]. The algorithm was implemented using CUDA 9.0 [40] (NVIDIA, Santa Clara, CA, USA) and compiled for compute capability 6.1. All CUDA kernels were configured with 1024 threads (32 threads per warp) and a shared memory size of 48 kB. The evaluation was performed on a six-core 3.6 GHz processor (Xeon E5–1650, Intel, Santa Clara, CA, USA) with 40 GB memory and a 1.58 GHz GPU with 3584 CUDA cores (GeForce GTX 1080 Ti, NVIDIA, Santa Clara, CA, USA) with 11 GB memory.

3.2 Data Access

Within this work, the image data are stored as set of bitwise encoded pixels. This reduces memory access times and allows processing multiple pixels in parallel within a single thread. The image rows are padded to a multiple of 32 pixels. This ensures that each pixel is aligned with its north and south neighbors. Let $\mathbf{p}(i, j)$ be a vector of 32 sequential pixel values stored within a single integer, where j is the row index and the column indices are given by the interval $[i \cdot 32, (i + 1) \cdot 32 - 1]$. Each thread reads a single 32-bit integer $\mathbf{p}(i, j)$ from global memory and stores it in shared memory. The vectors containing its north $\mathbf{n}_0(i, j) = \mathbf{p}(i, j - 1)$ and south $\mathbf{n}_4(i, j) = \mathbf{p}(i, j + 1)$ neighbors can then be accessed using two shared memory reads. The east and west neighbors $\mathbf{n}_2(i, j)$ and $\mathbf{n}_6(i, j)$ are obtained by simply shifting $\mathbf{p}(i, j)$ by a single bit to the left and right respectively. To ensure data consistency for the first and last bit, the adjacent integers $\mathbf{p}(i - 1, j)$ and $\mathbf{p}(i + 1, j)$ have to be included. Therefore, the neighbor vectors $\mathbf{n}_2(i, j)$ and $\mathbf{n}_6(i, j)$ can be obtained by

$$\mathbf{n}_2(i, j) = (\mathbf{p}(i, j) \ll 1) \vee (\mathbf{p}(i + 1, j) \gg 31) \quad (4)$$

and

$$\mathbf{n}_6(i, j) = (\mathbf{p}(i, j) \gg 1) \vee (\mathbf{p}(i - 1, j) \ll 31), \quad (5)$$

where $b \ll x$ and $b \gg x$ denote a bit shift of b by x bits to the left and right respectively. The newer generation of GPUs allows exchanging data between the threads of a single warp using shuffle instructions [40], which can be used to efficiently access $\mathbf{p}(i - 1, j)$ and $\mathbf{p}(i + 1, j)$. Finally, the neighbors $\mathbf{n}_1$, $\mathbf{n}_3$, $\mathbf{n}_5$, and $\mathbf{n}_7$ can be obtained by bit-shifting $\mathbf{n}_0(i, j)$ and $\mathbf{n}_4(i, j)$ in the same way.

3.3 Blockwise Processing

If the whole image is processed within a single CUDA block, global memory access can be reduced to a single read and write per thread. Since each thread processes 32 pixels, the total amount of integer reads and writes can be calculated by the number of pixels divided by 32. The total amount of shared memory that is required per block is

$$\left(\frac{N_{\text{pixel}}}{32} + 32 \right) \cdot 4 \text{ Bytes}, \quad (6)$$

where N_{pixel} is the number of pixels in the image. The additional 32 integers are required to count the removed pixels in each iteration for all 32 warps. This is necessary to determine when the algorithm terminates, i.e. no pixels are removed within a full iteration. To calculate the number of changes within a block, a parallel reduction scheme is used as described in [41]. First, the sum is calculated for each of the 32 warps using shuffle instructions (warp-reduction) and the results are written to a shared memory array. Subsequently, each thread of the first warp reads one of the 32 intermediate results and another reduction step is performed for the first warp only. Due to the shared memory limitation (48 kB per block), the maximum number of pixels that can be processed within a single CUDA block is 392, 192 pixels, which corresponds to an image size of e.g. 766×512 pixels. Larger images have to be divided into multiple blocks. This introduces the disadvantage that the image data in global memory have to be updated after each subiteration. To avoid costly reads and writes to global memory, a technique with overlapping blocks is described here.

Therefore, the image is divided into equally sized and distinct regions R_{ij} with size $r_x \times r_y$ pixels. Additionally, a transition region of o pixels is added in each direction as illustrated in Fig. 2. The total region including R_{ij} and its transition regions is denoted R_{ij}^* . Two adjacent regions R_{ij}^* and $R_{(i+1,j)}^*$ share a common region of $(2o) \times (2o + r_y)$ pixels. Each CUDA block reads a single region R_{ij}^* from global memory and processes it independently for a maximum of o iterations. During these iterations, no global memory update is necessary. The CUDA block then only writes the region R_{ij} without the transition regions to the output buffer in global memory. The transition region is updated by neighboring CUDA blocks, where it is part of the respective central regions as shown in Fig. 2. This process is repeated until the thinning algorithm is completed. For the boundary pixels of the transition regions, it is assumed that its outside neighbors are 0. The described technique creates the same results as if the global memory was updated after each subiteration, but allows independent processing of each block for o iterations and therefore considerably reduces the global memory access. We can prove that the described technique provides accurate results using the following two statements:

- A.1** only a single layer of pixels is removed in each iteration and
- A.2** the removal of a pixel only depends on its local 3×3 neighborhood.

In the beginning of the first iteration, all the data in R_{ij}^* represent valid image data.

According to assumption A.2, as long as the pixel has a valid 3×3 neighborhood, the algorithm can accurately determine whether it should be removed or not. This is true for all pixels except the boundary pixels of the transition regions. I.e. after the first iteration, the outer most layer of pixels in R_{ij}^* is invalid, whereas for all other pixels the result would be the same as if the whole image would be covered by R_{ij}^* . If we continue this logic argumentation, after the k^{th} iteration the outermost k layers of pixels are invalid in R_{ij}^* .

Therefore, it can be concluded that after o iterations the transition regions are invalid, but the inner region R_{ij} is still valid.

3.4 Implementation for Small Images or Blocks

In some applications, preprocessing is applied to divide larger images into small regions, which only contain a single object (e.g. OCR). For these small images with a maximum width and height of 32 pixels, the algorithm can be implemented efficiently without using shared memory. In these cases, the east and west neighbors can be accessed using a single bit-shift without taking into account the adjacent integers. Therefore, (4) and (5) simplify to

$$\mathbf{n}_2(i, j) = \mathbf{p}(i, j) \ll 1 \quad (7)$$

and

$$\mathbf{n}_6(i, j) = \mathbf{p}(i, j) \gg 1. \quad (8)$$

To access the north and south neighbors, the shuffle instructions can be used, since each integer contains a full row of pixels. The same technique can also be applied to larger images with an isometric block size of 32 pixels. However, the global read and write operations have to be split up into bitwise operations in order to be able to generate overlapping blocks in bit-encoding direction. Finally, for smaller images the GPU might not be working at its capacity when processing a single image. Therefore, one should consider processing multiple images simultaneously to increase throughput.

4 Extension to 3 Dimensions

Compared to the 2D case, topology-preserving thinning in three dimensions is a much more complex task. One of the reasons for the increased complexity is the possibility of special topological configurations like ‘*tunnels*’, which make the definition of centerlines and simple voxels difficult [38]. Additionally, larger neighborhoods increase the number of possible configurations, which makes it challenging to determine whether the deletion of a voxel preserves the topology. Three different kind of neighbors are usually considered in three-dimensional problems. A pair of voxels x_1 and x_2 are considered 6-connected neighbors if $\|x_1 - x_2\|_1 = 1$, 18-connected if $\|x_1 - x_2\|_1 = 2$, and 26-connected if $\|x_1 - x_2\|_1 = 3$, where $\|\square\|_1$ denotes the Manhattan distance [42]. Within this work, we assume the background is 6-connected, while the object is 26-connected. The concept of topology and topology preserving thinning in three dimensions has been thoroughly studied in [21, 26, 38, 43]. A voxel v along the boundary surface of an object can be removed from a 3D binary image, if it satisfies all of the following assumptions [26]:

- C3.1** v has exactly one 26-connected object component in its 26-neighborhood,
- C3.2** v has exactly one 6-connected background component in its 18-neighborhood, which is 6-connected to p , and
- C3.3** v has at least one 6-connected background neighbor.

For more details on the characterization of simple points in three dimension, we refer the reader to [26,38]. The algorithm presented here is based on the technique proposed in [35], which uses a set of six $3 \times 3 \times 3$ templates, which can be rotated in 90° steps around the axis orthogonal to the border face of each subiteration. The thinning is performed in 6

subiterations, where a voxel is removed if it matches any of the templates or its rotations. The templates contain four types of elements: 1, 0, '*don't care*', and groups of voxels with at least one 1 ('*one of*'). In order to apply the techniques proposed here for the 2D case, the templates have to be converted to Boolean equations. In general, this can be done for each template T using

$$T = \left(\bigwedge_{\forall i | n_i \in N_0} \bar{n}_i \right) \wedge \left(\bigwedge_{\forall i | n_i \in N_1} n_i \right) \wedge \left(\bigvee_{\forall i | n_i \in N_{\odot}} n_i \right), \quad (9)$$

where N_0 , N_1 , and $N_{\odot}$ are the sets of all neighbors n_i , where the template contains a 0, 1, or '*one of*' respectively. However, in order to include 90° , 180° and 270° rotations of the template, this equation has to be repeated for each rotation of each template. Further optimization reduced the number of operations in the final Boolean equation to 121. As for the 2D case, this equation is calculated using 32-bit integers, which allows the parallel processing of 32 voxels in a single thread. Therefore, less than four operations are required on average to process a single voxel. The full equation is given in appendix B.

Another problem to consider when applying the proposed concepts to three-dimensional images is the limitation of the number of threads and the amount of shared memory per block. The largest block size, where each thread processes a single 32-bit integer is 32^3 voxels, due to the restriction of 1024 blocks per thread. The global read and write operations, however, have to be replaced by multiple 8-bit integer operations in order to enable overlapping blocks in bit-encoding direction. In this case, the overlap is restricted to 16 voxels. A larger block size of 64^3 voxel can be accomplished by processing 8 32-bit integers per thread. This enables 16-bit integer read and write operations to global memory with an overlap of 32 voxels. Finally, using bit encoding in x-dimension would cause all threads in a single warp to read data from different rows, which results in slow memory access times. In order to achieve strided memory access, the image is encoded such that the y-dimension is bit-encoded. This allows all threads within a single warp to read adjacent data from global memory.

5 Evaluation

The performance of the 2D thinning algorithms is evaluated using the NIST database for handwritten character recognition [44]. It contains 810,000 isolated and binarized character images with an image size of 128×128 pixels. For the purpose of this work, we randomly selected 100,000 images from this dataset. Additionally to the original NIST data, sets of images with sizes 32×32 pixels, 512×512 pixels, and 1024×1024 pixels were created. Therefore, the NIST images were scaled using nearest neighbor interpolation. Before downsizing, the images were cropped to avoid excessive scaling of the characters. All images contain at least a one-pixel thin boundary region of zeros. The evaluation of the 3D algorithm is based on retrospectively analyzed clinical data. A set of 11 3D digital subtraction angiography (DSA) images from human subjects was used, which were created from a native and a contrast enhanced 3D cone beam CT acquisition. Subtracting the two images eliminates the anatomical background and only shows the vasculature filled with contrast medium. The dataset contains 9 images of the brain and 2 images of the lung

vasculature with a native resolution of $512 \times 512 \times 512$ voxels. The images were scaled to create three datasets with a resolution of 128^3 voxels, 256^3 voxels, and 512^3 voxels respectively, using trilinear interpolation and subsequent binarization based on global thresholding. For the 2D case, the proposed algorithm was implemented as a single-block (BEGPUs) as well as a multi-block (BEGPU_M) version. The single-block version also allows processing multiple images at the same time to increase the utilization of the GPU. The block size and overlap differed depending on the image size. For small images up to a size of 128×128 pixel, the overlap was set to 0 with a single block containing the whole image. The main difference to the actual single-block implementation is that only a single image is processed at a time. A full list of all block sizes and overlaps is given in Table 1. Additionally, a CPU reference implementation of the border-sequential algorithm (CPUREF) proposed in [3] and a multi-threaded implementation (6 threads corresponding to the number of cores on the CPU) of the border-parallel algorithm (CPUBP) as proposed in [3] were evaluated for comparison. For the 3D case, the proposed algorithm was implemented with a block size of 32^3 voxels and an overlap of 8 voxels (BEGPU₃₂) as well as a version with a 64^3 block size and an overlap of 16 voxels (BEGPU₆₄). For comparison, a reference CPU implementation (CPUREF) of the algorithm proposed in [35] was evaluated as well as a 'naïve' GPU implementation (GPUREF) of the same algorithm using CUDA textures. The performance of each algorithm is measured in terms of the average computation time per image. Since the results of different thinning algorithms can vary depending on the order in which points are removed as well as the technique used to determine simple points, additional quality measures are included to describe the results quantitatively. Therefore, we calculate the number of points, the number of endpoints, the number of segments, the number of junction points, the average number of neighbors per junction point and the average segment length. An increased number of endpoints may indicate that the algorithm is more sensitive to boundary noise, while the total number of points, the number of junction points and the number of neighbors per junction may be indicators for the "thinness" of the centerlines. The number of segments and the average segment length may be influenced by both, boundary noise and thinness.

6 Results

The results of the 2D thinning algorithms are listed in Table 2. For the smallest image size of 32×32 pixels, the proposed single block algorithm requires an average computation time of 3.53 ns per image, which corresponds to over 283 million images per second. This is about 11, 682 times faster than the CPU reference implementation and 18, 380 times faster than the multi-threaded border-parallel algorithm. The largest image size, which can be processed in a single block (512×512 pixel), requires an average computation time of 6.10ps, which corresponds to 164, 043 images per second and is 3163 times faster than the reference CPU implementation and 335 times faster than the multi-threaded border-parallel algorithm. For the set of 1024×1024 images, single block processing was not possible. The multi-block implementation required a computation time of 0.25 ms per image, which corresponds to 3975 images per second and is 605 times faster than the CPU reference implementation and 52 times faster than the multi-threaded border-parallel algorithm. All quantitative measures of the proposed GPU algorithms and the reference CPU implementation results are

equivalent. The results of the border-parallel algorithm contain on average less points, endpoints, segments and junctions than the other approaches for most image sizes except the 128×128 pixel dataset. A visual comparison between the results of all four algorithms is shown in Fig. 3.

The results of the 3D thinning algorithms are shown in Table 3. For the smallest volume size of 128^3 voxels, the proposed GPU algorithm with an isometric block size of 32 voxels is the fastest with a computation time of 0.27 ms per volume. This is 402 times faster than the CPU reference and 32 times faster than the GPU reference implementation. For the largest volume size (512^3 voxels), the proposed algorithm with an isometric block size of 64 voxels is the fastest with a computation time of 20.32 ms. This is 703 times faster than the CPU reference and 46 times faster than the GPU reference. The number of points, endpoints, junctions and segments differ between the different algorithms, however using one-way analysis of variance (ANOVA) [45] no significant difference was found for $p < 0.05$. A visual comparison between the results of each 3D algorithm is shown in Fig. 4.

The presented timing measurements do not include transfer times from host to GPU memory before the thinning algorithm can be applied and from GPU to host memory to transfer the results back. This applies to the presented algorithms as well as the reference GPU implementation. The median data transfer rate from host to GPU was 5.0 GB/s and from GPU to host 5.4 GB/s. This corresponds to an overall data transfer time of 0.37 μ s for a 32×32 image and 0.38 ms for a 1024×1024 image.

7 Discussion

The proposed techniques allow an efficient implementation of 2D and 3D binary image skeletonization and performs between 32 to 18,380 times faster than the compared CPU and GPU implementations. This enables real-time processing for both 2D and 3D applications using consumer graphics cards. The 2D results confirm that processing the complete image within a single CUDA block performs better in terms of computation time than multi-block processing. This is expected since high GPU occupancy rates can be achieved by processing multiple images at the same time and the single-block implementation does not require updating the global memory during the thinning process. Additionally, no overlap is required, which further reduces the amount of reads and writes to global memory. For larger image sizes, multi-block processing is necessary. However, the results show that very fast computation times can still be reached and the proposed algorithm outperforms the multi-threaded border-parallel algorithm by a factor of 46. The thinning results of the proposed algorithms and the reference CPU implementation are equivalent, since all simple pixels with a common border are removed in each subiteration. The removal of a simple pixel does not influence whether other pixels with the same border are simple or not. Therefore the order in which pixel are removed in each subiteration does not matter and equivalent results are expected. The border-parallel algorithm however deletes pixels with different borders in a single iteration, which might cause different results compared to the border-sequential approaches. The results show that the border-parallel algorithm is less sensitive to boundary noise, which causes a smaller number of segments and endpoints. However, it might also

miss small segments as can be seen in Fig. 3, where for the letter ‘d’ a small segment near the starting point of the lower arc is missing.

The 3D results show that for smaller volumes up to a size of 256^3 voxels, a block size of 32^3 voxel with a 16 voxels overlap is faster compared to a block size of 64^3 voxels and a 32 voxel overlap. This is because the overhead of having a more complex CUDA kernel, where each thread reads and processes 8×32 voxels outweighs the advantage of being able to perform more iterations before updating the data in global memory. This changes for larger volumes of 512^3 voxels, where the 64^3 voxel block size implantation is faster.

Unlike the 2D case, where equivalent results are achieved for all border-sequential algorithms, the 3D thinning results differ slightly for different implementations and even block sizes. This is because the templates [35] used to form the Boolean equation and determine whether voxels can be removed, do not represent the full set of simple voxels. This means that even though good results can be achieved using these templates, the order in which voxels are removed within a single subiteration may alter the results slightly. Larger differences between the CPU and any of the GPU implementations can be explained by the in-place implementation, where all pixels are sequentially processed and updated.

8 Conclusion

The bitwise encoding and processing scheme in combination with transition regions for larger images, as proposed in this work, go beyond just increasing the number of threads using SIMT processors. The algorithm allows processing up to 32 pixels in parallel on a single thread and reduces memory access and requirements considerably, resulting in computation times at least one order of magnitude lower than the reference GPU and CPU approaches. Our techniques could be used to implement real-time device tracking in medical images as proposed in [9] without compromising frame rates or causing considerable delays. Additionally, computer vision tasks such as object tracking as described in [8] could benefit from fast skeletonization techniques since the reduction of computation time could be used to increase the frame rate or to perform more complex object detection and tracking approaches.

Acknowledgements

Research reported in this work was supported by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) and by the National Heart, Lung, And Blood Institute of the National Institutes of Health under award numbers R21EB024553 and R01HL116567. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

Appendices

A Verification of 2D Equation

Fig. 5 shows all 3×3 neighborhoods of a pixel which are not simple, i.e. eq. (3) is 0 for all subiterations.

Fig. 5.

Visualization of all non-simple 3×3 neighborhoods, where removal of the center pixel would result in disconnected regions. Background pixels are shown in black, object pixels in white.

Fig. 6 shows all 3×3 neighborhoods of a pixel which are simple, i.e. eq. (3) is 1 for at least one subiteration.

Fig. 6.

Visualization of all simple 3×3 neighborhoods, which may be removed in at least one of the four subiterations. Background pixels are shown in black, object pixels in white. Red boxes represent the border pixels corresponding to the subiterations where the center pixel can be removed.

B 3D Equation

The Boolean equation for the 3D thinning algorithm is described in terms of its 26-neighborhood, where the neighbors are numbered in clockwise order for each XY-layer

ending with the center voxel from top to bottom (see Fig. 7). In the first subiteration considering only ‘top’ boundary points, a voxel can be considered removable if it satisfies

$$\bar{n}_4 \wedge (((\bar{n}_1 \bar{n}_3 \bar{n}_5 \bar{n}_7) \wedge ((n_{21} m_4) \vee ((\bar{n}_0 \bar{n}_2 \bar{n}_6 \bar{n}_8) \wedge ((n_{21} m_1) \vee (\bar{n}_{21} \wedge (m_5 \vee m_6)))))) \vee (n_{21} \wedge (m_2 \vee m_3))) = 1,$$

where variables m_1 to m_6 are given by

$$\begin{aligned} m_1 &= t_7 \vee t_8 \vee t_9, \\ m_2 &= (\bar{n}_1 \bar{n}_2 \bar{n}_5 \wedge ((\bar{n}_0 \bar{n}_3 n_{15}) \vee (\bar{n}_7 \bar{n}_8 n_{12}))) \vee (\bar{n}_7 \bar{n}_6 \bar{n}_3 \wedge ((\bar{n}_5 \bar{n}_8 n_{10}) \vee (\bar{n}_0 \bar{n}_1 n_{13}))), \\ m_3 &= (\bar{n}_3 n_{13} \vee ((\bar{n}_7 \bar{n}_6 n_{10}) \vee (\bar{n}_0 \bar{n}_1 n_{15}))) \vee (\bar{n}_5 n_{12} \wedge ((\bar{n}_1 \bar{n}_2 n_{15}) \vee (\bar{n}_7 \bar{n}_8 n_{10}))), \\ m_4 &= (\bar{n}_0 \bar{n}_6 \wedge ((n_2 n_{11} \bar{n}_8) \vee (n_8 n_{16} \bar{n}_2))) \vee (\bar{n}_2 \bar{n}_8 \wedge ((n_0 n_9 \bar{n}_6) \vee (n_6 n_{14} \bar{n}_0))), \\ m_5 &= ((t_7 \vee t_8) \wedge ((\bar{n}_1 \bar{n}_4 \bar{n}_2 t_1) \vee (\bar{n}_9 \bar{n}_1 t_2))) \vee ((t_7 \vee t_9) \wedge ((t_5 t_6) \vee (\bar{n}_{11} \bar{n}_{16} \bar{n}_{19} \bar{n}_{25} t_3))), \\ m_6 &= (\bar{n}_{24} \bar{n}_{15} n_{18} t_4 t_6) \vee (\bar{n}_{18} \bar{n}_{10} n_{24} t_5) \vee (t_3 t_2) \vee (t_3 t_1), \end{aligned}$$

and the variables t_1 to t_9 are given by

$$\begin{aligned} t_1 &= \bar{n}_{15} \bar{n}_{16} n_{18} \bar{n}_{24} \bar{n}_{25}, \\ t_2 &= \bar{n}_{11} \bar{n}_{10} \bar{n}_{18} \bar{n}_{19} n_{24}, \\ t_3 &= \bar{n}_{13} \bar{n}_{22} n_{20}, \\ t_4 &= \bar{n}_{12} \bar{n}_{20} n_{22}, \\ t_5 &= t_4 \bar{n}_9 \bar{n}_{17}, \\ t_6 &= \bar{n}_{14} \bar{n}_{23}, \\ t_7 &= n_9 \vee n_{10} \vee n_{11} \vee n_{12} \vee n_{13} \vee n_{14} \vee n_{15} \vee n_{16} \vee n_{17} \vee n_{19} \vee n_{23} \vee n_{25}, \\ t_8 &= n_{20} \vee n_{22}, \text{ and} \\ t_9 &= n_{18} \vee n_{24}. \end{aligned}$$

The corresponding equations for all other subiterations can be obtained by substitution.

Fig. 7.
Indexing of 3D voxel neighborhood. Voxels are numbered in clockwise order for each XY-layer and from top to bottom. The center voxel in the top and bottom layer are labeled last for each layer.

References

1. Kahan S, Pavlidis T, and Baird HS, "On the recognition of printed characters of any font and size," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 9, pp. 274–288, 3. 1987. DOI: 10.1109/TPAMI.1987.4767901. [PubMed: 21869397]
2. Arica N and Yarman-Vural FT, "An overview of character recognition focused on offline handwriting," *IEEE Trans Syst Man Cybern C*, vol. 31, pp. 216–233, 5 2001. DOI: 10.1109/5326.941845.
3. Hilditch CJ, "Comparison of thinning algorithms on a parallel processor," *Image and Vision Computing*, vol. 1, pp. 115–132, 8. 1983. DOI: 10.1016/0262-8856(83)90063-X.
4. Hilditch CJ, "Linear Skeletons From Square Cupboards," in *Machine Intelligence 4*, pp. 403–420, Edinburgh University Press, 1969.
5. Rutovitz D, "Automatic chromosome analysis," *British medical bulletin*, vol. 24, pp. 260–267, 9. 1968. DOI: 10.1093/oxfordjournals.bmb.a070647. [PubMed: 5672195]
6. He L, Han CY, and Wee WG, "Object recognition and recovery by skeleton graph matching," in *Multimedia and Expo, 2006 IEEE International Conference on*, pp. 993–996, IEEE, 7 2006. DOI: 10.1109/ICME.2006.262700.
7. Bal G, Diebold J, Chambers EW, Gasparovic E, Hu R, Leonard K, Shaker M, and Wenk C, "Skeleton-Based Recognition of Shapes in Images via Longest Path Matching," in *Research in Shape Modeling*, vol. 1 of Association for Women in Mathematics Series, pp. 81–99, Cham: Springer International Publishing, 2015. DOI: 10.1007/978-3-319-16348-2_6.
8. Yilmaz A, Javed O, and Shah M, "Object tracking: A survey," *ACM Computing Surveys*, vol. 38, pp. 13–es, 12. 2006. DOI: 10.1145/1177352.1177355.

9. Wagner M, Schafer S, Strother C, and Mistretta C, "4D interventional device reconstruction from biplane fluoroscopy," *Medical Physics*, vol. 43, pp. 1324–1334, 3. 2016. DOI: 10.1118/1.4941950. [PubMed: 26936717]
10. Ji L, Yi Z, Shang L, and Pu X, "Binary Fingerprint Image Thinning Using Template-Based PCNNs," *IEEE Trans Syst Man Cybern B*, vol. 37, pp. 1407–1413, 10. 2007. DOI: 10.1109/TSMCB.2007.903369.
11. Yu Z and Bajaj C, "A Structure Tensor Approach for 3d Image Skeletonization: Applications in Protein Secondary Structure Analysis," in 2006 International Conference on Image Processing, pp. 2513–2516, IEEE, 10. 2006. DOI: 10.1109/ICIP.2006.312804.
12. Martínez-Pérez MP, Jiménez J, and Navalón JL, "A thinning algorithm based on contours," *Computer Vision, Graphics, and Image Processing*, vol. 39, pp. 186–201, 8. 1987. DOI: 10.1016/S0734-189X(87)80165-2.
13. Kimmel R, Shaked D, Kiryati N, and Bruckstein AM, "Skeletonization via Distance Maps and Level Sets," *Comput. Vision Image Understand*, vol. 62, pp. 382–391, 4. 1995. DOI: 10.1006/cviu.1995.1062.
14. Ogniewicz RL and Kübler O, "Hierarchic Voronoi skeletons," *Pattern Recognit*, vol. 28, pp. 343–359, 3. 1995. DOI: 10.1016/0031-3203(94)00105-U.
15. Niblack C, Capson D, and Gibbons P, "Generating skeletons and centerlines from the medial axis transform," in *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, vol. i, pp. 881–885 vol.1, 6 1990. DOI: 10.1109/ICPR.1990.118234.
16. Shih FY and Pu CC, "A skeletonization algorithm by maxima tracking on Euclidean distance transform," *Pattern Recognit*, vol. 28, pp. 331–341, 3. 1995. DOI: 10.1016/0031-3203(94)00104-T.
17. di Baja GS and Thiel E, "Skeletonization algorithm running on path-based distance maps," *Image and Vision Computing*, vol. 14, pp. 47–57, 2. 1996. DOI: 10.1016/0262-8856(95)01039-4.
18. Malandain G and Fernández-Vidal S, "Euclidean skeletons," *Image and vision computing*, vol. 16, no. 5, pp. 317–327, 1998.
19. Choi W-P, Lam K-M, and Siu W-C, "Extraction of the Euclidean skeleton based on a connectivity criterion," *Pattern Recognition*, vol. 36, pp. 721–729, 3. 2003. DOI: 10.1016/S0031-3203(02)00098-5.
20. Hassouna MS, Farag A, and others, "Robust centerline extraction framework using level sets," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 458–465, IEEE, 7 2005. DOI: 10.1109/CVPR.2005.306.
21. Yung Kong ART, "Digital topology: Introduction and survey," *Comput. Vis., Graph., Image Process*, vol. 48, pp. 357–393, 12. 1989. DOI: 10.1016/0734-189X(89)90147-3.
22. Bouix S, Siddiqi K, and Tannenbaum A, "Flux driven automatic centerline extraction," *Med. Image Anal*, vol. 9, pp. 209–221, 6 2005. DOI: 10.1016/j.media.2004.06.026. [PubMed: 15854842]
23. Stefanelli R and Rosenfeld A, "Some Parallel Thinning Algorithms for Digital Pictures," *J. Assoc. Comput. Mach*, vol. 18, pp. 255–264, 4. 1971. DOI: 10.1145/321637.321646.
24. Zhang TY and Suen CY, "A Fast Parallel Algorithm for Thinning Digital Patterns," *Comm. ACM*, vol. 27, pp. 236–239, 3. 1984. DOI: 10.1145/357994.358023.
25. Arcelli C, Cordella L, and Levialdi S, "Parallel thinning of binary pictures," *Electron. Lett*, vol. 11, pp. 148–149, 4. 1975. DOI: 10.1049/el:19750113.
26. Hall RW, "Connectivity-preserving parallel operators in 2d and 3d images," in *Applications in Optical Science and Engineering*, pp. 172–183, International Society for Optics and Photonics, 4. 1993. DOI: 10.1117/12.142167.
27. Gokmen M and Hall RW, "Parallel shrinking algorithms using 2-subfields approaches," *Computer Vision, Graphics, and Image Processing*, vol. 52, pp. 191–209, 11. 1990. DOI: 10.1016/0734-189X(90)90054-Y.
28. Chin RT, Wan H-K, Stover DL, and Iverson RD, "A one-pass thinning algorithm and its parallel implementation," *Computer Vision, Graphics, and Image Processing*, vol. 40, pp. 30–40, 10. 1987. DOI: 10.1016/0734-189X(87)90054-5.

29. Jang BK and Chin RT, "One-pass parallel thinning: analysis, properties, and quantitative evaluation," IEEE Trans. Pattern Anal. Mach. Intell, vol. 14, pp. 1129–1140, 11. 1992. DOI: 10.1109/34.166630.
30. Lam L and Suen CY, "An Evaluation of Parallel Thinning Algorithms for Character Recognition," IEEE Trans. Pattern Anal. Mach. Intell, vol. 17, pp. 914–919, 9. 1995. DOI: 10.1109/34.406659.
31. Zhou RW, Quek C, and Ng GS, "A novel single-pass thinning algorithm and an effective set of performance criteria," Pattern Recognit. Lett, vol. 16, pp. 1267–1275, 12. 1995. DOI: 10.1016/0167-8655(95)00078-X.
32. Deng W, Iyengar SS, and Brener NE, "A Fast Parallel Thinning Algorithm for the Binary Image Skeletonization," The International Journal of High Performance Computing Applications, vol. 14, pp. 65–81, 2. 2000. DOI: 10.1177/109434200001400105.
33. Huang L, Wan G, and Liu C, "An Improved Parallel Thinning Algorithm.," in ICDAR, vol. 3, p. 780, 11. 2003. DOI: 10.1109/ICWAPR.2016.7731637.
34. Ma CM and Sonka M, "A Fully Parallel 3d Thinning Algorithm and Its Applications," Computer Vision and Image Understanding, vol. 64, pp. 420–433, 11. 1996. DOI: 10.1006/cviu.1996.0069.
35. Palágyi K and Kuba A, "A 3d 6-subiteration thinning algorithm for extracting medial lines," Pattern Recogn. Lett, vol. 19, pp. 613–627, 5 1998. DOI: 10.1016/S0167-8655(98)00031-2.
36. Pudney C, "Distance-Ordered Homotopic Thinning: A Skeletonization Algorithm for 3d Digital Images," Comput. Vision Image Understand, vol. 72, pp. 404–413, 12. 1998. DOI: 10.1006/cviu.1998.0680.
37. Xie W, Thompson RP, and Perucchio R, "A topology-preserving parallel 3d thinning algorithm for extracting the curve skeleton," Pattern Recognition, vol. 36, pp. 1529–1544, 7 2003. DOI: 10.1016/S0031-3203(02)00348-5.
38. Ma CM, "On Topology Preservation in 3d Thinning," CVGIP: Image Understanding, vol. 59, pp. 328–339, 5 1994. DOI: 10.1006/ciun.1994.1023.
39. Lindholm E, Nickolls J, Oberman S, and Montrym J, "NVIDIA Tesla: A unified graphics and computing architecture," IEEE micro, vol. 28, 3. 2008. DOI: 10.1109/MM.2008.31.
40. Cheng J, Grossman M, and McKercher T, Professional CUDA C Programming. John Wiley & Sons, 9. 2014.
41. Luitjens J, "Faster Parallel Reductions on Kepler." <https://devblogs.nvidia.com/faster-parallel-reductions-kepler/>, 2. 2014. Accessed: 2019-04-02.
42. Minkowski H, Speiser A, and Weyl H, Gesammelte Abhandlungen. B.G. Teubner, 1911.
43. Rosenfeld A, "Three-dimensional digital topology," Information and Control, vol. 50, pp. 119–127, 8. 1981. DOI: 10.1016/S0019-9958(81)90177-7.
44. Grother P and Hanaoka K, "NIST Special Database 19 Handprinted Forms and Characters 2nd Edition," Tech. Rep. NIST SD-19, National Institute of Standards and Technology, Gaithersburg, MD, 8. 2016.
45. Fisher R, "Xv.-the correlation between relatives on the supposition of Mendelian inheritance.," Trans R Soc Edinb, vol. 52, pp. 399–433, 1. 1919. DOI: 10.1017/S0080456800012163.

Fig. 1.

Labeling of neighbor pixels in the local 3×3 neighborhood. Neighbors are labeled clockwise from n_0 to n_7 starting at the north neighbor pixel.

Fig. 2.

Illustration of image partition for blockwise processing. Black grid represents distinct blocks, where a single region R_{ij} is marked in green. Its transition region is shaded in red. The combined area R_{ij}^* is shown with a blue border.

Fig. 3.

Thinning results for four different letters. Columns 1 to 4 contain results of the CPU reference implementation, Multi-threaded CPU implementation, GPU single-block implementation and GPU multi-block implementation respectively. Columns 1, 3, and 4 show identical results. Column 2 does not show some of the smaller segments extracted by the border-sequential algorithms.

Fig. 4.

Example results for the 3D thinning algorithms. Top left image shows a surface rendering of the original volume extracted from a computed tomography scan of the cerebral vasculature. Subsequent images show 3D thinning results of the CPU reference implementation, proposed GPU algorithm, and GPU reference implementation.

Table 1

Block Sizes and Overlaps in 2D

Image Size (pixel)	Block Size (pixel)	Overlap (pixel)
32×32	32×32	0
128×128	128×128	0
512×512	512×128	64
1024×1024	1024×256	128

Table 2**2D Thinning Results**

	Width	CPUREF	CPUBP	BEGPU _S	BEGPU _M
Computation Time	32	41.19 μ s	64.82 μ s	3.53 ns	5.34 μ s
	128	0.43 ms	0.13ms	0.12 μs	8.67 μ s
	512	19.28 ms	2.04 ms	6.10 μs	46.31 μ s
	1024	0.15s	13.18 ms	N/A	0.25 ms
Number of Points	32	72	66	72	72
	128	63	64	63	63
	512	780	666	780	780
	1024	805	738	N/A	805
Number of Endpoints	32	4.00	3.00	4.00	4.00
	128	2.60	2.50	2.60	2.60
	512	6.15	4.00	6.15	6.15
	1024	6.41	3.56	N/A	6.41
Number of Segments	32	11.00	5.00	11.00	11.00
	128	6.20	6.50	6.20	6.20
	512	19.90	12.75	19.90	19.90
	1024	20.73	11.70	N/A	20.73
Number of Junctions	32	4.00	1.00	4.00	4.00
	128	2.00	2.30	2.00	2.00
	512	7.60	4.95	7.60	7.60
	1024	7.90	4.74	N/A	7.90
Average Junction Neighbors	32	3.00	3.00	3.00	3.00
	128	3.00	3.00	3.00	3.00
	512	3.03	3.05	3.03	3.03
	1024	3.02	3.00	N/A	3.02
Average Segment Length	32	7.55	14.00	7.55	7.55
	128	11.90	10.63	11.90	11.90
	512	20.28	27.01	20.28	20.28
	1024	38.39	54.59	N/A	38.39

CPUREF and CPUBP refer to the reference and border parallel CPU implemented 2D thinning algorithms. BEGPU_S and BEGPU_M are the bit-encoded GPU implementations using a single block and multi-block approach respectively.

Table 3

3D Thinning Results

	Width	CPUREF	GPUREF	BEGPU ₃₂	BEGPU ₆₄
Computation Time	128	0.11s	8.60 ms	0.27 ms	0.51 ms
	256	1.12s	88.23 ms	2.70 ms	2.90 ms
	512	14.29 s	0.94s	25.78 ms	20.32 ms
Number of Points	128	230.46	224.46	227.00	226.36
	256	567.55	548.27	547.90	409.91
	512	1285.45	1210.55	1211.91	1221.18
Number of Endpoints	128	298.64	289.00	2.60	2.60
	256	445.55	407.91	6.15	6.15
	512	681.27	560.18	N/A	6.41
Number of Segments	128	6.20	6.50	6.20	6.20
	256	19.90	12.75	19.90	19.90
	512	20.73	11.70	N/A	20.73
Number of Junctions	128	2.00	2.30	2.00	2.00
	256	7.60	4.95	7.60	7.60
	512	7.90	4.74	N/A	7.90
Average Junction Neighbors	128	3.00	3.00	3.00	3.00
	256	3.03	3.05	3.03	3.03
	512	3.02	3.00	N/A	3.02
Average Segment Length	128	11.90	10.63	11.90	11.90
	256	20.28	27.01	20.28	20.28
	512	38.39	54.59	N/A	38.39

CPUREF and GPUREF refer to the reference 3D thinning algorithm implemented on CPU and GPU respectively. BEGPU₃₂ and BEGPU₆₄ are the bit-encoded GPU implementations using a 32³ and 64³ block size respectively.