Efficient Grayscale Thinning via Parallel Processing

Christer Karlsson and John M. Weiss
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Rapid City, SD 57701-3995, USA
christer.karlsson@sdsmt.edu and john.weiss@sdsmt.edu

Abstract

Grayscale thinning is a fundamental image processing/computer vision operation, but unfortunately a rather slow process. This paper examines the theoretical foundations of grayscale thinning as an extension of binary mathematical morphology, and presents a parallel processing approach that speeds up processing time by almost two orders of magnitude on parallel hardware.

1 Introduction

The most fundamental set operation is set membership. An element $x$ is either a member of a set $S$, in which case $x \in S$ is true, or it is not, and $x \notin S$ is false. For binary images, intensity values translate directly into set membership. An intensity value of 1 indicates that a pixel is in the set, and 0 indicates that it is not in the set.

Mathematical morphology applies set theory to the field of image processing. Seminal work in mathematical morphology began in the 1960’s and 1970’s with Matheron [11] and Serra [12]. Morphological operations such as erosion, dilation, opening, closing, thinning, thickening, and skeletonizing are all well defined for binary images.

Fundamental binary morphological operations include erosion and dilation, and related operations based upon them: opening, closing, smoothing, boundary detection, etc. Another basic binary operation is the hit-or-miss transform, which can be used to derive thinning, thickening, skeletonizing, pruning, and other morphological processes.

Morphological operations on grayscale (or color) images are less straightforward. Although some morphological operations extend efficiently to grayscale images, others do not. For example, grayscale erosion and dilation operations may be implemented using neighborhood minimum and maximum operators, respectively. Morphological processes based upon erosion and dilation (opening, closing, etc.) are also readily extended to grayscale. The hit-or-miss transform, however, and those morphological operations which derive from it (thinning, thickening, skeletonizing, pruning), are not so easily applied to grayscale images.

2 Threshold superposition

The classic approach to extending binary morphological operations to grayscale is the umbra [6,12] or threshold superposition [10] technique. A grayscale image may be decomposed into a series of binary images, corresponding to binary thresholding at every possible intensity level. In threshold superposition, the grayscale image is recovered by summing all these binary images. For example, an 8-bit grayscale image is decomposed into 255 binary images, by thresholding at intensity levels 1,2,3,...,255. The grayscale image is the sum of these 255 binary images.

Figure 1: Grayscale lightning image
Any binary morphological operation may be extended to grayscale by applying the binary operator to each of the 255 binary images, and then summing the results. This method may be used to thin a grayscale image, as shown in Figures 1 and 2, but is quite slow.

For an NxN grayscale image with M intensity levels, the binary morphological operation must be applied MxNxN times. An 8-bit grayscale image therefore takes roughly 255X longer to thin than a binary image. This result is born out in theory as well as practice. Even moderate-sized grayscale images take several seconds to thin, rendering this approach impractical for real-time image processing applications. For applications such as skeletonization, multiple iterations of thinning will extend processing time by a factor corresponding to the number of iterations.

For processing color images, the best approach is to map 24-bit RGB color intensities to a color model that separates intensity from chromaticity, such as the HSI (hue, saturation, intensity) representation. The intensity component is processed using threshold superposition, and then converted back to RGB values for display.

### 3 Thinning

Thinning is an important operation in image processing and computer vision, with many practical applications. Repeated thinning may be used to find skeletons (or ridges) in an image. Skeletons of linear structures are an important representation in the human visual systems. Ridge detection (also known as linear delineation [3]) has applications in photogrammetry [2], medical imaging [3], remote sensing [15], and other areas.

Thinning may be viewed as a restricted form of erosion. Boundary pixels are eroded only when pixel removal will not disconnect or eliminate a region. In morphological thinning, 4 (or 8) rotated templates are applied sequentially to each image pixel using the hit-or-miss transform [4].

![Figure 3: Structuring elements for morphological thinning. X indicates a “don’t care” value.](image)

For an ad-hoc binary thinning schemes have been devised, based on boundary erosion rather than the hit-or-miss transform. One of the most popular is the Zhang and Suen algorithm [16]. In practice, the Zhang and Suen algorithm is significantly faster than morphological thinning, and generally produces fewer undesirable artifacts in the thinned image.

### 4 Thinning approximations

Several efficient grayscale thinning approximations have been proposed [9,13,14]. In these fast approximations, threshold superposition is replaced with a local threshold estimation approach. The basic approach is the following:

1. Estimate a local binary threshold inside a small (typically 3x3) neighborhood, and apply this threshold to produce a neighborhood binary image.
2. Perform binary thinning on the center pixel in the neighborhood. If it satisfies conditions for pixel removal, its value is replaced by another (smaller) grayscale intensity. If not, its value is left unchanged.

The basic problems in this approach are 1) how to select the local threshold, and 2) how to select the new grayscale intensity. The answer to the first problem is straightforward: the intensity of the center pixel in the neighborhood is used as the local binary threshold.
Selecting the replacement gray level intensity is more problematic. Because thinning is closely related to erosion, and grayscale erosion may be implemented as a neighborhood minimum operation, Weiss [13,14] suggests using the neighborhood minimum grayscale intensity as a replacement value. This leads to reasonable results, and speeds up processing by over two orders of magnitude.

The local threshold/minimum approach effectively thins a grayscale image, as shown in Figure 4. It is far more efficient than the threshold superposition technique, and speeds up processing by over two orders of magnitude.

The local threshold/minimum method yields results that are highly similar, but not identical, to the threshold superposition technique. A comparison of Figures 2 and 4 illustrates that thinning is greater per iteration for the approximation method, but produces more artifacts and visually poorer results. The problem is the selection of the neighborhood minimum as the replacement gray scale value.

Figure 4: Local threshold/minimum thinning approximation (2 iterations)

5 Thinning on parallel hardware

Improvements in hardware sometimes make it possible to perform exact computations rather than relying on fast approximation techniques. This study shows that parallel hardware can make threshold superposition grayscale thinning roughly as efficient as the fast local threshold/minimum approximation method.

The parallel hardware used in this study was the following:
- 2 x Intel® Xeon Processor E5-2630 (6 cores, 12 threads per processor, 2.3 GHz, 15 MB smart Cache)
- 2 x Intel® Xeon Phi Coprocessor 3120A (57 cores, 228 threads per card, 1.1 GHz, 28.5 MB L2 Cache)

In this research, parallel algorithms were benchmarked on one Phi Coprocessor card [1]. This card supports 57 cores, with 4 hardware threads per core (for a total of 228 hardware threads). This “hyperthreading” permits more efficient instruction pipelining, but is clearly not equivalent to additional processors. The speedup due to hyperthreading depends on the application.

There are several ways in which the grayscale thinning algorithm might be parallelized. On a massively parallel machine, one could assign a processor to each pixel in the image, and thin all pixels in parallel. But using the Phi Coprocessor card (with 228 hardware threads), the most obvious approach is to perform binary thresholding followed by binary thinning in parallel. This is illustrated in Figure 5.

Figure 5: Parallel threshold superposition thinning.

First, the original grayscale image \( f(x,y) \) is converted into 255 binary images, by thresholding at all possible intensity values. The parallel algorithm dynamically schedules a process for each of the 255 threshold values. Each of these parallel processes traverses every pixel in the \( i^{th} \) binary thresholded image \( b_i(x,y) \). If the binary pixel
value is not zero, binary thinning on this pixel is performed. Results are stored in the thinned binary image \( b'(x,y) \). Finally, the thinned binary images are summed into the thinned grayscale image \( t(x,y) \). Note that the summation of binary thinning must be performed sequentially, else race conditions could result.

To determine whether parallel speedup is dependent on the binary thinning algorithm, both morphological thinning and the Zhang and Suen binary thinning algorithms were implemented. This only requires a change to the bin_thin() function that is called in the parallel thinning algorithm given in Figure 6.

```
func thinning(img f, img t)
    do parallel for num_processes
        for i = 1 to 255 do
            for all pixels in f do
                if \( f[x,y] > i \) then
                    bt = bin_thin( f, x, y )
                    do atomic
                        t[x,y] += bt
```

Figure 6: Parallel threshold superposition thinning algorithm.

All code was compiled using the Intel C++ Compiler in the Parallel Studio XE [7,8]. The code was first compiled using the –O2 optimization options, enabling vectorization. After testing to validate the results, the code was recompiled with the –O3 option, which enables vectorization with more aggressive loop and memory-access optimizations, such as scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking to allow more efficient use of cache, and additional data prefetching. This optimization option may improve certain types of applications and, in our case, gave an extra performance boost.

The algorithms were run sequentially (on a single core and single thread of the Xeon Phi coprocessor) and in parallel. Parallel processing was performed on 55 cores of one Xeon Phi board, with 2 cores reserved for the operating system. The impact of hyperthreading was investigated by running 1, 2, and 4 threads per core.

The results of this parallelization are shown in the following tables, using the lightning image (860710 pixels) of Figure 1. Runtimes are given in milliseconds, speedup is the ratio of sequential to parallel processing time, and efficiency is the % maximum theoretical speedup, based on the number of hardware threads.

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single thread (sequential)</td>
<td>8590</td>
<td>1X</td>
<td>100%</td>
</tr>
<tr>
<td>55 threads (1 per core)</td>
<td>224</td>
<td>38X</td>
<td>70%</td>
</tr>
<tr>
<td>110 threads (2 per core)</td>
<td>144</td>
<td>60X</td>
<td>54%</td>
</tr>
<tr>
<td>220 threads (4 per core)</td>
<td>122</td>
<td>70X</td>
<td>32%</td>
</tr>
</tbody>
</table>

Table 1: Parallel morphological thinning algorithm (image data)

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single thread (sequential)</td>
<td>3068</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>55 threads (1 per core)</td>
<td>83</td>
<td>37</td>
<td>67%</td>
</tr>
<tr>
<td>110 threads (2 per core)</td>
<td>51</td>
<td>60</td>
<td>55%</td>
</tr>
<tr>
<td>220 threads (4 per core)</td>
<td>44</td>
<td>70</td>
<td>32%</td>
</tr>
</tbody>
</table>

Table 2: Parallel Zhang and Suen thinning algorithm (image data)
In addition to benchmarking parallel code on real images, we also timed performance on random pixel data. These results are given in the following tables:

<table>
<thead>
<tr>
<th>256x256</th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single thread (sequential)</td>
<td>563</td>
<td>1X</td>
<td>100%</td>
</tr>
<tr>
<td>55 threads (1 per core)</td>
<td>15.4</td>
<td>37X</td>
<td>66%</td>
</tr>
<tr>
<td>110 threads (2 per core)</td>
<td>9.77</td>
<td>58X</td>
<td>52%</td>
</tr>
<tr>
<td>220 threads (4 per core)</td>
<td>8.84</td>
<td>64X</td>
<td>29%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>512x512</th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single thread (sequential)</td>
<td>3291</td>
<td>1X</td>
<td>100%</td>
</tr>
<tr>
<td>55 threads (1 per core)</td>
<td>90.9</td>
<td>36X</td>
<td>66%</td>
</tr>
<tr>
<td>110 threads (2 per core)</td>
<td>56.5</td>
<td>58X</td>
<td>53%</td>
</tr>
<tr>
<td>220 threads (4 per core)</td>
<td>48.8</td>
<td>67X</td>
<td>31%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1024x1024</th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single thread (sequential)</td>
<td>13380</td>
<td>1X</td>
<td>100%</td>
</tr>
<tr>
<td>55 threads (1 per core)</td>
<td>367</td>
<td>36X</td>
<td>66%</td>
</tr>
<tr>
<td>110 threads (2 per core)</td>
<td>238</td>
<td>56X</td>
<td>51%</td>
</tr>
<tr>
<td>220 threads (4 per core)</td>
<td>205</td>
<td>65X</td>
<td>30%</td>
</tr>
</tbody>
</table>

Table 3: Parallel morphological thinning algorithm (random data)

Table 4: Parallel Zhang and Suen thinning algorithm (random data).

Benchmarks for random data are consistent with those of actual image data. As expected, runtimes increase linearly with the total number of pixels processed. The parallel speedup increases from 36X to 73X as the number of threads increase, but efficiency drops from 66% to 30%.

There are a number of reasons why the speedup is limited to about 70X, despite the fact that the number of hardware threads is 220. First, there are only 55 cores. The effectiveness of hyperthreading (2 or 4 threads per core) may depend strongly on the application. Second, there is a need for mutual exclusion when writing to the resulting thinned grayscale image. The thinned binary images must be summed sequentially, otherwise race conditions could cause incorrect intensities to be stored.
6 Conclusions

The extension of binary morphological operations, notably binary thinning, to grayscale images is considered in this paper. Efficient grayscale thinning methods, of great importance in image processing and computer vision, have not been well developed in the past.

A parallel approach to efficient grayscale thinning is presented in this paper. This approach thins a grayscale image efficiently, with a 70-fold speedup over sequential processing. This speedup is almost as great as that afforded by approximation techniques, and yields exact results. The performance increase is dependent upon the number of processors and hardware threads, with efficiencies ranging from 30% to 70% on our hardware. Hyperthreading improves performance, but not as much as additional cores.

Parallel hardware is becoming more affordable every year. The results presented in this paper suggest that parallel algorithms will see extensive application to image processing problems in the near future.

7 References