EC527 Final: Watershed Algorithm

Rahul Arasikere
Computer Science
Boston University
Boston, United States
rahulav@bu.edu

Xavier Ruiz
Computer Science
Boston University
Boston, United States
xruiz@bu.edu

***Abstract*—This paper implements and analyzes the watershed algorithm used for image segmentation. The paper further goes on to evaluate the various speed ups gained from the various methods. It also implements hardware enabled version of the serial reference code to discuss the benefits of implementing a CUDA enabled version over using OpenMPI and other various CPU methods.**

***Index Terms*—watershed, CUDA, parallelization, OpenMPI**

I. Introduction

The watershed algorithm is a non-parametric algorithm that is run on images to produce a segmented image. This has many applications from AI to the medical field.

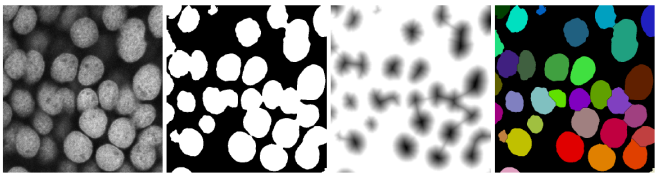


Fig. 1. From left to right, image of cell nuclei, binary mask after filtering, distance transform of mask, and watershed image after applying labels.

The star in the Watershed Algorithm is the waterfall transform, but before that the image must be thresholded and distance transformed.

A. Thresholding

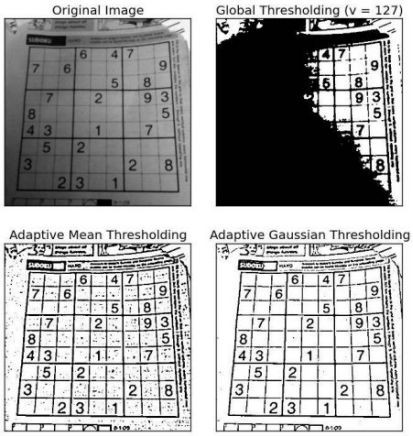
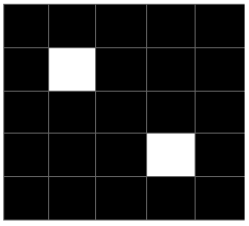


Fig. 2. There are various thresholding techniques that can be applied to an image. Generally, Adaptive Gaussian Thresholding performs best.

Thresholding of the image is necessary in order to apply the distance transform. There are various methods to threshold an image, but Adaptive Gaussian Thresholding works best as it takes a Gaussian-weighted sum of a pixel's neighbors minus a constant C. This performs better due to the fact that an image can have different lighting in different areas, and it also ignores noise.

B. Distance Transform

The distance transform (DT) simply sets up "basins" for our waterfall transform to do its work. For each pixel, it sets its value to its distance from the closest white pixel.



Input

1.41	1	1.41	2.24	3.16
1	0	1	2	2.24
1.41	1	1.41	1	1.41
2.24	2	1	0	1
3.16	2.24	1.41	1	1.41

Distance Transform

Fig. 3. The distance transform applied to a binary image.

However, by applying the DT this way, large, flat basins will form. In order for the waterfall transformation to function correctly, the basins cannot be flat and must be a single pixel in size.

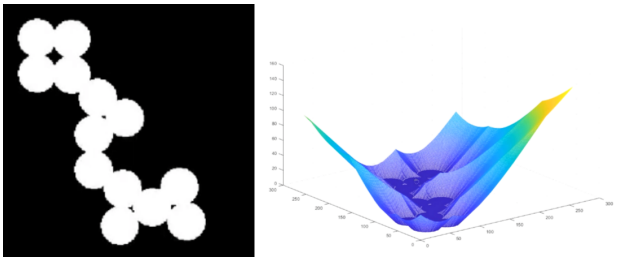


Fig. 4. Binary image on left and DT image on right.

This can be fixed easily by first inverting the image, applying the DT, and then negating the DT.

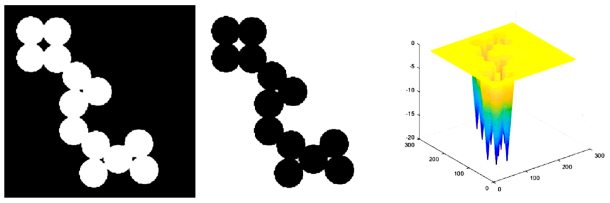


Fig. 5. From left to right: the original image, inverted image, and the negated DT image.

By utilizing this method, we still get the basins, but they are now of a single point, perfect for the next step.

C. Waterfall Transform

The waterfall transform starts by arbitrarily numbering the basins and starts and arbitrarily chooses which basin to start at. It will then "flood" this basin until it flows into another basin. To "flood" just means to outwardly mark (from the center of the basin) pixels as flooded.

Then, it will flood the basin that was flooded into. If this second basin floods back into the original one, then this is called symmetrical flooding. This means that these segments of the image were very similar in average value, and therefore can be grouped into a larger mega-basin.

If the second basin does not flow back into the original, then this basin is non-symmetric and this boundary should be preserved and marked as a Watershed Line (WL). The process can now start over from this basin that did not flow back into the original.

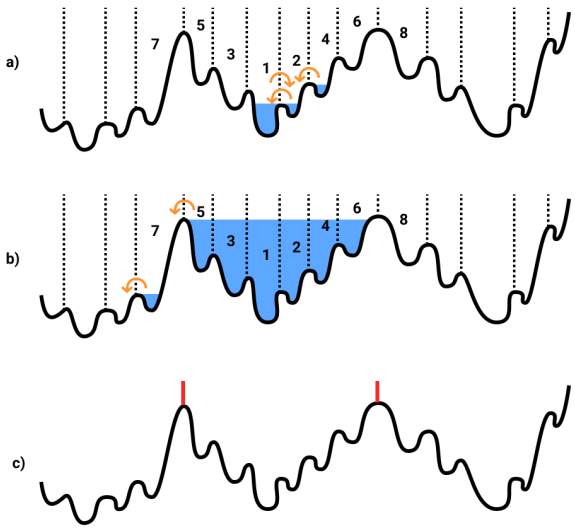


Fig. 6. The above images show the WT being applied to a 2D image. (a) Basins 1 and 2 flow into each other, they are symmetric. In fact, all basins 1-6 are symmetric with each other and form one mega-basin. (b) The mega-basin (1-6) flows into basin 7, but when 7 is filled it does not flow back. (c) The edge between the mega-basin and basin 7 is preserved as a WL as well as the edge between the mega-basin and basin 8.

With that, the watershed transform is complete.

II. Parallelizing the Algorithm

A simple serial algorithm has been implemented with 4 distinct kernels based on the description of the algorithm. The CUDA version will develop further on these 4 kernels. At a simple glance, each kernel has dependencies on the neighboring cells.

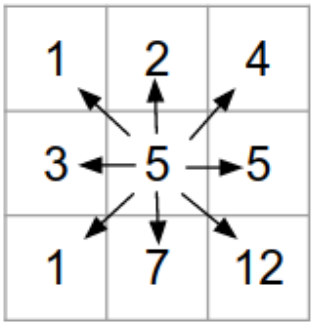


Fig. 7. Dependencies of a single pixel value.

An efficient way of parallelizing this would be to have the CPU coordinate the synchronization between the individual GPU threads. Following single instruction, multiple data architecture (SIMD), we split the threads so that each thread only operates on a single pixel cell.

III. Implementation

A. Serial Algorithm

The data is loaded in as a flat array of single integer values. We start by passing a results pointer and the image data pointer to a kernel function, which reassigns the results pointer to be whatever it did to the image. A typical kernel would look something like this:

```
void kernel(int* in, int** out, int width,
            int height)
{
    int* results = (int*)calloc(width * height,
                   sizeof(int*));
    for(int i=0; i<height; i++) {
        for(int j=0; j<width; j++){
            // do something here
        }
    }
    out = results;
}
```

Fig. 8. The kernel creates a new image and reassigns the NULL out pointer it was passed to its results image.

These kernels are the steps such as the adaptive Gaussian threshold, distance transform, and the waterfall transform.

The last kernel's results are saved to an image file.

B. OpenMP Enabled Algorithm

In the OpenMP version, we added `#pragma omp parallel for` above each of the outer for loops for the kernels.

C. CUDA Enabled Algorithm

Based on the previous serial implementation, the algorithm is split into 4 different kernels and an atomic variable is utilized to check if the algorithm has reached a stable state.

The atomic variable is incremented only if a pixel value is updated, and the CPU loop to call each individual kernel will look like this:

Algorithm 1: CPU scheduling of kernels.

$old \leftarrow -1$;
$new \leftarrow -2$;
while $old \neq new$ **do**
    $old \leftarrow new$;
    Call the kernel passing new as an argument;
    Synchronize between threads;
end
Done with the kernel;
Result: Stable values reached.

The image data is stored in the texture memory location on the GPU in CUDA enabled devices. The texture memory allows for fast read access between the individual threads in the GPU.

The block size is 16 x 16 threads (256 threads in each block) and the grid size is selected based on the width and height of the image in pixels.

D. Optimizing the Code

We used a few techniques in optimizing the serial reference code as well as implementing a CUDA enabled version of the algorithm.

We used the compiler optimization flags, mainly $-\mathbf{O1}, -\mathbf{O2}$ and $-\mathbf{O3}$ flags. We also did loop unrolling, when it came to checking the neighbouring cells but the optimization flags for the compiler also to a certain extend did the same thing.

We bench-marked the code against images of various sizes, 128x128, 256x256, 512x512, 1024x1024 and 2048x2048 pixel sizes.

IV. Results

With CPU methods for speed up we achieved an 1.5x speedup over the serial reference code. The gcc compiler flag $-\mathbf{O3}$ enables various flags along with loop unrolling and reducing branches. As this algorithm is dependent on the neighboring cells, it is effective in providing a significant speedup.

The best way to perform loop unrolling is to check two or more neighboring pixels per iteration, and can be done by hard coding the relative pixel indexes into an array.

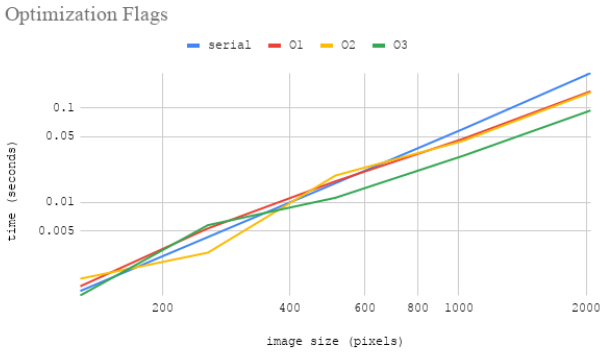


Fig. 9. The above graph is the time required by the algorithm with various image sizes when various optimization flags were enabled.

However we saw much faster gains with the CUDA enabled version of the algorithm providing an average speedup 9.5x with a best speedup of 11.5x with an even greater speedup seen by the OpenMPI version with a whopping 1585x speedup on average.

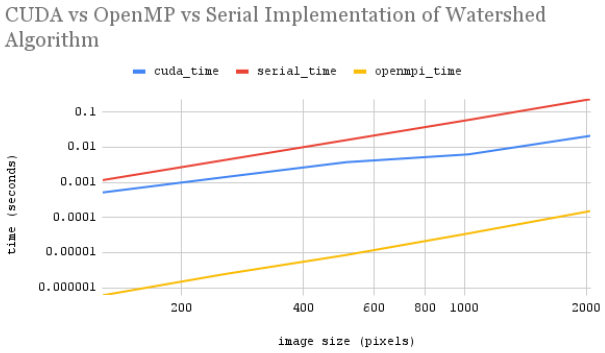


Fig. 10. The above graph is the time required by the algorithm with various image sizes.

In terms of accuracy, the serial version of the code is more accurate in detecting various surfaces in the images, this is due to the fact the GPU operates on blocks of pixels and at the boundary of this blocks, the values tend to change slower as they are effected by the two different blocks and data races arising from the data. However the performance gains makes the extra iterations of each kernel negligible.

The results from the algorithm can be seen in figure 11 below. The input image was preprocessed and thresholded via simple python script that performs a Gaussian blur on the input image.

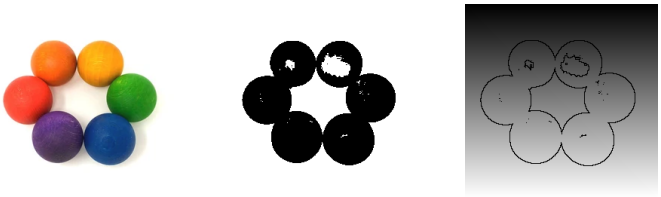


Fig. 11. From left to right: original image, threshold-ed image and segmented image.

REFERENCES

[1] Vitor B, Körbes A. Fast image segmentation by watershed transform on graphical hardware. In: Proceedings of the 17th International Conference on Systems, Signals and Image Processing, pp. 376-379, Rio de Janeiro, Brazil.

[2] Körbes A et al. 2009. A proposal for a parallel watershed transform algorithm for real-time segmentation. In: V Workshop de Visão Computacional, São Paulo, Brazil.

[3] Distance Transform Watershed, ImageJ. [Online]. Available: https://imagej.net/Distance_Transform_Watershed. [Accessed: 07-May-2021].

[4] Image Thresholding, OpenCV. [Online]. Available: https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html. [Accessed: 07-May-2021].