Julia for Accelerated Cell Segmentation

Rebecca Carlson, Katharina Hoebel, and Olivia Waring[1]

6.338 Final Project Report

Abstract

For our 6.338 final project, we implemented a marker-based Watershed algorithm for cell segmentation in Julia and compared its performance to an existing Python algorithm using a dataset of 900 images of over 200,000 cultured cells. We improved performance using multi-threaded and distributed parallelization schemes and implemented the first Julia U-Net, which we were able to successfully train on a subset of our training data as a proof-of-concept.

1. Introduction and Background

Traditionally, images of cultured cells for biological research were both acquired and reviewed manually; however, in the past several decades there has been a rapid increase in the throughput of biological image acquisition and therefore a resulting interest in developing methods for automated image analysis. For example, a recent paper by Feldman et al. [1] describes a novel method for systematically knocking out thousands of genes in millions of cells. A genome-wide screen using this method would require approximately 100 million cells. Microscopy is used to determine both the identity of the gene that was knocked out in each cell and to read out proteins of interest in the cell that are stained with fluorescent antibodies. Datasets such as these represent rich sources of new biological information; however, it is not possible to manually analyze millions of cells. In particular, all steps of the analysis depend on the quality of the cell segmentation that is performed. Therefore, selection of an appropriate segmentation algorithm is critical. In addition, cell segmentation can be a very time-consuming process for such large datasets, but it is difficult to parallelize this process with high-level languages commonly used by biologists like Python or R.

In images of cultured cells, nuclei can generally be easily identified via thresholding as they are much brighter than background and are usually not touching (Figure 1). The nuclei can then be used as markers from which a region is grown and cells are segmented using a method termed marker-controlled watershed [2]. Figure 2 shows the results of this type of cell segmentation for the same image shown in Figure 1. While the marker-controlled watershed algorithm achieves high accuracy for cultured cells that are not overlapping (the original paper achieved 98.8% accuracy), it performs less well for images from tissue samples (Figure 8). Not all of the nuclei are correctly identified due to larger differences in staining efficiency in different sections of the tissue and there is more oversegmentation. In addition, even with good performance on cultured cells, the algorithm is currently implemented in Python (as are most other available open-source pipelines for biological image analysis) and therefore is not easy to parallelize, leading to long runtimes on large datasets. For instance, segmentation alone would require about 1 day for a dataset of 100 million cells required for a genome-wide screen. In addition, this could require more time if the cells are low-density.

[1]Contribution notes: Rebecca Carlson implemented the Julia serial watershed algorithm, Olivia Waring experimented with parallelization and conducted performance analysis, and Katharina Hoebel encoded the U-Net and performed training and testing.

In this project, we implemented parallelized marker-based Watershed in Julia, reducing the segmentation time (relative to the serial version) as much as ten-fold using multi-threaded and distributed parallelization. We also implemented the first U-Net architecture in Julia, which we showed could successfully learn to overfit a few training examples, and which we hope can more robustly segment various hard-to-analyze cells in the future. As Julia matures, we believe that U-Net based segmentation architectures could be used by biologists to quickly and easily train models to their own data, as U-Nets often require few training examples.

2. Watershed Implementation

2.1. The Marker-Based Watershed Algorithm

The naive Watershed algorithm treats an image as a topological surface, in which dark pixels represent low points in the terrain and lighter pixels represent higher points. By progressively "flooding" the image, the algorithm identifies "catchment basins" and "watershed ridge lines," which correspond to discrete objects (for our purposes, cells). The marker-based watershed algorithm avoids over-segmentation by stipulating in advance which pixels constitute "foreground" and "background" regions of the image. A marker-based watershed algorithm implemented with Meyer's priority flooding [9] might proceed as follows:

- Preprocessing:
    1. Convert the image to grayscale.
    2. Compute the gradient magnitude of the image (gradients should be highest at the borders of discrete objects) and use this as a segmentation function.
    3. Designate the foreground and background objects, by using a combination of techniques including an erosion followed by a dilation ("opening") or erosion followed by a morphological reconstruction ("opening by reconstruction").

- Meyer's Priority Flooding
    1. Using the results of Step 3 in the preprocessing phase, choose the "markers" (pixels from which flooding will begin - in our case, these will be pre-segmented nuclei) and assign a unique label to each. Each marker corresponds to a single "object" in the image to be segmented.
    2. For each distinct marker, arrange the surrounding pixels into a priority queue, with the pixels corresponding to a lower gradient magnitude appearing earlier in the queue.
    3. Take the first pixel in the priority queue; if its already-labeled neighbors share the same label, apply that label to the pixel in question. Add unlabeled neighbors to the priority queue, according to their respective gradient magnitudes.
    4. Continue to execute Step 3 until the priority queue is empty; any unlabeled pixels correspond to watershed boundaries.

2.2. Serial Julia Implementation

The serial Julia implementation of the Watershed algorithm ran far more slowly than the corresponding Python version: for a suite of 1000 grayscale test images, the Python code completed in 164.3 seconds and the Julia code terminated in 606.2 seconds, more than a three-fold slow-down (see Figure 5).[2] This unfortunate finding heightens the necessity of exploiting Julia's capacity for parallelization to expedite the native Watershed algorithm, as will be discussed in the following section.

[2]This is in contrast to the data we showed during our project presentation on Wednesday, December 5th. That analysis was conducted using different versions of the Python/Julia code, and upon more rigorous inspection, was not entirely accurate. Our apologies for the misinformation.

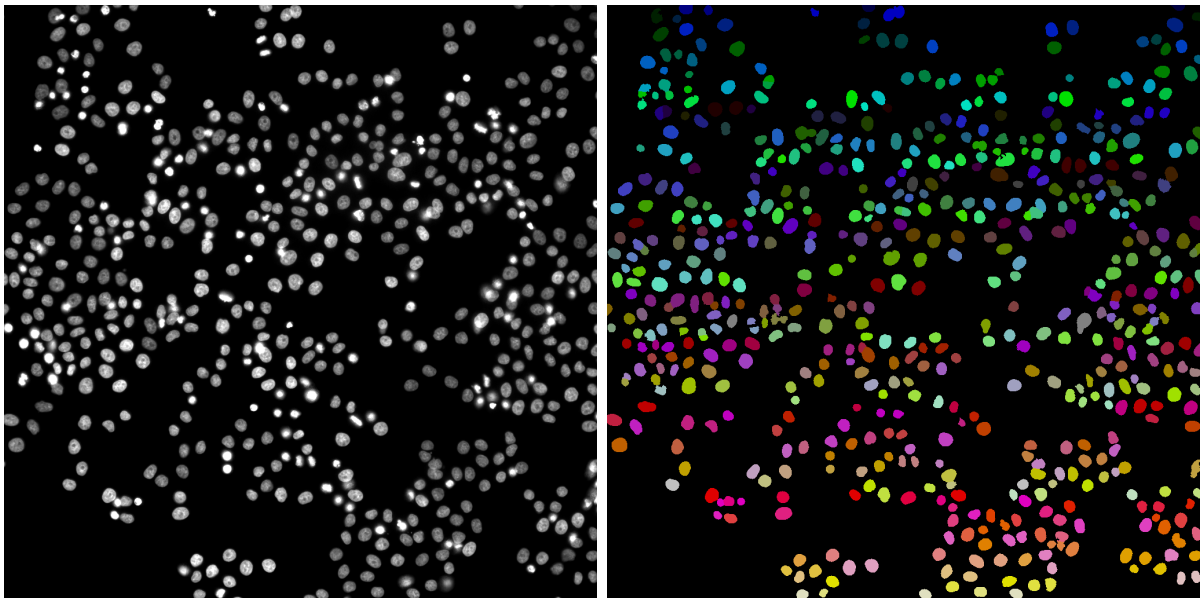

Figure 1: Sample image of cell nuclei and nucleus identification via thresholding.

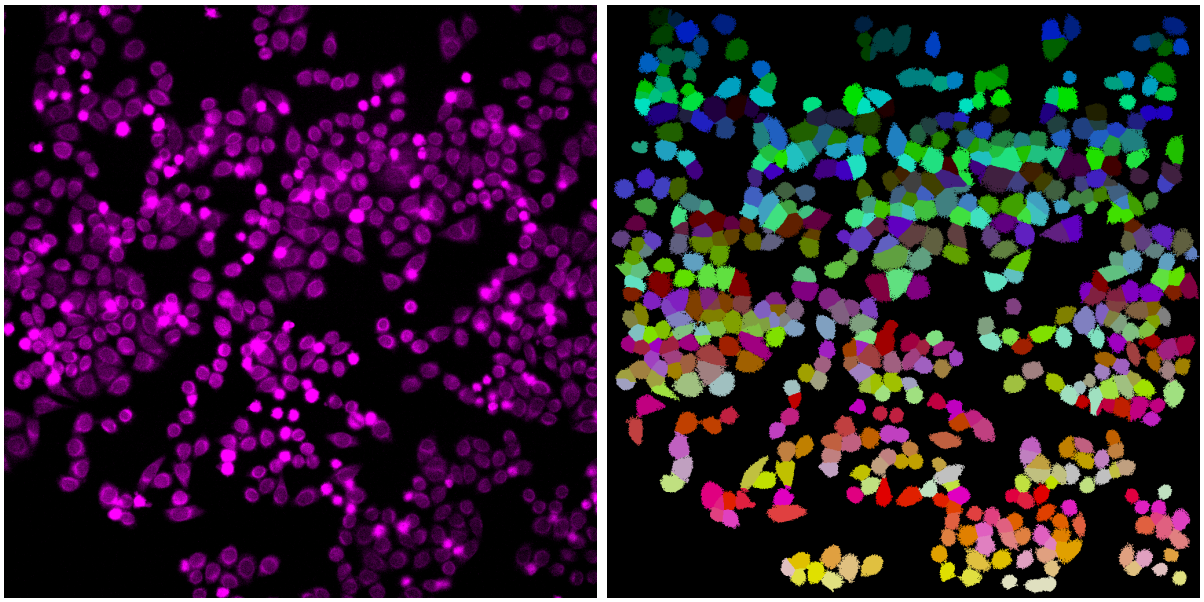

Figure 2: Sample image of cell (left) and cells segmented via marker-controlled watershed (right).

3. Parallelization

Julia allows for three possible methods of parallelization:

1. the Tasks (or Coroutines) module, also known as "Green Threading."
2. Multi-Threading
3. Distributed Computing

3.1. Coroutines

The Coroutines approach (alternatively known as Green Threading) relies on a VM dynamic scheduler to allocate resources to particular tasks at different times and exploits the Tasks module to switch among multiple computations. A Green Thread is, in essence, a thread scheduled by a virtual machine, which allows for thread-like behavior without any specific OS dependencies. We ultimately decided not to implement this parallelization approach, since it should be comparable to true multi-threading.

3.2. Multi-Threading

Multi-threading, as defined by the Julia Documentation [3], occurs when program "execution is forked and an anonymous function is run across all threads." These threads occupy the same address space in memory, thus facilitating rapid communication across the threads, and they are "ultimately be joined in Julia's main thread to allow serial execution to continue." The number of threads available to the Julia processor at any given time is set by toggling the JULIA_NUM_THREADS environment variable in the console. When dealing with multi-threading, in Julia or any other language, it is crucial to guard against race conditions (which occurs when multiple threads attempt to modify the same memory at the same time) and other unintended side effects. For this reason, multi-threading is well-suited to our purposes, since each thread acts on an independent object in memory (in this case, an image). It is also useful to note that multi-threading in Julia 1.0 is still in its experimental stages.

Because multi-threading is among the more straightforward parallelization methods in Julia, we experimented extensively with performance across a wide range of parameters. Using the @threads and @time macros, we forked execution of a pared-down watershed algorithm[3] on **n** images over **p** threads and recorded the runtime, memory allocation, and percent of time in the Garbage Collector for each. Ten experiments were conducted for each combination of **n** and **p** values, so as to minimize the impact of random effects.

As can be observed in Figure 3, increasing the number of threads results in a commensurate decrease in the processing time. Similar phenomena are observed for both the n=100 and n=1000 cases, as is to be expected. Memory allocation was roughly constant across all runs, in accordance with prediction. Presumably this chart would plateau at a certain number of threads, as the computing capacity of the particular CPU in question were reached. Somewhat counter-intuitively, the p=2 runs spent the most amount of time in Garbage Collection mode. These statistics are recorded in full in the corresponding Julia notebook.

[3]We encoded a minimal version of the watershed algorithm for proof-of-concept. We did not wish to use the entire marker-based watershed algorithm for performance testing, since it would have been prohibitively time-consuming on large data sets; however, our pared-down version provides enough functionality to still shed light on the merits of parallelization. In a similar vein, rather than loading unique full-sized images, we instead populated an array with copies of the same image, which should not have altered the runtime ratios. See the Julia notebook for details.

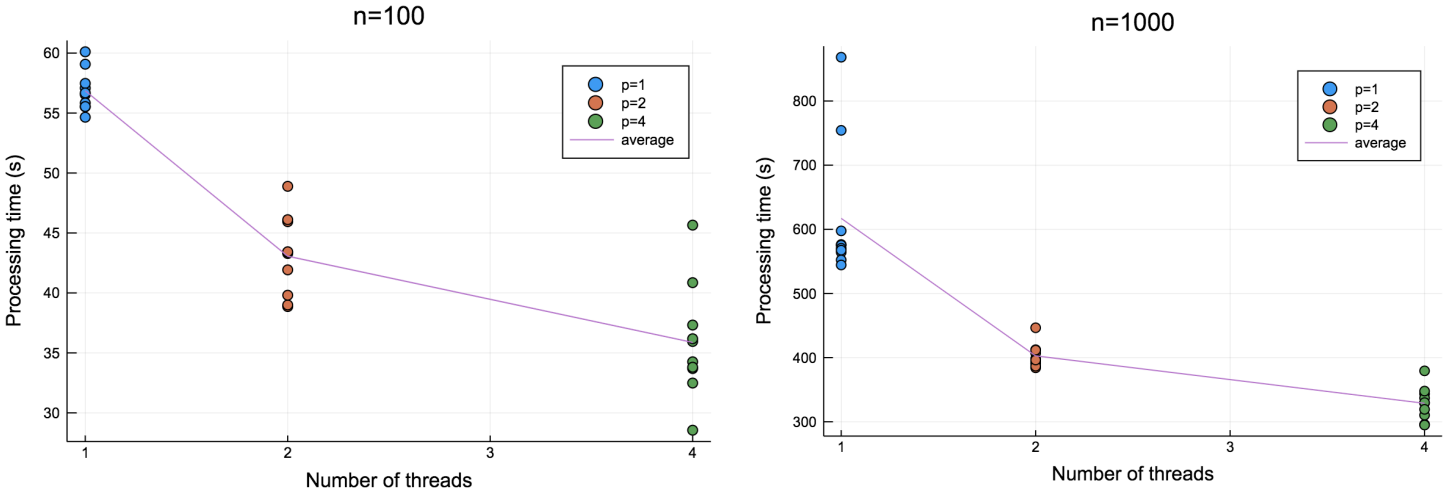

Figure 3: Two experiments, one using 100 images and one using 1000, that show the performance gains made by increasing the number of threads. (For the purpose of performance analysis, we repeatedly sampled the same images, which accounts for the n=1000 run for a data set of only 900 total images.)

3.3. Distributed Computing

Multi-core (or distributed) processing in Julia divides operations across multiple CPUs, thus leveraging the power of multiple computing engines at once. Distributed computing is limited both by the processing speeds of the individual CPUs and speed of communication among the various nodes. Thus, operating on a single node (computer) obviously gives faster memory access, but distributing operations across multiple computers increases computing power. In our experiments with a distributed Watershed algorithm, we used a 3.1 GHz Intel 4-Core i5 MacBook Pro and launched the Julia terminal with all 4 cores activated (\$ julia −p 4). We experimented with 2, 4, and 8 parallel processes operating over 4 cores: processing time was much faster relative to multi-threading, but did not vary significantly with the number of concurrent processes, as shown in Figure 4.

We limited our experiments to a quad-core processor, but other studies have been conducted that expand the number of cores to as many as 16. As noted in the PhD thesis of Ekanathan Natarajan, which includes an extensive treatment of a distributed Watershed algorithm coded in Julia [7], increasing the number of cores can yield pronounced performance gains. While a thorough deconstruction of the Wasp algorithm for dynamic scheduling [8] lies somewhat beyond the scope of this investigation (given limits on our computational resources and time), Natarajan's findings suggest a near-inverse linear correlation between the log2 of the number of processor cores and the log2 of the runtime, as shown in Figure 4.

3.4. Comparison with Python implementation

Ultimately, perhaps the most fruitful comparison to be drawn is with the original Python implementation of the Watershed algorithm.[4] As can be seen in Figure 5, the serial Julia code is markedly expedited by both the types of parallelization we tried, with the most significant speed-up observed for the quad-core distributed algorithm with 8 concurrent processes, followed by the quadruple threaded algorithm. This is unsurprising, since multi-threading tends to involve substantial processing overhead. Perhaps the most startling result to be gleaned from this graph is that - as described above - the serial Julia implementation is over three times

[4]For performance comparison purposes, we used a modified version of the Python Watershed algorithm that more closely corresponds to the Julia implementation we used for testing.

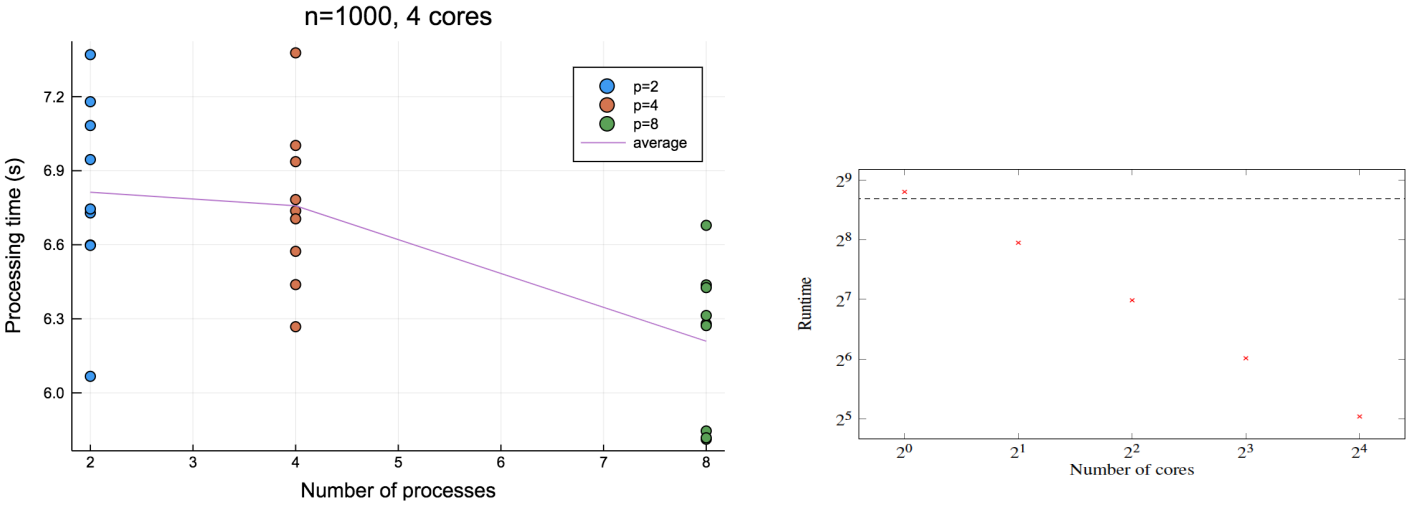

Figure 4: A) Varying the number of concurrent processes, distributed over computing 4 cores. B) A figure drawn from [7], demonstrating the relationship between the runtime of a distributed implementation of the Watershed algorithm on a 2D random matrix of size 16384 x 16384 (using Wasp scheduling) and number of processor cores. The dotted line represents the performance of the serial Watershed algorithm on a comparable random matrix.

slower than the serial Python implementation. There are a variety of possible explanations for this, ranging from compiler optimization issues (such as JIT lags) to overhead associated with start-up operations; but the most salient (and reassuring!) take-away is that parallelization can significantly expedite the process of automated cell segmentation in Julia.

4. U-Net Implementation

4.1. Data Preprocessing for Cell Segmentation Using an U-Net

The original 900 1024x1024 images were cropped to 14,400 256x256 images and scaled to have 0 mean and unit variance. The segmented masks that originally encoded 0's for background and unique integer values for each cell were then altered to have 0's for the background and 1's in the foreground. As the dataset is large, only the first ten data instances have been made available on the GitHub as an HDF file.

4.2. U-Net Architecture

Figure 6 shows the deep convolutional network architecture implemented in Knet for this project. U-Nets were first described in 2015 by Ronneberger et al. for the task of segmenting cell bodies on electron microscopy images [4]. U-Nets have become one of the most widely-employed architectures for segmentation tasks in biomedical image analysis mostly because they are easy to train and require fewer training samples to provide precise segmentations.

An advantage of convolutional networks is that they can work with images of different sizes, so even though we chose to train on images of size 256x256 with GPU with sufficient memory we could train the same network on images of size 1024x1024.

Convolutional operations are followed by batch normalization and activation using ReLU. In the encoding arm (layers 1 to 3) each layer the input image is passed through two convolutions, the number of channels is increased, followed by a max pooling operation which reduces the image size by a factor of 2. The bottleneck (layer 4) consists of two convolutions without max pooling. Here the number of channels reaches the defined maximum of 512. In the decoding arm of the network (layers 5 to 7), the images are upsampling by a factor of two (bilinear upsampling), followed by concatenation with the output of the encoder layer on the same

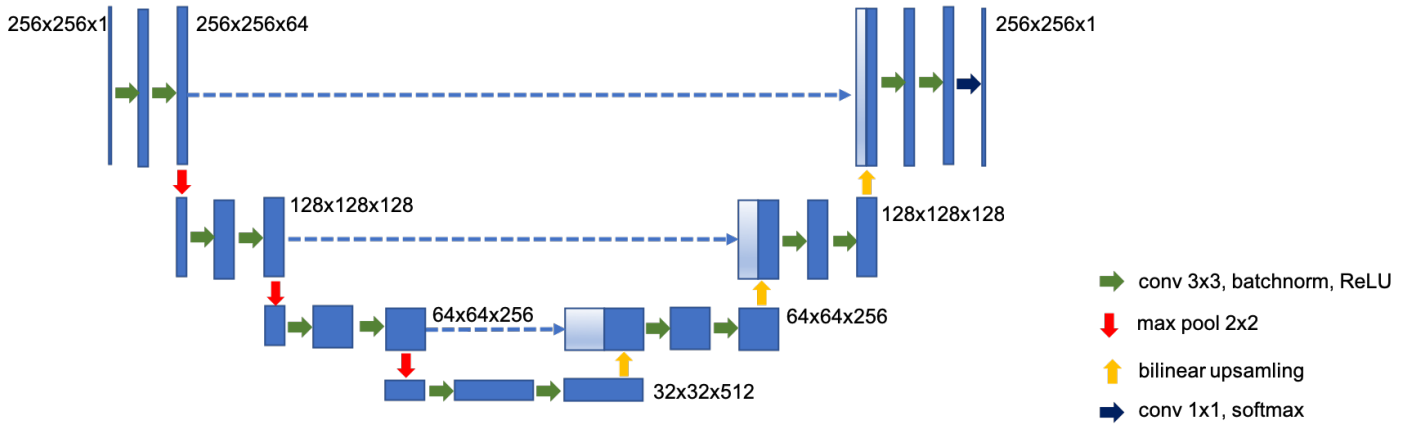

Figure 6: U-Net architecture implemented in Knet for this project.

level (doubling the number of channels), allowing for better propagation of context information, and two convolutions. The output layer (layer 7) has one more convolution that reduces the number of channels from 64 to 1 without batch normalization and softmax activation. The threshold for the final binarized mask is set to 0.5.

We use binary cross-entropy as loss function and dice coefficient as accuracy metric. The size of a mini-batch is set to 2 and we chose ADAM with an initial learning rate of $1.0 * 10^{-3}$ as the optimizer.

4.3. Results U-Net Segmentation

We achieved a reasonably good result on this first attempt of using our U-Net implementation in Knet. Training the aforementioned U-Net implementation on a small subset of 100 images of the full data set (14,400 images) for 17 epochs results in the segmentation depicted on the right panel in figure 7. Dice score on this particular segmentation result was 0.92.Unfortunately, we were not able to save the model (or rather the batch norm moments) and therefore cannot provide a trained model with our submission.

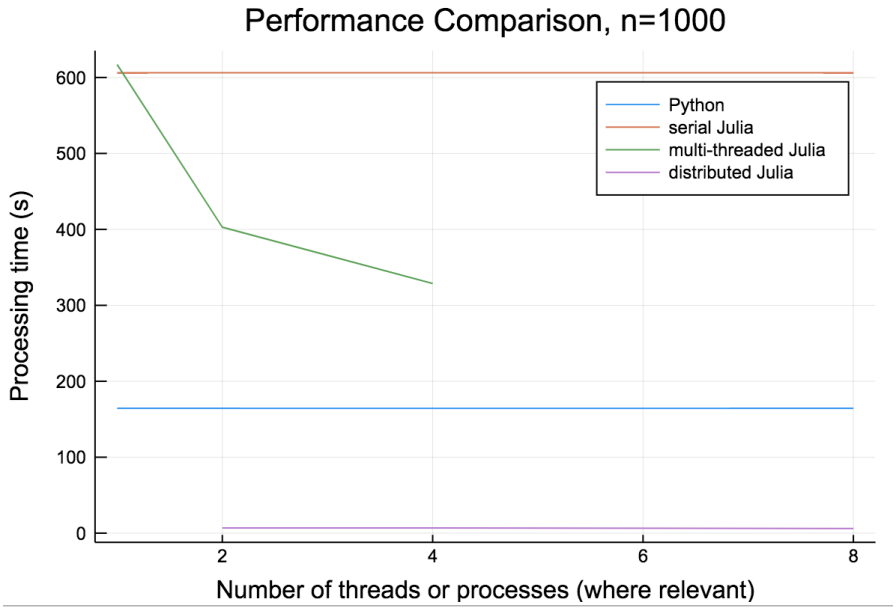

Figure 5: A comparison of the native Python Watershed implementation, a serial Julia version, and two parallelized Julia versions.

5. Conclusions and Future Directions

In this project we explored the usability of Julia for automatic cell segmentation. To do this, we used a training dataset with over 200,000 cultured HeLa cells and 900 fields of view from 36 different wells of a 96-well plate, which allowed us to evaluate the algorithm's performance in the presence of some data heterogeneity, since no illumination correction was performed.

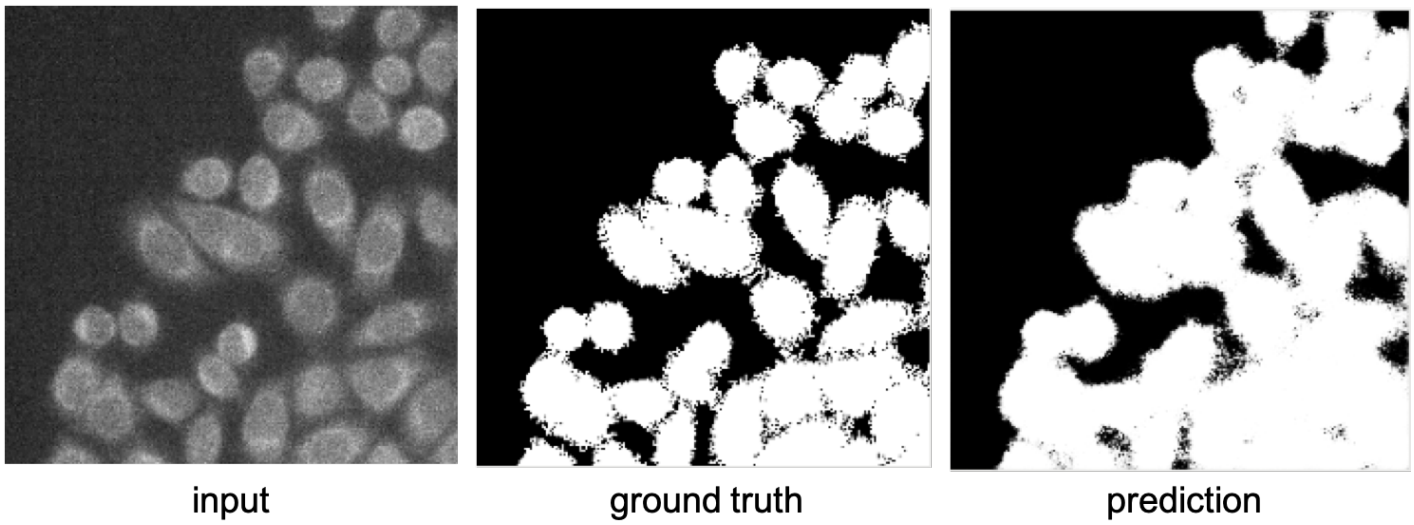

Figure 7: Cell segmentation using our U-Net implementation. Input image (left), ground truth (middle), and binarized segmentation output of an independent test data set image.

We first encoded a marker-based Watershed segmentation algorithm in Julia and validated its performance against a Python implementation. A version of the algorithm was parallelized using multi-threading and distributed computing, with commensurate improvements in efficiency, thus demonstrating that Julia can be leveraged as a powerful tool for high-throughput image processing.

Finally, we implemented a U-Net model in Julia for fully automatic cell segmentation and were able to demonstrate that this model, when trained on a small subset (100 images) of the training data, can achieve reasonable results. The training images have not been provided on the GitHub, since the size of the entire dataset was prohibitively large.

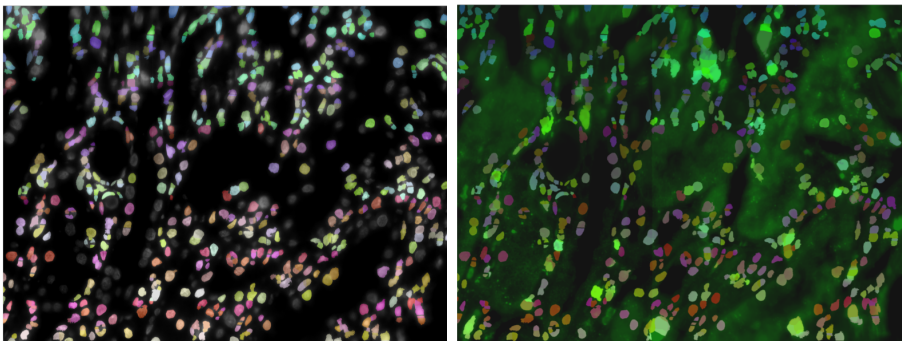

Figure 8: Sample images of nuclei (left) and cells (right) from a kidney biopsy identified via marker-controlled watershed segmentation.

Possible future extensions of our project could include a more thorough analysis of parallelization in Julia, including an in-depth exploration of the Tasks module and pmap functionality, in the hopes of ascertaining which Julia parallelization strategy is the most powerful and generalizable. Time permitting, we would also explore the impacts of multi-threading and distributed computing on a full-fledged version of the marker-based Watershed algorithm.

We would also endeavor to fully train our U-Net model and compare the resultant segmentation output to Watershed segmentation. It would be instructive to assess the robustness of Deep Learning (DL) segmentation relative to the Watershed approach. We would also like to explore the possibility of transfer learning: that is, can we use the trained U-Net to perform less straightforward segmentation tasks, such as the following:

- cells that are organized in dense structures

- cells that have unusual shapes (e.g. multi-nucleated, mitotic, etc.)
- cells in 3-D tissue sections

Presumably, the DL segmentation approach is also a candidate for parallelization. Finally, we would like to explore the possibility of parallelizing other imaging operations in Julia, such as Laplacian of Gaussian filtering, resizing images, segmenting subcellular compartments, and more.

6. Acknowledgments

Many thanks to Ranjan, Shashi, and Valentin for the many hours they spent helping us debug. We also greatly appreciate the wonderful teaching of Professor Edelman and the soothing presence of his adorable corgi.

[1] David Feldman, Avtar Singh, Jonathan L Schmid-Burgk, Anja Mezger, Anthony J Garrity, Rebecca J Carlson, Feng Zhang, and Paul Blainey. *Pooled optical screens in human cells* `https://www.biorxiv.org/content/early/2018/08/02/383943`. bioRxiv, 2018.

[2] Xiaodong Yang, Houqiang Li, and Xiaobo Zhou. *Nuclei Segmentation Using Marker-Controlled Watershed, Tracking Using Mean-Shift, and Kalman Filter in Time-Lapse Microscopy.* IEEE 2006.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski and Viral B. Shah. *Julia: A Fresh Approach to Numerical Computing.* (2017) SIAM Review, 59: 6598.

[4] Olaf Ronneberger, Phillip Fischer, Thomas Brox *U-Net: Convolutional Networks for Biomedical Image Segmentation.* Medical Image Computing and Computer-Assisted Intervention (MICCAI)

[5] *Automated Training of Deep Convolutional Neural Networks for Cell Segmentation.* Scientific Reports 2017

[6] Fuyong Xing, Lin Yang *Robust Nucleus/Cell Detection and Segmentation in Digital Pathology and Microscopy Images: A Comprehensive Review.* IEEE Rev Biomed Eng. 2016; 9: 234-263

[7] Ekanathan Palamadai Natarajan. *Portable and Productive High-Performance Computing.* PhD Thesis submitted to the MIT Department of Electrical Engineering and Computer Science, February 2017.

[8] Yan Cao, Yanli Yang, Huamin Wang. *Integrated Routing Wasp Algorithm and Scheduling Wasp Algorithm for Job Shop Dynamic Scheduling.* 2008 International Symposium on Electronic Commerce and Security.

[9] Barnes, R., Lehman, C., Mulla, D. *Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models.* 2014. Computers Geosciences 62, 117127.