Image processing techniques and their parallel performance comparisons using CUDA

Luke Unterman

Problem:

- Identifying and conducting analysis on the various components within an image is the cornerstone of different computer vision applications
  - Optical character recognition
  - Medical imaging
  - Obstacle detection to improve automotive safety
- The dimensionality requirements for image analysis applications makes using the CPU infeasible
  - 4k image RGB image dimensions: 3840x2160x3=**24,883,200** pixels
- This project compares the GPU and CPU performance of several image processing techniques within a pipeline in order to produce the distinct segments.

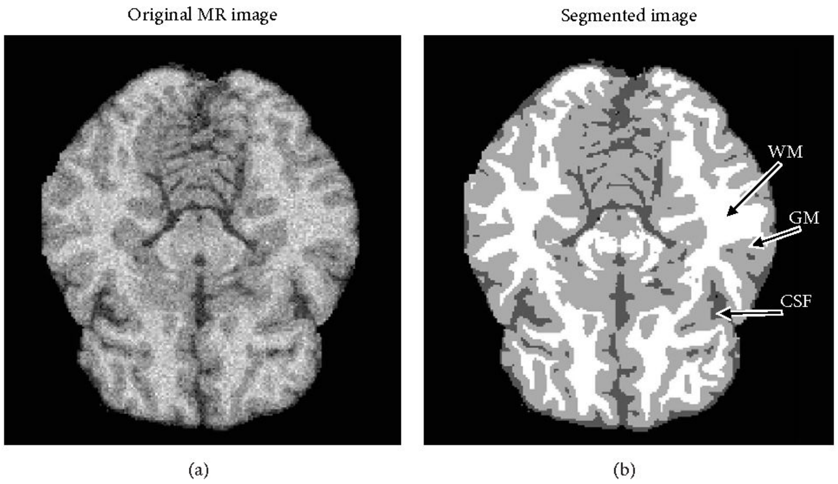
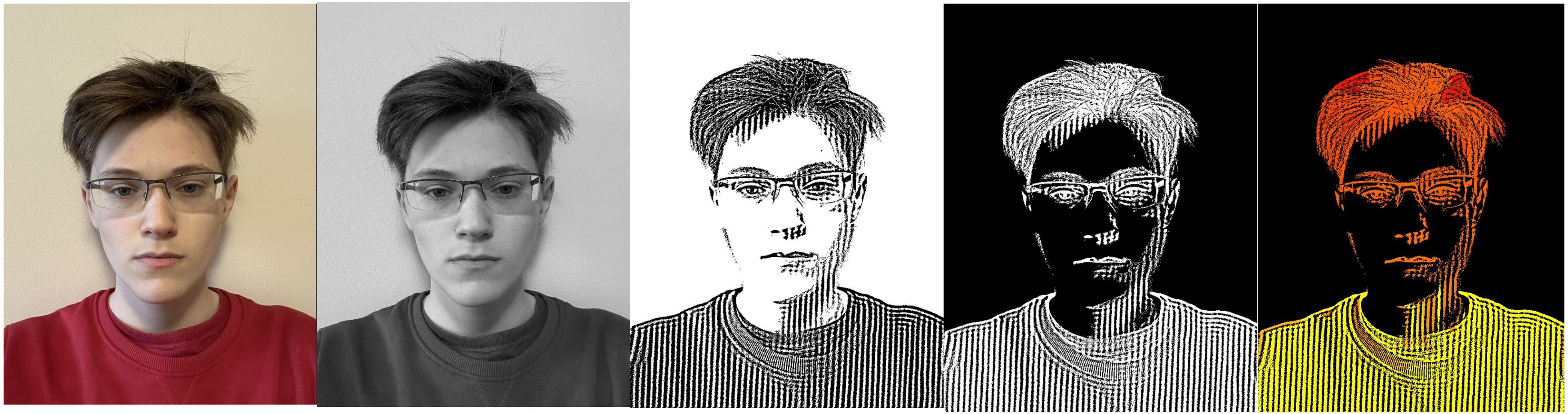


Figure 1. Identifying white matter, grey matter, and cerebrospinal fluid in brain. [1]

- Greater WM volume associated with progression of Alzheimer's disease [2]

Image Segmentation Algorithm Pipeline



RGB Image (accessed with OpenCV) → RGB To Gray $O(M*N)$ → Binarization $O(M*N*W^2)$ → Bitwise Not $O(M*N)$ → Connected Component Labeling $O(M*N)$

Most of these img. processing algorithms are embarrassingly parallel

Algorithm 1 SauvolaBinary

Require: Grayscale image, window size, parameter k, and dynamic range R

Ensure: Converts the image into a binary image using Sauvola's thresholding

for each pixel in the image **do**
  Define a window centered around the pixel
  Compute the mean intensity within the window
  Compute the standard deviation of intensities within the window
  Calculate the threshold:

$$\text{threshold} = \text{mean} \times \left(1 + k \times \left(\frac{\text{std\_dev}}{R} - 1\right)\right)$$

  if pixel intensity $>$ threshold **then**
    Set pixel to white (255)
  else
    Set pixel to black (0)
  end if
end for

Algorithm 2 BitwiseNot

Require: Grayscale image

Ensure: Negates each pixel value by performing a bitwise NOT operation

for each pixel in the image **do**
  Replace the pixel value with its bitwise NOT

$$\text{new\_value} = 255 - \text{current\_value}$$

end for

Algorithm 3 PadImage

Require: Image, padding size, and padding value

Ensure: Creates a padded version of the image

for each pixel in the padded image **do**
  if pixel is in the padding region **then**
    Set pixel value to the padding value
  else
    Copy the corresponding pixel value from the input image, adjusted by the padding offset
  end if
end for

The nested for loops required for the M * N rows and columns can be removed via a 2-d CUDA block mapping

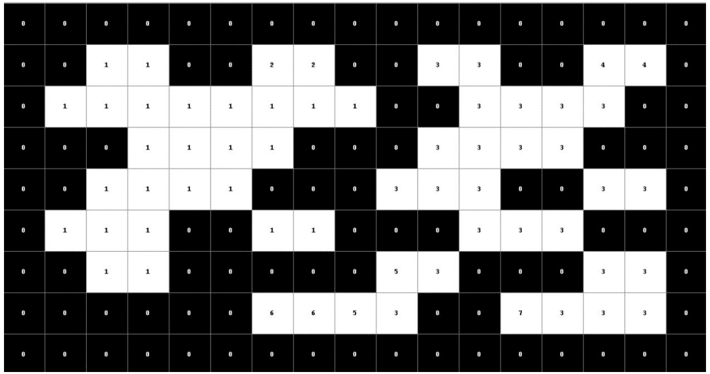
The Connected Component Labeling algorithm is much more complex...

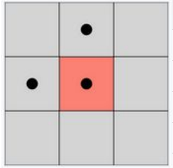
Algorithm 4 First Pass: Assign Provisional Labels and Record Equivalences

Require: Padded image `labelImage`, initial label index `k = 1`
Ensure: Assigns provisional labels and builds equivalence table

for each pixel (row, col) in `labelImage` **do**
  if current pixel is black (0) **then**
    continue
  end if
  Get the labels of the 4-connected neighbors
  Filter out zero neighbors
  if no neighbors have labels **then**
    Assign a new label k to the pixel and increment k
  else
    Assign the smallest neighbor label to the pixel
    Record equivalences between the assigned label and all neighbor labels
  end if
end for

Figure 2. First pass of two-pass connected component labeling algorithm. 4-connectivity.





4-connectivity

Set ID	Equivalent Labels
1	1,2
2	1,2
3	3,4,5,6,7
4	3,4,5,6,7
5	3,4,5,6,7
6	3,4,5,6,7
7	3,4,5,6,7

Note: CUDA implementation could require a global k value and global equivalence table

The Connected Component Labeling algorithm is much more complex...

Algorithm 4 Second Pass: Replace Provisional Labels with Resolved Labels

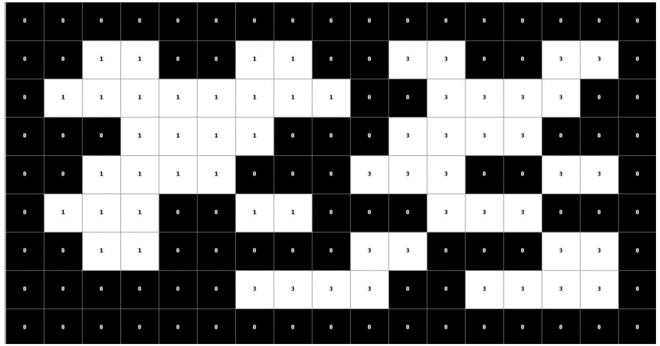
Require: Padded image `labelImage`, `flatEquivalenceMap`

Ensure: Updates labels in `labelImage` using resolved equivalences

```
for each pixel (row, col) in labelImage do
    Get the label of the current pixel
    if label exists in flatEquivalenceMap then
        Replace label with resolved label
    end if
end for
Remove the padding from labelImage
```

There is an intermediate step of flattening the equivalence table. Note: the second pass is much more parallelizable.

Figure 3. Second pass of two-pass connected component labeling algorithm. 4-connectivity.



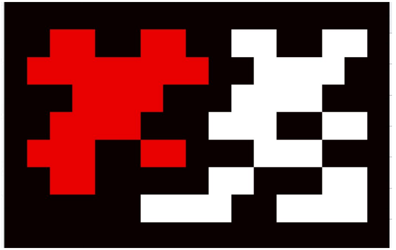


Figure 4. Image broken into distinct connected components

Parallelization Approach

- To exploit the **GPU's** inherent speed when it comes to parallel **SIMD** computation, each of these algorithms will be parallelized with **CUDA**.
    - Higher instruction and memory bandwidth than the **CPU**
- The **same** parallelization procedure can be applied to almost every single one of the image processing algorithms in the pipeline

General Parallelization Procedure:

1. Read in RGB image using OpenCV
2. Declare cv::Mat outputImage with same shape as input
3. Declare same variables on CUDA device, allocate memory, and copy inputImage memory from Host to Device
4. Devise blockDim and gridDim s.t. each thread is responsible for a pixel
5. Execute kernel to apply image transformation
6. Copy result from Device to Host
7. Save image to file

```
dim3 blockDim(threadsPerBlockPerDim, threadsPerBlockPerDim);
dim3 gridDim(gridDimx,gridDimy);
```

```
int column = blockIdx.x * blockDim.x + threadIdx.x;
int    row = blockIdx.y * blockDim.y + threadIdx.y;
```

Figure 5. Above: Designation of kernel block dimensions and grid dimensions. Below: assignment of unique index for each thread.

Example CUDA kernels for GPU-based image processing

- All non-CCL algorithms apply the same approach to divide work
- Nested for loop to iterate over pixels removed
- Highly parallel

Algorithm 1 Sauvola Binarization in CUDA

Require: *gray*: Grayscale image, *binary*: Output binary image, *width*, *height*: Dimensions of the image, *window_size*: Window size, k: Sauvola parameter

$x = blockIdx.x \cdot blockDim.x + threadIdx.x$
$y = blockIdx.y \cdot blockDim.y + threadIdx.y$
if $x < width$ **and** $y < height$ **then**
  $half\_window = window\_size/2$
  $mean = 0.0,\ std\_dev = 0.0,\ count = 0$
  for $wy = -half\_window$ **to** $half\_window$ **do**
    for $wx = -half\_window$ **to** $half\_window$ **do**
      $nx = \min(\max(x + wx, 0), width - 1)$
      $ny = \min(\max(y + wy, 0), height - 1)$
      $pixel = gray[ny \cdot width + nx]$
      $mean = mean + pixel$
      $std\_dev = std\_dev + pixel \cdot pixel$
      $count = count + 1$
    end for
  end for
  $mean = mean/count$
  $std\_dev = \sqrt{\left(\frac{std\_dev}{count}\right) - (mean \cdot mean)}$
  $threshold = mean \cdot \left(1 + k \cdot \left(\frac{std\_dev}{R} - 1\right)\right)$
  if $gray[y \cdot width + x] > threshold$ **then**
    $binary[y \cdot width + x] = 255$
  else
    $binary[y \cdot width + x] = 0$
  end if
end if

Algorithm 2 Connected Components Kernel (CUDA)

Require: *labelImage*: Image of labels, *equivalenceMap*: Map for label equivalences, *numberRows*: Number of rows, *numberColumns*: Number of columns

$column = blockIdx.x \cdot blockDim.x + threadIdx.x$
$row = blockIdx.y \cdot blockDim.y + threadIdx.y$
$idx = row \cdot numberColumns + column$
if $row < numberRows$ **and** $column < numberColumns$ **and** $column \geq 1$ **and** $row \geq 1$ **then**
  $label = labelImage[idx]$
  if $equivalenceMap[label] > 0$ **then**
    $labelImage[idx] = equivalenceMap[label]$
  end if
end if

Algorithm 3 Connected Components Serial Code (Host)

$ccTuple = CCInit(inputImage, paddedImage)$
$mapMax = ccTuple[0]$
$equivalenceMap = ccTuple[1]$
$flatEquivalences = \text{vector}(mapMax + 1, 0)$
for each $(key, value)$ **in** equivalenceMap **do**
  $flatEquivalences[key] = value$
end for
Allocate memory to d_equivalence on device
Copy flatEquivalences memory on host to d_equivale
Execute GPU ConnectedComponents kernel
Copy labelImage from device to outputImage on host

- Connected components labeling algorithm broken into serial and parallel components
- 1st-pass done on Host, 2nd-pass done on Device
- Naive, but does provide speedup

The nested for loops required for the M * N rows and columns can be removed via a 2-d CUDA block mapping`

Performance Evaluation of CPU and GPU methods

- Runtime (ms) recorded for serial and parallel image processing methods
- Constant image “selfie.jpg” with original dimensions 3088 x 2316 x 3 reduced by different factors (1, 2, 4, 8, and 16)
- Speedup and Scalability calculated for each of the parallel image processing algorithms
- Speedup defined as $\frac{T_s}{T_p}$ where T_s = time spent running algorithm serially and T_p = time spent running algorithm in parallel.
- Scalability defined as $\frac{T_{p=1}}{T_{p=N}}$ where p=1 implies threadsPerDimPerBlock = 1, and p=N implies threadsPerDimPerBlock=N.
- Constant threadsPerDimPerBlock value of 16 for the purposes of this experiment.
- All algorithmic executions performed on **cs.maple.vcu.edu** using NVIDIA GeForce 1080 Ti

Results

- Time required for both CPU and GPU implementations of each image operation plotted
- Time(ms) for each operation ranked in increasing order:
    1. Bitwise Flip
    2. Padding
    3. Grayscale
    4. Sauvola Thresholding
    5. CCL
- Operations 1-3 clustered closely together (parallel, same time complexity)
- Operation 4 is 2nd highest due to increased time complexity (iteration over window size)
- Operation 5: Same time complexity as 1-3, but reliant on serial pass.

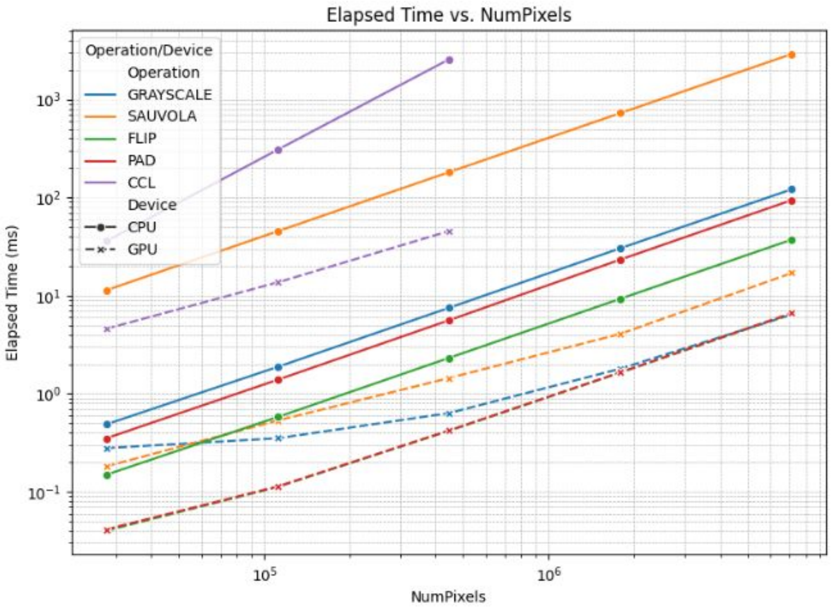


Fig. 1. Plot of elapsed time spent running CPU and GPU algorithms over the number of pixels in inputImage used. Note: log x, y, scale.

Results

- **Scalability** plotted over numPixels.

- Due to reliance on serial section, **CCL is least scalable** (no change despite increase in image size)

- Flip, and Padding algorithms mildly scalable , with peak at dimension reduction by scale of 8

- Sauvola thresholding algorithm highly scalable due to increased work

- Grayscale operation **exponentially scalable** (log plot) with increasing image size

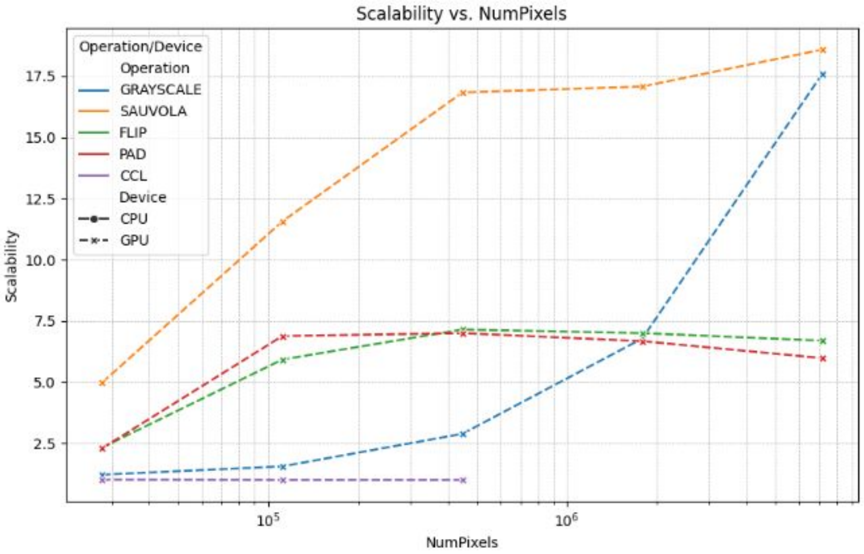


Fig. 2. Plot of the scalability of the parallel image processing operations at threadsPerNumPerDim=1,16 over number of pixels in inputImage. Note: log x scale.

Results

- **Speedup** plotted over numPixels.

- **Sauvola** thresholding GPU implementation maintains highest speedup value due to **increased proportion** of GPU work over **overhead** ($O(W^2)$ vs $O(MNW^2)$).

- Grayscale algorithm's speedup is slightly higher than the Pad/Flip algorithms due to **additional work** dealing with 3 image channels.

- CCL, **despite serial sections**, achieves moderate speedup.

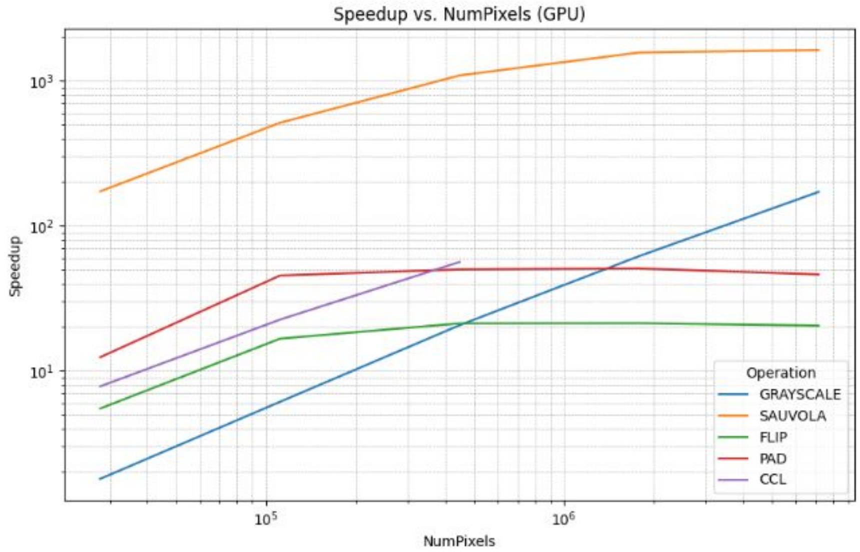


Fig. 3. Plot of the speedup of each of the parallel image-processing algorithms on the GPU over number of pixels in inputImage. Note: log x, y, scale.

Conclusions

1. This project compared the **runtime**, **speedup**, and **scalability** values of several different image processing techniques on the GPU and CPU (**ranked in increasing order** of runtime in ms).
    a. Bitwise Not <
    b. Image Padding <
    c. Grayscale Conversion <
    d. Binarization/Thresholding <<
    e. Connected Component Labeling
2. In particular, the **Sauvola thresholding binarization** algorithms performs **quite well** on the GPU due to its large dimensionality reduction ($O(MNW^2) \Rightarrow O(W^2)$).
3. Despite the serial sections in the parallel CCL implementation, it still achieves moderate speedup.
4. Image processing algorithms in general are fairly easy to implement on the GPU

Citations

- Brain scan:
  - Heckemann, R. A., Keihaninejad, S., Aljabar, P., Rueckert, D., Hajnal, J. V., Hammers, A., May 2010. Improving intersubject image registration using tissue-class information benefits robustness and accuracy of multi-atlas based anatomical segmentation. NeuroImage 51 (1), 221-227. http://dx.doi.org/10.1016/j.neuroimage.2010.01.072
  - Garnier-Crussard, A., Bougacha, S., Wirth, M., Dautricourt, S., Sherif, S., Landeau, B., Gonneaud, J., De Flores, R., de la Sayette, V., Vivien, D., Krolak-Salmon, P., & Chételat, G. (2022). White matter hyperintensity topography in Alzheimer's disease and links to cognition. *Alzheimer's & dementia : the journal of the Alzheimer's Association*, *18*(3), 422–433. https://doi.org/10.1002/alz.12410
- Connected component labeling images:
  - Wikipedia contributors. (2024, November 5). Connected-component labeling. In Wikipedia, The Free Encyclopedia. Retrieved 02:41, December 10, 2024, from https://en.wikipedia.org/w/index.php?title=Connected-component_labeling&oldid=1255521016