Image processing techniques and their performance comparison using CUDA

Luke Unterman
Department of Computer Science
Virginia Commonwealth University
Richmond, USA
untermanlm@vcu.edu

***Abstract*—This report summarizes the results achieved for improving the serialized versions of different image processing algorithms using NVIDIA's CUDA architecture. These different image processing algorithms include algorithms that are used to 1) convert three-channel RGB images into single-channel grayscale images (Grayscale), 2) binarize grayscale images via Sauvola's adaptive thresholding, 3) return the inverse of a binary image (BitwiseNot), 4) pad an image, and 5) find the connected components from within the image (Connected Component Labeling, CCL). When ranking each operation by its range of runtimes in increasing order, it is clear that BitwiseFlip $<$ Padding $<$ Grayscale $<$ Sauvola Thresholding $<$ CCL as a result of the Grayscale algorithm's requirement of handling the pixels of an RGB image as opposed to a grayscale image, Sauvola Thresholding's additional requirement of calculating the local mean and standard deviation to devise a local threshold value, and parallel CCL's reliance upon serial code to complete the first pass and resolve step(s) of the 4-connectivity two-pass connected components algorithm. Furthermore, the most scalable of the parallel image-processing implementations is the Sauvola Thresholding algorithm, once again as a result of its parallel time complexity of $O(W^2)$ instead of $O(1)$ like the other algorithms.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

This report summarizes the results achieved for improving the serialized versions of different image processing algorithms using NVIDIA's CUDA architecture. These different image processing algorithms include algorithms that are used to:

- 1) Convert three-channel RGB images into single-channel grayscale images
- 2) Binarize grayscale images via global and adaptive local thresholding
- 3) Return the inverse of a binary image
- 4) Pad an image
- 5) Find the connected components from within the image

The serial implementations of these algorithms have time complexities of $O(MN)$, $O(MN)$/$O(MNW^2)$, $O(MN)$, and $O(MN)$ respectively, where M is equal to the number of rows in an image (height), N is equal to the number of columns in an image (width), and W refers to the window size of a local region in an image. The parallel runtime complexities of these algorithms become $O(1)$, $O(1)$/$O(W^2)$, $O(1)$, and $O(1)$ by mapping each thread in a CUDA block to a unique row and column value.

The RGB-to-Gray algorithm converts images composed of three channels into a grayscale image with a single channel. This operation is often the basis for other image processing techniques, as it reduces an image into only its luminance. This conversion can be done by replacing an input image's original three R, G, and B channels with their associated Luma value. The Luma value, Y' [1] according to the ITU-R BT.601 standard is equal to

$$Y' = 0.299R' + 0.587G' + 0.114B'$$

Thus, the serial algorithm for this conversion is as follows:

- For each row in M, for each column in N:
  - RGBPixel = RGBImage[row][col]
  - Luma = 0.299 * RGBPixel[0] + 0.587 * RGBPixel[1] + 0.114*RGBPixel[2]
  - Grayscale[row][col] = Luma

Since the Luma value is calculated in constant time, the time complexity of this algorithm is $O(MN)$.

Image Binarization converts a grayscale image into a binary image as another preprocessing step for image processing tasks such as image segmentation. A binary image is a single-channel image solely composed of either white pixels or black pixels. The binarization of an image is done to distinctly separate an image from its foreground and its background, and can be done quite simply with a global threshold value. Assuming that the global threshold value is passed in as a parameter, then a grayscale image can be converted into a binary image in $O(MN)$ time. Simply, if the pixel of a grayscale image has a Luma value greater than this threshold, set the Luma value of the pixel to 255 (white). Otherwise, set its Luma value to 0 (black). Thus, the algorithm's pseudocode is:

- For each row in M, for each column in N:
  - Luma = Grayscale[row][col]
  - If Luma $>$ threshold, Grayscale[row][col] = 255 else Grayscale[row][col] = 0

A clear problem based on this threshold input parameter emerges, however, as the optimal threshold to binarize an image varies from image to image, and different subsections of an image have variable Luma ranges. A method called Otsu's thresholding [2] solves the first problem and calculates an optimal global threshold (assuming a bimodal histogram)

for an image by finding the threshold which minimizes the intra-class variance between background pixels and foreground pixels. However, it alone does not solve the second problem of having subregions within the grayscale image with variable Luma ranges.

A subset of thresholding algorithms which do consider the local subregions of a grayscale image are referred to as Adaptive Thresholding algorithms. Sauvola thresholding [3] is a type of adaptive thresholding algorithm which is particularly adapted for document analysis. The Sauvola algorithm computes a threshold value for each pixel based on the mean and standard deviation of the pixel values within a window centered at the original pixel. This threshold value is calculated as follows:

$$\text{threshold} = \text{local\_mean} \cdot \left[1 + k \cdot \left(\frac{\text{local\_std}}{R} - 1\right)\right]$$

The variables k and R are input parameters which 1) modify how much influence the local standard deviation has on the local threshold value and 2) represent the dynamic range of standard deviation, respectively. Thus, because each pixel at (row, col) has to calculate the sum of values from within a 2-d window, the time complexity of this algorithm is $O(MNW^2)$. For the entire image processing pipeline required to get the connected components of an image, this can be a bottleneck depending on the window size.

There are numerous different types of CCL algorithms [5] but the implementation in this paper uses a basic two-pass 4-connectivity algorithm described by Lumia et al [4]

For this algorithm, given a padded binary image, all non-padded pixels are checked to see if they are white. If they are, and they don't have any neighbors to the left of them or above them that are white, then they are set to equal a distinct label and the label is incremented. If they do have white neighbors, then the non-padded pixel is set to the minimum of the neighbors' labels and the non-minimum labels are recorded in an equivalence table. After this first pass, a resolve step occurs to link together the equivalences, and then in a second pass the non-padded pixels are set as equal to their equivalent label. The first and second scans occur in $O(MN)$ time, and the resolve step is bounded by the total number of pixels in the image and thus occurs in $O(MN)$ time as well. Thus, overall, the time complexity of this algorithm is $O(MN)$.

Finally, the Pad and Inverse algorithms are self explanatory as they apply a border/padding around an image and set an image's contents to the inverse of its pixel value.

II. Parallelization Approach: CUDA

Because the GPU is specifically designed to be able to process images in parallel as a result of its large number of simple cores for parallel SIMD computation, and because the GPU provides higher instruction and memory bandwidth than the CPU, parallelizing this image processing pipeline using CUDA seems to be the best approach. CUDA [6], which stands for Compute Unified Device Architecture, is a parallel computing platform and API developed by NVIDIA expressly to parallelize the type of image processing techniques previously described in this paper. Thus, this section will discuss the implementation technique and approach used for parallelization with CUDA.

A. Basic Protocol for Image Processing Parallelization with CUDA

The general procedure used in this article for parallelizing each of the aforementioned image processing operations is as follows:

- Read in the given RGB image through system arguments using OpenCV's cv::imread() function or through cv::VideoCapture() and save it as a cv::Mat type.
- Also, declare another cv::Mat for the outputImage and set its type (unsigned char for all except padding, which is an integer).
- Declare pointers to CUDA device variables for the given input image and for the output image to be returned by the CUDA kernel function. Allocate memory to these CUDA device variables, and copy the memory of the input image on the host to the input image on the device.
- Using 16 threads per block for each dimension, calculate gridDimX, gridDimY, and blockDim so that each CUDA thread is responsible for its own pixel. This essentially eliminates the nested for loops in every serial image processing algorithm responsible for the $O(MN)$ time complexity.
- Call the kernel function with the required input parameters.
- Each thread performs a calculation to manipulate their respective pixel's value, and this altered value is saved to the output image/Mat.
- Copy the device's output image to host memory, and write the image to file using cv::imwrite().
- Free the device variables which were allocated memory.

The two algorithms of the five aforementioned image processing techniques that are highlighted in this section are the Sauvola Thresholding and Connected Component Labeling algorithms, as they are slightly more in depth than the other algorithms. The other algorithms (RGBToGray, Global Binarization, Padding, and BitwiseNot) are essentially the same as their serial counterparts except that their nested for loop of time complexity $O(MN)$ is removed, so they will only be shown in the Appendix.

The Sauvola binarization/adaptive thresholding algorithm's (Algorithm 1) parallel implementation is slightly more interesting, as it involves the use of a sliding window to allow each CUDA thread to compute its own local mean and standard deviation values. This works through the use of the indexing values nx and ny and min/max operations to ensure that the indices are no more than one-half of the window to the left, right, above, or below the thread's own index (the thread's idx is the center of the window). Then, the local threshold for each thread is calculated using the previously mentioned Sauvola formula via the local mean and standard deviation values.

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

Finally, depending on what the thread's associated pixel value is, the output image's pixel is set to either black or white.

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

The Connected Component Labeling algorithm's implementation is composed of one serial section (Algorithm 3) for the first-pass of the algorithm and the equivalence table resolving intermediate step, and a parallel section for the second-pass where each label in inputImage is set as equal to its associated equivalent label. The serial work is first completed by the CCInit method, which applies exactly the same steps as the original serial implementation. Then, once the equivalence table has been populated with values and the inputImage has been updated to contain label values, a tuple is returned to the main function where the unordered_map data structure used in the serial method is converted into a vector so that it can be copied to the CUDA device's memory. Then, the device's equivalence table is allocated memory and copied to the CUDA device's memory, and is used as an input parameter to the parallel ConnectedComponents kernel (Algorithm 2) along with a padded image.

Algorithm 3 Connected Components Serial Code (Host)

$ccTuple = CCInit(inputImage, paddedImage)$
$mapMax = ccTuple[0]$
$equivalenceMap = ccTuple[1]$
$flatEquivalences =$ 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_equivalence
Execute GPU ConnectedComponents kernel
Copy labelImage from device to outputImage on host

This kernel essentially performs the second pass of the two-pass CCL algorithm, where each CUDA thread is responsible for updating the labelImage with its equivalent label. Finally, after the kernel has finished running, labelImage is copied to host memory and written to a folder as a file. While the parallelization approach to the 4-connectivity Connected Components algorithm described in this section is naive, it has still achieved a moderate speedup in comparison to the serial algorithm.

III. Performance Evaluation

In order to determine how each of my parallel image processing algorithms increase in runtime, speedup, and scalability as the size of an input image increases, the runtime performance in milliseconds was recorded for different sizes of an input image "selfie.png" with original dimensions of 3088 x 2316. More specifically, the performance of each of the image processing algorithms on the CPU and GPU was recorded for different downsized versions of the input image (1x, 2x, 4x 8x, 16x). The runtime for each CUDA implementation was recorded twice with threadPerBlockPerDim values of 1 and 16 in order to be able to calculate their respective scalability values.

Speedup is defined as

$$\text{Speedup} = \frac{T_s}{T_p}$$

where T_s is equal to the serial time required to run the algorithm and T_p is equal to the time in parallel required to run the algorithm.

For the purposes of this runtime performance analysis, scalability is defined as

$$\text{Scalability} = \frac{T_{p=1}}{T_{p=N}}$$

where $T_{p=1}$ refers to an algorithm's parallel performance at threadsPerDimPerBlock = 1, and $T_{p=N}$ refers to an algorithm's parallel performance at threadsPerDimPerBlock = 16.

Note that the GPU used for these runtime performances was an NVIDIA GeForce GTX 1080 Ti.

IV. Results and Discussion

In Figure 1, the time required for both the CPU and GPU implementations of the Grayscale, Sauvola Thresholding, BitwiseFlip, Padding, and CCL image processing techniques is plotted over the number of pixels used in the given input image. When ranking each operation by its range of runtimes in increasing order, it is clear that BitwiseFlip $<$ Padding $<$ Grayscale $<$ Sauvola Thresholding $<$ CCL. More specifically, the runtimes of the BitwiseFlip, Padding, and Grayscale algorithms are clustered closely together as a result of their 1) lack of additional for loops within the CUDA kernel and 2) lack of serial sections. While the kernel function for Sauvola Thresholding employs the use of a nested for loop to compute the local mean and standard deviations surrounding each thread's index, and therefore has a higher computational complexity than the parallel CCL algorithm, it is faster than the CCL algorithm as a result of its lack of additional serial sections. Note that the CPU and GPU implementations of the CCL algorithm lack two points in comparison to the other algorithms. This is because the CPU implementation of the CCL algorithm runs out of memory with inputImage size reductions of 1 and 2.

In Figure 2, the scalability of the CUDA-based parallel implementations is plotted over the number of pixels in the given input image. Notably, the scalability of the CCL algorithms does not visibly increase despite the increase in number of threadsPerNumPerDim. This indicates that the performance of the GPU-based CCL algorithm is heavily tied to its prominent serial section. The BitwiseNot/Flip and Padding algorithms peak in their scalability value with an input image size of M/8, N/8, and their scalability remains constant after this peak. The scalability of the Grayscale algorithm never peaks, indicating that it is the most scalable out of all of the different image-based algorithms. Finally, the Sauvola Thresholding algorithm's scalability grows exponentially until the inputImage is downsized by a factor of 2, at which point the scalability decreases.

Finally, in Figure 3, the speedup of each of the parallel image-processing algorithms on the GPU is plotted over the number of pixels in the given input image. The speedup of the Sauvola Thresholding algorithm is the highest out of each image processing operation, likely as a result of the increased work required to compute the local mean and standard deviation values in comparison to the other algorithms. The speedup of the CCL algorithm increases with the size of the input image due to its usage of parallel code run on the GPU for its second pass. The BitwiseNot and Padding algorithms have initially higher speedup values than the Grayscale algorithm as a result of working with one image channel instead of three, but the speedup of the Grayscale algorithm surpasses that of the BitwiseNot and Padding algorithms as the size of the input image increases.

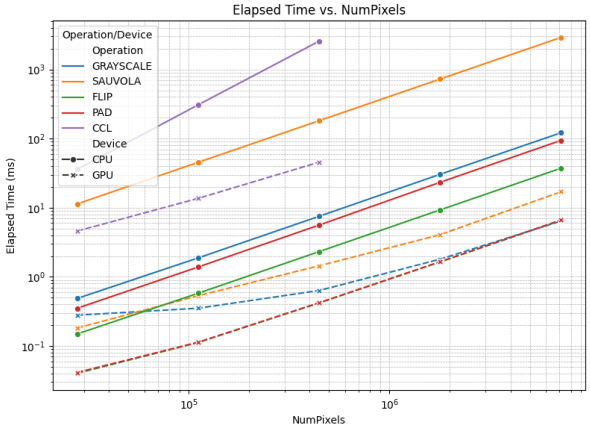

Fig. 1. Plot of elapsed time spent running CPU and GPU algorithms over the number of pixels in inputImage used. Note: log x, y, scale.

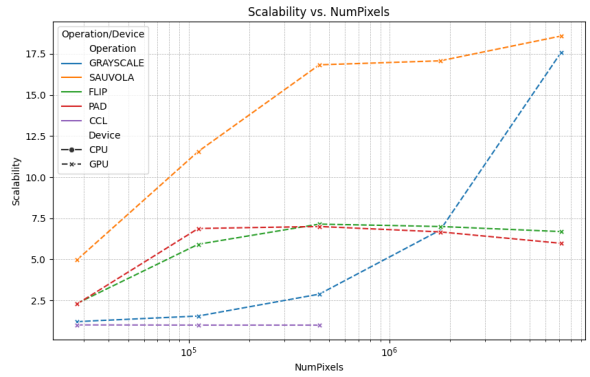

Fig. 2. Plot of the scalability of the parallel image processing operations at threadsPerNumPerDim=1,16 over number of pixels in inputImage. Note: log x scale.

Conclusion

In conclusion, we have presented the serial and parallel CUDA-based Grayscale, Sauvola Thresholding, Padding, BitwiseNot, and Connected Component Labeling algorithms. When ranking each operation by its range of runtimes in increasing order, it is clear that BitwiseFlip $<$ Padding $<$ Grayscale $<$ Sauvola Thresholding $<$ CCL as a result of the Grayscale algorithm's requirement of handling the pixels of an RGB image as opposed to a grayscale image, Sauvola Thresholding's additional requirement of calculating the local mean and standard deviation to devise a local threshold value,

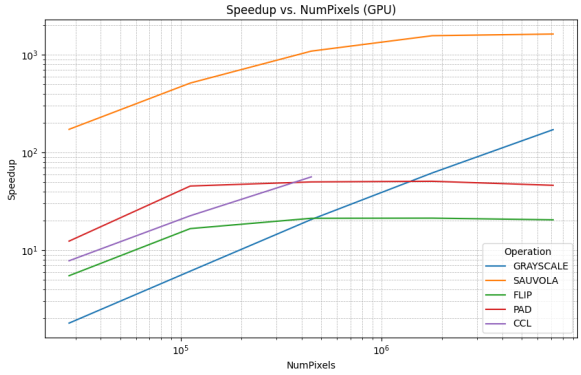

Fig. 3. Plot of the speedup of each of the parallel image-processing algorithms on the GPU over number of pixels in inputImage. Note: log x, y, scale.

and parallel CCL's reliance upon serial code to complete the first pass and resolve step(s) of the 4-connectivity two-pass connected components algorithm. Furthermore, the most scalable of the parallel image-processing implementations is the Sauvola Thresholding algorith, once again as a result of its parallel time complexity of $O(W^2)$ instead of $O(1)$ like the other algorithms.

References

[1] Bezryadin, S., Bourov, P., & Ilinih, D. (2007, January). Brightness calculation in digital image processing. In *International symposium on technologies for digital photo fulfillment* (Vol. 1, pp. 10-15). Society for Imaging Science and Technology.

[2] Otsu, N. (1979). A threshold selection method from gray level histograms. *IEEE Transactions on Systems, Man, and Cybernetics, 9*(1), 62–66.

[3] Sauvola, J., & Pietikäinen, M. (2000). Adaptive document image binarization. *Pattern Recognition, 33*(2), 225-236.

[4] Lumia, R., Shapiro, L., & Zuniga, O. (1983). A new connected components algorithm for virtual memory computers. *Pattern Recognition Letters, 22*(2), 287–300. https://doi.org/10.1016/0734-189X(83)90071-3

[5] He, L., Ren, X., Gao, Q., Zhao, X., Yao, B., & Chao, Y. (2017). The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition, 70*, 25-43. https://doi.org/10.1016/j.patcog.2017.04.018

[6] Sanders, J., & Kandrot, E. (2010). *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley Professional.

V. Appendix

A. CUDA Kernels

Algorithm 4 BGR to Grayscale Conversion (CUDA Kernel)

Require: $BGRImage$: Input BGR image, $grayImage$: Output grayscale image, $numberRows$: Number of rows, $numberColumns$: Number of columns

$column = blockIdx.x \cdot blockDim.x + threadIdx.x$
$row = blockIdx.y \cdot blockDim.y + threadIdx.y$
if $column < numberColumns$ **and** $row < numberRows$ **then**
  $pixel = BGRImage[row \cdot numberColumns + column]$
  $grayValue = (0.114 \cdot pixel.x) + (0.587 \cdot pixel.y) + (0.299 \cdot pixel.z)$
  $grayImage[row \cdot numberColumns + column] = grayValue$
end if

Algorithm 5 Bitwise NOT Operation (CUDA Kernel)

Require: $binary$: Input binary image, $binaryFlip$: Output flipped binary image, $numberRows$: Number of rows, $numberColumns$: Number of columns

$column = blockIdx.x \cdot blockDim.x + threadIdx.x$
$row = blockIdx.y \cdot blockDim.y + threadIdx.y$
if $column < numberColumns$ **and** $row < numberRows$ **then**
  $pixel = binary[row \cdot numberColumns + column]$
  $binaryFlip[row \cdot numberColumns + column] = \sim pixel$
end if

Algorithm 6 Padding Image (CUDA Kernel)

Require: $input$: Input image, $output$: Padded output image, $numRows$: Number of rows, $numCols$: Number of columns, $padNumRows$: Number of rows after padding, $padNumCols$: Number of columns after padding, $padSize$: Size of padding, $padValue$: Padding value

$column = blockIdx.x \cdot blockDim.x + threadIdx.x$
$row = blockIdx.y \cdot blockDim.y + threadIdx.y$
if $column < padNumCols$ **and** $row < padNumRows$ **then**
  $value = padValue$
  $notInPadRow = row - padSize \geq 0$
  $notInPadCol = column - padSize \geq 0$
  if $notInPadRow$ **and** $notInPadCol$ **and** $row < numRows + padSize$ **and** $column < numCols + padSize$ **then**
    $value = input[(row - padSize) \cdot numCols + (column - padSize)]$
  end if
  $output[row \cdot padNumCols + column] = value$
end if