Nvidia Warp Based Image Processing Report

Lucas Sudol
Sebastian Kula

Table of Contents

How to Run Program

Usage: python3 imageProcess.py <algType> <kernSize> <param> <inFileName> <outFileName>

Arguments:

<algType>	Algorithm type: specify "-s" for sharpening or "-n" for noise removal
<kernSize>	Size of the kernel (e.g., 3, 5, 7)
<param>	Parameter for the algorithm (e.g., intensity factor)
<inFileName>	Input image file name (e.g., input.jpg)
<outFileName>	Output image file name (e.g., output.jpg)

Example:

python3 imageProcess.py -s 3 0.3 input.jpg output.jpg

Image Processing Algorithms Implemented

Image Sharpening (Unsharp Mask Algorithm)

This method sharpens the image by first applying a mean averaging filter to create a blurred version. The blurred image is subtracted from the original to extract the edges. These edges are then added back to the original image with a scaling factor determined by the parameter param.

Noise Reduction (Gaussian Filter)

Noise reduction is achieved using a Gaussian filter. Pixels near the center of the kernel have higher weights, while those further away contribute less. The spread of the filter is controlled by param, which also influences the size of the kernel.

How kernel was Implemented in Warp API

Unsharp Mask Algorithm

```
    # sharpening kernel
    i, j, k = wp.tid()
    flipx = 1
    flipy = 1

    #Get offsets for i,j for kernel size
    for x in range(-(kern_size-1)/2, (kern_size-1)/2 + 1):
        for y in range(-(kern_size-1)/2, (kern_size-1)/2 + 1):
            #Border Handling by reflection
            if i+x < 0 or i+x > output_image.shape[0]: #check if tile is
below 0 or above max x value
                flipx = -1
```

```
            if j+y < 0 or j+y > output_image.shape[1]: #check if tile is
below 0 or above max y value
                flipy = -1

         #Accumulate tiles to output image
            output_image[i, j, k] += input_image[i + x*flipx,
                                                 j + y*flipy, k]
            flipx = 1
            flipy = 1

    #Divide by number of tiles to find mean
    output_image[i, j, k] *= (1.0/float(kern_size * kern_size))

   #calculate edge image, and add to original multiply based on parameter
    output_image[i, j, k] = (input_image[i,j,k] +
                             param*(input_image[i,j,k]
                             - output_image[i,j,k]))


    #Check for value rollover
    if(output_image[i, j, k] > 255.0):
        output_image[i, j, k] = 255.0

    if (output_image[i, j, k] < 0.0):
        output_image[i, j,k ] = 0.0
```

The sharpening kernel begins by iterating over the offsets for every pixel within the kernel size. So for a 3x3 kernel, the offsets will range from -1,-1 to 1,1, from the origin pixel in the center of the kernel. These loops gather pixel values from the surrounding area to calculate the local average, effectively blurring the image. Each pixel from the neighbourhood is accumulated to the output array index of the center pixel. When encountering pixels that would lay outside the image boundaries, the offsets are reflected according to the border handling strategy discussed later on. The sum of all the pixels in the kernel are then divided by the size of it, getting the mean value. The local average is then subtracted from the original pixel value to create an edge image. This is an image that exaggerates high intensity areas of the image. This edge is then added back to the original image, using a multiplicative constant passed in as a parameter. A larger constant will increase the image's overall sharpness. The final pixel values are then restricted between 0 and 255 to prevent rollovers.

Gaussian Filter

```
# noise removal kernel
    i, j, k = wp.tid()
    flipx = 1
    flipy = 1

    weights = float(0.0)
    weight = float(0.0)


    #Get offsets for i,j for kernel size
    for x in range(-(kern_size-1)/2, (kern_size-1)/2 + 1):
        for y in range(-(kern_size-1)/2, (kern_size-1)/2 + 1):
            #Border Handling by reflection
            if i+x < 0 or i+x > output_image.shape[0]: #check if tile is
below 0 or above max x value
                flipx = -1

            if j+y < 0 or j+y > output_image.shape[1]: #check if tile is
below 0 or above max y value
                flipy = -1

            weight = wp.exp(-(float(x*x) + float(y*y)) / (2.0 * param
                                                          * param))
            weights += weight

            #Accumulate tiles to output image
            output_image[i, j, k] += (input_image[i + x*flipx,
                                                  j + y*flipy, k] * weight)
            flipx = 1
            flipy = 1

    #Divide by number of tiles
    output_image[i, j, k] *= (1.0 / weights)
```

The noise removal kernel also iterates over every pixel using i and j as row and column indices, respectively. It applies a Gaussian filter by summing pixel values from a surrounding neighbourhood, with weights determined by the Gaussian function. The nested for loops iterate over the neighbourhood, and for each offset (x, y), the weight is calculated based on the distance from the center of the kernel. Based on the passed parameter, the spread of the weights can be

increased or reduced. A lower value will cause values closer to the kernel center to be weighted higher. If a neighbor lies outside the image, reflection-based border handling mirrors the values back into the valid range. The accumulated weighted sum is then normalized by dividing by the total weight sum (weights), effectively reducing noise while preserving important details.

Border Handling

We handle borders using a reflection method, where any pixel accessed outside the image boundaries is replaced by a corresponding pixel from within the image, effectively mirroring the values. For example, if a pixel in the top left of the grid is out of bounds, the kernel will utilize the value in the bottom right of the grid. This applies to any grid size and ensures that all pixels, including those at the edges and corners of an image, are processed correctly without leaving any gaps or artifacts.

For example:

(red indicates pixels beyond the image border)

A B C
D E F
G H I

When value A is read, the value from G is returned.
When value B is read, the value for H is returned, and so on.

```
flipx = 1
flipy = 1

#Border Handling by reflection
           if i+x < 0 or i+x > output_image.shape[0]: #check if tile is
below 0 or above max x value
                flipx = -1

            if j+y < 0 or j+y > output_image.shape[1]: #check if tile is
below 0 or above max y value
                flipy = -1

output_image[i, j] += (input_image[i + x*flipx, j + y*flipy] * weight)
```