Project report
Advanced programming for HPC

Ho Duc Trung

January 2023

1 Theory

1.1 Convert image RGB to HSV

HSV is a common color system, consisting of three components: Hue (H), saturation (S), and value (V). To convert the image from RGB to HSV, use the following steps:

- Normalize R, G, B from [0..255] to [1..255]
- Find max and min of R, G, B after normalization.
- Calculate Δ = max - min

$$H = \begin{cases} 0^o & \Delta = 0 \\ 60^o \times (\frac{G-B}{\Delta} \bmod 6) & max = R \\ 60^o \times (\frac{B-R}{\Delta} + 2) & max = G \\ 60^o \times (\frac{R-G}{\Delta} + 4) & max = B \end{cases}$$

$$S = \begin{cases} 0 & max = 0 \\ \frac{\Delta}{max} & max \neq 0 \end{cases}$$

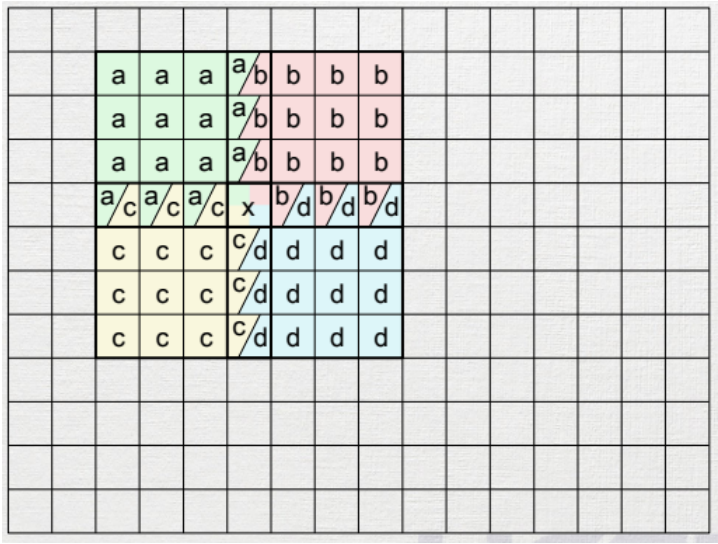
$$V = max$$

In this project, when calculating with the Kuwahara filter, only the value of V is needed to be used, so the steps to calculate H and S can be skipped.

1.2 Kuwahara Filter

Kuwahara filter is used in image processing to smooth the image, effectively reducing noise and blurring edges. This filter after being applied also creates

an effect that makes the photo look like a painting.
Each image pixel is processed by calculation based on the windows around it, defined as the figure shows:
x is the pixel being considered, and the 4 surrounding windows are a, b, c, and d respectively. Do Standard Deviation (SD) calculations for these windows, and select the window with the largest SD to calculate the color value for pixel x.



Applying the Kuwahara filter to each color channel R, G, and B we get a new image, this is the final result we want to get after using the Kuwahara filter.

2 Project implementation

In this project, implementing Kuwahara filter calculation for 2 CPU and GPU cases (without using Shared Memory).

2.1 CPU calculation

The function calculates V matrix

```
def calV(src):
  dst = np.zeros((src.shape[0],src.shape[1]))
  for tidx in range(src.shape[0]):
    for tidy in range(src.shape[1]):
      R = src[tidx, tidy, 2]/255
      G = src[tidx, tidy, 1]/255
      B = src[tidx, tidy, 0]/255

      dst[tidx][tidy]= max(R, G, B)
  return dst
```

The function calculates SD and average color for a window

```
def calDS(cMat,vMat,fw,a,b,c,d):
  windowSum = 0
  windowSD = 0
  windowColor = 0

  f1 = a
  t1 = b
  f2 = c
  t2 = d

  if a < 0:
    f1 = 0
  if b > len(cMat)-1:
    t1 = len(cMat)-1
  if c < 0:
    f2 = 0
  if d > len(cMat[0])-1:
    t2 = len(cMat[0])-1

  for i in range(f1,t1+1):
    for j in range(f2,t2+1):
        windowSum += vMat[i][j]
        windowColor += cMat[i][j]
  windowMean = windowSum/((fw+1)*(fw+1))

  for i in range(f1,t1+1):
    for j in range(f2,t2+1):
        windowSD += (vMat[i][j] - windowMean)*(vMat[i][j] - windowMean)/((fw+1)*(fw+1))

  result = math.sqrt(windowSD)
  windowMeanColor = windowColor/((fw+1)*(fw+1))
  return result, windowMeanColor
```

The Function to apply Kuwahara filter

```
def kuwahara(cmat, vmat, w):
  dst = np.zeros((len(cmat),len(cmat[0])))
  for tidx in range(len(cmat)):
    for tidy in range(len(cmat[0])):
      dsA, colorA = calDS(cmat, vmat, w, tidx - w, tidx, tidy - w, tidy)
      dsB, colorB = calDS(cmat, vmat, w, tidx, tidx + w, tidy - w, tidy)
      dsC, colorC = calDS(cmat, vmat, w, tidx - w, tidx, tidy, tidy + w)
      dsD, colorD = calDS(cmat, vmat, w, tidx, tidx + w, tidy, tidy + w)

      minWl = min(dsA, dsB, dsC, dsD)
      if minWl == dsA:
        dst[tidx, tidy] = colorA
      if minWl == dsB:
        dst[tidx, tidy] = colorB
      if minWl == dsC:
        dst[tidx, tidy] = colorC
      if minWl == dsD:
        dst[tidx, tidy] = colorD
  return dst
```

Main Function

```
img = root.copy()

v_mat = calV(img)

blue_channel = img[:,:,0]
green_channel = img[:,:,1]
red_channel = img[:,:,2]

w = 5

blue_mat = kuwahara(blue_channel, v_mat, w)
green_mat = kuwahara(green_channel, v_mat, w)
red_mat = kuwahara(red_channel, v_mat, w)

combine_img = np.dstack((blue_mat,green_mat,red_mat))
imgplot = plt.imshow(img)
plt.show()
imgplot = plt.imshow(combine_img)
plt.show()
```

2.2 GPU calculation (without using Shared Memory)

The function calculates V matrix

```
@cuda.jit
def calV(src, dst):
    tidx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    tidy = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y

    R = src[tidx, tidy, 2]/255
    G = src[tidx, tidy, 1]/255
    B = src[tidx, tidy, 0]/255

    dst[tidx, tidy, 0] = max(R, G, B)
```

The function calculates SD and average color for a window

```
@cuda.jit(device=True)
def calDS(cMat,vMat,fw,a,b,c,d):
  windowSum = 0
  windowSD = 0
  windowColor = 0

  f1 = a
  t1 = b
  f2 = c
  t2 = d
  if a < 0:
    f1 = 0
  if b > len(cMat)-1:
    t1 = len(cMat)-1
  if c < 0:
    f2 = 0
  if d > len(cMat[0])-1:
    t2 = len(cMat[0])-1

  for i in range(f1,t1+1):
    for j in range(f2,t2+1):
        windowSum += vMat[i][j][0]
        windowColor += cMat[i][j]
  windowMean = windowSum/((fw+1)*(fw+1))

  for i in range(f1,t1+1):
    for j in range(f2,t2+1):
        windowSD += (vMat[i][j][0] - windowMean)*(vMat[i][j][0] - windowMean)/((fw+1)*(fw+1))

  result = math.sqrt(windowSD)
  windowMeanColor = windowColor/((fw+1)*(fw+1))
  return result, windowMeanColor
```

The Function to apply Kuwahara filter

```
@cuda.jit
def kuwahara(src, dst, w, cmat):
  tidx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
  tidy = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y

  dsA, colorA = calDS(cmat, src, w, tidx - w, tidx, tidy - w, tidy)
  dsB, colorB = calDS(cmat, src, w, tidx, tidx + w, tidy - w, tidy)
  dsC, colorC = calDS(cmat, src, w, tidx - w, tidx, tidy, tidy + w)
  dsD, colorD = calDS(cmat, src, w, tidx, tidx + w, tidy, tidy + w)

  minWl = min(dsA, dsB, dsC, dsD)

  if minWl == dsA:
    dst[tidx, tidy] = colorA
  if minWl == dsB:
    dst[tidx, tidy] = colorB
  if minWl == dsC:
    dst[tidx, tidy] = colorC
  if minWl == dsD:
    dst[tidx, tidy] = colorD
```

Main Function

```
# Calculate V matrix
img = root.copy()
devdata = cuda.to_device(img)
devOutput = cuda.device_array((img.shape[0], img.shape[1], 1),np.float64)
blockSize = (8,8)
gridSize = (math.ceil(img.shape[0] / blockSize[0]), math.ceil(img.shape[1] / blockSize[1]))
calV[gridSize, blockSize](devdata, devOutput)
v_mat = devOutput.copy_to_host()
```

```
# Separate R, G, B channels
blue_channel = np.ascontiguousarray(img[:,:,0])
green_channel = np.ascontiguousarray(img[:,:,1])
red_channel = np.ascontiguousarray(img[:,:,2])
```

```
# Calculate new color for each channel
window_size = 3
devdata1 = cuda.to_device(v_mat)
devOutput1 = cuda.device_array((img.shape[0], img.shape[1]),np.float64)

blockSize = (8,8)
gridSize = (math.ceil(img.shape[0] / blockSize[0]), math.ceil(img.shape[1] / blockSize[1]))

rc = cuda.to_device(red_channel)
kuwahara[gridSize, blockSize](devdata1, devOutput1, window_size, rc)
red_mat = devOutput1.copy_to_host()

gc = cuda.to_device(green_channel)
kuwahara[gridSize, blockSize](devdata1, devOutput1, window_size, gc)
green_mat = devOutput1.copy_to_host()

bc = cuda.to_device(blue_channel)
kuwahara[gridSize, blockSize](devdata1, devOutput1, window_size, bc)
blue_mat = devOutput1.copy_to_host()

combine_img = np.dstack((blue_mat,green_mat,red_mat))
imgplot = plt.imshow(img)
plt.show()
imgplot = plt.imshow(combine_img)
plt.show()
```

3 Result

Running the algorithm using GPU takes about 3s while running on CPU takes more than 4 minutes. Window size using w = 4