Seam Carving on PyOpenCL arhitecture

Niko Colnerič

August 26, 2015

Abstract

The aim of this work is to test the ability of parallel computation architectures. We first implement seam carving algorithm for content aware image resizing in pure Python. Next, we implement the crucial parts of the algorithm in PyOpenCL. Doing so, we are able to run it in parallel using either multiple CPU cores or on GPU. The results show up to 53× speed-up in both cases in comparison to single thread Python code.

1 Introduction

Seam carving [1] is an algorithm for content-aware image resizing. In contrast to basic resizing techniques — like squeezing or cropping — it takes into account the content of the image and tries to resize it by removing its less important parts. To define the importance image's region we employ some kind of energy function. The goal of this function is to tell us how important is each pixel of the image. One simple approach is to just take the discrete derivatives of the image like in Eq. 1. This was used throughout the majority of the original paper and has shown to be successful so we also employed this energy function.

$$e_1(\boldsymbol{I}) = \left|\frac{\partial}{\partial x}\boldsymbol{I}\right| + \left|\frac{\partial}{\partial y}\boldsymbol{I}\right| \tag{1}$$

Once the importance of pixels is defined, we resize the image to desired size by iteratively removing *seams*. Seam is an 8-connected path of pixels either from left to right (horizontal seam) or from top to bottom (vertical seam). Examples of energy plot and a vertical seam are depicted in Fig. 1. Seams are calculated with the dynamic programming approach in a top to bottom (and analogously left to right) manner. For each pixel we sum its energy along with the minimal energy of the optimal path to the three pixels above (up, up-and-left, up-and-right). When we traverse the whole image the last row contains the min-cost paths to each of the pixels in the last row. We select the one with the minimal cost and then backtrack from its position to the top of the image to obtain a min-cost seam.

If we are only resizing in one dimension — either by height or width — we do this by iteratively finding one seam and removing it. By repeating this n times, we shrink the image by its height (width) by n pixels. However, when shrinking in both dimensions, the problem of optimal order arises. Since our goal is to preserve as much *energy* in the image as possible and different permutations of seam removal results in different loss of energy, we must find

Figure 1: Original image is shown on left. The right image presents the energy of the image, as defined in Eq. 1, along with one vertical seam in red color.

the optimal order. Again, we employ a dynamic algorithm. We initialize the transportation map $\boldsymbol{T}(0,0) = 0$ and the first row/column with the cost of removing just horizontal/vertical seams. $\boldsymbol{T}(i,j)$ will contain the optimal cost of removing i horizontal and j vertical seams. We fill the transportation map $\boldsymbol{T}(i,j)$ by considering the best of two options: take $\boldsymbol{T}(i-1,j)$ and remove one vertical seam, or take $\boldsymbol{T}(i,j-1)$ and remove one horizontal seam. Backtracking from $\boldsymbol{T}(n,m)$ gives the optimal order of seam removal for shrinking image's height by n pixels and its width by m pixels.

Note that seam carving can also be used for enlarging images, object removal, object amplification or constructing multi-size images. However, we only focus on image shrinking by either one or both dimensions (along with the optimal order of seam removal).

2 Implementation

We first implemented seam carving in pure Python for shrinking the image's height and width along with the optimal order of seam removal. All the source code is attached in appendix A and also available online at `https://github.com/nikicc/seam-carving`. Here we will only discuss the crucial parts that were also implemented in PyOpenCL — that is energy calculation and seam finding — and compare the single thread Python code to the PyOpenCL kernel code.

2.1 Image's energy calculation

To calculate the image's energy we used Eq. 1. Since we processed color images, we sum the derivatives for each of the colors. The crucial part of Python code, presented in Lst. 1, is coming from the method `def get_energy_image(img)`. We loop through each image's pixel, calculate the derivative and store it in the `energy[h, w]`. Note that this code could be written much more efficiently using numpy on per-row basis rather than per-pixel basis. We did not optimize it like this, to prevent numpy from automatically utilizing multiple cores for such calculations.

Listing 1: Python code for energy calculation.

```python
for h in range(height):
    for w in range(width):
        w_l = max(0, w-1)
        w_r = min(w+1, width-1)
        h_u = max(0, h-1)
        h_d = min(h+1, height-1)
        energy[h, w] = sum(abs(img[h_u, w, :] - img[h_d, w, :])) + \
                       sum(abs(img[h, w_l, :] - img[h, w_r, :]))
```

Code from Lst. 1 was rewritten to OpenCL code shown on Lst. 2. Note that the $3D$ image array was first linearized and sent as a $1D$ array. Therefore, we first need to translate the linear index back to x, y and z coordinates. Later we also need the inverse mapping back to linear index which is provided as macro `LINEAR`. In this kernel code we calculate the derivatives for each color separately and we sum them up for all colors afterwards in Python.

Listing 2: OpenCl kernel code for energy calculation.

```
#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))
#define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))
#define ABS(x) ((x)<0 ? -(x) : (x))
#define LINEAR(h, w, d, H, W, D) (h*(W*D) + w*(D) + d)

__kernel void energy(__global float* img, __global float* res,
                     int H, int W, int D)
{
    // get liner index
    int index = get_global_id(0);

    // convert index to 3D coordinates
    int d = index % D;
    int h = index / (W*D);
    int w = (index / D) % W;

    // sanity check
    /*
    int back = LINEAR(h, w, d, H, W, D);
    if(back==index){
        res[index] = -42;
    }
    */

    // "border safe" indices
    int wl = MAX(0, w-1);
    int wr = MIN(w+1, W-1);
    int hu = MAX(0, h-1);
    int hd = MIN(h+1, H-1);

    // derivative value
    float der = 0;

    // left-right
    int bl = LINEAR(h, wl, d, H, W, D);
    int br = LINEAR(h, wr, d, H, W, D);
    der += ABS(img[bl] - img[br]);


```

```
    // up-down
    int bu = LINEAR(hu, w, d, H, W, D);
    int bd = LINEAR(hd, w, d, H, W, D);
    der += ABS(img[bu] - img[bd]);

    res[index] = der;
}
```

2.2 Finding optimal seams

Once we fill the `energy` matrix, we can use it to find optimal min-cost seams. Python version of this is presented in Lst. 3 and comes from method `def find_seam_vertical(energy)`. We use array `M` to store costs of optimal paths to each of the pixels and `backtrace` for backtracking the min-cost seam.

Listing 3: Python code for finding vertical seams. Code for horizontal seams is analogous.

```
for h in range(1, height):
    for w in range(width):
        w_l = max(0, w-1)
        w_r = min(w+2, width)

        ind, val = min_argmin(M[h-1, w_l:w_r])
        M[h, w] = energy[h, w] + val
        backtrace[h, w] += ind - 1
```

The code from Lst. 3 was rewritten to OpenCL kernel code presented in Lst. 4. We calculate this on per-row basis, meaning that we sent one row for execution to OpenCL. Then get back results and execute the next row. This could be implemented much more efficiently using synchronization methods.

Listing 4: OpenCl kernel code for finding optimal seams.

```
__kernel void find_seam(__global float* m, __global float* energy,
                        __global float* new_m, __global float* back, int W)
{
    // get index
    int w = get_global_id(0);

    // "border safe" indices
    int wl = MAX(0, w-1);
    int wr = MIN(w+1, W-1);

    // enlarge the path by one
    float val = INFINITY;
    int ind;
    for(int i = 0; (i+wl) <= wr; i = i+1){
        if(m[i + wl] < val){
            val = m[i + wl];
            ind = i;
        }
    }

    // store energy
    new_m[w] = energy[w] + val;


```

```
25      // store backtrack pointers
26      // special case for first element, since there is no "left"
27      if(w == 0){
28          ind += 1;
29      }
30      back[w] = ind-1;
31  }
```

3 Results

We presents the time measurements of out Python and PyOpenCL implementations. All following results are obtained on a series of one image in different sizes — from 32×20 up to 1024×640 pixels. Experiment were run on a computer containing Intel Core i7 processor (4 cores, 8 threads) and Intel Iris Pro integrated GPU. We measured the total time to remove 25 vertical seams and we report on the average times per seam. Fig. 2 shows total times for removing a seam, Fig. 3 and Fig. 4 show the times for calculating energy and finding a seam respectively. All plots are in logarithmic scale. We also show exact times per seam in Tab. 1.

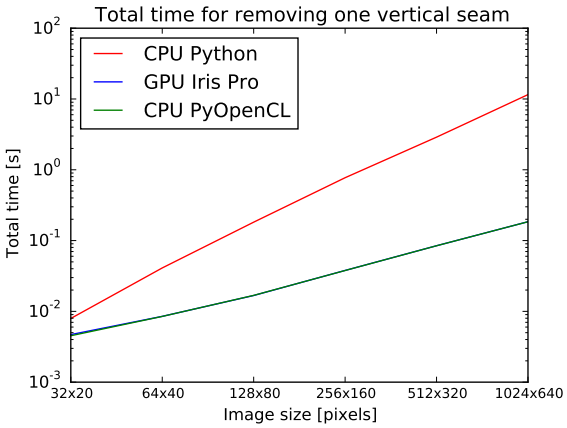

Figure 2: Total times for removing one seam.

Size [pixels]	Python [s]	PyOpenCL CPU [s]	Speedup
32x20	0.007	0.004	1
64x40	0.035	0.010	3
128x80	0.155	0.021	7
256x160	0.695	0.043	16
512x320	2.727	0.093	29
1024x640	10.243	0.191	53

Table 1: Total times for removing one vertical seam on CPU with pure Python vs. PyOpenCL.

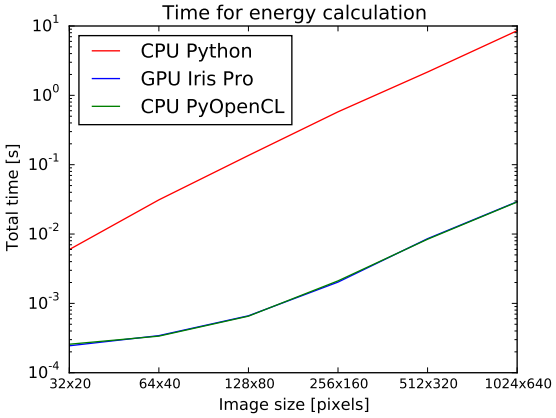

Figure 3: Times for calculating image's energy.

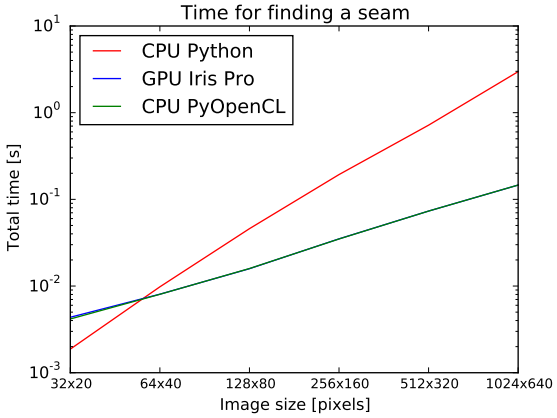

Figure 4: Times for finding one vertical seam.

Using PyOpenCL we were able to significantly speedup out algorithm — even up to 53× for image with size 1024×640. The time for removing one seam was brought down from $10s$ to 0.2s. Interestingly, running PyOpenCL implementation all 8 CPU threads requires almost exactly the same time as running it on integrated GPU. Most importantly, Fig. 2 shows that the time spent grows more slowly with PyOpenCL that pure Python implementations, which is indicated by a smaller slope.

4 Usage

Fig. 5 shows one source image re-targeted to different sizes. We show images shank by $100px$ in height, width, and height & width. Image is more suitable for shrinking by width, due to the sky. When shrinking by height, we introduce noticeable irregularities since sky pixels on same height have different colors. Note that important parts — house, bench, pavement and trees — remain more or less the same on all images. The majority of reduction in done in the sky and grass and is therefore almost unnoticeable, especially when only reducing the width.

Figure 5: Original image ($1024px \times 640px$) is shown on top left position. Top right is shrank by height for $100px$. Bottom left is shrank by width for $100px$. Bottom right is shrank by height and width, each for $100px$.

5 Conclusion

We implemented seam carving algorithm first in pure Python and then also in PyOpenCL. The beauty of PyOpenCL implementation is that it can be run on heterogeneous systems; that is, out code can be run either on CPU or GPU. Our main goal was to test whether we can speed-up seam carving algorithm on parallel architectures. Results have shown that using either multiple CPU cores or using GPU we are able to achieve significant speed-ups, even up to 53×. Note, however, that the GPU we were using — Intel Iris Pro — is and integrated one and not a dedicated one. We presume the speed-up could be even larger on a capable dedicated GPU.

References

[1] Avidan, S., and Shamir, A. Seam carving for content-aware image resizing. *ACM Transactions on Graphics 26* (2007), 10.

Appendices

A Source code

We list here all the source code. File *get_energy.cl* contains the OpenCL kernel code, file *seam-carve.py* contains the pure Python code along with the PyOpenCL code that calls *get_energy.cl*. All this code is also available online at https://github.com/nikicc/seam-carving.

../get_energy.cl

```
#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))
#define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))
#define ABS(x) ((x)<0 ? -(x) : (x))
#define LINEAR(h, w, d, H, W, D) (h*(W*D) + w*(D) + d)


__kernel void energy(__global float* img, __global float* res,
                     int H, int W, int D)
{
    // get liner index
    int index = get_global_id(0);

    // convert index to 3D coordinates
    int d = index % D;
    int h = index / (W*D);
    int w = (index / D) % W;

    // sanity check
    /*
    int back = LINEAR(h, w, d, H, W, D);
    if(back==index){
        res[index] = -42;
    }
    */

```

```
    // "border safe" indices
    int wl = MAX(0, w-1);
    int wr = MIN(w+1, W-1);
    int hu = MAX(0, h-1);
    int hd = MIN(h+1, H-1);

    // derivative value
    float der = 0;

    // left-right
    int bl = LINEAR(h, wl, d, H, W, D);
    int br = LINEAR(h, wr, d, H, W, D);
    der += ABS(img[bl] - img[br]);

    // up-down
    int bu = LINEAR(hu, w, d, H, W, D);
    int bd = LINEAR(hd, w, d, H, W, D);
    der += ABS(img[bu] - img[bd]);

    res[index] = der;
}

__kernel void find_seam(__global float* m, __global float* energy,
                        __global float* new_m, __global float* back, int W)
{
    // get index
    int w = get_global_id(0);

    // "border safe" indices
    int wl = MAX(0, w-1);
    int wr = MIN(w+1, W-1);

    // enlarge the path by one
    float val = INFINITY;
    int ind;
    for(int i = 0; (i+wl) <= wr; i = i+1){
        if(m[i + wl] < val){
            val = m[i + wl];
            ind = i;
        }
    }

    // store energy
    new_m[w] = energy[w] + val;

    // store backtrack pointers
    // special case for first element, since there is no "left"
    if(w == 0){
        ind += 1;
    }
    back[w] = ind-1;
}
```

../seamcarve.py

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from operator import itemgetter
import pyopencl as cl
import numpy as np
import time
```

```
import os


def min_argmin(l):
    """
        Returns the minimal element and its index.

    :param l: List
    :return: index, min value
    """
    ind, val = min(enumerate(l), key=itemgetter(1))
    return ind, val


def show_image(num, img, title, height, width, pth=None):
    """
        Plot the image.

    :param num: Image number
    :param img: Image
    :param title: Title
    :param height: height
    :param width: width
    :param pth: The seam path
    :return: None
    """
    if img is not None:
        plt.figure(num)
        plt.clf()
        plt.imshow(img)
        if pth is not None:
            plt.plot([i[1] for i in pth], [i[0] for i in pth], 'r-')
        plt.title(title)
        plt.xlim([0, width])
        plt.ylim([height, 0])


def get_energy_image(img):
    """
        Calculate the energy function as the discrete derivative of the image.

    :param img: Input image
    :return: The discrete derivatives for each pixel.
    """
    t = time.time()

    # select between PyOpenCL and single thread python implementation
    if USE_PYOPENCL:
        energy = opencl.get_energy(img)
    else:
        height, width = img.shape[:2]
        energy = np.zeros((height, width))

        for h in range(height):
            for w in range(width):
                w_l = max(0, w-1)
                w_r = min(w+1, width-1)
                h_u = max(0, h-1)
                h_d = min(h+1, height-1)
                energy[h, w] = sum(abs(img[h_u, w, :] - img[h_d, w, :])) + \
```

```
                                sum(abs(img[h, w_l, :] - img[h, w_r, :]))

    ENERGY_CALC_TIMES.append(time.time()-t)
    return energy


def find_seam_vertical(energy):
    """
        Back track ideology:
            -1   0    1
              \  |  /
                [ ]

    :param energy: Energy plot
    :return: List of seam coordinates (y, x) pairs.
    """
    t = time.time()

    height, width = energy.shape
    M = np.zeros(energy.shape, dtype=np.float32)
    M[0, :] = energy[0, :]
    backtrace = np.zeros(energy.shape, dtype=int)

    # select between PyOpenCL and single thread python implementation
    if USE_PYOPENCL:
        for h in range(1, height):
            M[h, :], backtrace[h, :] = opencl.find_seam(M[h-1, :], energy[h, :])
    else:
        backtrace[:, 0] = 1     # since first column has no element to the left
        for h in range(1, height):
            for w in range(width):
                w_l = max(0, w-1)
                w_r = min(w+2, width)

                ind, val = min_argmin(M[h-1, w_l:w_r])
                M[h, w] = energy[h, w] + val
                backtrace[h, w] += ind - 1

    # backtrack to get the path
    ind, _ = min_argmin(M[-1, :])
    seam = [ind]
    for h in range(height-1, 0, -1):
        ind += backtrace[h, ind]
        seam.append(ind)
    seam.reverse()                  # order from top to bottom
    seam = list(enumerate(seam))    # transform to coordinates

    SEAM_SEARCH_TIMES.append(time.time()-t)
    return seam


def find_seam_horizontal(energy):
    """
        Back track ideology:
            -1
              \
            0 - []
              /
            1

```

```
    :param energy: Energy plot
    :return: List of seam coordinates (y, x) pairs.
    """
    t = time.time()

    height, width = energy.shape
    M = np.zeros(energy.shape)
    M[:, 0] = energy[:, 0]
    backtrace = np.zeros(energy.shape, dtype=int)

    # select between PyOpenCL and single thread python implementation
    if USE_PYOPENCL:
        for w in range(1, width):
            M[:, w], backtrace[:, w] = opencl.find_seam(M[:, w-1], energy[:, w])
    else:
        backtrace[0, :] = 1     # since first column does not have element up
        for w in range(1, width):
            for h in range(height):
                h_u = max(0, h-1)
                h_d = min(h+2, height)

                ind, val = min_argmin(M[h_u:h_d, w-1])
                M[h, w] = energy[h, w] + val
                backtrace[h, w] += ind - 1

    # backtrack to get the path
    ind, _ = min_argmin(M[:, -1])
    seam = [ind]
    for w in range(width-1, 0, -1):
        ind += backtrace[ind, w]
        seam.append(ind)
    seam.reverse()                                      # order from top to bottom
    seam = [(y, x) for x, y in enumerate(seam)]     # transform to coordinates

    SEAM_SEARCH_TIMES.append(time.time()-t)
    return seam


def remove_one_seam_from_image(img, trace, horizontal):
    """
        Remove the seam from the image.

    :param img: Image
    :param trace: Seam as a list of (y, x) coordinates.
    :param horizontal: Whether we remove horizontal or vertical seam.
    :return: Smaller image, the cost of reduction.
    """
    # dirty hack, since reshaping does only work this way,
    # otherwise the image is corrupted
    if horizontal:
        img = img.swapaxes(0, 1)
        trace = [(x, y) for y, x in trace]

    height, width, depth = img.shape
    mask = np.ones(img.shape, dtype=bool)
    cost = np.sum(img)

    # create the mask
    for h, w in trace:
        mask[h, w, :] = False
```

```
187
188     # remove the pixels on the seam
189     img = img[mask].reshape(height, width-1, depth)
190     cost -= np.sum(img)
191
192     # flip back for horizontal
193     if horizontal:
194         img = img.swapaxes(0, 1)
195
196     return img, cost
197
198
199 def shrink_height(img):
200     """
201         Find the best horizontal seam to be removed from the image
202         and return image without that seam.
203
204     :param img: Input image
205     :return: Image with one pixel less height.
206     """
207     eng = get_energy_image(img)
208     path = find_seam_horizontal(eng)
209     img, cost = remove_one_seam_from_image(img, path, horizontal=True)
210     return img, eng, path, cost
211
212
213 def shrink_width(img):
214     """
215         Find the best vertical seam to be removed from the image
216         and return image without that seam.
217
218     :param img: Input image
219     :return: Image with one pixel less width.
220     """
221     eng = get_energy_image(img)
222     path = find_seam_vertical(eng)
223     img, cost = remove_one_seam_from_image(img, path, horizontal=False)
224     return img, eng, path, cost
225
226
227 def seam_carve(img, dw=0, dh=0):
228     """
229         Shrink image with seam-carving to desired size.
230
231     :param img: Image
232     :param dw: Shrink width for dw pixels
233     :param dh: Shrink height for dh pixels
234     :return: Smaller image
235     """
236     eng, path = None, None
237     progress = 0
238     final = 0
239
240     def update_progress():
241         nonlocal progress
242         progress += 1
243         print("\rProgress {:5.2f}%".format(100*progress/final), end='')
244         if progress == final:
245             print("\r", end='')
246
```

```
    if dh == 0 and dw == 0:
        return img, None, None

    elif dw == 0:   # remove just horizontal seams
        final = dh
        for i in range(dh):
            img, eng, path, _ = shrink_height(img)
            update_progress()
        return img, eng, path

    elif dh == 0:   # remove just vertical seams
        final = dw
        for i in range(dw):
            img, eng, path, _ = shrink_width(img)
            update_progress()
        return img, eng, path

    else:           # remove both
        final = 2*dw*dh + dh + dw
        REMOVE_HORIZONTAL = '^'
        REMOVE_VERTICAL = '<'

        # init
        img_map = np.zeros((dh+1, dw+1), dtype=object)      # store images
        backtrace = np.zeros((dh+1, dw+1), dtype=object)    # backtrack order
        T = np.zeros((dh+1, dw+1))                          # optimal cost

        # store the initial image
        img_map[0, 0] = img

        # fill horizontal border
        for i in range(1, dh+1):
            current_img = np.copy(img_map[i-1, 0])
            current_img, eng, path, cost = shrink_height(current_img)
            T[i, 0] = T[i-1, 0] + cost
            img_map[i, 0] = current_img
            backtrace[i, 0] = REMOVE_HORIZONTAL
            update_progress()

        # fill vertical border
        for i in range(1, dw+1):
            current_img = np.copy(img_map[0, i-1])
            current_img, eng, path, cost = shrink_width(current_img)
            T[0, i] = T[0, i-1] + cost
            img_map[0, i] = current_img
            backtrace[0, i] = REMOVE_VERTICAL
            update_progress()

        # use dynamic programing to find the best order
        for ih in range(1, dh+1):
            for iw in range(1, dw+1):
                # option 1: remove one horizontal seam from image above
                current_H = np.copy(img_map[ih-1, iw])
                current_H, eng_H, path_H, cost_H = shrink_height(current_H)
                cost_H += T[ih-1, iw]
                update_progress()

                # option 2: remove one vertical seam from image left
                current_V = np.copy(img_map[ih, iw-1])
                current_V, eng_V, path_V, cost_V = shrink_width(current_V)
```

```
                cost_V += T[ih, iw-1]
                update_progress()

                # select the optimal option
                if cost_H < cost_V:
                    T[ih, iw] = cost_H
                    img_map[ih, iw] = current_H
                    backtrace[ih, iw] = REMOVE_HORIZONTAL
                    eng, path = eng_H, path_H
                else:
                    T[ih, iw] = cost_V
                    img_map[ih, iw] = current_V
                    backtrace[ih, iw] = REMOVE_VERTICAL
                    eng, path = eng_V, path_V

            # clear previous line of images
            img_map[ih-1, :] = None
        return img_map[-1, -1], eng, path


class PyOpenCLDriver:
    def __init__(self):
        self.ctx = cl.create_some_context()
        self.queue = cl.CommandQueue(self.ctx)
        self.program = None

    def load_program(self, filename):
        f = open(filename, 'r')
        f_str = "".join(f.readlines())
        self.program = cl.Program(self.ctx, f_str).build()

    def get_energy(self, img):
        mf = cl.mem_flags

        H, W, D = map(np.int32, img.shape)
        img = img.astype(np.float32).reshape(-1)
        res = np.empty_like(img)

        img_buf = cl.Buffer(self.ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
                            hostbuf=img)
        res_buf = cl.Buffer(self.ctx, mf.WRITE_ONLY, res.nbytes)

        self.program.energy(self.queue, img.shape, None,
                            img_buf, res_buf, H, W, D)
        cl.enqueue_read_buffer(self.queue, res_buf, res).wait()

        res = res.reshape((H, W, D))
        return np.sum(res, axis=2)

    def find_seam(self, m, energy):
        mf = cl.mem_flags

        W = np.int32(m.shape[0])
        m = m.astype(np.float32)
        energy = energy.astype(np.float32)
        new_m = np.empty_like(m)
        back = np.empty_like(m)

        m_buf = cl.Buffer(self.ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=m)
        energy_buf = cl.Buffer(self.ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
```

```
                                hostbuf=energy)
        new_m_buf = cl.Buffer(self.ctx, mf.WRITE_ONLY, new_m.nbytes)
        back_buf = cl.Buffer(self.ctx, mf.WRITE_ONLY, back.nbytes)

        self.program.find_seam(self.queue, m.shape, None,
                               m_buf, energy_buf, new_m_buf, back_buf, W)
        cl.enqueue_read_buffer(self.queue, new_m_buf, new_m).wait()
        cl.enqueue_read_buffer(self.queue, back_buf, back).wait()
        return new_m, back


#
#   Settings
#
USE_PYOPENCL = True      # If False, use python, else use PyOpenCL
PLOT_RESULTS = True      # If True the results are also shown
ENERGY_CALC_TIMES = []   # For time measurements
SEAM_SEARCH_TIMES = []   # For time measurements

opencl = PyOpenCLDriver()
opencl.load_program("get_energy.cl")
os.environ["PYOPENCL_CTX"] = "0:0"  # Select the device on which to run OpenCL
os.environ["PYOPENCL_COMPILER_OUTPUT"] = "1"
print("Using: {}".format("PyOpenCL" if USE_PYOPENCL else "single CPU"))


if __name__ == "__main__":
    image = mpimg.imread(os.path.join('img', 'nature_1024.png'))
    original = np.copy(image)

    # delta height, delta width
    DW = 50
    DH = 0

    t0 = time.time()
    image, energy, path = seam_carve(image, dw=DW, dh=DH)
    t1 = time.time()

    print("Image shape:\n\t- original:\t\t{}\n\t- seam-carved:\t{}".format(
        original.shape, image.shape))
    print("Times:\n"
          "\t- one energy calc:\t{:.4f}s [avg of {}]\n"
          "\t- one seam search:\t{:.4f}s [avg of {}]\n"
          "\t- total:\t\t\t{:.4f}s".format(
        sum(ENERGY_CALC_TIMES)/len(ENERGY_CALC_TIMES),
        len(ENERGY_CALC_TIMES),
        sum(SEAM_SEARCH_TIMES)/len(SEAM_SEARCH_TIMES),
        len(SEAM_SEARCH_TIMES),
        t1-t0))

    # save energy plot
    #c = np.max(energy)+1
    #for y, x in path:
    #    energy[y, x-1:x+2] = c
    #mpimg.imsave(os.path.join('results', 'energy-plot.png'), energy)

    # save smaller image, fill removed parts with white
    #white = np.ones(image.shape)[:, :DW, :]
    #image = np.hstack((image, white))
    #white = np.ones(image.shape)[:DH, :, :]
```

```
427     #image = np.vstack((image, white))
428     #mpimg.imsave(os.path.join('results', 'nature_1024-{}w-{}h.png'.
429     #                          format(DW, DH)), image)
430
431     # plot
432     if PLOT_RESULTS:
433         H, W = original.shape[:2]
434         show_image(1, original, "Original", H, W)
435         show_image(2, energy, "Last seam on energy plot", H, W, path)
436         show_image(3, image, "Seam carving", H, W)
437         plt.show()
```