Engineering: Parallel Implementation of Seam Carving

8

Chapter Summary
In this chapter we present the parallelization of Seam Carving - a content-aware image resizing algorithm. Effective parallelization of seam carving is a challenging problem due to its complex computation model. Two main reasons prevent effective parallelization: computation dependence and irregular memory access. We show, which parts of the original seam carving algorithm can be accelerated on GPU and which parts cannot, and how this affects the overall performance.

Seam carving is a content-aware image resizing technique where the image is reduced in size by one pixel of width (or height) at a time. Seam carving attempts to reduce the size of a picture while preserving the most interesting content of the image. Seam Carving was originally published in a 2007 paper by Shai Avidan and Ariel Shamir. Ideally, one would remove the "lowest energy" pixels (where energy means the amount of important information contained in a pixel) from the image to preserve the most important features. However, that would create artifacts and not preserve the rectangular shape of the image. To balance removing low-energy pixels while minimizing artifacts, we remove exactly one pixel in each row (or column) where every pixel in the row must touch the pixel in the next row either via an edge or corner. Such a connected path of pixels is called seam. If we are going to resize the image horizontally, we need to remove one pixel from each row of the image. Our goal is to find a path of connected pixels from the bottom of the image to the top. By "connected", we mean that we will never jump more than one pixel left or right as we move up the image from row to row. A **vertical seam** in an image is a path of pixels connected from the top to the bottom with one pixel in each row. Each row has exactly only one pixel which is the part of the vertical seam. By removing the vertical seams iteratively, we can compress the image in the horizontal direction. The seam carving method produces a resized image by searching for the seam which has the lowest user-specified "image energy". To shrink the image, the lowest energy

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_8

seam is removed and the process is repeated until the desired dimensions are reached. Seam Carving is a three-step process:

1. Assign an energy value to every pixel. This will define the important parts of the image that we want to preserve.
2. Find an 8-connected path of the pixels with the least energy. We use dynamic programming to calculate the costs of every potential path through the image.
3. Follow the cheapest path to remove one pixel from each row or column to resize the image.

Following these steps will shrink the image by one pixel. We can repeat the process as many times as we want to resize the image as much as necessary. What we need to is to implement a function which takes an image as an input and produce a resized image in one dimension or two dimensions as an output which is expected by the users.

Why is seam carving interesting for us? Effective parallelization of seam carving is a challenging problem due to its complex computation model. There are two main reasons why effective parallelization is prevented: (1) Computation dependence: dynamic programming is a key step to compute an optimal seam during image resizing and takes a large fraction of the program execution time. It is very hard to parallelize the dynamic programming on GPU devices due to the computation dependency. (2) Intensive and irregular memory access: in order to compute various intermediate results a large number of irregular memory access patterns is required. This worsens the program performance significantly. In this chapter, we are not going to find or present a better algorithm for seam carving that can be parallelized. We are just going to show, which part of the original seam carving algorithm can

Fig. 8.1 Original *cyclist* image to be made narrower with seam carving

be accelerated on GPU and which parts cannot and how this affects the overall performance. For illustration purposes, we will use the *cyclist* image from Fig. 8.1 as an input image, which is to be made narrower with seam carving.

8.1 Energy Calculation

What are the most important parts of an image? Which parts of a given image should we eliminate first when resizing and which should we hold onto the longest? The answer to these questions lies in the energy value of each pixel. The energy value of a pixel is the amount of important information contained in that pixel. So, the first step is to compute the energy value for every pixel, which is a measure of its importance—the higher the energy, the less likely that the pixel will be included as part of a seam and eventually removed.

The simplest and frequently most-effective measure of energy is the gradient of the image. An image gradient highlights the parts of the original image that are changing the most. It is calculated by looking at how similar each pixel is to its neighbors. Large uniform areas of the image will have a low gradient (i.e., energy) value and the more interesting areas (edges of objects or places with a lot of detail) will have a high energy value. There exist a variety of energy measures (e.g., Sobel operator). In this book, which is not primarily devoted to image processing, we will use a very simple energy function, although a number of other energy functions exist and may work better. Let each pixel (i, j) in the image has its color denoted as $I(i, j)$. The energy of the pixel (i, j) is given by the following equation:

$$E(i, j) = |I(i, j) - I(i, j+1)| + |I(i, j) - I(i+1, j)| + |I(i, j) - I(i+1, j+1)| \tag{8.1}$$

A sequential algorithm for pixel energy calculation is given in Algorithm 5.

Algorithm 5 SEAM CARVING: ENERGY CALCULATION

Input: I - RxC image

1: **for** $i = 1 \ldots R$ **do**
2: **for** $j = 1 \ldots C$ **do**
3: $E(i, j) = |I(i, j) - I(i, j+1)| + |I(i, j) - I(i+1, j)| + |I(i, j) - I(i+1, j+1)|$
4: **end for**
5: **end for**

Output: E - RxC energy map

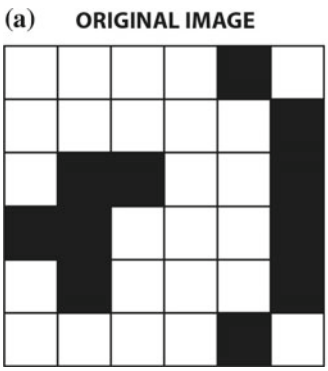

Fig. 8.2 **a**) Original black and white image. **b**) Energies of the pixels in the image

We will illustrate this step on a simple example. Let us suppose a black and white image as in Fig. 8.2a and let us suppose that the color of black pixels is coded with the value 1, and the color of white pixels is coded with 0. Figure 8.2b shows energy map for the image from Fig. 8.2a. Energies of the pixels in the last column and the last row are computed with the assumption that the image is zero-padded. For example, the energy of the first pixel in the fourth row (black pixel) is according to Eq. 8.1:

$$E(3,0) = |1-1| + |1-0| + |1-1| = 1,$$

and the energy of the last pixel in the fifth row is:

$$E(4,5) = |1-0| + |1-0| + |1-0| = 3.$$

The resulting *cyclist* image of this step is shown in Fig. 8.3. We can see that large uniform areas of the image have a low gradient value (black) and the more interesting edges of objects have a high gradient value (white).

8.2 Seam Identification

Now that we have calculated the value of each pixel, our next objective is to find a path from the bottom of the image to the top of the image with the least energy. The line must be 8-connected: this means that every pixel in the row must be touched by the pixel in the next row either via an edge or corner. That would be a vertical seam of minimum total energy. One way to do this would be to simply calculate the costs of each possible path through the image one-by-one. Start with a single pixel in the bottom row, navigate every path from there to the top of the image, keep track of the cost of each path as you go. But we will end up with thousands or millions of

Fig. 8.3 The calculated energy function of the *cyclist* image

possible paths. To overcome this, we can apply the dynamic programming method as described in the paper by Avidan and Shamir. Dynamic programming lets us touch each pixel in the image only once, aggregating the total cost as we go, in order to calculate the final cost of an individual path. Once we have found the energy of every pixel, we start at the bottom of the image and go up row by row, setting each element in the row to the energy of the corresponding pixel plus the minimum energy of the 3 possibly path pixels "below" (the pixel directly below and the lower left and right diagonal). Thus, we have to traverse the image from the bottom row to the first row and compute the cumulative minimum energy M for all possible connected seams for each pixel (i, j):

$$M(i, j) = E(i, j) + min(M(i + 1, j - 1), M(i + 1, j), M(i + 1, j + 1)) \quad (8.2)$$

In the bottom most row cumulative energy is equal to pixel energy, i.e., $M(i, j) = E(i, j)$. A sequential algorithm for cumulative energy calculation is given in Algorithm 6.

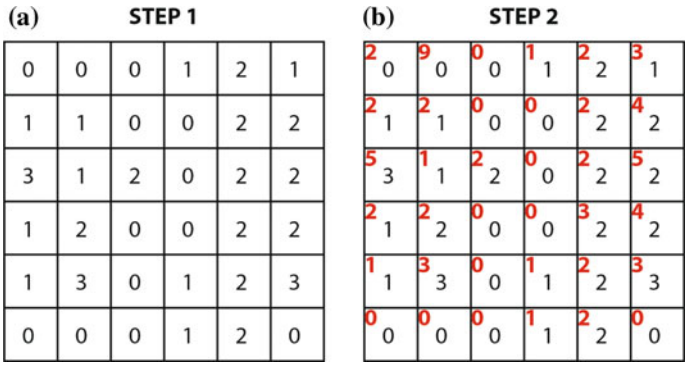

Fig. 8.4 **a**) Energies of the pixels. **b**) Cumulative energies

Algorithm 6 SEAM CARVING: CUMULATIVE ENERGY CALCULATION

Input: E - RxC energy map

1: **for** $j = 1 \ldots C$ **do**
2: $M(R, j) = E(R, j)$
3: **end for**
4: **for** $i = R - 1 \ldots 1$ **do**
5: **for** $j = 1 \ldots C$ **do**
6: $M(i, j) = E(i, j) + min(M(i + 1, j - 1), M(i + 1, j), M(i + 1, j + 1))$
7: **end for**
8: **end for**

Output: M - RxC cumulative energy map

We will illustrate this step with Fig. 8.4. We start with the last (bottom most) row. Cumulative energies in that row are the same as pixel energies. Then, we move up to the fifth row. Cumulative energy of the fifth pixel in the fifth row is the sum of its energy and the minimal energy of three pixels below it:

$$M(4, 4) = 2 + min(1, 2, 0) = 2.$$

On the other hand, cumulative energy of the last pixel in the fifth row is

$$M(4, 5) = 3 + min(2, 0, \infty) = 3.$$

As the element (4, 6) does not exist, we assume it has the maximal energy. In other words, we ignore it. Once we have computed all the values M, we simply find the lowest value of M in the top row and return the corresponding path as our minimum energy vertical seam.

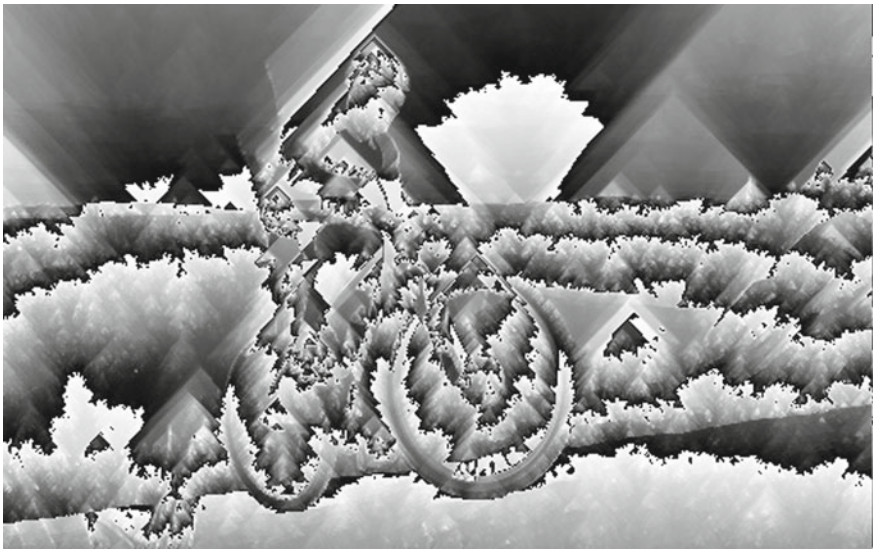

Fig. 8.5 The calculated cumulative energy function of the *cyclist* image. Please note that due to summation, almost all pixel values are greater than 255 and are represented only with least significant eight bits in the image

This effect of this step is easy to see in Fig. 8.5. Notice how the spots where the gradient image was brightest are now the roots of inverted triangles of brightness as the cost of those pixels propagate into all of the pixels within the range of the widest possible paths upwards. For example, the brightest inverted triangle at the center of the image (in the form of the cyclist) is created because the white edge at horizon propagates upwards. When we arrive at the top row, the lowest valued pixel will necessarily be the root of the cheapest path through the image. Now, we are ready to start removing seams.

8.3 Seam Labeling and Removal

The final step is to remove all of the pixels along the vertical seam. Due to the power of dynamic programming, the process of actually removing seams is quite easy. All we have to do to calculate the cheapest seam is to start with the lowest value M in the top row and work our way up from there, selecting the cheapest of the three adjacent pixels in the row below. Dynamic programming guarantees that the pixel with the lowest value M will be the root of the cheapest connected path from there. Once we have selected which pixels we want to remove, all that we have to do is go through and copy the remaining pixels on the right side of the seam from right to left and the

image will be one pixel narrower. A sequential algorithm for seam removal is given in Algorithm 7.

Algorithm 7 SEAM CARVING: SEAM REMOVAL

Input: M - RxC cumulative energy map
Input: I - RxC original image

1: $min = M(1, 1)$
2: $col = 1$
3: **for** $j = 2 \ldots C$ **do**
4: **if** $M(1, j) < min$ **then**
5: $min = M(1, j)$
6: $col = j$
7: **end if**
8: **end for**
9: **for** $i = 1 \ldots R$ **do**
10: **for** $j = col \ldots C$ **do**
11: $I(i, j) = I(i, j + 1)$
12: **end for**
13: **if** $M(i + 1, col - 1) < M(i + 1, col)$ **then**
14: $col = col - 1$
15: **end if**
16: **if** $M(i + 1, col + 1) < M(i + 1, col - 1)$ **then**
17: $col = col + 1$
18: **end if**
19: **end for**

Output: I - RxC resized image

We will illustrate this step with Fig. 8.6. We start with the top most row and find the pixel with the smallest value M. In our case, this is the third pixel in the

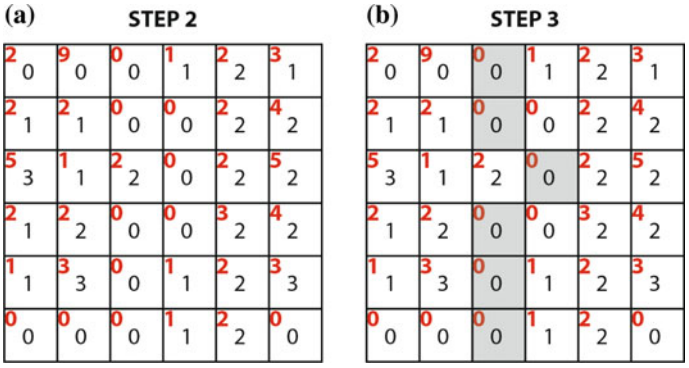

Fig. 8.6 **a**) Cumulative energies. **b**) The seam (in grey) with the minimal energy

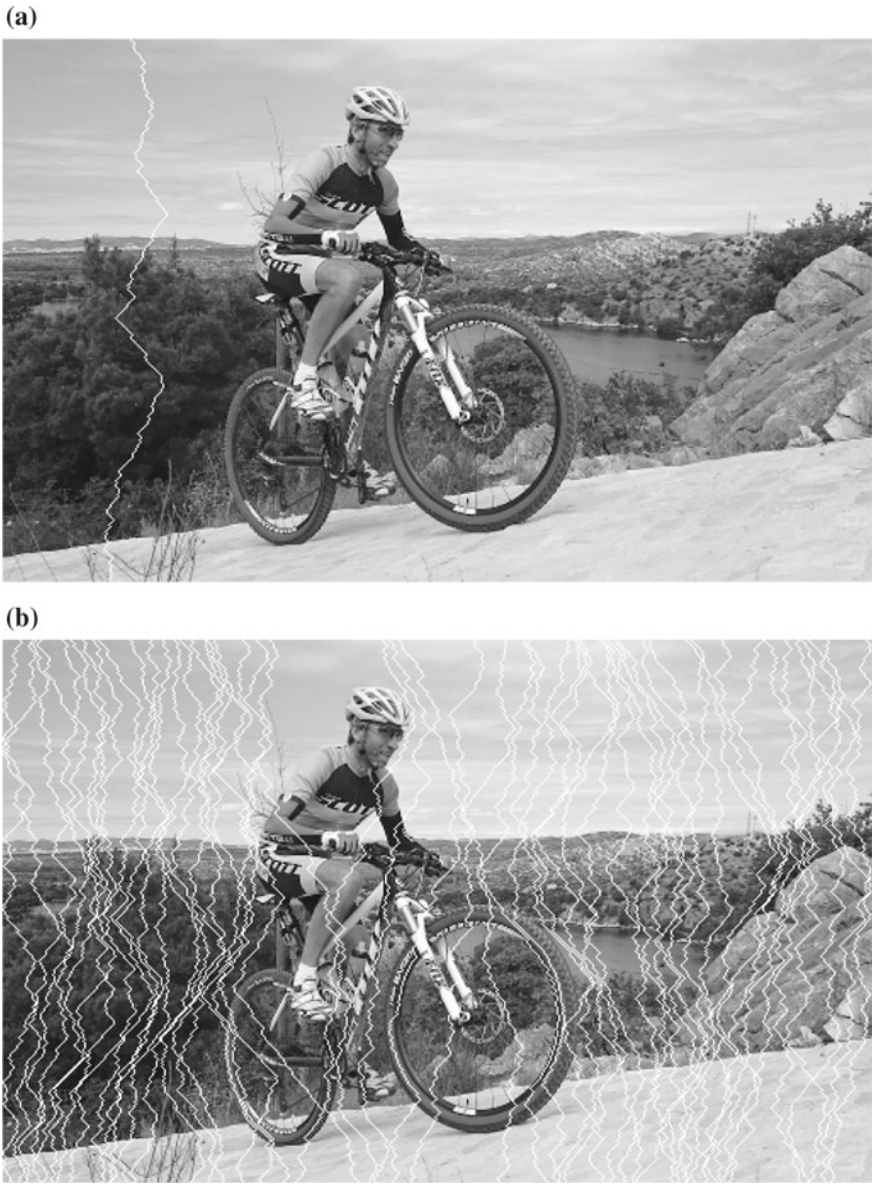

Fig. 8.7 **a**) The first seam in the *cyclist* image. **b**) The first 50 seams in the *cyclist* image

first row with $M(0, 2) = 0$. Then, we select the pixel below that one with the minimal M. In our case, this is the third pixel in the second row with $M(1, 2) = 0$. We continue downwards and select the pixel below the current with the minimal cumulative energy. This is the fourth pixel in the third row with $M(2, 3) = 0$. We

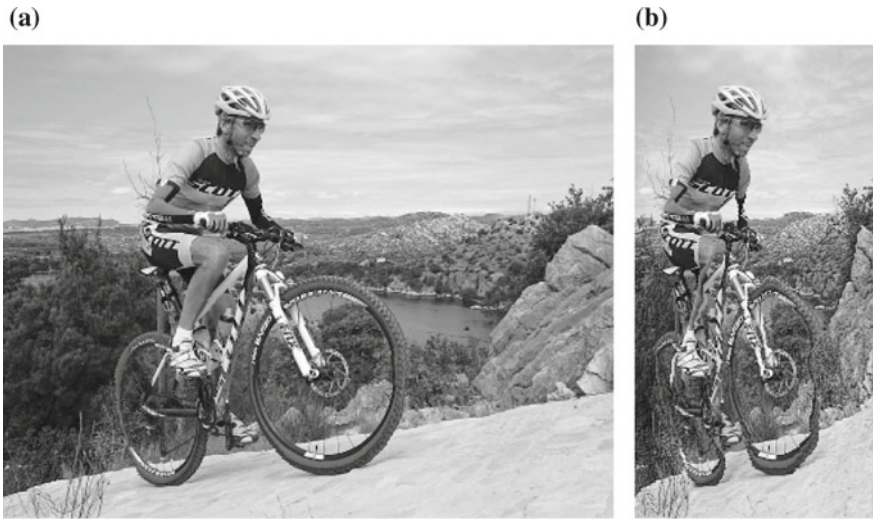

Fig. 8.8 **a**) The image resized by removing 100 seams. **b**) The image resized by removing 350 seams

continue this process until the last row. The seam with minimal energy is depicted in gray in Fig. 8.6b.

Figure 8.7 shows the labeled (with white pixels) seams in the *cyclist* image. The very first vertical seam found in the *cyclist* image is depicted in Fig. 8.7a. It goes through the darkest parts of the energy map form Fig. 8.3 and thus through the pixels with minimal amount of information. Figure 8.7b depicts the first 50 seams found in the *cyclist* image. It can be observed how seams are "avoiding" the regions with the highest pixel energy and thus the highest amount of information.

Once we have labeled a vertical seam we go through the image and move the pixels that are located at the right of the vertical seam from right to left. The new image would be one pixel narrower than the original. We repeat the whole process for as many seams as we like to remove. Figure 8.8 shows two *cyclist* images reduced in size by (a) 100 pixels and (b) 350 pixels.

8.4 Seam Carving on GPU

In this section, we present and compare the implementations of seam carving on CPU and GPU.

8.4.1 Seam Carving on CPU

We will first present the CPU code for seam carving. Emphasis will be only on the functions that implement the main operations of the seam carving algorithm. Other helper functions and code are available on the book's companion site.

Energy Calculation on CPU
Listing 8.1 shows how to calculate pixel energy following Algorithm 5. The function `simpleEnergyCPU` reads input PGM image `input`, calculates the energy for every pixel and writes the energy to the corresponding pixel in PGM image `output`.

```
void simpleEnergyCPU(PGMData *input, PGMData *output, int new_width)
{
    int i, j;
    int diffx, diffy, diffxy;
    int tempPixel;

    for(i=0; i<(input->height); i++)
        for(j=0; j<new_width; j++)
        {
            diffx = abs(getPixelCPU(input, i, j) -
                        getPixelCPU(input, i, j+1));
            diffy = abs(getPixelCPU(input, i, j) -
                        getPixelCPU(input, i+1, j));
            diffxy = abs(getPixelCPU(input, i, j) -
                         getPixelCPU(input, i+1, j+1));

            tempPixel = diffx + diffy + diffxy;
            if( tempPixel>255 )
                output->image[i*output->width+j] = 255;
            else
                output->image[i*(output->width)+j] = tempPixel;
        }
}
```

Listing 8.1 Compute pixel energy

The function Listing 8.1 implements image gradient, which highlights the parts of the original image that are changing the most. The image gradient is calculated using Eq. 8.1, which looks at how similar each pixel is to its neighbors. The third argument `new_width` keeps track of the current image width.

Cumulative Energies on CPU
Now that we have calculated the energy value of each pixel, our goal is to find a path of connected pixels from the bottom of the image to the top. As we previously said, we are looking for a very specific path: the one who's pixels have the lowest total value. In other words, we want to find the path of connected pixels from the bottom to the top of the image that touches the darkest pixels in our gradient image. Listing 8.2 shows how to calculate cumulative pixel energy following Algorithm 6. The function `cumulativeEnergiesCPU` reads input PGM image `input`, which contains the energy of pixels, and writes the cumulative energy to the corresponding pixel in PGM image `output`.

```
void cumulativeEnergiesCPU(PGMData *input, PGMData *output, int new_width){
    //Start from the bottom-most row:
    for(int i = input->height-2; i >= 0; i--){
        for(int j = 0; j < new_width; j++){
            output->image[i*(input->width) + j] =
                input->image[i*(input->width) + j] +
                    getPreviousMin(output, i, j, new_width);
        }
    }
}
```

Listing 8.2 Compute cumulative energies

Using dynamic programming approach, we start at the bottom and work our way up, adding the cost of the cheapest below neighbor to each pixel. This way, we accumulate cost as we go—setting the value of each pixel not just to its own cost, but to the full cost of the cheapest path from there to the bottom of the image. The helper function `getPreviousMin()` returns the minimal energy value from the previous row. It contains a few compare statements to find the minimal value. As we can see, each iteration in the outermost loop depends on the results from the previous iterations, so it cannot be parallelized and only the iterations in the innermost loop are mutually independent and can be run concurrently. Also, to find the minimal value from the previous row, we should use conditional statements in the helper function `getPreviousMin()`. We already know that these statements will prevent the work-items to follow the same execution path and thus it will prevent effective execution of warps.

Seam Labeling and Removal on CPU

The process of labeling and removing a seam with minimal energy is quite easy. All we have to do to is to start with the darkest pixel with minimal cumulative energy in the top row and work our way down from there, selecting the cheapest of the three adjacent pixels in the row below and changing the color of the corresponding pixel in the original image to white. Listing 8.3 shows how to color the seam with minimal energy and Listing 8.4 shows how to remove the seam with minimal energy.

```
void seamIdentificationCPU(PGMData *input, PGMData *output, int new_width){
    int column = 0;
    int minvalue = input->image[0];
    //find the minimum in the topmost row (0) and return column index:
    for(int j = 1; j < new_width; j++){
        if (input->image[j] < minvalue) {
            column = j;
            minvalue = input->image[j];
        }
    }

    //Start from the top-most row :
    for(int i = 0; i < input->height; i++){
        output->image[(i)*(input->width) + column] = 255;
        column = getNextMinColumn(input, i, column, new_width);
    }
}
```

Listing 8.3 Labeling the seam with the minimal energy

```
void seamRemoveCPU(PGMData *input, PGMData *output, int new_width){
    int column = 0;
    int minvalue = input->image[0];
    //find the minimum in the topmost row (0) and return column index:
    for(int j = 1; j < new_width; j++){
        if (input->image[j] < minvalue) {
            column = j;
            minvalue = input->image[j];
        }
    }

    //Start from the top-most row:
    for(int i = 0; i < input->height; i++){
        // make this row narrower:
        for(int k = column; k < new_width; k++){
            output->image[i*(input->width) + k] =
                output->image[i*(input->width) + k+1];
        }
        column = getNextMinColumn(input, i, column, new_width);
    }
}
```

Listing 8.4 Seam removal

We can see in Listing 8.4 that seam removal starts with the loop in which we locate the pixel with minimal cumulative energy, i.e., the first pixel in the vertical seam with the minimal energy. Then we proceed to the loop nest. The outermost loop indexes rows in the image. In each row we remove the seam pixel—we move the pixels that are located at the right of the vertical seam from right to left. After that, we have to find the column index of the seam pixel in the next row. We do this using the helper function `getNextMinColumn()`. As in the previous step, we use conditional statements in the helper function `getPreviousMin()`, which will prevent the work-items to follow the same execution path.

8.4.2 Seam Carving in OpenCL

In this subsection , we will discuss the possible implementation of seam carving on GPU. The host code would be responsible for the following steps:

1. Load image from a file into a buffer. For example, we can use grayscale PGM images, which are easy to handle.
2. Transfer the image in the buffer to the device.
3. Execute four kernels: the energy calculation kernel, the cumulative energy kernel, the seam labeling kernel, and the seam removal kernel.
4. Read the resized image from GPU.

As discussed before, seam carving consists of three steps. We will implement each step as one or more kernel functions. The complete OpenCL code for seam carving can be found on the book's companion site.

OpenCL Kernel Function: Energy Calculation

The first step of seam carving is embarrassingly (or perfectly) parallel, because the calculation of the energy of individual pixels is completely independent of the energy of adjacent pixels. Each work-item will calculate the energy of the pixel whose index is the same as its global index. Energy is calculated from the values of adjacent pixels. Depending on the energy function used, we may need three, eight or even 24 adjacent pixels. Here, we will use our simple energy function from Eq. 8.1. The parallel algorithm for the energy calculation is given in Algorithm 8.

Algorithm 8 SEAM CARVING: PARALLEL ENERGY CALCULATION

Input: I - RxC image

1: **for all** work-item(i, j) in 2-dim NDRange **do**
2: $E(i,j) = |I(j,j) - I(rowGID, j+1)| + |I(i,j) - I(i+1,j)| + |I(i,j) - I(i+1,j+1)|$
3: **end for**

Output: E - RxC energy map

Listing 8.5 shows the code for the energy calculation kernel.

```
__kernel void simpleEnergyGPU(
                        __global int* imageIn,
                        __global int* imageOut,
                        int width,
                        int height,
                        int new_width) {

    // global thread index
    int columnGID = get_global_id(0);    // column in NDRange
    int rowGID = get_global_id(1);       // row in NDRange
    int tempPixel;
    int diffx, diffy, diffxy;

    diffx = abs_diff(imageIn[rowGID * width + columnGID],
                imageIn[rowGID * width + columnGID  + 1]);
    diffy = abs_diff(imageIn[rowGID * width + columnGID],
                imageIn[(rowGID+1) * width + columnGID]);
    diffxy = abs_diff(imageIn[rowGID * width + columnGID],
                imageIn[(rowGID+1) * width + columnGID + 1]);
    tempPixel = diffx + diffy + diffxy;

    if( tempPixel>255 )
        imageOut[rowGID*width+columnGID] = 255;
    else
        imageOut[rowGID*width+columnGID] = tempPixel;
}
```

Listing 8.5 The energy calculation kernel

Each pixel will also affect the energy of other adjacent pixels, so its will also be read by other work-items from the global memory. That means that the same word from the global memory will be accessed multiple times. Therefore, it makes sense to first load a block of pixels and their neighbors into the local memory and only

then start the calculation of energy. The reader should add the code for collaborative loading of the pixel block into local memory. Do not forget to wait for other work-items at the barrier before starting to calculate the pixel energy.

OpenCL Kernel Function: Seam Identification
Prior to calculating the cumulative energy of the pixels in one row, we should have already calculated the cumulative energy of all the pixels in the previous row. Because of this data dependency, we can only run as many work-items at a time as the number of pixels in one row. When all work-items finish the calculation of the cumulative energy in one row, they move on to the next row. One work-item will calculate the cumulative energy of all pixels in the same column, but it will move to the next row (pixel above) only when all other work-items have finished the computation in the current row. Therefore, we need a way to synchronize work-items that calculate cumulative energies. We can synchronize work-items in two different ways:

1. We can run all work-items in the same (only one) work-group. The advantage of this method is that we can synchronize all work-items using barriers. The disadvantage of this approach lies in the fact that only one block of work-items can be run, so only one compute unit on GPU will be active during this step. In this approach, we will enqueue one kernel. The parallel algorithm for the cumulative energy calculation is given in Algorithm 9.

Algorithm 9 SEAM CARVING: PARALLEL CUMULATIVE ENERGY CALCULATION IN ONE WORK- GROUP

Input: E - RxC energy map

1: **for all** work-item(j) in 1-dim NDRange **do**
2: $M(R, j) = E(R, j)$
3: **for** $i = R - 1 \ldots 1$ **do**
4: $M(i, j) = E(i, j) + min(M(i + 1, j - 1), M(i + 1, j), M(i + 1, j + 1))$
5: barrier()
6: **end for**
7: **end for**

Output: M - RxC cumulative energy map

2. We can organize work-items in more than one work-group. The disadvantage of this approach is that work-groups cannot be synchronized with each other using barriers. But we can synchronize blocks by running one kernel at a time. One kernel, consisting of several work-groups will compute cumulative energies in just one row. When finished, we will have to rerun the same kernel, but with different arguments (i.e., the address of the new row). Thus, we will enqueue the same kernel in a loop from the host code.

Which of two presented approaches is more appropriate depends on the size of the problem. For smaller images, the first approach may be more appropriate, while the

second approach is more appropriate for images with very long rows since we can employ more compute units.

Listing 8.6 shows the code for the cumulative energy calculation kernel, which will be used for testing purposes in this book. The kernel does not use local memory and does not implement collaborative loading. The reader should implement this functionality and compare both kernels in terms of execution times. The reader should also implement the kernel for the second approach and measure the execution time.

```
__kernel void cumulativeEnergiesGPU(
                        __global int* imageIn,
                        __global int* imageOut,
                        int width,
                        int height,
                        int new_width) {

    // global thread index
    int columnGID = get_global_id(0);    // column in NDRange

    //Start from the bottom-most row:
    for(int i = height-2; i >= 0; i--){
        imageOut[i*width+columnGID] = imageIn[i*width+columnGID] +
                                getPreviousMinGPU(imageOut, i, columnGID,
                                                  width, height, ↩
     new_width);
        // Synchronise to make sure the tiles are loaded
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 8.6 Cumulative energy calculation kernel - one work-group approach

OpenCL Kernel Function: Seam Labeling and Removal

The last step is to label and remove the seam with the minimal energy. Unfortunately, this step is inherently sequential because each pixel position in the seam strongly depends on the position of the previous pixel in the seam. So we cannot label all pixels in the seam in parallel. How can we implement this sequential function as a kernel on a massively parallel computer such as GPU? One possible solution would be that only one work-item labels and removes the whole seam. So the function for the GPU kernel would be almost the same as the function in Listing 8.3. The NDrange would have the dimension (1,1), i.e., we run only one work-item in the NDrange.

A better solution would be to divide this step into to operations: seam labeling and seam removal. The first step (seam labeling) should return column indices of every pixel in the seam. The kernel for this step will run in NDrange of dimension (1,1), i.e., only one work-item labels the whole seam. While the first step is inherently sequential, the second could be parallelized as follows. We run as many work-items as the number of rows in the input image. Each work-item removes the seam pixel in its row and copies the remaining pixels on its right one position to the left. The second step uses the array of column indices from the first step to locate its seam pixel. A parallel algorithm for seam removal is given in Algorithm 10.

Algorithm 10 SEAM CARVING: SEAM REMOVAL

Input: I - RxC original image
Input: C - $1xR$ vector of column indices

1: **for all** work-item(i) in 1-dim NDrange **do**
2: column = C(i)
3: **for** $j = col \ldots C$ **do**
4: $I(i, j) = I(i, j + 1)$
5: **end for**
6: **end for**

Output: I - RxC resized image

Listing 8.7 shows the code for the seam labeling kernel and Listing 8.8 shows the code for the seam removal kernel.

```
__kernel void getSeamGPU(
                __global int* imageIn,
                __global int* seamColumns,
                int width,
                int height,
                int new_width) {

    int column = 0;
    int minvalue = imageIn[0];

    //find the minimum in the topmost row (0) and
    //   return column index:
    for(int j = 1; j < new_width; j++){
        if (imageIn[j] < minvalue) {
            column = j;
            minvalue = imageIn[j];
        }
    }

    //Start from the top-most row:
    for(int i = 0; i < height; i++){
        column = getNextMinColumnGPU(imageIn, i,
                column, width,
                height, new_width);
        seamColumns[i] = column;
    }
}
```

Listing 8.7 Seam labeling kernel

```
__kernel void seamRemoveGPU(
                        __global int* imageIn,
                        __global int* imageOut,
                        __global int* seamColumns,
                        int width,
                        int height,
                        int new_width) {

    int iGID = get_global_id(0);       // row in NDRange

    // get the column index of the seam pixel in my row:
```

```
    int column = seamColumns[iGID];

    // make my row narrower:
    for(int k = column; k < new_width; k++){
        imageOut[iGID*width + k] = imageOut[iGID*width + k+1];
    }
}
```

Listing 8.8 Seam removal kernel

To analyze the performance of the seam carving program, the sequential version has been run on a quad-core processor Intel Core i7 6700HQ running at 2,2 GHz, while the parallel version has been run on an Intel Iris Pro 5200 GPU running at 1,1 GHz. This is a small GPU integrated on the same chip as the CPU and has only 40 processing elements. The results of seam carving for an image of size 512×320 are presented in Table 8.1.

As can be seen from the measured execution times, noticeable acceleration is achieved only for the first step. Although this step is embarrassingly parallel, we do not achieve the ideal speedup. The reason is that when calculating the energies of individual pixels, the work-items irregularly access the global memory and there is no memory coalescing. The execution times could be reduced if the work-items used local memory, as we did in matrix multiplication.

At the second step, the speedup is barely noticeable. The first reason for this is the data dependency between the individual rows. The other reason is, as before, irregular access to global memory. And the third factor that prevents effective parallelization is the usage of conditional statements when searching for minimal elements in previous rows. Here too, the times would be improved by using local memory.

At the third step, we do not even get speed up, but almost a 10X slow down! The reason for such a slowdown lies in the fact that only one thread can be used to mark the seam.

And last but not least, the processing elements on the GPU runs at a 2X lower frequency than the CPU.

Table 8.1 Experimental results

Step	CPU time [s]	GPU time [s]	Speedup
Energy calculation	0.010098	0.000698	14.46
Cumulative energy	0.004696	0.003276	1.43
Seam removal	0.000601	0.005690	0.11
Total	0.014314	0.009664	1.48