15-418 Final Project
Parallel Seam Carving

Abigail Li (abigail2@andrew.cmu.edu), Jonathan Liu (jsliu2@andrew.cmu.edu)

1 Summary

We implemented Seam Carving in OpenCV and used OpenMP to parallelize the computation of the cumulative energy map. We compared the performance of the PSC machines with our local machines and produced a GIF of an input image being seam-carved to highlight both the intermediate and the final results.

2 Background

Seam Carving is an image resizing algorithm that works by removing seams of the least importance. It takes in an image and removes a specified number of seams to resize it before outputting the final resized image. A seam is defined as a path from the top to bottom or left to right of an image of adjacent pixels such that there is exactly one pixel per row on the path. Removing a vertical seam adjusts the image's width while removing a horizontal seam adjusts the height. A pixel is considered adjacent to another pixel if the pixels are in adjacent rows and their column indices differ by at most 1. A seam of least importance is determined first by calculating an energy matrix closely resembling the gradient of the image, then by computing the cumulative energy matrix from top to bottom (for vertical seams), finding the lowest cumulative energy on the bottom row, and removing the corresponding seam. We briefly discuss our approach to computing the energy matrix, but the main focus of the project is on optimizing the computation of the cumulative energy matrix and removing the seams in batches.

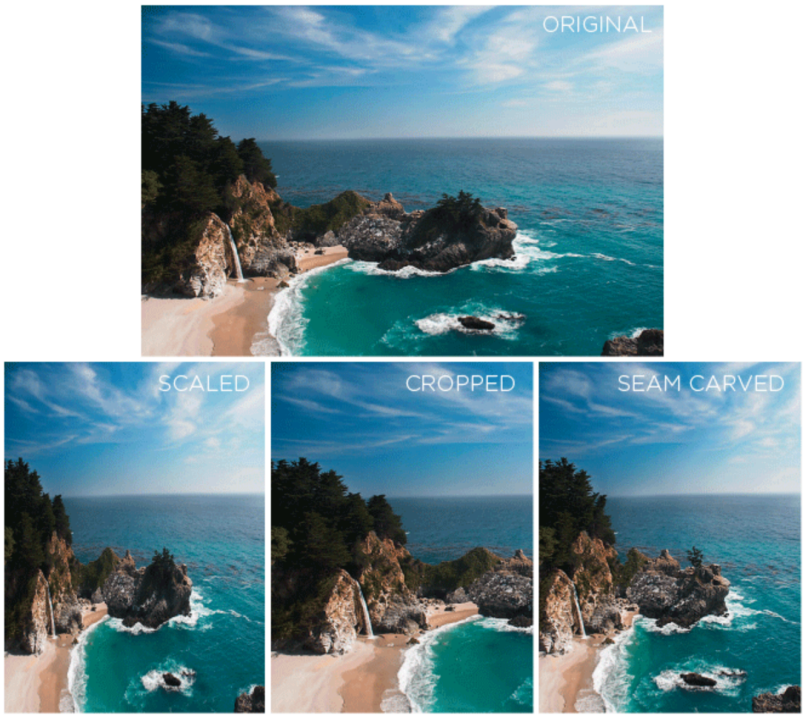

Figure 1: an image that has been scaled, cropped, and seam-carved respectively. The seam-carved photo retains more of the essence of the original image and is friendlier to the human eye compared to the scaled and cropped versions.

2.1 Computing the Energy Matrix

The energy matrix distinguishes the important and unimportant areas to keep during the resizing process by assigning an energy value at every pixel. High energies often correspond to the presence of edges or significant changes in the image, while low energies often correspond to the lack thereof. We examined several image importance measures discussed in the literature and considered using each of these as the energy function for the seam carver. After experimenting with various energy functions, we determined that the best energy functions are the ones that perform some measurement of the magnitude of the gradient [1]. Moreover, the gradient magnitude can be calculated using several methods, and among these, we chose to use the Scharr operator, which is designed to have better rotational symmetry and can improve the accuracy of edge detection, particularly for diagonal edges. This is because the coefficients in the Scharr kernels are optimized to provide a better balance between the horizontal and vertical derivatives. We also decided to apply a Gaussian blur on the image before applying the Scharr operator because we found that it would help reduce noise and produce high-quality carved images.

Computation of the energy matrix under the Scharr operator is a trivially highly parallelizable task: each pixel just needs the value of its neighbors in order to apply the convolution kernel. For this reason, and since OpenCV already has built-in libraries to accomplish this, we did not focus on improving an already highly optimized task and instead placed our focus on computing the cumulative energy matrix, which has a lot more dependencies and is more difficult to parallelize.

2.2 Computing the Cumulative Energy Matrix

The cumulative energy matrix is calculated as follows: find the minimum cumulative energy value of the three pixels directly above it and add that minimum to the pixel's own energy gradient value. In the image below, the cumulative energy of the yellow pixel is the sum of the minimum of the cumulative energy of the three green pixels plus the energy value at the yellow pixel:

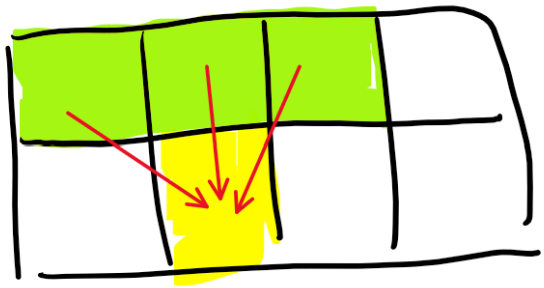

Figure 2: the cumulative energy of the yellow pixel depends on the cumulative energy of the green pixels

Computing the Cumulative Energy Matrix is difficult to parallelize because each row of the matrix can only be calculated after the previous row has been calculated. More specifically, the cumulative energy value of a pixel can only be determined once the cumulative energy values of the three pixels directly above it are determined. Thus, although the computation of each energy pixel in itself is inexpensive because all that is being done is reading from the three pixels above and writing to the pixel's slot in the cumulative energy matrix, the synchronization costs become large.

Because each pixel depends on the energy values of the pixels above it and we need to access the cumulative energy values later to find the minimum seam, we store the Cumulative Energy Matrix as a 2D Mat object in OpenCV.

2.3 Intuition for parallelism

There is an obvious source of data parallelism in that the computation of each pixel in a row does not depend on the computation of any other pixel in the same row or the rows after it. But although there is the option to parallelize the computation of an entire row of the image, this does result in significant speedup because there is too little work to be done before having to synchronize and compute the next row. The task of computing the cumulative energy at a single or a handful of pixels is too small an amount of work compared to the overhead introduced by a task scheduler. There is a little bit of temporal locality in that a value in the energy matrix is going to be read by the three pixels directly below it and a bit of spatial locality in that only adjacent pixels are read for each computation.

This is not amenable to SIMD execution because the synchronization time between rows is far greater than the benefits of naively parallelizing each row with SIMD.

3 Approach

3.1 Libraries and Technologies Used

We borrowed code from https://github.com/dwxiao/seam-carving, which uses the OpenCV library to process images before they are seam-carved. We then parallelized the algorithm via OpenMP and tested our speedup on the PSC machines.

3.2 Optimization Process and Approaches

For context, this was the initial code that computes the Cumulative Energy Matrix
We loop over every pixel and and calculate the cumulative energy for each pixel as described.

```
for (int row = 1; row < rowsize; row++) {
  for (int col = 0; col < colsize; col++) {
    // this finds the cumulative values at the three pixels above a
given pixel
    a = cumulative_energy_map.at<double>(row - 1, max(col - 1, 0));
    b = cumulative_energy_map.at<double>(row - 1, col);
    c = cumulative_energy_map.at<double>(row - 1, min(col + 1,
colsize - 1));

    // this computes the min
    cumulative_energy_map.at<double>(row, col) =
energy_image.at<double>(row, col) + std::min(a, min(b, c));
  }
}
```

We did not seriously start testing speedup on PSC until we were fairly certain that our local testing approach was optimized enough.

3.2.1 Naive Parallelization by Columns

Our first attempt at parallelization was parallelized by column, so we still followed the row-by-row format except that we would compute each row in parallel. Based on the structure of the sequential code, this was very easily achievable. The code looks like this:

```
for (int row = 1; row < rowsize; row++) {
  #pragma omp parallel for private(a, b, c)
  for (int col = 0; col < colsize; col++) {
    // this finds the cumulative values at the three pixels above a
given pixel
    a = cumulative_energy_map.at<double>(row - 1, max(col - 1, 0));
    b = cumulative_energy_map.at<double>(row - 1, col);
    c = cumulative_energy_map.at<double>(row - 1, min(col + 1,
colsize - 1));

    // this computes the min
    cumulative_energy_map.at<double>(row, col) =
energy_image.at<double>(row, col) + std::min(a, min(b, c));
  }
}
```

The only difference is the bolded line above. However, we realized that this was not the most effective because the work per row isn't enough that parallelizing by columns would make a significant difference, especially because each row needed to synchronize before the next row could be started. Through brief local testing, we determined that this approach achieved a speedup of nearly 1.8x on 4 cores and did not have additional improvement when the number of cores was increased beyond that. This was consistent with our hypothesis that such an approach would not produce significant speedups due to task scheduling overhead. Since local

testing already proved our point, we decided to not test on PSC until we reached a more optimized solution locally.

3.2.2 Parallelization via Triangles

We understood that to improve the algorithm, we needed to ensure that the granularity of each task would be on the order of hundreds of pixels. We deemed this necessary since the naive parallel approach suffered from too small a grain size. This gave rise to the second iteration of our parallel algorithm, where we adjusted the grain size to the order of tens of pixels instead of just a single pixel:

```
for (int row = 1; row < rowsize; row++) {
  #pragma omp parallel for private(a, b, c)
  for (int block = 0; block < (colsize + BLOCK_SIZE - 1) /
BLOCK_SIZE; block += BLOCK_SIZE) {
     for (int col = block; col < min(block + BLOCK_SIZE, colsize);
col++) {
        ...
     }
```

Unfortunately, this barely improved performance. We realized that by increasing the granularity of each task, we would not have enough tasks to schedule before having to synchronize. Besides the overhead of having to fork a bunch of threads to run for only a short period, we also suffer from an imbalanced workload for the threads. To simultaneously have a sufficiently large grain size and also sufficiently many tasks, we found it absolutely necessary to have the tasks span multiple rows, and this finally led us to discover a new approach. We observed that the dependency graph for computing a single entry in the cumulative energy matrix formed a triangle, and furthermore, that a single thread could do work on a subsequent row so long as it also did the necessary work above it. We formulated our ideas into what we termed the “triangle method,” which goes as follows:

First, we separate our image into strips as shown in the image below for vertical seam removal:

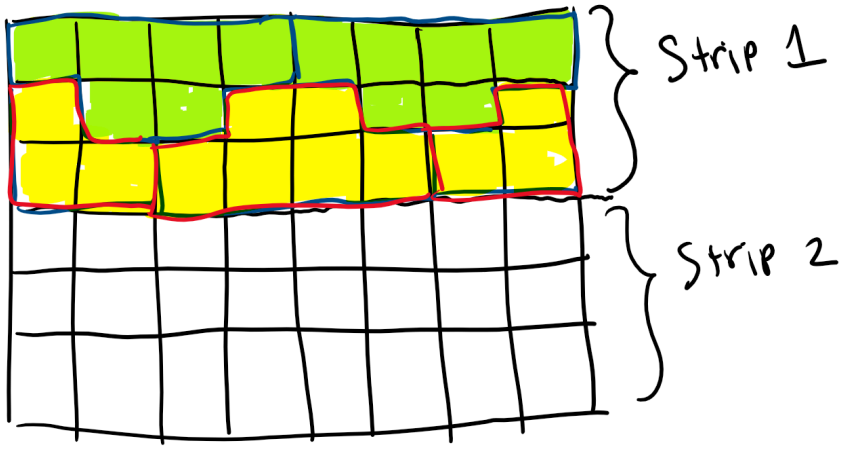

Figure 3: visualization of how the work of computing the cumulative energy matrix is partitioned at each strip of the matrix

We then iterate strip by strip, so Strip 1 is processed before Strip 2. Then we parallelize across all the green triangles in the strip and find the cumulative energy of each pixel in each triangle. After that's done we parallelize across all the yellow triangles in the strip and find the cumulative energy of each pixel in each triangle. Within each triangle, we go from up to down and left to right, as usual.

The height of each triangle/the number of pixels composing a strip is flexible, so we further fine-tuned our algorithm for what should be the best height of a strip, which we will refer to as the JUMP parameter.

We mapped each triangle in either the top half or bottom half of a strip to a task that would be done by a single thread.

3.2.3 Optimizing via Batched Seam Removal

After we had optimized our triangle solution, we were still not achieving the speedup we wanted (this will be discussed more in the results section), so we ended up implementing batch seam removal.

The idea of batch seam removal is to identify multiple seams for removal for a single computation of the cumulative energy matrix. This approach will yield different results compared to the sequential algorithm and is lossy with respect to finding the absolute lowest energy seams. By not recomputing the cumulative energy matrix, we lose out on considering any seams that intersect the seam just removed, so the seams we identify must be disjoint. As an initial idea, we tried to cut the image in half and compute two separate cumulative energy matrices, but this fell short for two reasons: first, by cutting the image in half, we limit the amount of row parallelism afforded to us by decreasing the length of each row. Secondly, this approach does not necessarily guarantee a good solution and may produce poorly carved images. For example, if all the seams lie in one section of the image and our "cut" of the image does not subdivide this section, then we would be forced to remove seams of higher energy than we would like.

We tried removing the two seams with the lowest cumulative energy value in parallel as described above, but that was not super effective. In the photo below, the climber's body is very distorted from the original image.

Figure 4: Removing 2 seams at one by finding the two lowest energy seams

Given this, we also tried doing another estimate where we sacrificed accuracy for speed and removed a seam that was a specified width of pixels, which we will refer to as a fat seam. We decided to try this approach because we realized that the lowest energy seams of an image are often nearly adjacent to one another.

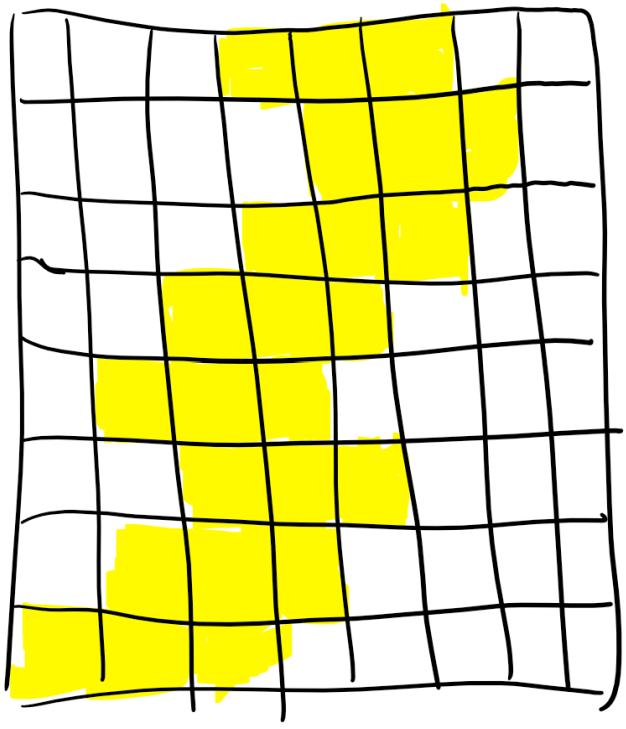

Figure 5: A fat seam that is 3 pixels wide is highlighted in yellow

4 Results

4.1 Experiment Setup

We chose to measure speedup in the time it took to compute the seam-carved image overall, with a particular focus on computing the cumulative energy matrix and identifying multiple seams at once.

We ran our experiments by vertically seam-carving a 720 x 1280 image 300 times. We ran local experiments on the ideal size of the JUMP parameter (the width of a strip) and experiments on the number of processors vs speedup on the PSC machines. Experiments were performed on the following image:

Figure 6: the baseline image on which experiments were run

For our baseline, we ran our starter code as our naive sequential version on the PSC machines. This is because we discovered that each of our local machines had very different results when running the code despite the relative results appearing the same, so we wanted to have our PSC results consistent with whatever baseline we took.

We thought it was not super important to report results for different image sizes: given a large enough input image (roughly 1000x1000 pixels), the one variable that we care about regarding problem size is the number of seams we carve out. We performed tests on images of varying sizes and depictions of different scenes and got very consistent results. Thus, we chose a single image to use in a more cohesive set of experiments measuring the impact of our various optimizations. We also noted that carving out a greater number of seams would result in a greater amount of speedup due to the image getting smaller as more seams are removed. For this reason, we chose, as a nice balance, to remove 300 vertical seams from a 720 x 1280 pixel

image. This is further supported by how the speedup from 1 to 16 processors only increased by about 0.2x going from carving 300 seams to carving 500 seams. Currently, different workloads do not exhibit different execution behavior.

4.2 Visual Results

This was our original image after it had been seam-carved 300 times:

Figure 7: the result of removing 300 seams using the lossless approach

Below is a GIF of the image being seam-carved. Currently, the right part of the image becomes more blurry as more seams are removed in order to maintain the video's aspect ratio.

faster_output.mp4
(link here:
https://drive.google.com/file/d/1_6ez5_4vXA2TCg1s9ON2GoixIKUBfNVM/view?usp=sharing)

4.3 Data from Parallelizing via Triangles

This is the graph of all the JUMP parameters we tried versus the time it took on a single CPU locally. We determined that the best JUMP parameter was around 90.

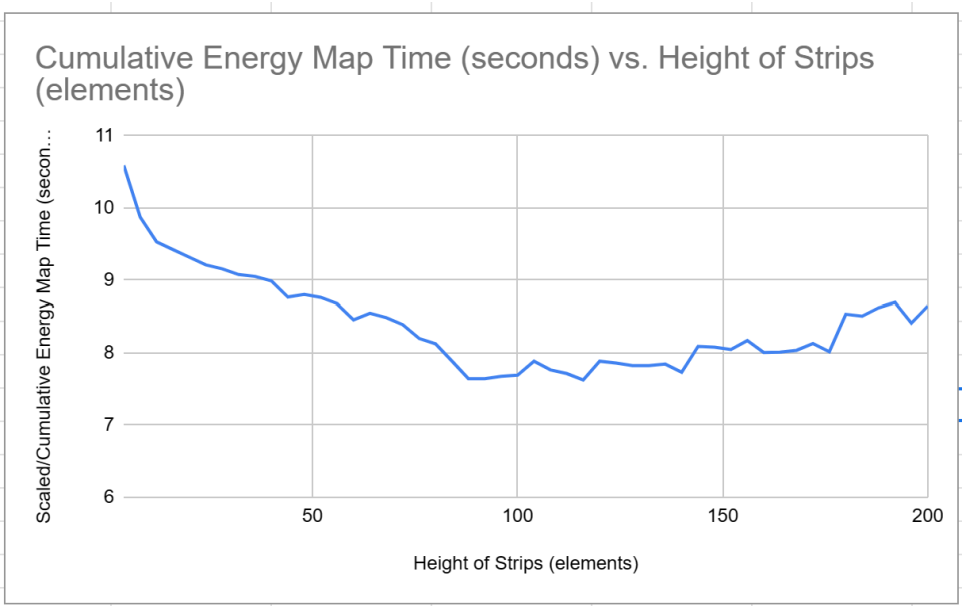

Figure 8: varying the JUMP parameter introduces a tradeoff between the number of tasks we have with the size of each task

Using the optimal JUMP parameter, we also measured the speedup with different values of N_{procs}, the number of processors.

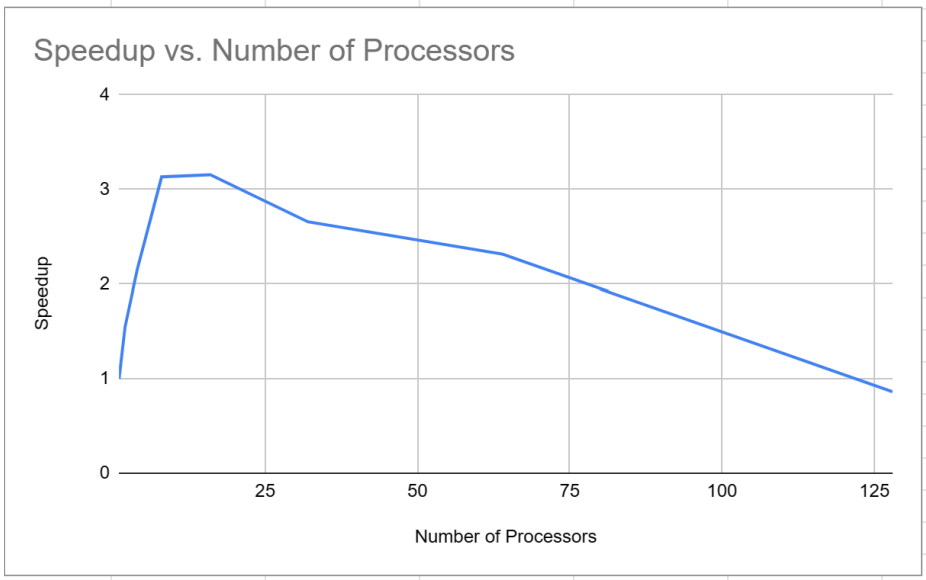

Figure 9: a plot of the number of processors versus speedup in the time it took to compute the cumulative energy map.

We hypothesized that a combination of lack of parallelism and synchronization overhead limited our speedup. We ran a series of experiments using **perf stat -e cache-misses** and noticed that both our total cache misses and cache misses per thread scaled very nicely with the number of threads, as shown below.

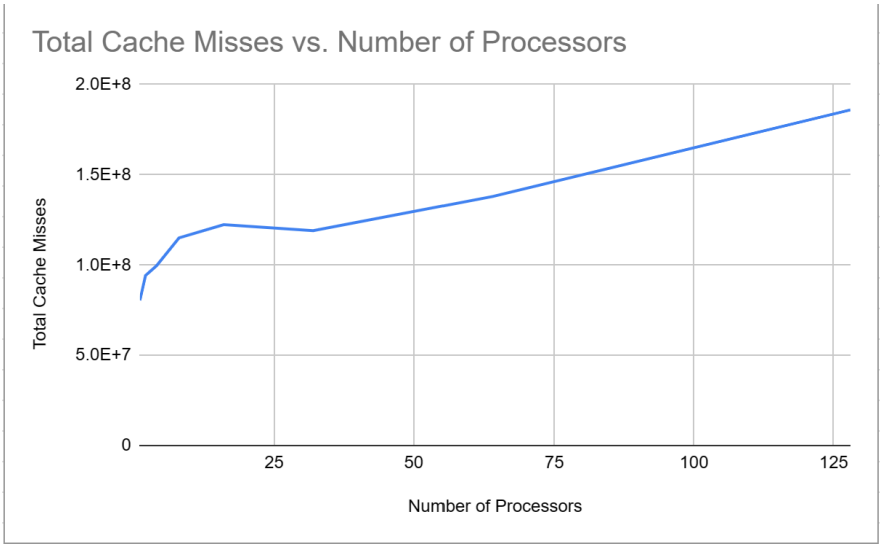

Figure 10: the total cache misses as a function of the number of processors

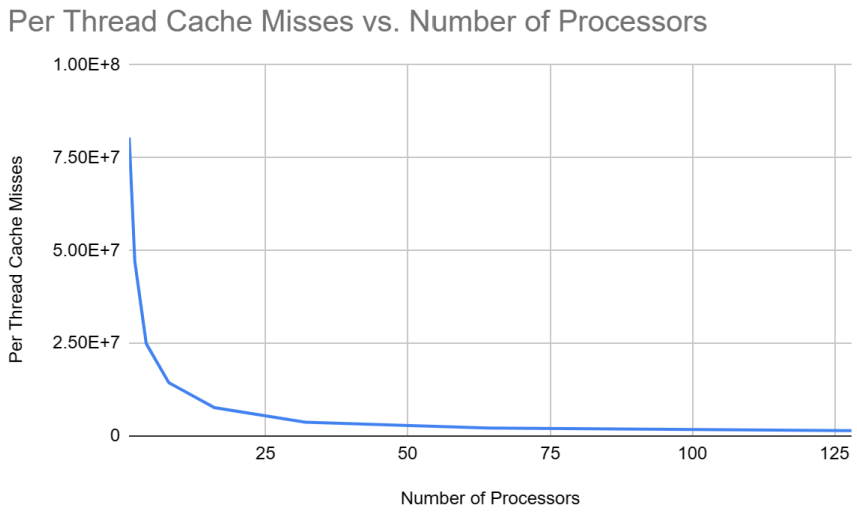

Figure 11: the per-thread cache misses as a function of the number of processors

The total cache misses do not increase that much and the cache misses per thread decrease exponentially as the number of processors increases. This means that we have had pretty effective cache utilization and that is not the limiting factor behind our lack of perfect speedup.

In light of our effective cache utilization, we think that a lack of parallelism/synchronization overhead is the bottleneck behind our lack of parallelism. In our current approach, the top triangles of each strip need to synchronize before the bottom triangles can be calculated. Both of these are limitations to our approach and we proposed further directions we could go to address these problems.

On average, the execution of the initial lossless approach naively parallelizing by columns can be broken down as follows:

Process	Time (seconds)
Finding the energy of image	2.488
Making the cumulative energy map	5.65 - 12.89 (depending on the number of threads)
Finding the minimum seam	0.0389
Reducing image to take out the seam	0.3791

Table 1: a breakdown of the execution times of the different components of the pipeline for the initial lossless approach

On average, our execution time for the best lossless approach can be broken down as follows:

Process	Time (seconds)
Finding the energy of image	2.464
Making the cumulative energy map	2.25 - 8.2 (depending on the number of threads)
Finding the minimum seam	0.0375
Reducing image to take out the seam	0.3872

Table 2: the execution times for the best lossless approach

The time not spent making the cumulative energy map is consistent across the number of threads.

In the best speedup we achieved, we spent 43.8% of the time making the cumulative energy map, 48% of the time finding the energy of the image, 0.7% of the time finding the minimum seam, and 7.5% of the time reducing the image.

4.4 Data from Batched Seam Removal

We hit a maximum speedup of about 2.8x on 16 threads using our triangle approach, but we wanted to see if we could further optimize that to get closer to a 16x speedup for 16 threads. We ended up experimenting on how much speedup we could get vs the width of a fat seam with the same baseline we used earlier and these were the results:

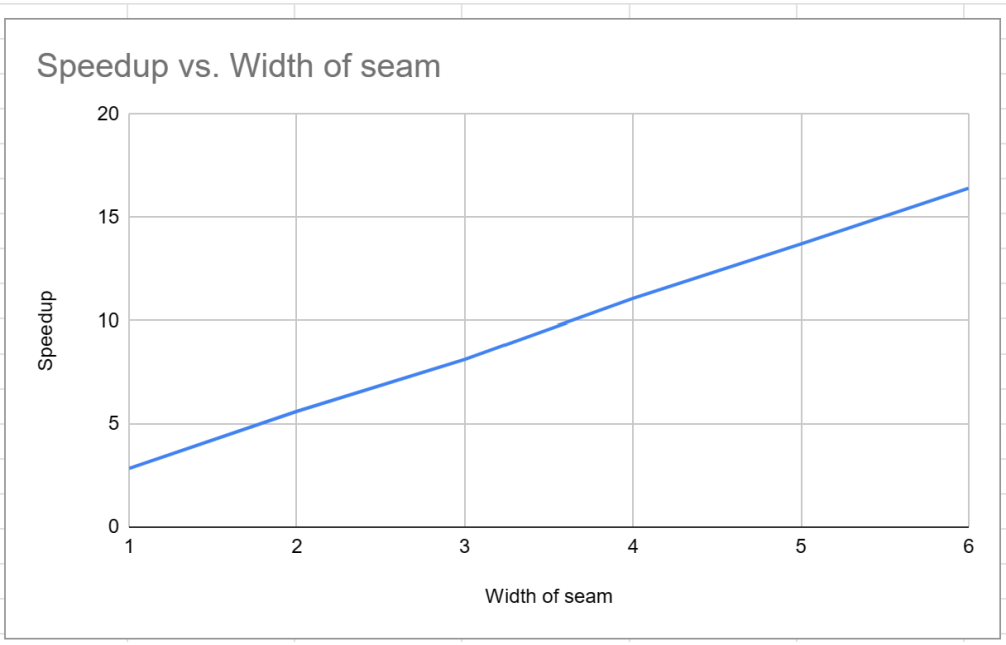

Figure 12: the speedup of the total time spent computing the cumulative energy matrix for different widths of the removed seam

We were able to achieve a lot more speedup because we had to compute the cumulative energy map fewer times if we took out more seams each iteration.

In terms of accuracy, we lose a little accuracy but not by much:

Figure 13: Removing 1 seam at a time

Figure 14: Removing 2 seams at a time

Figure 15: Removing 3 seams at once

Figure 16: Removing 4 seams at once

Figure 17: Removing 5 seams at once

Figure 18: Removing 6 seams at once

As the width of a fat seam increases, the edges of the photo get a little rougher as shown on the shadow of the leftmost green rock close. However, the body of the climber and the overall photo is well-preserved.

We ran **perf stat -e cache-misses** once again and noticed that the total cache misses and cache misses per thread both continued decreasing as the width of a fat seam increased. Again, this makes sense because we compute the cumulative energy map fewer times when we take out more seams each iteration. The perf graphs are shown below:

Figure 19: total cache misses as a function of the width of the removed seam using 8 threads

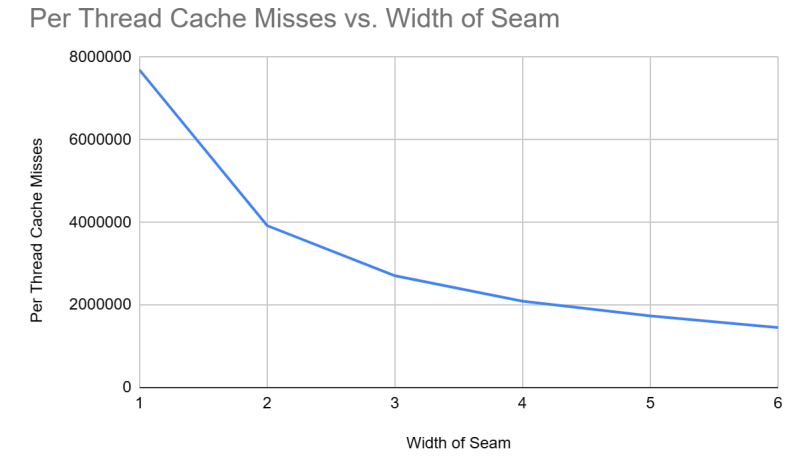

Figure 20: the per-thread cache misses as a function of the width of the removed seam using 8 threads

4.5 Further Improvement

One place we hypothesize can be improved is how we split the triangles. Currently, we are splitting into disjoint triangles but we believe that making the downward-pointing triangles overlap a little could potentially improve speedup. This technique is useful when we are trying to carve vertical seams from images that are not very wide – this lets us increase the size of triangles from the small limit imposed by disjoint triangles and possibly gain more speedup. This comes with the sacrifice of having repeated work, of course, since overlapping the triangles means that the intersection of two triangles will be computed twice.

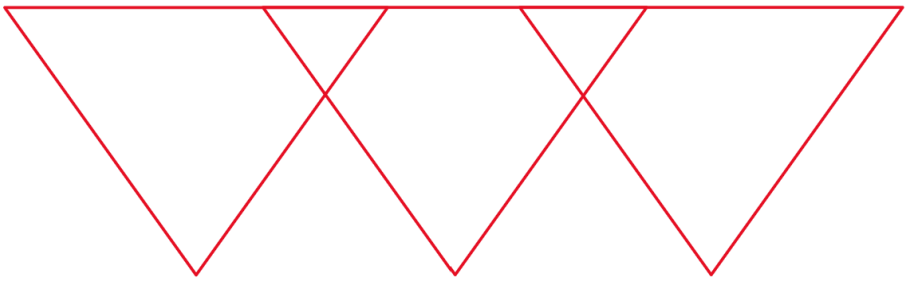

Figure 21: an illustration of how allowing some repeated work may offer greater parallelism

5 References

[1] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. ACM Trans. Graph., 26(3), July 2007.

6 Division of Work

We think that the credit should be distributed 50/50.

Jonathan

- Wrote naive parallel code parallelizing columns for vertical seams
- Wrote optimized parallel code parallelizing by triangles for vertical seams
- Adapted triangle code for horizontal seams
- Tested best JUMP parameter for triangles
- Wrote/tested versions of batch seam removal
- Final Report

Abby

- Found starter code, adapted starter code to run locally
- Wrote most of the midpoint project report
- Wrote code that takes all the intermediate images and produces a GIF of the seam-carving process (demo purposes only)
- Ran experiments to obtain speedups on higher core count on PSC machines for optimized triangle and batch seam removal versions
- Ran experiments on cache misses vs number of threads using perf on PSC machines for optimized triangle and batch seam removal versions
- Debugged performance issues based on PSC results
- Final Report