Cody Yang
Kevin Zheng
Kimberly Lim

Content-aware Re-scaling using Seam Carving in Image Processing

Proposed Problem

Typical image rescaling techniques include cropping and scaling, neither of which take into account the actual content within the image. Cropping an image potentially omits important content from the image. Scaling distorts all features in the image.

Proposed Solution and Algorithm

Content-aware rescaling takes into account features within the image, and intelligently targets parts of the image to remove or interpolate during the rescaling process. Seam carving is our proposed algorithm for content-aware rescaling.
The algorithm involves:
(1) two 2D convolutions with X and Y Sobel kernels to compute an energy map,
(2) a downward pass of the entire frame to compute all minimum paths, and
(3) an upward pass to find the least-energy seam.

State-of-the-art Techniques

There different content-aware rescaling techniques including deep learning, shrinkability maps, and seam carving. NVIDIA uses deep learning in their AI-powered content aware fill, which produces remarkable results. Several papers have also proposed using shrinkability maps to preprocess videos and allow for dynamic resizing. Since seam carving is a relatively simple algorithm, we will implement and optimize the seam carving technique.

Problem Scale and Constraints

Seam carving is applicable to both images and video. Images typically contain hundreds to thousands of pixels in each dimension. Video frames go up to a resolution of 4096x2160 at 60 frames per second. Seam carving would be well-suited for a graphical image or video editor, so it would be ideal to have an interactive response time as a user resizes a window. As such, the scale of this problem is limited to the resources of a single workstation-level device.

Intended Architecture

We target an Intel Broadwell x86-64 processor. This processor supports AVX/AVX2 SIMD instructions. We chose this architecture because it’s easily accessible in the CMU computing environment.

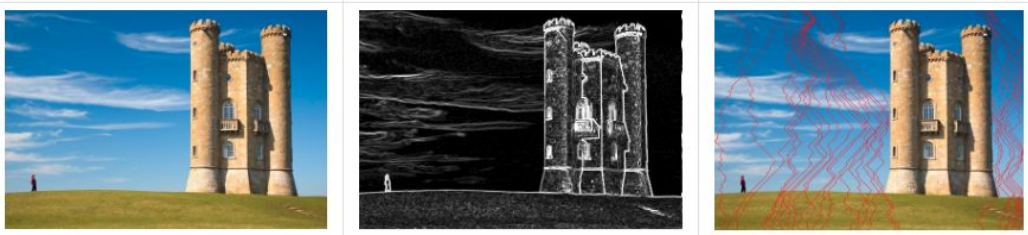


Figure 1 Simple Carving Illustrated. Left shows the sample image; Center shows the energy map of image; Right shows image with the seams applied

Seam carving algorithm

Step 1: Compute an energy map of the image

The energy map quantifies how important each pixel of the image is relative to the entire image. The higher a pixel's energy, the more important it is, and the more it should be avoided during the subsequent carving process. Generally, the image's edges are more important than smooth gradients. The edge detection algorithm we use to locate the image's edges is 2D convolution with a Sobel filter.

2D convolution works by applying a small kernel on each pixel of the image. Suppose that the input image has *m* rows and *n* columns, and the kernel has *x* rows and *x* columns. The 2D convolution produces an energy map with the same size as the input image. Each pixel *i*, *j* of the energy map is computed from the element-wise product between an *x* by *x* square of the input image centered at *i*, *j*, and the kernel. The products are summed, normalized by the magnitude of the kernel, and written to *i*, *j* in the energy map.

The 2D convolution kernel we chose is the Sobel kernel. Two convolutions - one for edges in the X dimension, and another for edges in the Y dimension - are performed, and their results combined to find all edges.

-1	0	+1
-2	0	+2
-1	0	+1

x filter

+1	+2	+1
0	0	0
-1	-2	-1

y filter

Figure 2 Sobel Kernels

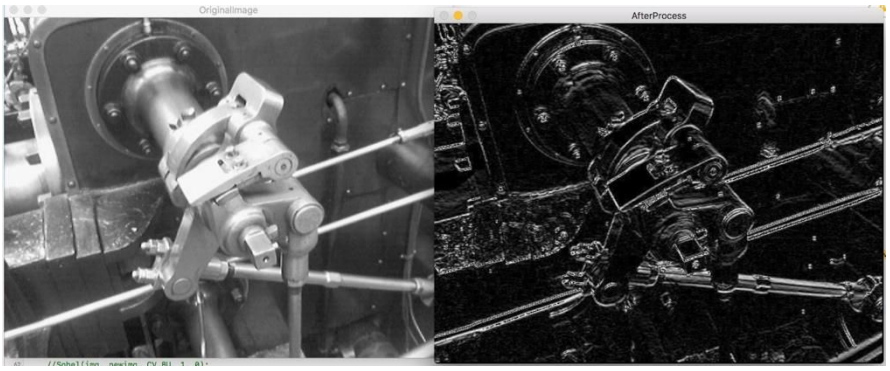


Figure 3 Left is Original Image; Right is energy map produced by 2D convolution with Sobel Kernel

Step 2: Compute all path sums

The next step is to compute all least-sum paths in the dimension of the image to be carved. The minimum path step will use the results from this step to identify the seam path with the smallest energy. Brute forcing the search for the least energy seam by trying all possible seams is clearly infeasible, since the number of seams scales exponentially with image size. Instead, we use a dynamic programming technique to find and keep track of the least energy path for each pixel in the image.

To find a vertical seam, the path sums are computed row by row, from top to bottom. Clearly, the path with the lowest energy for only the first row of the image is simply the value of each pixel in that row. For each pixel in the second row, the path from the top of the image with the least energy is the value of that pixel in the second row, plus the minimum path value of the three adjacent pixels in the row above. In general, for each pixel in the *n*th row, the path is the value of that pixel, plus the minimum path energy of the three adjacent pixels above.

Step 3: Find the minimum path

After completing step 2, the pixel values on the bottom of the image are the smallest path sums for paths that end on that pixel. First, select the smallest value in the last row, and label this column *j* and row *i*. Next, look at row *i-1*, columns *j-1*, *j*, and *j+1*. The minimum of each of these three pixels is the next pixel in the seam. Restart the algorithm from this pixel, and repeat until you reach the top. The collection of pixels is the seam to be removed from the image.

Implementation Design and Performance Peaks

Datatypes and layout

There are two data structures that our implementation uses:

- gray_image: Row-major arrays of 8-bit unsigned integers, which represent grayscale image data
- energymap: Row-major arrays of 32-bit signed integers, which represent energy values.
    - During the 2D convolution, these can be values can be negative. The actual energy map only contains non-negative values.

The grayscale images use 8-bit unsigned integers, because seam carving does not require high granularity pixel values, and we would like to minimize the amount of memory accessed.

The energy map uses 32-bit integers because during the minpath array calculation, energy values can scale at up to the height of the image. 16-bit integers would not be sufficiently ranged to prevent overflow. For instance, with 16-bit integers, an image with a height of 5000 pixels would only allow for an average pixel energy value of up to 2^{15} / 5000, or about 13. 32-bit integers prevent overflow for reasonably sized images.

Each image type also contains additional metadata:

- width: The width of the image, in pixels
- height: The height of the image, in pixels
- buf_width: The width of the buffer, in pixels
- buf_start: The starting column of each row, in pixels

Each row of the image occupies a size of buf_width in memory, regardless of its actual width. That is, pixel (*i, j*) is actually located at index (*i * buf_width + j + buf_start*) in memory. This is an optimization to minimize data movement during the seam removal step, explained below.

2D convolution

Independent operations. There are two levels of independent work in this step. Firstly, each output pixel energy value is independent of each other. Thus, each pixel in the output can be computed in parallel, since there are no data dependencies. We use SIMD to achieve parallelize this step. Secondly, to compute each output pixel, each load and multiplication is independent. We use this to our advantage in scheduling the instructions to help ensure all relevant functional units are as saturated as possible during this step.

SIMD Optimization. The 2D convolution with a 3x3 kernel uses AVX SIMD instructions to simultaneously compute the convolutions of 8 adjacent pixels in a single row. Each lane in the SIMD vector computes a single pixel in the output image.

To illustrate, this is what happens in a single SIMD lane *n*: To compute the convolution of pixel *i,j* in the output:

1. The values *((i-1,j-1), (i-1, j), (i-1, j+1), (i, j-1), (i, j), (i, j+1), (i+1, j-1), (i+1, j), (i+1, j+1))* are loaded into the same lane *n* of 9 different SIMD vectors.
2. Then, the 9 kernel values are loaded into the same lane *n* of 9 different SIMD vectors.
3. Each of the 9 input vectors and 9 kernel vectors now contain a single value from the input image and a single value from the kernel.
4. The 9 input vectors are multiplied by the 9 kernel vectors, yielding 9 partial result vectors. Note each multiply is independent.
5. A reduction is performed across the 9 partial results, eg. ((r1+r2) + (r3+r4)) + ((r5+r6) + (r7+r8)) + r9. Since there are 2 integer vector addition units, each with a latency of 1 cycle, the addition units are fully saturated by this reduction tree.
6. The result is scaled by the magnitude of the kernel.
7. The absolute value of the result is taken to prevent negative values.

8. The result is written to the output.

Since each lane computes an adjacent pixel, the other lanes perform the same operation, but with the input pixels shifted over by one.

Kernel Reformulation. We already optimized in previous phases by using SIMD operations to compute 8 output pixels at a time. We reformulated as below to use fewer operations.

Naive: { 9 (ld), 18 (x), 17 (+), 2 (>>), 2 (abs) }, 48 ops

After : { 8 (ld), 5 (+), 5 (-), 4 (>>), 2 (abs) }, 24 ops

This reformulation is possible since all convolution kernel values are { -2, -1, 0, 1, 2 } (in 3 * 3 kernel there are repeated values):

Naive:
$resultX = abs(i_{00}kx_{00} + i_{01}kx_{01} + i_{02}kx_{02} + i_{10}kx_{10} + i_{11}kx_{11} + i_{12}kx_{12} + i_{20}kx_{20} + i_{21}kx_{21} + i_{22}kx_{22}) / sum(kx)$
$resultY = abs(i_{00}ky_{00} + i_{01}ky_{01} + i_{02}ky_{02} + i_{10}ky_{10} + i_{11}ky_{11} + i_{12}ky_{12} + i_{20}ky_{20} + i_{21}ky_{21} + i_{22}ky_{22}) / sum(ky)$

Actual kernel values:
$resultX = abs(-1*i_{00} + -2*i_{01} + -1*i_{02} + 0*i_{10} + 0*i_{11} + 0*i_{12} + 1*i_{20} + 2*i_{21} + 1*i_{22}) / 8$
$resultY = abs(-1*i_{00} + 0*i_{01} + 1*i_{02} + -2*i_{10} + 0*i_{11} + 2*i_{12} + -1*i_{20} + 0*i_{21} + 1*i_{22}) / 8$

OptimizedAfter:
$sc = (i_{22} - i_{00});$
$resultX = abs((i_{21} - i_{01}) << 1 + sc + (i_{20} - i_{02})) >> 3$
$rexultY = abs((i_{12} - i_{10}) << 1 + sc + (i_{02} - i_{20})) >> 3$

From here, our performance is bounded by integer vector ALU throughput:

Performance bound of integer ALU throughput. The SIMD (+, -, abs) instructions share an integer vector ALU pipeline with the (1b -> 4b) upcast SIMD load. Thus, there is extremely high pressure on that pipeline, since the majority of the work done are (+, -, abs, store) operations. So to improve performance, we optimized away as many ALU instructions as possible using several techniques:

(1) Algebraic simplification. Rewriting the naive expression to eliminate multiplies by 0, replacing mutiplies by -1 with subtracts, multiplies by (-2, 2) with two adds/subtracts, and divides with right shifts greatly cuts down on the number of ALU instructions needed. Modern optimizing compilers can help to an extent here, but there is no substitute for writing out the expressions by hand here.

(2) Loop fusion. There are two Sobel kernels - one for the X direction, and another for the Y direction - the two convolutions are fused into a single convolution, such that each load of the input image is used in both convolutions, reducing the number of loads. This also allows us to compute the shared term (*sc* above) only once, saving a subtraction.

(3) Replace (+) with (<<). Instead of adding or subtracting the same value twice for the -2 and 2 terms in the Sobel kernel, and to avoid a high-latency integer multiplication instruction, we use a left shift instruction (which uses a different ALU pipeline from the other operations) to eliminate two adds.

Finally, unrolling the main loop to compute several output pixel vectors in a single iteration eliminates the conditional branch between output vectors, which helps the CPU's out of order execution engine effectively pipeline the load and arithmetic instructions across output vectors.

Performance peak. The performance peak of this step is 1.25 cycles per output pixel, bound by the throughput of Broadwell's two vector integer ALU functional units. Each output pixel requires 20 operations in these pipelines. There are sufficient independent instructions to fully hide the 3 cycle latency of the loads, and prevent pipeline bubbles. Assuming two instructions can be completed per cycle, the theoretical peak is (20 instr / 2 func units) / (8 pixels / output vec) = 1.25 cycles / pixel. A full visualization of the instruction scheduling is in the appendix.

Compute path sums

This operation is relatively straightforward to vectorize. Vector A would contain pixels (i,j), (i,j+1), (i,j+2), (i,j+3). Vector B would contain pixels (i, j+1), (i,j+2), (i,j+3), (i,j+4). Vector C would contain pixels (i, j+2), (i,j+3), (i,j+4), (i,j+5) and so on. The output vector is a reduction minimum of these three vectors and summed with the corresponding vector in row i+1. A downward pass is performed in order to compute all path sums and the minimum seam can be determined from the minimum path sum in the last row. Each minimizing operation is independent since comparisons are with different pixels in the same row. We optimized our data access pattern by using SIMD permute and blend to avoid reloading data as Vectors B and C have the same values of Vector A just shifted left by one and two elements respectively. Since Vector A is missing the extra one/two elements being shifted in, we preload the next iteration's Vector A in order to blend in the extra elements into the first and second element positions of current Vector A. SIMD permute is then used to perform the actual left shifting. In addition, we unrolled the loop to try to saturate the load functional unit pipeline.

Performance Peak. In pathsum, we load values from the energy map computed before, take the minimum of those values and store it to a another energy map. If we disregard the computation for finding min, we can compare pathsum to a memcpy. memcpy achieves 9.2 GB/s on the ghc machines, as does a basic AVX2 copy. Our pathsum step currently achieves about 8.95 GB/s on the ghc machines. Accounting for the cycles needed for min computation, we can say that we are at least close to 8.95/9.2 ~= 97% of the memory bound.

If we assume no memory latency or bandwidth limitations (that is, all data resides in the L1 cache), then our implementation has a theoretical peak of 1.5 cycles per output vector, bound by the throughput of the two memory load functional units. However, because the entire image is typically too large to reside soley in the L1 cache, the implementation is actually bound by the memory access latency. See the appendix for a sample instruction scheduling.

Find the minimum path

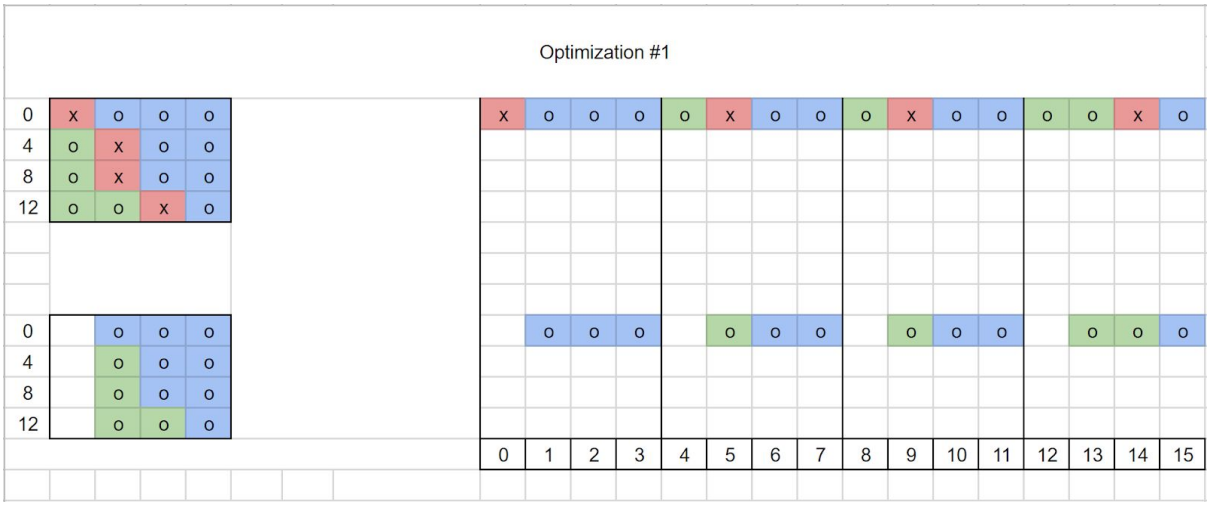
This stage is inherently sequential, so no vectorization will be used. It may be possible to cache multiple minimum path sums on the downward pass in the previous step, but this will be a future optimization to think about.

Performance Peak. Finding the minimum path is entirely sequential as each pixel to be removed depends on the pixel before it. Thus, this operation is limited by the latency of the compare instruction. 2 dependent compares must be made per pixel in the path, so a total of 2*m* dependent compares must be made on the entire image. The bottleneck is thus the latency of the compare instruction. Broadwell's *cmov* instruction has a latency of 1 cycle, so the theoretical peak of this step is 2*m* cycles.

Seam Removal

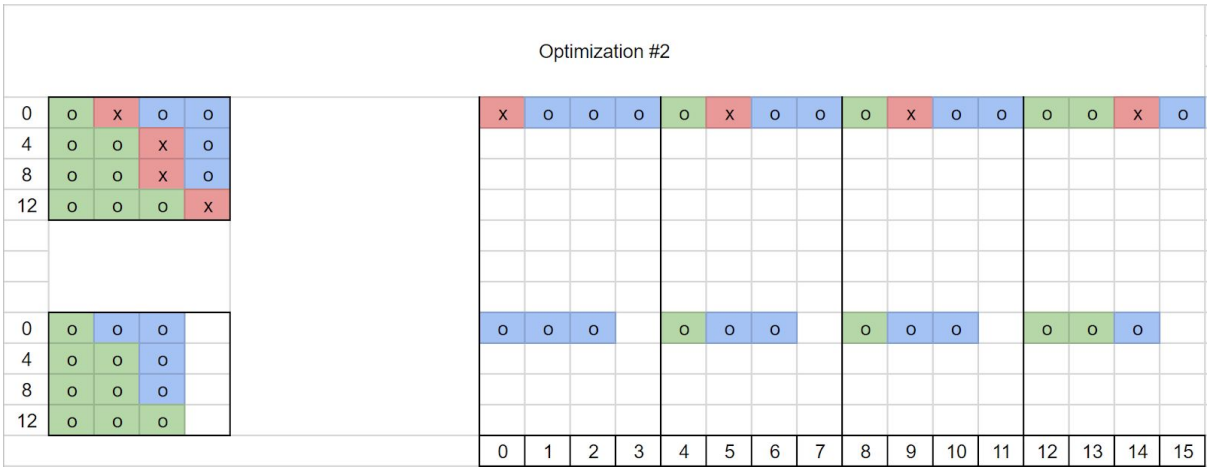
Removing a seam from the image is an exercise in data movement. There is no arithmetic intensity: The only thing going on here is moving part of each row over by one pixel to remove the carved seam. As such, this step is bound by the memory latency, so minimizing the amount of data moved in each carve is critical in maximizing performance. There are two main optimizations at play here:

Optimization 1: Removing a seam from an image will not re-pack the image. To tightly pack an image with a seam removed, part of the first row needs to be shifted by 1 pixel, the second row by 2 pixels, and the *n*th row by *n* pixels. Instead, only shrink within the initial rows of the image, so that each pixel is shifted at most by one pixel. On average, ½ of the image needs to be touched. This is why buf_width needs to be kept around in the image structure.

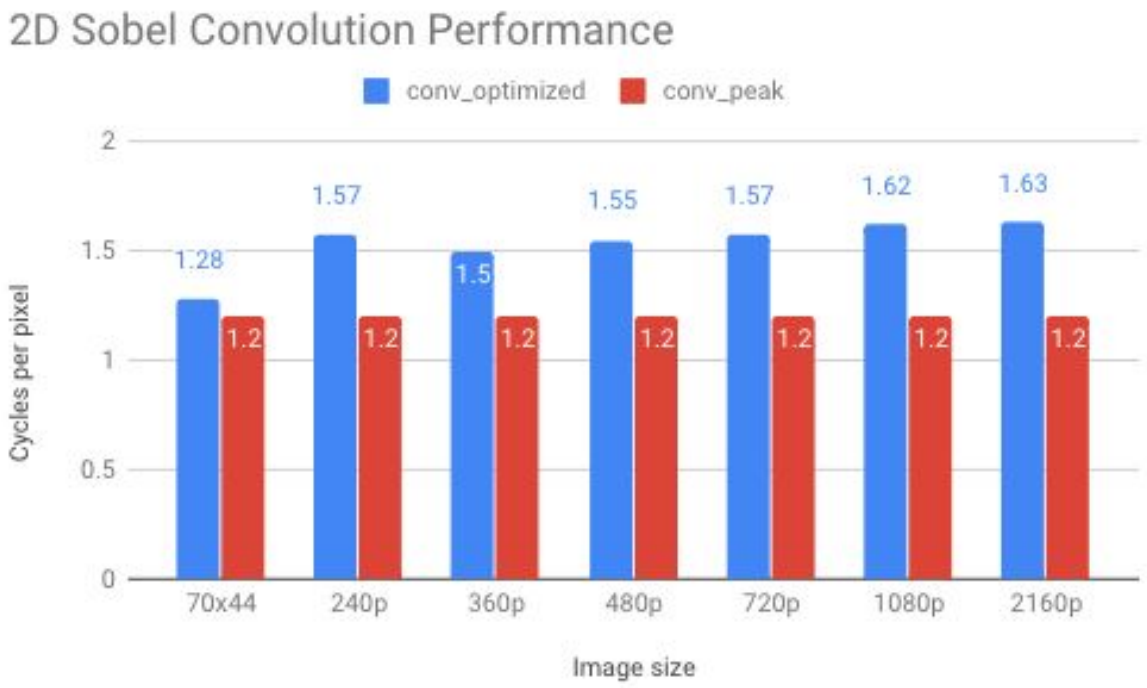


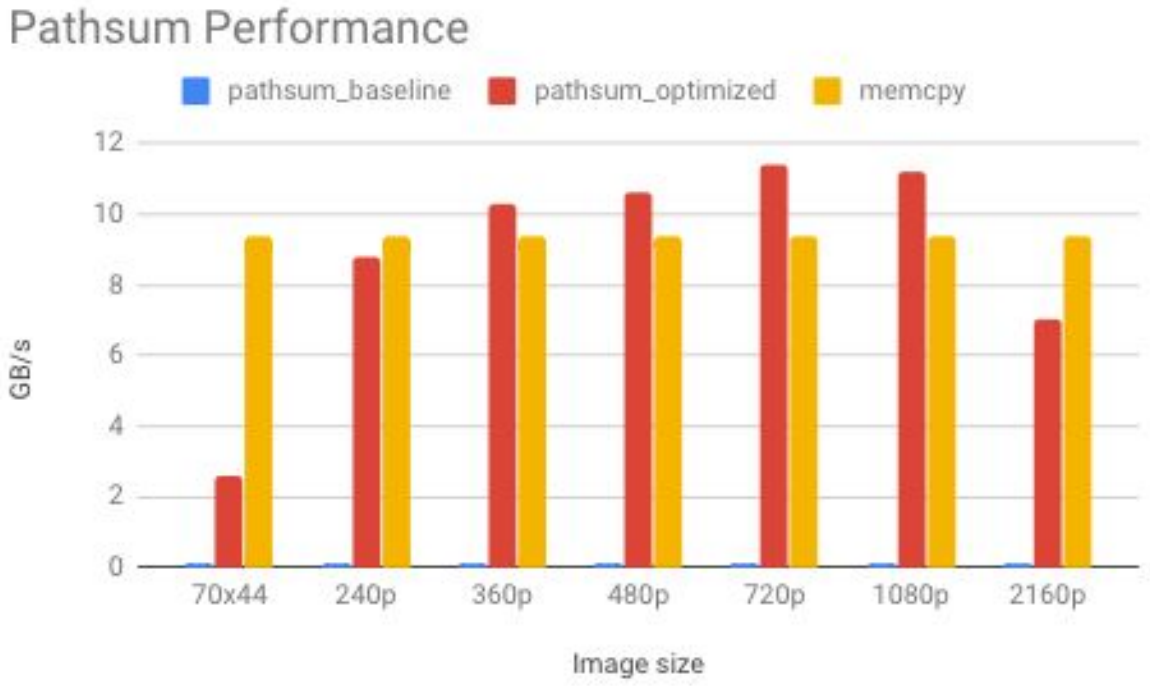
Optimization 2: Shift in the direction that results in the least movement. If a seam is near the left edge of the image, shift the left hand side of each row to the right by one pixel. If a seam is near the right edge of the image, shift the right hand side of each row to the left by one pixel. On average, ¼ of the image needs to be touched, a 2x improvement over the first optimization. This is why

buf_start needs to be kept around in the image structure.



Performance Plots




Pathsum Performance
pathsum_baseline
pathsum_optimized
memcpy
12
10
8
6
4
2
0
GB/s
70x44
240p
360p
480p
720p
1080p
2160p
Image size

Lessons Learnt

1. Don't trust the compiler

In our 2D convolution step, we saw that our vector integer multiplies were not being optimized into bit operations. We reformulated the kernel to overcome this.

2. Never above the theoretical peak

The theoretical peak we calculated in Phase 3 of our project were based on an approximation of the average width of the image as we removed seams: w=(old_width+new_width)/2. As such, our performance plots showed that we were sometimes above the theoretical peak. This happened when the number of seams removed was low. The theoretical peak we calculated did not reflect our optimizations in seam removal and the difference was exacerbated most when the number of seams removed was small. In our final phase, we took a different approach to calculating the theoretical peak - we segmented each operation of the algorithm and calculated the peak for that operation. As such, we got more accurate benchmarking plots and also allowed us to determine which operation could be tuned better (if step A is close to theoretical peak, focus on the other steps that are not).

3. Parallelization not free (overhead)

Using OpenMP operations like pragma omp does not guarantee faster performance. In many of our tests, it even decreased performance.

Improvements

In our proposal, we explained how seam-carving can be applied to both images and video. Given further time, we could expand our implementation into content-aware re-scaling for videos. Two proposals for how video re-scaling could be done are

1) Seam Carving on video frames

Video frames go up to a resolution of 4096x2160 at 60 frames per second. We can seam-carve each frame of the video and restitch modified frames. This is a more naive implementation of video-rescaling as it ignores the content present in the time-axis for videos.

2) 2D Seam Carving

Instead of removing 1D seams from 2D images we remove 2D seam manifolds from 3D space-time volumes. To achieve this, replace the dynamic programming method of seam carving with graph cuts that are suitable for 3D volumes. In the new formulation, a seam is given by a minimal cut in the graph and you can construct a graph such that the resulting cut is a valid seam.

Appendix

2D convolution instruction scheduling

ARIT LOAD	LOAD	LOAD	ARITH	SHIFT	STORE
0.0			0.0		
0.1	0.0		0.0	0.0	
0.2	0.1	0.0	0.0	0.0	
0.3	0.2	0.1	0.0		
0.4	0.3	0.2	0.0		
0.5	0.4	0.3	0.0		
0.6	0.5	0.4	0.0		
0.7	0.6	0.5	0.0		
0.0	0.7	0.6	0.0		
0.0		0.7	1.0	0.0	
1.0			1.0	0.0	
1.1	1.0		0.0	1.0	
1.2	1.1	1.0	1.0	1.0	0.0
1.3	1.2	1.1	1.0		0.0
1.4	1.3	1.2	1.0		0.0
1.5	1.4	1.3	1.0		0.0
1.6	1.5	1.4	1.0		
1.7	1.6	1.5	1.0		
1.0	1.7	1.6	1.0		
1.0		1.7	2.0	1.0	
2.0			2.0	1.0	
2.1	2.0		1.0	2.0	
2.2	2.1	2.0	2.0	2.0	1.0
2.3	2.2	2.1	2.0		1.0
2.4	2.3	2.2	2.0		1.0
2.5	2.4	2.3	2.0		1.0

3 left columns: The shared upcast load and vector integer ALU pipeline. Each output's 8 loads are represented by (0.0-0.7). The loads have a latency of 3 cycles, and throughput of 1 instr / cycle. The two 0.0s are two of the remaining arithmetic instructions.

ARITH column: The other fully saturated integer vector ALU pipeline. Each instruction here has a latency of 1 cycle. There instructions are independent enough to ensure no stalls here.

SHIFT column: The separate bit shift pipeline. There are only 4 shifts per output pixel, and shifts have a latency of 1 cycle, so this pipeline is not fully utilized.

STORE column: The memory write pipeline. Each write has a latency of 4 cycles. This pipeline is not fully utilized.

In the end, the operation is bottlenecked by the two ARITH pipelines. There is one 8-wide vector store completed every 10 cycles, corresponding to 1.25 cycles per pixel.

Pathsum calculation instruction scheduling (ideal)

L3 T0.5						L1 T0.33	L3 T1			L1 T0.5		L1 T0.5	L4 T1			
LD	LD	LD	LD	LD	LD	BL	PERM	PERM	PERM	MIN	MIN	ADD	ST	ST	ST	ST
0	0															
0	0	0	1													
0	0	0	1	1	1											
2	2	0	1	1	1	0										
2	2	2	3	1	1	1	0									
2	2	2	3	3	3		0	1								
4	4	2	3	3	3	2	0	1								
4	4	4	5	3	3	3		1	2	0						
4	4	4	5	5	5		3		2	1	0					
6	6	4	5	5	5	4	3		2		1	0				
6	6	6	7	5	5	5	3	4		2		1	0			
6	6	6	7	7	7			4	5	3	2		0	1		
8	8	6	7	7	7	6		4	5		3	2	0	1		
8	8	8	9	7	7	7	6		5	4		3	0	1	2	
8	8	8	9	9	9		6	7		5	4			1	2	3
10	10	8	9	9	9	8	6	7			5	4			2	3
10	10	10	11	9	9	9		7	8	6		5	4		2	3
10	10	10	11	11	11		9		8	7	6		4	5		3
		10	11	11	11	10	9		8		7	6	4	5		
				11	11	11	9	10		8		7	4	5	6	
								10	11	9	8			5	6	7
								10	11		9	8			6	7
									11	10		9	8		6	7
										11	10		8	9		7
											11	10	8	9		
												11	8	9	10	
														9	10	11
															10	11
															10	11
																11

LD = SIMD memory load (2 functional units, latency 3 cycles)
BL = SIMD blend (3 functional units, latency 1 cycle)
PERM = SIMD permute (1 functional unit, latency 3 cycles)
MIN = SIMD integer minimum (2 functional units, latency 1 cycle)
ADD = SIMD integer add (2 functional units, latency 1 cycle)
ST = SIMD memory store (1 functional unit, latency 4 cycles)

As shown above, the memory load functional unit is fully saturated by loading data from memory. The rest of the functional units have bubbles in their pipelines, meaning the load is the bottleneck. The memory stores complete at a rate of 1.5 cycles per store, so the theoretical peak is 1.5 cycles per output vector. Note this is an ideal case, where all loads have a latency of 3 cycles (ie. data is in L1 cache).

Pathsum calculation instruction scheduling (realistic)

L?? T0.5						L1 T0.33		L3 T1			L1 T0.5		L1 T0.5	L4 T1		
LD	LD	LD	LD	LD	LD	BL	BL	PERM	PERM	PERM	MIN	MIN	ADD	ST	ST	ST
0	0															
0	0	1	1													
0	0	1	1	2	2											
0	0	1	1	2	2											
0	0	1	1	2	2											
0	0	1	1	2	2											
0	0	1	1	2	2											
0	0	1	1	2	2											
0	0	1	1	2	2											
0	0	1	1	2	2											
3	3	1	1	2	2	0	0									
3	3	4	4	2	2	1	1	0								
3	3	4	4	5	5	2	2	0	0							
3	3	4	4	5	5			0	0	1						
3	3	4	4	5	5			1	0	1	0					
3	3	4	4	5	5			1	2	1		0				
3	3	4	4	5	5			1	2	2	1		0			
3	3	4	4	5	5				2	2		1		0		
3	3	4	4	5	5					2	2		1	0	1	
3	3	4	4	5	5							2		0	1	2
6	6	4	4	5	5	3	3						2	0	1	2
6	6	7	7	5	5	4	4	3							1	2
6	6	7	7	8	8	5	5	3	3							2
6	6	7	7	8	8			3	3	4						
6	6	7	7	8	8			4	3	4	3					
6	6	7	7	8	8			4	5	4		3				
6	6	7	7	8	8			4	5	5	4		3			
6	6	7	7	8	8				5	5		4		3		
6	6	7	7	8	8					5	5		4	3	4	
6	6	7	7	8	8							5		3	4	5
		7	7	8	8	6	6						5	3	4	5
				8	8	7	7	6							4	5
						8	8	6	6							5
								6	6	7						
								7	6	7	6					
								7	8	7		6				
								7	8	8	7		6			
									8	8		7		6		
										8	8		7	6	7	
												8		6	7	8
													8	6	7	8
															7	8
																8

In a realistic case, the memory loads have a latency that's much greater than 3 cycles. As shown above, the memory loads stall the remainder of the pipelines, so the actual peak performance is roughly equivalent to how quickly data can be brought in.