GPU Accelerated Seam Carving

Aditya Bhandaru (akbhanda)
Matt Sarett (msarett)

Overview of the Problem:

Seam carving is an algorithm for image resizing developed by Shai Avidan of Mitsubishi Electric Research Laboratories. More trivial methods of resizing images exist, such as cropping or just scaling down and resampling. Seam carving is a promising alternative because it is content aware. It functions by identifying seams, or paths of least importance, through an image. These seams can either be removed or inserted in order to change the size of the image. The purpose of the algorithm is to reduce image distortion in applications where images cannot be displayed at their original size.

Figure 1: Illustrating the Identification of One Seam

Figure 2: Illustrating the Removal of Many Seams

It is clear that seam carving is a promising method of image resizing because it is able to preserve the most important features of an image. This is not true of cropping or scaling as they omit or can resample out details respectively.

Literature and Related Work:

Seam carving is a relatively well-known algorithm that has been explored in numerous applications. Adobe Systems acquired a license for the software and implemented it as a feature of Photoshop CS4, calling it content aware scaling. GIMP, digiKam, ImageMagick, and iResizer also have released implementations of this technique.

Jacob Stultz provides an interesting approach to solving the seam carving problem in a paper from 2008. He first suggests the parallelization of the energy calculation. Essentially, the energy of each pixel can be calculated individually on separate processors. The only requirement is that each pixel must be able to access the values of adjacent pixels. This calculation only must be performed once because, after a seam is removed, only the energy of the adjacent pixels must be recalculated.

Stultz also suggests an approach to the minimum path calculation for each seam. Parallelizing the minimum path calculation is more complicated because each processor may depend on the results of other processors in order to continue its calculation. On the last pixel per row block, the processor must send a message to another processor to provide it with the result of the calculation. Stultz acknowledges that a bottleneck may occur while one processor waits on another, as the processors are not guaranteed to be synchronized.

Observations from various past courses at Carnegie Mellon provide support for these findings. In 15-211, seam carving was implemented and studied sequentially in Java. The energy calculations are trivial and occur on a per pixel basis. The minimum path calculation, however, is implemented in a similar manner to Stultz. While only one processor is utilized, bottom down dynamic programming ensures that each step of a potential path is only considered once. The result is saved and can be accessed by subsequent path calculations.

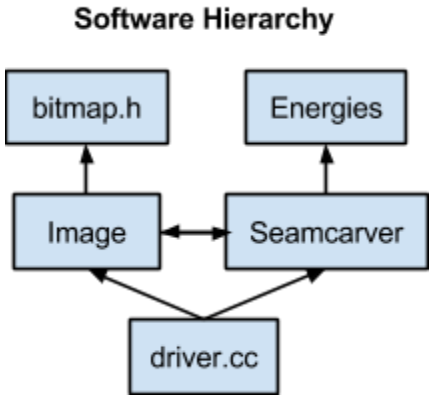
The parallelization of seam carving was explored further in 15-210. An advantage of this approach is that a purely functional programming language (SML) is used. This allows the programmer to fully expose the parallelism. The energy calculations are performed in constant span. The path calculations may also occur in parallel, but, as Stultz suggests, the sharing of information must be synchronized. It is important to note that while

programming in SML exposes parallelism, the code is not run on parallel hardware. Therefore, the potential performance gains were never realized.

We have discussed some advanced approaches to implementing seam carving, each of which takes advantage of the parallel nature of the problem. However, the massively parallel capabilities of the GPU and additionally its ability to synchronize threads suggest that this platform will allow for a superior approach to the problem.

Detailed Description of Techniques:

Baseline Sequential Implementation



For comparison and correctness, a sequential implementation of seam carving in C++ has been written. A `Driver` class will utilize `Image`, `Energy`, and `SeamCarver` classes to load an image, shrink it horizontally, and output the shrunk image to a file.

The first step of the sequential implementation is to load the image from a file. Images are stored as bitmaps, so they can be considered as a two dimensional array of pixels. The image is loaded and saved in the `Image` class. The `width` and `height` are also identified as properties.

In the next step, the energy of each pixel is computed in the `Energy` class. A gradient calculation is used for energy. A difference between two pixels is calculated as the sum of squared differences between the RGB components. The gradient of any given pixel is equal to the sum of its difference from its bottom and its right neighbors.

After computing the gradients, the next step is to identify the vertical path of minimum importance through the image. By removing the pixels in this path, the image can be shrunk horizontally. The identification and removal of this seam occurs in the `SeamCarver` class. In fact, this class can remove a variable number of seams from the image, as instructed by the `Driver`.

The minimum vertical path is computed as follows. First the gradient table is converted to a minimum cost table. The minimum cost of any given pixel is equal to its gradient value added to the minimum of the gradient values of the three adjacent pixels from the above row. The creation of the minimum cost naturally proceeds row by row from top to

the bottom of the image. At the conclusion of the creation of the minimum cost table, the bottom pixel of the minimum seam is simply the pixel in the bottom row with the minimum cost value. After this pixel is identified, the seam can be built by tracing up through the image. The minimum of the three adjacent pixels from the above row is also a member of the minimum seam.

Figure 3: Calculation of Minimum Cost Table

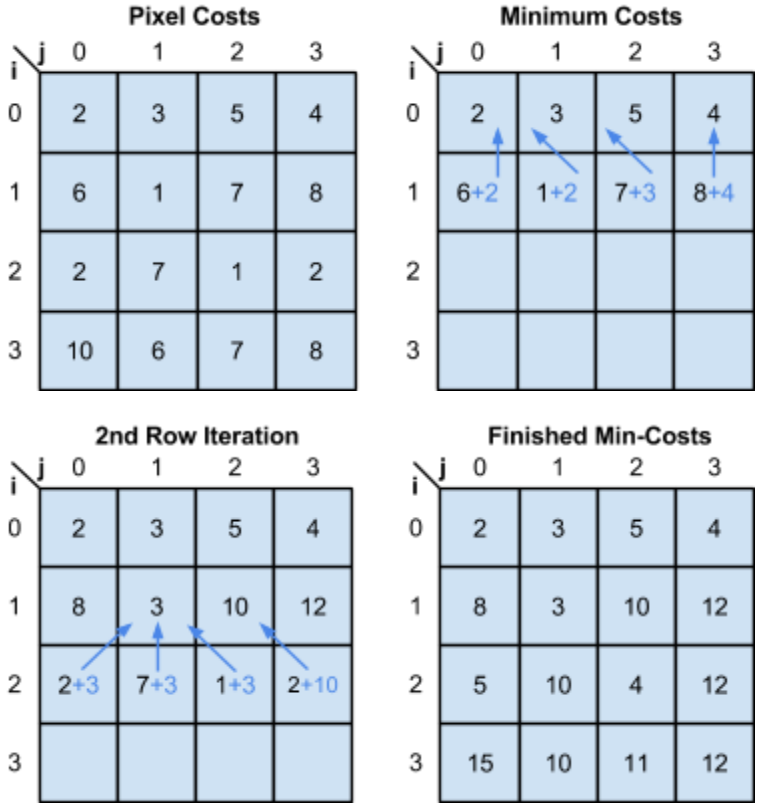
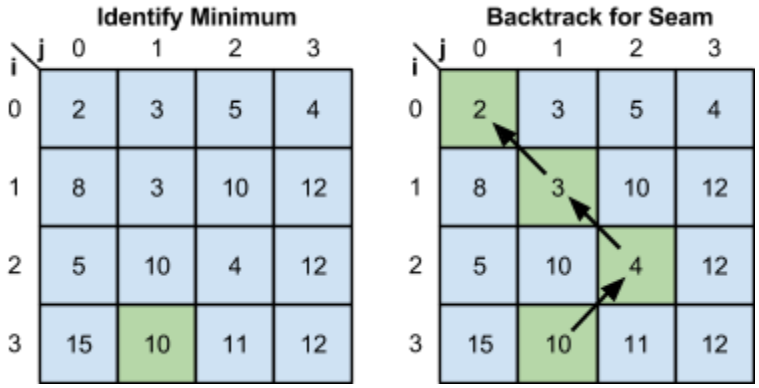


Figure 4: Identification of Minimum Seams



It is worth noting that this implementation is an example of bottom up dynamic programming. The minimum cost of each location in the image is computed only once and used by multiple pixels in the row below. This is also a good example of the use of backtracking to construct the solution to a problem.

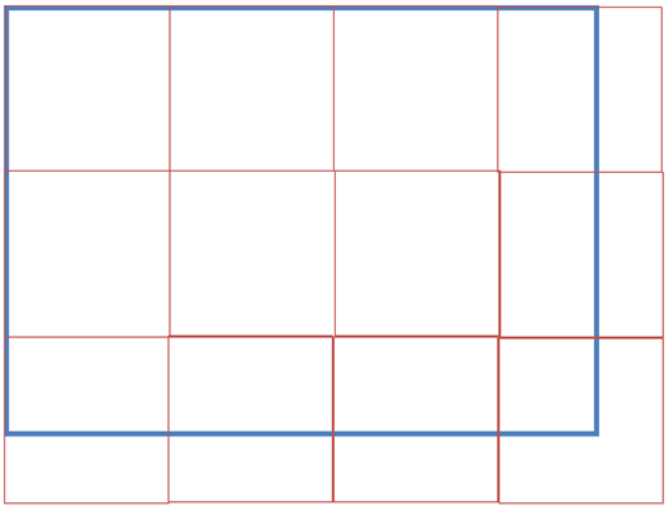
Finally, the resized image is saved to a file utilizing a method in the `Image` class. This is the conclusion of the algorithm.

Optimized GPU Implementation

GPU Optimization 1: Compute Energies on the Kernel

In the first optimization, the CUDA kernel is utilized to compute the energy of each pixel. A single thread is responsible for computing the energy of each specific pixel. The kernel function is called with 32 x 32 thread blocks. Enough blocks will be created to provide a thread for each pixel in the image.

Figure 5: A Depiction of How 32 x 32 Thread Blocks Will Cover a Variably Sized Image



To compute the energy of a pixel, the pixel values from below and to the right must be fetched and compared to the value of the current pixel. To avoid costly DRAM operations when fetching the pixels, each pixel will be loaded into shared memory by the thread that will be responsible for computing its energy. After synchronizing the threads, each thread can obtain the values it needs from the shared memory array. Boundary checks are implemented for threads on the edge of the block that may need to access neighboring pixels through DRAM.

After computing the energies, the resulting array is copied back from device memory to host memory.

GPU Optimization 2: Compute Minimum Cost Table on the Kernel

Another kernel function will be used to compute the minimum cost table. The minimum cost table is a representation of the cost of taking potential paths through an image. The minimum cost of each pixel is equal to its energy added to the minimum of the costs of the three neighboring above pixels. This is naturally an operation that occurs row by row.

In the call to this kernel function, threads will be created for each column. The minimum cost table will be created row by row, with each thread computing the minimum cost for one pixel in the row. Since each row of the minimum cost table depends on the above row, the threads will be synchronized after each row. The simplicity of this synchronization is a significant advantage of using the GPU.

While each row of the minimum cost table will be stored in DRAM, the previous row of the minimum cost table will be maintained in a shared memory vector array. This will ensure that, when using the above row to compute the next minimum cost, fetches are from fast shared memory rather than slow DRAM.

After the computation is complete, the resulting array is copied from device memory to host memory.

For simplicity, our implementation of this optimization only support images with widths that are less than 1024 pixels. The results of the experimentation suggest that this implementation could be extended for images of any size and still obtain significant speedup.

GPU Optimization 3: Performing a Minimum Reduction on the Kernel

After computing the minimum cost table, the minimum value in the bottom row of the table must be located. Starting at this pixel, backtracking will be used to discover the minimum seam in the image. This is the seam that will be removed.

Instead of searching for the minimum linearly, a minimum reduction will be performed on the GPU to identify the bottom of the minimum seam. The bottom row is stored in shared memory on GPU to ensure that the operations necessary for the reduction occur

without costly fetches to DRAM. It is important to note that this reduction must track both the minimum value and the index of that minimum value.

Once the minimum index has been determined, it is copied back to the host for use in backtracking the minimum seam.

Analysis of Results:

All tests were performed on the Gates-Hillman Center cluster machines. In particular, we used ghc71.ghc.andrew.cmu.edu. This machine has an *NVIDIA GeForce GTX 480* graphics card with 1024 threads per block and 48KB of shared memory. The full specifications for this GPU can be found on pages 180-183 here:
http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Correctness Methodology

The following figures show actual seam identification and removal from our implementation. All of the resulting images are identical for the sequential and CUDA implementations.

To check for correctness, we compared the binary image outputs of each implementation and made sure they were correct within some degree of tolerance. This error is to account for the various floating point operations used when computing and comparing energies. When rewriting the sequential code for use in a CUDA kernel, we changed the order of some operations, and therefore knowingly introduce a source of floating point error.

Figure 6: The Original Image (Width = 512 pixels)

Figure 7: Identification of a Single Minimum Importance Seam

Figure 8: Identification of 100 Minimum Importance Seams

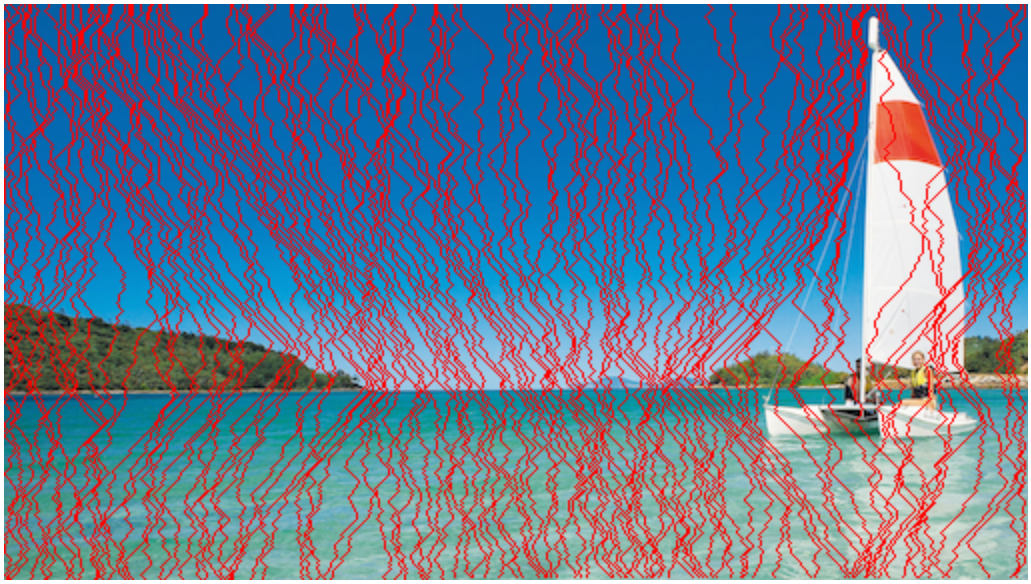


Figure 9: Removal of 100 Seams (Original Width = 512 pixels)

Figure 10: Removal of 200 Seams (Width = 312 pixels)

Performance Evaluation Methodology

The tables and graphs below display the speedups we obtained based on seams removed and image size. All times are measured in CPU cycles provided by the standard `ctime` library.

Table 1: Performance Gains as a Function of Number of Seams Removed

Num Seams	Sequential	Cuda	Speedup
	Avg. (cycles)	Avg. (cycles)	Normalized
1	80000	53333	1.500
50	4416667	586667	7.528
100	8863337	1113333	7.961
200	15906667	2093333	7.599
400	29216667	3793333	7.702
800	44380000	6103333	7.271
Average			6.594

The above table shows the average number of cycles each implementation takes to remove a variable number of seams from a 1024px by 576px image. You will notice that as we amortize over the number of seams removed, we realize substantial gains in performance.

Table 2: Performance Gains as a Function of Image Size

Image Size	Sequential	Cuda	Speedup
(px width)	Avg. (cycles)	Avg. (cycles)	Normalized
256	720000	250000	2.880
384	1916667	396667	4.832
512	3793333	626667	6.053
768	8916667	1283333	6.948
1024	15906667	2093333	7.599
Average			6.662

The above table shows the average number of cycles each implementation takes to remove 200 seams from a variably sized image. We see substantial performance gains even for small images, and much more compelling results for the larger ones.

Figure 11: An Analysis of Speedup Based on Number of Seams Removed

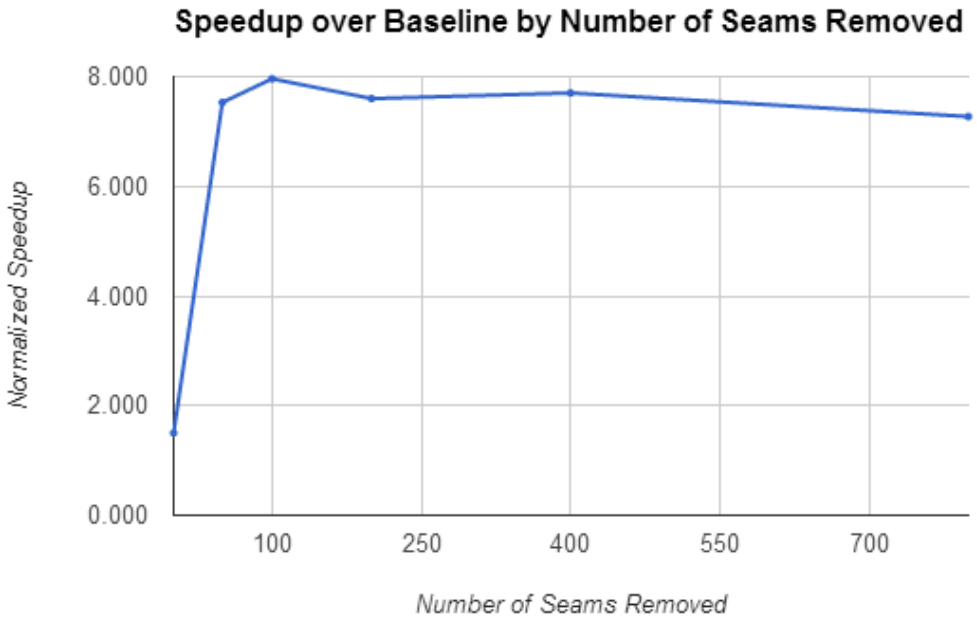
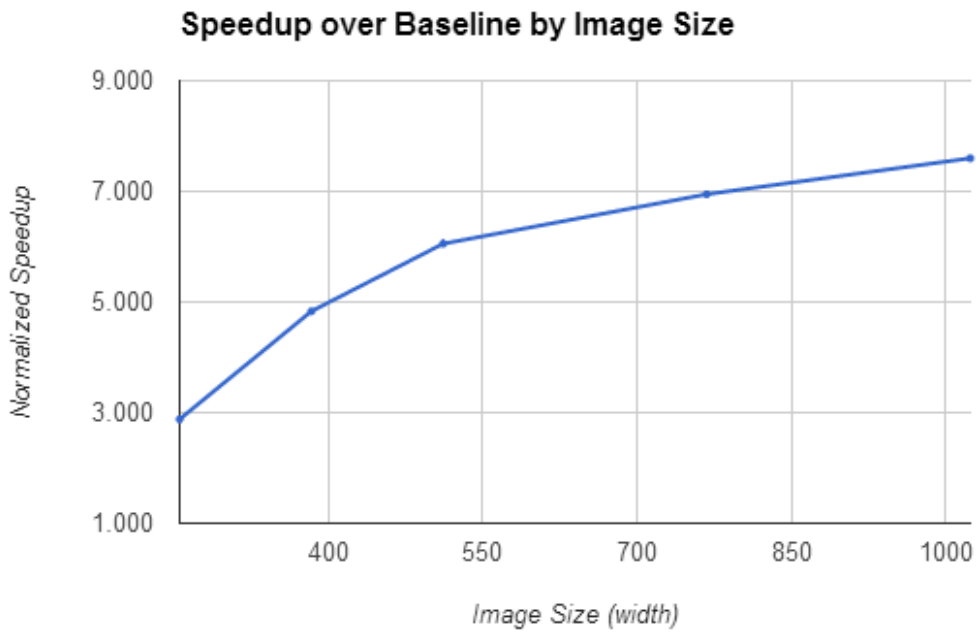


Figure 12: Analysis of Speedup Based on Image Size



Conclusion:

It is clear that by parallelizing three operations on the CUDA kernel, we can obtain significant performance improvements on a sequential seam carving implementation. The speedup is generally within the 2-8x range with it approaching 8x as the image size or number of seams removed increases.

It is not surprising that the GPU implementation has a significant performance advantage over the sequential implementation. We have matched a massively parallel problem with a platform that allows us to exploit this parallelism. This a strong demonstration of the power of CUDA and the importance of parallel computing.

Although resizing and cropping are both faster alternatives, they do not preserve the details in the image. An optimized version of the seam carving makes a compelling case of the viability of content-aware image resizing in high performance applications.

Future Work:

Further optimizations could be made by maintaining memory on the GPU device.

Upon examining our kernel methods, it is clear that the first two both utilize the energies table and the last two both utilize the bottom row of the minimum cost table. However, in each call to the kernel, memory is allocated, copied, and freed from the host. A speedup could be obtained by maintaining memory on the kernel. This optimization was not implemented because the class structure, organization, and readability of our program would be negatively affected by sharing these pointers.

The code can also be improved by avoiding a full recomputation of the energies and minimum cost tables after removing a single seam. Only energies and minimum costs that neighbor the removed seam must be recalculated.

Work Cited:

Avidan, Shai. "Seam Carving for Content-Aware Image Resizing"
http://www.win.tue.nl/~wstahw/edu/2IV05/seamcarving.pdf

Stultz, Jacob. "Seam Carving: Parallelizing a novel new image resizing algorithm"
http://beowulf.lcs.mit.edu/18.337-2008/projects/reports/stultz-6338.pdf

Seam Carving. Wikipedia. http://en.wikipedia.org/wiki/Seam_carving

15-210 Dynamic Programming Lab. https://autolab.cs.cmu.edu/15210-f13/dplab

15-211 Seam Carving Lab. http://www.cs.cmu.edu/

Appendix:

The source code repository can be found here:
https://github.com/abhandaru/gpu-seamcarving.