# FPGA Accelerated Seam Carving for Video

**Eshani Mishra, Shruti Narayan, Kimberly Lim**

Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—**Traditional methods of video resizing such as cropping or scaling distorts important regions in the frame, whereas content-aware retargeting preserves these areas and thus, creates more aesthetically pleasing results. Seam carving is a popular technique for content-aware retargeting. We illustrate video retargeting using an improved seam carving that selects 2D seams from 3D voxels. Since computational complexity bottlenecks the execution speed, we present a hardware-oriented approach and algorithmic modification to improve performance. The performance of the algorithm is evaluated on an FPGA board, which shows an improvement of 13.67x. We also present heuristics for video-retargeting based on popular video-quality metrics**.**

***Index Terms*— FPGA, Memory, Seam Carving**

## I. Introduction

The use of portable devices is rapidly expanding – in 2019, the number of mobile phone users is forecasted to reach 4.68 billion. Because of the various display aspect ratios (DARs) for these devices, images and videos must be adjusted to fit the screen. Traditional methods of video resizing include scaling and cropping, but these methods either remove important contents completely (cropping) or distorts the entire screen content (scaling). Thus, the resultant video becomes undesirable. The typical case of scaling with additional black-colored backgrounds does preserve the original images' contents, but for portable devices with small displays, the resultant images become so small that is hard to see. For users with restricted vision, this approach goes against technology accessibility as it may produce images that are impossible for them to see. Furthermore, video retargeting allows for better information delivery by drawing attention to important contents in the video through the reduced focal region.

Avidan [1] proposed a novel method coined Seam Carving for content-aware image resizing algorithms. The approach uses seams - monotonic and connected paths of pixels going from the top of the image to the bottom, or from left to right. Dynamic programming can be used to determine the path of least energy importance and this seam is then removed to reduce the image size. This can be done to reduce width or height as well as add dimension by duplicating the seam of least importance. A naive extension of seam carving to video is to treat each video frame as an image and resize it independently as shown in Fig 1. c. This creates jittery artifacts due to the lack of temporal coherency, and a global approach is required.

To process video, Avidan [3] proposed an improved seam carving operator, *static-seam*, in *Improved Seam Carving for Video Retargeting* by replacing the spatial energy map in favor of a global energy map consisting of both temporal and spatial elements to determine 2D seam manifolds from 3D voxel volumes that are fully connected. However, the computational complexity become the bottleneck of the implementation. For example, on an Intel(R) Core (TM) i5-4260U CPU @ 1.40GHz, removing 20 vertical seams from a video of resolution 640x360 and total frames of 43 took 15.7018 seconds.

We implemented a hardware-oriented approach for seam carving on the DE-10 Standard FPGA to increase performance speed by leveraging on the high number of functional units. In our design report, we proposed a modification on seam-carving that is to remove the multiple best seams during each run of the algorithm instead of only the optimal seam. However, we found this was no longer conducive for our design specifications; this is discussed further in II. Design Requirements. We targeted an improvement of at least 5x compared to a C++11 implementation based on Avidan's *static-seam* approach and positive user testing results for selecting output videos from both implementations. We achieved our targets for speed, user testing and PSNR, but were 2% shy of our SSIM metric.

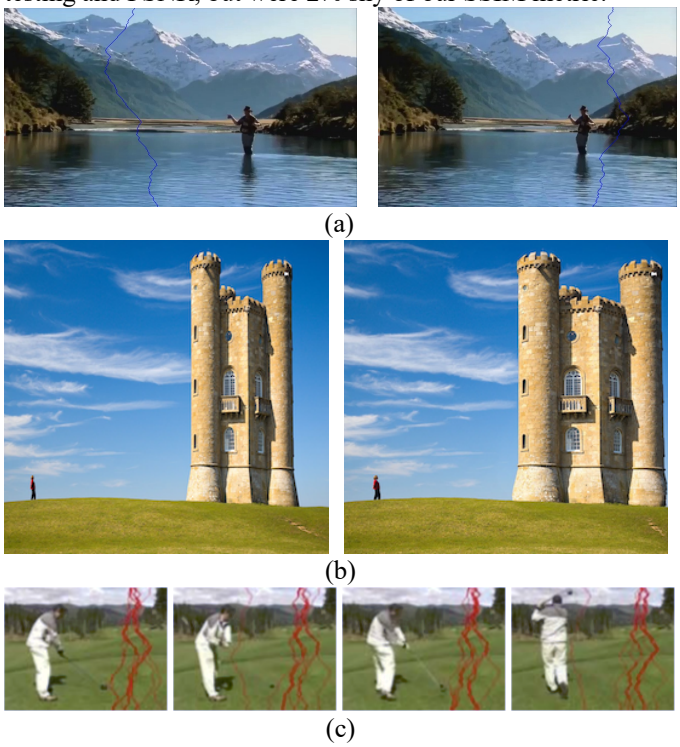


Fig. 1. (a) Left: An extracted seam. Right: An example of a resized image by using seam carving. (b) Left: Scaling. Right: Seams. (c) Seam carving on each video frame independently creates locally optimal seams that can be totally different across video frames. This creates a jittery resized video. A video resized using this method can be seen here: https://www.youtube.com/watch?v=Qb-l4ZWI8qc

## II. Design Requirements

We focus our design requirements in terms of 3 aspects: utility, timing and video quality.

### A. Utility

We meet our requirements if we able to resize any user-supplied video within the constraints to the correct specified resolution and this resized video is viewable on a monitor. We constrain our video input resolution to 128x128. This is because of the constraint of the amount of memory available on our FPGA; In our design report, we had targeted 256x144 but our hardware design was not able to fit onto the board. We will discuss this further in III. Design Trade Studies. We illustrate this memory mapping in III Architecture And/Or Principle of Operation.

We originally constrained the resolution of the resultant video to be a minimum of half the width of the original video to preserve viewability of the result video on the monitor. However, this constrain no longer applies because we changed our design to remove one seam at a time, and the user can rerun the program to remove any desired number of seams from the video.

### B. Timing

We meet our requirements if we get an improvement of at least 5x in terms of running time (seconds) when compared to a C++11 implementation based on Avidan's static-seam approach based on our approach. The transition from C++ to a FPGA implementation allows parallelization of computation over the columns of pixels and this is dependent on the resolution of the image. This would imply that we are able to compute 128 columns in parallel and this would be an improvement of 128x but this does not account for the time required for data transfer between the HPS and FPGA, differences in how functional units are used in the CPU vs FPGA, optimizations done by the CPU compiler vs FPGA compiler, communication for parallel computation in the FPGA and so on. Thus, we aim for an improvement of at least 5x to be err on the conservative.

### C. Video Quality

We meet our requirements if the result video preserves content better than the alternative cropped, scaled, or low-resolution versions. We will verify this with user testing and video quality metrics.

We designed our user testing as a double-blind experiment where one teammate sets up the study but then has another teammate collect the data from participants. We will show 3 videos to a participant: 1) The original video 2) Result video from static-seam C++ implementation and 3) Result video from FPGA implementation. We did not tell the participants which video came from which implementation. We then asked the participants to choose between 3 options: 1) video 2 chose a better seam, 2) video 3 chose a better seam, or 3) the seams in videos 2 and 3 and indistinguishable. We decided we will pass our user testing requirement if over 90% of participants choose either option 2 or option 3.

Our specification for video quality was that we wanted a processed result of similar quality to the result of our software benchmarking code. We wanted to use two objective video quality metrics to test our results, peak signal to noise ratio (PNSR) and spatio-temporal SSIM. Both of these video quality metrics are used to compare the amount of distortion in a processed video compared to the original video. PSNR estimates absolute errors and spatio-temporal structural similarity index (SSIM) measures similarity in luminance, contrast, and structure of pixels. We calculated these values using FFmpeg. For our benchmark values PNSR and SSIM values, we compared the resulting video from our software implementation of seam carving to the original video. Then we compared the resulting video from our FPGA implementation of seam carving to the original video and calculate PNSR and SSIM values for this comparison. Therefore, we have two sets of PNSR and SSIM values. We will compare the PNSR and SSIM values from the FPGA implementation of seam carving to the PNSR and SSIM values from the software implementation of seam carving. Our goal was to have at most 10% difference in the quality metrics from the two implementations for SSIM. We will run PSNR on the result video and we meet requirements if we are within 80dB of the PSNR of the original video.

## III. Architecture and/or Principle of Operation

We build on and extend the work of Avidan [3]. The general approach of the static seam carving algorithm is 1) compute the energy map, 2) compute all path sums of seams and 3) find the minimum path seam. These 3 steps are repeated as many times as number of seams desired to be removed.

We propose multi-seam removal for each global energy map computed to increase throughput. The number of seams removed serves as a parameter that balances throughput and video result quality. The higher the number of seams, the higher the throughput as the number of times the entire algorithm is ran is reduced. However, it is important to note that removing the top 5 minimum path seams from 1 energy map is not the same as selecting the minimum path seam from 5 energy maps computed consecutively like in the original seam carving approach. After synthesizing our FPGA implementation onto the DE10 Standard, we achieved a 13.67x performance speed relative to the C++ implementation, most of which was bottlenecked by data transfer. Thus, we no longer found multi-seam removal to fit our design specification as speedup would be minimal compared to speeding up data transfer and we would still be losing video quality. We explain this tradeoff further in IV. Design Trade Studies.

Our minimum viable product will reflect the following. Five seconds of input video of resolution 128x128 and 30 fps with total number of frames=300 will be taken externally and passed into the DE10-Standard file system. The HPS on the DE10-Standard will handle video preprocessing such as gray-scaling and conversion to hex files through scripts run on Linux. The video will be stored as a modified (see subsystems for details) array in the SDRAM, accessible by the FPGA through AXI bridges. The FPGA will load workable frames into embedded memory (M10K blocks) and run the three stages of the seam carving algorithm. It will find the indices of pixels in the best seams and send this to the HPS, which will run post-processing

scripts to remove these pixels from the frames and display the video onto a monitor. A user can toggle between the three output options on the monitor – the original video, the video with the identified seam highlighted, and the video with the seam removed.

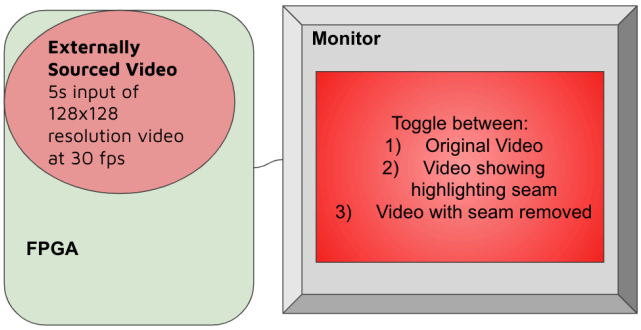



Fig. 2. User Perspective of Minimum Viable Product

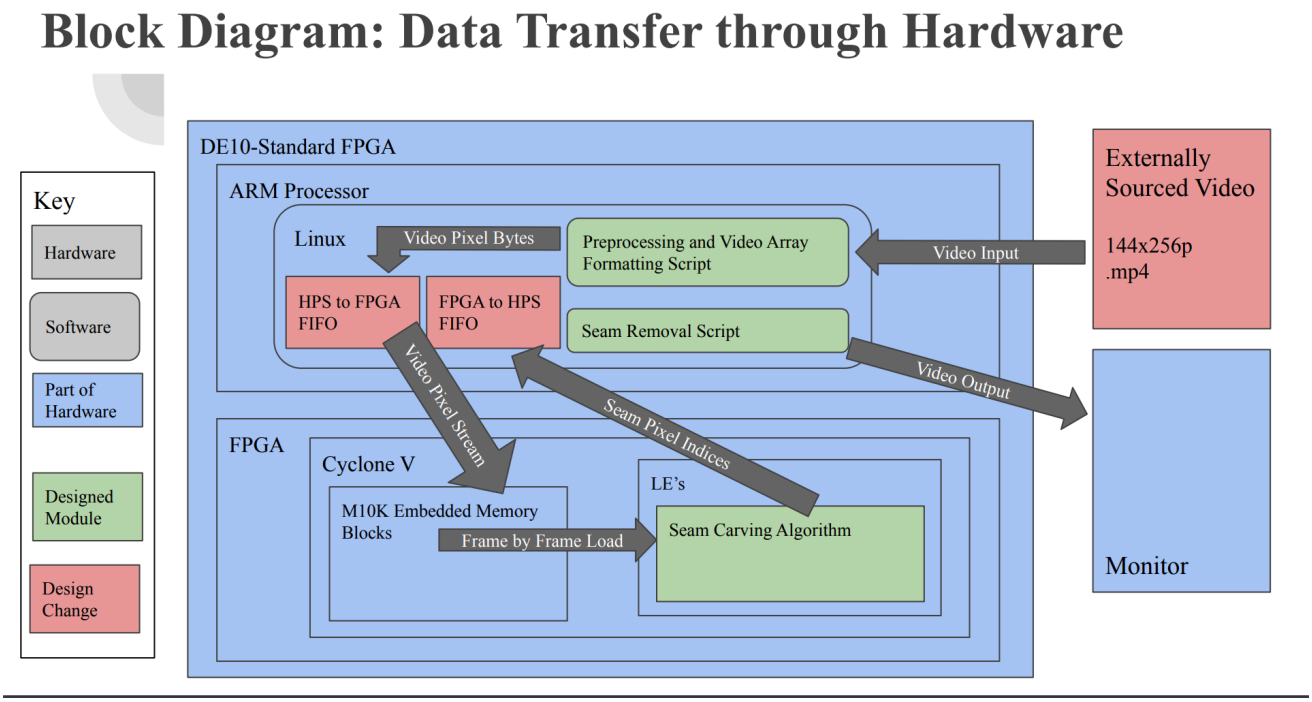



Fig. 3. Block Diagram of our Data Transfer

## IV. Design Trade Studies

### A. Timing Performance vs Video Quality

In our previous design report, we proposed multi-seam removal for each iteration of the algorithm. The multi-seam removal would have given n times the throughput where n is the number of seams removed. Our proposed testing method is shown below in Fig 4, however this was not used in our final testing. This produces a higher throughput as we skip energy map calculations that are computationally heavily. This increased performance of speed is not free as we are not selecting the optimal seam for every single pixel reduction in width (or height).However, after testing our FPGA implementation for single seam removal, we realized that our implementation was already so much faster (13.67x) than our baseline. Thus, sacrificing video quality for speed was no longer necessary for us to reach our performance projections.

Another reason is that our performance was being bottle-necked by the data transfer between the HPS and FPGA and not the algorithm itself. We would have gotten much better performance by leveraging the full width of the AXI bus between the FPGA and HPS. Currently, we send 8 bits of data which is composed of the 0-255 value of a grayscale pixel. The bus is capable of sending 128 bits. This change would involve packing 16 pixels (128 bits / 8 bits) in the HPS through C++ and updating the finite state machine on the FPGA to handle 16 pixels at a time from the pixel stream from the HPS.

| Metrics | Number of Seams per Algorithmic Run (NSAR) | | |
|---|---|---|---|
| | *1* | *3* | *5* |
| Quality | High | Medium | Low |
| Execution time | Longest | Medium | Shortest |
| User Testing | Most Satisfied | Medium Satisfaction | Low Satisfaction |

Fig. 4.

### B. FPGA Memory Allocation

The algorithm bottleneck encountered in software is the high volume of data and computation needed for the full video. Implementing the algorithm on hardware will increase parallelizing capability, but we are still limited by the amount of data that can be fit on the FPGA. The full video can be stored frame by frame in the HPS file system, and sent over to the FPGA side in a serial stream (row by row). The most computationally heavy part of the algorithm is in stage 1, calculating the energy map – to maximize parallelizing capability, we would want to process each column at once and sequentially process rows. If we store the frames in embedded memory, each column must be stored in its own separate block to be able to have read/write access to all of them at once. With our given frame size of 128x128p, a column more than definitely fits into a M10K block (which can hold 1024 bytes). The DE10-Standard only has 557 M10K blocks of embedded memory – this means we can create a "bank" of 128 blocks, each holding a column of a video frame, to fully store the video, as well as other necessary arrays in the algorithms (energy map, accumulation paths – which are the size of a single video frame).

We choose to instantiate two "banks" of this nature to hold frames to process on – in stage 1, we will need at most this many at a time. (Quartus analysis revealed that the memory blocks and more importantly, the logic (ALM) utilization was full for this frame size). In stage 1, bank 2 will be the most used – both the spatial and temporal energy maps use 16 bit data points rather than 8 bit, and since they are stored together in the same bank, each corresponding pixel will require 32 bits – this means there are 256 spots in each memory block, still more than enough for a 128 pixel column. The accumulation values are only needed for a single row at a time, so these can be stored in a set of 128 registers.

Figure 5 further explains each stage's use of the two instantiated "banks."

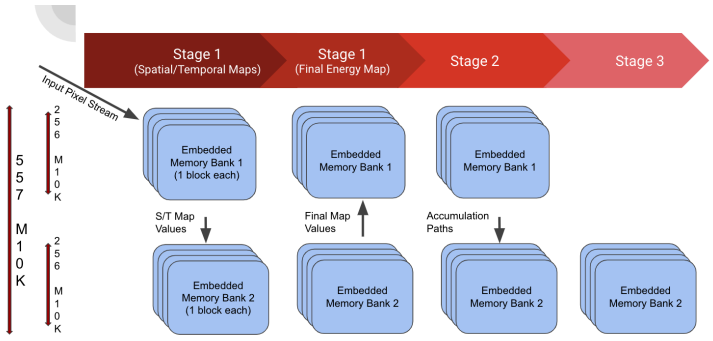
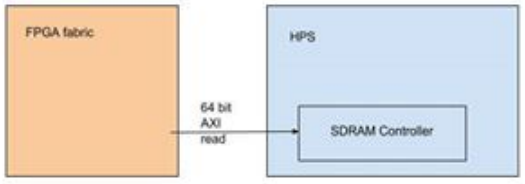



Fig. 5. Embedded Memory Allocation

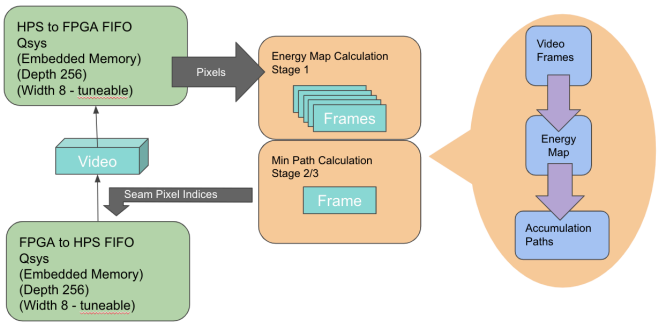
## V. System Description

In our design report, we intended the FPGA to SDRAM interface on the DE-10's FPGA to read the video array from the HPS's SDRAM into the FPGA's memory. This data transfer is done using 2 64 bit read ports in a master-slave architecture to allow the FPGA's peripherals to access the HPS's SDRAM.



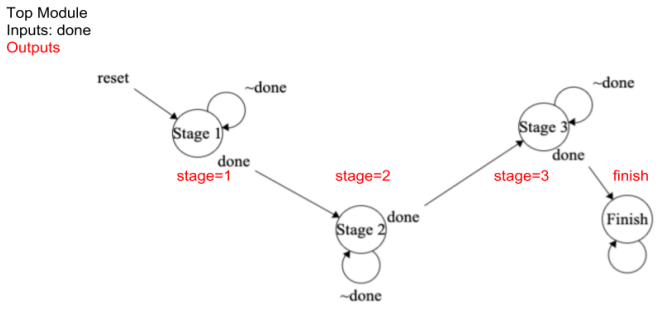

In our original design, we stored video data in the HPS's SDRAM. However, we realized this was not the most optimal solution because data in SDRAM does not persist when the FPGA is turned off. Therefore, we decided to store our video, as well as all of its frames as images on the Linux filesystem of

the HPS. We decided to use FIFOs to communicate between the HPS and the FPGA.




Our current design uses a C program, which runs on the HPS, to read from the frames of the video on the filesystem, and write each pixel onto the HPS to FPGA FIFO. This write is done using a 32-bit AXI write using memory mapped bridges on the DE-10 Standard that are used to communicate between the HPS and the FPGA. We set up these connections on the board with the use of QSYS, a development tool that handles the FPGA's interconnect with external modules. Once the pixel data is in the FIFO, we use a state machine to read from the FIFO and send a pixel to the top module each time the top module sends a data ready signal.

The top module of the FPGA design will handle switching between stages - within a stage its corresponding FSMs will be triggered and when the stage calculations are complete the done




signal will move the top FSM to the next stage. Stage One will handle receiving the incoming pixel stream and Stage Three will send back the pixel indices of the seam(s). The Finish state will indicate the completion of the algorithm on the FPGA end.

Recall that the algorithm is separated into 3 stages.

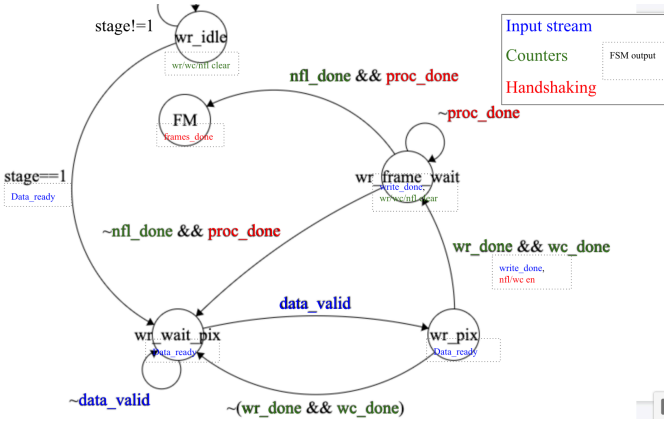
The first stage involves computing an energy map of the pixels (the size of a frame), with both a spatial and temporal aspect. The spatial energy map will be calculated for each frame (across x and y), and will give high energy values to pixels that have most difference (edges, etc). For each value of a pixel over time, the largest spatial energy calculated will be kept for the final energy map. The temporal aspect will involve looking at the difference in a single pixel value over time (across z, or t for time). The final energy map is a weighted sum of the two.

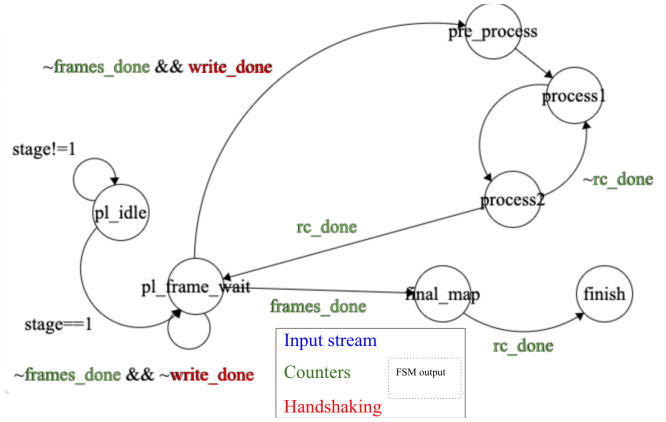
The second stage is an accumulation stage, in which we generate an accumulation matrix. The energy value of a pixel is added to the minimum of the top three adjacent accumulation values to find the current pixel accumulation value, and this process is iterated over rows (edges will give an accumulation value of 0). This path of minimums represents a seam, and thus all the possible seams in the video will be found in this stage.

The final stage will involve picking the minimum value(s) of the end accumulated row and following back on the path for the given seam to remove. The resultant index of each pixel in this seam will be sent back as found.

### *A. Stage 1*

Stage 1 involves two FSMs for the double buffering - one to process the current loaded frame and the other to handle loading the next frame block into the embedded memory. The following diagram outlines the control signals needed to do handshaking between the FSMs. (wr_ indicates the loading FSM and pl_ indicates the processing FSM).




Regarding the processing:

We calculate the temporal and spatial energies in parallel, using a synchronized pipelined data calculation and transfer that will store the final values for each index at the same time. The pipeline will have an initial latency, but once that flush of values occurs energy map values will be stored every other clock cycle (on a two cycle pattern for alternate read and write states – the map values are constantly

As mentioned previously, we will store a column per embedded memory block, using bank 1 for our initial data read. Bank 2 will store the spatial and temporal map in tandem – the pipeline latency is outlined by the grey dotted lines to show the synchronous calculation of each value for the same pixel index.

To calculate the spatial map, we will need to apply a Sobel filter over the x and y axes and take the norm (or the sum of the absolute values, since this is a cheaper operation).

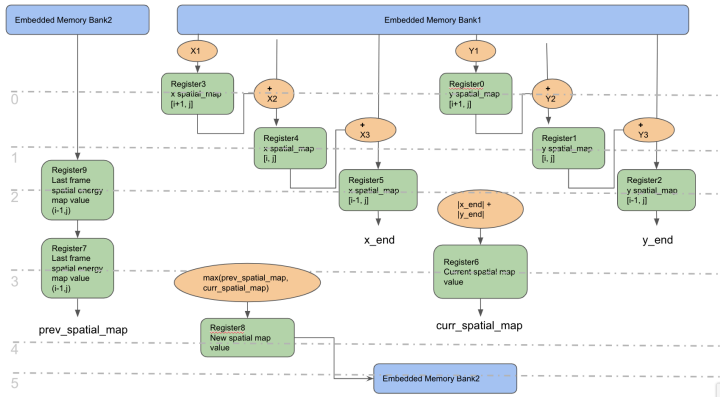
Pixel value [i,j] 0<=i<128 0<=j<128
Y1: y spatial_map [i-1, j] += image[i, j-1] + 2*image[i,j] + image[i, j+1]
Y2: y spatial_map [i, j] += 0
Y3: y spatial_map [i-1,j] += -image[i, j-1] - 2*image[i,j] - image[i, j+1]

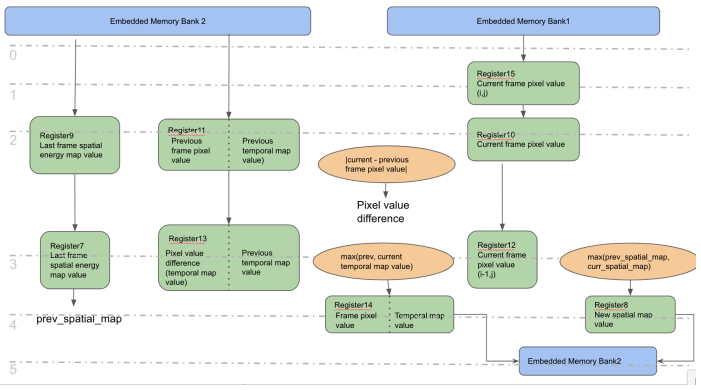
X1: x spatial_map [i-1, j] += -image[i, j-1] + image[i, j+1]
X2: x spatial_map [i,j] += -2*image[i, j-1] + 2*image[i, j+1]
X3: x spatial_map [i-1, j] += -image[i, j-1] + image[i, j+1]

The equations represent the partial calculations of the Sobel kernel if we only read 3 bytes at a time. These will be calculated through a modified pipelined set of adders, as depicted in the following diagram.

The final norm will be compared to the stored best spatial energy value for the pixel over all previous frames, and then the maximum of those two from the comparison will be stored in the spatial map representation in memory.

The temporal energy map is calculated as the largest pixel value difference between frames - so a running comparison of the best differences will be calculated for each pixel each frame. To do this a record of the previous pixel value needs to be stored, as well as the thus far largest difference (which will be the final temporal energy for a given pixel). The pipeline for this value calculation is coordinated with that of the spatial so that the value being stored at every clock tick corresponds to the same pixel (and address).

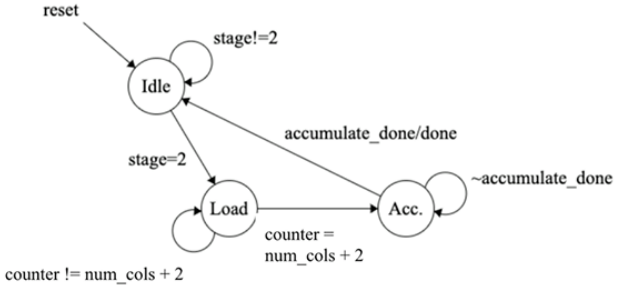


Once both the spatial and temporal maps are found, the processing FSM will enter its final state in which a weighted sum of the two energy maps will be calculated and stored in embedded memory as the final energy map. The weight value will be a tunable parameter.

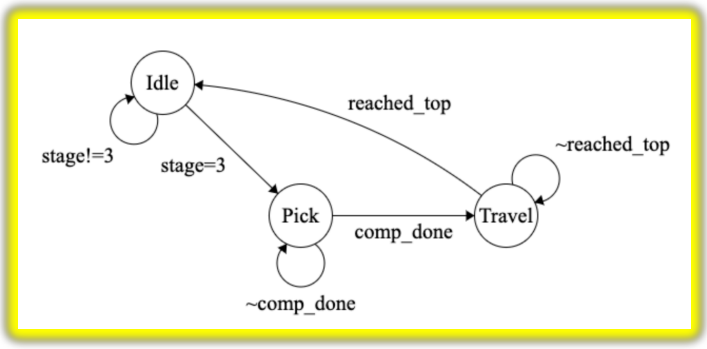
### *B. Stage 2*

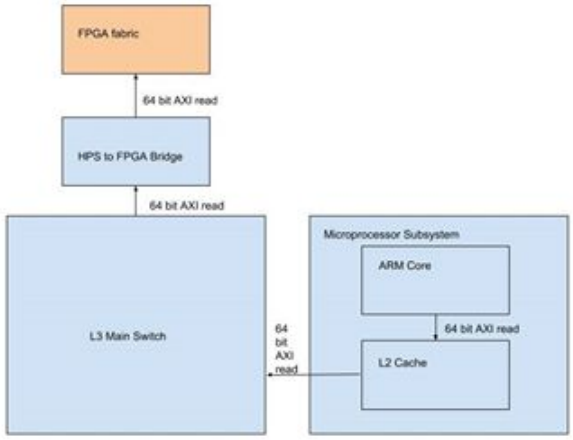
The FSM for stage 2 will wait in idle until the top FSM gives it the signal to begin loading the first row of the energy map into intermediate registers that will hold the accumulation row. Then it will move into the Accumulation state. The accumulation value for a cell is found by adding the current cell's energy value to the minimum accumulation value of the adjacent top three cells. The paths followed by this accumulation will be stored in embedded memory, the same size as a frame. Each cell in this accumulation paths matrix will hold the index of the pixel that was the minimum of the top three adjacent ones.




### *C. Stage 3*

This will involve first (in the Pick state) running the final accumulation row in the registers through a modified series of pipelined comparators that will yield the pixel index of the minimum value (the starting point of our seam). We used a 2 stage pipeline of comparators that compared 8 values at once. Thus, to find the minimum of 128 values, we took 17 cycles (1 additional cycle for first stage). Then, in the Travel state, we used the accumulation paths matrix from stage 2 and iterated through that matrix with the starting point found from the Pick state (similar to traveling through a linked list) and store all the indices into a FIFO buffer queue. Each cell in the matrix will give the index of the next cell - this index will be stored in the matrix and then used as the address for the next cell, and this will repeat until we reach the top row. We send back the pixel indices to the HPS through the FPGA-HPS FIFO. We used two different FIFOs as the FPGA-HPS FIFO and rate of arrival for a new pixel index is at different latencies.




### *D. HPS Post-processing*

After we calculate the pixel indices of the seams to remove on the FPGA, this data needs to be read by the HPS in order to remove these seams from the video array. This data transfer will be done through a HPS to FPGA bridge, which allows the HPS to read from the FPGA's peripherals through the L3 main switch on the HPS. The L3 main switch is connected to the ARM Core through the L2 cache, which allows the main processor to receive the indices after they have been read from the FPGA. All of these data transfers will be done through a 64 bit AXI read. The bridge is linked to our FPGA to HPS FIFO, which allows our scripts to read in the pixel indices. Once the ARM core receives the indices to be removed, we can run our seam removal scripts on Linux to remove the pixels at those indices from the video arrays.

## VI. Project Management

### *A. Schedule*

We had to push our schedule back about 2 week from our original plans due to delays in the shipping of the camera, additional time required to select which FPGA to use for our project, as well as additional time required to learn the tools required to use the DE-10 standard, including the Embedded Design Suite, and Megafunction Wizard.

(See attached last page)

### *B. Team Member Responsibilities*

We all chose to be involved in the top level algorithmic understanding and discussion of overall implementation and tunable parameters. We each researched different viable solutions and compared them together, settling on that which has been described. From there Kimberly took the initiative to write the C++ software benchmark of the system and reported on the timing metrics. Eshani worked on researching the DE10-Standard HPS and understanding how to interface with the fabric. She also is in charge of our quality metrics analysis. Shruti has taken the high level algorithmic approach and designed the hardware implementation (tradeoff between parallelization and use of resources) to the FSM and datapath. From here each member wrotes certain modules and testbenches. Shurti worked on stage 1, Eshani worked on stage 2, and Kimberly worked on stage 3. Kimberly worked on quality metrics.

### *C. Budget Items*

a. Camera - $131.30

Note: Due to design changes, ended up unused

b. DE-10 - Borrowed ($0)

c. Monitor - Borrowed ($0)

### *D. Risk Management*

We are transferring a lot of data between different parts of the DE-10 board in our algorithm. One risk involved here is data corruption so we can plan to write unit tests to check for data integrity between each of the modules. Another one of our main risks is that memory on the FPGA would be more constrained than our theoretical calculations. Our contingency plan for this would be to use the SoC to divide the video into blocks more manageable by FPGA memory and sent in intervals, or alternatively to constrain the video resolution. We chose to preprocess the video on the SoC for this reason so we could easily the size of blocks we are computing on at a time if required. Another risk we have is overlapping seams, since we have the ability to remove up to 5 seams at a time. Our plan to mitigate this risk is to check for duplicate indices when the processor receives the indices and remove only the seam of lowest accumulated weight that includes the duplicated indices.

We are transferring a lot of data between different parts of the DE-10 board in our algorithm. One risk involved here is data corruption so we can plan to write unit tests to check for data integrity between each of the modules. Another one of our main risks is that memory on the FPGA would be more constrained than our theoretical calculations. Our contingency plan for this would be to use the SoC to divide the video into blocks more manageable by FPGA memory and sent in intervals, or alternatively to constrain the video resolution. We chose to preprocess the video on the SoC for this reason so we could easily the size of blocks we are computing on at a time if required.

In our design report, we discussed another risk we had which was the removal of 5 seams per algorithmic run. Our plan was to mitigate this risk by checking for duplicate indices when the processor receives the indices and remove only the seam of lowest accumulated weight that includes the duplicated indices. However, we no longer needed to account for this risk because we updated our design to only always choose the optimal seam and recalculating the energy map before choosing the next seam to remove. We discussed our motivations for doing so in the previous sections.

## VII. Related Work

Setlur et al. [2] proposed *Automatic Image Retargeting* which 1) identify regions of interest in image, 2) segments the image based on those regions, 3) fills the resulting gaps, 4) resizes the remaining areas and then 5) re-inserts important regions to obtain the output. The results produced by this method are aesthetically satisfying as it preserves important features but, this method require many sequential steps and is thus, time consuming.

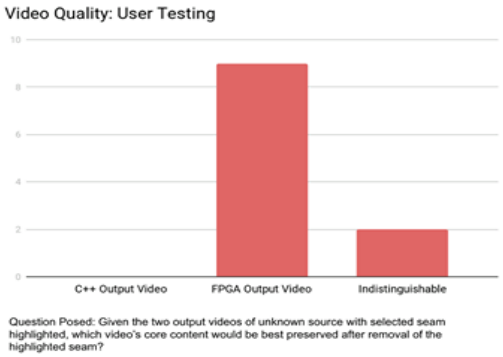
Avidan [3] proposed in *Improved Seam Carving for Video* a formulation of the seam carving operator as a *minimum cost graph cut* problem on images and then extended it to video. They define a video seam as a connected 2D manifold surface in space-time that cuts through the video 3D cube. The intersection of the surface with each frame defines one seam in this frame. To implement minimum cost graph cut, they construct a grid-like graph from the image in which every node represents a pixel and connects to its neighboring pixels. Virtual terminal nodes, S (source) and T (sink) are created and connected with infinite weight arcs to all pixels of the leftmost and rightmost columns of the image respectively. The optimal seam is defined by the minimum cut which is the cut that has the minimum cost among all valid cuts. The results produced by this method are aesthetically satisfying but requires both large amounts of memory and time to construct the graph for the entire video.

Yasuhide [5] presents a hardware-oriented seam carving algorithm for *images* in *Performance evaluation of hardware-oriented seam carving algorithm*. The algorithm gives a dedicated processor for each pixel in a row/column of an image, and the parallel computation for the pixels can be done. The performance of the algorithm is evaluated on an FPGA board, and it turns out that the algorithm can achieve two thousands of performance as much as that for the original one. The implementation works very well as they are able to fit the entire image onto the FPGA's memory, but this is infeasible for videos which require much more memory.

Jin [6] presented a method for calculating the removal seam for each frame of a video separately in his website project *Seam Carving*. Temporal coherency between frames is also preserved by using look-ahead energy, a linear combination of energies from future frames. The optimal seam for one frame is achieved by finding the minimum cut on the cube which consists of the current frame and the next 4 frames. In this way, the speed is greatly increased when compared to a graph cut on the entire voxel cube.

## VIII. Summary

Our MVP for this concept captured some of the basic functionality of seam carving with the added benefits of running on the FPGA. We met our timing requirements because we got a 13x speedup. We also met our user testing requirements, as shown below.




However, there were many planned extensions and additions that we did not get to and would be worth exploring in the future. To begin with, there is still much scope for streamlining and optimizing of the current design and features. We ran into issues with allocating memory vs timing and other such tradeoffs, and future work could entail finding a better balance than what we ended up with. The target max video size can be increased so larger device sizes can be accommodated – pipelining over half frames (or smaller segments) would allow use of the same number of functional units with an increase of time for a larger video that otherwise caused overfitting for the board. The current design also involves reinputting a video for each seam to remove – our previous solution of storing the video in SDRAM rather than the HPS file system and removing seams directly each iteration over the algorithm might improve timing.

### A. Future work

There is also much to be added to the algorithm itself. We implemented the algorithm using static seams without forward energy – the next extension we should explore to better pick seams would be to also implement forward energy, and then use the graph cut method to better target videos with moving subjects or camera. We can also extend the algorithm use – we can use it to add seams instead of just removing, we can use a frame look ahead method and process for a live video feed, we could use implement object/facial detection to better identify high energy regions in a frame… there are lots of cool extensions!

### B. Lessons Learned

Our main lessons learned were that it's important to have a simple but guaranteed solution, as many contingency plans can fail. It's also important to start with a simple plan and optimize later once the components work because planning for the most optimized solution from the beginning can lead to many bugs. We learned this lesson as we wanted to use the SDRAM on the board to store data for faster data transfer but we ran into many issues with the SDRAM and we realized a more effective solution would be to use FIFOs to handle the data transfer.
Although we had planned the overall system architecture together, we decided to parallelize the work by writing the modules on our own and then integrating those modules together at the end. This decision was not bad - but we had to fail to consider how long it would actually take to integrate all our modules together. We should have given much more time

for this in our schedule. Another obstacle also came with this late integration process – after combining all our modules, compiling and synthesizing, we could not fit our entire design onto the board. We had used more than 25% of the available Adaptive Logic Module (ALM) on the board. Because of this, we were again faced with another design trade-off. We considered our options which included optimizing Quartus's compiler for area instead of speed, and this gave a 5% reduction in area, but we were no longer meeting the timing requirements. After conversing with our project advisor, we decided to constrain the number of columns we were computing in parallel by instead. This solved our issue of area but at the cost of less parallelization. We then discussed other methods to increase performance which included pipelining stage 1 and packing multiple pixels into the data sent through the HPS-FPGA FIFO.

Another lesson we learnt is to choose your hardware very wisely. When we chose the DE10-Standard board, we made the assumption that all the Cyclone V boards were similar and that we could reuse libraries for the DE10-Nano and DE1-SOC for our own board. However, this was not the case as each board carried different mappings for memory and had different sizes for that as well. Although the differences seemed subtle, it proved to be breaking. Many of the documentation and tutorials online were also geared towards the DE10-Nano and DE1-SOC as those boards were more price accessible. In hindsight, we should have verified our FPGA-HPS communication on the DE10-Standard much earlier before fully committing to the board as we would have more time to pivot on our choice of board if so. On the bright side, this challenge pushed us to understand our hardware on a much deeper level than we had originally anticipated.

Another option we considered to contend the memory constraint of the DE10-Standard was to use a Xilinx board which has much more memory on it. However, using a Xilinx board would have meant that we would have to learn a new toolchain, Vivado, and after careful consideration, we decided that working on the DE10-Standard would allow us to focus on our implementation and even pushed us to design innovatively within the memory constraints of the board. This is not to say that we did not learn new tools while working on the project; we had trouble debugging once we had placed the design on the board as we were using LEDs to communicate information and that was not very informative. We should have consulted others who have used the board before as we would have found out about a much richer debugging tool, SignalTap. SignalTap is an Embedded Logic Analyzer megafunction where one can select signals, set up triggers, configure memory and display waveforms; these functions proved very helpful in debugging and verifying our design and we should have known tools available to us from the start.

Furthermore, we could have consulted with other teams to further verify our design. There was another team working on a similar project as us who also faced similar problems on communication between the SoC and FPGA and the slow data transfer between the two components. We realized that by verifying our design just between ourselves and advisors, we had developed our own echo chamber on what the design was and not what it could have been. By connecting with the other team, we could have given and received feedback that was highly specific to our problems. We should have taken our own initiatives to leverage on the intellectual and collaborative environment in CMU as we would have been able to design better solutions for our project quicker.

## REFERENCES

[1] Avidan, S., AND Shamir, A. 2007. Seam carving for content-aware image resizing. In Proceedings of SIGGRAPH, Article No. 10.

[2] Setlur, V., Takagi S., Raskar R., Gleicher M., AND Gooch B. 2005. Automatic image retargeting. In Proceedings of MUM '05, Proceedings of the 4th international conference on Mobile and ubiquitous multimedia, 59-68.

[3] Avidan, S., Rubinstein, M., AND Shamir, A. 2008. Improved seam carving for video retargeting. In Proceedings of SIGGRAPH, Article No. 16.

[4] Yasuhide K. AND Yoshihisa D. 2014. Performance evaluation of hardware-oriented seam carving algorithm. 2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE).

[5] Cheung C. AND Jin. R. Seam Carving. 2016. [Online]. Available: http blackruan.github.io/seam-carving/ [Accessed: 4-Mar-2019].

## SCHEDULE


2019
April
Lab demo
edit GHRD to tr
send indices to HPS
write accumulation matrix module
conduct user testing
set up FPGA to HPS data
HPS read from SDRAM to output video for metric testing
design user pol
write unit tests for SoC modules
write spatial and temporal energy map
test stage 1 module - waveform vie
No work
connect embedded memory to top module
stage 3 module
calculate benchmark metrics
CARNIVAL
running unit tests and ve
final testing
May
In lab demo
FINAL DEMO
REPORT DUE
conduct user testing
design interfac
Q15'S data connections
visual quality metrics
integrate with C code to use FIFOs
calculate quality metrics
module correctness
design interface for parar
calculate qualit
get results of user testing
integrate with C code to use FIFOs
timing metrics
read from file system in C
final testing
final presentation slides
handle multiple seams
synthesize modules
decide how to handle input for number of se
poster
plan for final demo
final report
March
instantiate test matrices
run nios II tutorials
set up scp to Sc
run m
edit GHRD
read through de
use nios II to write data to sdram and read to console
set up FPG
write top module and all FSMs
write unit tests for fpga modules
Planned out input/output and conti
Create embedd
DBM Camera VGA output monitor
deleting seam (HPS)
writing algorithm slack time
Integrating demo work
design report due
test all versions of Quartu
set up
test GPIO camera
display camera output on
spring break
eshani
shruti...
klimjinx
team