3D Reconstruction by Stereo Imaging

Brian Axelrod Amartya Shankha Biswas Xinkun Nie

November 4, 2015

1 Overview

Stereo vision is the process of extracting 3D depth information from multiple 2D images. Conventionally, two horizontally separated cameras are used to obtain two different perspectives on a scene. Because the cameras are separated, each feature in the scene appears at a different coordinate in both images. This difference between these coordinates is called the disparity and the depth of each point in the scene can be computed from it's disparity. The difficult part is computing the disparity at each point.

Algorithms for computing features between images are generally complex, memory inefficient and require random access to large portions of memory. The state of the art Stereo Algorithm is based on Semi Global Matching (SGM), a global correspondence optimization. This algorithm performs very well in in practice but is extremely memory inefficient.

However, SGM can be modified with some heuristics such that the amount of temporary memory it uses is proportional to the number of pixels in the image. This improved memory complexity makes efficient-SGM (eSGM) ideal for implementation on FPGAs.

Writing a complete stereo pipeline requires many diverse components. A full stereo pipeline based on eSGM requires rectified camera images, and rendering the output in point clouds and depth maps. Furthermore we will need to design a memory architecture that can maintain a high throughput and utilize our computation resources effectively.

2 Design

Our project is primarily designed to be easy to debug. The guiding principles of the design are

- Use small testable modules
- Use Vivado HLS and c++ test benches wherever possible
- Use a standard streaming interface to make it easy to replace modules
- Seperate big components of the project with writes to memory, making it easy to replicate functionality of just part of the pipeline.
- Use the softcore for control and configuration

Our overall design does the following:

1. Grabs frames from the cameras
2. Processes the frames to remove noise, distortion and applies a census transform and stores the result in memory
3. Grab the frames from memory and merge them into a interlaced frame that can be processed by the SGM module and store the results in memory
4. Stream the frame in the forwards direction from memory through the SGM first pass module. Store the results in DDR RAM.
5. Stream the frame in the reverse direction from memory through the SGM first pass module. Store the results in DDR RAM.
6. Stream the results of the first pass modules through a merge module and compute the disparities. Stream the results to memory for rendering on the screen.
7. Stream the rendered image from memory to the screen

Each module's output is an AXI stream which will make it much easier to interchange modules later on.

See the flowchart for an explanation to better see the flow.

Here's a list of bigger modules in the flowchart and a brief descrition of their purpose:

2.0.1 Filtering

We do not want noise present in a single pixel to affect the result of our algorithm. In order to make our system more robust to noise we apply a standard technique in computer vision—applying a guassian blur. We apply a gaussian kernel to the image, essentially blurring it by making each pixel a weighted average of it's neighbors.

2.0.2 Rectification

The basic premise of most stereo algorithms is that you want to find corresponding patches along epipolar lines. In a perfect world these epipolar lines would simply be horizontal lines. Optical distortion bend the epipolar lines, which will be made to align with the horizontal axis after rectification.

We plan to rectify the images by first calibrating the cameras off-line to get a rectification matrix. The streamed frames would then be multiplied by this matrix to get a rectified image.

2.0.3 Census Transform

The matching cost over all pixels is part of the SGM cost function that needs to be globally optimized. We plan to use the census transform for the matching cost. We use a 5x5 window to get information around each pixel to perform the Census transform.

2.0.4 SGM Cost Calculator

The SGM algorithm goes from left to right, and top to bottom in the frame in its first pass. Only the line above the current line and the current line need to be stored in the DDR. For each pixel, we look at the pixel above it, right left to it, above and left to it, and above and right to it. We compute the cost associated with each disparity value for the current pixel.

2.0.5 SGM Merge Module

After two passes (from top left to bottom right, and from bottom right to top left in the frame) of calculating the SGM cost, we need to merge the results and compute the disparity value that gives the lowest global cost.

3 Implementation

In order to manage the large complexity of our project we are enforcing a modular design with lots unit testing. On a high level our project consists of five parts.

1. Camera Capture and preprocessing
2. Semiglobal Matching Algorithm
3. Output Rendering
4. Memory subsystem

Each of these systems can be tested individually. We describe our testing and development procedure for each one of the subsystems below.

3.1 Camera Capture and preprocessing

The camera capture and preprocessing part of the system includes several modules that form a larger module with a single AXI ouput. These modules are listed below.

- AXI Camera Capture
- AXI buffering
- Camera Rectification
- Image Filtering
- Census Transform

3.1.1 AXI Camera Capture

This module will be based on Weston's reference code for camera capture. It will be extended to support the AXI interface. This will be tested individually by just showing a camera feed on a VGA display.

3.1.2 AXI Buffering

Properly conforming to the AXI specification requires only transmitting data when the receiver asserts a ready signal. In order to be able to do so with Weston's camera code we have to add a FIFO buffer module that reads from Weston's code's output, and buffers it to an output using the AXI interface. This will be tested with a testbench and a live test with a display.

3.1.3 Camera Rectification

Camera rectification requires remapping and interpolating pixels to compensate for optical distortion and improper alignment of the cameras. This will require off-line calibration of the camera to get the rectification matrix, which represents the internal distortion of the cameras. The rectification process multiplies this matrix with the pixel location for each pixel from the frame. The newly transformed image is then cropped to contain only the locations that have pixel information

3.1.4 Image Filtering

We use the Gaussian smoothing operator to blur the input frames in order to reduce noisy input. We plan to use a 3x3 discretized Gaussian filter to convolve with the input frames.

3.1.5 Census Transform

The Census transform is used as the matching cost for each pixel. This matching cost is part of the global cost function that we maximize in the

Semi Global Matching algorithm. We have chosen to use the Census Transform because this matching cost appears to have the highest radiometric robustness [2].

The details of the census transform can be found from [3]. In short, we convert the RGB values of each pixel into the Gaussian color model (GCM) by having a matrix transform. We plan to use a 5x5 window to compute the census transform, which produces a bit string of 1's and 0's. We compute the matching cost by calculating the sum of the Hamming Distance between the census transform of the two given pixels.

3.2 Memory Efficient Semi-Global Matching

3.2.1 Semi-Global Matching

We want to reconstruct a 3D depth image from two stereo camera inputs. This involves matching corresponding pixels between the two images.

This gives us a **disparity** value D_p for each pixel p, where D_p is the difference in the position of the pixel across the two images. The 3D depth of each pixel can then be computed from it's disparity.

The state of the art algorithm for matching pixels is Semi-Global Matching (SGM) [1]. SGM uses dynamic programming to minimize a global cost function along the epipolar lines. Unlike other dynamic programming methods, it does not recurse only along the epipolar lines. Instead, the minimization is done along eight directions. This prevents streaking artifacts.

We use the 5×5 Census Transform as a metric to assign cost values $C(p, d) = \|I_L(p) \oplus I_R(p - d)\|$ to each pixel p and disparity value d. Here I_L and I_R are the values of the Census Transform and the cost is calculated as the Hamming Distance. Then we define the cost of each path ending at a pixel as $L_r(p, d)$. where d is the disparity value at pixel p, and r is one of the eight directions. $L_r(p, d)$ is computed according to the recurrence –

$$
\begin{aligned}
L_r(p,d) = C(p,d) + \min\{ & L_r(p-r,d), \\
& L_r(p-r,d-1) + P_1, \\
& L_r(p-r,d+1) + P_1, \\
& \min_i\{L_r(p-r,i) + P_2\}\} - \min_k\{L_r(p-r,k)\}
\end{aligned}
$$

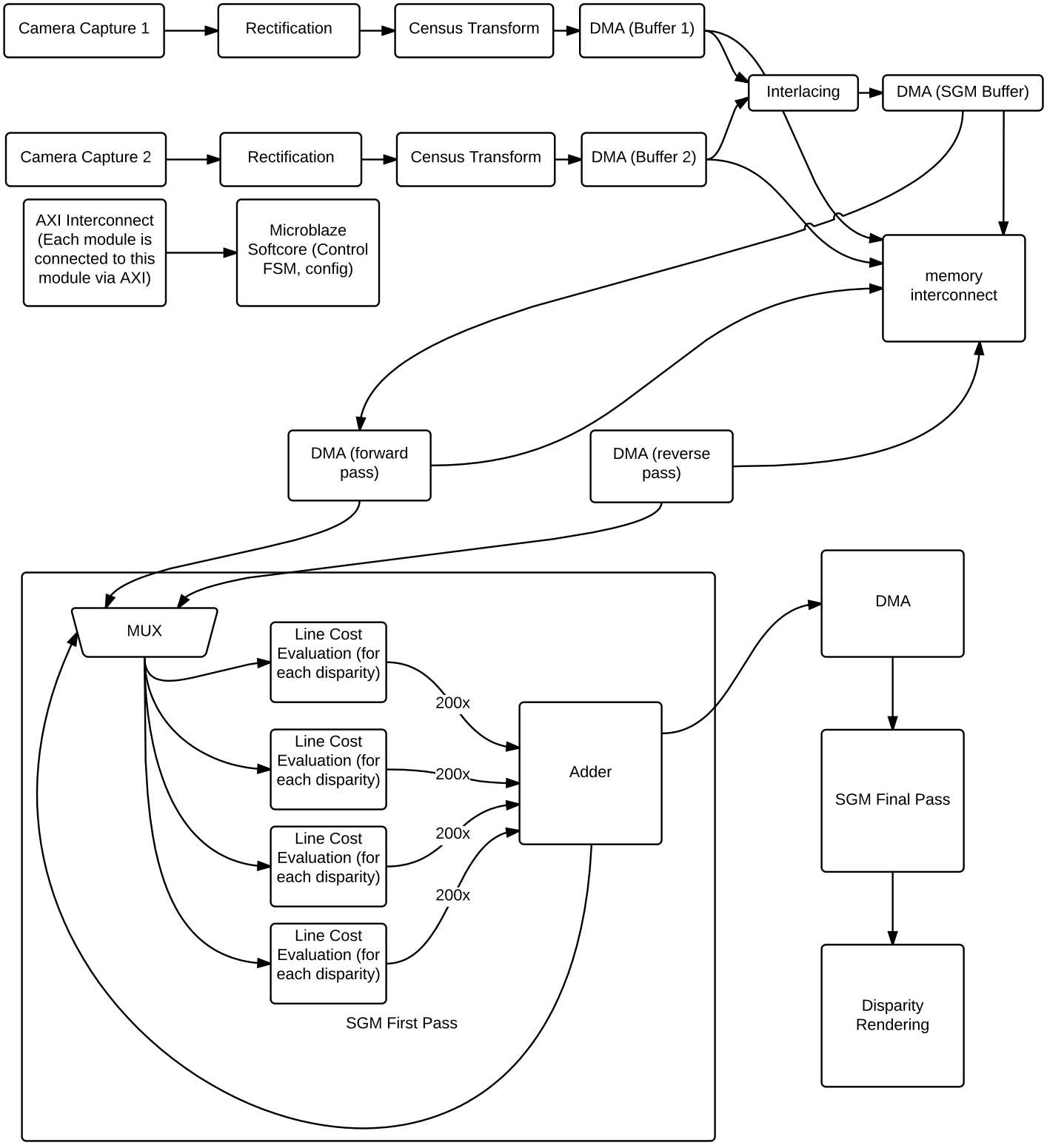
Then we compute the cost of each disparity value d at pixel p as the sum of costs of each of the eight paths.

$$S(p,d) = \sum_r L_r(p,d)$$

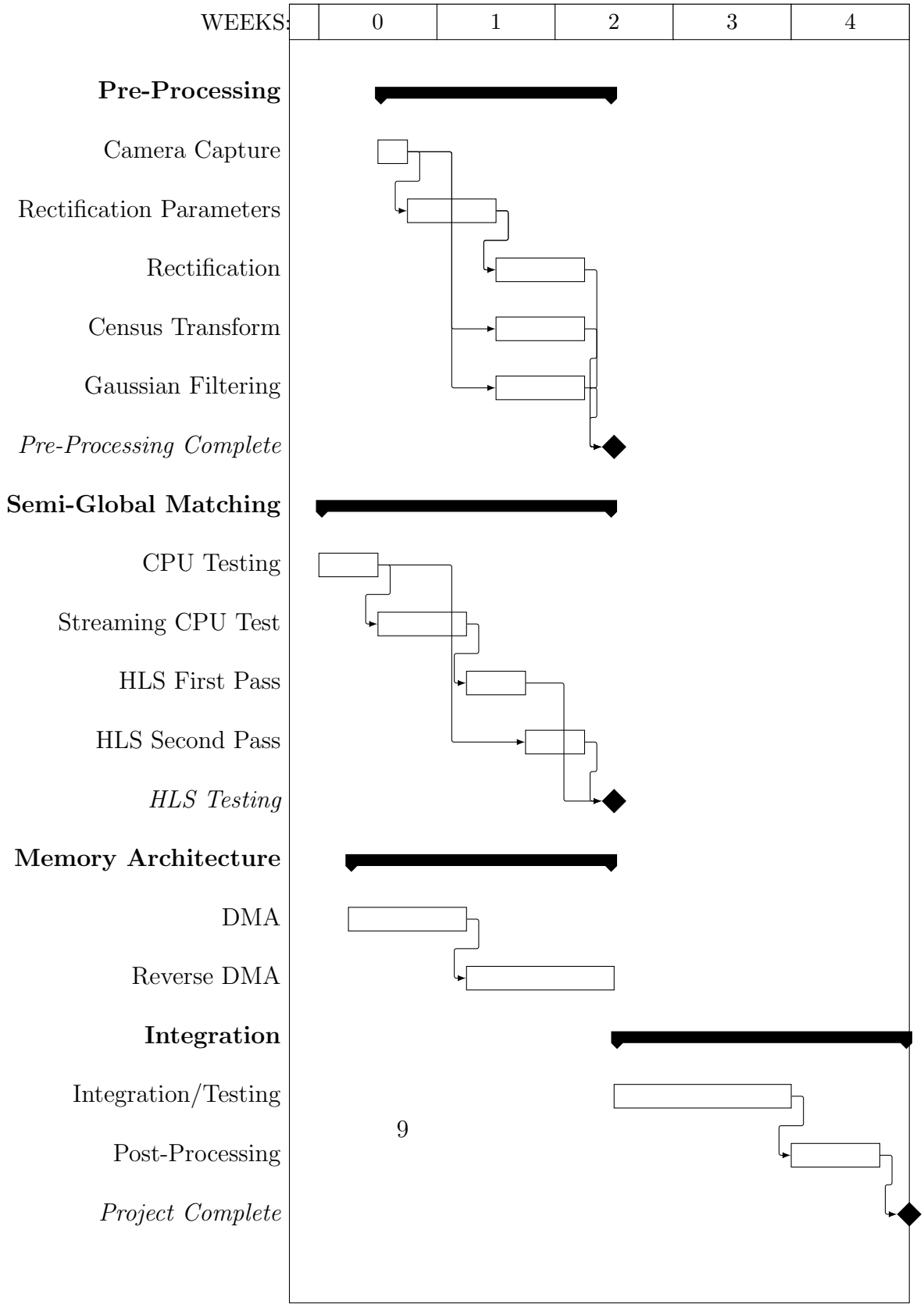
Finally, the true disparity of each pixel corresponds to the minimum cost.

$$D_p = \arg\min_d\{S(p,d)\}$$

To implement this algorithm, we perform two passes on the image. In the first pass, we start at the top left and scan line by line and compute the $L_r(p,d)$ values for four directions (left, top-left, top and top-right). In the second pass, we start at the bottom right and scan in reverse line by line and compute the $L_r(p,d)$ values for the remaining directions (right, bottom-right, bottom and bottom-left). After the two passes, we aggregate the $L_r(p,d)$ values to obtain $S(p,d)$.


Camera Capture 1
Rectification
Census Transform
DMA (Buffer 1)
Interlacing
DMA (SGM Buffer)
Camera Capture 2
Rectification
Census Transform
DMA (Buffer 2)
AXI Interconnect (Each module is connected to this module via AXI)
Microblaze Softcore (Control FSM, config)
memory interconnect
DMA (forward pass)
DMA (reverse pass)
MUX
Line Cost Evaluation (for each disparity)
Line Cost Evaluation (for each disparity)
Line Cost Evaluation (for each disparity)
Line Cost Evaluation (for each disparity)
200x
200x
200x
200x
Adder
SGM First Pass
DMA
SGM Final Pass
Disparity Rendering

4 Timeline


WEEKS:
0
1
2
3
4
Pre-Processing
Camera Capture
Rectification Parameters
Rectification
Census Transform
Gaussian Filtering
Pre-Processing Complete
Semi-Global Matching
CPU Testing
Streaming CPU Test
HLS First Pass
HLS Second Pass
HLS Testing
Memory Architecture
DMA
Reverse DMA
Integration
Integration/Testing
Post-Processing
Project Complete

5 Testing

The basic testing for each HLS module consists of writing C++ test benches that verify the module. The C++ test benches are very important in terms of verifying the correctness of our logic. Furthermore they are very easy to run and very fast allowing us to catch errors very quickly in development. Once we are passing the C++ testbench we will use the vivado HLS cosimulation tools to test the generated verilog with the C++ testbench. We will perform integration testing once the unit testing is finished. We will have a couple of different vivado projects based on shared IPs for integration tests that will test various parts of the pipeline. Again we intend to make it very easy to run a large number of tests and quickly identify and isolate issues.

For the camera portion of the project, we want to test displaying a test image with VGA, rendering camera stream by writing it to DDR and displaying it with VGA, rendering camera stream by using AXI to stream out the data to VGA, verify rectification and filtering work fine.

For the SGM portion of the project, we want to compute the correct disparity value for each pixel. We will have a reference ground truth computed from our C++ implementation of SGM. Our C++ testbench will compare the accuracy of our FPGA implementation to the original known working C++ implementation. We will also test individual parts of the SGM algorithm in this manner since it is easy to obtain known good results using our C++ reference implementation and the vivado HLS testing framework.

The memory infrastructure will be tested with simple, fake test pattern modules that will spit out fake AXI streams. We will use the microblaze softcore to make sure that the memory contents are correct.

Finally the rendering pipeline will be tested with fake modules that spit out memorized SGM outputs.

Once we've tested all the individual parts we will run larger integration tests and use on chip debugging and test code running on the microblaze to make sure that the hardware we generate is behaving as expected at every stage.

This will also test the overall control and finite state machine that coordinates the work of the various modules.

6 Resources

This project requires two cameras in addition to the Nexys 4 FPGA. The output is sent to a monitor through VGA.

The two cameras will be rigidly mounted next to each other and separated by $60mm$. This is roughly the gap between human eyes and will be used to obtain a pair of stereo images.

References

[1] Heiko Hirschmuller. Semi-global matching-motivation, developments and applications. 2011.

[2] Heiko Hirschmüller and Daniel Scharstein. Evaluation of stereo matching costs on images with radiometric differences. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(9):1582–1599, 2009.

[3] Soo-Chang Pei and Yu-Ying Wang. Color invariant census transform for stereo matching algorithm. In *Consumer Electronics (ISCE), 2013 IEEE 17th International Symposium on*, pages 209–210. IEEE, 2013.