3D Reconstruction by Stereo Imaging

Brian Axelrod Amartya Shankha Biswas Xinkun Nie

November 13, 2015

1 Overview

Stereo vision is the process of extracting 3D depth information from multiple 2D images. This 3D information is important to many robotics applications ranging from autonomous cars to drones. 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 its disparity. Computing the disparity at each point accurately and efficiently is quite difficult.

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 matching algorithm is based on Semi Global Matching (SGM). This algorithm performs very well in in practice but is extremely memory and processing inefficient. This makes it difficult to process it on small computers that can fit on small robots like drones. Since FGPAs are fairly low power, an FPGA implementation of SGM would allow us to use SGM on small platforms such as drones.

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

In order to be able to compute high quality disparity maps we must combine many complicated modules to compute SGM and pre and post processes our images. Our design decisions are primarily driven by the need to manage this complexity without sacrificing performance. Thus we establish a design pattern based on good software engineering patterns that have been adapted to the Vivado workflow. The main idea is that our design should be split into small manageable pieces that can be tested individually. We will leverage Vivado HLS and C++ test benches to quickly create thorough testbenches based on real data. We will also use standard streaming interfaces which will make it easy to replace modules and design tests. This will make it easy for us to understand exactly what we want to get out of a module and verify that it is correct. We will also use a softcore for running tests on the FPGA and running the state machine. This will allow us to use code that has been auto-generated by the Xilinx tools and avoid having to write and test more code.

Our design revolves around a pipeline for processing stereo images shown in Figure 1.

The first part of the pipeline grabs frames from the cameras. It handles synchronization and passes on the results over an AXI stream that feeds into the preprocessing module. The preprocessing module applies the rectifications transformation, the gaussian blur that mitigates the effect of noise, and applies a census transform to compute a value that describes the neighborhood of each pixel. The result is streamed into ddr memory through a Direct Memory Access (DMA). We then take the results and pass it through the SGM module twice, first in the forward direction and then in the reverse di-

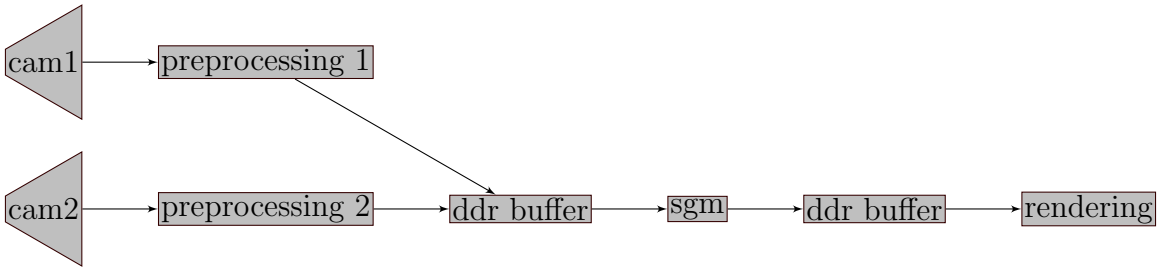

Figure 1: A high level overview of the design

rection. Then the second part of the SGM module combines the information from these two runs to compute the disparity values and stores the results in ddr memory using a DMA. Then a rendering module reads the disparity values and renders them.

See the detailed block diagram in figure 2 for more information. Here's a list of modules in the detailed flowchart and a brief descrition of their purpose:

2.1 Filtering

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.

2.2 Rectification

To handle a camera's intrinsic optical distortions and extrinsic rotation and translation shifts, we plan to rectify the incoming images. The basic premise of most stereo algorithms is 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 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.

2.3 Census Transform

We use the Census Transform to compute the matching cost over all pixels, which is a term in the SGM cost function that needs to be globally optimized. We use a 5x5 window to get information around each pixel to perform the Census transform.

2.4 SGM Cost Calculator

The SGM algorithm finds the optimal disparity value for each pixel by minimizing over a global cost function. The algorithm iterates through the pixels in two passes.

In the first pass, the iterator moves from left to right, and top to bottom in the frame. Only the line above the current line and the current line need to be stored in the DDR memory. 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.

In the second pass, the iterator moves from right to left, and bottom to top in the frame. Only the line below the current line and the current line need to be stored in the DDR memory. For each pixel, we look at the pixel below it, right to it, right below to it and left below to it.

We compute the cost associated with each disparity value for the current pixel.

3 Block Diagram

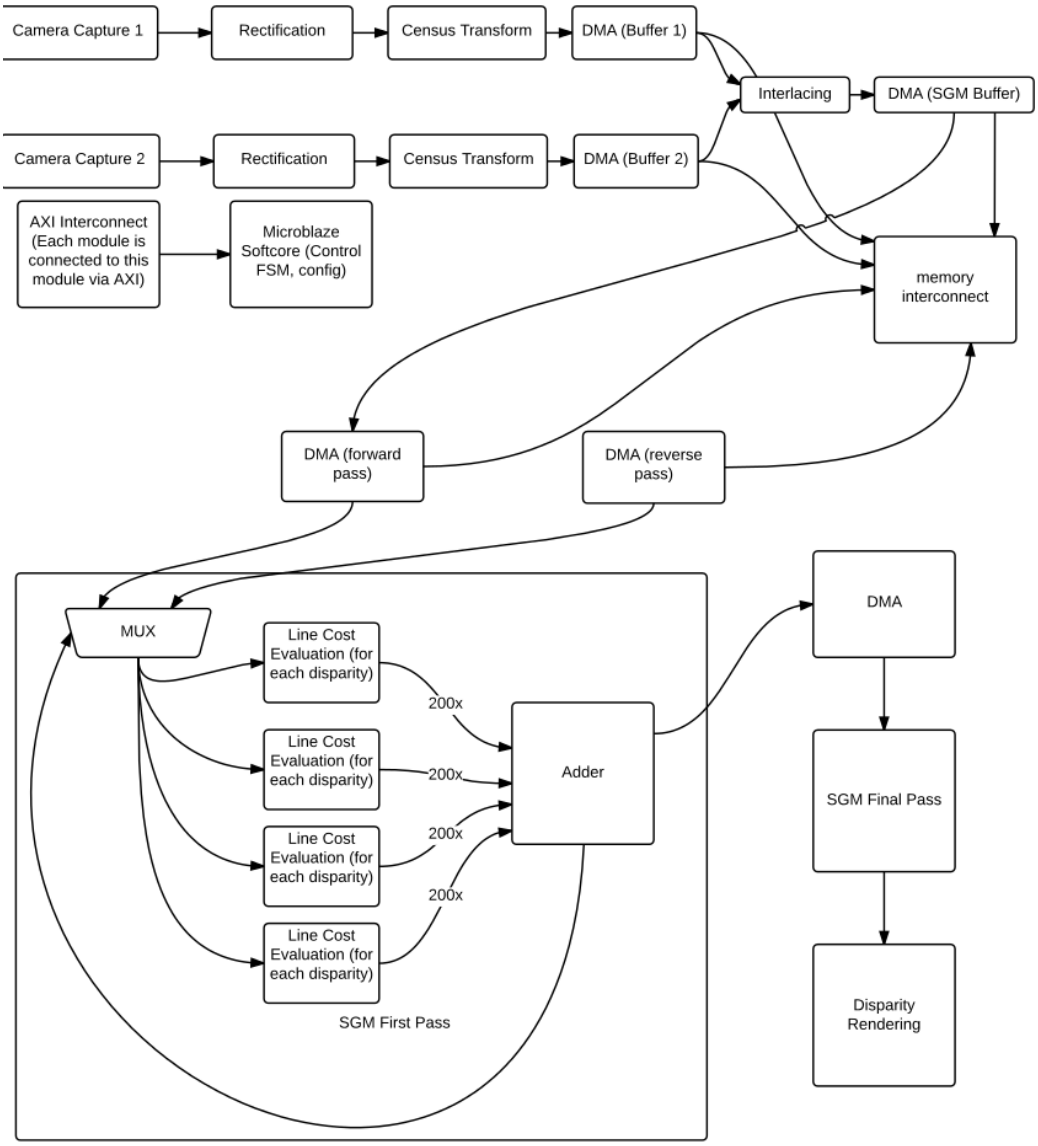

Figure 2: Detailed Block Diagram

4 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.

4.1 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

4.1.1 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.

4.1.2 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.

4.1.3 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

4.1.4 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.

4.1.5 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 [**?**].

The details of the census transform can be found from [**?**]. 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.

4.2 Memory Efficient Semi-Global Matching

4.2.1 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. Figure 3 shows a pair of stereo images and the depth map we computed during CPU testing.

Figure 3: Left image, Right image and computed Depth Map

The state of the art algorithm for matching pixels is Semi-Global Matching (SGM) [**?**]. 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 (Figure 4).

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 –

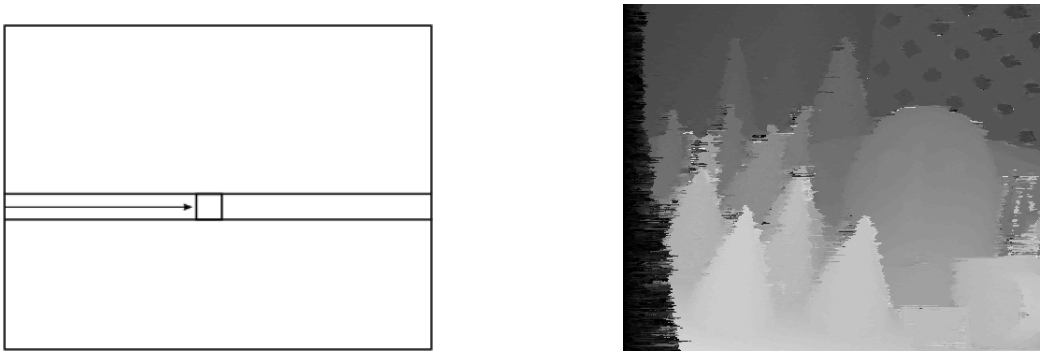

Figure 4: Simple Dynamic programming leads to Streaking Artifacts

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

Then we compute the cost of each disparity value d at pixel p as the sum of costs of each of the eight paths (Figure 5).

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

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

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

To perform this recursion, we use the Census cost values (read from DDR memory) and compute the overall cost (aggregate of path costs), which is then written to DDR. Because it is difficult and inefficient to perform random access on DDR, we implement this algorithm in a streaming fashion (Figure 6). This means that at every clock cycle, one pixel is processed. So, the throughput of this block is one pixel per clock cycle whereas the latency is the time taken to compute cost values for a single pixel.

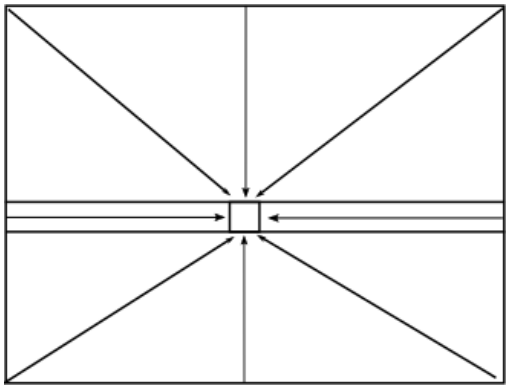

Figure 5: Dynamic Programming from eight directions

The block diagram shown (Figure 6) is used to compute the 1D optimized cost along a single direction at a time. This greatly reduces the amount of resources (registers and BRAM) used by this module. The algorithm performs eight streaming *passes* through this block, to compute the $L_r(p, d)$ values for each r direction (Figure 5). At each pass, these values are aggregated with the total cost stream (read from DDR), and written back to the same location in DDR. After all eight passes are complete, the final total cost $S(p, d)$ is stored in DDR.

In the detailed block diagram (Figure 6), the Census cost values from the two images are read in from an input stream and buffered in order to be able to compute the matching costs ($C(p, d)$). The L_r values for the previous prow of pixels and the current row of pixels is stored in BRAM. This is to allow fast random access to the L_r values. These matching costs and previous L_r values are used to compute the L_r value for the current pixel. This operation is pipelined to ensure that the pixels are processed in a stream (thoughput = 1 pixel per clock cycle).

To implement this algorithm, we also have to stream the image in two different ways. In the first case, 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 case, 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 sets of four passes, we have aggregated all the $L_r(p, d)$ values to obtain $S(p, d)$.

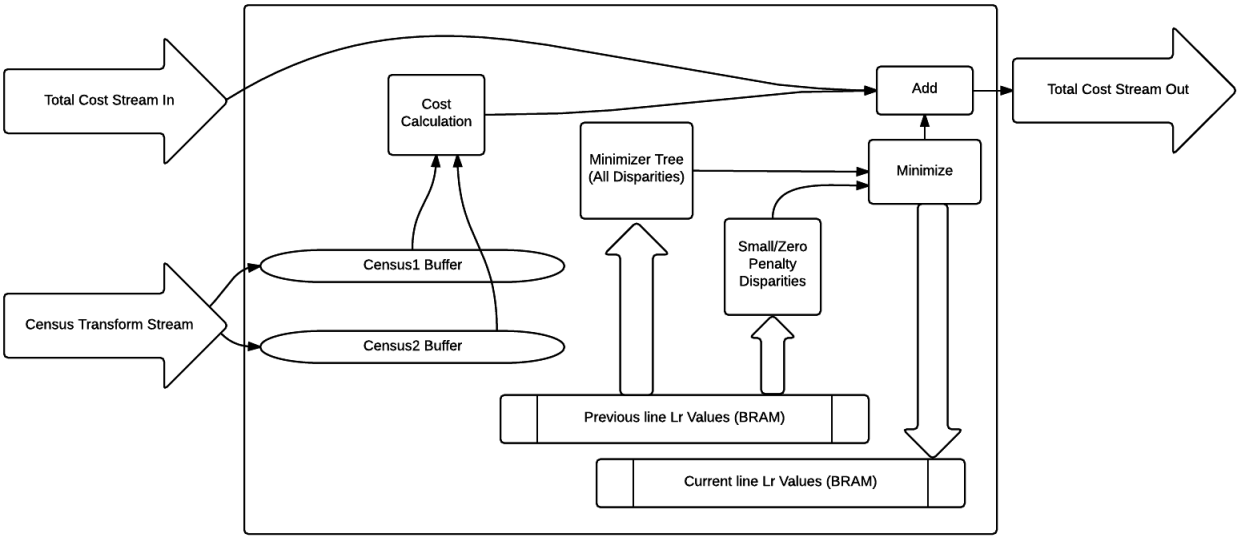

Figure 6: SGM Block Diagram

5 Timeline

Figure 7 shows a Gantt Chart with our planned schedule. Initially, all three of us will work in parallel on separate parts of the project.

- Pre-Processing This includes Camera Capture, Rectification, Census Transform and Gaussian Filtering. This part of the project is assigned to Xinkun Nie.
- SGM Algorithm This is the module that implements the SGM Algorithm. This part of the project is assigned to Amartya Shankha Biswas.
- Memory Architecture This allows us to write to and read from DDR memory. This part of the project is assigned to Brian Axelrod.

We will be spending the remaining time together on Integration of the different modules and testing

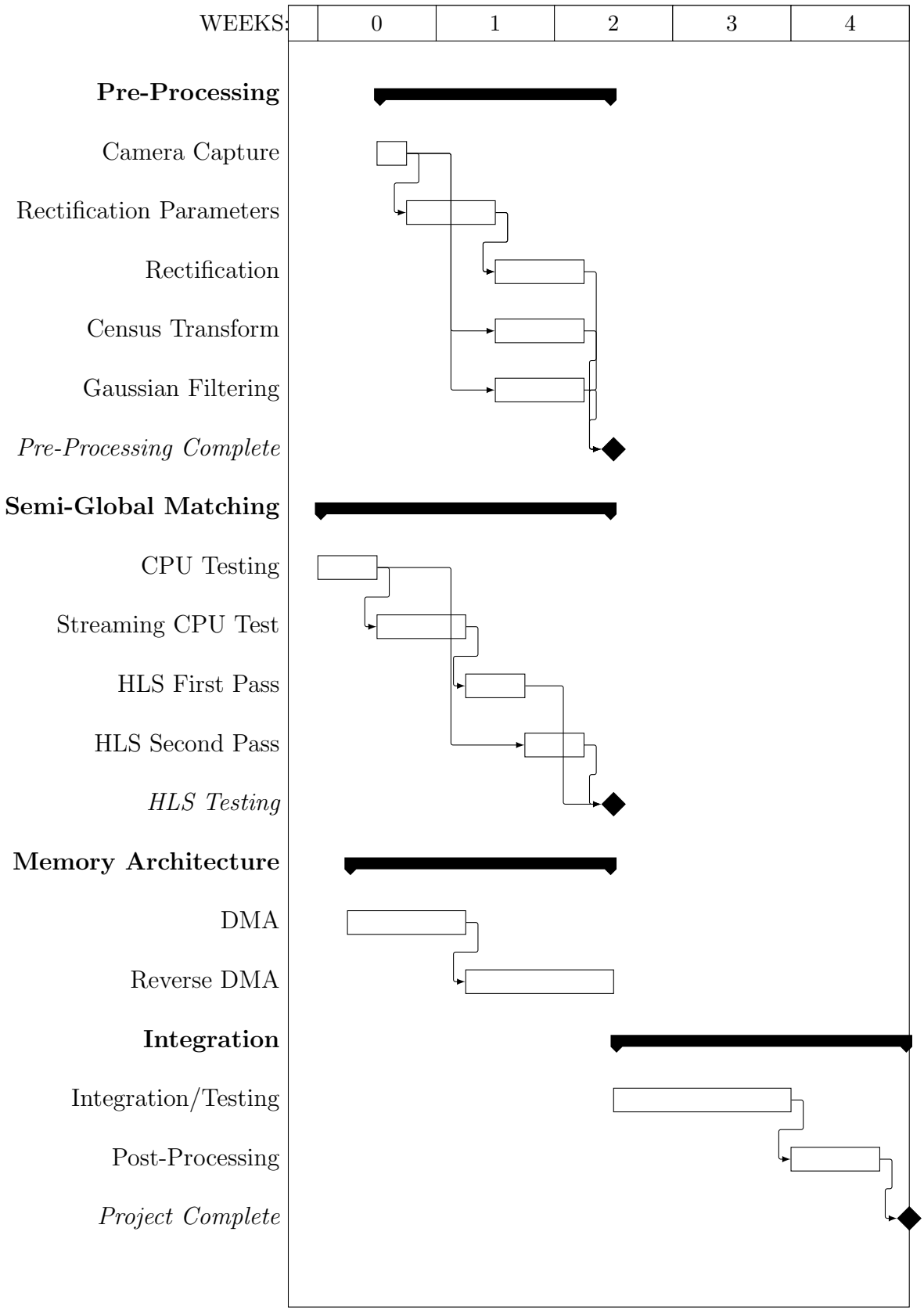

Figure 7: Timeline for Project

6 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.

7 Resources

This project requires two 6.111 lab 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.