# 6.375 Final Project Report
# **Stereo Vision for Driverless Cars**

MARC DE CEA FALCO, RAVI RAHMAN

December 11, 2019

## Contents

## 1 Introduction and Application

For any autonomous system it is essential to accurately and quickly perceive and measure distances to potential obstacles. Several techniques have been proposed to perform this task, with varying degrees of complexity, accuracy, cost and area coverage. Amongst them, one of the most simple, yet promising, approaches is the use of stereo vision [1].

Stereo vision can infer the depth of objects (such as the distance to an identified obstacle in the case of a self driving car) using images from two cameras a known distance apart. Given the position of the same object in the two images (one coming from each camera) one can determine the 3-D coordinates for the given point using simple geometrical considerations.

In this project, we want to implement a stereo vision algorithm in an FPGA that will be used by the MIT Driverless Racecar team. This team competes in the autonomous formula student series, where the car has to follow a path which is marked by cones. Cones are detected via a neural network and stereo vision is used to determine the positions of the detected cones, so that the car can correct its direction if necessary. The image pairs in figure 1 show a sample input for the stereo vision system taken by the racecar.

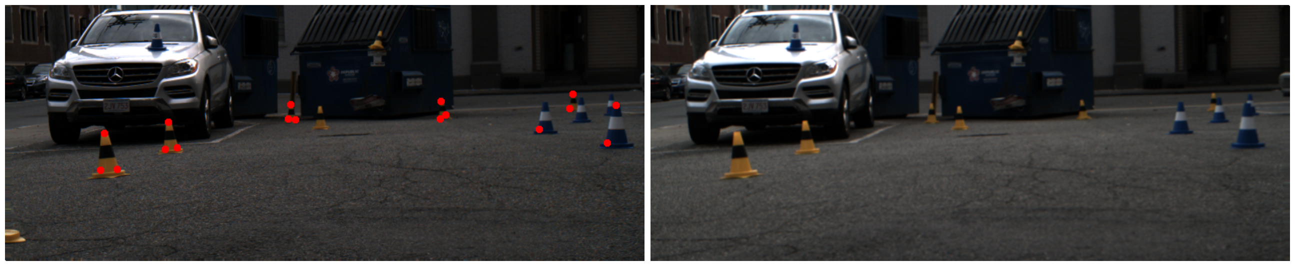


Figure 1: Sample stereo images taken by the MIT Driverless Racecar. The red points on the left image represent the coordinates of the points for which we want to compute real-world position coordinates using the stereo vision algorithm.

## 2 Project Objectives

As mentioned above, the goal of this project is to implement a fast stereo vision system to be used by the MIT Driverless Racecar team.

Since the latency with which we compute a safe path for the car to follow determines the maximum speed at which it can drive, the most relevant metric for our system is its latency. There are 15 ms allocated for the stereo vision algorithm, and therefore we want to synthesize an FPGA that reaches this target.

Therefore, we require that the FPGA be capable of computing real-world distances on 20 points from 800x320 pixel images at 60fps with less than 15ms latency. We do not have a required clock speed so long as the throughput and latency goals are met.

### 2.1 Assumptions

a. We will assume that the target points (red dots in Fig. 1) are an input to the stereo vision system. This is true in the case of the MIT racecar, since the target points are computed using a neural network.

b. We will assume that the two stereo images are rectified, which means that any distortion or misalignment between the two cameras has been corrected for. CPU implementations for image rectification are already very efficient, so implementing these on an FPGA might not give a big advantage.

## 3 Stereo Vision Basics

As briefly mentioned above, stereo vision can infer the depth of objects (such as the distance to an identified obstacle in the case of a self driving car) using only two images from two different cameras sitting a known distance apart. Given the position of the same object in the two images one can determine the 3-D coordinates for the given point using simple geometrical considerations.

Efficient stereo vision algorithms use rectification, in which the images are corrected for non-linear features such as distortion or misalignments between the pair of cameras. Given two rectified images, a pixel from one image corresponds to another pixel on the epipolar line in the other image. These two corresponding pixels represent the same position in 3-D space. Specifically, given two images from cameras with only horizontal displacement, both pixel-space coordinates will share the same vertical coordinate and will thus be on the same horizontal line (see Fig. 3).

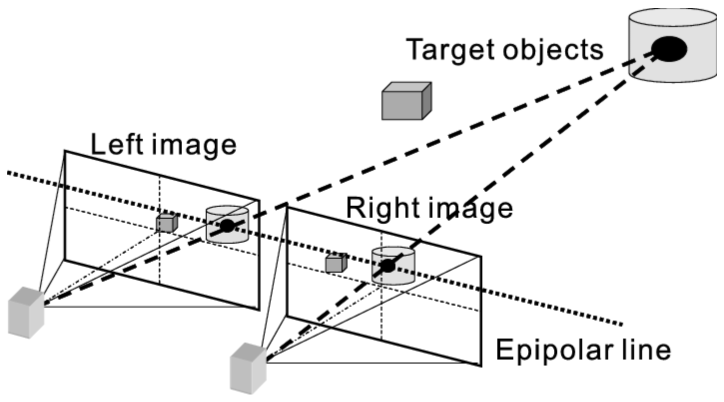


Figure 2: Stereo vision geometrical depiction [3]. Given the position of the same object in two images taken from two different cameras horizontally displaced, the 3D position of the object can be inferred.

Different algorithms exist to match the points in one image to the other. The most usual one is called Sum of Absolute Differences (SAD), and it consists in computing the sum of the absolute value of the pixel by pixel difference of a block of BxB pixels around the pixel of interest. Let $p_l(x, y)$ and $p_r(x, y)$ correspond to the pixel at location $(x, y)$ in the left and right images, respectively. Then, :

$$SAD(x,y) = \sum_{a=-B/2}^{a=B/2} \sum_{b=-B/2}^{b=B/2} |p_r(x+a, y+b) - p_l(x+a, y+b)| \tag{1}$$

The position (x,y) with the lowest SAD score is considered the point that matches the object in the other image. For example, if we are trying to match the pixel in position $(x_{left}, y)$ in the left image, the corresponding pixel in the right image will be located at:

$$x_r = \arg\min_{i=0\ldots S}(SAD(x_l + i, y)) \tag{2}$$

Once the matching coordinates are found, we can calculate the 3D position in the real-world for any given point using the following equations [2]:

$$X = x_l \frac{T}{(x_l - x_r) * p} \tag{3}$$

$$Y = y \frac{T}{(x_l - x_r) * p} \tag{4}$$

$$Z = f \frac{T}{(x_l - x_r) * p} \tag{5}$$

In the above equations, $T$ is the distance between the two cameras, $f$ is the focal length, $x_l$ and $x_r$ are the horizontal pixel coordinates from the left and right images, respectively, $y$ is the vertical pixel coordinate in both images, and $p$ is the pixel pitch (the distance between two neighboring pixels in the camera).

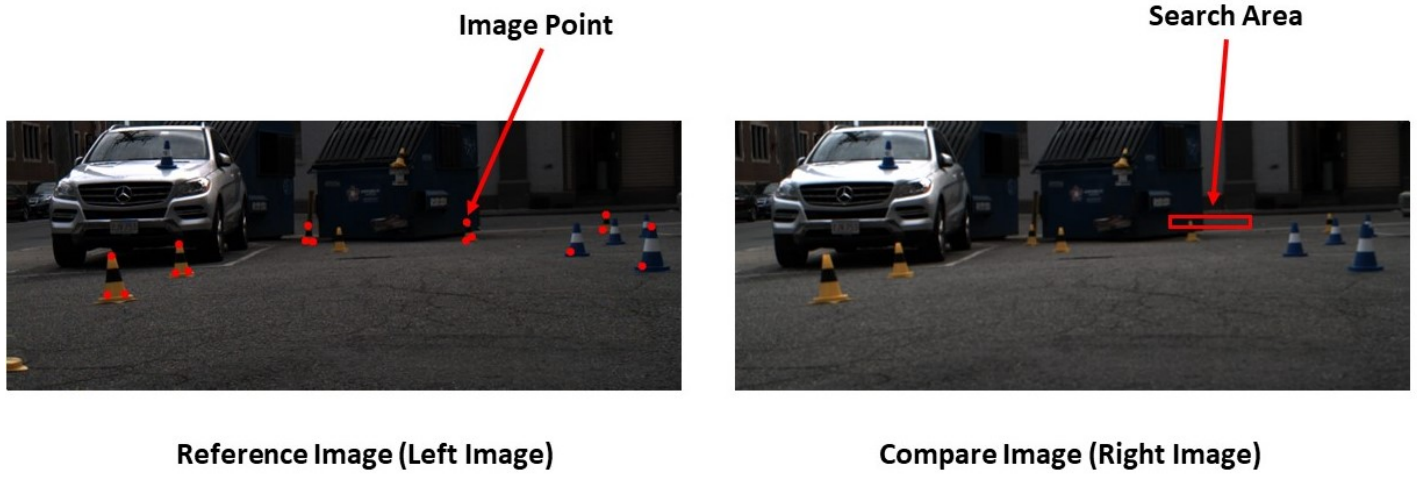


Figure 3: Sample pair of rectified, stereo images. For the Image Point identified on the left, the corresponding search area is shown on the right.

## 3.1 Definitions and Terminology

For the sake of clarity, it is of interest to define the terminology that will be used through the rest of this report. These terms are depicted in Fig. 3.

- Reference Image: This is the image where the target points are computed. In our case, this will be the left image.

- Compare Image: Image where the target points have to be found to compute the real-world position coordinates. In our case, this will be the right image.
- Image Point: These are the coordinates (x,y) specifying the upper left of the block on the reference image that has to be found in the compare image.
- Search area: This is the area (in pixels) around the image point coordinates where the reference block is looked for in the compare image. The search area is S pixels wide by B pixels tall, starting at the Image Point and going left (see Eq. 2).
- An image block is BxB pixels (see Eq. 1).

## 4 High Level Design

Figure 4 shows the high level design of our system. The communication between the host processor and the FPGA will is established through connectal.

The input to the FPGA consists in:

- A pair of stereo images taken from the two cameras in the car. These are RGB images in bitmap format, each of size 800x320 pixels.
- A list of coordinates (x,y) specifying the center of the image blocks whose distance to the cameras we want to compute.

Additional design variables for our design are the block size in pixels (B), the search area (S), and the number of points to compute in parallel (N). These parameters are hard-coded in the hardware implementation.

The high level process for our stereo vision system is depicted in Fig. 5. For every new pair of images taken by the car, the stereo vision performs the following steps:

a. First of all, the two 320x800 pixel images are loaded into the FPGA's DRAM memory using the interface method loadDRAM(). Images are written to memory in row-major RGBA format, with every pixel represented by 32 bits (8 bits for each R, G, B and A). The left image is stored starting at address 0, and the right image starts at location 16384.

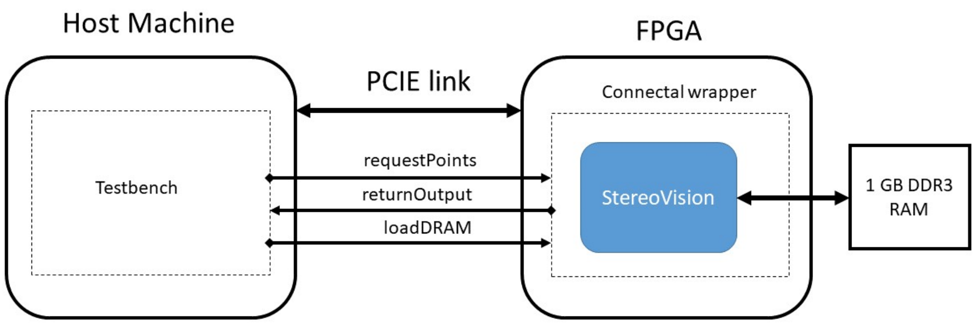


Figure 4: The stereo vision system setup.

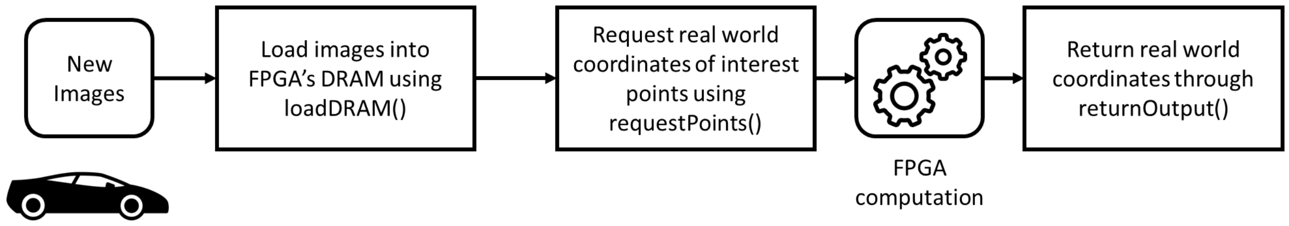

New Images
Load images into FPGA's DRAM using loadDRAM()
Request real world coordinates of interest points using requestPoints()
FPGA computation
Return real world coordinates through returnOutput()

Figure 5: The high level process followed by the stereo vision system. First, the images are loaded into the FPGA memory. Then, the interest points are requested. After the FPGA finishes the computations, it returns the real world coordinates to the host.

b. Once the images have been loaded, we send a list of the image point coordinates of which we want to compute the position in 3D space. This points are sent as a list to the FPGA.

c. Once the FPGA has received the image points, it starts the computation of the real world distances right away.

d. Once the real world distances have been computed, the FPGA returns the computed distances back to the host CPU.

Once a pair of images has been processed, we start the process again with a new pair of images, which are loaded in the same addresses where the old ones where stored.

## 5 Test Plan

We have implemented the stereo vision algorithm in software, both in Matlab and C++. This way, we can compare the output of our hardware implementation to the (correct) software output.

We will use real images taken by the MIT Driverless Racecar as inputs to our stereo vision implementation.
In addition, we wrote unit test cases for all modules to validate functional correctness. These tests cases enabled identification of bugs before they propegated throuhg the latter components.

## 6 Microarchitectural description

Our microarchitectural design consists of a pipelined design for Image Point. This pipelined design is then replicated to compute real-world coordinates for N pixel-coordinate points in parallel. The pipelined design for processing an Image Point consists of multiple components: two LoadBlocks instances, a ComputeScore block, one UpdateScore block, and one ComputeRealWorldDistance block. In addition, we added a DDR3ReaderWrapper to the DRAM interface to tag read requests responses with the address being requested.

### 6.1 DDR3ReaderWrapper

```
typedef Bit#(26)  DDR3_Addr;
typedef Bit#(512) DDR3_Line;

typedef struct {
    Bool write;
```

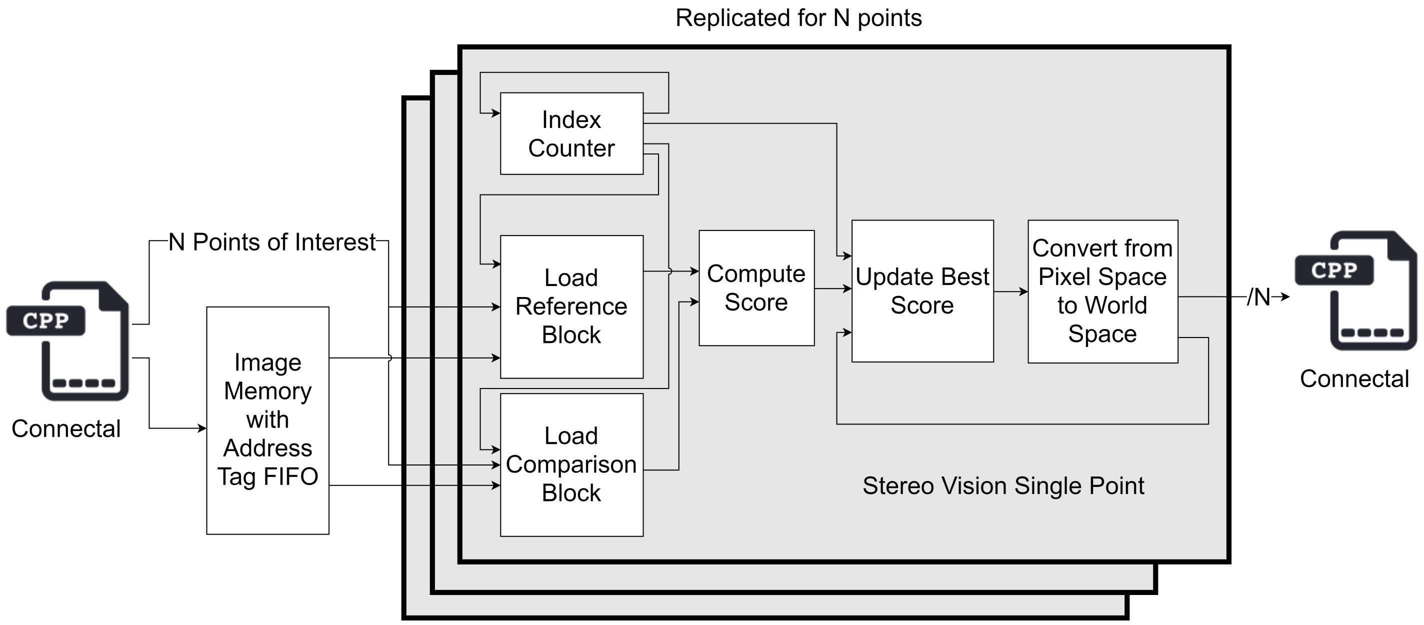


Figure 6: Microarchitecture schematic. A pipelined microarchitecture is chosen. N image points are computed in parallel.

```
    DDR3_Addr line_addr;
    DDR3_Line data_in;
} DDR3_LineReq deriving (Bits, Eq, FShow);

typedef struct {
    DDR3_Addr addr;
    DDR3_Line data;
} DDR3_LineRes deriving (Bits, Eq, FShow);

interface DDR3ReaderWrapper;
    interface Put#(DDR3_LineReq) request;
    method Maybe#(DDR3_LineRes) get;
endinterface
```

This block tags all READ requests to FPGA memory with the address being requested. It is implemented with a large FIFO that records the addresses being requested from FPGA memory. When FPGA memory responds to a memory request, this module dequeues from its FIFO and tags the response with the appropriate address. All READ memory requests are routed through this block, and there is only one instance of it in our entire system. This block enables our system to request from memory from multiple, independent modules without needing to implement a global ordering for memory requests. Without this block or a global ordering, it would not be possible to correlate memory read requests with responses.

## 6.2 LoadBlocks

```
typedef struct {
    UInt#(pb) x;
    UInt#(pb) y;
} XYPoint#(...) deriving(Bits, Eq);


typedef Server#(
    XYPoint#(pb),
    Vector#(TMul#(npixelst, npixelst), Pixel#(pd, pixelWidth))
) LoadBlocks#(...);
```

The LoadBlocks module loads the pixels for a given block from main memory. Each block is a 5x5 region of pixels. The LoadBlocks module is implemented circularly. For each point in the block being requested, it computes the DRAM address where the point is stored and sends a request for that address through `DDR3ReaderWrapper`. Because the DRAM module is serial, we used a circular implementation to request one memory address per cycle.

In a separate rule, on each cycle, this module calls `DDR3ReaderWrapper,get()` to retrieve and process fulfilled memory requests. For each memory address, it computes which, if any, pixels to keep, and then loads these desired pixels into registers. Once all pixels have been returned and loaded, it queues the block of pixels from the block of registers into the response FIFO and dequeues the original request from the request FIFO.

Note that the underlying DRAM component pipelines memory requests. As such, memory requests and responses may be interleaved but are never fulfilled simultaneously. Even when replicated, LoadBlocks component properly handles interleaving of memory requests and responses.

### 6.3 ComputeScore

```
typedef struct {
    Vector#(TMul#(npixelst, npixelst), Pixel#(pd, pixelWidth)) refBlock;
    Vector#(TMul#(npixelst, npixelst), Pixel#(pd, pixelWidth)) compBlock;
} BlockPair#(...) deriving(Bits, Eq);

typedef UInt#(TAdd#(pixelWidth, TLog#(TMul#(TMul#(npixelst, npixelst), pd))))
    ScoreT#(numeric type npixelst, numeric type pd, numeric type pixelWidth);

typedef Server#(
    BlockPair#(npixelst, pd, pixelWidth),
    ScoreT#(npixelst, pd, pixelWidth)
) ComputeScore#(...);
```

The ComputeScore module compares two blocks – one from the reference image, and one from the comparison image – and computes the sum of absolute difference (SAD) in color between these blocks. We refer to this ”difference” as score, and a lower score implies a closer match.

### 6.4 UpdateScore

```
typedef struct {
    ScoreT#(npixelst, pd, pixelWidth) score;
    UInt#(pb) distance;
} ScoreDistanceT#(...) deriving(Bits, Eq);


typedef Server#(
    ScoreDistanceT#(pb, npixelst, pd, pixelWidth),
    UInt#(pb)
) UpdateScore#(...);
```

The UpdateScore module records the disparity for the best, or lowest, SAD score computed thus far for the current image point. It uses registers to record the best disparity and corresponding score, which is updated at the beginning of loading a new Image Point or a better score is found for the current Image Point.

## 6.5 ComputeDistance

```
typedef struct {
    UInt#(pb) x;
    UInt#(pb) y;
} XYPoint#(...) deriving(Bits, Eq);


typedef struct {
    XYPoint#(pb) point;
    UInt#(pb) distance;
} XYPointDistance#(...) deriving(Bits, Eq);


typedef Server#(
    XYPointDistance#(pb),  // pixel x, pixel y, and disparity (distance)
    Vector#(3, FixedPoint#(fpbi, fpbf))  // real-world x, y, and z
) ComputeDistance#(...);
```

The ComputeDistance module computes the real-world distance for an Image Point given the best disparity. Upon compilation, it precomputes a lookup table for $\frac{T}{d*p}$ for each possible d, where T is the distance between the camera lenses and p is the pixel pitch. T and p camera-specific constants. This precomputation enables calculation of the real-world distance via fixed-point multiplication, not division, which significantly shortens the critical path.

## 6.6 StereoVisionSinglePoint

```
typedef struct {
    UInt#(pb) x;
    UInt#(pb) y;
} XYPoint#(...) deriving(Bits, Eq);

typedef Server#(
    XYPoint#(pb),
    Vector#(3, FixedPoint#(fpbi, fpbf))
) StereoVisionSinglePoint#(...);
```

Each StereoVisionSinglePoint module contains two LoadBlocks modules, and one ComputeScore block, one UpdateScore block, and one ComputeDistanece block.

One of the LoadBlocks modules is initialized for the reference image, and the other is initialized for the comparison image. When first processing an Image Point, the reference image LoadBlocks module is invoked, and the pixels for the reference block are stored in registers. After loading the reference block, the module pipelines the processing of each block within the search range. At the beginning of each iteration, the LoadBlocks module for the comparison image fetches the appropriate block from memory; then this block and the pre-loaded reference block are passed into the ComputeScore Block. The result from ComputeScore and the disparity, which is the same as the iteration count, is passed into UpdateScore. Once the pipeline completes, the best disparity and the original Image Point are passed into ComputeDistance to calculate and return real-world coordinates.

### 6.7 StereoVisionMultiplePoints

```
typedef struct {
    UInt#(pb) x;
    UInt#(pb) y;
} XYPoint#(...) deriving(Bits, Eq);


typedef Server#(
    Vector#(n, XYPoint#(pb)),
    Vector#(n, Vector#(3, FixedPoint#(fpbi, fpbf)))
) StereoVisionMultiplePoints#(...);
```

The one StereoVisionMultiplePoints module contains copies of StereoVisionSinglePoint so that multiple points can be computed in parallel. Its request and response interfaces proxy individual points and real-world distances, respectively, to each replica of StereoVisionSinglePoint. This interface is used by Connectal.

## 7 Implementation Challenges

The most challenging part to implement was the interface with the DDR3 memory. We have multiple stereo vision modules working in parallel, all issuing requests to the same memory, so we had to implement a 'scheduler' that would correctly manage all these parallel requests. We also had to modify the interface to the memory to return not only the information stored in the requested address, but also the address itself. This way, each stereo vision module is able to identify if the information being returned by the memory is of interest.

Another challenging issue with the DDR3 implementation was the endianness. The host processor uses the opposite endianness of the FPGA, and therefore when loading the images into the DRAM we need to make sure that this is accounted for.

Finally, we also came up with several issues when implementing the communication between the host processor and the FPGA. Since both the returnOutput() and the requestPoints() connectal methods send lists of values between the FPGA and the host (the former sends a list of real world coordinates and the latter a list of image point coordinates to be processed), we had to figure out a way to send these lists through tte PCIE link using connectal. We first tried using *structs*, but we later settled in the use of vectors as they were more flexible.

Table 1 shows the lines of code for each of the modules we implemented in bluespec.

| Module | Lines of code |
|---|---|
| Types definitions | 97 |
| ComputeScore | 77 |
| ComputeDistance | 52 |
| LoadBlocks | 145 |
| UpdateScore | 51 |
| StereoVisionSinglePoint | 190 |
| StereoVisionMultiplePoints | 71 |
| Connectal Interface (MyDut) | 246 |
| **Total** | **929** |

Table 1: Lines of code used to implement the different Bluespec modules.

## 8 Design Exploration

### 8.1 Precomputation of Division

Our first FPGA design had a critical path of 59 nanoseconds. This critical path originated from division required by the Stereo Vision formula 3. Fixed point division is an extremely expensive operation, for it is an iterative process. Upon further inspection, we realized that the maximum number of possible divisors was the size of the search area (S) plus the block size (B), totaling 55 (pixels). As such, our first design change was precomputing all 55 possible values and results in a lookup table. During runtime, we now only need fixed point addition, subtraction, and multiplication. These changes enabled us to reduce our clock period down to 21 nanoseconds.

### 8.2 Parameter Tuning

With a sufficiently fast clock speed, we then modified our design the number of image points (N) being computed in parallel. Increase the number of points computed that the FPGA can process in parallel would directly increase throughput. However, every additional `StereoVisionSinglePoint` module requires additional hardware resources, Build-times complexity increase because of the additional routing and layout constraints. Table 2 illustrates how hardware and performance metrics change as the number of `StereoVisionSinglePoint` modules increase.

Note that the cycle counts and latency measurements depend on DRAM access delays and vary between runs. The standard deviation for these measurements is less than 5%.

There is no significant difference in latency with these different architectures. The lower hardware utilization of the two-point approach results in significantly faster build times.

In addition, it is also possible to vary the size of the search area or the size of a comparison block through Bluespec static elaboration. Varying such parameters would be trivial, for they only affect the number of iterations (cycles) needed to loop over the search area or

| N | LUTs | LUTs (% of total available) | Cycle Count (excluding image loading) | Latency (ms) (excluding memory loading) |
|---|---|---|---|---|
| 2 | 77852 | 25.64% | 5876 | 0.000233 |
| 4 | 1211152 | 39.91% | 7158 | 0.000222 |
| 10 | 256446 | 84.47% | 4756 | 0.000243 |

Table 2: The effect of varying the number of parallel stereo vision modules, N, on circuit parameters and performance metrics. Cycle counts and latency were measured on the same image with 7 Image Points. This image required 4, 2, and 1 pass, respectively, on N=2, 4, and 10 points in parallel.

a block. They do not materially affect the amount of hardware resources required.

## 8.3 ROS Integration

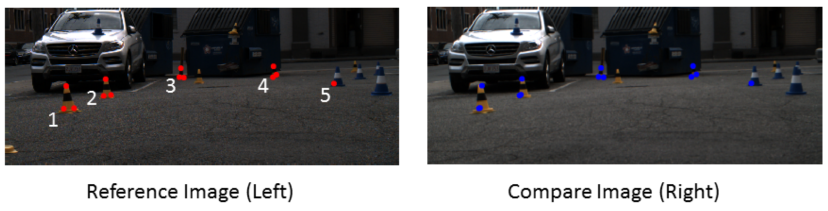
In addition to being able to process images stored in files on the local disk, we integrated the FPGA (through connectal) with the Robotics Operating System (ROS). ROS is a publisher-subscriber framework designed for robotics applications. The stereo vision camera on the MIT Driverless Racecar, along with the neural network which identifies the points of interest, publish this information through ROS messages. We dynamically linked the ROS C++ sdk to connectal interface. Thus, stereo image pairs and points for processing can be sent directly to the FPGA without having to read from and write to disk.

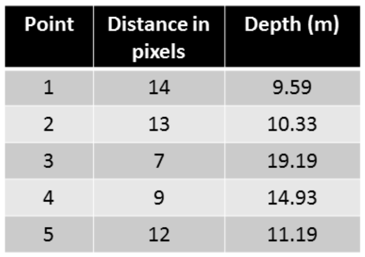
# 9 Performance Evaluation

## 9.1 Correctness

As stated in section 5, it is straightforward for us to evaluate the correctness of our implementation by comparing it to the outputs generated by the pure software implementations we did in Matlab and C++. By doing this, we confirmed that our FPGA implementation produces the correct results (subject to the precision of the fixed point multiplication).

Figure 7 shows an example test case. The red dots in the reference image (the left image) are the input points to the fpga, and the blue dots in the compare image (the right image) show the locations where the fpga located these points. As is observable, the identified points in the compare image match the input target points.





| Point | Distance in pixels | Depth (m) |
|---|---|---|
| 1 | 14 | 9.59 |
| 2 | 13 | 10.33 |
| 3 | 7 | 19.19 |
| 4 | 9 | 14.93 |
| 5 | 12 | 11.19 |

Figure 7: An example output generated by the FPGA. The red dots in the reference image are the input points, and the blue dots in the compare image are the matching points computed by the FPGA.

### 9.2 Area

With N=2 `StereoVisionSinglePoint` modules in parallel, that is, computing 2 target points in parallel, the total LUT utilization is 25.64%.

### 9.3 Speed/Latency

Our design runs with a 21 ns clock period, which corresponds to a frequency of 47.7 MHz. The critical path is in the LoadBlock module, and is set by the interface to the DDR3 memory.

We also recorded the time it takes the FPGA to complete a full computation cycle (loading the two images to the FPGA, requesting the points and receiving the points back). The total time for this process to complete is 141 ms. Of these, 140 ms correspond to the time needed to load the two images into the DRAM (70 ms per image). Once the images are loaded, requesting the target points and getting the computed distances back takes less than 1 ms.

Since in a real system we would not need to copy the images into the FPGA DRAM, but instead we would use Direct Memory Acess (DMA) techniques, the 140 ms taken to load the images into memory (which represents 99.3% of the time taken by our test implementation) does not have to be accounted for. Therefore, the latency of our implementation is $<1$ ms, which fulfils our goal of less than 15ms latency by more than one order of magnitude.

## 10 Conclusion

We have successfully implemented a stereo vision system in an FPGA. By instantiating several modules that perform the stereo vision algorithm on a single point, we can compute multiple points in parallel, speeding up the computation. The time taken by the FPGA to receive the target points, compute the real world coordinates and return them is only 1 ms, which beats by more than 10x our goal of 15 ms latency. Our test demonstration was limited by the long time ($\approx$140 ms) it takes for the two stereo images to be loaded into the FPGA's DRAM, something that would not be necessary in a real stereo vision system implementation.

## References

[1] M. Bertozzi, A. Broggi, A. Fascioli, and S. Nichele. Stereo vision-based vehicle detection. In *Proceedings of the IEEE Intelligent Vehicles Symposium 2000 (Cat. No.00TH8511)*, pages 39–44, Oct 2000.

[2] P. Delmas. Basics of Computational Stereo Vision. `https://www.cs.auckland.ac.nz/courses/compsci773s1t/lectures/773-GG/topCS773.htm`, 2009. Accessed: 26-10-2019.

[3] M. Hariyama, N. Yokoyama, and M. Kameyama. Design of a trinocular-stereo-vision vlsi processor based on optimal scheduling. *IEICE Transactions on Electronics*, 91-C:479–486, 04 2008.