

Roadrunner Final Report

Sean Boerhout

EECS Department

Massachusetts Institute of Technology

Cambridge, MA, USA

seanboe@mit.edu

Claudius Tewari

Mechanical Engineering Department

Massachusetts Institute of Technology

Cambridge, MA, USA

cttewari@mit.edu

Theo Gerst

EECS Department

Massachusetts Institute of Technology

Cambridge, MA, USA

the0@mit.edu

Abstract—We present a design for an optical flow-based VIO (Visual Inertial Odometry) sensor to determine vehicle velocity relative to the ground. Ground velocity, or ground speed, is critical for effective driver-assist or fully-autonomous systems in the automotive industry. We implement the Lucas-Kanade optical flow algorithm [1] [2] in hardware on an FPGA (Field Programmable Gate Array). A ToF (Time-of-Flight) sensor is used to determine sensor height above the ground. The optical flow velocities are fused with an accelerometer and a 9-axis IMU (Inertial Measurement Unit), and results are output over UART to a client. The sensor was compared with a COTS (commercial off the shelf) sensor and velocity outputs at low speeds matched with an error of 0.05m/s.

Index Terms—Field Programmable Gate Array, Optical Flow, Visual Inertial Odometry, Sensor Fusion

I. BACKGROUND

In recent years the automotive industry has been moving rapidly towards driver-assist or fully-autonomous systems for controlling vehicles. As the complexity of these systems grows so does the desire for accurate odometry, a measurement of the vehicle's motion across the ground. One common method of odometry for automotive vehicles relies on the use of sensors monitoring tire rotation. Under the assumption that tires roll without slipping across the ground, this can be used to estimate the vehicle's ground speed. However, slippage is quite common on vehicles in the real-world in conditions with variable road conditions or when the the vehicle is executing dynamic maneuvers.

Given this, research has been put into non-contact methods that measure vehicle ground speed regardless of the state of the tires. The non-contact method pursued in this paper is optical, utilizing a camera pointed at the ground with an algorithm to derive optical flow, the change in the pixels from frame to frame. This optical flow is proportional to vehicle velocity at a fixed camera height above the ground. An inertial measurement from an IMU is also generated and fused with the optical flow velocity to improve its accuracy.

II. TECHNICAL CHALLENGES

Since the optical flow algorithm is observing the change in the pixels of the image from frame to frame, the higher the camera framerate in FPS (frames-per-second) the higher the velocity you can measure. Additionally, running upstream controls on the vehicle velocity output requires as little latency

as possible in the pipeline. This makes an FPGA particularly suited to the task of optical flow image processing.

Optical flow measurements are susceptible to errors through a variety of mechanisms, including difficulty dealing with visual obstructions, featureless environments, and variable lighting conditions. In contrast, inertial measurement units (IMUs) tend to perform consistently, but can be affected by noise and are prone to drift. As such, we implement a complementary filter to fuse the benefits of both odometry techniques to provide the most accurate VIO (visual inertial odometry) vehicle velocity estimate.

III. MATHEMATICAL FOUNDATION

To calculate optical flow we make use of a modified version of the Lucas-Kanade [1] [2] algorithm. The key assumption behind the algorithm is that for a given pixel $f(x, y, t)$, its brightness is constant with a small change in position or time between two frames, i.e.

$$f(x, y, t) = f(x + \Delta x, y + \Delta y, t + \Delta t).$$

Approximating by a first-order Taylor series, we get

$$f(x, y, t) = f(x, y, t) + f_x \Delta x + f_y \Delta y + f_t \Delta t$$

$$0 = f_x \Delta x + f_y \Delta y + f_t \Delta t$$

$$0 = f_x \frac{\Delta x}{\Delta t} + f_y \frac{\Delta y}{\Delta t} + f_t$$

$$0 = f_x v_x + f_y v_y + f_t,$$

where v_x and v_y are the x and y optical flow velocities, estimated as difference quotients. This equation holds for every pixel in the image, and so long as the image is not in significant rotation, all pixels have the same velocity. We have an over-constrained system

$$\begin{bmatrix} f_x^1 & f_y^1 \\ f_x^2 & f_y^2 \\ \dots & \dots \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} -f_t^1 \\ -f_t^2 \\ \dots \end{bmatrix},$$

where f_x^1 denotes f_x evaluated at the first pixel in some ordering of the frame pixels. Applying least squares, we get

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \sum_i (f_x^i)^2 & \sum_i f_x^i f_y^i \\ \sum_i f_x^i f_y^i & \sum_i (f_y^i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_x^i f_t^i \\ -\sum_i f_y^i f_t^i \end{bmatrix}.$$

Therefore, we only need to store five unique summations (denoted by S_{xx} , S_{xy} , S_{yy} , S_{xt} , and S_{yt}) before computing a single matrix multiplication. We estimate the spatial derivatives f_x and f_y by center difference ($f_x(x, y, t) = f(x + 1, y, t - 1) - f(x - 1, y, t - 1)$) while the time derivative is a single difference ($f_t(x, y, t) = f(x, y, t) - f(x, y, t - 1)$). The above matrix equation is what our FPGA design aims to compute, in addition to filtering steps that fuse the flow output with sensor data that will be described later on.

IV. ALGORITHM OVERVIEW

From the analysis above, we break the primary algorithm into four steps:

- 1) Buffering frame data to provide six pixel coordinates: those around the pixel of interest, and the pixel of interest and its past value (spacetime module).
- 2) Taking differences and multiplying to provide $f_x f_x$, $f_x f_y$, $f_y f_y$, $f_x f_t$, and $f_y f_t$ (einstein module).
- 3) Accumulating over the span of a single frame to provide S_{xx} , S_{xy} , S_{yy} , S_{xt} , and S_{yt} (five_spices module).
- 4) Performing matrix operations to finally obtain v_x and v_y , in units of pixels per frame (six_seven module).

From here, we have two remaining goals: converting to real-world units and filtering against an IMU. We can do two operations in parallel with the four steps above:

- Using a ToF sensor to convert the flow from pixels per frame to meters per second based on measured altitude (crocodilo module).
- Reading accelerometer, gyroscope, and magnetometer data from the IMU (eric module).

Once all above steps have completed, the remainder of the algorithm is as follows:

- 1) Convert v_x and v_y into SI units (pix_hawk module).
- 2) Filter v_x and v_y against accumulated IMU acceleration to produce steady velocity values $\overline{v_x}$, $\overline{v_y}$ (steinmeyer module).
- 3) Filter IMU angular rate w_z against magnetometer readings to produce a steady angular rate value $\overline{w_z}$ (zhou module).
- 4) Transmit $\overline{v_x}$, $\overline{v_y}$, and $\overline{w_z}$ in a UART packet (yapper module).

A block diagram is provided in Fig. 1 to demonstrate the idea.

V. MODULE DESCRIPTIONS

Implementation descriptions for less straightforward modules are given below.

A. spacetime

Our design uses the OV5640 camera, which provides RGB565 data for each pixel in raster order. We ignore color and convert to brightness, which means that spacetime receives $f(x, y, t)$ in raster order. Given $f(x, y, t)$ as input, it must output $f(x, y, t)$, $f(x, y, t - 1)$, $f(x - 1, y, t - 1)$, $f(x, y - 1, t - 1)$, $f(x + 1, y, t - 1)$, and $f(x, y + 1, t - 1)$. To do this, it stores a full BRAM frame-buffer of camera input which it overwrites

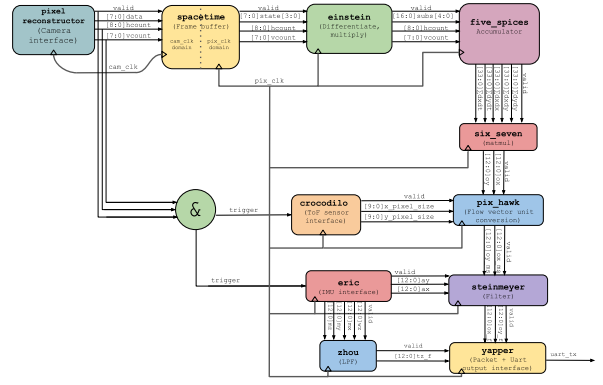


Fig. 1. Block diagram of the Roadrunner pipeline.

as it processes the current pixel, in addition to another BRAM line buffer that stores the $y, y - 1$ rows from the previous frame. Seven BRAM operations are needed in total: reading the five non-current values, writing index (x, y) into the frame buffer, and overwriting the same index with the current value $f(x, y, t)$ in the updated frame. Since our BRAM is dual-port and we utilize two of them, we can accomplish this in four clock cycles (two BRAM cycle times). Note that reading $f(x, y, t - 1)$ and writing $f(x, y, t)$ must not occur in the same step, as they access the same index of the BRAM.

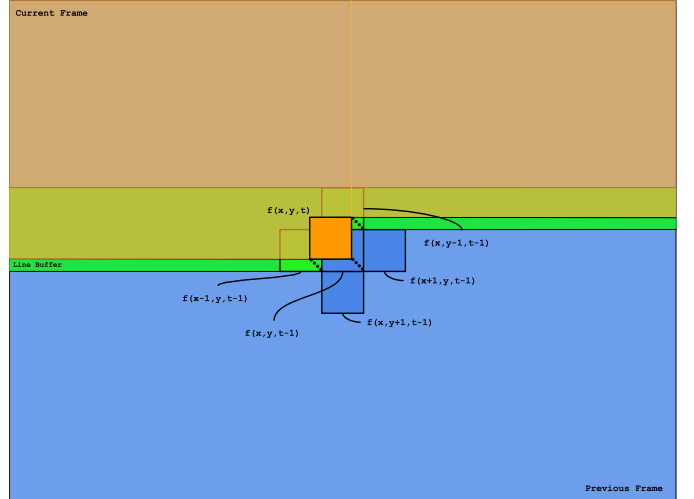


Fig. 2. Illustration of the spacetime BRAM state.

Fig. 2 demonstrates the state of the buffer at the time of operation on (x, y, t) . The orange section represents the pixels in the current frame that have been written to BRAM, and the light blue represents the past frame in BRAM that is being overwritten. The light green represents the frame buffer storing the pixel values of the past frame (that get overwritten by orange) since we output pixel values from the past frame.

B. six_seven

Expanding the matrix equation for v_x and v_y , we get

$$v_x = \frac{1}{\det A} (S_{xy} S_{yt} - S_{yy} S_{xt})$$

$$v_y = \frac{1}{\det A} (S_{xy}S_{xt} - S_{xx}S_{yt}),$$

where $A = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix}$ and so $\det A = S_{xx}S_{yy} - S_{xy}S_{xy}$. Therefore `six_seven` must perform a total of 6 multiplications, 3 subtractions, and 2 divisions. We divide the process into a 4-state FSM, with a nested FSM for each divider. The FSM and module diagram are shown in Fig. 3 and Fig. 4, respectively.

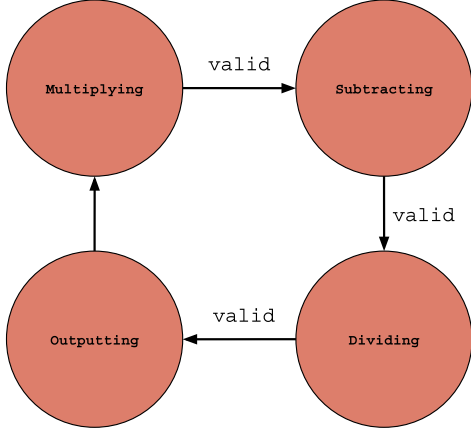


Fig. 3. Finite state machine for the `six_seven` module.

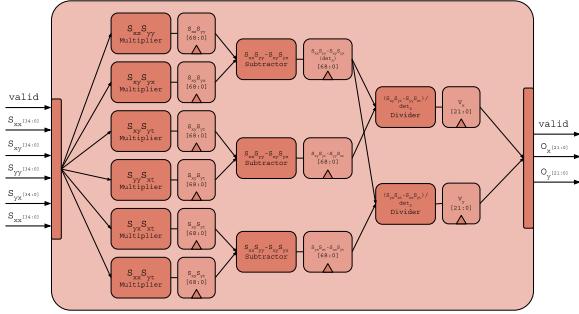


Fig. 4. Cutaway view of the `six_seven` module.

C. crocodilo

We use a VL53L0X ToF sensor to determine our altitude. Unfortunately, its I2C interface is highly complex and its drivers are proprietary, so we use a manufacturer-provided interface library, running on a micro-controller, to read its registers and transmit altitude in a predictable format as an I2C slave (on a separate bus, connected to the FPGA). A separate custom I2C driver, `I2C`, is used to handle the transaction - our API for it is documented here. Once the altitude is obtained, we can compute pixel size by the geometry shown in Fig. 5.

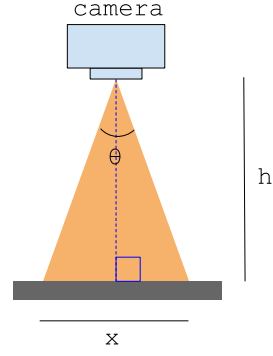


Fig. 5. Geometry of the camera altitude.

This gives us $2h \tan(\theta/2)$ for the width of the frame, so the pixel width is given by $2h \tan(\theta/2)/N$, where N is the width of the frame in pixels. An identical calculation can be performed for the pixel height. Note that since $2 \tan(\theta/2)/N$ is known at compile time, `crocodilo` just needs to multiply altitude by a constant to determine pixel size.

D. zhou & steinmeyer (filtering methodology)

We use an IIR filter on two inputs to do sensor fusion and remove high-frequency artifacts. If $x(t)$ and $y(t)$ are inputs, then we set

$$z(t) = \alpha x(t) + \beta y(t) + (1 - \alpha - \beta)z(t-1),$$

where α and β are determined on bench.

For `zhou`, the angular rate filter, we would set $x(t)$ to the angular rate from the IMU and set $y(t)$ to the derivative of $\theta_z = \arctan(-\frac{m_y}{m_x})$, where m_y and m_x are inputs from the magnetometer. We would use a custom CORDIC to compute the arctangent. Here, fusing gyroscope data with magnetometer data allows us to eliminate steady state drift from the gyroscope. Unfortunately, we found while testing that both the gyroscope and magnetometer cause issues in the IMU (see the Modifications section), so `zhou` was not implemented.

For `steinmeyer`, the velocity filter, we set $x(t)$ to the optical flow velocity v_x and $y(t)$ to the time integral of a_x , the x -acceleration from the accelerometer. Another filter is implemented for v_y and $\int a_y dt$. Fusing this data complements our optical flow velocity with data from a sensor that has already been validated. In addition, we use $z(t-1) + a_x(t)\Delta t$ instead of $\sum_t a_x(t)\Delta t$ to compute the integral of a_x , mitigating steady-state error from an unchecked integral.

E. yapper

Although a production device only needs to output the finalized, filtered optical flow values, we included many intermediate outputs as pictured in Figure 6 to allow for better system evaluation.

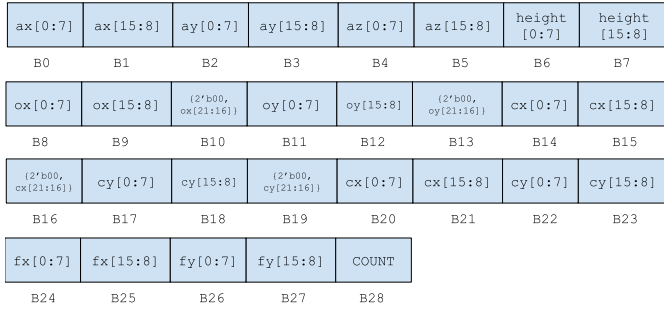


Fig. 6. UART buffer format, after implementation. ax, ay, az: IMU acceleration. ox, oy: raw optical flow. cx, cy: optical flow in m/s. fx, fy: filtered velocity. Last byte is a message counter.

VI. EVALUATION

A. Modifications

A few key modifications were made to the design during implementation. First, we had to disable the gyroscope and magnetometer, since the former introduced noise to the accelerometer (due to chip power throttling) and the latter was difficult to interface with. Due to the nature of the ICM20948, the magnetometer and accelerometer/gyroscope are on separate dies. SPI interfacing with both requires a complex configuration setting up the accelerometer as an I2C master that forwards magnetometer data to its own memory. We were unable to debug this system, and though we implemented a circular, hardware-tested CORDIC to achieve the heading output, this effort wasn't fruitful. That said, our linear velocity pipeline is fully functional. With respect to the optical flow pipeline, we chose to crop the summation viewing window (instead of taking the entire camera output) since the wide camera FOV can view components on the testbed, which are static and disrupt the pipeline. Finally, many of our bitwidths changed due to small adjustments in our pipeline. An updated block diagram is shown in Figure 7.

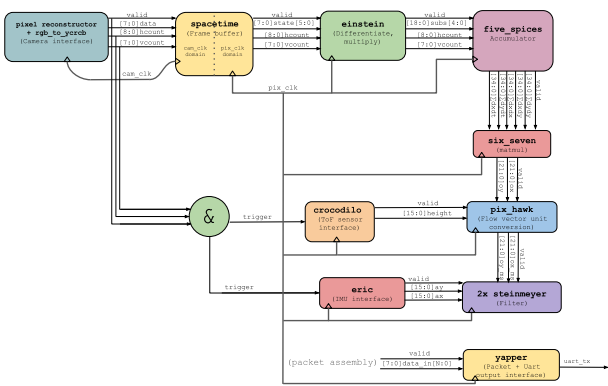


Fig. 7. Block diagram of the Roadrunner pipeline, after implementation.

B. Test Benches

We utilized testbenches on practically all of our modules to save FPGA debugging time and to verify theoretical performance. Some testbenches, such as `cordic_sim`,

utilize assertion checks for automated validation. Others, primarily testbenches for communication controllers (such as `crocodile`, `i2c`, and `yapper`), only output waveforms for manual functionality testing. A testbench that was particularly critical to the design of our system was that for `steinmeyer`, our filtering module. Though complementary filters aren't very complex, testing our design under different simulated dynamics and edge cases exposed errors in fixed-point math (integration and weight multiplication, specifically) that we wouldn't have caught. In addition, it was critical to our efforts in tuning filter weights.

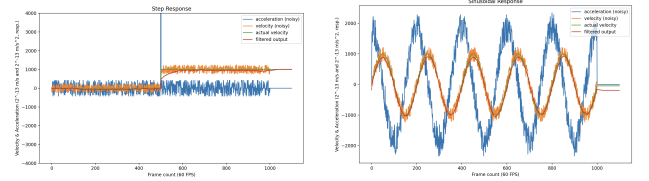


Fig. 8. `steinmeyer` testbenches simulating the hardware filter's reaction to a step response and sinewaves.

C. Hardware Testing

In order to evaluate our system, we attached our electronics to a testbed used by MIT's FSAE racecar team. Although we originally planned on using our MY25 race car for testing, we had an incident that caused the high-voltage battery to break, preventing us from being able to run it.

We use a custom PCB from MIT FSAE's car called a "sensornode" to forward data from the OMS Race (which has a CAN output) to USB. Roadrunner is configured to send sensor readings (IMU and time-of-flight), raw optical flow values, and outputs from the filter over USB as well. A client laptop was connected to both USB networks, and we used a multi-threaded python script to graph received values in real time.

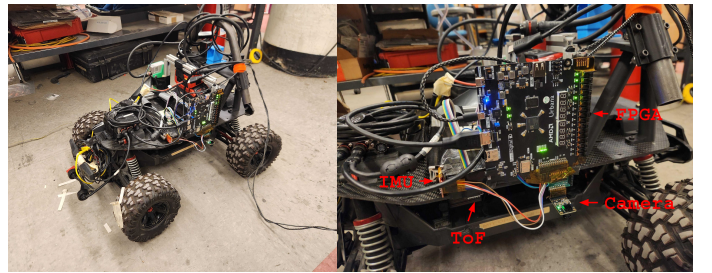


Fig. 9. The testbed used for evaluation of the roadrunner system.

Although the error compared to the OMS Race varied with the velocity pattern, it tends to be no more than 0.05m/s with a time delay of 0.1s (Figure 10). However, we noticed that the OMS Race sometimes behaves strangely since it is designed for much higher speeds, so we also compared the raw optical flow output to integrated IMU acceleration to determine whether the two filter inputs agree. As shown in Figure 11 (left), the signals agree quite well, with a maximum error of roughly 0.05m/s and no time delay. By filtering the signals

together, we can avoid the short spikes on the optical flow signal and the accumulated drift that results from integrating IMU acceleration, shown in Figure 11 (center).

Additionally, we noticed that the OMS Race has limited precision at low speeds, and low amplitude oscillations (created by shaking the testbed) were picked up by our sensor but not by the OMS - see Figure 11 (right).

D. FPGA Resource Allocation

Our pipeline consists of 35 DSP blocks. The vast majority of the DSP blocks are used in `six_seven` to compute the large 68-bit multiplications. In its original implementation, `six_seven` shifted the factors down by 16 bits (down to 18 bits, the size of the multipliers) and the product back up by 16 to reduce resource allocation at the expense of computational accuracy. We later determined that this error was too significant, however, and chose to take full-fidelity products instead. Given the accuracy of our final system, we consider this to be a valuable tradeoff.

Throughout Roadrunner, 49 RAMB36 blocks and 2 RAMB18 blocks are utilized for caching different aspects of the camera frame. 32 RAMB36 blocks are used in the camera frame buffer from class, with most of the rest being used in `spacetime` to cache the pixel frame and linebuffers. Due to the nature of the linebuffer and the necessity of recording the full past frame to our system, we believe that `spacetime` is an efficient use of these RAM blocks.

E. Throughput and Latency

We meet timing with 0.597ns of slack. Overall, our system has a measured frame buffering latency of 120ns (12 seconds) per pixel, a computational latency of 37us, a data offload (USB transmission) latency of 31us, and a throughput of 25FPS (limited by the camera) in normal operation. The only limiting factor in our latency is the camera itself, which is relatively slow - it takes 16 cycles to transmit a pixel from the camera (computationally, it takes us 12 cycles to process it), adding up to approximately 9ms latency across all 57,600 pixels. Our 12 cycle processing speed allows for a theoretical maximum latency of approximately 6.5ms. Given our goal of designing a system with a low-enough latency for race controls, and the fact that our MIT FSAE racecar runs vehicle controls at 100Hz, we were able to meet our system latency target.

VII. CUSTOM BOARD

The camera we were given in class is the OV5640. This is a rolling shutter (the frame is acquired pixel-by-pixel) CMOS sensor that can output 16-bit RGB565 around 25FPS with the settings given in the rom file for the class. However, rolling shutter cameras often leave skew artifacts on the image making it difficult to run precise optical flow at higher speeds. A solution to this is a global shutter camera, which acquires every pixel in the frame at the same time. Most COTS optical flow modules use a very high FPS (500-1000FPS), low resolution, global shutter camera. Given this, we wanted to interface with the OV7251, a global shutter camera that can do up to

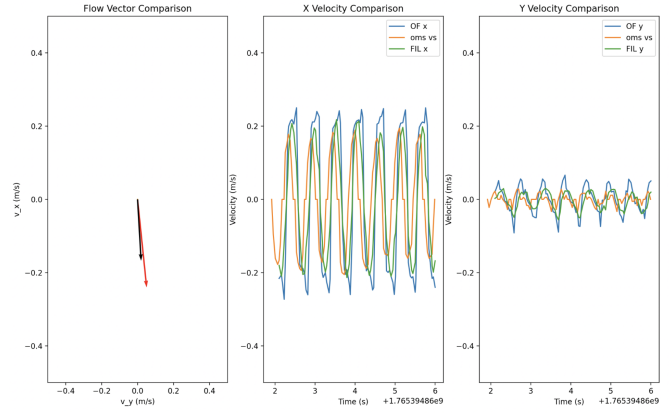


Fig. 10. The live plotting interface we used to evaluate our system.

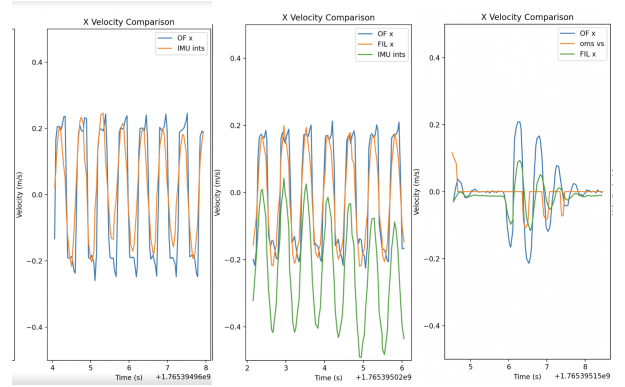


Fig. 11. All images compare the X-axis for brevity. Left: Comparison of raw optical flow versus integrated IMU acceleration. Center: Comparison of filter inputs to filtered value. Right: Comparison of sensor precision.

360fps at 160x120. This proved difficult to do with the Urbana development board from class since it didn't have the MIPI CSI-RX interface we would need to run the full 800Mbps out of the camera. Instead, we designed a custom shield for the Alchitry Au V2, a FPGA development board based around an Artix 7 [12]. The custom board has a MIPI D-PHY, an IMU, a level shifter to convert the LVCMOS33 out of the FPGA to 1.8V signals for the IMU, an a HDMI port for video out, and a few buttons for interfacing. We designed the board in Altium, sent it out with JLCPCB, and assembled it by hand in the FSAE shop.

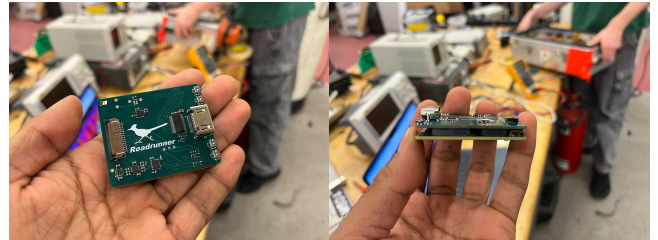


Fig. 12. The custom PCB and Alchitry development board designed for the roadrunner system.

A. MIPI

Interfacing a 7-series FPGA with a MIPI CSI camera is tricky. MIPI has both high speed and low speed signals sent over a single differential pair so a DPHY is required to combine them and ensure proper logic levels. The UltraScale series of FPGAs has built in MIPI D-PHYs but the older 7-series does not. Thus, we needed to use a resistor network to create a custom D-PHY on our custom board as discussed in XAPP894. We also had to verify that the I/O standard we were using for the MIPI pins worked with the I/O banks broken out on the Alchitry. We ended up putting the MIPI pins on Bank 34 set to a VCCO voltage of 2.5V. This allowed us to run the MIPI high speed signals with LVDS_25 and the low speed signals at HUSL_12 (HUSL is compatible with any bank voltage and only depends on a 0.6V internal V_{REF} which LVDS_25 works with).

B. Challenges

We brought up all the voltages rails and the level shifter on the custom board and were successfully able to program the Alchitry. However, we ran into a lot of difficult getting the MIPI IP to work with Vivado due to clock routing issues internal to the IP module. These were very difficult to debug since most of the internals of the MIPI CSI-RX IP are hidden so we did not have time to finishing bringing up the custom board. We plan to finish the bring up in the future and push the limits of our pipeline.

VIII. CODE REPOSITORY

<https://github.mit.edu/6205F25/fa25-6205-team22>

IX. INDIVIDUAL CONTRIBUTIONS

A. Project

Theo wrote `einstein`, `spacetime`, `five_spices`, and `steinmeyer`. Sean wrote `crocodilo`, `pix_hawk`, `eric`, `yapper`, and our unimplemented CORDIC. Claudius did the algorithm research, wrote `fast_divider`, `s_divider`, and designed our custom board. Theo and Claudius collaborated on debugging `spacetime` and the test bench software, and Sean and Claudius collaborated on writing and debugging `six_seven`. All of us worked on `top_level`.

B. Report

Theo wrote the algorithm overview. Sean wrote the evaluation sections. Sean and Theo wrote the module descriptions. Claudius wrote the background, and hardware testing sections. We all collaborated on the figures and images.

REFERENCES

- [1] B. Lucas and T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision," Proc. 7th Intl. Joint Conf. on Artificial Intelligence 1981, pp. 674–679, August 1981.
- [2] D. Patel and S. Upadhyay, "Optical Flow Measurement using Lucas kanade Method," International Journal of Computer Applications, vol. 61, no. 10, pp. 6–10, January 2013.