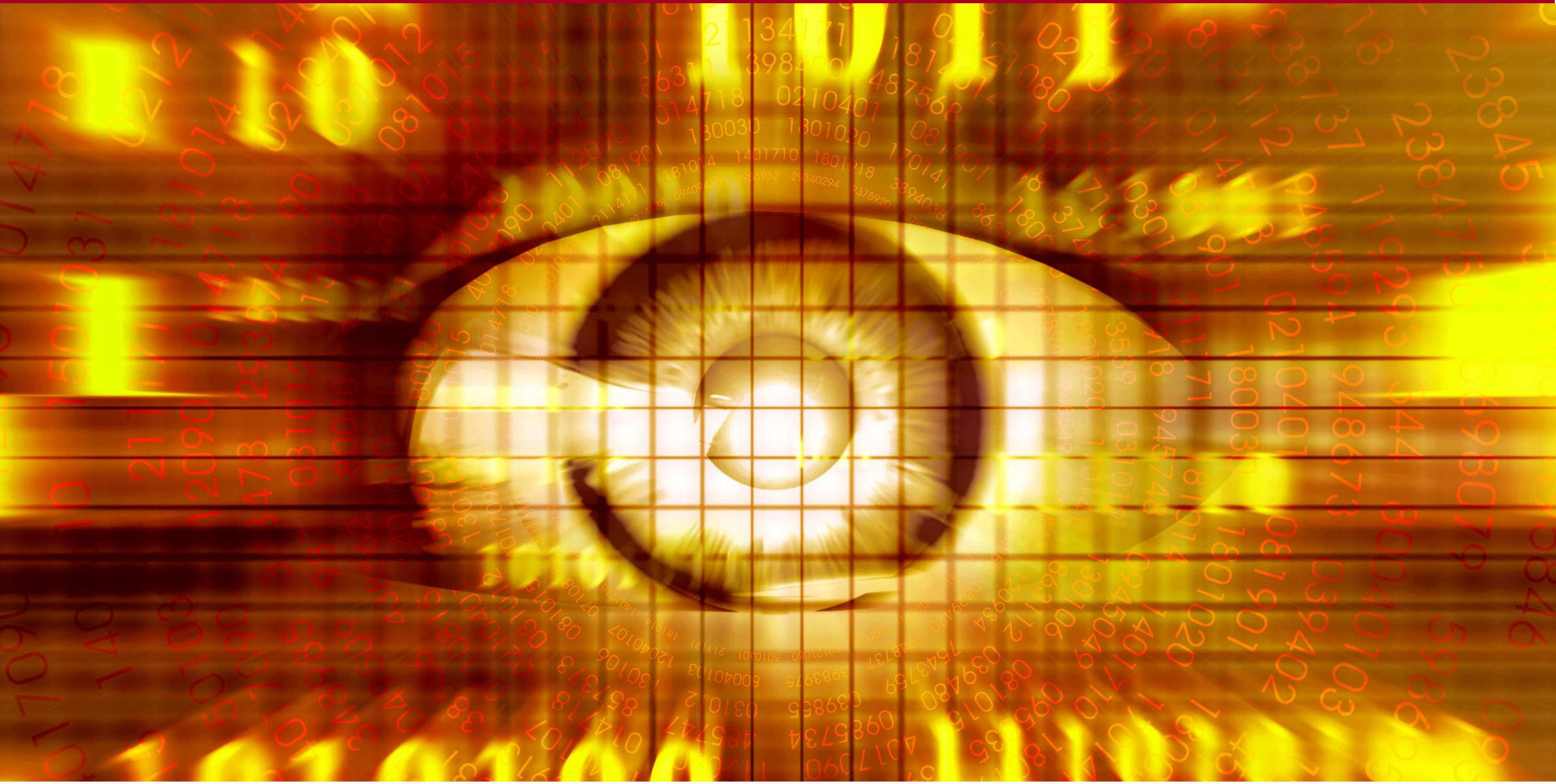
Real-time Computer Vision on Quadcopters



Names: Erick Garcia
Project group: Computer Graphics Laboratory
Mentors: Patrick Hanrahan, Jonathan Ragan-Kelley, Niels Joubert.

Computer Vision

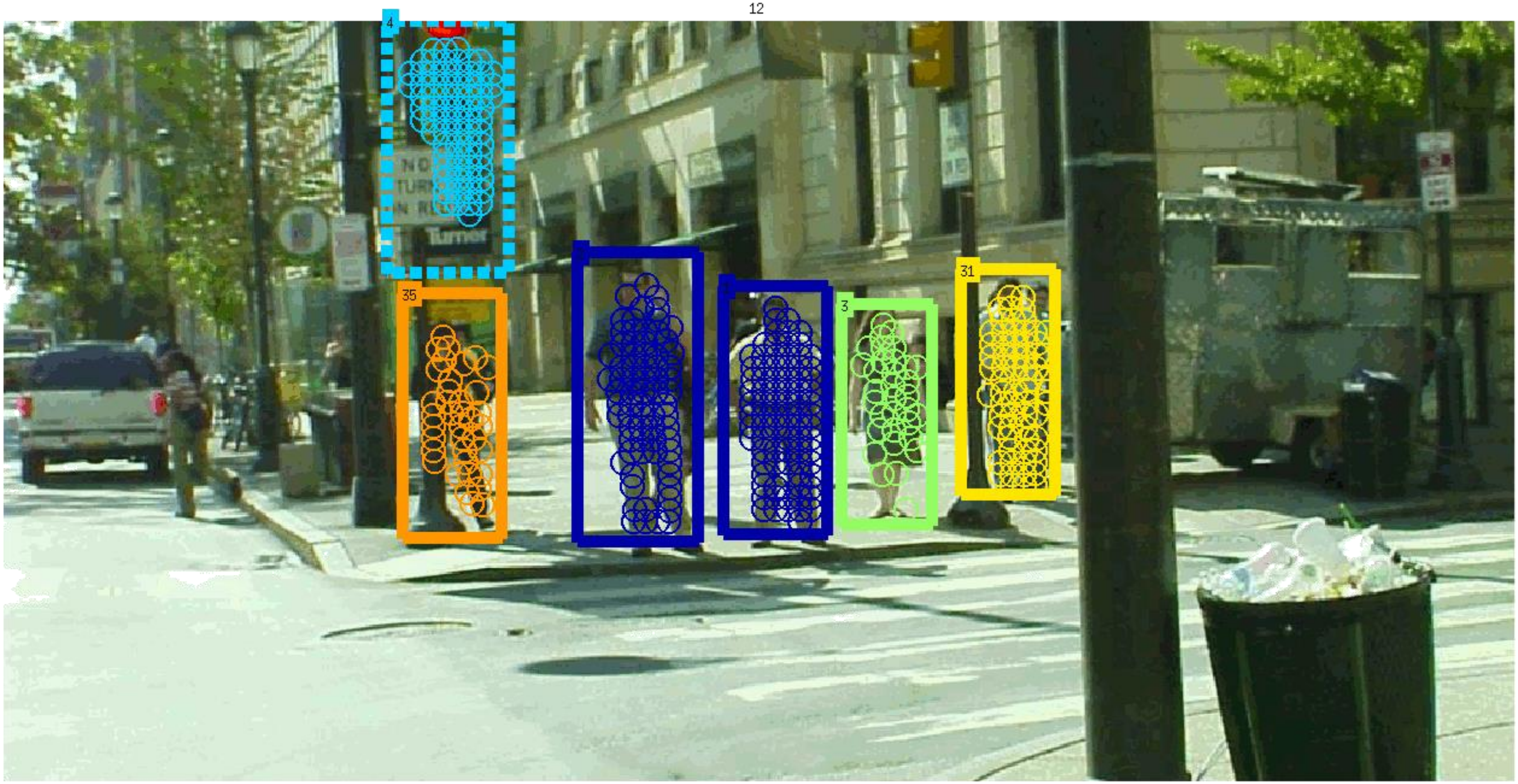
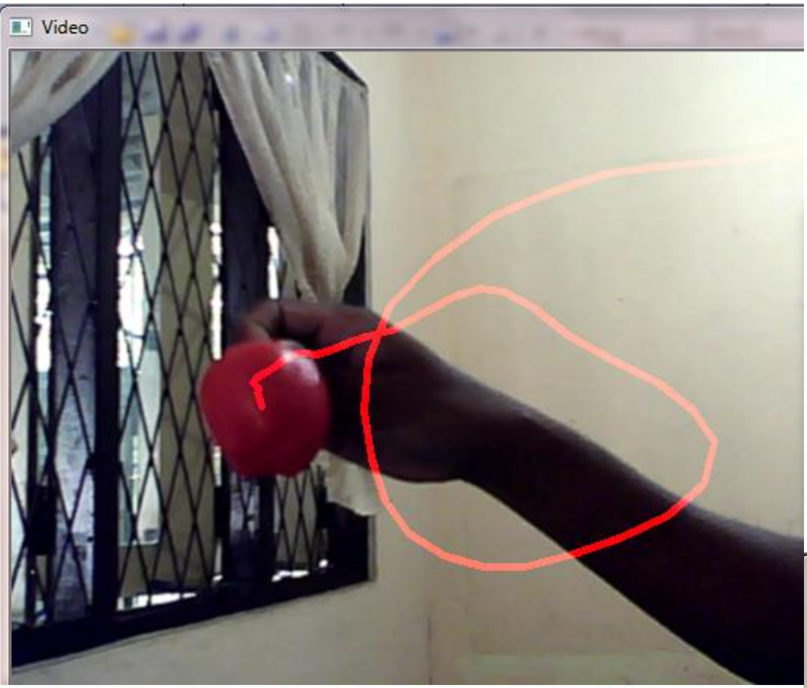
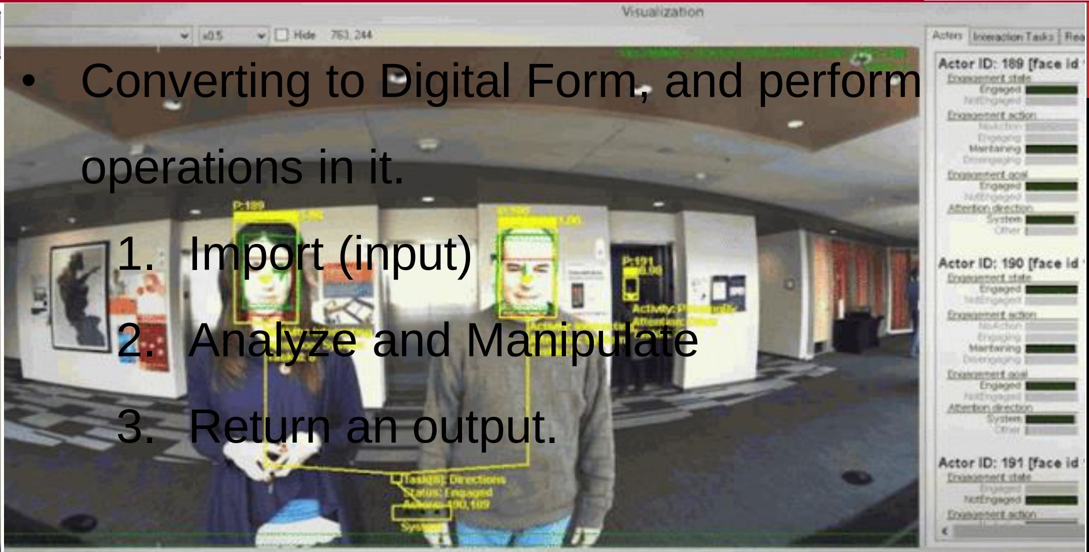


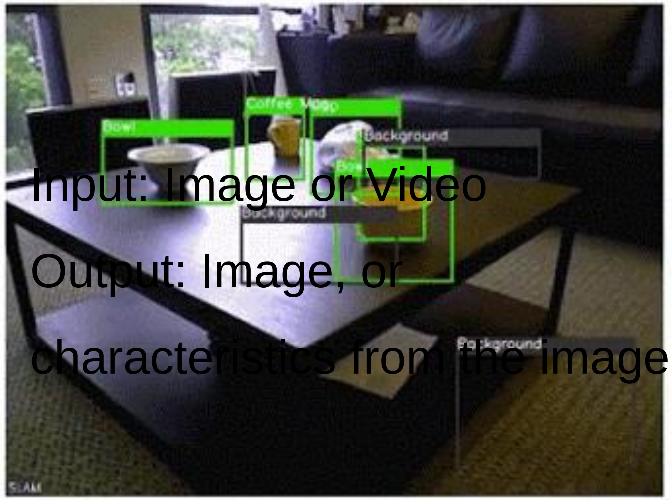
Fig. Different people being recognized as separate objects by a camera

Image Processing





- Converting to Digital Form, and perform operations in it.
  1. Import (input)
  2. Analyze and Manipulate
  3. Return an output.



- Input: Image or Video
- Output: Image, or characteristics from the image.

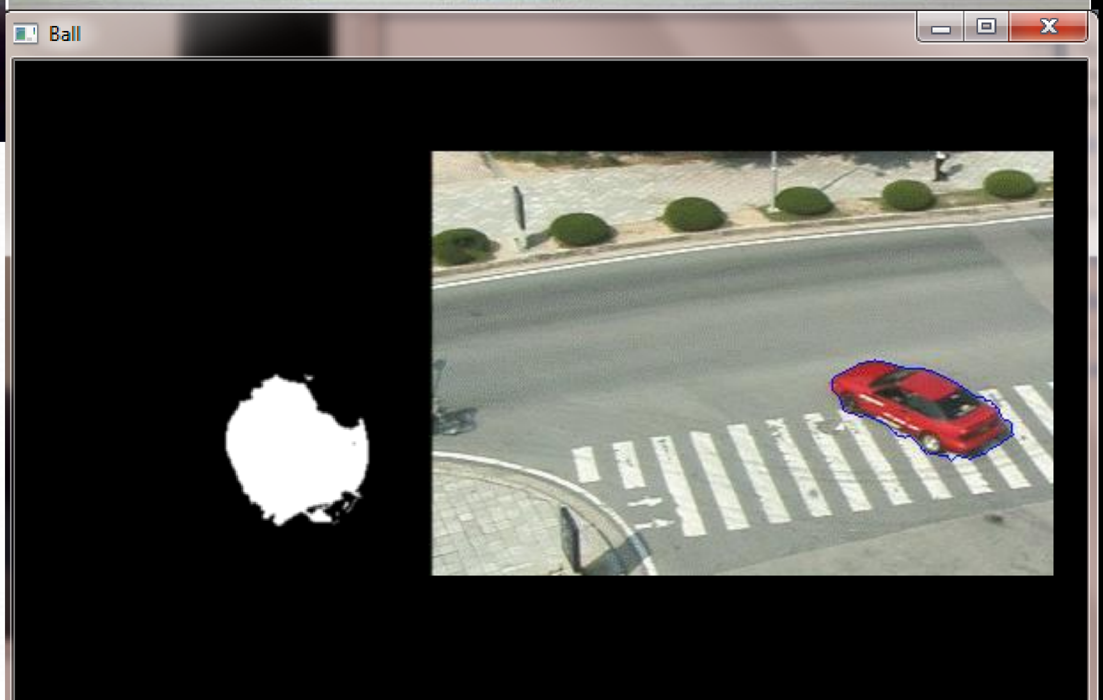


Image Processing

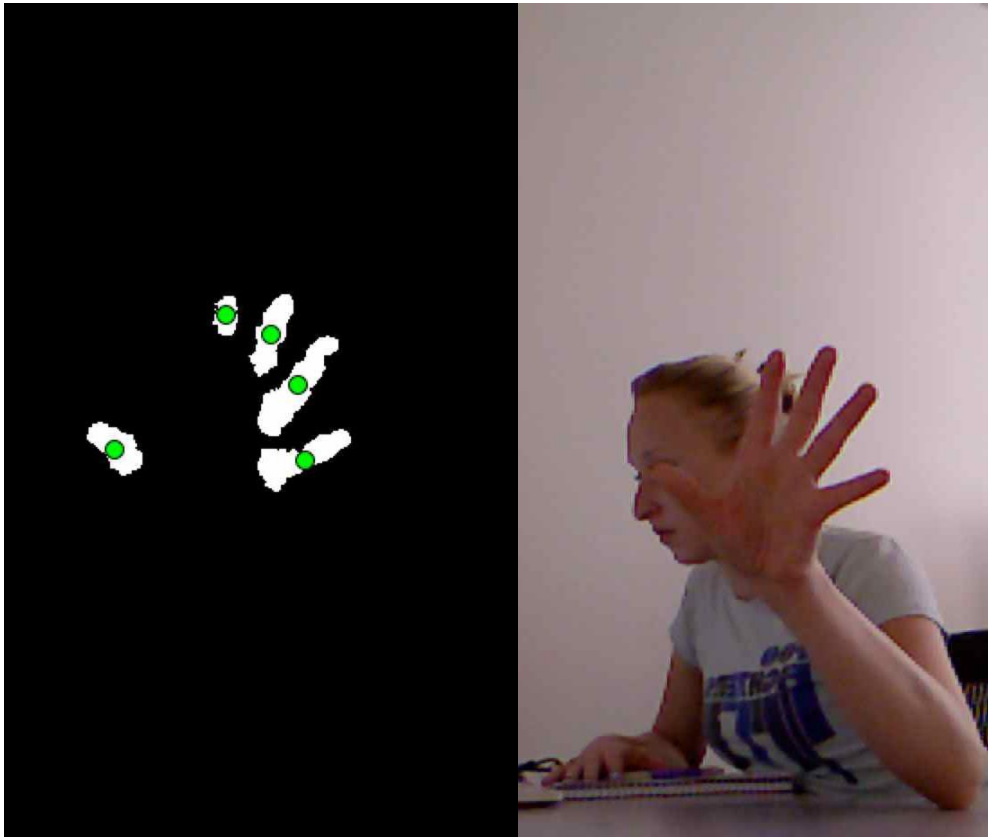


Fig. Abstraction and segmentation of a human hand with xx filter

Applications

- Navigation
- Detecting
- Interaction
- Automatic

Challenges

- Which image components "belong together"?
- How can objects be recognised without focusing on detail?

3DR Solo

Fig. 3DR's Quadcopter the Solo Smart Drone

3DR Solo

Specifications:

- OS: 3DR Poky (based on Yocto Project Reference Distro) 1.5.1
- CPU: ARMv7 Processor
- Camera: GoPro® Hero4
- HDMI Decoder: i.MX6 (h.264 encode/gstreamer)
- Autopilot: Pixhawk 2

3DR Solo

Fig. 3DR Solo Gimball stabilizing GoPro Hero4

Objective

Load an image processing pipeline/library in the ARM Central Processing Unit that will enable developers to load custom code to unleash the 3DR Solo's potential. To go from a real-time video streaming to real-time image processing, thus making the 3DR Solo, the first true smart drone.

Technical Challenges

- Not knowing anything about the 3DR Solo except the information available to the public such as the user manual, and the specs of the device.
- The Solo does not have the option to connect to the internet.
- Our environment did not have gcc or any way to build it into the 3DR.
- These problems would be more easily solvable if the code was Open Source sadly this last release was Closed Source.

Methodology

1. Talking with the 3DR Solo’s developers.
2. SSH (Secure Shell) into the 3DR Solo
3. Research about potentially useful image processing pipelines/libraries.
4. Design a workaround for embedding custom code.
5. Create a Custom Yocto Project Image with our needs.
6. Cross compile the custom code, and execute.

Halide

Halide:
A new programming language embedded in C++. It is a pipeline designed to make it easier to write high performance image procesing code on modern machines.

C/C++ is slow

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate temporary array

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

9.96 ms/megapixel
(quad core x86)

Hand-optimized C++

9.9 → 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
         a = _mm_loadu_si128((__m128i*)(inPtr-1));
         b = _mm_loadu_si128((__m128i*)(inPtr+1));
         c = _mm_load_si128((__m128i*)(inPtr));
         sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
         avg = _mm_mulhi_epi16(sum, one_third);
         _mm_store_si128(blurxPtr++, avg);
         inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Fig. Comparing the speed of un-optimized image processing code (9.96 ms/mp) vs optimized code (0.9 ms/mp)

Halide

The resulting code is:

- Simple
- Well Structured
- Easy to read
- Efficient

Fig. Comparing the speed of un-optimized image processing code (9.96 ms/mp) vs optimized code (0.9 ms/mp)

Halide

Halide

0.9 ms/Megapixel

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

Hand-optimized C++

9.9 → 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
         a = _mm_loadu_si128((__m128i*)(inPtr-1));
         b = _mm_loadu_si128((__m128i*)(inPtr+1));
         c = _mm_load_si128((__m128i*)(inPtr));
         sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
         avg = _mm_mulhi_epi16(sum, one_third);
         _mm_store_si128(blurxPtr++, avg);
         inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Fig. Comparing Halide optimized code vs Hand-optimized C++ code

Results

A method of getting custom image processing code running in the ARM processor of the 3DR Solo, for the purpose of achieving Real-time Computer Vision. This opens up a whole new level of possibilities of image processing and drones in general. All the possible practical applications for the truly first Smart Drone.

Future Work?

Thank you

I wish to sincerely thank the Army High Parallel Computing Center for providing us with this wonderful research opportunity and all the knowledge I got from this experience.