Wilfrid Laurier University

CP493A Winter 2017

Raspberry Pi Focus Stacking Camera

Author:

Harold Hodgins

Supervisor:

Dr. Hongbing Fan

April 22, 2017

Contents

1 Introduction

We made a focus stacking camera using a Raspberry Pi 3.0 to take the pictures to set the focus. We also wrote code to stitch the images together and investigated using the Raspberry Pi GPU to to speed up the focusing stacking process.

2 Background

2.1 Focus Stacking

Focus stacking is a method used to increase the depth of field of an image by combining several images[1] with different focal points into a new image. Usually the camera is fixed on a tripod[2] and the focus is changed or it's attached to a macro rail which moves the camera towards the subject[3].

Figure 1: An example of a focus stacked image [1]

2.2 The Raspberry Pi Computer

The Raspberry Pi is a credit card sized computer which retails for ∼$35. The latest version (3.0) has following key specs[4] :

- A 1.2GHz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- 1GB RAM

- 40 GPIO pins
- Camera interface (CSI)
- VideoCore IV 3D graphics core capable of 24 GFLOPS[5][1]

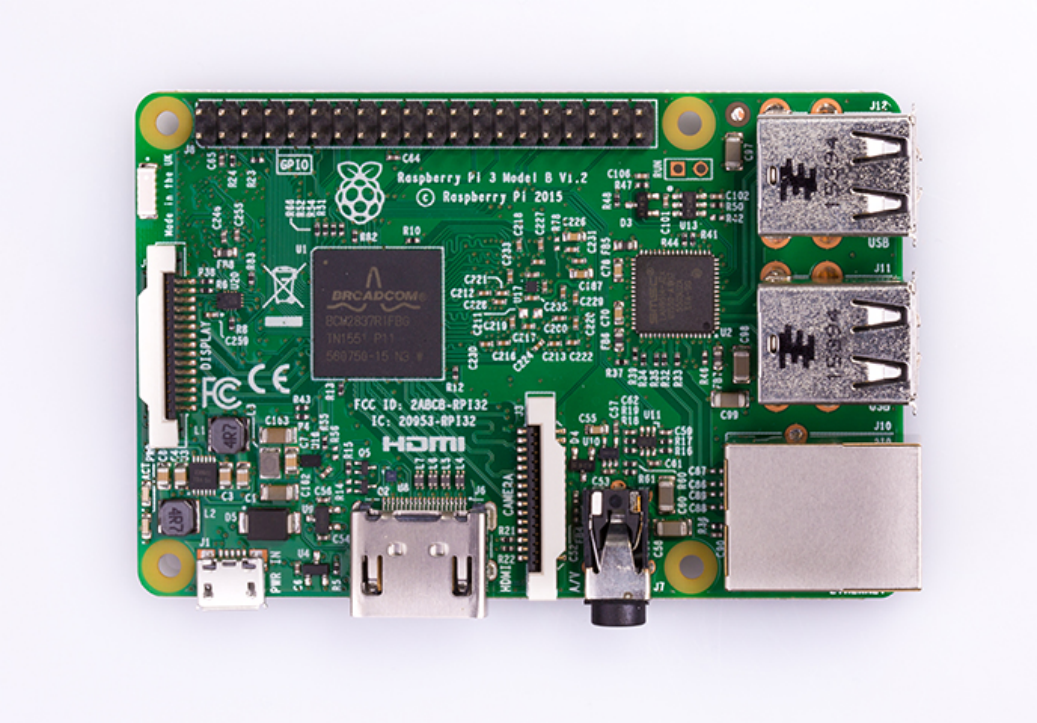
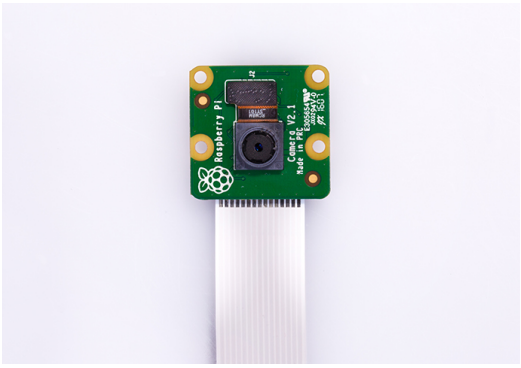


Figure 2: Raspberry Pi 3.0 [4]

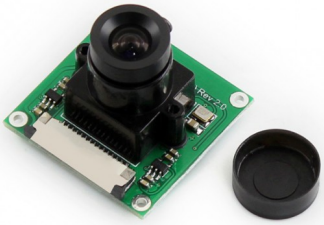
[1]More details on the GPU can be found in Appendix A.

2.3 The Raspberry Pi Camera Module

An 8-megapixel camera module with adjustable focus is available which connects to the Pi via the CSI. Alternatively a 5-megapixel camera module is available from Waveshare Electronics which connects using the interface.



(a) Rasberry Pi camera module[6]



(b) Wave Share camera module[7]

Figure 3: Raspberry Pi camera modules

3 Previous Work

In 2013 David Hunt made a macro rail using a Raspberry Pi to control the stepper motor of a flatbed scanner and to trigger the shutter on a DSLR camera mounted on the scanner rail [3][8]. He then used CombineZM to stitch the images together.

Figure 4: Examples from David's work[3]

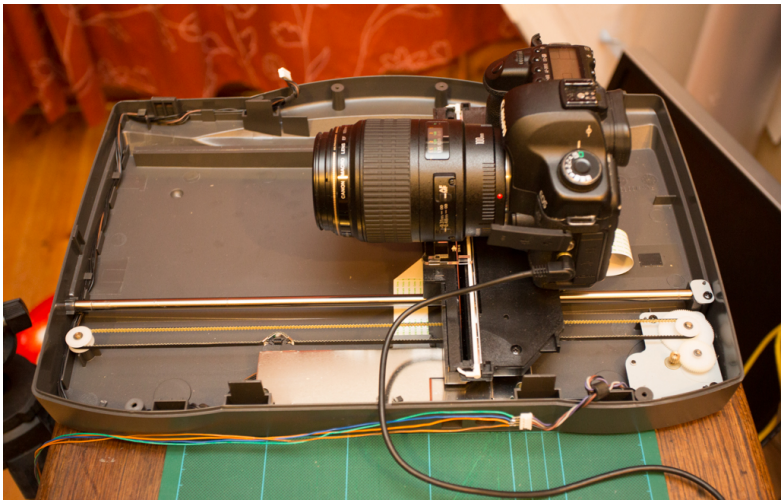


Figure 5: David's macro rail[3]

4 Current Work

4.1 Remote Access and Control of the Pi

We wanted to be able to remotely access the Pi so it wasn't always tethered to a monitor, keyboard, and mouse. We also we wanted to make the final prototype portable and controllable via a web interface. We started by configuring the Pi to auto connect to the local wifi on startup.

4.1.1 Particle IO

"Particle is a scalable, reliable and secure Internet of Things device platform..." [9]. Specifically it's a cloud based platform for remotely programming and controlling microcontrollers and devices like the Raspberry Pi. It's currently free for up to 25 devices, 5 team members and 250K events/month[10].

We signed up for access to the Raspberry Pi Beta and followed the instructions to download and install the Particle daemon. We tested the Hello World example and decided that it wasn't practicle for controlling things like the motor remotely since the function responsible for web requests takes a string as the parameter which then has to be parsed.

4.1.2 Dataplicity

Dataplicity is a web service that lets you access your Pi from any web browser. It gives you a web based terminal[2] and allows you to redirect web traffic to the Pi securely. It's currently free for your first device and has no restrictions on the number of connections.

After signing up and installing the client we were able to get their sample code for live streaming the camera to a webpage working in a few hours[11][3]. The only downside is that the client currently only supports forwarding port 80. All other ports are blocked so if you want to run two web things on different ports it won't work[4].

4.1.3 Flask

We wanted a web interface for the camera and decided to use Flask which "... is a microframework for Python based on Werkzeug, Jinja 2 and good intentions"[12] . After getting the Hello World example to work we modified it to drive the motor n steps forward with a x ms pause in between each step.

[2]With non-root user access by default.

[3]Most of that time was used compliling a few libraries.

[4]It might be possible to set Apache to handle different pages with different programs, but I haven't tried this yet.

In the end we ran out of time and did not get a full web interface for the camera working.

4.2 Hardware

4.2.1 Camera Housing

The housing for the Rapsberry Pi was designed in OpenScad and printed on the Physics department 3D printer[5]. Holes were left in the sides for access to the micro SD card and the various ports. The lid was printed seperately with holes for the camera lens, the motor, and the homing sensor, and standoffs for the camera module.

[5] A few early prototypes were designed in Solid works and printed on the Science Maker Space 3D printer

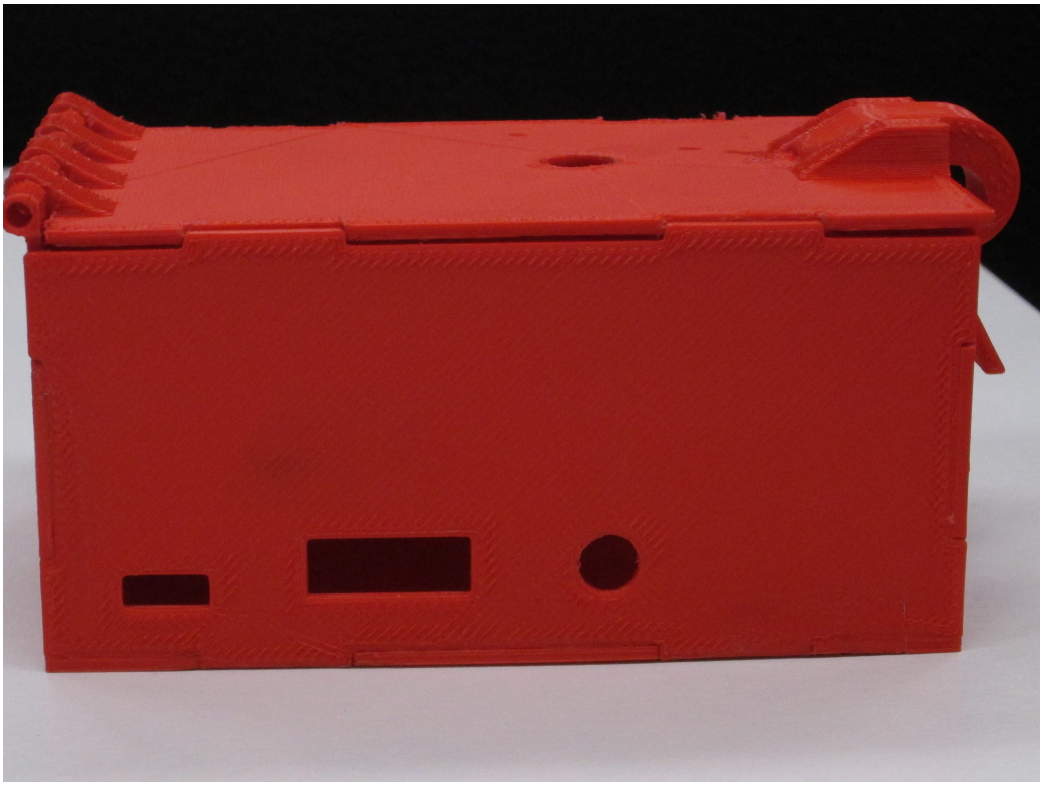


Figure 6: Early Camera Box Prototype

4.2.2 Camera Focus Mechanism & Camera Module Choice

In order to set the camera focus we need a way to rotate the lens meaning we need to connect something to the lens and we need a homing mechanism so the camera knows when its at one of the focal extremes and doesn't fall out of its socket.

We started with the Raspberry Pi camera module and were going to use

pulleys to drive the lens from the motor. The idea was to make a 1:3 ratio from the motor to the lens so when the motor did one full revolution the lens did three. By putting a notch in the motor pulley we could sense when it was at its home position.

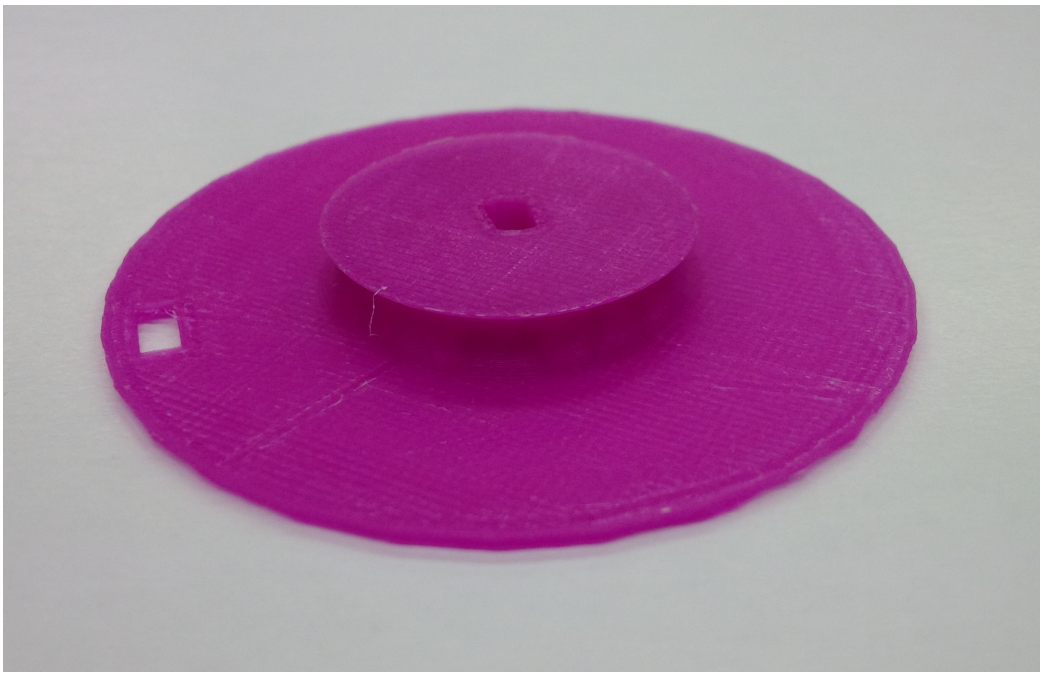


Figure 7: Test Pulley

This didn't work very well since the Pi camera module has a very small lens (see Figure 13a) and was very hard to attach anything to. We went through many prototypes (see Appendix D) before giving up and switching to the Waveshare camera module.

The Waveshare camera module has a different lens with a bigger surface area, a much bigger travel distance, and replacement lens available on Amazon[6]. In the end we used gears with a 3:1 (motor turns three times for every one turn of the lens gear) to connect the lens to the stepper motor. We made sure the lens gear big enough to make sure the homing mechanism would work.

[6]We asked Robot Shop and the Raspberry Pi Foundation but neither sold replacement lens for the Raspberry Pi camera module.

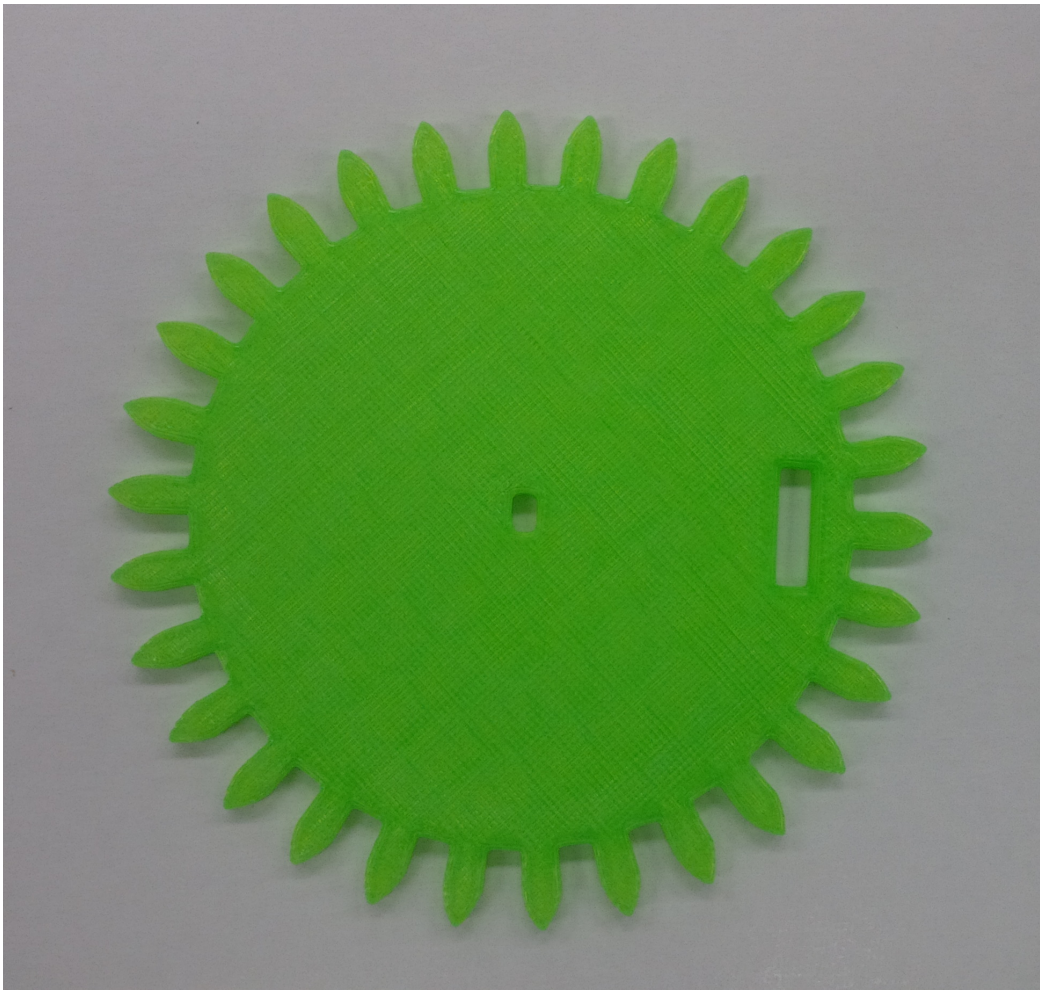


Figure 8: Test Gear

4.2.3 Gear Design

Dr. Rainer Hessmer has a web page which use OpenJsCad to generate meshing involute spur gears[13]. The key parameters which can be set are the

number of teeth (N) and the circular pitch (CP). Ultimately we want gears which mesh correctly and are a specific distance from center to center. Using some formulas[14] we get $OD = \frac{N+2}{DP}$ (OD = outer diameter, DP = diametric pitch). If x is the distance from center to center then

$$x = \frac{OD_1}{2} + \frac{OD_2}{2} \tag{1}$$

$$2x = OD_1 + OD_2 = \frac{N1+2}{DP} + \frac{N2+2}{DP}$$

$$2DPx = N1 + N2 + 4$$

Since $DP = \frac{\pi}{CP}$ we get

$$\frac{2\pi x}{CP} = N1 + N2 + 4$$

and

$$CP = \frac{2\pi x}{N1 + N2 + 4} \tag{2}$$

So for a given x, N1, and N2 we can calculate CP and use those values on the web page to generate two meshing gears which can then be downloaded, imported into OpenScad, modified as needed, and 3D printed. A key thing to remember is that equation 2 uses the outer diameter of the gears so if x is precisely the center to center the gears won't actually mesh but will just touch. To compensate make x a little bigger then the center to center distance and use some trial and error to find the best fit.

4.2.4 Homing mechanism

To sense when a gear or pulley was in a specific location we used an optical interrupt sensor. Initially we used one out of an old scanner[7] but replaced it with a board from Wavewshave when we changed camera modules. The Waveshare board comes with appropriate resistors already attached making it easier to use. We also put a ridge on the bottom of the lens gear so it would trigger the sensor when it was closest to the lid since it turned more then one revolution between focal extremes.

4.2.5 Stepper Motor

We used a 5V stepper motor from Robot Shop[8] which has 513 steps per revolution. We tried driving it using the sample code on the Adafruit website using a L293D driver chip and with a driver board which came with some motors from Hong Kong but the results were inconsistent. Sometimes it would turn normally and sometimes it would just stutter. In the end we were unable to get it to work consistently and just manually set the focus for testing.

[7] And promptly destroyed it by connecting it without resistors.

[8] Resold from Adafruit.

4.3 Focus Stacking Software

The core of the focusing stacking code is the OpenCV image processing library which is "an open source computer vision and machine learning software library"[15] with "C++, C, Python, Java and MATLAB interfaces"[15]. We followed Adrian Rosebrock's tutorial (How to install OpenCV 3 on Raspbian Jessie) and installed OpenCV and all its dependencies on the Pi making sure to make backup images of the SD card as we went.

We then tested a few algorithms for finding the focused portion of an image and decided to use the fixed window max variance of the Laplacian. This is a scaled down version of Adrian's code to see if an entire image was blurry[16]. Instead we applied his algorithm to fixed regions of the image and then save the region from the image with the highest focus. The code can be found at https://bitbucket.org/harohodg/focus-stacking-software. We tested our algorithms on the following image.

Figure 9: Test Image[17]

4.3.1 General Algorithm

1. Load all images

2. Align images[9]

3. Convolve a 3x3 Laplacian kernel with a grayscale copy of each image

4. Compare the variance of fixed windows across each image. Copy the original color data from the image with the highest variance in each

[9]Didn't implement due to time constraints.

region to the final image

5. Save final image to disk

4.3.2 GPU Algorithms

The QPUlib library creates a C++ interface for the Raspberry Pi GPU. More details on the library can be found in Appendix A. In general we create shared memory arrays of 32 bit floats or ints which are multiples of 16 elements long.

We translated the key steps to work on the GPU. For the conversion to gray scale we passed in four vectors; Red, Green, Blue, and Gray and computed

$$Gray = 0.299Red + 0.587Blue + 0.114Green$$

The key thing to remember is that OpenCV stores images as BGR in memory and to pack the Red, Green, Blue vectors accordingly.

To convolve a Laplacian Kernel with the the grayscale image we searched through the OpenCV source code and deterimined that the they are using

$$K = \begin{bmatrix} 2 & 0 & 2 \\ 0 & -8 & 0 \\ 2 & 0 & 2 \end{bmatrix}$$

when a 3x3 kernel is requested.

We pack the grayscale data into 5 vectors. TopLeft, TopRight, Centre, BottomLeft, BottomRight corresponding to the pixels used in the calculation. Then on the GPU side

$$result = 2*(TopLeft+TopRight+BottomLeft+BottomRight)-8*centre$$

One problem we encountered is when we tried to export the laplacian to look at it.

Figure 10: Proper Laplacian

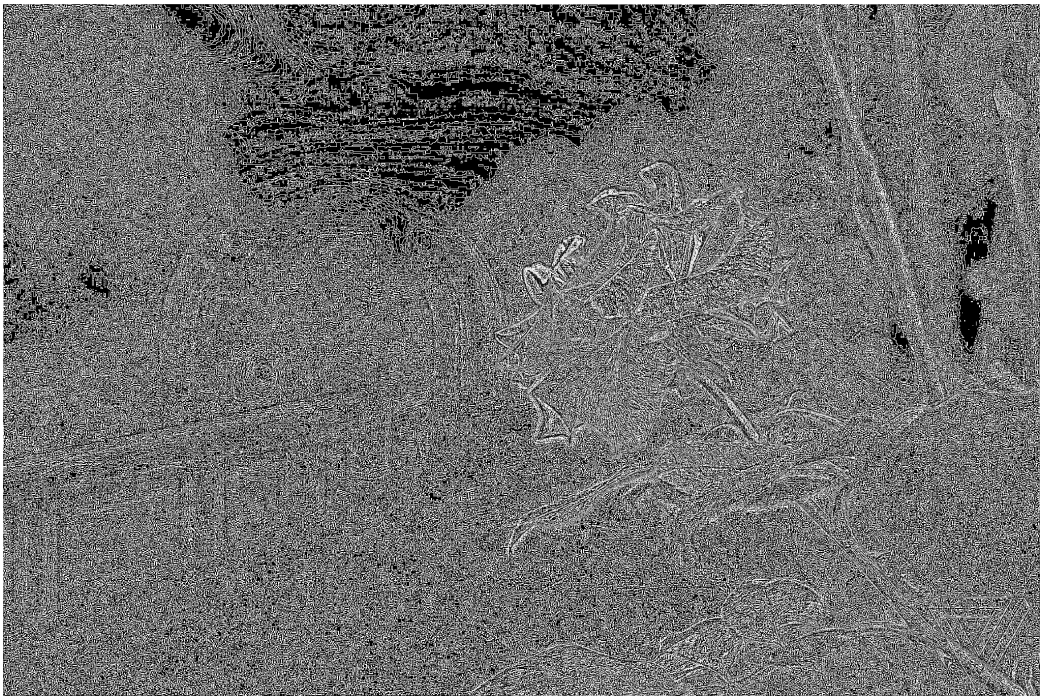


Figure 11: Improper Laplacian

The problem seems to lie in a difference between the Python and C++ calls to the same function. The Python code returns negative values which are converted to values between 0 and 255 when saving the image to disk. The C++ code returns only positive values between 0 and 255 even though it's working at a 16 bit data size.

Our code returns negavive values similar to the Python code so since we only use the laplacian in an intermediate step we can ignore this for now.

The variance is a little more complicated since we need the mean of each

window and we ideally want to compute it all at once without use a loop to setup and send the data over. So we pack each window sequentially in the input vector padding with zeros to make sure they are all the same multiple of 16 wide. Then we sum each chunk together into the same chunk of the output vector. After we return we add up the 16values in the vector and divide by n.

To then compute the variance we send the GPU X and m where m is the mean of each chunk aligned with the relavent x value. We then compute

$$temp = x - m; x = x * -1 where x < 0; result = result + temp * temp$$

A key constraint of the Raspberry Pi GPU is the amount of available memory since the GPU and the CPU share the main memory. For our tests we set a 50/50 split giving each 512MB.

Initially we tried to pack multiple images into each vector.

The test image takes up ∼10MB per vector (with naive packing) so with 5 images the grayscale conversion to took ∼200MB. The size vectors for the laplacian take up another ∼300MB. This led to the program rapidly running out of memory. The solution is to pack the vectors better and/or process each image separately and deallocate the memory once it's not needed. Packing the vectors requires unpacking on GPU side. For example if we pack the

grayscale vectors as [RED/GREEN/BLUE/GRAY] then

$$gray = 0.299(input >> 24) + 0.587(input >> 16) + 0.114(input >> 8)$$

The library supports this but we were unable to get it to work. In the end we processed each image separately using a function which takes image and returns the values of the variance of each window. We also deallocate the memory for each vector as soon as it isn't needed.

The tradeoff is that every time the function is called space is allocated on the stack. The stack on the Raspbian defaults to 8MB which isn't near enough. To change that we need to run "ulimit -s unlimited" before we run our code.

4.3.3 Camera Software

Taking the actually pictures can either be done using the raspistill terminal command,the PiCamera library, or the OpenCV library. We went with using the PiCamera and gpiozero libraries to live preview the image on a monitor and to take a picture when we pressed a button after manully adjusting the focus.

5 Results

5.1 Hardware

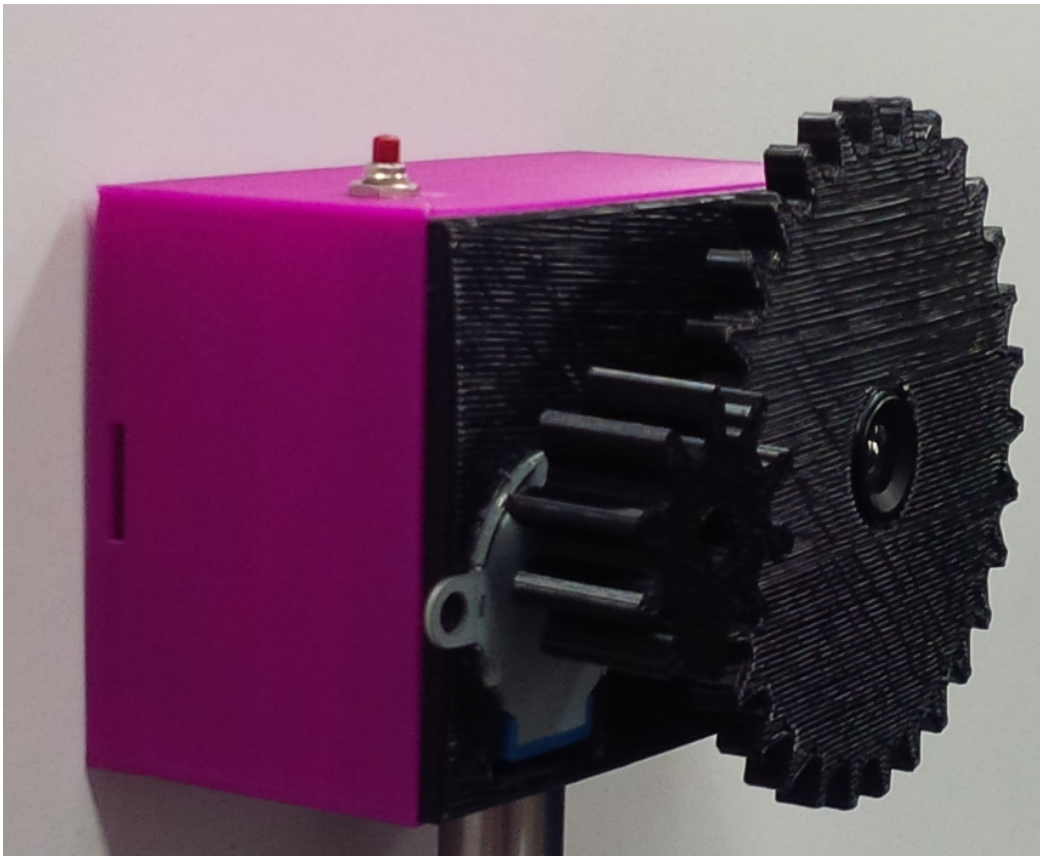
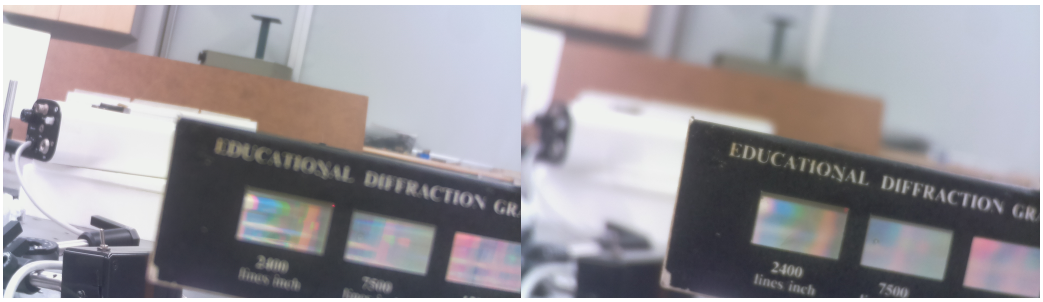


Figure 12: Final Prototype

5.2 Software stitching



(a) Sample stack image (b) Final Python merged image

Figure 13: Focus Stacking result

See Appendix C for all images in the stack and resulting images by program (Python, C++, GPU).

5.3 Software Benchmarks

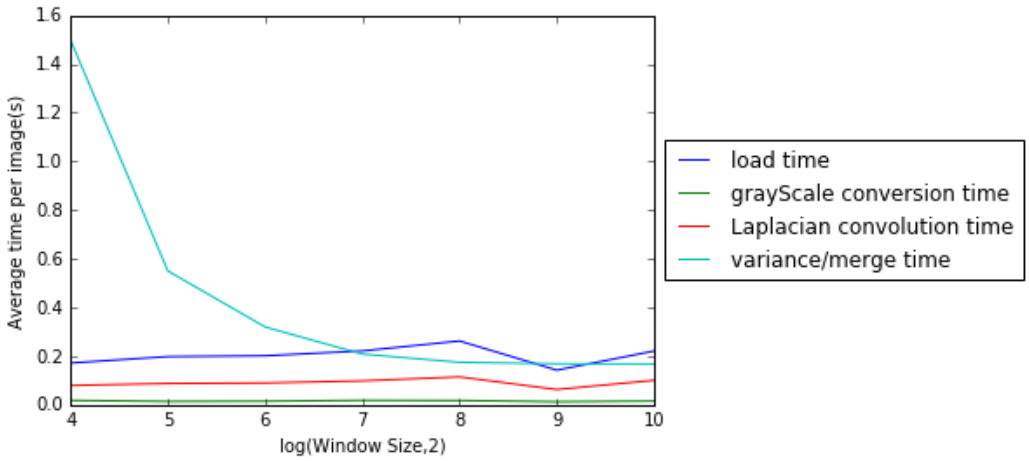


Figure 14: Python Timings

As expected the load and conversion times are pretty consistent across all window sizes. The timing improves the most for the variance computation since a larger window means a fewer number of windows in total.

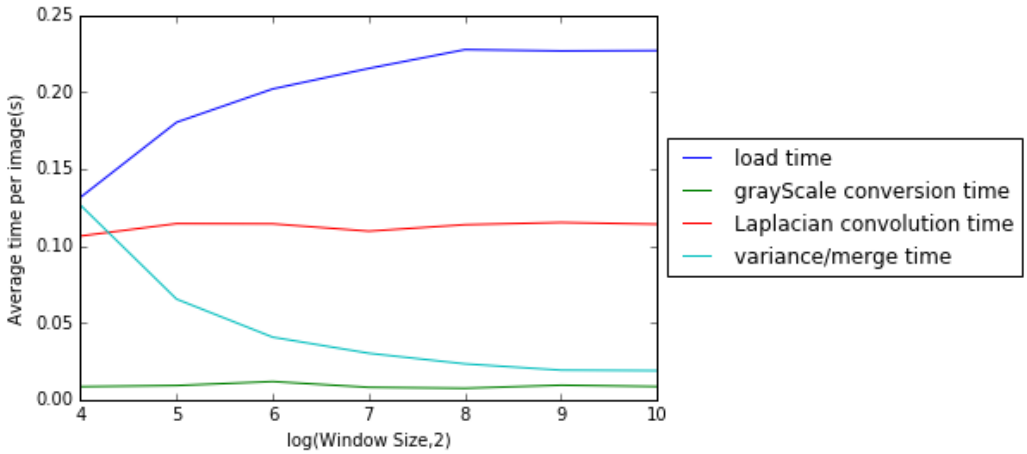


Figure 15: C++ Timings

We see similar results for the C++ code. I'm not sure why the load time goes up. I predict it's a fluke. The spread in timing between the laplacian and gray scale calculations is interesting. Especially since for Python they were similar.

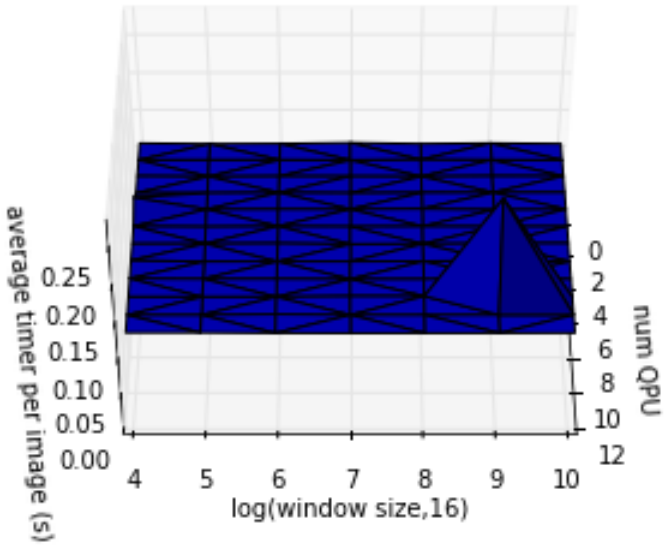


Figure 16: GPU image load time

Load time is consistent across number of GPU and window size. The one peak is almost certainly an outlier.

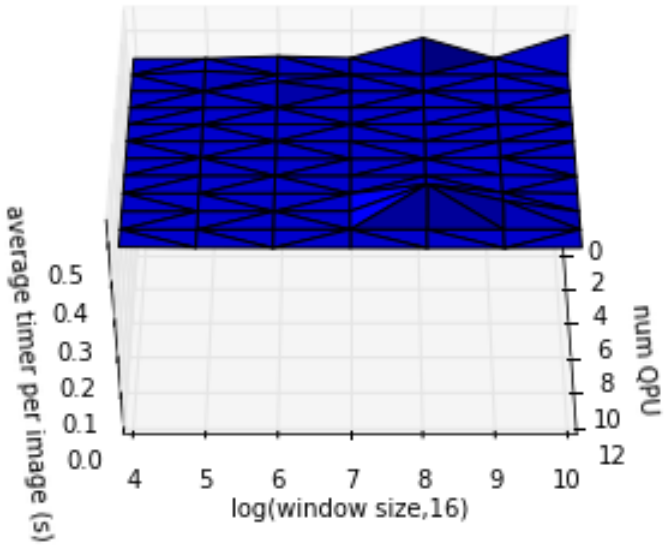


Figure 17: GPU Gray Scale memory load time

Memory load time for converting an image to Gray Scale is consistent across number of GPU and window size. The few peaks are almost certainly outliers.

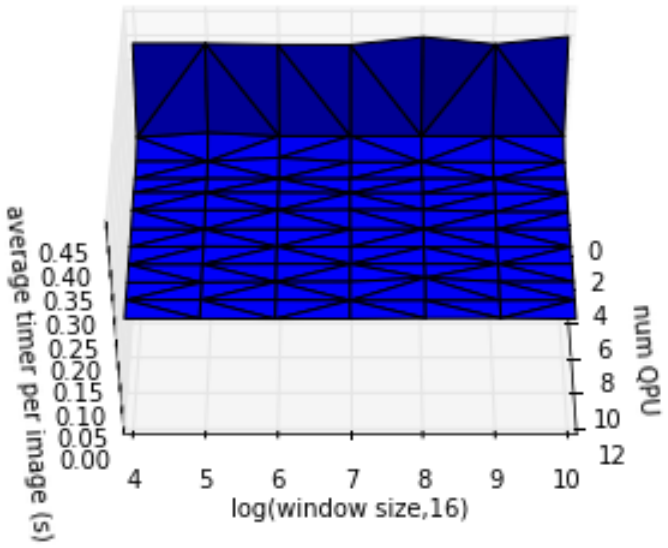


Figure 18: GPU Gray Scale compute time

We do see a slight improvement in gray scale computation time as we add more QPUs. Changing the window size doesn't make as much of a difference. Computing the grayscale is slightly faster then loading the memory in preparation.

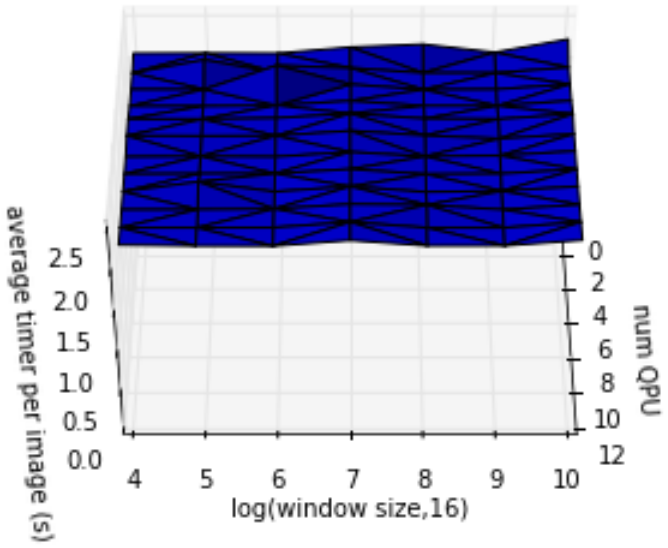


Figure 19: GPU Laplacian memory load time

We see a similar patten with loading the laplacian memory as we do when loading the gray scale memory.

Again we see an improvement when we add more QPUs. We also see a much better ratio for computation to memory load time then we do for the gray scale calculation.

Interestingly the variance memory load time is not flat like the others. I suspect it has to do with different array sizes do to different window sizes and needing to make each region a multiple of 16 elements in size.

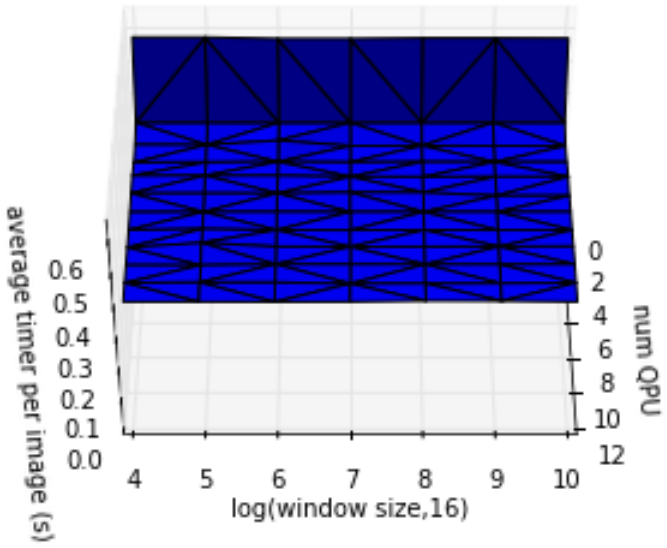


Figure 20: GPU Laplacian compute time

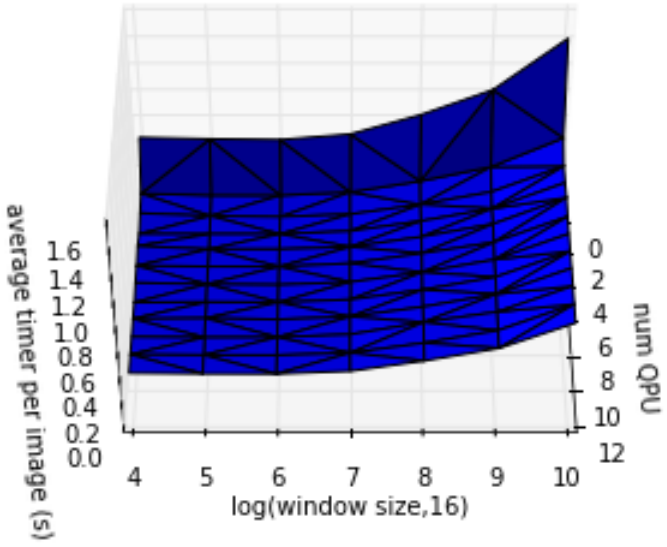


Figure 22: GPU Variance compute time

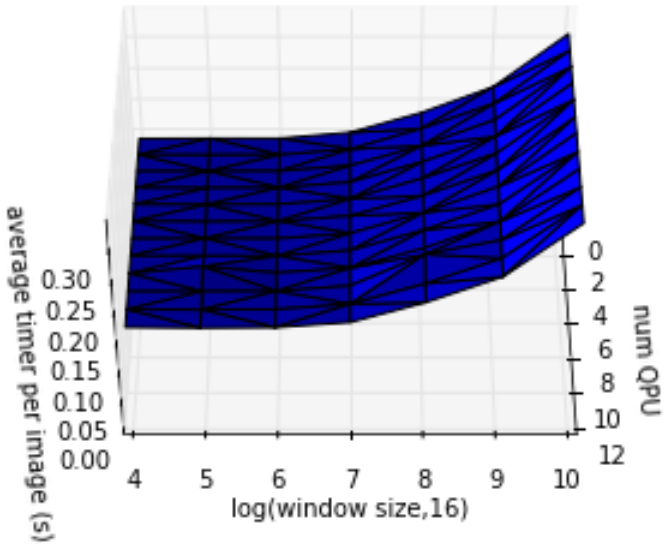


Figure 21: GPU Variance memory load time

The computation time for the variance also improves with more QPU but oddly also with a smaller window size.

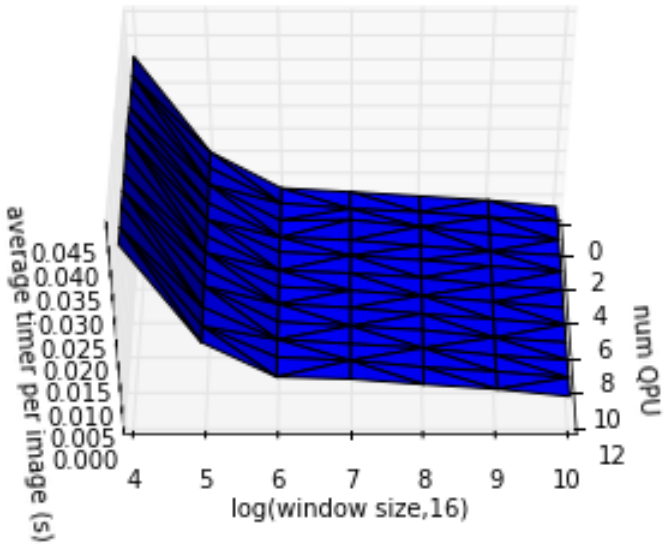


Figure 23: GPU merge time

We see a similar pattern for merge time as we did with the Python and C++ code. Specifically a larger window is good. In this case more QPUs didn't make a difference as expected since they weren't part of the merging process.

6 Future Work

6.1 Memory Packing

Packing the data in memory would make the program much more memory efficient and should also speed up memory access overall since fewer fetches would need to be done.

6.2 Loop Unrolling

In the variance code there is a loop on the GPU side to process each window. With some preprocessing that could be eliminated by replacing it with the equivelent number of QPU instructions.

6.3 Prefetching Data

Prefetching the data would help speed up the QPU computations since it would be fetching and exectuting at the same time. This is something that QPU supports.

6.4 Alternative Algorithms

A potentially simpler algorithm would be to average the pixels (with or without alignment) since regions of higher focus should somehow contribute more to the final image.

6.5 Image Alignment

Aligning the images before applying the laplacian and the variance would help ensure that regions wouldn't overlap in the final image.

6.6 Motor Control

It would be nice to get the motor working consistently. It would make getting consistent images with different focuses a lot easier.

6.7 GPU Compute Cluster

A few years ago built a Raspberry Pi 2.0 compute cluster. It would interesting to see if there are any problems that would benefit from a Pi GPU cluster. Especially if the code was designed to use the CPU at the same time.

7 Conclusion

We successfully made a focus stacking Pi camera and were able to write code for the GPU. Overall we learned that :

- The current QPU code we've written for focus stacking is slower then the CPU code but it shows promise. With further optimizations I think we could domuch better.

- For the variance and laplacian more QPU cores made a bigger difference then for the grayscale conversion. Probably because the ratio of computation to memory access was better.

- Memory access is the bottleneck overall. Better data packing and fetching while computing can make difference.

8 References

References

[1] Chen Wei Li. An introduction to focus stacking. https://digital-photography-school.com/an-introduction-to-focus-stacking/. Accessed: 2017-04-03.

[2] Cambridge in Colour. Focuss stacking & depth of field. http://www.cambridgeincolour.com/tutorials/focus-stacking.htm. Accessed: 2017-03-03.

[3] David Hunt. Macro pi – focus stacking using raspberry pi. http://www.davidhunt.ie/macro-pi-focus-stacking-using-raspberry-pi/. Accessed: 2017-03-03.

[4] Raspberry Pi Foundation. Raspberry pi 3 model b. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Accessed: 2017-04-10.

[5] Matthew Naylor. A language and compiler for the raspberry pi gpu. https://github.com/mn416/QPULib. Accessed: 2017-04-10.

[6] Raspberry Pi Fooundation. Camera module v2. https://www.raspberrypi.org/wp-content/uploads/2016/02/Pi-Camera-web.jpg. Accessed: 2017-04-11.

[7] Waveshare Electronics. Rpi camera (b), adjustable-focus. http://www.waveshare.com/product/mini-pc/raspberry-pi/cameras/rpi-camera-b.htm. Accessed: 2017-04-11.

[8] Liz Upton. Focus-stacking with raspberry pi for macro photography. https://www.raspberrypi.org/blog/focus-stacking-with-raspberry-pi-for-macro-photography/. Accessed: 2017-04-10.

[9] Particle. About particle. https://www.particle.io/about-particle. Accessed: 2017-04-11.

[10] Particle. Pricing). https://www.particle.io/pricing. Accessed: 2017-04-12.

[11] Dataplicity. Stream live video from your pi. https://docs.dataplicity.com/docs/stream-live-video-from-your-pi. Accessed: 2017-04-11.

[12] Armin Ronacher. Welcome — flask (a python microframework). http://flask.pocoo.org/. Accessed: 2017-04-11.

[13] Dr. Rainer Hessmer. Involute spur gear builder. http://hessmer.org/gears/InvoluteSpurGearBuilder.html. Accessed: 2017-04-10.

[14] LLC. Engineers Edge. Gear design equations and formula. http://www.engineersedge.com/gear_formula.htm. Accessed: 2017-04-10.

[15] OpenCV team. About opencv. http://opencv.org/about.html. Accessed: 2017-04-11.

[16] Adrian Rosebrock. Blur detection with opencv). http://www.pyimagesearch.com/2015/09/07/blur-detection-with-opencv/. Accessed: 2017-04-11.

[17] Humam Helfawi. Isolate the non blurred part of foucsed image). https://dsp.stackexchange.com/questions/22355/isolate-the-non-blurred-part-of-foucsed-image. Accessed: 2017-04-13.

[18] Adrian Rosebrock. How to install opencv 3 on raspbian jessie). http://www.pyimagesearch.com/2015/10/26/

how-to-install-opencv-3-on-raspbian-jessie/. Accessed: 2017-04-11.

[19] OpenCV. Color conversions. http://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html. Accessed: 2017-04-13.

[20] Broadcom. Videocore (r) iv 3d architecture reference guide. https://docs.broadcom.com/docs/12358545. Accessed: 2017-04-13.

[21] Pete Warden. How to optimize raspberry pi code using its gpu. https://petewarden.com/2014/08/07/how-to-optimize-raspberry-pi-code-using-its-gpu/. Accessed: 2017-04-13.

[22] Andrew Holme. Gpu_fft. http://www.aholme.co.uk/GPU_FFT/Main.htm. Accessed: 2017-04-13.

[23] Herman H Hermitage. videocoreivqpu. https://github.com/hermanhermitage/videocoreiv-qpu. Accessed: 2017-04-13.

[24] Raspberry Pi Playground. Hacking the gpu for fun and profit (pt. 1). https://rpiplayground.wordpress.com/2014/05/03/

hacking-the-gpu-for-fun-and-profit-pt-1/. Accessed: 2017-04-13.

[25] Patrick Koenig & Nate Horan. Pimd. http://pkoenig10.github.io/pimd/. Accessed: 2017-04-13.

9 Acknowledgments

A sincere thank you to the following individuals for their help with or contributions to this project.

- Dr. Fan for supervising the project and providing constructive feedback along the way
- Ron Daniels for helping me design and print the early prototypes on the Maker Space 3D printer
- The author of QPULib for fixing a bug I found in the library which was keeping me from compiling my code

Appendix A : Raspbery Pi GPUb

The core of the Pi GPU is 12 special purpose floating-point shader processors[25], called Quad Processors (QPUs). Each QPU is a 4-way SIMD (single instruction, multiple data) processor multiplexed 4× over four cycles[20]. Ie a QPU instruction takes four system clock cycles to process one 16-element result vector.

The QPUs are organized in groups called slices which share resources including an instruction cache, special function units (for calculating less frequent things like recipricals, sqrt, log, exp), and Texture and Memory lookup Units (TMU)[20].

Each QPU contains four general-purpose accumulators as well as two large register-file memories[25] and two independent ALU units. One for adding and one for multiplying[20]. There is also support for unpacking 32-bit fields and for repacking results.

QPU scheduling is done automatically in hardware by the QPU scheduler (QPS). As data is available for processing the next available QPU is scheduled to process it.

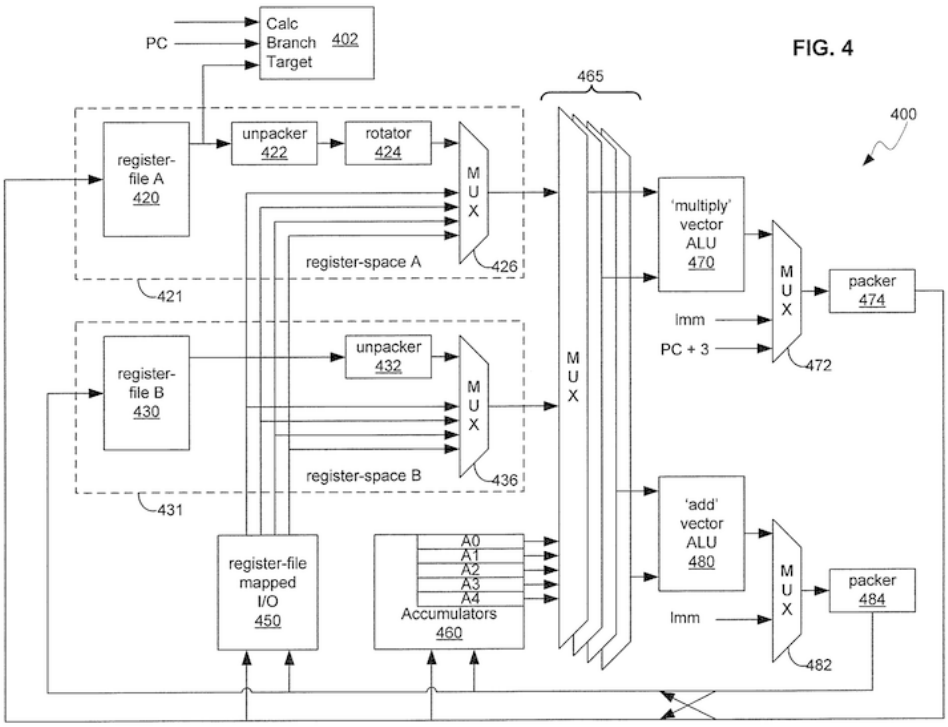


Figure 24: Instruction and Register Pipeline[23]

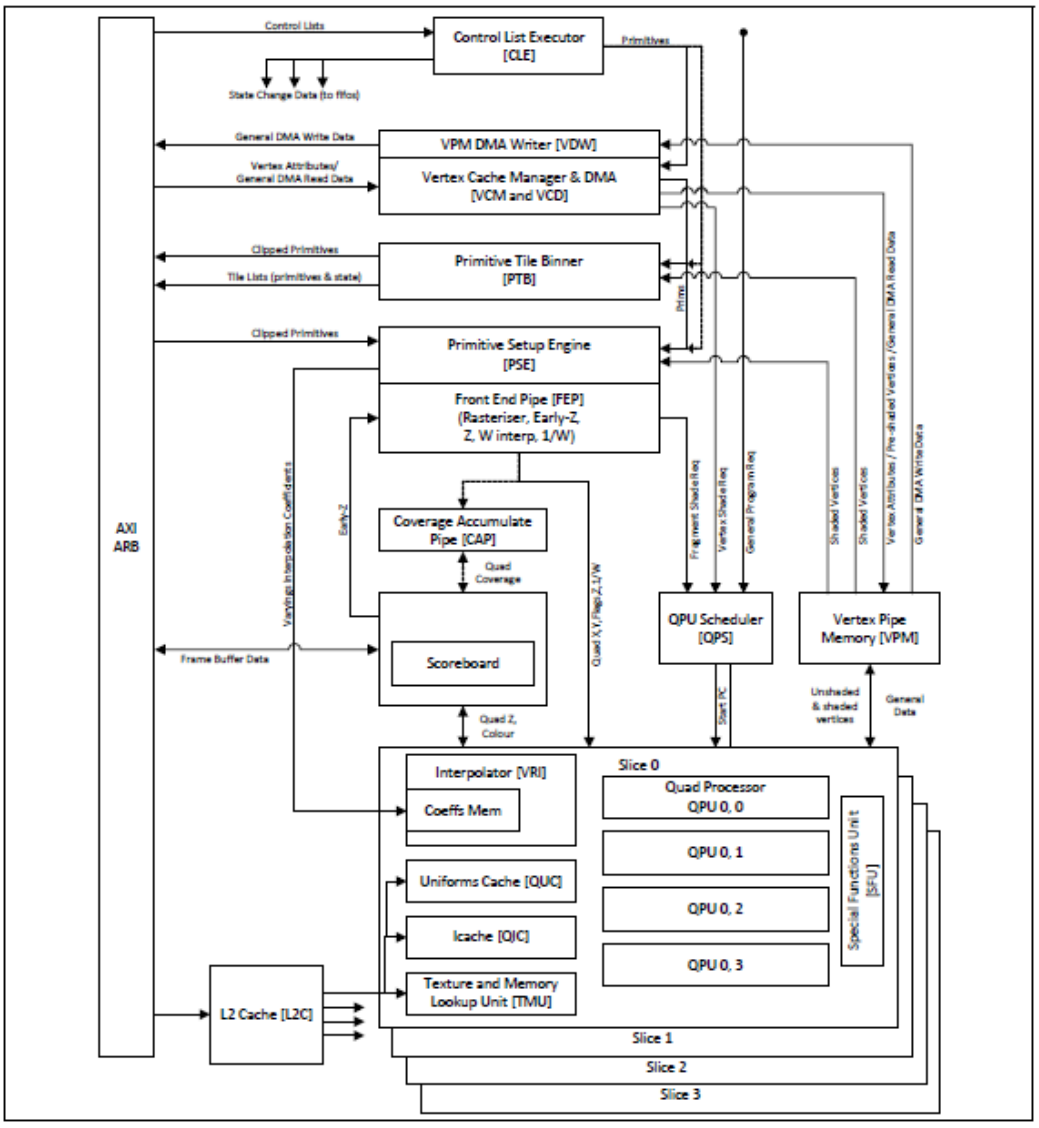


Figure 25: System Block Diagram for VideoCore IV 3D[20]

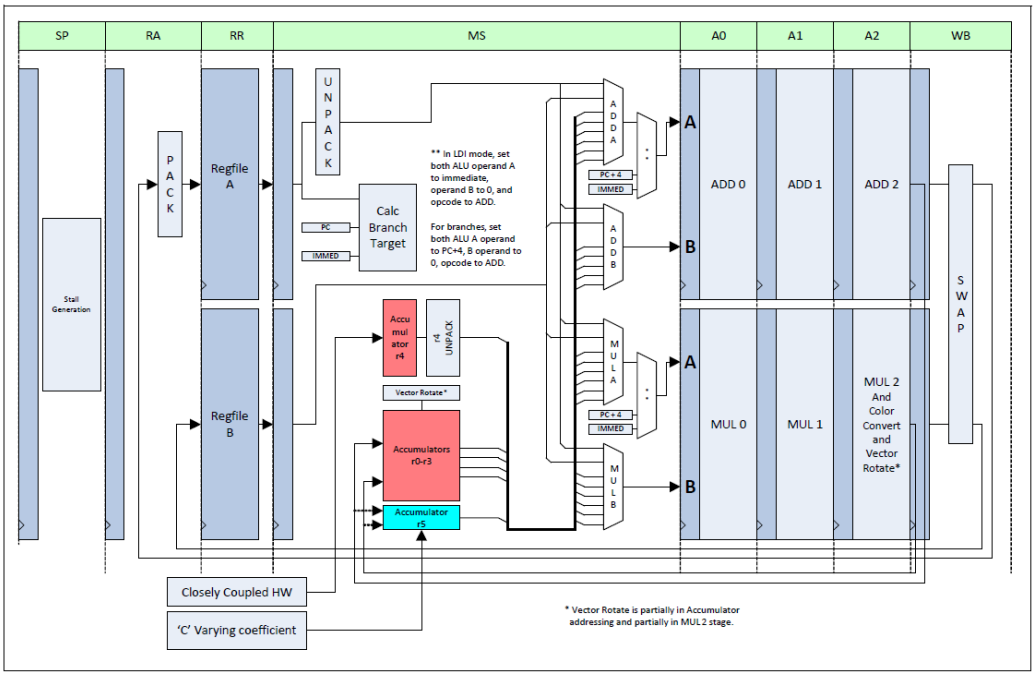


Figure 26: QPU Core Pipeline[20]

Talking to the QPUs

In order to talk to the QPUs we needs a few functions defined in mailbox.h/c written by Broadcom.

For example

```
struct memory_map {
    unsigned int code[MAX_CODE_SIZE];
    unsigned int uniforms[NUM_QPUS][2];
```

```
    unsigned int msg[NUM_QPUS][2];
    unsigned int results[NUM_QPUS][16];
};
```

defines the memory layout to share memory between the host and GPU[24].

To initialize the mailbox interface, send a message to enable the QPU and make the address space visible to the host use

```
int mb = mbox_open();
if (qpu_enable(mb, 1)) {
    fprintf(stderr, "QPU enable failed.\n");
    return -1;
}
```

To allocate and map some GPU memory

```
unsigned handle = mem_alloc(mb, size, 4096, GPU_MEM_FLG);
unsigned ptr = mem_lock(mb, handle);
void *arm_ptr = mapmem(ptr + GPU_MEM_MAP, size);
```

“ Now we have two addresses to refer to the memory. ‘ptr’ is the GPU (VC) address that the GPU understands. When passing pointers (such as the return address for the buffer to write the results into or the address of

the code and uniforms for the QPU), we need to use VC addresses. When accessing the memory from the host (for initializing the values in the first place and reading the result buffer), we have to use the mapped address which is a valid host address.

Next we do some pointer arithmetic to set the structure fields to point to the proper VC addresses. To execute a QPU program through the mailbox interface, we pass an array of message structures that contain a pointer to the uniforms to bind to the QPU program and then a pointer to the address of the QPU code to execute:

"[24]

Finally to execute the code

```
unsigned ret = execute_qpu(mb, NUM_QPUS, vc_msg, 1, 10000);
```

There a three primary ways to transfering data to the QPUs.

1. Uniforms are 32-bit values stored in memory which arrive via a queue sequentially[24]. They can be thought of as function call arguments or constants[24].

2. Textures are similar to Uniforms but don't they have a limit on how many or what order[24].

3. VPM is a 4KB memory buffer shared between all the QPUs[25][24].

Appendix B : QPUlib

The QPUlib documentation[5] states that " QPULib is a programming language and compiler for the Raspberry Pi's Quad Processing Units (QPUs). It is implemented as a C++ library that runs on the Pi's ARM CPU, generating and offloading programs to the QPUs at runtime. " It also says that " It's worth reiterating that QPULib is just standard C++ code: there are no pre-processors being used other than the standard C pre-processor. All the QPULib language constructs are simply classes, functions, and macros exported by QPULib. "

This makes it very nice library to use. It has several well documented examples which can be found at https://github.com/mn416/QPULib

In general to use the library the program will have the following features :

- `#include <QPULib.h>`
- QPU kernel function definition
- QPU kernel compilation
- Allocating shared memory

- Loading memory with data
- QPU kernel execution
- Processing returned results

When compiling use make QPU=1 to compile the program to run on baremetal and use make ... to compile it in emulation mode.

Appendix C : Sample stack input output

Figure 27: Test Image 1

Figure 28: Test Image 2

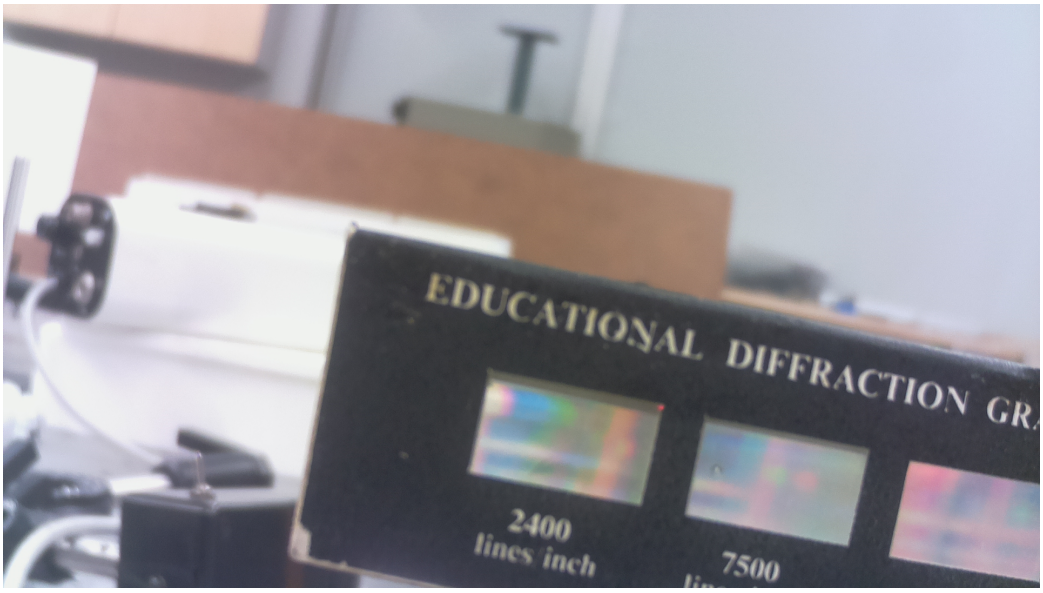


Figure 29: Test Image 3

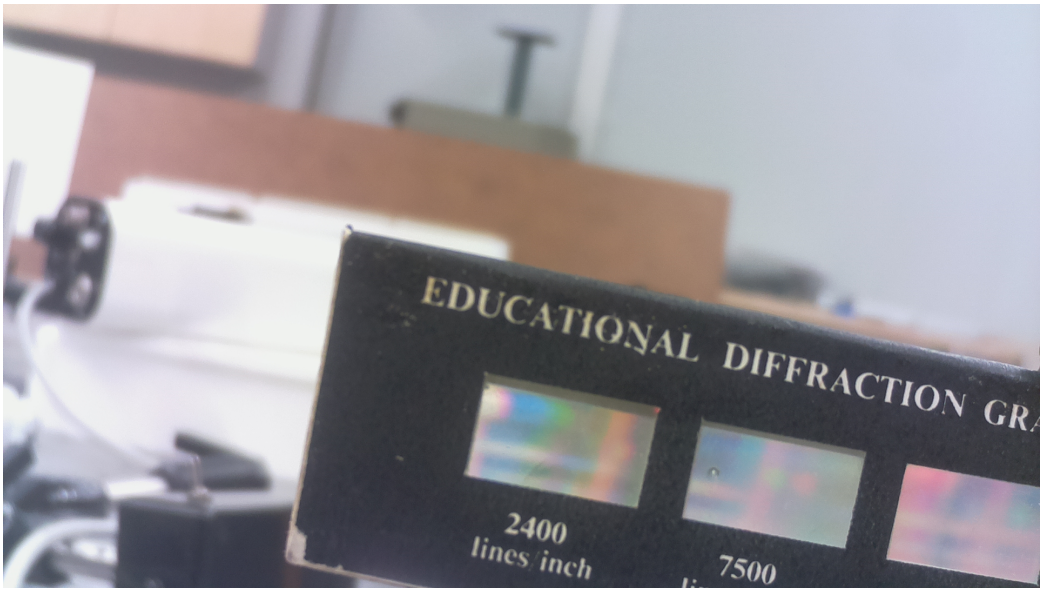


Figure 30: Test Image 4

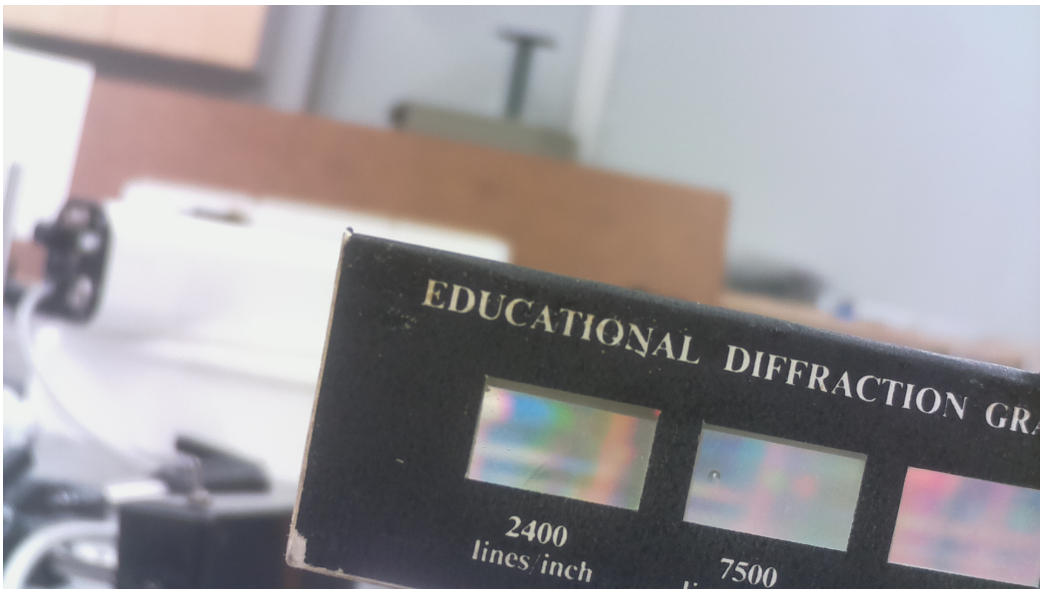


Figure 31: Test Image 5

Figure 32: Test Image 6

(a) window size 32 (b) window size 1024

Figure 33: Python focus stacked images

(a) window size 32 (b) window size 1024

Figure 34: $C++$ focus stacked images

(a) window size 32, 12 QPU (b) window size 1024, 12 QPU

Figure 35: GPU focus stacked images

Appendix D : Images of Prototypes

Figure 36: Camera box lids

Figure 37: Camera box base

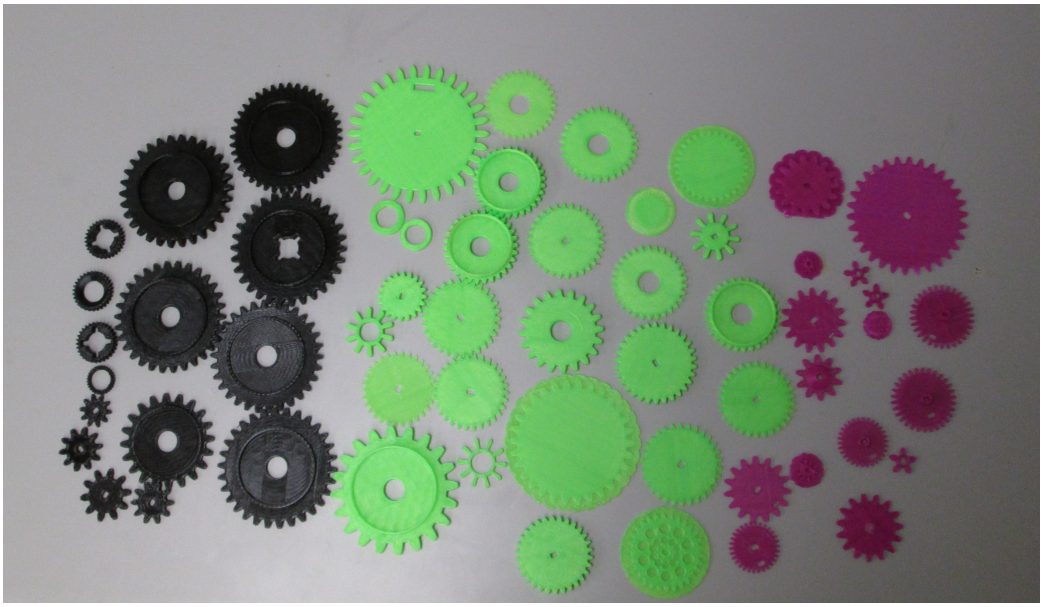


Figure 38: Gears

Figure 39: Pulleys and attachement methods