Application Interface

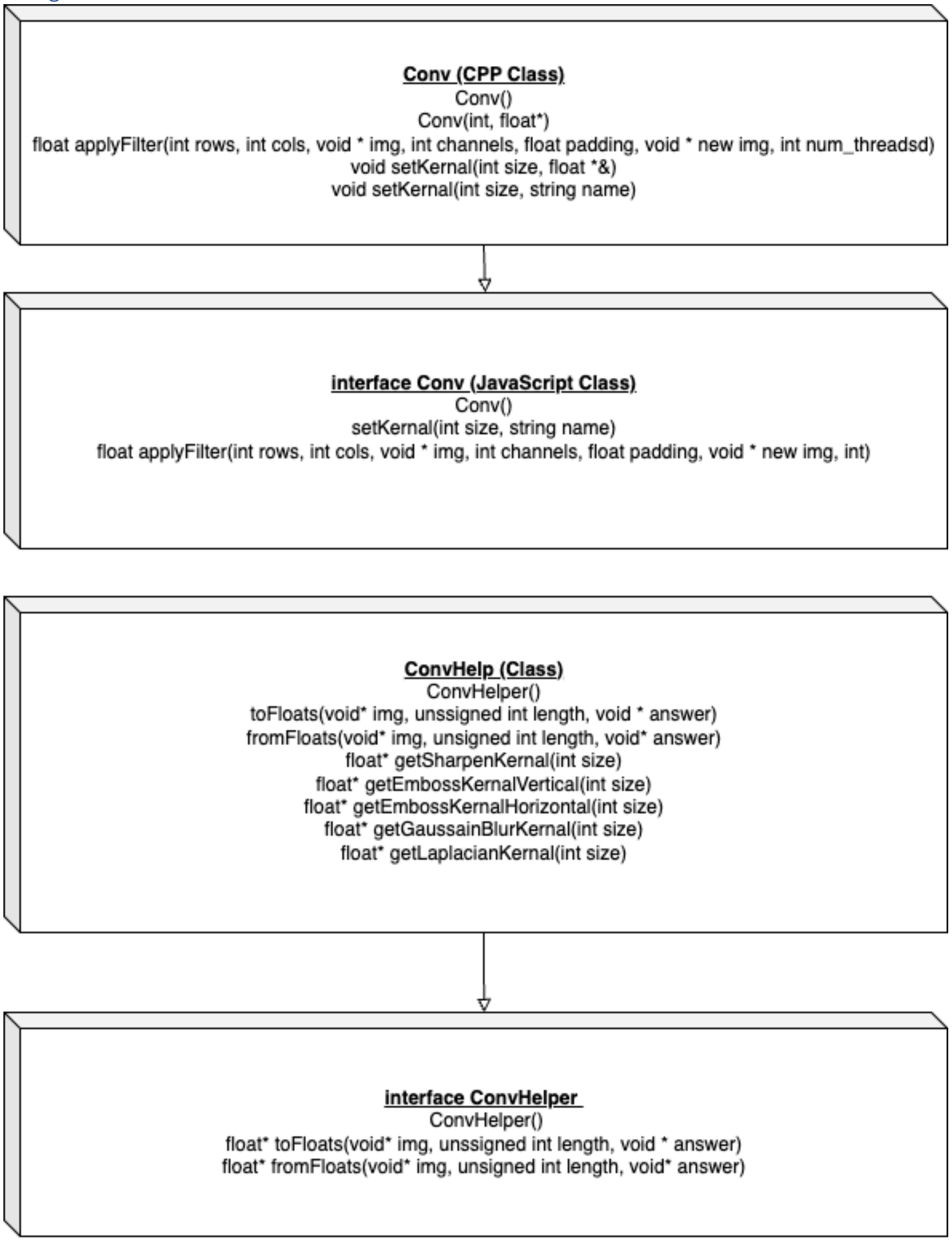
Purpose

This library is meant to allow fast image processing in the web browser. As everyone knows JavaScript code is very slow and, loops are usually the culprit. Leveraging the emscripten transpiler this library applies convolutions to images in an efficient manner, efficient enough for live video editing.

There is 2 classes output with this library. On top of all other emscripten dependencies adding output.js and output.wasm from the bin folder to any JavaScript project will grant that program access to the following classes.

This program can use a variable number of threads to complete its task. This will also be a factor to be analyzed by this report. In particular, how does threading the library affect the runtime as well as what might be the optimal number of threads to run the program.

Design


Conv (CPP Class)
Conv()
Conv(int, float*)
float applyFilter(int rows, int cols, void * img, int channels, float padding, void * new img, int num_threadsd)
void setKernal(int size, float *&)
void setKernal(int size, string name)
interface Conv (JavaScript Class)
Conv()
setKernal(int size, string name)
float applyFilter(int rows, int cols, void * img, int channels, float padding, void * new img, int)
ConvHelp (Class)
ConvHelper()
toFloats(void* img, unssigned int length, void * answer)
fromFloats(void* img, unsigned int length, void* answer)
float* getSharpenKernal(int size)
float* getEmbossKernalVertical(int size)
float* getEmbossKernalHorizontal(int size)
float* getGaussainBlurKernal(int size)
float* getLaplacianKernal(int size)
interface ConvHelper
ConvHelper()
float* toFloats(void* img, unssigned int length, void * answer)
float* fromFloats(void* img, unsigned int length, void* answer)

Method Description

Conv

- **`Conv();`**

Basic constructor to create the class. It will set the kernel to a 3x3 kernel with all values of 1/9. This is exported to the javascript class.

- **`Conv(int, float&*)`**

Constructor to set the size of the kernel and kernel on creation

- **`float applyFilter(int rows, int cols, void* img, int channels, float padding, void* new_img, int num_threads)`**

The main function of the class. Img is the float representation of the image. The data in new_img will be the data in img with the filter applied to it. It is important all void* are actually pointing to float arrays and are allocated of length row*cols*channels. This is exported to the javascript class.

A thread is created to calculate one row of the new image and then once num_threads gets created the program will wait for them all to finish. Once that is done more will spawn and continue until the new image is completed.

- **`void setKernal(int size, float *&)`**

Simple setter for the kernal

- **`void setKernal(int size, std::string name)`**

Setter that will use the private ConvHelper in the class to set the kernel. This is exported to the javascript class.

String literal to pass as a parameter	ConvHelper function that is used to set the kernal
`“sharpen”`	getSharpenKernal
“embossVertical”	getEmbossVertical
“embossHorizontal”	getEmbossHorizontal
“gassianBlurKernal”	getGaussoanBlurKernal
“laplacianKernal”	getLaplacianKernal
“edgeHorizontal”	getHorizontalEdgeKernal
“edgeVertical”	getVerticalEdgeKernal
“sharpen2type”	getSharpen2type

This is the suggested way to alter the kernel while executing in javascript.

- **`float * getKernal()`**

Getter to get the current kernal.

- **`int getPosition(int x, int y, int c, int num_channels, int img_width, int img_length)`**

Since the image is stored as a 1-demensional array for speed of processing, this function takes the image properties and the target x,y,c of a pixel and returns the place in the 1 dimensional array where that pixel is. This is not typically used outside of the class, but is accessible.

- **`void printKernal();`**

Print the kernal that is being used for filtering. This is mainly helpful for debugging purposes.

- **`float getPixelValue(int rows, int cols, float* img, int channels, float padding, int r, int c channel)`**

Similar to `getPosition`, this function is used to get the new value of a pixel with the img data and the position data. This function is not typically used outside of the application

ConvHelper

- **`float toFloats(void* img, unsigned int length, void* answer)`**

Helper function to turn chars (0-255 unsigned ints) into floats. Will return the time it took to complete the transfer

- **`float fromFloats(void* img, unsigned int length, void* answer)`**

Helper function to turn floats into chars (0-255 unsigned ints). Will return the time it took to complete the transfer

These are the filters that are built into the library. When using the setKernal method in the Conv class these will be used. They are stored and maintained in each instance of the ConvHelper class.

- **`float* getSharpenKernal(int size)`**

-1.0	-1.0	-1.0
-1.0	9.0	-1.0
-1.0	-1.0	-1.0

-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	25.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0

- **`float* getEmbossKernalVertical(int size)`**

0.0	1.0	0.0
0.0	0.0	0.0
0.0	-1.0	0.0

0.0	0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
0.0	0.0	-1.0	0.0	0.0
0.0	0.0	-1.0	0.0	0.0

- **`float* getEmbossKernalHorizontal(int size)`**

0.0	0.0	0.0
1.0	0.0	-1.0
0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
1.0	1.0	0.0	-1.0	-1.0
0.0	0.0	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0

- **float* getGaussianBlurKernal(int size)**

1.0/16.0	2.0/16.0	1.0/16.0
2.0/16.0	4.0/16.0	2.0/16.0
1.0/16.0	2.0/16.0	1.0/16.0

1.0/256.0	4.0/256.0	6.0/256.0	4.0/256.0	1.0/256.0
4.0/256.0	16.0/256.0	24.0/256.0	16.0/256.0	4.0/256.0
6.0/256.0	24.0/256.0	36.0/256.0	24.0/256.0	6.0/256.0
4.0/256.0	16.0/256.0	24.0/256.0	16.0/256.0	4.0/256.0
1.0/256.0	4.0/256.0	6.0/256.0	4.0/256.0	1.0/256.0

- **float* getLaplacianKeranl(int size)**

0.0	1.0	0.0
1.0	-3.0	1.0
0.0	1.0	0.0

0.0	0.0	1.0	0.0	0.0
0.0	1.0	2.0	1.0	0.0
1.0	2.0	-15.0	2.0	1.0
0.0	1.0	2.0	1.0	0.0
0.0	0.0	1.0	0.0	0.0

- **float* getHorizontalEdge(int size)**

1.0	2.0	1.0
0.0	0.0	0.0
-1.0	-2.0	-1.0

1.0	4.0	6.0	4.0	6.0
2.0	8.0	12.0	8.0	4.0
0.0	0.0	0.0	0.0	0.0
-2.0	-8.0	-12.0	-8.0	-4.0
-1.0	-4.0	-6.0	-4.0	-1.0

- **float* getVerticalEdge(int size)**

1.0	0.0	-1.0
2.0	0.0	-2.0
1.0	0.0	-2.0

1.0	2.0	0.0	-2.0	-1.0
4.0	8.0	0.0	-8.0	-4.0
6.0	12.0	0.0	-12.0	-6.0
4.0	8.0	0.0	-8.0	-6.0
1.0	2.0	0.0	-2.0	-1.0

- **`float* getSharpen2Type(int size)`**

0.0	-1.0	0.0
-1.0	5.0	-1.0
0.0	-1.0	0.0

0.0	0.0	-1.0	0.0	0.0
0.0	0.0	-4.0	0.0	0.0
-1.0	-4.0	21.0	-4.0	-1.0
0.0	0.0	-4.0	0.0	0.0
0.0	0.0	-1.0	0.0	0.0

Code Example

C++

Right now the way this library is used in C++ is to add the source code to your code base. You will have to add conv.cpp, conv.h, convHelper.cpp convHelper.h. Then include the *.h files as normal where appropriate. Main.cpp is an example of using opencv to read in images, but whatever method of reading in images will work as long as it can be translated to an float or char array.

1. Include classes

```
#include "convHelper.h"
#include "conv.h"
```

2. Set up classes

```
    ConvHelper helper = ConvHelper();
    Conv conv = Conv();
```

3. Set Kernal and load the data. The variable data is needs to be set to the char array containing the pixels of the image. If floats are already present that part can be skipped.

```
    conv.setKernal(3, “sharpen”);
    unsigned char * arr = img.isContinuous()? img.data: img.clone().data;
    unsigned int length = img.total()*img.channels();
```

4. Allocated needed arrays

```
    float * img_float = new float[length];
    float * conv_img_float = new float[length];
    unsigned char* new_img_char = new unsigned char[length];
```

5. Convert to floats if needed

```
    float time = helper.toFloats(img, length, img_float);
```

6. Apply filter and convert back from floats.

```
    time += conv.applyFilter(rows, cols, img_float, channels, 1.0, conv_img_float, 8);
    time += helper.fromFloats(conv_img_float, length, new_img_char);
```

At this point the altered image will be in the char representation at the address of new_img_char. If you save method needs floats then you can omit the last line and use the variable conv_img_float.

If main.cpp from the repo is going to be used to test opencv will have to be set up (There is already a CMakeList.txt for this) and it should be executed in the Code folder.

1. In the code folder execute `cmake .`
2. In the code folder execute `make`
3. This will create an a.out executable in the Code folder. Execute ./a.out ../Data/<filename>
    a. For this to run there will need to be a folder set up in Data similar to the images. For example if filename = dog.png then the folder /Data/dog must exist.

Javascript

To use the javascript version of this library simply include the output.wasm and output.js files in your code like you would any other library.

1. Set up the classes

```
    var conv = new Module.Conv();
    var convHelper = new Module.ConvHelper();
```

2. Allocate and set the image data. Keep in mind that the image variable is a javascript array

```
    image = removeAlpha(image)
    var imageCharHeap = Module._malloc(image.length * image.BYTES_PER_ELEMENT);
```

3. Allocate float array and set it. Only nessary if the data was read in as chars

```
    var imageFloatHeap = Module._malloc(image.length *Float32Array.BYTES_PER_ELEMENT);
    var timeFirstConverion = convHelper.toFloats(
imageCharHeap,image.length,imageFloatHeap);
```

4. Allocate the needed arrays

```
    var appliedImageFloat = Module._malloc(image.length *
Float32Array.BYTES_PER_ELEMENT);
    var newImageHeap = Module._malloc(image.length * image.BYTES_PER_ELEMENT);
```

5. Set Kernal and apply the filers

```
    conv.setKernal(size, filter);
    time += conv.applyFilter(rows, cols, imageFloatHeap, channels, 1.0,
newImageFlaotHeap, 8)
```

app.js is an example of this, Jimp is required for app.js to run, “npm install jimp” can satisfy that requirement. While in the Code folder simply execute “node bin/app.js ../Data/dog.png”. On top of Jimp this program requires node version 19.0 for the threads to compile correctly.

In addition to app.js there is a website altering the image live from a webcam. To run this at the top of the repo `run python3 Code/website/simple_cors_server.py` In addition then navigate to http://localhost:8003/Code/website/imdex.html

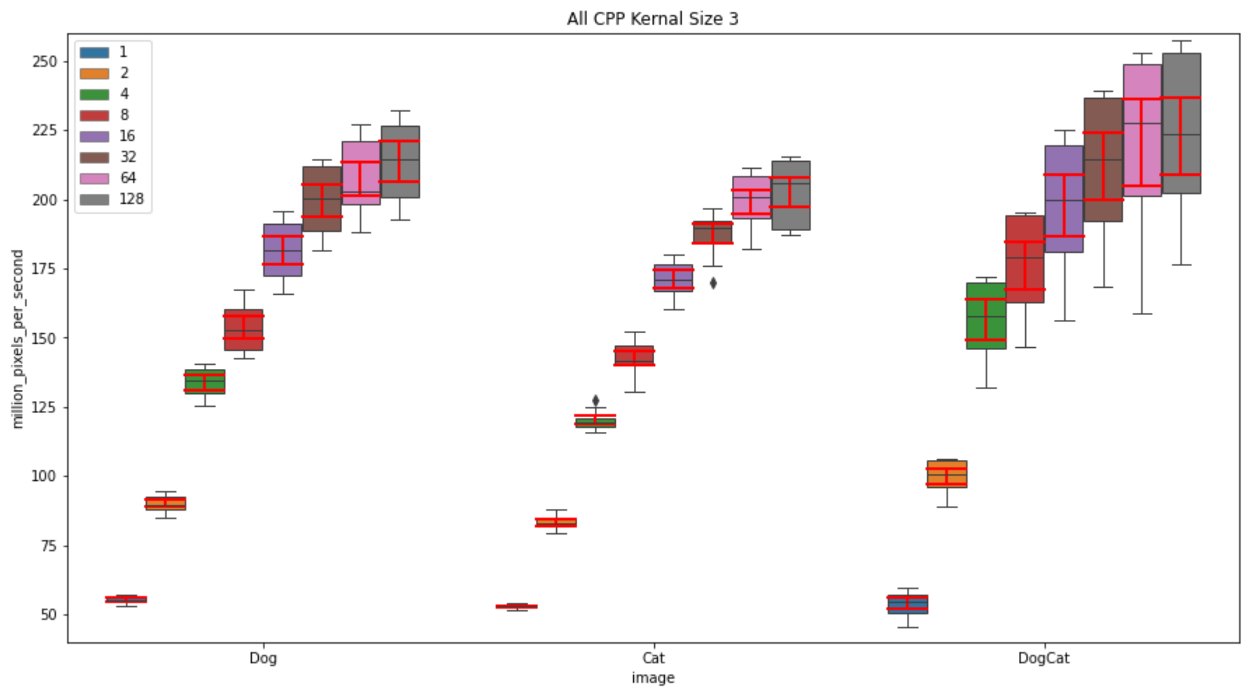
Comparison of Runtimes

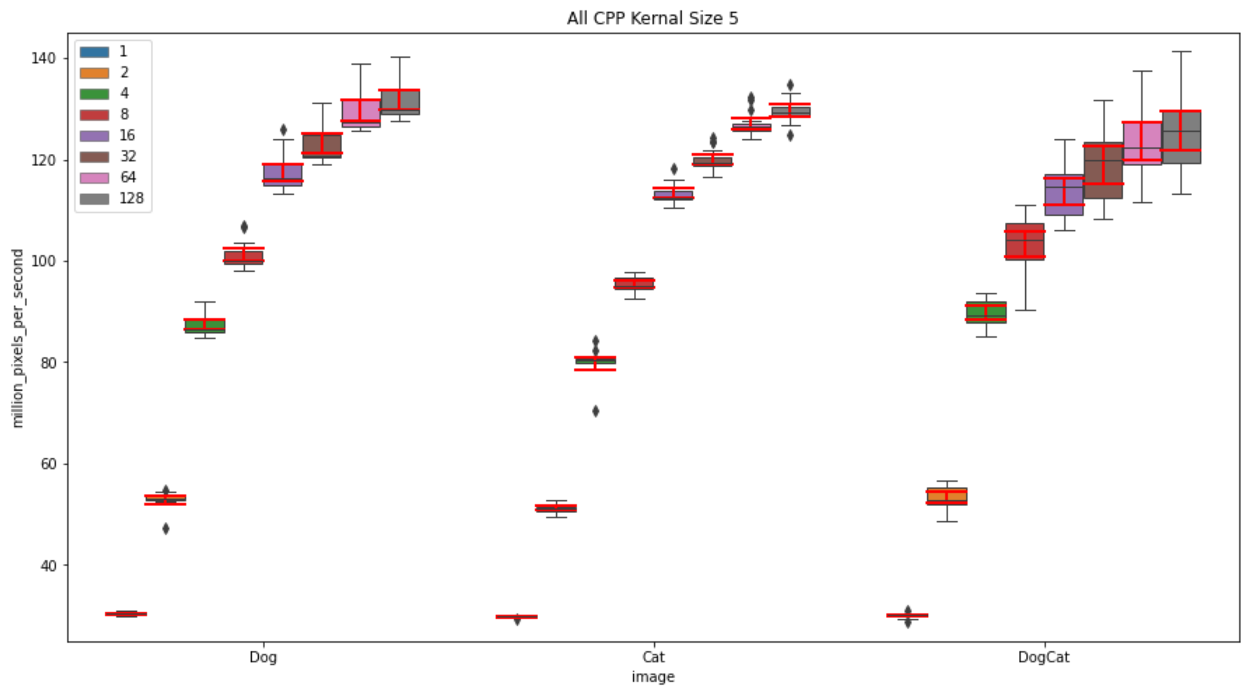
Experiment 1 What is the correct number of threads

For the following charts a decorated boxplot is used. The shaded region with a line in represent the median and the inner two quartiles. The whiskers attached to the plot represent the outer two quartiles and the ride lines represent a confidence interval of 95%. Any outliers outside of the 2 outer quartiles will be marked as points.

Native Code

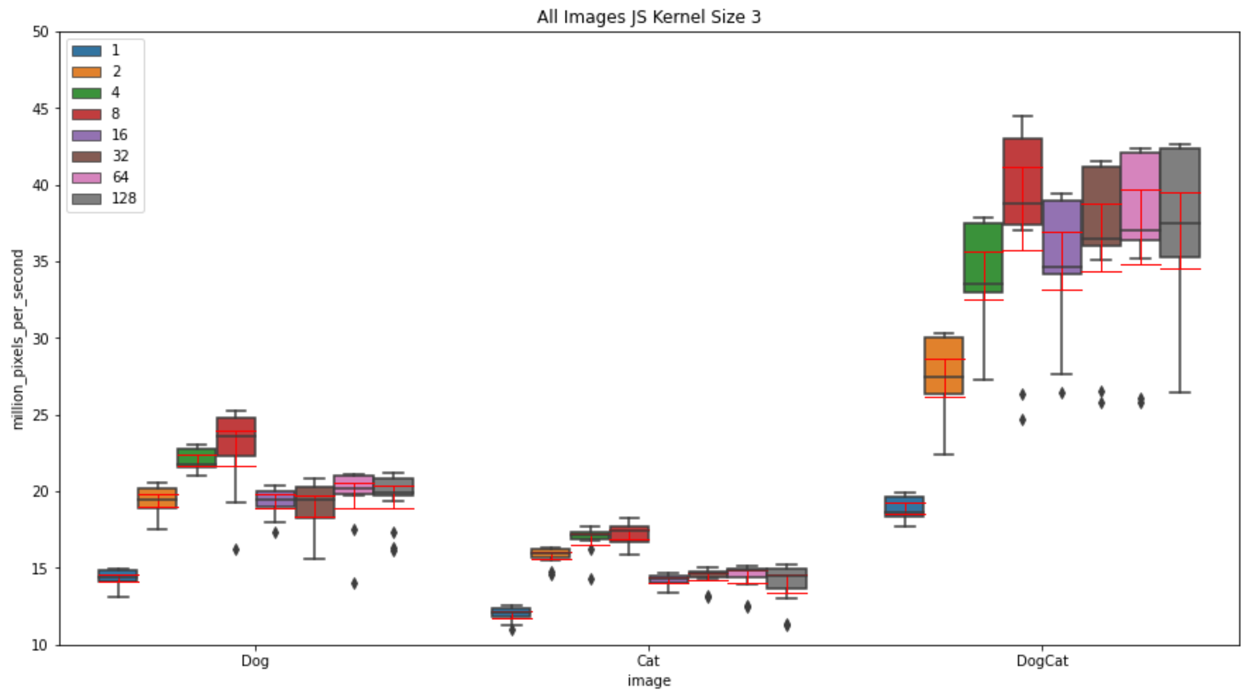
The first step is to find the correct number of threads for execution.

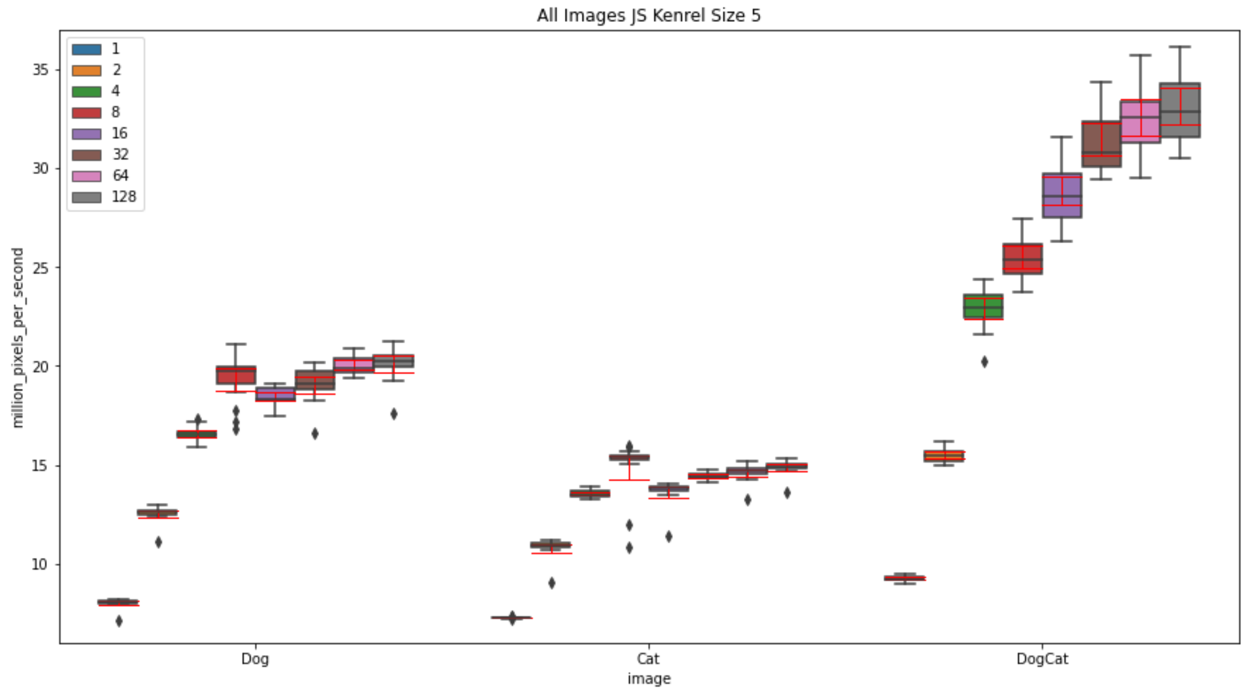




With the native code increasing the threads seems to increase the performance all the way to 128. This would likely not be the case on a system that had full access to the processor. The machine this program ran on had 8 cores, so the increase in performance likely came from the operating system choosing the program's threads with a higher likelihood since there is more of them. There is a clear benefit in threading the program. For later sections 128 threads will be used to compare to the single threaded version.

JavaScript Code



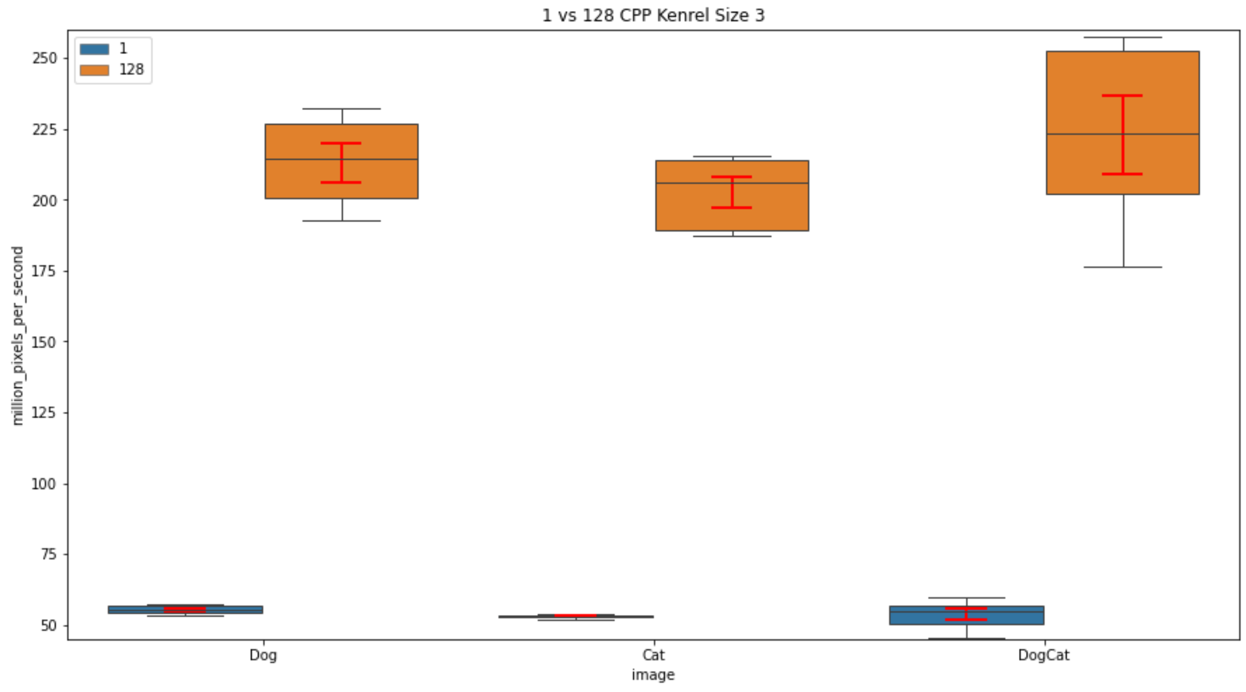


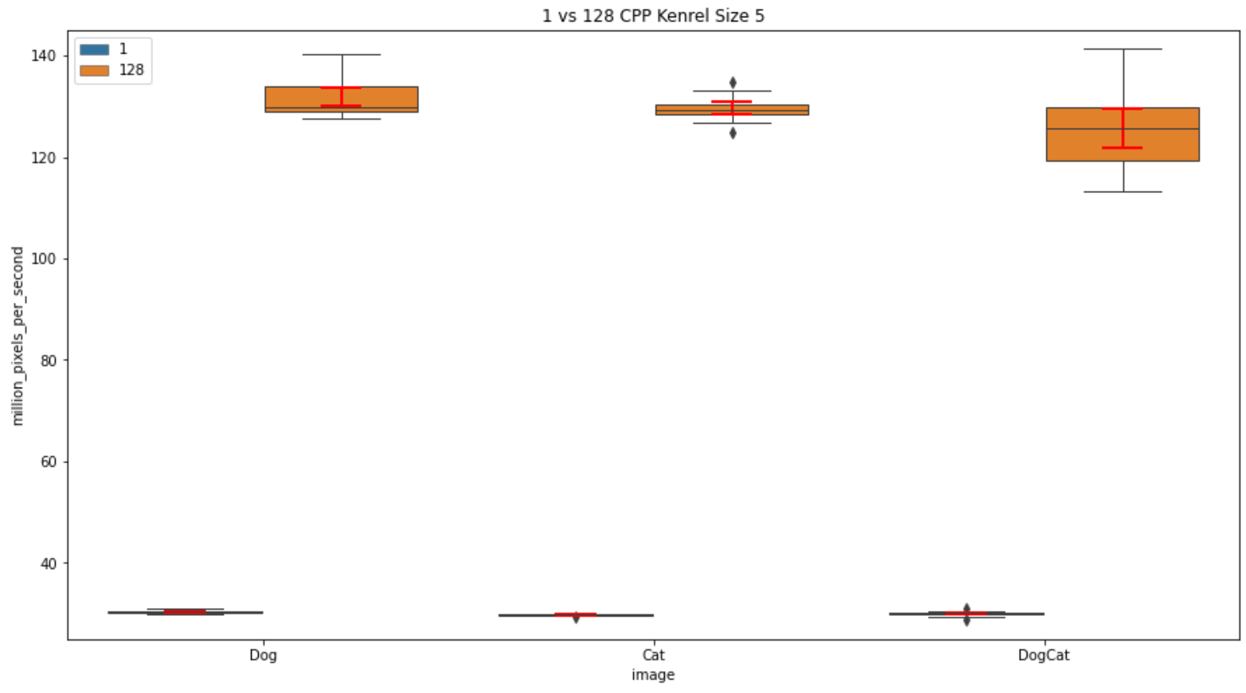
The story around JS improvement is a little different. For the smaller two images 8 threads seems to be the best option. For the largest image There is an increase in performance for kernel size 5, and there is an unclear benefit for kernel size 3. That being considering the image that usually gets streamed. There is a lot of reasons that could result in the drop in performance past 8 threads, but it is expected that native C++ code would handle threads more efficiently than its transpiled counterpart. For analysis purposes 8 threads will be compared to the single threaded version in the next sections.

Experiment 2 Best of threads versus 1 Thread

Native Code

For native code 1 thread is going to be compared to its 128 thread counterpart.

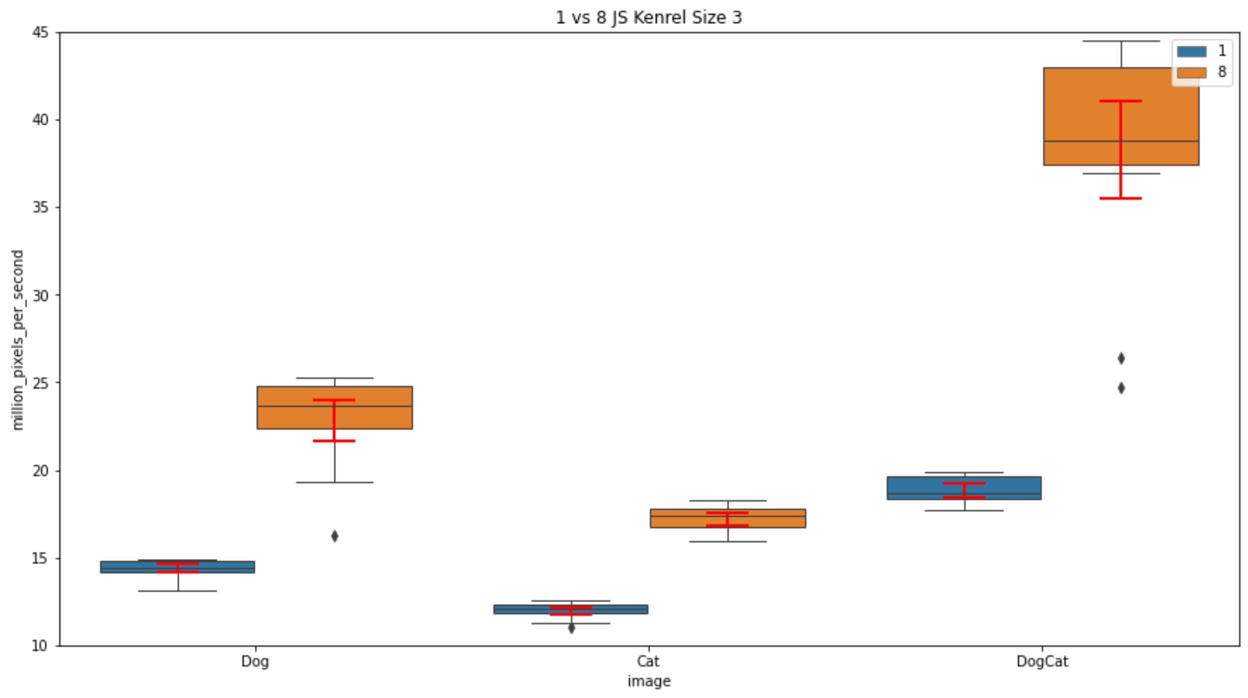


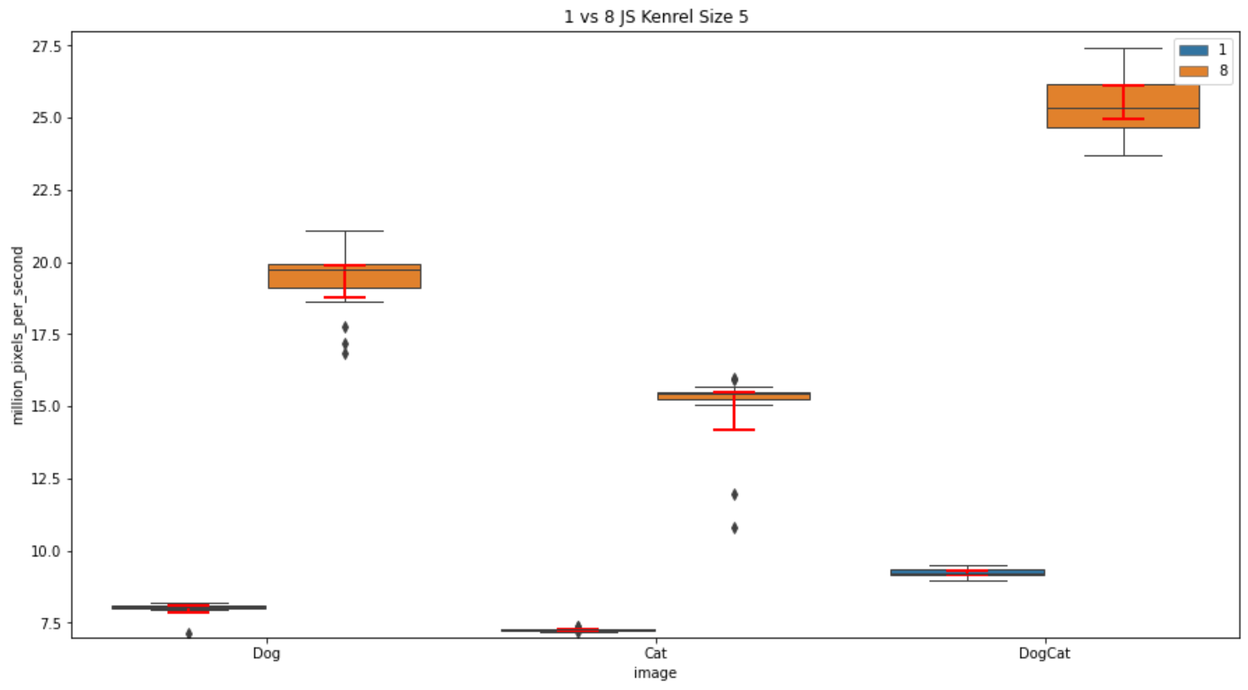


These charts speak for themselves when inspecting the confidence intervals. While 128 threads might be an overkill number of threads, but the improvement is clear.

JavaScript Code

When comparing the single threaded version to 8 threads is significantly fast as expected.





In a very similar fashion, the confidence intervals in the above charts clearly show the improvements available for threading the application. Given this improvement 8 threads give a clear benefit and will be used in the next analysis.

Frame Rate Analysis in JavaScript

Before starting the frame rate analysis, it is appropriate to apply a T-test to the means in question to ensure there is a statistical difference between them.

Image	Kernel Size	Execution	Threads	Mean Million pixels /sec	Confidence
Dog	3	CPP	1	55.41	
Dog	3	CPP	128	213.5	1.913×10^{-28}
Cat	3	CPP	1	52.96	
Cat	3	CPP	128	202.8	4.40×10^{-31}
DogCat	3	CPP	1	54.02	
DogCat	3	CPP	128	223.0	8.102×10^{-20}
Dog	5	CPP	1	30.42	
Dog	5	CPP	128	131.8	8.916×10^{-39}
Cat	5	CPP	1	29.80	
Cat	5	CPP	128	129.6	2.508×10^{-45}
DogCat	5	CPP	1	30.02	
DogCat	5	CPP	128	125.5	2.337×10^{-29}
Dog	3	JS	1	14.39	
Dog	3	JS	8	22.92	2.021×10^{-14}
Cat	3	JS	1	12.00	
Cat	3	JS	8	17.18	1.092×10^{-21}
DogCat	3	JS	1	18.87	
DogCat	3	JS	8	38.56	2.511×10^{-14}
Dog	5	JS	1	8.023	
Dog	5	JS	8	19.37	1.721×10^{-26}
Cat	5	JS	1	7.269	
Cat	5	JS	8	14.96	8.490×10^{-20}
DogCat	5	JS	1	9.246	
DogCat	5	JS	8	25.50	3.958×10^{-31}

After applying a ttest to all of the pairing of single threaded vs. multithreaded there is a clear statistical. The following section will contain theoretical frame rates based on these speeds.

Dog image in CPP 1920x1080	
Kernel Size 3 – 1 thread $2.07\ mill\ pixels * \left(\frac{1\ sec}{54.41\ mill\ pixels}\right)$ $= 0.0380 \sec or\ 26.29\ Hz$	**Kernel Size 3 – 128 thread** $2.07\ mill\ pixels * \left(\frac{1\ sec}{213.5\ mill\ pixels}\right)$ $= 0.009696 \sec or\ 103.1\ Hz$
Kernel Size 5 – 1 thread $2.07\ mill\ pixels * \left(\frac{1\ sec}{30.42\ mill\ pixels}\right)$ $= 0.06804 \sec or\ 14.70\ Hz$	**Kernel Size 5 – 128 thread** $2.07\ mill\ pixels * \left(\frac{1\ sec}{131.8\ mill\ pixels}\right)$ $= 0.0380 \sec or\ 63.67\ Hz$

Cat image in CPP 1344x837	
Kernel Size 3 – 1 thread $1.124\ \ mill\ pixels * \left(\frac{1\ sec}{52.96\ mill\ pixels}\right)$ $= 0.02122 \sec or\ 47.12\ Hz$	**Kernel Size 3 – 128 threads** $1.124\ \ mill\ pixels * \left(\frac{1\ sec}{202.8\ mill\ pixels}\right)$ $= 0.005542 \sec or\ 180.4 Hz$
Kernel Size 5 – 1 thread $1.124\ \ mill\ pixels * \left(\frac{1\ sec}{29.80\ mill\ pixels}\right)$ $= 0.03772 \sec or\ 26.51\ Hz$	**Kernel Size 5 – 128 thread** $1.124\ mill\ pixels * \left(\frac{1\ sec}{129.6 mill\ pixels}\right)$ $= 0.008673 \sec or\ 115.3\ Hz$

DogCat image in CPP 4040x2693	
Kernel Size 3 – 1 thread $10.8\ mill\ pixels * \left(\frac{1\ sec}{54.02\ mill\ pixels}\right)$ $= 0.1999 \sec or\ 5.002\ Hz$	**Kernel Size 3 – 128 threads** $10.8\ \ mill\ pixels * \left(\frac{1\ sec}{223.0\ mill\ pixels}\right)$ $= 0.04843 \sec or\ \ 20.65\ Hz$
Kernel Size 5 – 1 thread $10.8\ \ mill\ pixels * \left(\frac{1\ sec}{30.02\ mill\ pixels}\right)$ $= 0.3598 \sec or\ \ 2.780\ Hz$	**Kernel Size 5 – 128 thread** $10.8\ mill\ pixels * \left(\frac{1\ sec}{125.5\ mill\ pixels}\right)$ $= 0.0861 \sec or\ \ 11.62\ Hz$

With all of these calculations we can see a clear frame rate increase. If the goal was to have an application stream the filters in real time for the 1080p image (dog.png) thee frame rate is well above what would be needed. The large image still doesn't have a good frame rate, but 11.62 Hz is a whole lot more reasonable for a user than 2.780. The next section will compare JavaScript application; 1 vs. 8 threads.

Dog image in JS 1920x1080	
Kernel Size 3 – 1 thread $2.07\ mill\ pixels * \left(\frac{1\ sec}{14.39\ mill\ pixels}\right)$ $= 0.1438\ \text{sec}\ or\ 6.952\ Hz$	**Kernel Size 3 – 8 threads** $2.07\ mill\ pixels * \left(\frac{1\ sec}{22.92\ mill\ pixels}\right)$ $= 0.09031\ \text{sec}\ or\ 11.07\ Hz$
Kernel Size 5 – 1 thread $2.07\ mill\ pixels * \left(\frac{1\ sec}{8.023\ mill\ pixels}\right)$ $= 0.2580\ \text{sec}\ or\ 3.876\ Hz$	**Kernel Size 5 – 8 thread** $2.07\ mill\ pixels * \left(\frac{1\ sec}{19.37\ mill\ pixels}\right)$ $= 0.1069\ \text{sec}\ or\ 9.357\ Hz$

Cat image in JS 1344x837 1344x837	
Kernel Size 3 – 1 thread $1.124\ \ mill\ pixels * \left(\frac{1\ sec}{12.00\ \ mill\ pixels}\right)$ $= 0.09367\ \text{sec}\ or\ 10.68\ Hz$	**Kernel Size 3 – 8 threads** $1.124\ \ mill\ pixels * \left(\frac{1\ sec}{17.18\ mill\ pixels}\right)$ $= 0.06542\ \text{sec}\ or\ 15.28\ Hz$
Kernel Size 5 – 1 thread $1.124\ \ mill\ pixels * \left(\frac{1\ sec}{7.264\ mill\ pixels}\right)$ $= 0.1547\ \text{sec}\ or\ 6.463\ Hz$	**Kernel Size 5 – 8 thread** $1.124\ mill\ pixels * \left(\frac{1\ sec}{14.96\ mill\ pixels}\right)$ $= 0.07513\ \text{sec}\ or\ 13.31\ Hz$

DogCat image in JS 4040x2693	
$10.8\ mill\ pixels * \left(\frac{1\ sec}{18.87\ mill\ pixels}\right)$ $= 0.5723\ \text{sec}\ or\ 1.747\ Hz$	**Kernel Size 3 – 8 threads** $10.8\ \ mill\ pixels * \left(\frac{1\ sec}{38.56\ mill\ pixels}\right)$ $= 0.2801\ \text{sec}\ or\ 3.570\ Hz$
Kernel Size 5 – 1 thread $10.8\ \ mill\ pixels * \left(\frac{1\ sec}{9.240\ mill\ pixels}\right)$ $= 1.168\ \text{sec}\ or\ 0.8556\ Hz$	**Kernel Size 5 – 8 thread** $10.8\ mill\ pixels * \left(\frac{1\ sec}{25.50\ mill\ pixels}\right)$ $= 0.4235\ \text{sec}\ or\ 2.361\ Hz$

The increase in performance is clear in JavaScript as well. From a user's perspective an increase from 6.463 Hz to 13.31 Hz would be dramatic. Threading this application clearly increases the resolution that would be possible to be used.

Appendix

To see the application run in action you can serve up the website.

1. `(base) stevenlarsen@Stevens-Mini Code % python website/simple_cors_server.py`
    a. You must serve the website from the Code folder.
2. Then navigate to http://localhost:8003/website/imdex.html

To see all the graphics and calculations there is a python notebook created in the Report folder titled “Report Diagrams.ipynb”.