Lab 3 CS 7349

Design Documentation of API Convolutions

1 Goal and Predefined filters.

Provide developers a library that can apply different predetermined convolutional filters to input images. The predefined filters are listed in section 3.

2 Class Diagrams/Available API's

2.1 From the C++ Perspective

ConvoutionFilters

```
ConvoutionFilters(int rows, int cols, int channels, debug=0)
void setInputImg(float *img, int size);
float* applyFilterCPP(const char* filterName);
float* getInputImgPtr();
void setThreadCount(long num);
```

We exposed the above functionality to the C++ library for use. The user must know the size of the image they want to run convolutions on. Once the object is created the user must call `setInputImg()` to populate the class variable `inputImg,` this is done with a deep copy. Then `applyFilterCPP()` can be called to perform the convolution.

1. `setInputImg(float *img, int size)`

   This method will perform a deep copy from the float array img to the class float array inputImg.

2. `applyFilterCPP(const char* filterName)`

   Performs the filter that is passed in as a char*. The returned float* is a reference to the class variable inputImg and contains the image output.

3. getInputImgPtr()
   Returns a pointer to the data that contains either the input image, before applyFilterCPP is called or the pointer to the output image after applyFilterCPP is called. This is the same pointer that is returned by that function.

Extra Exposed C++ Functions

```
float* applyConv(float *kernel, int kernelSize, int channels)
float* setInputImg(float* inputImg, int size)
float* applySharpen(int channels)
float* applyBlur(int channels)
float* applyGaussBlur(int channels)
float* edgeDetection(int channels)
float* applyEmboss(int channels)
float* applyLoG(int channels)
```

1. `float* applyConv(float *kernel, int kernelSize, int channels)`

   creates a new float array that contains the output of applying kernel to the inputImg. It is the responsibility of the caller to free the returned pointer.

2. The rest of the functions listed make a call to `applyConv`, delete inputImg and replace it with the output of the `applyConv` removing the responsibility of deleting the output array.

2.2 Memory Concerns

`applyFilterCPP()` will return a float pointer that represents the resulting image. This pointer will be freed up when the object is deconstructed. If there is a need for the data to live beyond the object it will have to be copied to somewhere else. When `applySharpen(), applyBlur(), applyGaussBlur, edgeDetection(), applyEmboss(), applyLoG(),` the pointer will live as long as the object is around. However when using `applyConv()`, it is the caller responsibility to clean up the pointer returned. This function will not be exposed to the JavaScript Library.

2.3 From the JavaScript Library Perspective

```
ConvoutionFilters(int rows, int cols, int channels, debug=false)
long getInputImgPtr()
long applyFilter(DOMString filterName)
void setThreadCount(long count);
```

1. ConvolutionFilters(int rows, int cols, int channels, bool debug)
2. long getInputImgPtr()
3. long applyFilter(DOMString filterName)

The exposed JavaScript functions are limited with the intention of simplifying the process. To set everything complete the following steps:

1. Create an instance of `ConvoutionFilters()` with the specified size and channels.
2. Get the pointer to the input Img using getInputImgPtr()
3. Set the data in `inputImg` (using `Module.set(…)`) passing in a JavaScript Typed array of Floats

4. Call `applyFilter()` with the specified filter and maintain the pointer that is returned. It points to the image data we need to save.
5. Call setThreadCount() to set the number of threads you want to run. The default will be 8 unless this method is called before applyFilter()

3 Predefined Filters

- Edge Detection – returns a black and white image with high pixels representing the pixels that are edges. Filters(3x3):
  - ```
    -1.0 -2.0 -1.0
    0.0   0.0  0.0
    1.0   2.0  1.0
    ```
  - ```
    -1.0  0.0  1.0
    -2.0  0.0  2.0
    -1.0  0.0  1.0
    ```
- Emboss – returns the resulting image of
  - ```
    1/2   -2.0 -1.0  0.0
          -1.0  1.0  1.0
           0.0  1.0  2.0
    ```
- Blur – Returns the result of the blur kernel provided. Goal is to blur edges of the image.
  - ```
    1/9    1.0  1.0  1.0
           1.0  1.0  1.0
           1.0  1.0  1.0
    ```
- Gaussian Blur – A bigger kernel based on the Gaussian blur technique, same goal as above.
  - ```
            1   4   6   4   1
            4  16  24  16   4
    1/256   6  24  36  24   6
            4  16  24  16   4
            1   4   6   4   1
    ```
- Laplacian of Gaussian – Referred to as LoG in the code. Runs a Laplacian filter then a Gaussian filter
    ```

- ○
```
 0.0 -1.0  0.0
-1.0  4.0 -1.0
 0.0 -1.0  0.0
```
- ○
```
-1.0 -1.0 -1.0
-1.0  8.0 -1.0
-1.0 -1.0 -1.0
```

- Sharpen – Applies a filter that will sharpen the edges of an image with the below filter:
  - ○
```
 0.0 -0.5  0.0
-0.0  8.0 -0.5
 0.0 -0.0  0.0
```

4. Threading Utilization

4.1 Thread Methodology

The only part of this library that is threaded is applyConv() from either environment. The library will spawn the number of specified threads and each thread will be assigned a set of rows to process. For example if the image was 100x100 and we use 10 threads each thread will process 10 threads.

4.2 Thread Debugging

To debug threads, pass in 2 to the debug parameter. It will only print out at the end of each thread specifying the rows that it processed and how long it took. While this is useful, it should be noted this uses C++ `cout` and that is not thread safe, the output may be irregular.

5. Code Examples:

Call `applyFilter` from C++

```
            ConvolutionFilters cf(img.rows, img.cols, img.channels(), 0);
            cf.setThreadCount(threadCount);
            cf.setInputImg(imgFloat, length);
```

Here we have an array of floats in `imgFloat` represented as a `float*`. The variable `rows, cols` and `channels` represents the shape of our image. setInputImg does a deep copy of the data in imgFloat. Then the string filter must be one of the following options: “EdgeDetection”, “GaussBlur”, “Blur”, “Sharpen”, “Emboss”, “Log”, any other string will apply the identity convolution. Output comes back as a float* that references the memory space in cf. Therefore, it is not the callers responsibly to free it. However, it will be freed up when `cf` leaves scope or is freed.

Call applyFilter from Javascript

```
const Module = require('./output.js');
```

You must require the output for the emcc compile to use the Module. Then you can create a instance of the module and use it. Like before in C++ we have to set the input image. But on this end it is easiest to use the Module.HEAP32.set() function to set it:

```
    var cf = new Module.ConvolutionFilters(src.rows, src.cols, 3, false);

    var inputImgAddresss = cf.getInputImgPtr();
    Module.HEAPF32.set(imgFloat, inputImgAddresss>>2); // int has 4 bytes

    cf.setThreadCount(threads);
    cf.applyFilter(filter);

    inputImgAddresss = cf.getInputImgPtr()/4;
    var localCopy = new Float32Array(Module.HEAPF32.subarray(inputImgAddresss,
(inputImgAddresss) + size));
```

After a call is made to applyFilter() the data in inputImg inside the module will have the result we want. To get at it, we will need to make a call to Module.HEAPF32.subarray() seen above.