Brice Ngnigha

Id# bdn423

Digital Image Processing with a GPU

Abstract

This paper describes the process of filtering a digital image using a Graphic Processor Unit. Image processing is one of the most important and widely used operation in distributing visual contents. The applications of this process can be extended to real time video processing, therefore requiring a very short window within which the computation should take place without causing a lag. In general the faster the process, the fastest the next stage. In this document, we will describe how to achieve optimal execution time using parallel computation and algorithm for processing.

1. Introduction

The purpose of processing a digital image is to extract or introduce various features to an original in order to improve the quality of the original image. A variety of algorithms are used for filtering the images such as the mean, the edge filter, the gaussian filter, many of which we will describe next. Then we will move on to the design goals, memory layout, and further we will describe few algorithms and how they fared at completing the task. We will conclude and provide some sample images obtained after applying some filters.

2. Techniques of Image Filtering

Many of the image filtering techniques form the linear filter subset. The linear filter replaces each pixel with a linear combination of its neighbors and convolution kernel is used in prescription for the linear combination. The linear filters have the mathematical formula,

$$\mathrm{y(t)}=\int_{-\infty}^{\infty}(h(r).\,x(t-r)dr),$$

where an input signal X is applied another heuristic signal H over a range to produce signal Y. Mathematically, we say that Y, with Y = X * H, is the convolution of signal X and H. Let's expand more on convolutions.

2.1. Convolution

A convolution is the process of combining two input signals to form a third signal. The interesting notion about the convolution is that they can be expressed as a matrix, in which case the matrix is called a convolution matrix, or kernel or mask. We will refer to convolution matrix in this paper as a kernel. Computing the convolution of two input signals becomes a matrix operation, such as the example next.

2.2. Example

Consider a 5 x 5 input matrix and a 3x3 kernel. The output matrix of this operation will be a 5x5 matrix, obtained by computing the below formula per input matrix entry.
To compute the entry at row 3 and column 3, we proceed as per figure 1.

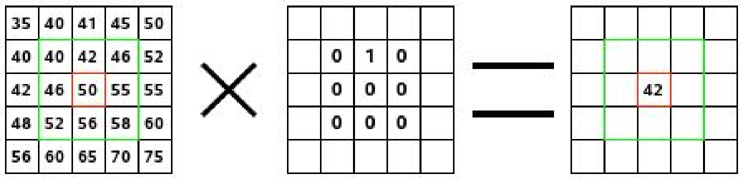


Figure1. (40*0)+(42*1)+(46*0) + (46*0)+(50*0)+(55*0) + (52*0)+(56*0)+(58*0) = 42

2.3. Kernels

In choosing the kernels to apply to an image we have to keep in mind few things:

- The kernel must be a square matrix. The row and column sizes must be an odd number, to ensure that each pixel has a consistent count of neighbors to compute its value. For a 3x3 kernel, each pixel will have exactly one neighbor in each direction. With 5x5 kernel, twp neighbors.
- The kernel must be normalized in order to avoid modifying the brightness of the original image
- Multiple kernels matrices can be convoluted to form a single kernel before applying it to the input image.

Some of the common kernels are:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 2. Left to right. Edge detection, Sharpen, Gaussian Blur, kernels

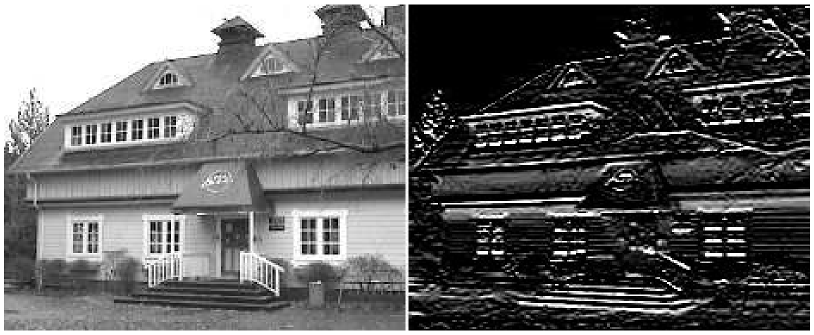


Figure3. Left original image, Right filtered with edge detection kernel

3. Design Goals

Most application that involve image processing focus around both the quality and the execution time. The quality can be addressed via the choice of the kernels to be applied to the image. The part of interest to us as we design this tool are the execution time, which can be achieved by utilising the parallelism, optimizing the memory access, Reduce the synchronization between the threads, aim for the work optimal solution.

3.1. Use of Parallelism

We will use the GPU to perform the convolution computation in parallel. GPUs are high throughput devices, slightly slower than modern day CPUs on clocking, but that along with a good choice of algorithm can perform a lot of work fast.

3.2. Optimize Memory Access

How fast we access data in memory depends how we align it first. We want to align the memory such that each pixels to be computed can locally obtain its neighbors values and avoid memory misses which are very expensive. Accessing the global memory is usually much slower than the shared memory in the Cuda language. To this effort we will use shared memory.More on this in Memory Layout section.

3.3. Reduce Thread Synchronization

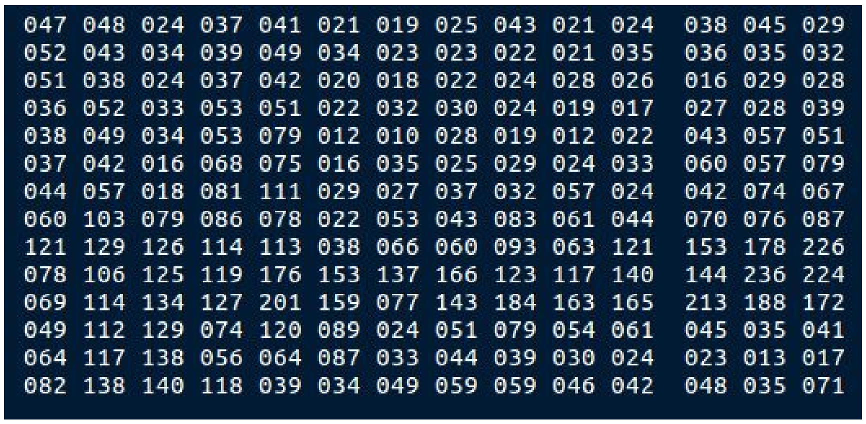
We will partition the data such that a thread need not to always synchronize by locally reading from the shared memory but writing the result directly to the global memory

3.4. Optimize Work

Dedicate a single worker for shared computation such as the starting address of a block, before other threads can compute their own address. Using that block start address. Aim for the work optimal solution.

4. Memory Layout

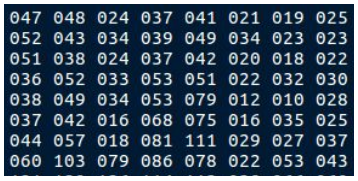
Given a picture 14x14 with the below R(of RGBA) values. Memory is aligned in a 1 dimension array. We want to apply a 3x3 kernel to this image. We select to use Cuda blocks of size 8x8 to compute the convolution of each pixel.



```
047 048 024 037 041 021 019 025 043 021 024  038 045 029
052 043 034 039 049 034 023 023 022 021 035  036 035 032
051 038 024 037 042 020 018 022 024 028 026  016 029 028
036 052 033 053 051 022 032 030 024 019 017  027 028 039
038 049 034 053 079 012 010 028 019 012 022  043 057 051
037 042 016 068 075 016 035 025 029 024 033  060 057 079
044 057 018 081 111 029 027 037 032 057 024  042 074 067
060 103 079 086 078 022 053 043 083 061 044  070 076 087
121 129 126 114 113 038 066 060 093 063 121  153 178 226
078 106 125 119 176 153 137 166 123 117 140  144 236 224
069 114 134 127 201 159 077 143 184 163 165  213 188 172
049 112 129 074 120 089 024 051 079 054 061  045 035 041
064 117 138 056 064 087 033 044 039 030 024  023 013 017
082 138 140 118 039 034 049 059 059 046 042  048 035 071
```

Figure 4. R values of a 14x14 image

Using shared memory of size 8x8 will allow to copy the global memory and obtain the following representation of the shared memory in block 0.

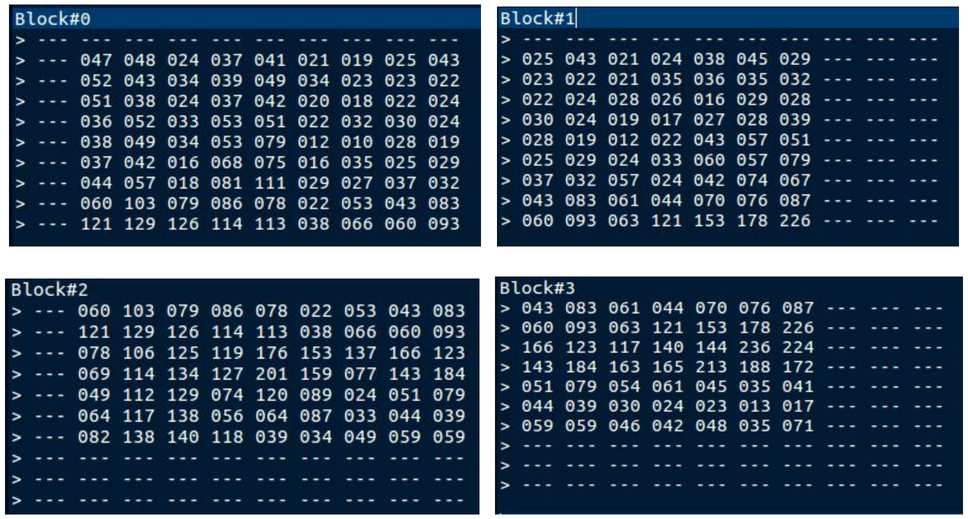


```
047 048 024 037 041 021 019 025
052 043 034 039 049 034 023 023
051 038 024 037 042 020 018 022
036 052 033 053 051 022 032 030
038 049 034 053 079 012 010 028
037 042 016 068 075 016 035 025
044 057 018 081 111 029 027 037
060 103 079 086 078 022 053 043
```

Figure 5. Shared memory of size 8x8

The pitfall of this representation is that to compute the convolution at the edges, we will have to access the global memory. For example computing the convolution of pixel at row seven column seven, bottom right pixel (043), we will be accessing the global input array to query another five pixels. This issue can be circumvented by increasing the block dimension by 2n extra threads for both the row and the columns, where n is the number of neighbors.

That is for, for our 14x14 image, 3x3 kernel, n = 1 neighbor in each direction of a pixel, rather than a block size of 8x8, we will use a 10x10 = (8+2*n, 8+2*n) block of threads and copy over the neighboring pixels from the input image to the shared memory, to ensure locality of data within the block. For pixels that are not accessible from the input image, we will mar the thread as invalid for computation.
After this task we will have the following representation of the four block in this particular set.



```
Block#0
> --- --- --- --- --- --- --- --- --- ---
> --- 047 048 024 037 041 021 019 025 043
> --- 052 043 034 039 049 034 023 023 022
> --- 051 038 024 037 042 020 018 022 024
> --- 036 052 033 053 051 022 032 030 024
> --- 038 049 034 053 079 012 010 028 019
> --- 037 042 016 068 075 016 035 025 029
> --- 044 057 018 081 111 029 027 037 032
> --- 060 103 079 086 078 022 053 043 083
> --- 121 129 126 114 113 038 066 060 093
```

```
Block#1
> --- --- --- --- --- --- --- --- --- ---
> 025 043 021 024 038 045 029 --- --- ---
> 023 022 021 035 036 035 032 --- --- ---
> 022 024 028 026 016 029 028 --- --- ---
> 030 024 019 017 027 028 039 --- --- ---
> 028 019 012 022 043 057 051 --- --- ---
> 025 029 024 033 060 057 079 --- --- ---
> 037 032 057 024 042 074 067 --- --- ---
> 043 083 061 044 070 076 087 --- --- ---
> 060 093 063 121 153 178 226 --- --- ---
```

```
Block#2
> --- 060 103 079 086 078 022 053 043 083
> --- 121 129 126 114 113 038 066 060 093
> --- 078 106 125 119 176 153 137 166 123
> --- 069 114 134 127 201 159 077 143 184
> --- 049 112 129 074 120 089 024 051 079
> --- 064 117 138 056 064 087 033 044 039
> --- 082 138 140 118 039 034 049 059 059
> --- --- --- --- --- --- --- --- --- ---
> --- --- --- --- --- --- --- --- --- ---
> --- --- --- --- --- --- --- --- --- ---
```

```
Block#3
> 043 083 061 044 070 076 087 --- --- ---
> 060 093 063 121 153 178 226 --- --- ---
> 166 123 117 140 144 236 224 --- --- ---
> 143 184 163 165 213 188 172 --- --- ---
> 051 079 054 061 045 035 041 --- --- ---
> 044 039 030 024 023 013 017 --- --- ---
> 059 059 046 042 048 035 071 --- --- ---
> --- --- --- --- --- --- --- --- --- ---
> --- --- --- --- --- --- --- --- --- ---
> --- --- --- --- --- --- --- --- --- ---
```

Figure 6. Shared memory of size 8x8

We can now locally compute the convolution of the same pixel in block 0, row 7, column 7, with having locally its neighbors. Note that with this scheme, once the data is copied in the shared memory, no more synchronization in necessary as we need not to write the result of this operation to the shared memory.

5. Algorithms

We chose to evaluate two techniques in solving this problem. One using sequential computation of the convolution and the other using dynamic parallelism. Given an image of size NxM, and a kernel of size KxK, we do.

5.1. Algorithm with sequential convolution

We compute i= floor(sqrt(k)), the number of neighbors in each direction.

- For block dimension bY, bX, launch the GPU kernel with size bY+2i, bX+2i. Allowing to copy the ith neighbor if valid.
- Initialize some dynamic shared memory for each block of size blockDim.x by blockDim.y
- Use a constant function that maps index in the shared memory to an index in the input array

- In parallel copy from global to shared memory
- For thread within a block, compute the convolution of each entry and its i neighbors, with the kernel matrix if the pixel is valid. For example the pixel at 0,0 in block 0 will not be computed, but 1,1 will.
- Compute the convolution in time O(k^2) using a sequential convolution and iterating over the KxK kernel and the mapping in the shared memory.
- The algorithm therefore takes O(k^2)ut...

5.2. Algorithm with dynamic parallelism

For the same problem size, and kernel size do:

- Create enough blocks to cover the size of the input array
- Each thread to work on a single entry in the input array
- Each thread to spawn a kernel of block size that of the kernel
- In log(k^2) compute the convolution doig a map operation, then a parallel reduce and store the result directly to the global output array
- Synchronization needed only in the sub-blocks in computing the parallel reduce algorithm

6. Algorithm evaluation

It was found at running time that the first algorithm using sequential convolution and shared memory fared far better than the second algorithm with dynamic parallelism, although the time complexity showed contrary, O(k^2) vs O(log(k^2)) where k is the size of the kernel. This is because launching sub-kernels requires some overhead and the sub-threads in dynamic parallelism access the global memory to read and to write. For example for a 14x14 image, 196 first level threads and each to launch 9 second level threads, totaling 1764 threads that will read the global memory 1764 times. For a 1920x1080, this racks up to 2.1Millions threads for 18.5 million reads, causing a lot of cache misses. THe performance of the second algorithms was a thousand times slower than that of the first algorithm.

7. Future Work

In the future we will continue on this project to provide a Graphical User Interface in order to better experience the benefits of this tool, also add in the implementation of nonlinear filters such as the median filter. Implement Steganography and Watermarking with a GPU.

8. Conclusion

Digital image processing is requires the lowest possible execution time. It has been proven through this project that designing an algorithm for parallel computing requires more consideration than the theory of time and work complexities. The memory alignment and hierarchy as well as the amount of synchronization needed plays a role just as important if not more important. Therefore, one must put into consideration these parameters when designing a solution for image processing.

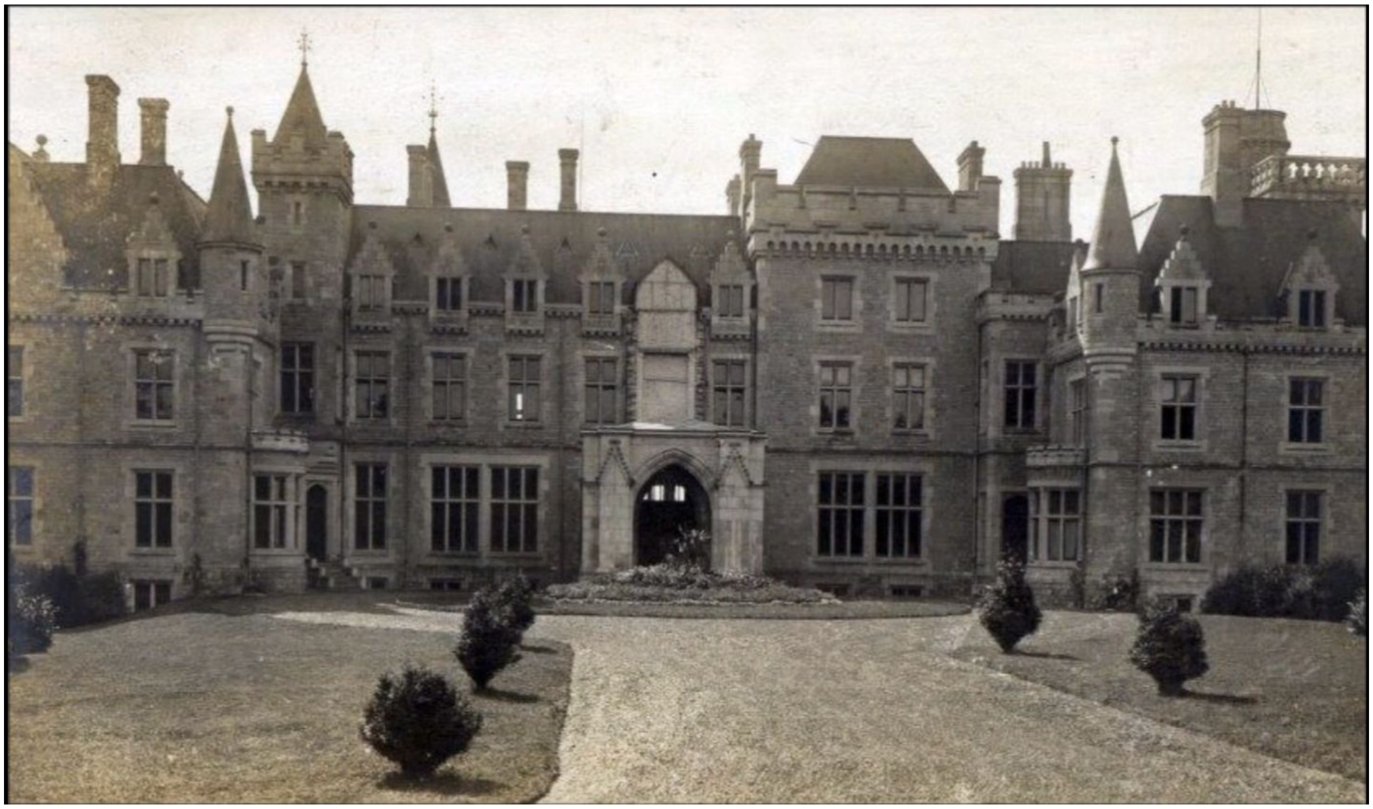


Figure 7. Original image

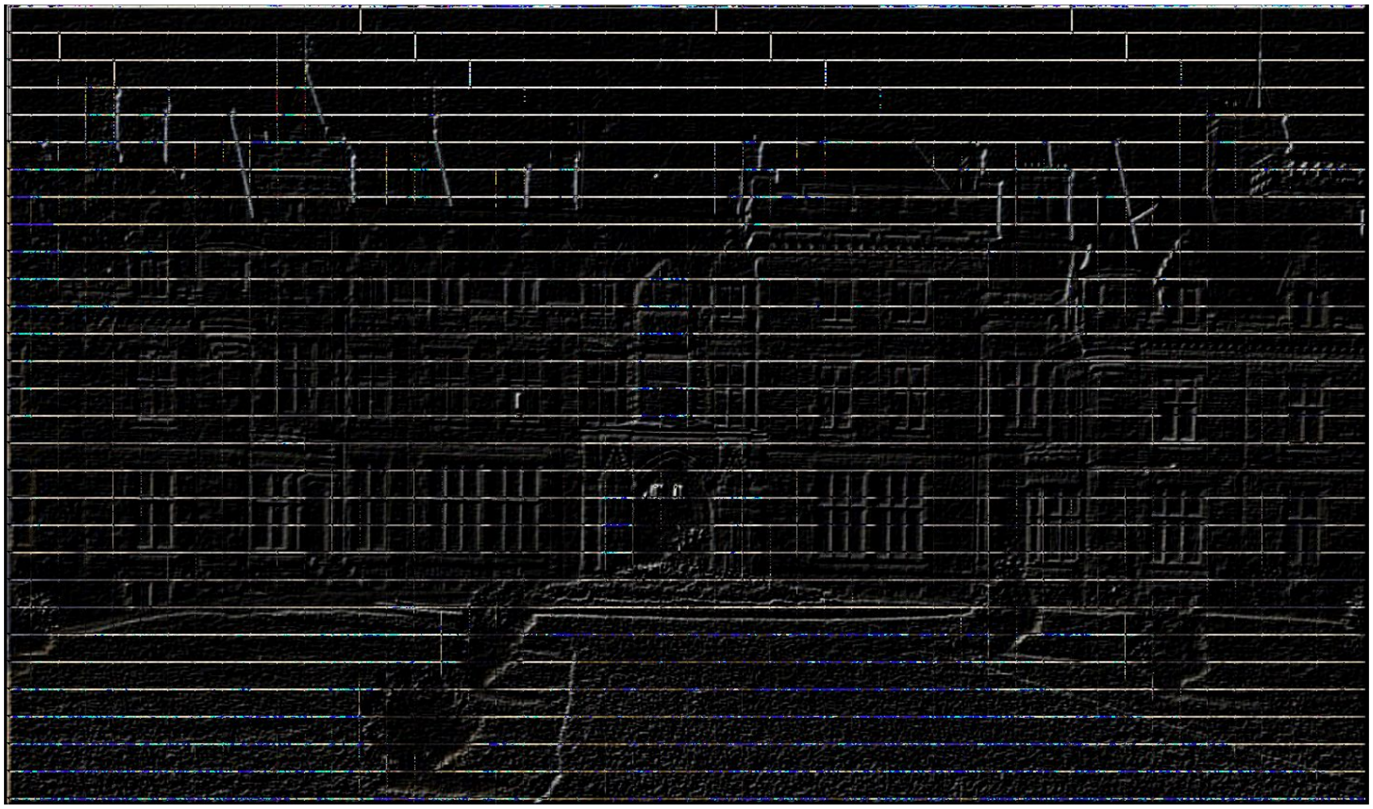


Figure 8. Emboss filtered

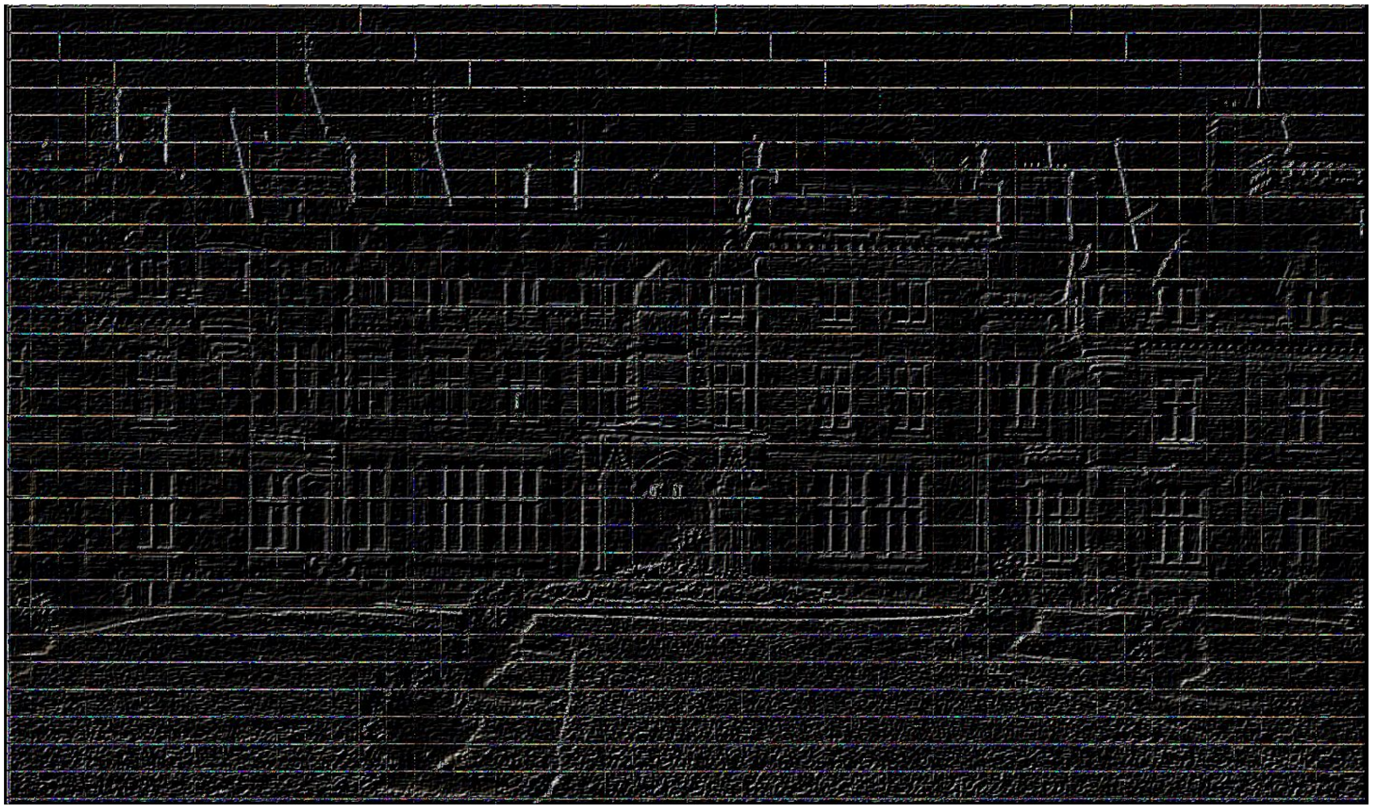


Figure 9. Emboss filtered then sharpen filter

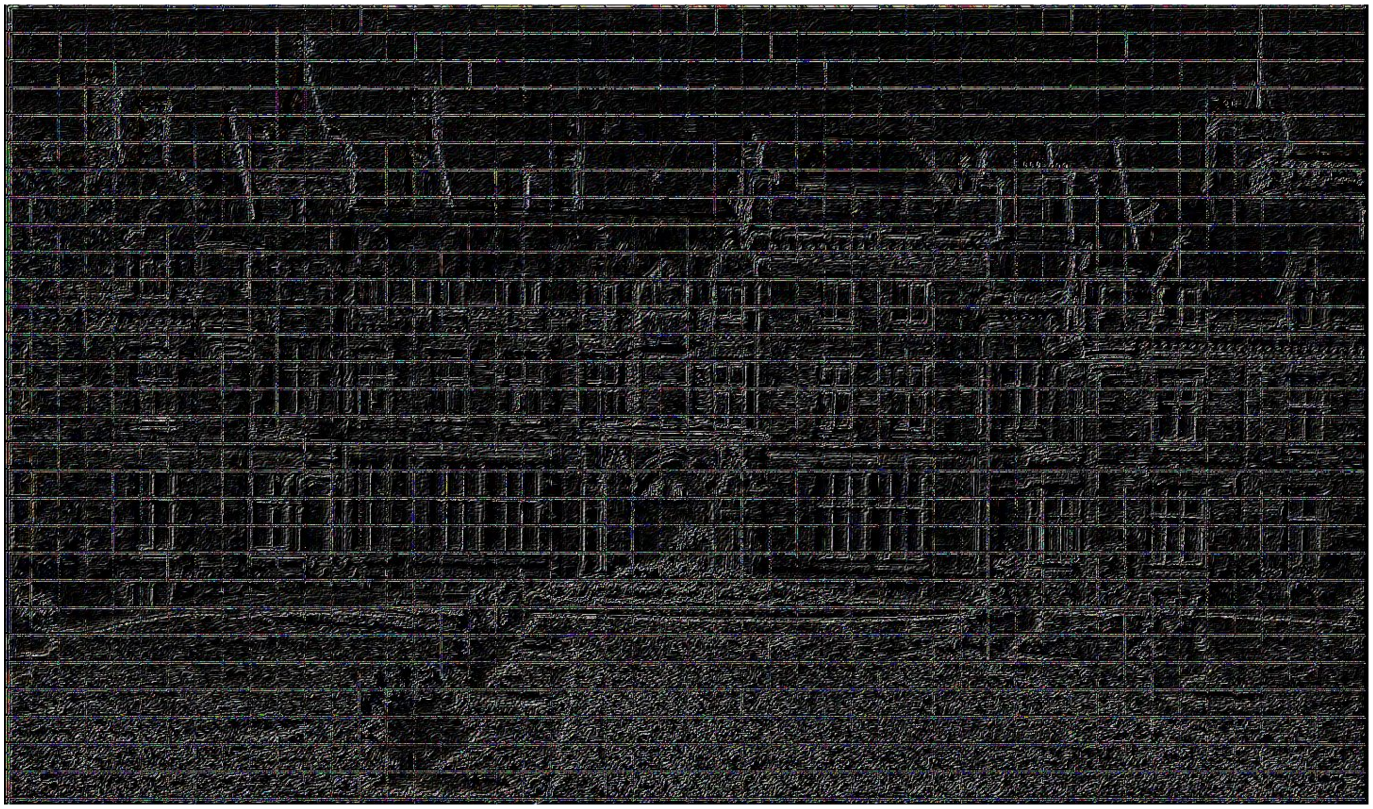


Figure 10. Emboss filtered then sharpen filter then edge finder filter applied