WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A Assignment 3 – Report

Wesley Earl Stander - 1056114 - Coms Hons

May 20, 2020

1 Design Choices

1.1 Sequential

The initial implementation of sequential convolution made use of an unsigned char array to store the output. This was found to cause incorrect results as the values were being truncated to fit into the data type. The output function was changed to use a float data type and an additional function was implemented to re-scale the output values to the range of 0 to 255 as well as convert the array to an unsigned char data type. The padding of the image is attained through a check where the addition of the value is omitted where it requires out of bounds values from the image to be acquired hence adding 0 similarly to using a padded array. The algorithm to calculate the pixel value was extracted to a function but saw loses presumably from the repeated function call for each pixel. The loss was large so the function call was removed. The performance speed up from removing the function call was 1.0795x. The speed up was inconclusive and it showed performance gains up to 2x. Therefore the version without the function call was kept. The offset for the kernel was using a round call to calculate it. The calculation was changed to an integer division increasing the performance by a speed up factor of 1.0337x. This was used for all CUDA kernels to increase performance.

1.2 Global and Constant

The initial implementation of the global kernel is very similar to the sequential kernel but instead of all pixels being calculated in a double loop, each thread calculates a pixel's value. The algorithm to calculate the pixel value was extracted to a device function but saw very small loses presumably from the single function call in each thread. The loss was negligible but noticeable so the function call was removed. The speed-up was 1.0018x. The kernel was initially accessing the memory in column major order using the x thread for the y coordinate. The program was then changed so that the x-thread uses the x coordinate accessing the memory from global memory in row major order hence increase the speed by 10x as much allowing for better memory coalescence between threads in the block. After the kernel size was increased past 3x3 the global kernel kept producing the error for too much launch resources so the kernel was improved with a __launch_bounds__ (1024) parameter to prevent the kernel from invoking more threads than the maximum. This was peculiar as the maximum threads were not being surpassed by the block size of 32x32 which is the maximum of 1024. After this the kernel worked without a problem on larger filter kernel sizes.

1.3 Shared and Constant

Initially the tiles were loading halo elements at their intersections by each tile handling a set of corner pixels, it was changed so that the threads which load the top and bottom halo cells also load both corner set of halo cells so that memory access is coalesced. This was changed before measurements were taken but it is assumed to increase performance due to memory access coalescence. This implementation proved unsuccessful as the speedup from global convolution to shared is 0.7497x. This negative speed-up is from the lack of consecutive thread memory access whereas, the final thread is accessing memory from the last element of it's tile and the last element of the previous tile. This is causing stride based memory access for the usage of halo cells. This implementation extended from the 1D convolution in the notes is causing much strided memory access and has slowed down the performance substantially. The strided memory access works considerably better in 1D although it struggles in a 2-dimensional implementation (this implementation will be referred to as the strided implementation). The next approach that was considered was to separate the kernels into 1D kernels to reduce the computational complexity. This is possible with the averaging filter and the edge detection filter but is not possible with the sharpening filter. As it is not possible with the sharpening filter, the approach was not implemented for this assignment. The next approach considered was a system of overlapping tiles, 1 for each corner that would load all the elements. This implementation loads each of the central values 4 times. This approach showed even slower times as compared to the strided memory access approach showing a speed up of 0.9768x. This is not optimal although the algorithm has space for improvement. For kernels smaller than the tile size, many values are copied multiple times. The first improvement was to increase the level of memory coalescence by making the tile size closer to the warp size for better memory reading. The filter kernel for testing was 3x3. This would mean to bring the memory access speed down the amount of memory access by a warp would need to be coalesced and hence 32 registers. For the strided memory access version the speed was maximum when the tile size + kernel width are less than 32 but as close to 32 as possible. Hence the fastest speed was acquired when the tile size was 16x16. The tiled memory access showed performance increases at this tile size as well. Greater performance increases than the strided memory access. The strided memory access in comparison to global achieved a speed up of 0.8147x which is much larger than the original tests at 8x8 tiles. The tiled memory access in comparison to global achieved a speed up of 0.8834x which is even higher than the strided and is getting closer to the speed of the global implementation. The strided implementation has diminishing gains from tile sizes bigger than 16 as the memory access becomes larger than a warp. The tiled memory access has increased speed up at tile size 32 as it accesses memory exactly within the size of a warp giving an increase even over the global implementation before optimization. The speed up of tiled memory access over global is 1.005x speed up. The performance of this kernel is good and struggles at small filter kernel sizes.

After the tiled kernel was optimized it was tested with filter kernels of larger sizes which saw large increases in performance over the global kernel. For a 9x9 filter kernel the speed up from tiled over global is 15.967x and theoretically increases proportionally with the filter kernel width. The strided also showed similar performance increases although slightly less. The speed up for this filter kernel from the tiled over the strided is 1.1267x. This is partly due to the better memory access coalescence. As the kernel size increases the strided and tiled shared implementations converge but they show more significant differences in performance at smaller tile sizes.

1.4 Texture

The texture implementations use the algorithm as the global implementation but access memory from the texture memory instead of global memory. The 2D texture memory implementation uses 2D texture memory to store the image and the filter kernel. The implementation that uses only 2D texture memory for the image and the filter kernel is slower than the implementation that uses texture memory for the image and constant memory for the kernel. The speed up with constant memory filter kernel over texture memory filter kernel is 1.16x. A 1D texture memory implementation was tried after the 2D implementation whereby the image is stored as 1D texture memory and the filter kernel is stored as 1D texture memory. For the image tested and a small filter kernel size the 1D texture memory kernel was 1.124x the speed of the 2D texture memory kernel. For larger filter kernel sizes the 2D filter kernel was much faster than the 1D filter kernel. The larger the filter kernel, the quicker the speed up was for using constant memory to store the filter kernel instead of using texture memory for both 1D and 2D implementations. The 1D implementations shows less performance than the 2D texture implementation at larger filter kernel sizes. The constant memory being used for the filter kernel speeds up both 1D and 2D texture implementation significantly. After these performance changes were noted, the next approach that was attempted was to use the faster shared memory implementation, the tiled implementation with 2d texture memory for the image and constant memory for the kernel. This implementation was significantly faster than the other texture implementations with a 4x speed-up with a filter size of 15x15 from 2d texture memory with constant filter. This implementation is far more optimal method to continue implementing texture memory usage. The tiled shared memory implementation was implemented for 1D texture memory with a constant kernel. This was done to see the range whereby each is better than the other. The 1d shared texture implementation performed slightly better at smaller filter sizes than the 2D implementation following a similar trend to the texture only implementations. At larger filter sizes the 2D implementation outperforms the 1D implementation.

2 Performance Comparison

2.1 Graphics card for tests

```
CUDA Driver Version / Runtime Version 10.2 / 9.1
---Device 0---
Name: "GeForce RTX 2060"
CUDA Capability Major/Minor version number: 7.5
--- Memory information for device ---
Total global mem: 5926 MB
Total constant mem: 65536 B
The size of shared memory per block: 49152 B
The maximum number of registers per block: 65536
The number of SMs on the device: 30
The number of threads in a warp: 32
The maximal number of threads allowed in a block: 1024
Max thread dimensions (x,y,z): (1024, 1024, 64)
Max grid dimensions (x,y,z): (2147483647, 65535, 65535)
```

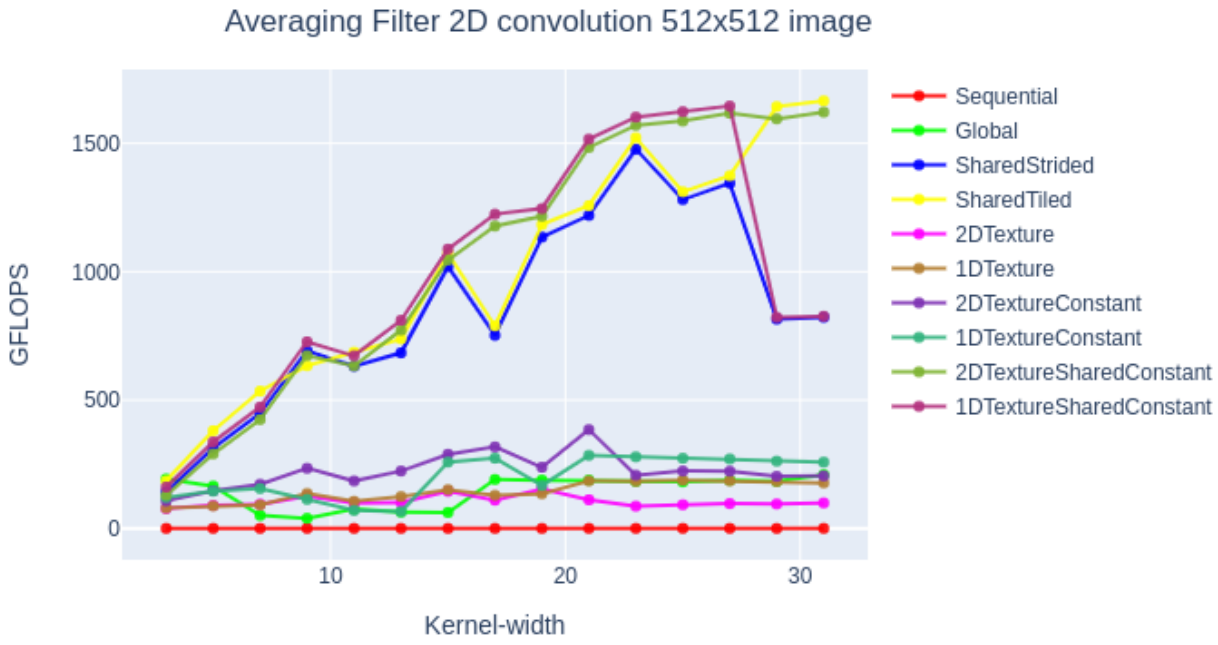

Figure 1: Averaging filter 2D convolution with varying filter kernel size and 512x512 image

All tests were averaged over 100 iterations. From the output of the program and the graph in figure 1 it can be seen that any CUDA kernel performs significantly better than the sequential algorithm with the worst performing kernel still achieving 188x speed-up. The best performing CUDA kernel was the shared tiled implementation. It can be seen that the shared implementations outperform all other implementations even achieving a speed up nearly 5x over the global implementation at the largest kernel size tested of 31x31. It is also noteworthy that performance of the kernel's that use texture memory perform better when the filter is not in texture memory but rather constant memory. The solely Texture based implementations perform worse than the global memory at the larger filter sizes and much worse than the shared memory implementations. It is assumed that texture memory would work better if larger images were used. The tiled implementation narrowly outperforms the strided implementation. The 2D texture implementation outperforms the 1D implementation at most kernel sizes except smaller than 5 and bigger than 23. This provides an option whereby specific implementations of texture memory can be chosen depending on how they perform on a GPU for their respective task. The 1D texture shared implementation and the strided shared implementation lose performance when the kernel sizes near the size of a warp causing memory coalescence issues. The 1D and 2D texture only implementations are almost the same but diverge at the larger filter sizes whereby the 1D texture only implementation outperforms the 2D texture only implementation. This is uncharacteristic of the other data and is due to the filter kernel originally being 1 dimensional and all other implementations read it in 1 dimension so the dimensional difference between all implementations and 2d texture only implementation cause a performance loss in comparison.

From the graph in figure 2 it can be seen that all algorithms follow the same trend with increases in image

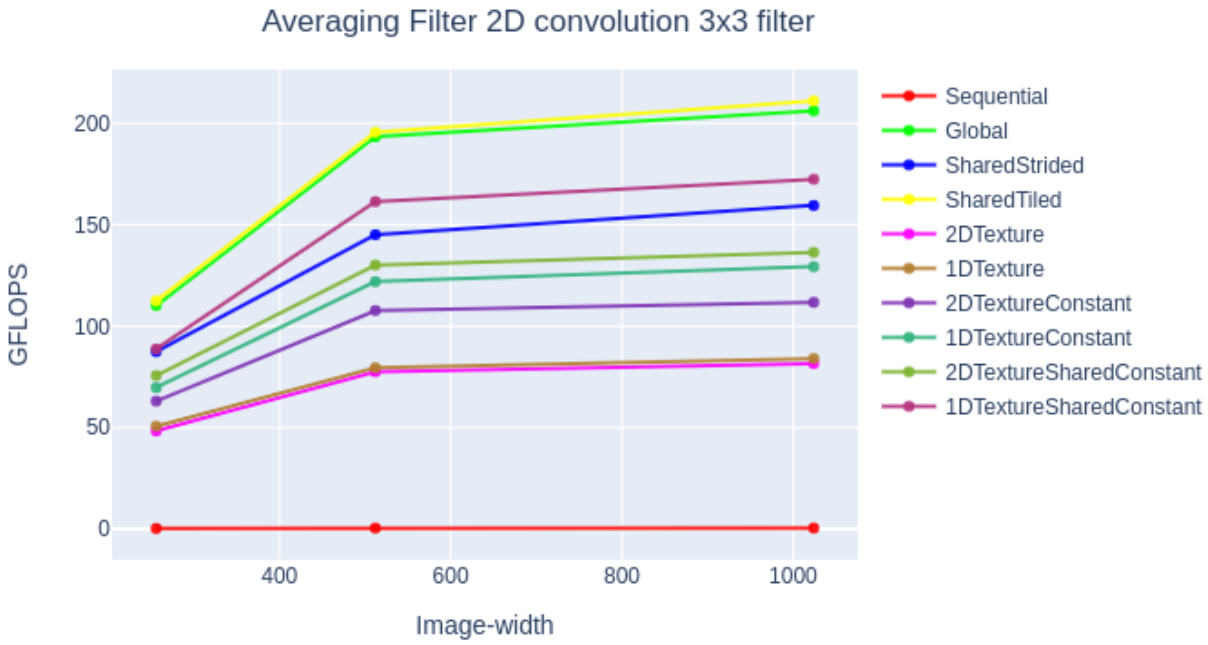

Figure 2: Averaging filter 2D convolution with varying image size and 3x3 kernel

size. The algorithms show a larger 1D texture performance at this small kernel than 2D texture. At larger kernel sizes the 2D texture memory implementation outperforms the 1D texture memory implementation as discussed in section 1.

The graph in figure 3 shows that the performance of the different algorithms differ greatly with the difference in filter kernel sizes. The larger kernels show better performance with the 2d texture implementations whereas the 1D texture implementation performs better at smaller kernel sizes. The image size also drastically affects the performance of the shared implementation showing large performance losses for the largest image size for the biggest kernel. This doesn't follow the same trend as in figure 2.

Overall it can be seen that the tiled shared implementation has the best performance narrowly better than the 2D texture with constant implementation. They shared implementation styles and it also informs that global memory is slightly faster than texture memory. The 2D implementations outperform the 1D implementations at larger filter kernel sizes and 1D implementations outperform 2D at smaller filter sizes. The global implementations can be seen outperforming texture only implementations. The largest performance increase achieved was 9.11x speed-up with the shared tiled implementation over the naive global memory implementation. Compared to the sequential implementation, the performances of the CUDA kernels make the sequential implementation non-viable in any situation except to verify the output of the kernel.

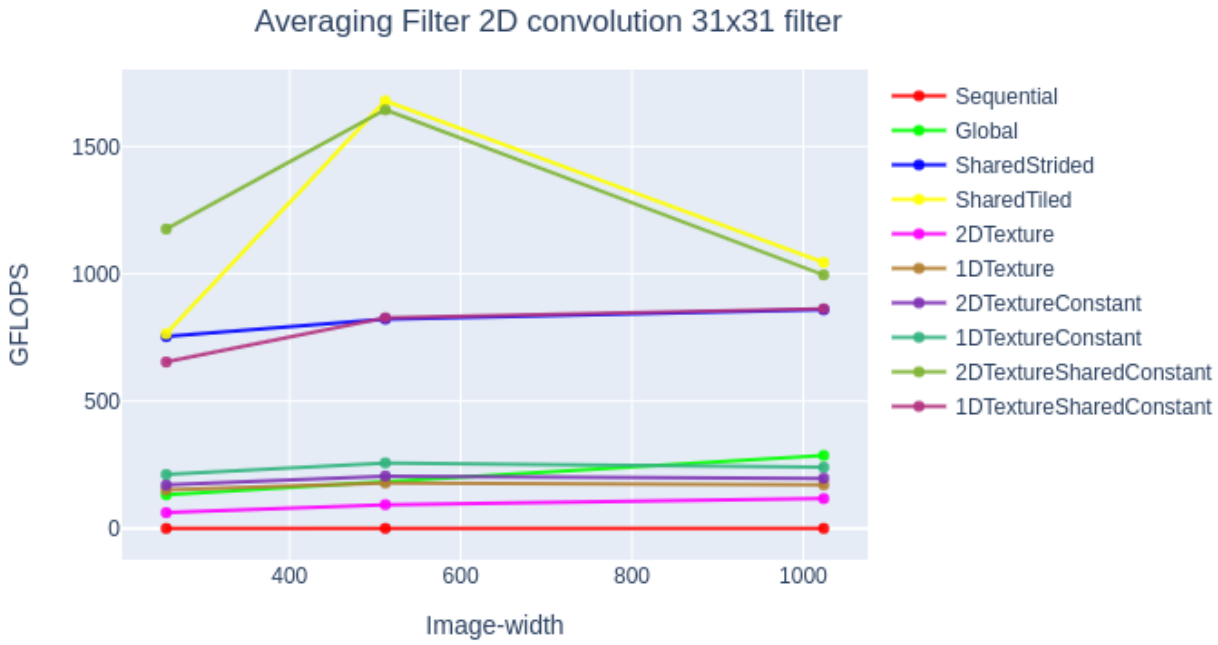

Figure 3: Averaging filter 2D convolution with varying image size and 31x31 kernel

3 Discussion on CUDA device memory

From the experiments performed, it would seem that implementations that utilize shared memory perform significantly better than all other implementations. Shared memory is specific to a block on the GPU grid so it's closeness to the processing unit helps speed up the processing. Texture memory is the furthest away from the processing unit and holds a similar scope as global and constant memory. Texture memory however is optimized for read, writes and performs similarly to global memory regardless of it's distance away from the processing units. Constant memory is optimized for reads and is significantly faster than texture and global memory. It may be faster than shared memory although experimentally this was not determined. Global memory is significantly slower than the shared memory and constant memory. As evaluated in the experiments above and using knowledge about the GPU memory design, the optimal implementation would make use of shared memory and constant memory where applicable. The observation made is that global memory is slightly faster than texture memory. This is understandable as global and texture memory have different purposes. The texture memory is meant to hold objects for multiple reads whereas global memory is practically used by everything so it needs to be optimized.

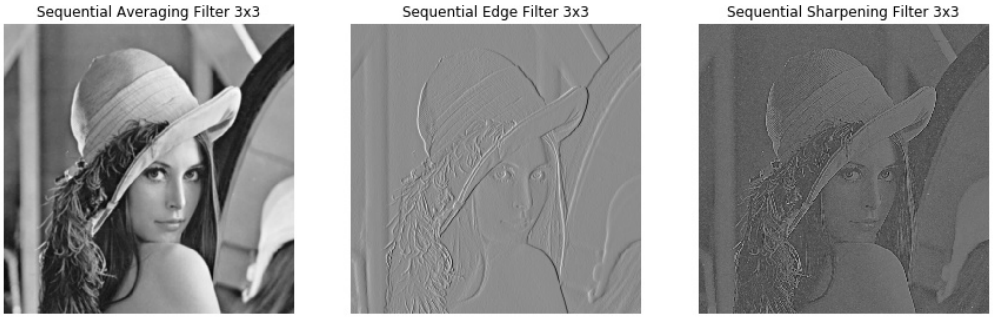

Figure 4: Sequential Results

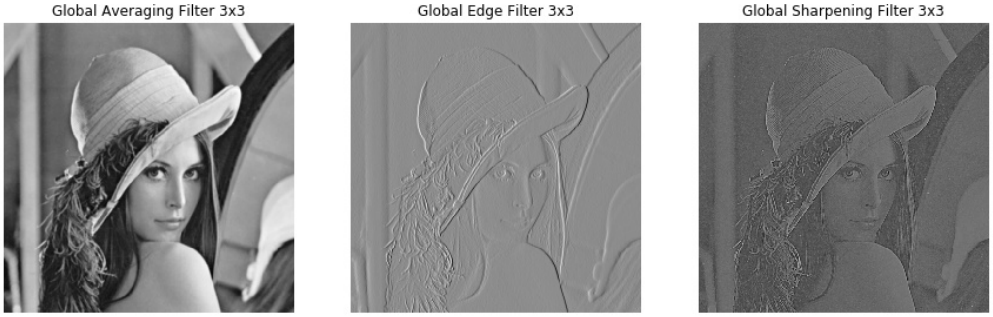

Figure 5: Global Results

4 Output Images

5 Questions and Answers

a) The number of floating point operations are equal to an addition and a multiplication for every kernel value for every pixel which is = image width * image height * kernel width * kernel height * 2.

b) The global memory reads of the global kernel are image width * image height * kernel width * kernel height. The global memory reads for the strided memory access are image width * image height + (((int)(kernel width / 2) * 2) + 32)**2 - 32**2. The global memory reads for the tiled memory access are ((image width/32 - (int)(kernel width/2))*2)**2.

c) Both strided and tiled implementations only write image width * image height times to global memory.

d) It starts to plateau in performance. According the the graph above both strided and tiled access produce a curve that flattens out at larger filter kernel sizes. This is due to the requirement to perform alot more global memory reads than at smaller filter kernel sizes.

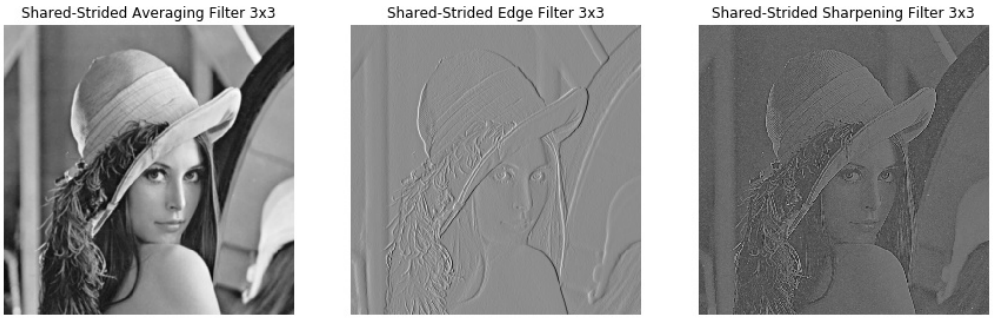

Figure 6: Shared Strided Results

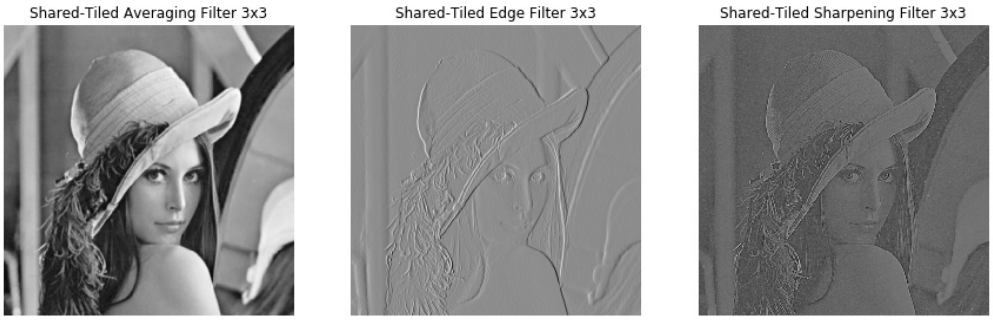

Figure 7: Shared Tiled Results

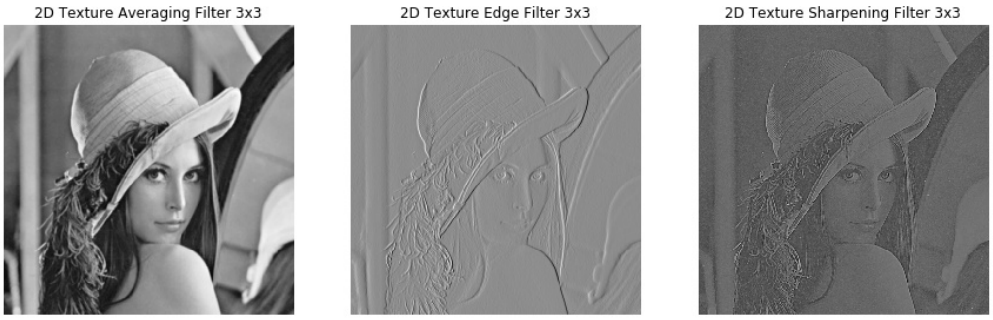

Figure 8: 2D texture only Results

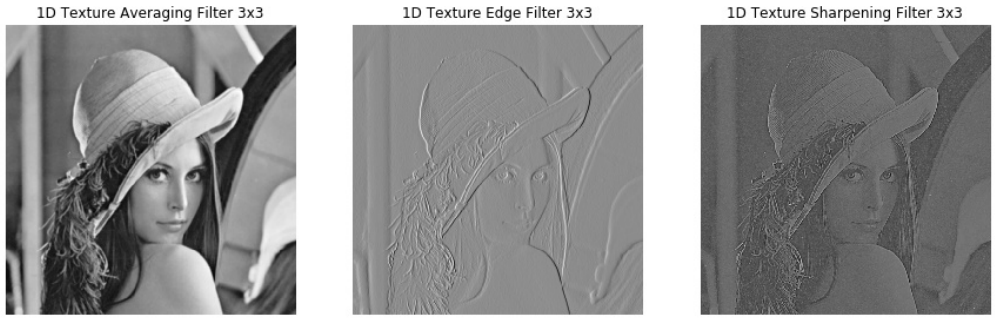

Figure 9: 1D texture only Results

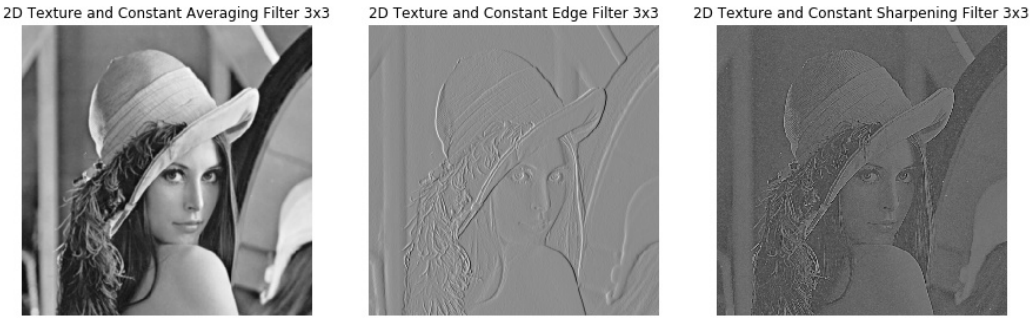

Figure 10: 2D texture and constant filter Results

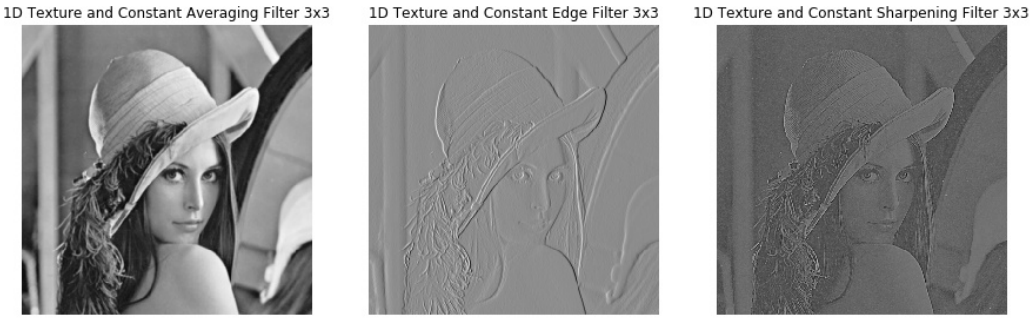

Figure 11: 1D texture and constant filter Results

Figure 12: 2D shared-tiled texture and constant filter Results

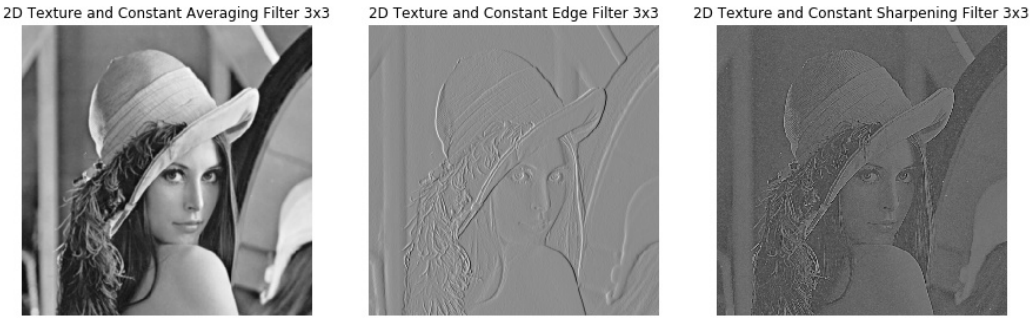

Figure 13: 1D shared-tiled texture and constant filter Results