# CSE435
# HW2

Burak Kocausta
1901042605

## Content



## 1- Installation, Compilation, Run

**Installation of NVCC:**

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
%load_ext nvcc_plugin
```

**Compilation:**

```
!nvcc -arch=sm_75 -o "/content/src/convolution_with_shm.o"
/content/src/convolution_with_shm.cu
```

- “stb_image.h” and “stb_image_write.h” files must be in the compilation path.

**Example Run:**

```
!/content/src/convolution_with_shm.o /content/src/image_03.png
/content/src/conv_03.png
```

## 2- Implementation of Convolution Operator without Using Shared Memory (Q1)

Implementation:

```
__global__ void convolutionKernel(const float *input, float *output, int width,
                                  int height, const float *kernel, int kernelSize) {
    int center = (kernelSize - 1) / 2;

    // x is the column index, y is the row index
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // each thread writes which pixel it is processing
    // printf("x: %d, y: %d\n", x, y);

    if (x >= center && x < width - center && y >= center && y < height - center) {
        float sum = 0.0;

        // apply convolution using shared memory for the kernel
        for (int ky = 0; ky < kernelSize; ++ky) {
            for (int kx = 0; kx < kernelSize; ++kx) {
                int imageX = x + kx - center;
                int imageY = y + ky - center;

                sum += input[imageY * width + imageX] * kernel[ky * kernelSize + kx];
            }
        }

        // set the output pixel value
        output[y * width + x] = sum;
    }
}
```

- Kernel function for non-shared part.

```
// configure and launch the CUDA kernel with shared memory
dim3 blockDim(blockDimX, blockDimY);
dim3 gridDim(gridDimX, gridDimY, gridDimZ);
```

```
// start timing
cudaEventRecord(start);
convolutionKernel<<<gridDim, blockDim>>>(d_input, d_output, width, height, d_kernel, kernelSize);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
```

- Calling kernel with given dimensions.

## 3- Implementation of Convolution Operator with Using Shared Memory (Q2)

Implementation:

```
__global__ void convolutionKernel(const float *input, float *output, int width,
                                  int height, const float *kernel, int kernelSize)
    int center = (kernelSize - 1) / 2;

    // x is the column index, y is the row index
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // fill kernel into shared memory
    // if (threadIdx.x == 0 && threadIdx.y == 0) {
    //     for (int i = 0; i < kernelSize * kernelSize; ++i) {
    //         sharedKernel[i] = kernel[i];
    //     }
    // }

    extern __shared__ float sharedKernel[];
    int K2 = kernelSize * kernelSize;
    if (threadIdx.x < K2) {
        sharedKernel[threadIdx.x] = kernel[threadIdx.x];
    }

    __syncthreads();
```

```
    if (x >= center && x < width - center && y >= center && y < height - center) {
        float sum = 0.0;

        // apply convolution
        for (int ky = 0; ky < kernelSize; ++ky) {
            for (int kx = 0; kx < kernelSize; ++kx) {
                int imageX = x + kx - center;
                int imageY = y + ky - center;

                sum += input[imageY * width + imageX] * sharedKernel[ky * kernelSize + kx];
            }
        }
        // set the output pixel value
        output[y * width + x] = sum;
    }
}
```

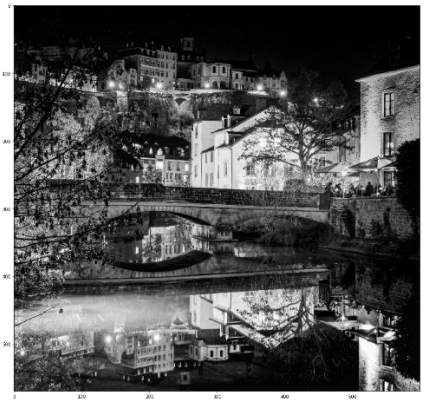
- Kernel function with shared memory. Kernel filling loop is commented out because of optimization. Other version gives better performance.

```
// Configure and launch the CUDA kernel with shared memory
dim3 blockDim(blockDimX, blockDimY);
dim3 gridDim(gridDimX, gridDimY, gridDimZ);
int sharedMemorySize = kernelSize * kernelSize * sizeof(float);
```

```
// start timing
cudaEventRecord(start);
convolutionKernel<<<gridDim, blockDim, sharedMemorySize>>>(d_input, d_output, width, height, d_kernel, kernelSize);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
```

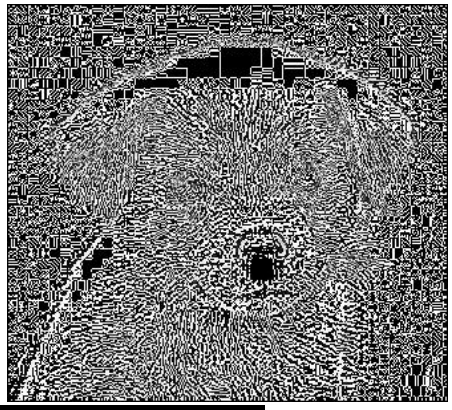
- Kernel call with shared memory. It gives kernel size as 3rd parameter. Other are same with previous one.

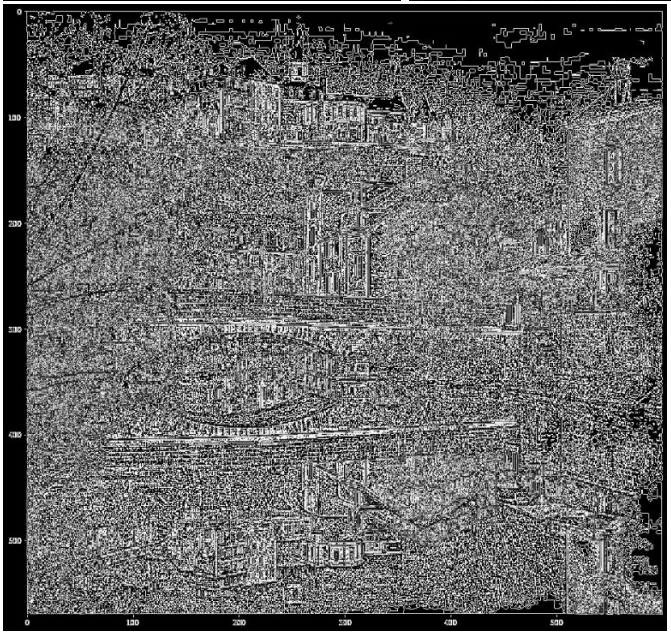
Inputs:



Outputs for 3x3 Kernel:

```
float kernel[9] = {
    0, 1, 0,
    1, -4, 1,
    0, 1, 0
};
int kernelSize = 3;
```





## 4- Result Analysis

- I tested with different grid and block configurations for shared and non-shared implementation.

```
width: 933
height: 882
```
Image width and height

# A. Shared vs Non-Shared

**Every 1-pixel row of the image is block (width x 1)**

```
number of threads in each block: 933
total number of blocks: 882
total number of threads: 822906
```

Shared Kernel (3x3): `average time taken: 0.110298 ms`

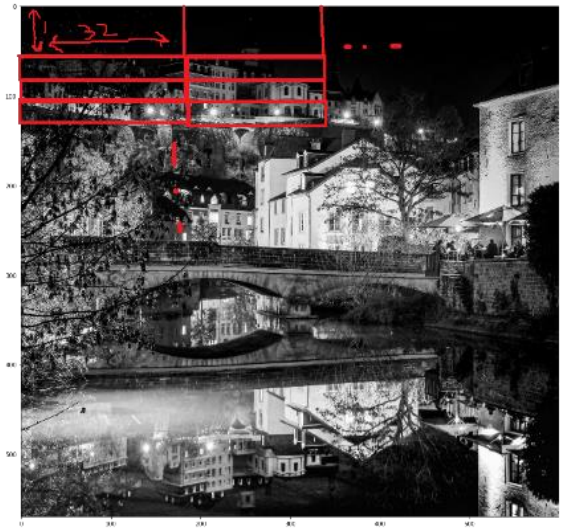
Global Kernel (3x3): `average time taken: 0.11048 ms`

Shared Kernel (5x5): `average time taken: 0.195282 ms`

Global Kernel (5x5): `average time taken: 0.215834 ms`

- Shared kernel is a little bit faster than global kernel. Difference is increasing if kernel size increases.

**Every 1-pixel row of the image is block (32 x 1)**




```
number of threads in each block: 32
total number of blocks: 26460
total number of threads: 846720
```

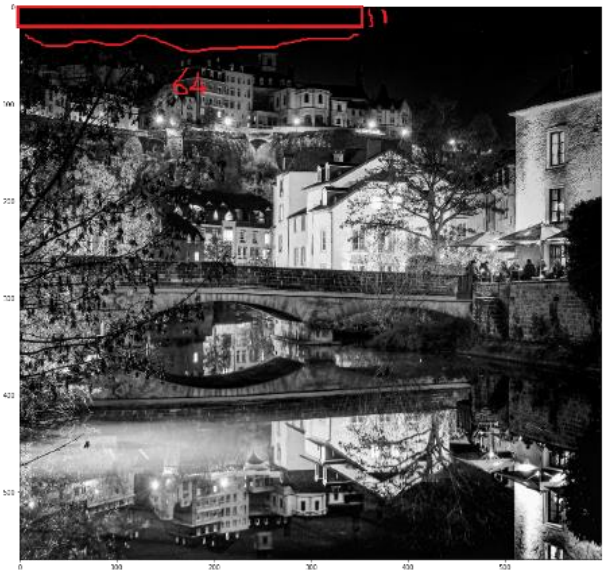
Shared Kernel (3x3): `time taken: 0.16608 ms`

Global Kernel (3x3): `time taken: 0.16944 ms`

Shared Kernel (5x5): `time taken: 0.252704 ms`

Global Kernel (5x5): `time taken: 0.269312 ms`

**Every 1-pixel row of the image is block (64 x 1)**




```
number of threads in each block: 64
total number of blocks: 13230
total number of threads: 846720
```

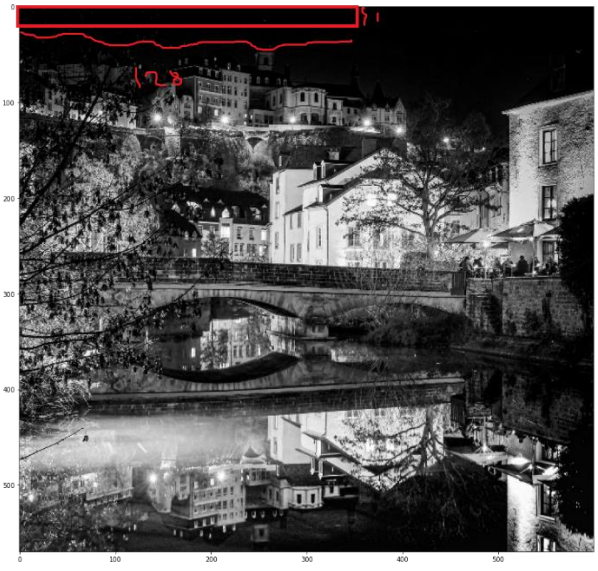
Shared Kernel (3x3): `time taken: 0.097312 ms`

Global Kernel (3x3): `time taken: 0.098304 ms`

Shared Kernel (5x5): `time taken: 0.172032 ms`

Global Kernel (5x5): `time taken: 0.173824 ms`

**Every 1-pixel row of the image is block (128 x 1)**




```
number of threads in each block: 128
total number of blocks: 7056
total number of threads: 903168
```

Shared Kernel (3x3): `time taken: 0.096416 ms`

Global Kernel (3x3): `time taken: 0.09712 ms`

Shared Kernel (5x5): `time taken: 0.175072 ms`

Global Kernel (5x5): `time taken: 0.19456 ms`

**Every 1-pixel row of the image is block (256 x 1)**

```
number of threads in each block: 256
total number of blocks: 3528
total number of threads: 903168
```

Shared Kernel (3x3): `time taken: 0.103872 ms`

Global Kernel (3x3): `time taken: 0.102432 ms`

Shared Kernel (5x5): `time taken: 0.19696 ms`

Global Kernel (5x5): `time taken: 0.184256 ms`

**Every 1-pixel row of the image is block (512 x 1)**

```
number of threads in each block: 512
total number of blocks: 1764
total number of threads: 903168
```

Shared Kernel (3x3): `time taken: 0.116736 ms`

Global Kernel (3x3): `time taken: 0.104448 ms`

Shared Kernel (5x5): `time taken: 0.182496 ms`

Global Kernel (5x5): `time taken: 0.18864 ms`

**Every 1-pixel row of the image is block (1024 x 1)**

```
number of threads in each block: 1024
total number of blocks: 882
total number of threads: 903168
```

Shared Kernel (3x3): `time taken: 0.110592 ms`

Global Kernel (3x3): `time taken: 0.110784 ms`

Shared Kernel (5x5): `time taken: 0.19456 ms`

Global Kernel (5x5): `time taken: 0.200832 ms`

**Every 1-pixel row of the image is block (8 x 8)**


```
number of threads in each block: 64
total number of blocks: 12987
total number of threads: 831168
```

Shared Kernel (3x3): `time taken: 0.134656 ms`

Global Kernel (3x3): `time taken: 0.13312 ms`

Shared Kernel (5x5): `time taken: 0.298144 ms`

Global Kernel (5x5): `time taken: 0.288544 ms`

**Every 1-pixel row of the image is block (16 x 16)**

```
number of threads in each block: 256
total number of blocks: 3304
total number of threads: 845824
```

Shared Kernel (3x3): `time taken: 0.115008 ms`

Global Kernel (3x3): `time taken: 0.11136 ms`

Shared Kernel (5x5): `time taken: 0.229184 ms`

Global Kernel (5x5): `time taken: 0.234912 ms`

**Every 1-pixel row of the image is block (32 x 32)**

```
number of threads in each block: 1024
total number of blocks: 840
total number of threads: 860160
```

Shared Kernel (3x3): `time taken: 0.116448 ms`

Global Kernel (3x3): `time taken: 0.118976 ms`

Shared Kernel (5x5): `time taken: 0.187968 ms`

Global Kernel (5x5): `time taken: 0.192512 ms`

**Every 1-pixel row of the image is block (width/2 x 1)**

```
number of threads in each block: 466
total number of blocks: 2646
total number of threads: 1233036
```

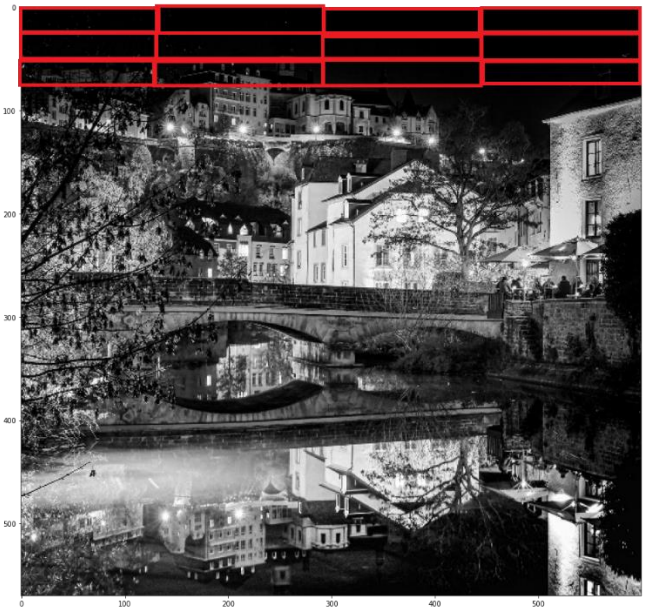
Shared Kernel (3x3): `time taken: 0.11056 ms`

Global Kernel (3x3): `time taken: 0.108544 ms`

Shared Kernel (5x5): `time taken: 0.190464 ms`

Global Kernel (5x5): `time taken: 0.200704 ms`

**Every 1-pixel row of the image is block (width/4 x 1)**



```
number of threads in each block: 233
total number of blocks: 4410
total number of threads: 1027530
```

Shared Kernel (3x3): `time taken: 0.106272 ms`

Global Kernel (3x3): `time taken: 0.10432 ms`

Shared Kernel (5x5): `time taken: 0.190464 ms`

Global Kernel (5x5): `time taken: 0.191904 ms`

**Every 1-pixel row of the image is block (width/2 x 2)**


```
number of threads in each block: 932
total number of blocks: 1323
total number of threads: 1233036
```

Shared Kernel (3x3): `time taken: 0.11776 ms`

Global Kernel (3x3): `time taken: 0.114144 ms`

Shared Kernel (5x5): `time taken: 0.202656 ms`

Global Kernel (5x5): `time taken: 0.204736 ms`

**Every 1-pixel row of the image is block (width/4 x 4)**


```
number of threads in each block: 932
total number of blocks: 1105
total number of threads: 1029860
```

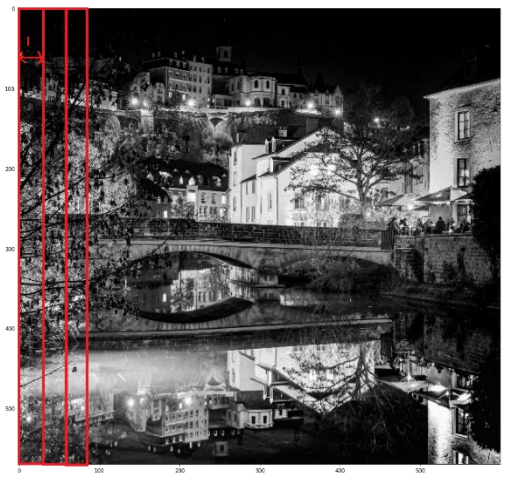
Shared Kernel (3x3): `time taken: 0.11264 ms`

Global Kernel (3x3): `time taken: 0.112128 ms`

Shared Kernel (5x5): `time taken: 0.200032 ms`

Global Kernel (5x5): `time taken: 0.200288 ms`

**Every 1-pixel row of the image is block (1 x height)**



```
number of threads in each block: 882
total number of blocks: 933
total number of threads: 822906
```

Shared Kernel (3x3): `time taken: 0.67952 ms`

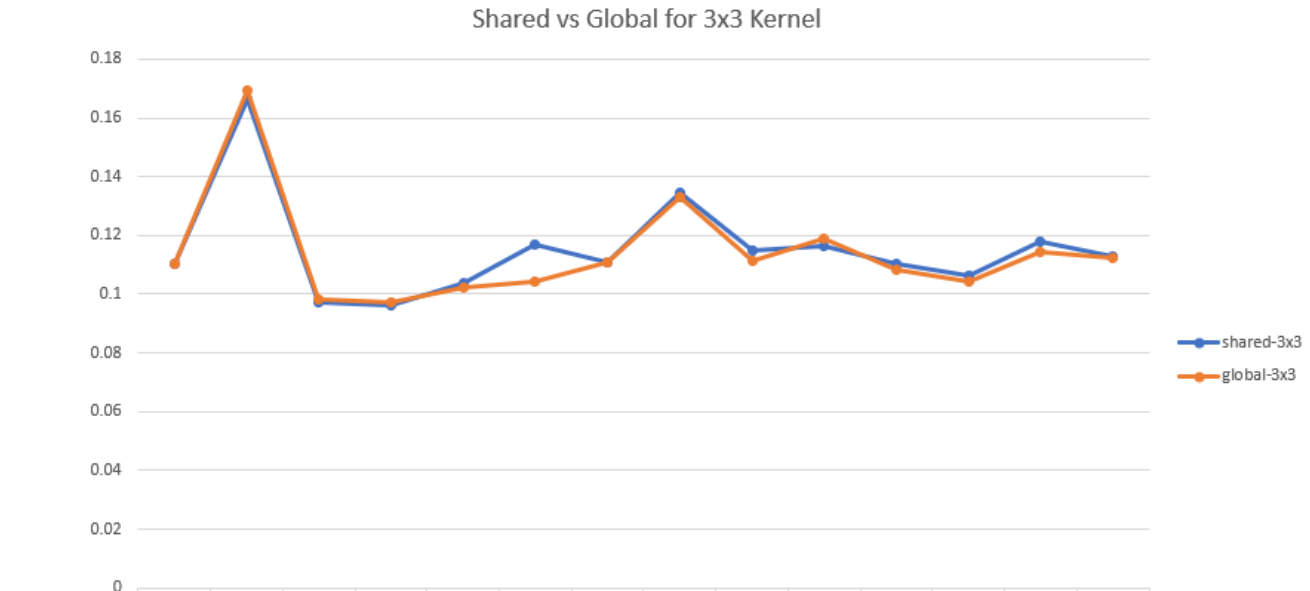
Global Kernel (3x3): `time taken: 0.679936 ms`

Shared Kernel (5x5): `time taken: 1.25955 ms`

Global Kernel (5x5): `time taken: 1.39101 ms`

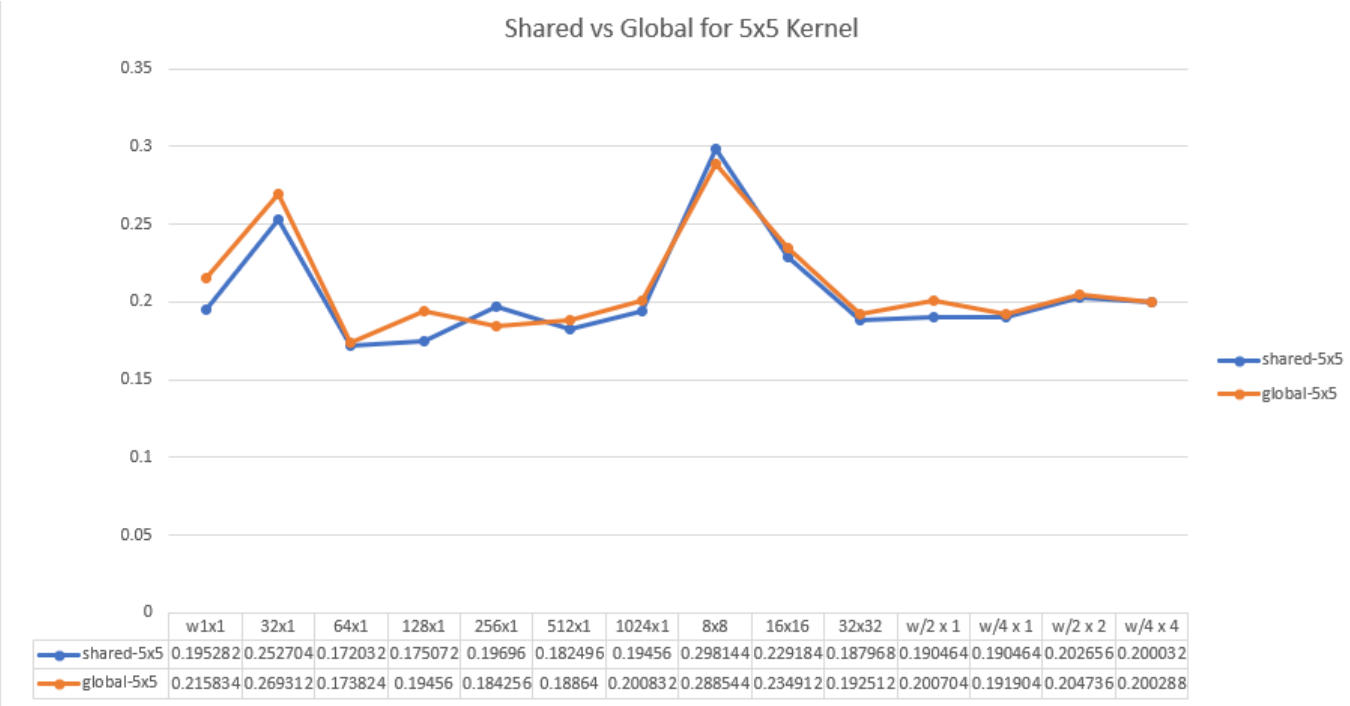
- Performance significantly decreases if we access column-wise. Worst is fully column one.
- It can be easily seen that if we increase size of the shared memory, using it becomes more advantageous than not using.

# Graphs




| | w1x1 | 32x1 | 64x1 | 128x1 | 256x1 | 512x1 | 1024x1 | 8x8 | 16x16 | 32x32 | w/2 x 1 | w/4 x 1 | w/2 x 2 | w/4 x 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| shared-3x3 | 0.110298 | 0.16608 | 0.097312 | 0.096416 | 0.103872 | 0.116736 | 0.110592 | 0.134656 | 0.115008 | 0.116448 | 0.11056 | 0.106272 | 0.11776 | 0.11264 |
| global-3x3 | 0.11048 | 0.16944 | 0.098304 | 0.09712 | 0.102432 | 0.104448 | 0.110784 | 0.13312 | 0.11136 | 0.118976 | 0.108544 | 0.10432 | 0.114144 | 0.112128 |

- For 9 sized shared kernels, performance didn't differ too much. Global and shared memory gave similar results.
- 128x1 fixed grid distribution with shared memory is the best configuration.
- Heightx1 gave the worst result. Accessing column-wise is the worst.




| | w1x1 | 32x1 | 64x1 | 128x1 | 256x1 | 512x1 | 1024x1 | 8x8 | 16x16 | 32x32 | w/2 x 1 | w/4 x 1 | w/2 x 2 | w/4 x 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| shared-5x5 | 0.195282 | 0.252704 | 0.172032 | 0.175072 | 0.19696 | 0.182496 | 0.19456 | 0.298144 | 0.229184 | 0.187968 | 0.190464 | 0.190464 | 0.202656 | 0.200032 |
| global-5x5 | 0.215834 | 0.269312 | 0.173824 | 0.19456 | 0.184256 | 0.18864 | 0.200832 | 0.288544 | 0.234912 | 0.192512 | 0.200704 | 0.191904 | 0.204736 | 0.200288 |

- Shared performance is better for larger kernel. Most of the configuration in shared memory is better than global.
- Best configuration for this kernel is using fixed 64x1.

## B. “dim3” type vs int

### direct

```
int numBlocks = height - kernelSize + 1;
int threadsPerBlock = width - kernelSize + 1;
```

```
// start timing
cudaEventRecord(start);
convolutionKernelShared<<<numBlocks, threadsPerBlock, kernelSize * kernelSize * sizeof(float)>>>(d_input, d_output, width, height, d_kernel, kernelSize);
cudaEventRecord(stop);
```

### using dim3

```
// Configure and launch the CUDA kernel with shared memory
dim3 blockDim(blockDimX, blockDimY);
dim3 gridDim(gridDimX, gridDimY, gridDimZ);
int sharedMemorySize = kernelSize * kernelSize * sizeof(float);
```

```
// start timing
cudaEventRecord(start);
convolutionKernel<<<gridDim, blockDim, sharedMemorySize>>>(d_input, d_output, width, height, d_kernel, kernelSize);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
```

direct: `Average time taken: 0.100262 ms`

dim3: `average time taken: 0.110236 ms`

- Not using dim3 a bit faster for given input.

## Conclusion

- My expectation was with using shared memory, performance must improve, because it caches the kernel data and data is on the chip of each core. Test results shows that it increases the performance, but dependent on the shared data is also important. For example, in 3x3 case data size were small, shared and global gave similar performance. But when we increase kernel size to 5x5, even if it is still small performance is better for shared memory. Because threads need to access more to kernel, and it affected the performance. Our tests show that, if we increase shared memory size, performance difference will larger between shared and non-shared.

- In 3x3 case shared memory performance didn't differ too much because, overhead of launching it is not covered by usage of it. Also filling the shared memory is another overhead. Those overheads are covered when we increase the size of it.