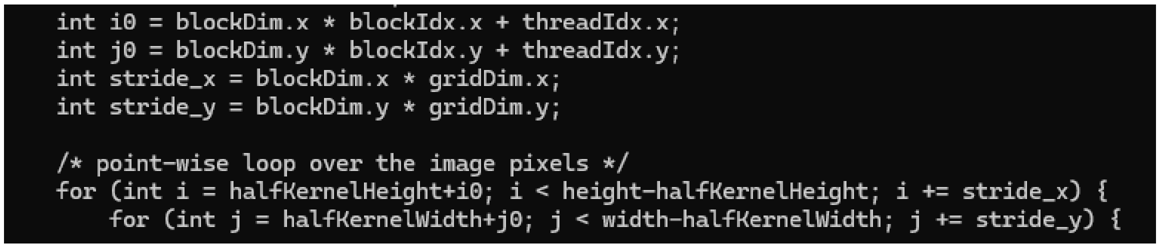
Edge kernel:

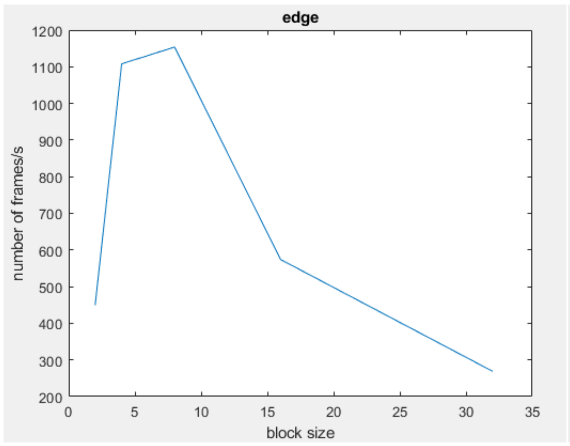
To implement the edge kernel, firstly I declared two indices: i0 = blockDim.x * blockIdx.x + threadIdx.x and j0 = blockDim.y * blockIdx.y + threadIdx.y. These two indices were for cuda to have each thread mapping to corresponding pixels. Then I used nested for loop to iterate through each pixel (i,j), where i,j was from (i0/j0 + half the kernel size) to (height/width of the video – half the kernel size). This is because the boundary condition is to ignore the boundary. Then at each pixel (i,j), I assigned it a new pixel value as (-1*surrounding 8 neighboring pixel values + 8*current pixel value). Such operations were done three times for RGB channels respectively. In all, an edge kernel was implemented.

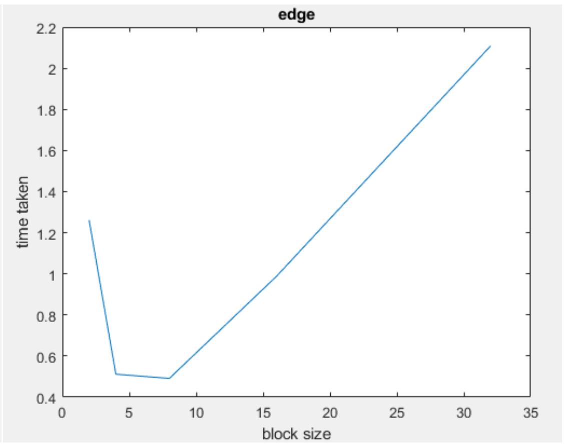
Two strides, stride_x = blockDim.x * gridDim.x and stride_y blockDim.y * gridDim.y are declared. Then the step size of the outer nested for loop over pixels are set to be stride_x and stride_y respectively as shown on the screenshot. Such step size solves the risk that the number of threads was not enough, because if the maximum number of threads is smaller than the pixels, then each thread would be assigned more than one pixel.



```
    int i0 = blockDim.x * blockIdx.x + threadIdx.x;
    int j0 = blockDim.y * blockIdx.y + threadIdx.y;
    int stride_x = blockDim.x * gridDim.x;
    int stride_y = blockDim.y * gridDim.y;

    /* point-wise loop over the image pixels */
    for (int i = halfKernelHeight+i0; i < height-halfKernelHeight; i += stride_x) {
        for (int j = halfKernelWidth+j0; j < width-halfKernelWidth; j += stride_y) {
```



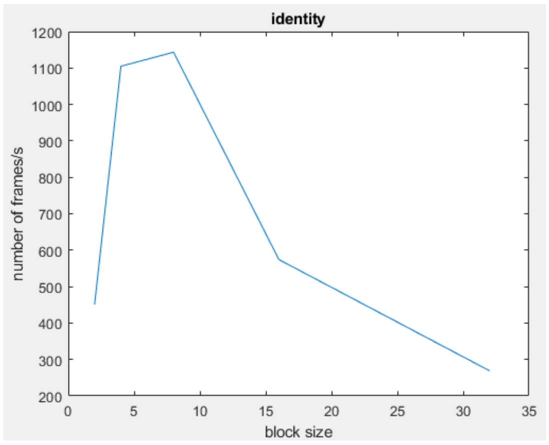


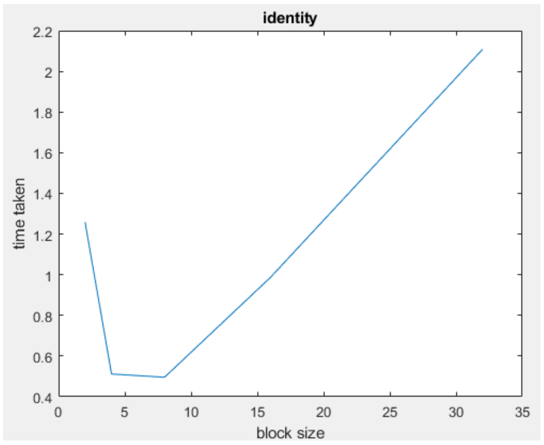
The performance on edge kernel was tested on grid sizes of 2,4,8,16,32 respectively. The shortest execution time is achieved at 8. As the grid size was away from 8, no matter increased or decreased, the time taken became longer and the number of frames per seconds became less. It was possible that if the block size is too big, then there would be overhead on memory shard by all threads inside each block; if the block size is too small, then the grid size will be big and there would be overhead on memory shard by all blocks. Thus, an optimum block size can be achieved by seeking for a balance between block size and grid size.

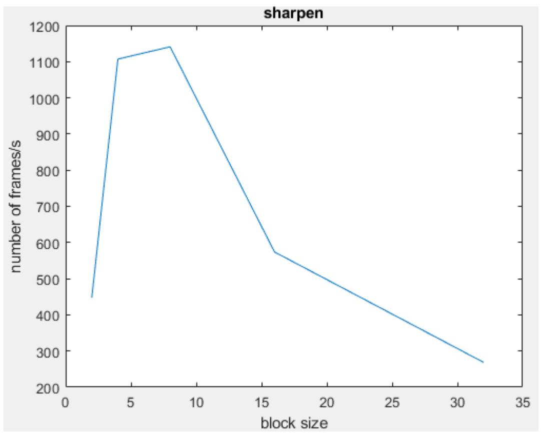
The implementation of the rest three kernels were almost the same as that for the edge kernel. The only differences were that identity(pixel(i,j)) = pixel(i,j); sharpen(pixel(i,j)) = 5*pixel(i,j) – 1* pixel(i-1,j) – 1* pixel(i+1,j) – 1* pixel(i,j-1) – 1* pixel(i,j+1); blur(pixel(i,j)) is on 25 neighboring pixels including the pixel itself.

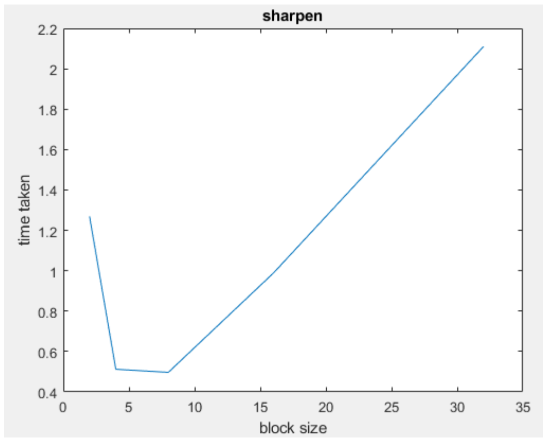
```
float identityKernel[9] = {0.0f,0.0f,0.0f,
                           0.0f,1.0f,0.0f,
                           0.0f,0.0f,0.0f};
float edgeKernel[9] = {-1.0f,-1.0f,-1.0f,
                       -1.0f, 8.0f,-1.0f,
                       -1.0f,-1.0f,-1.0f};
float sharpenKernel[9] = { 0.0f,-1.0f, 0.0f,
                          -1.0f, 5.0f,-1.0f,
                           0.0f,-1.0f, 0.0f};
float blurKernel[25] = {0.004f,0.016f,0.023f,0.016f,0.004f,
                        0.016f,0.064f,0.094f,0.064f,0.016f,
                        0.023f,0.094f,0.141f,0.094f,0.023f,
                        0.016f,0.064f,0.094f,0.064f,0.016f,
                        0.004f,0.016f,0.023f,0.016f,0.004f};
```

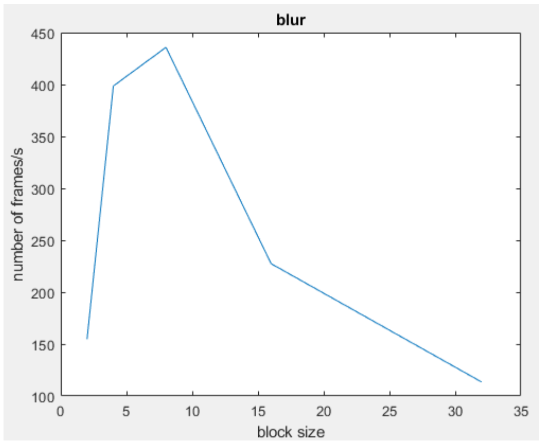
The ways the initial data were distributed were identical, with i0 and j0 declared and step size equaled stride_x and stride_y respectively.

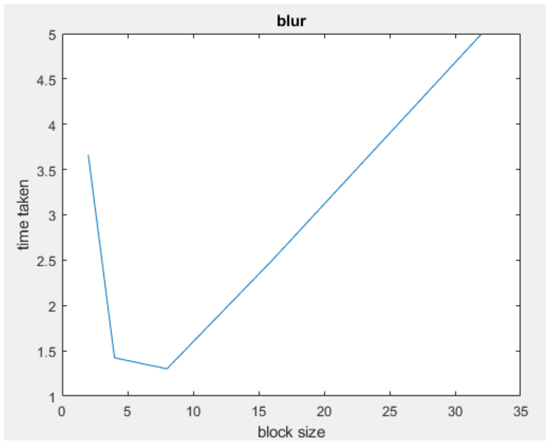












From the graphs above, the trends on all 4 kernels were all the same. Block size of 8 was the optimal value where the number of frames done per second was the largest and the time taken was the shortest. The further from the optimal value, the worse the performance.