Ryan Blair, Seth Barrios, Lenoy Avidan, Jonathan Banuelos

Blurred and original images are in the zip file

Run times:
GPU ~1.1 seconds
CUDA ~300 microseconds

Index of code, including horizontal/vertical kernels and calling of kernels:

```
__global__
void conv1D(uchar4* const rgbaImage,uchar4* const greyImage,int numRows, int numCols)
{

        //TODO Fill in the kernel to blur original image
        // Original Image is an array, each element of the array has 4 components .z -> R (red); .y -> G (Green) ; .x -> B (blue); .w ->
 A (alpha, you can ignore this one)
        //so you can read one input pixel like this:
        //B = rgbaImage[currow * numCols + curcol].x*M_d[curcolkernel];
        //G = rgbaImage[currow * numCols + curcol].y*M_d[curcolkernel];
        //R = rgbaImage[currow * numCols + curcol].z*M_d[curcolkernel];
        int pix_x = (blockIdx.x * blockDim.x) + threadIdx.x;
        int pix_y = (blockIdx.y * blockDim.y) + threadIdx.y;
        int cur_x;

        float blurValx = 0;
        float blurValy = 0;
        float blurValz = 0;
        float blurValw = 1;

        if (pix_x >= 0 && pix_x < numCols && pix_y >= 0 && pix_y < numRows) {
                for (int i = -2; i <= 2; i++) {
                        cur_x = pix_x + i;
                        if (cur_x >= 0 && cur_x < numCols) {
                                blurValx += rgbaImage[pix_y * numCols + cur_x].x * M_d[i+2];
                                blurValy += rgbaImage[pix_y * numCols + cur_x].y * M_d[i+2];
                                blurValz += rgbaImage[pix_y * numCols + cur_x].z * M_d[i+2];
                        }
                }
                greyImage[pix_y * numCols + pix_x].x = (int)blurValx;
                greyImage[pix_y * numCols + pix_x].y = (int)blurValy;
                greyImage[pix_y * numCols + pix_x].z = (int)blurValz;
                greyImage[pix_y * numCols + pix_x].w = (int)blurValw;
        }
}
```

```
__global__
void conv1DCol(uchar4* const rgbaImage,uchar4* const greyImage,int numRows, int numCols)
{

        //TODO Fill in the kernel to blur original image
        // Original Image is an array, each element of the array has 4 components .z -> R (red); .y -> G (Green) ; .x -> B (blue); .w ->
 A (alpha, you can ignore this one)
        //so you can read one input pixel like this:
        //B = rgbaImage[currow * numCols + curcol].x*M_d[curcolkernel];
        //G = rgbaImage[currow * numCols + curcol].y*M_d[curcolkernel];
        //R = rgbaImage[currow * numCols + curcol].z*M_d[curcolkernel];
        int pix_x = (blockIdx.x * blockDim.x) + threadIdx.x;
        int pix_y = (blockIdx.y * blockDim.y) + threadIdx.y;
        int cur_y;

        float blurValx = 0;
        float blurValy = 0;
        float blurValz = 0;
        float blurValw = 1;
        if (pix_x >= 0 && pix_x < numCols && pix_y >= 0 && pix_y < numRows) {
                for (int i = -2; i <= 2; i++) {
                        cur_y = pix_y + i;
                        if (cur_y >= 0 && cur_y<numRows) {
                                blurValx += rgbaImage[cur_y * numCols + pix_x].x * M_d[i + 2];
                                blurValy += rgbaImage[cur_y * numCols + pix_x].y * M_d[i + 2];
                                blurValz += rgbaImage[cur_y * numCols + pix_x].z * M_d[i + 2];
                        }
                }
                greyImage[pix_y * numCols + pix_x].x = (int)blurValx;
                greyImage[pix_y * numCols + pix_x].y = (int)blurValy;
                greyImage[pix_y * numCols + pix_x].z = (int)blurValz;
                greyImage[pix_y * numCols + pix_x].w = (int)blurValw;
        }
}
void your_rgba_to_greyscale(const uchar4 * const h_rgbaImage,
                                                uchar4 * d_rgbaImage,
                                                uchar4*  d_greyImage,
                                                size_t numRows,
                                                size_t numCols)
{
        float M_h[BLUR_SIZE]={0.0625, 0.25, 0.375, 0.25, 0.0625};  //change this to whatever 1D filter you are using
        cudaMemcpyToSymbol(M_d,M_h, BLUR_SIZE*sizeof(float)); //allocates/copy to Constant Memory on the GPU
        //temp image
        uchar4 *d_greyImageTemp;
        cudaMalloc((void **)&d_greyImageTemp, sizeof(uchar4) * numRows*numCols);
        cudaMemset(d_greyImageTemp, 0, numRows*numCols * sizeof(uchar4)); //make sure no memory is left laying around

        int threadSize=16;
        int gridSizeX=(numCols + threadSize - 1)/threadSize;
        int gridSizeY=(numRows + threadSize - 1)/threadSize;
        const dim3 blockSize(threadSize, threadSize, 1);
        const dim3 gridSize(gridSizeX, gridSizeY, 1);
        for (int i=0;i<30;i++){
                //row
                conv1D<<<gridSize, blockSize>>>(d_rgbaImage,d_greyImageTemp,numRows,numCols);
                cudaDeviceSynchronize();
                //col
                conv1DCol<<<gridSize, blockSize>>>(d_greyImageTemp,d_greyImage,numRows,numCols);
                cudaDeviceSynchronize();

                //swap
                d_rgbaImage=d_greyImage;
        }
}
```