IT477 Lab Assignment 4

# CUDA Image Filtering (Median Filter)

Monil Soni, 201601049
Bhavik Mehta, 201601223

October 2, 2019

# Table of Contents

# Context

### Brief description of the problem

To apply median filter on a given image. It requires surrounding pixel values and picks the median of these pixel values and assign it to current pixel value.

### Complexity of the algorithm

The serial code will operate with the time complexity of $O(n^2 * m * \log m)$ since we have to iterate across n rows and n columns, also we have to sort the median mask of size m.

### Optimization strategy

When we observe that every pixel operation is independent of the other pixels across the loops – in other words, there is no loop carried dependency and loop independent dependency. Hence, parallelization can be done on both the loops by removing the loops and operating on one pixel per thread. Further we observed that global memory access plays a huge role in determining the throughput and latency. So we took tiling approach to use shared memory of GPU and thus reduce global memory acesses and improve CGMA ratio.

### Problems faced in parallelization and possible solutions

- When operating on images of general size, control divergence becomes an issue because the image dimension along a particular axis may not be a multiple of the block dimension on that axis. Hence, the execution of a single warp will split into two phases increasing the time. To solve this issue, we need to alter the block size accordingly.
- However, while doing that we need to also check with the constraints of threads per block. blocks per SM, and threads per SM so that number of blocks does not increase beyond the point where it causes increase in number of execution phases which again will increase the total execution time.
- It also required extensive knowledge of tiling and how shared memory is created. We also took care of boundary conditions by paddding the shared memory matrix with nearby values.

# Code

## Serial Code

```
void filterImage(PPMImage *in, PPMImage *out, int height, int width) {
    int mask_size = (2 * N + 1) * (2 * N + 1);

    int mask[mask_size];
    for (int x = 0; x < width; ++x) {
        for (int y = 0; y < height; ++y) {
            // make the mask array
            for (int i = -1 * N, itr = 0; i <= 1 * N; ++i) {
                for (int j = -1 * N; j <= 1 * N; ++j) {
                    int p = min(max(0, x + i), width - 1);
                    int q = min(max(0, y + j), height - 1);

                    mask[itr] = in->data[p * width + q].red;
                    mask[itr] =
                        (mask[itr] << 8) + in->data[p * width + q].green;
                    mask[itr] =
                        (mask[itr] << 8) + in->data[p * width + q].blue;
                    itr++;
                }
            }
            // sort the array
            qsort(mask, mask_size, sizeof(int), cmpfunc);
            // pick the median and assign to output array
            out->data[x * width + y].red =
                (mask[(mask_size + 1) / 2] >> 16) & 0xFF;
            out->data[x * width + y].green =
                (mask[(mask_size + 1) / 2] >> 8) & 0xFF;
            out->data[x * width + y].blue =
                (mask[(mask_size + 1) / 2]) & 0xFF;
        }
    }
}
```

Parallel Code

```
__global__ void medianFilter(unsigned char * filterImg, unsigned char *rgbImage,
int width, int height) {

    int column = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = row * width + column;

    if(column<width && row<height){
        int mask[MASK_SIZE];
        for (int i = -1 * N, itr = 0; i <= 1 * N; ++i) {
            for (int j = -1 * N; j <= 1 * N; ++j) {
                int p = min(max(0, column + i), width - 1); //col
                int q = min(max(0, row + j), height - 1); //row

                int currOffset = q * width + p;

            mask[itr] = rgbImage[CHANNELS*currOffset]; //red
            mask[itr] =
                (mask[itr] << 8) + rgbImage[CHANNELS*currOffset+1];
            mask[itr] =
                (mask[itr] << 8) + rgbImage[CHANNELS*currOffset+2];
                itr++;
            }
    }

    for(int i = 0; i < MASK_SIZE ; i++){
        for(int j = 0 ; j < MASK_SIZE ; j++){
            if(mask[i]>mask[j]){
                int temp = mask[i];
                mask[i] = mask[j];
                mask[j] = temp;
            }
        }
    }
    filterImg[CHANNELS*offset] = (mask[(MASK_SIZE + 1) / 2] >> 16) & 0xFF;
    filterImg[CHANNELS*offset+1] = (mask[(MASK_SIZE + 1) / 2] >> 8) & 0xFF;
    filterImg[CHANNELS*offset+2] = (mask[(MASK_SIZE + 1) / 2]) & 0xFF;
    }
}
```

### Tiling Code

```
__global__ void medianFilterShared(unsigned char * filterImg, unsigned char
*rgbImage, int width, int height) {

    __shared__ unsigned int sharedMemory[TILE_SIZE + 2] [TILE_SIZE + 2];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int column = blockIdx.x * blockDim.x + threadIdx.x;

    int offset = row * width + column;


    sharedMemory[threadIdx.y+1][threadIdx.x+1] = rgbImage[CHANNELS*offset];

    sharedMemory[threadIdx.y+1][threadIdx.x+1] =
        (sharedMemory[threadIdx.y+1][threadIdx.x+1]<<8) + rgbImage[CHANNELS*offset +
1];

    sharedMemory[threadIdx.y+1][threadIdx.x+1] =
        (sharedMemory[threadIdx.y+1][threadIdx.x+1]<<8) + rgbImage[CHANNELS*offset +
2];



    //top left
    if(threadIdx.y == 0 && threadIdx.x == 0){
        sharedMemory[threadIdx.y][threadIdx.x] = sharedMemory[threadIdx.y+1]
[threadIdx.x+1];
    }

    //first row
    if(threadIdx.y==0){
        sharedMemory[threadIdx.y][threadIdx.x + 1] = sharedMemory[threadIdx.y+1]
[threadIdx.x + 1];
    }

    //bottom left
    if(threadIdx.y == blockDim.y-1 && threadIdx.x == 0){
        sharedMemory[threadIdx.y+2][threadIdx.x] = sharedMemory[threadIdx.y + 1]
[threadIdx.x + 1];
    }
```

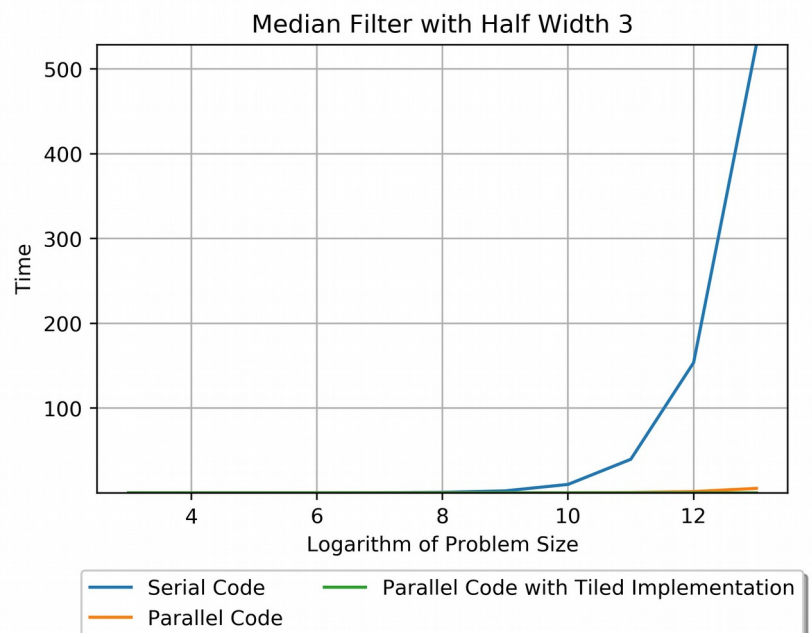
```
    //last row
    if(threadIdx.y==blockDim.y-1){
        sharedMemory[threadIdx.y+2][threadIdx.x+1] = sharedMemory[threadIdx.y+1]
[threadIdx.x+1];
    }

    //top right
    if(threadIdx.y == 0 && threadIdx.x == blockDim.x-1){
        sharedMemory[threadIdx.y][threadIdx.x + 2] = sharedMemory[threadIdx.y+1]
[threadIdx.x+1];
    }
    //first column
    if(threadIdx.x==0){
        sharedMemory[threadIdx.y + 1][threadIdx.x] = sharedMemory[threadIdx.y+1]
[threadIdx.x+1];
    }

    //bottom right
    if(threadIdx.y == blockDim.y+1 && threadIdx.x == blockDim.x+1){
        sharedMemory[threadIdx.y + 2][threadIdx.x+2] = sharedMemory[threadIdx.y+1]
[threadIdx.x+1];
    }

    //last column
    if(threadIdx.x==blockDim.x-1){
        sharedMemory[threadIdx.y+1][threadIdx.x+2] = sharedMemory[threadIdx.y+1]
[threadIdx.x+1];
    }

    __syncthreads();

    int mask[MASK_SIZE];

    for(int i = 0,itr=0; i< N; i++){
        for(int j = 0; j<N;j++){
            mask[itr] = sharedMemory[threadIdx.y+i][threadIdx.x+j];
            itr++;
        }
    }

    //sort
    for(int i=0;i<MASK_SIZE;i++){
        for(int j=0;j<MASK_SIZE;j++){
```

```
            if(mask[i]>mask[j]){
                int temp=mask[i];
                mask[i]=mask[j];
                mask[j]=temp;
            }
        }
    }

    filterImg[CHANNELS*offset] =
        (mask[(MASK_SIZE) / 2] >> 16) & 0xFF;
    filterImg[CHANNELS*offset+1] =
        (mask[(MASK_SIZE) / 2] >> 8) & 0xFF;
    filterImg[CHANNELS*offset+2] =
        (mask[(MASK_SIZE) / 2]) & 0xFF;

}
```

# Questions

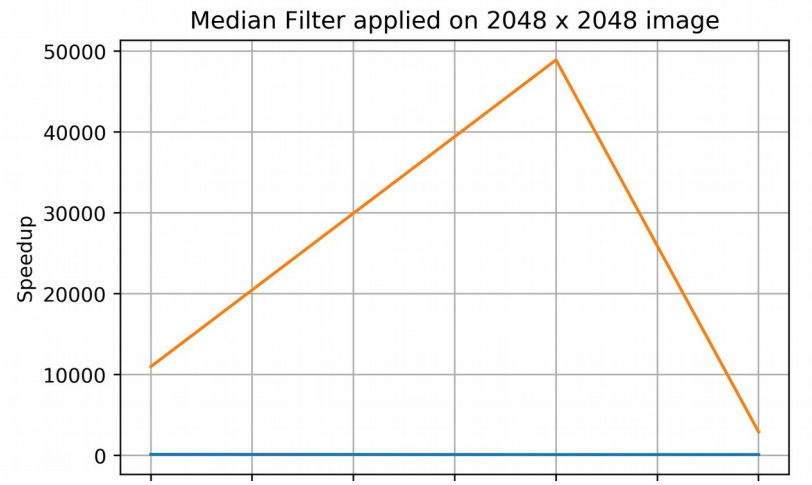
## Q-1) Obtaining a image-size vs run-time graph

Figure



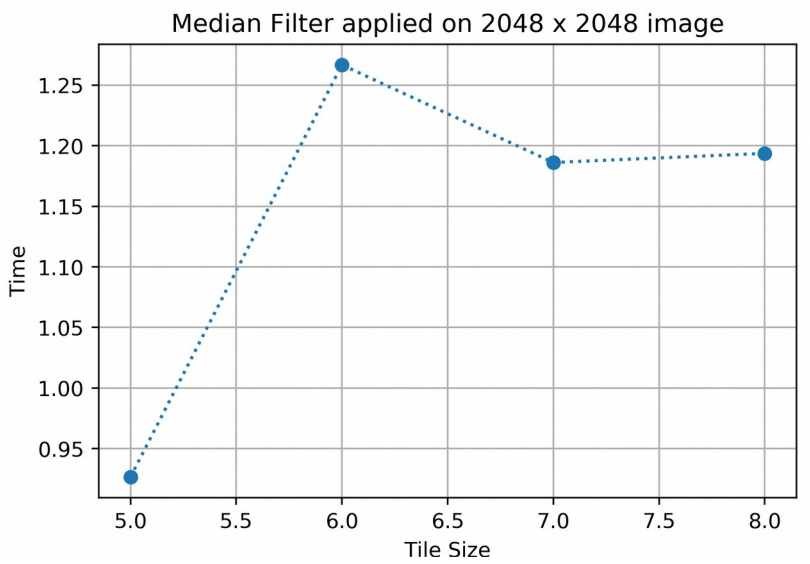

## Q-2) Obtaining a speedup vs problem size graph (for a fixed big image)

Figure



## Q-3) Investigate the effect of tile size on tiled implementation (for a fixed big image)

Figure




As we can observe from above graph that increasing tile size there is no significant increase in run-time of tile code.

## Q-4) Define your own investigations and summarize the significant insights.

- Increasing half width of the filter make image more blur. This is because more approximation is achieved to every pixel and thus it blurs the image.
- The run time of parallel code significantly improves over serial code and tiled code significantly improves parallel code. Thus, it is very beneficial to use shared memory logic whenever there is a scope of shared memory over global memory.
- Increasing half width also increases run time of all types of code - serial, parallel and tiled.