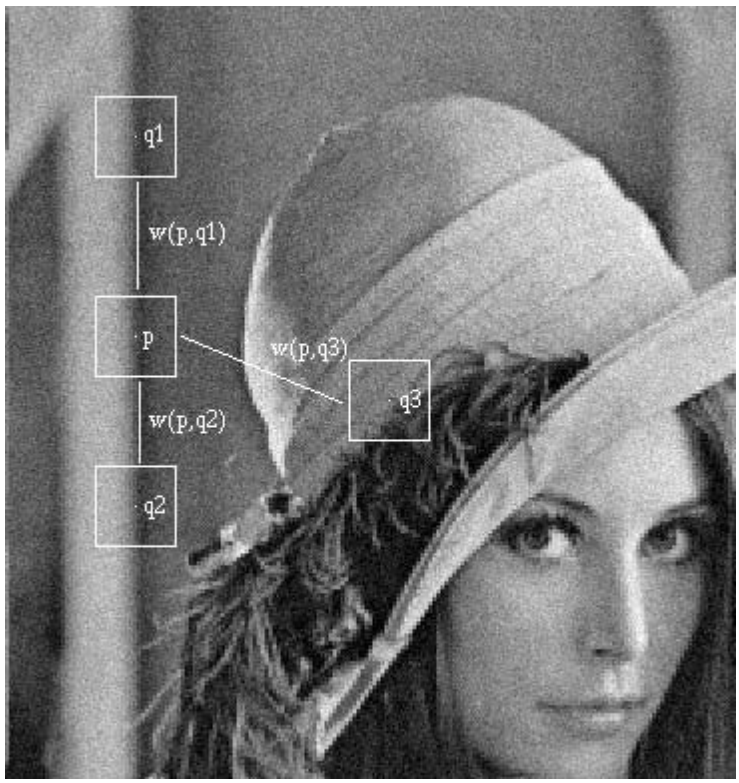
Parallel and Distributed Computer Systems - Task 3

Implementation of the Non-Local Means Denoising Algorithm in Cuda



Evangelou Alexandros
alexande@auth.gr
AEM: 8309

Mamalakis Evangelos
mamalakis@auth.gr
AEM: 8191

Introduction

We review some alternatives to reduce the execution time of the Non-Local Means image filter and present a CUDA-based implementation of it for GPUs, comparing it's performance with the CPU implementation.The goal of image denoising methods is to recover the original image from a noisy measurement.

The Non Local Means Algorithm

```
for each pixel in image:
        partialW = 0;
        for each other_pixel in image:
                partialW = 0;
                        for each npixel in neighbourhood:
                                partialW += gaussDistW(npixel) * (other_pixel-pixel)
                partialW = expf((-partialW / sigma));
                weightSum += partialW;
                fSum += partialW * other_pixel
        denoised_pixel = fSum / weightSum
```

GPU

Registers

We have a kernel that uses 32 registers. With a 16x16 block grid, we can run (10*16*16 = 2560. SM can hold 8192/2560 =) 3 blocks, meaning we will use 3*16*16 = 768 threads, which is within limits.

Blocks

An SM can have up to 8 blocks: 16x16 16*16 = 256 threads per block, 1024/256 = 4 blocks. An SM can take all blocks, then all 1024 threads will be active and achieve full capacity unless another resource overrule.

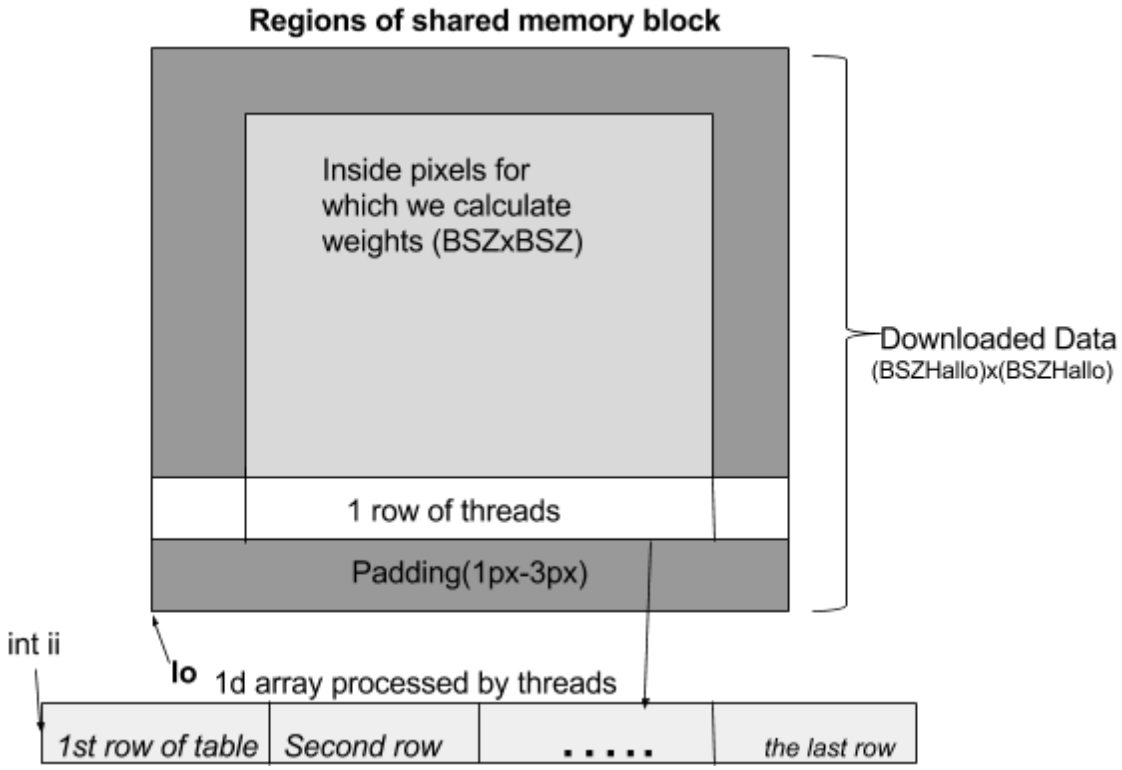
Shared Memory

- Small (48kB per SM)
- Fast (~4 cycles): On-chip
- Private to each block - Allows thread communication

We will use (16+6)*(16+6) * 4 = 1936KB of shared memory for multi = 1.

Our Implementation

We split the image in 16*16 pixel blocks, and each pixel is assigned to a thread. We then have to download each pixel to shared memory, including the padding of the block, so we download BSZ_HALO = (16+NeighbourhoodSize-1)^2 pixels as shown in the picture below. This way we can calculate the weightSums for the pixels in the block but we still need to calculate the weights for the pixels assigned to other *foreign* blocks.
In addition to the pixels assigned to the block, we allocate shared memory for the pixel of another block, the *foreignBlockImg[][]* array. In that array, we download the pixels assigned to other blocks -one block at a time- in order to calculate the weights of our block pixels with the foreign block pixels.



Our cuda kernel implements four functions:

__global__ void nlm()

Arguments

The global nlm function takes as arguments from MATLAB:

1. the (N+NeighSize-1)x(N+NeighSize-1) 1D noisy image, with outside padding applied in matlab,

2. the NxN 1D output array,
3. the size of the image and
4. the sigma value of the gaussian kernel.

Furthermore we have a (NeighSize x NeighSize) 1D array allocated in constant memory an array with the gaussian weights that correspond to the distance of a cell in the neighborhood from the center.

Procedure

First, we allocate 2 shared arrays, one with fixed values, the pixels of the block, and another where we successively download the pixels of all the other blocks, to calculate the weights with them.

The values of the partial weights with each block are stored in fSum, and weightSum registers. In the end we normalize the new pixel value(fSum) by dividing with the weightSum. We also make sure to replace the weight values equal to one with the maxWeight found.

The I0 index

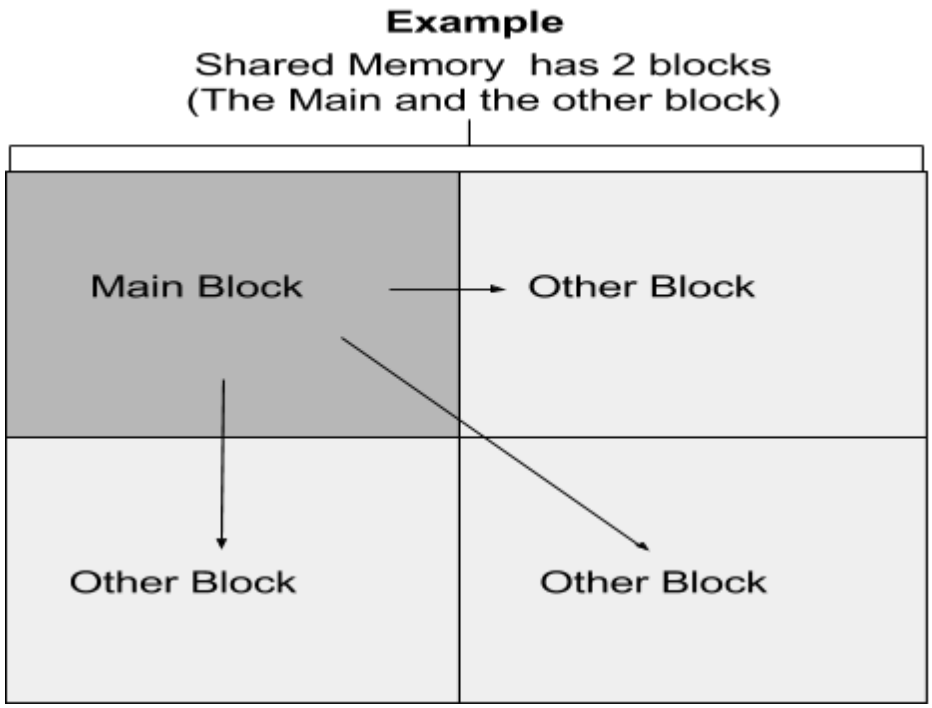
As shown in the figure above, this index points to the first pixel of each block, and is used to determine which block to download each time. The borders of the block are calculated relative to I0, using the block size value.

__device__ __forceinline__ void downloadAndCalculate()

Given a *blockImg[][]* in shared memory, with the pixels for which we are calculating the weights, this functions downloads another block(*foreignBlockImg*) from global memory and calculates the corresponding weights. *__syncthreads* is called before we start calculating to make sure that all pixels are downloaded.

__device__ __forceinline__ void getSharedBlock()

As we mentioned above, this function downloads the pixels assigned to a block from global memory to shared memory. The problem we had to solve was that, because we want to downloading the neighbourhood padding pixels that we need for the weight calculations, we have less threads than pixels. To cope with this, we assign an 1D index to each thread *ii* = (1…threadnum). Then we divide this index with the uter block dimensions(*BSZ_HL = BSZ + padding*) to get the indexes in the block of the pixel that will be downloaded*(I, J)*. Using these indexes, and the argument *I0* which is the index of the pixel in the global array. After we download the first threadnum pixels, we repeat the same process, this time with *ii*=(threadnum..2*threadnum) to download another batch of pixels, until we reach the end of the block. The inspiration for this function came from the code in p20 of [1].



__device__ __forceinline__ void getWeight()

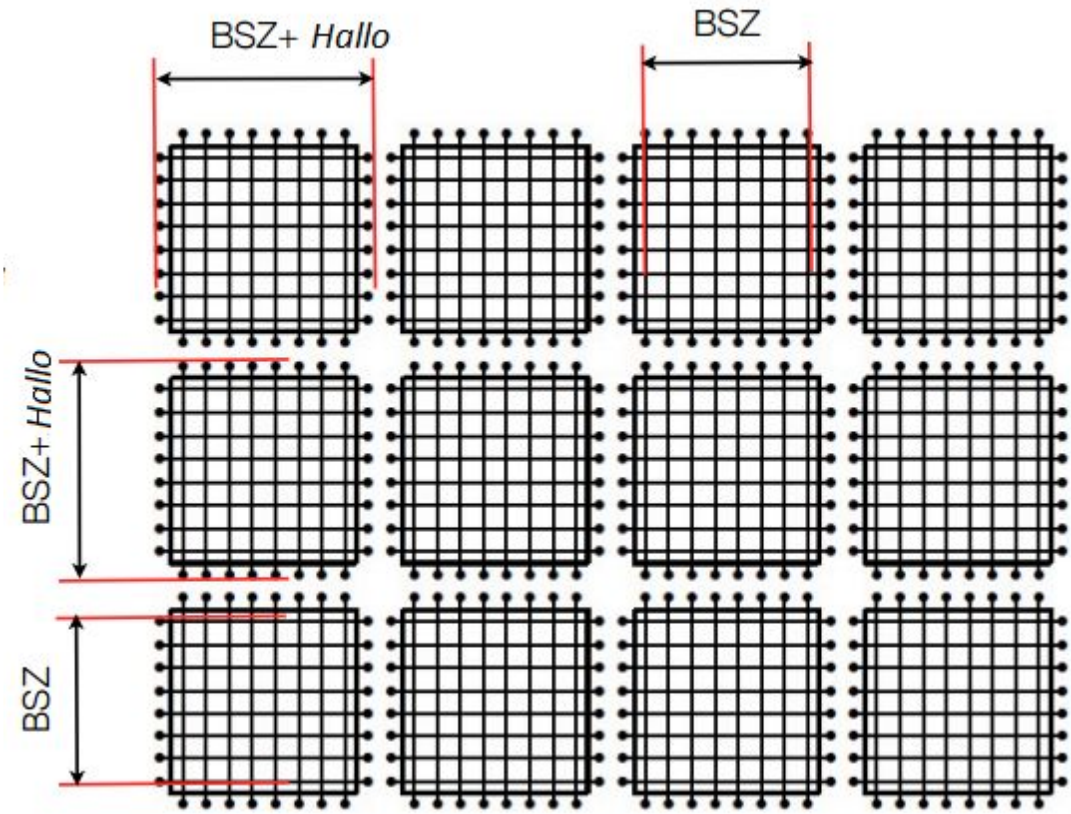
Given a main block and a foreign block, this function computes the partial weights for the pixels in the main block(*blockImg*) with the pixels in the foreign block(*foreignBlockImg*). The two outer for loops, loop through the pixels of the foreign block, and the inner for loops loop through the neighbourhood of the pixel. The results are stored per thread, partial weights in the weightSum register and the partial pixel value in the fSum register. We also keep track of the maximum weight value in maxWeight.

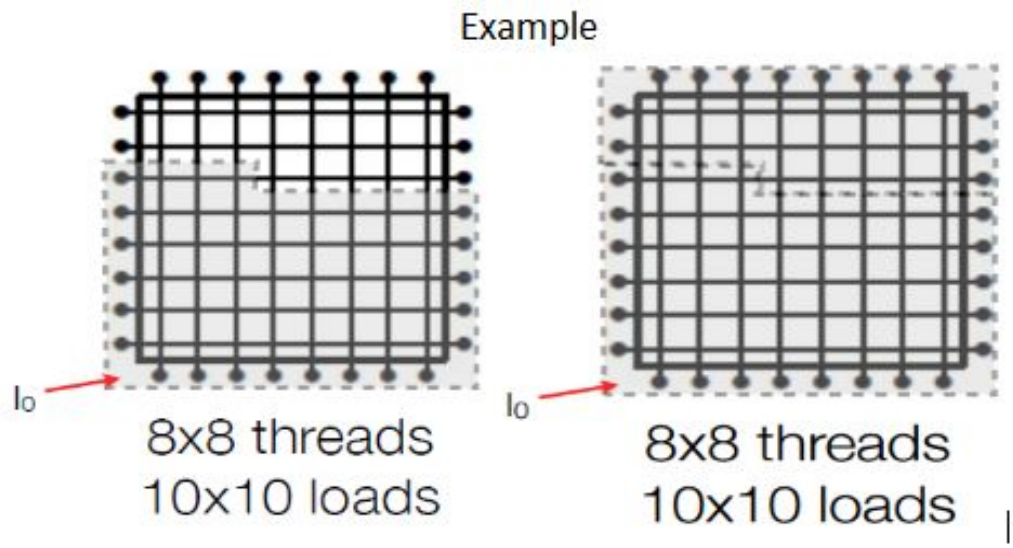
Optimizations

Shared Memory Implementation - The tiling technique

We are padding all the borders of the image depending on the patchSize.
The size of the table become (BSZ + NSZ - 1)x(BSZ + NSZ - 1). To accomplish the downloading of the pixels with less threads we change indexing so to jump in steps of BSZ-(NSZ - 1) instead of BSZ. We'll need N/(BSZ-2) blocks per dimension, instead of N/BSZ.

-Load in two stages
-First we load as many values as threads to shared memory(Some values won't load)
-After we load the remaining values.With this method all threads operate





Multiple pixels assigned to each thread (*multi*)

We experimented with a method to increase the size of the blocks, reducing the total number of blocks and the tranfers from global memory. Each threads calculates and saves more than one pixel every time. We have 256 threads, but each thread can calculate 4x pixels, if *multi* is set to 2. The number of blocks is the reduced by a factor of 4. We didn't manage to make this approach work faster than the normal *multi*=1, probably because we didn't reach full occupancy.

Bank conflicts

We optimized our code to avoid bank conflicts. For nvidia gpus local memory is divided into memory banks. Each bank can only address one dataset at a time, so if a halfwarp tries to load/store data from/to the same bank the access has to be serialized. Because each thread in a warp accesses successive 32bit values in shared memory there are no bank conflicts.

Code Statistics

```
ptxas info    : 99 bytes gmem, 196 bytes cmem[2], 32 bytes cmem[14]
ptxas info    : Compiling entry function '_Z3nlmPKfPfif' for 'sm_20'
ptxas info    : Function properties for _Z3nlmPKfPfif
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 27 registers, 3200 bytes smem, 56 bytes cmem[0], 24 bytes
cmem[16]
```

Image Inputs

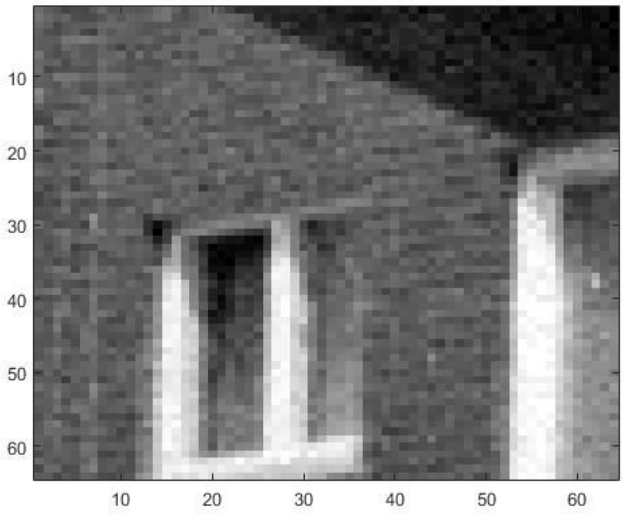
We used the house.mat file to test the result of our algorithm relative tothe given Matlab version. We copied the 64 x 64 input and created 128 x 128 and 256 x 256 files in order to test the algorithm for bigger inputs.

Results and execution time speedup

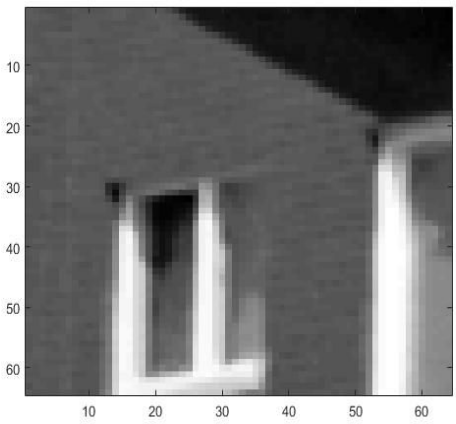
We run the code in GF100 (GeForce GTX 480) NVIDIA GPU card. We tested different input pictures and different neighbourhood sizes and below and we observed an 80x speedup. We present the results below:

64x64 Picture

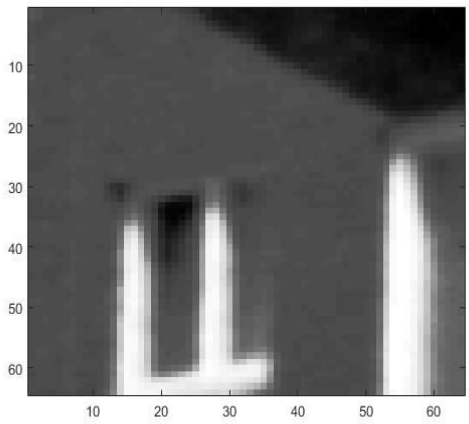
Input noisy image



Matlab cpu denoised image



Gpu denoised image

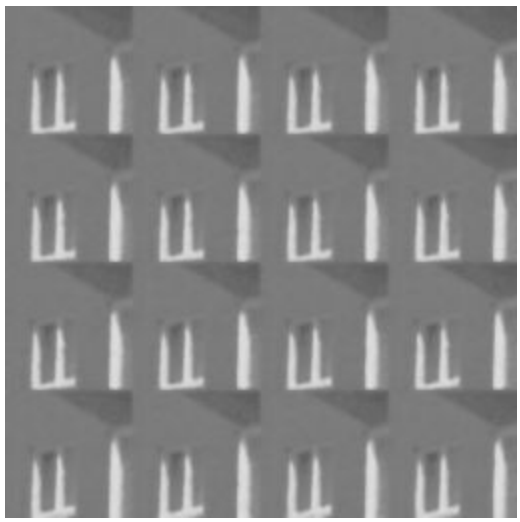


Picture Size	Patch Size	Execution time CPU	Execution time GPU
64x64	3	0.886331	0.005591
	5	0.913451	0.011286
	7	1.256972	0.017301

128x128 Picture

Picture Size	Patch Size	Execution time CPU	Execution time GPU
128x128	3	12.964427	0.045641
	5	14.094522	0.100808
	7	14.955366	0.153870

256x256 Picture

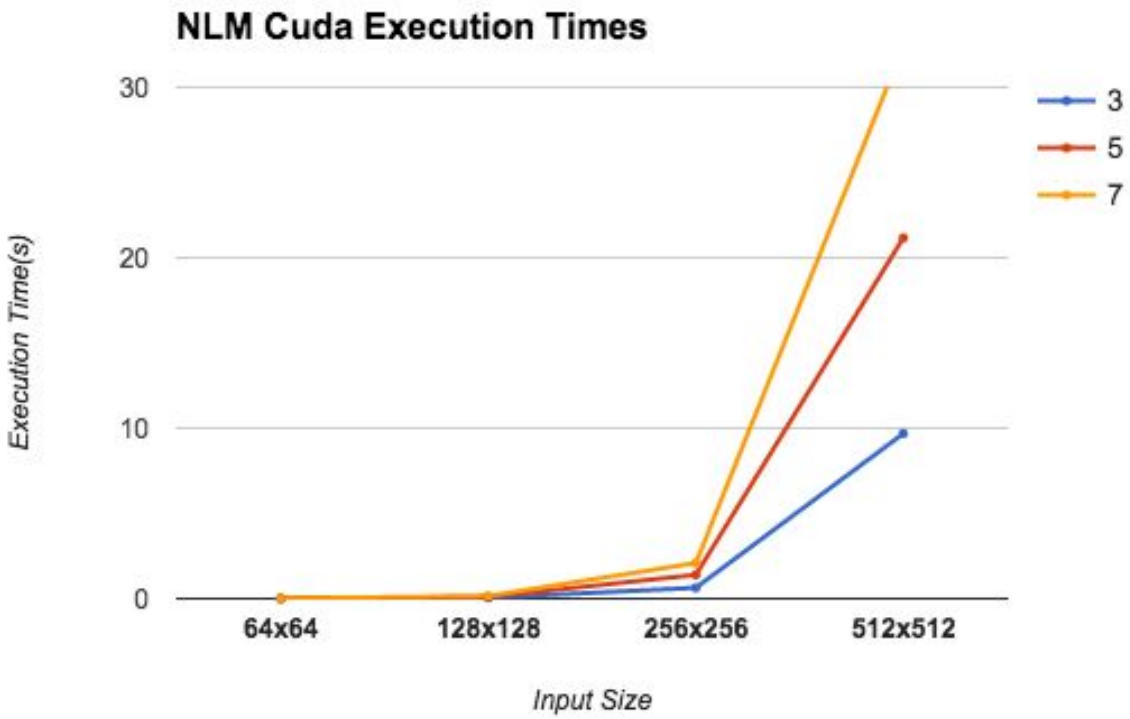


Picture Size	Patch Size	Execution time CPU	Execution time GPU
256x256	3	-	0.636227
	5	-	1.396885
	7	-	2.088931

512x512 Picture

Picture Size	Patch Size	Execution time CPU	Execution time GPU
512x512	3	-	9.685985
	5	-	21.171366
	7	-	32.400761

Summary diagram



Trial and Error

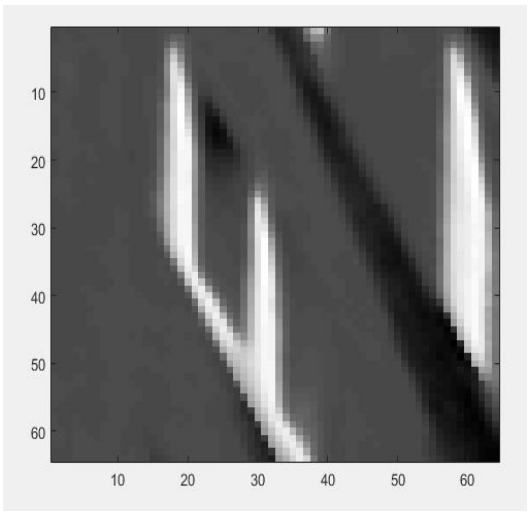
Single Precision

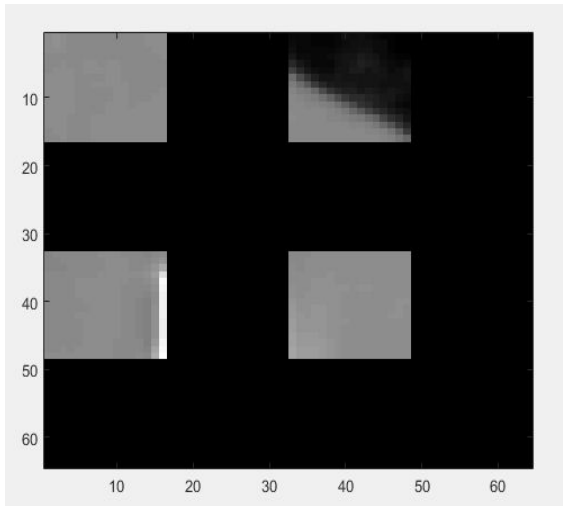
We noticed that just by sending the image to GPU and back we have an error of the the order of 10^-7. This is probably because of the single precision of the image.

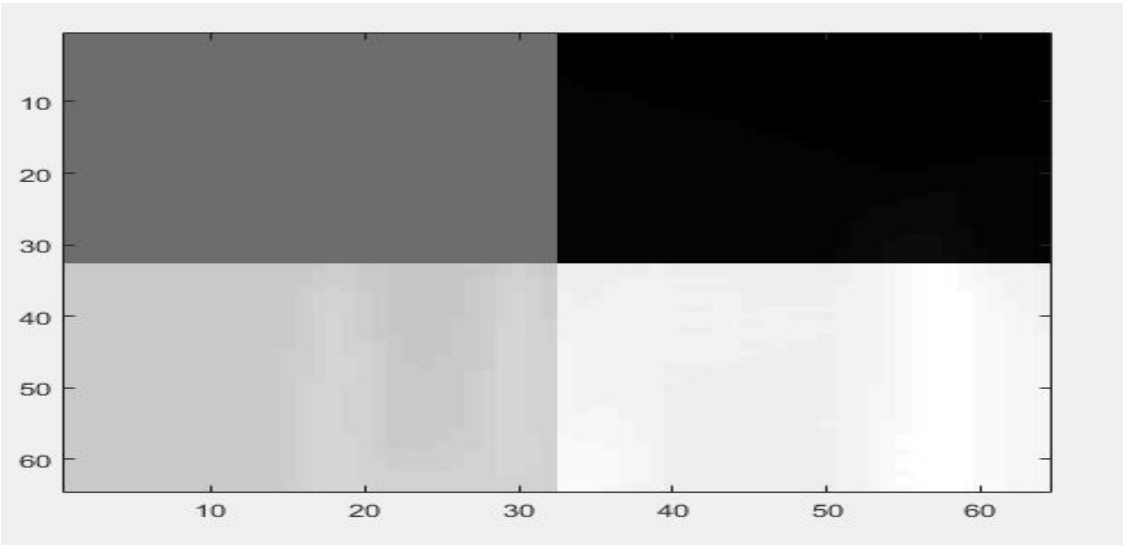
powf(diff, 2)

At first we used the cuda function powf() to calculate the square of the pixel difference in the weight calculation. The overhead of calling this function that cannot be inlined, was so significant that we had 30 bigger longer execution times.

Unsuccessful tries







References

[1]https://www.bu.edu/pasi/files/2011/07/Lecture31.pdf
https://www.mathworks.com/help/distcomp/run-cuda-or-ptx-code-on-gpu.html
http://supercomputingblog.com/cuda-tutorials/
https://en.wikipedia.org/wiki/CUDA
https://www.bu.edu/pasi/files/2011/07/Lecture31.pdf
https://en.wikipedia.org/wiki/Non-local_means
http://supercomputingblog.com/cuda-tutorials/
https://www.mathworks.com/help/distcomp/run-cuda-or-ptx-code-on-gpu.html
http://mc.stanford.edu/cgi-bin/images/5/5f/Darve_cme343_cuda_2.pdf
http://www.itp.cas.cn/kxjs/jzytz/pxyjz/201306/P020130624362235915673.pdf
http://stackoverflow.com/questions/14093692/whats-the-difference-between-cuda-shared-and-global-memory
https://developer.nvidia.com/cuda-faq
https://devblogs.nvidia.com/parallelforall/power-cpp11-cuda-7/
https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/
http://stackoverflow.com/questions/5531247/allocating-shared-memory/5531640#5531640