;

Department of Electrical and Computer Engineering

University of Thessaly

UNIVERSITY OF THESSALY

FOUNDED 1984

ECE415: High Performance Computing (HPC)

Lab 3: CUDA Implementation of a 2D Separable Convolution

Ioanna-Maria Panagou, 2962

Nikolaos Chatzivangelis, 2881

Fall Semester 2022-2023

Contents

1 Summary

In this lab, we had to implement a parallel GPU version of a code that applies a separable 2D convolution using a random mask on a 2D image using CUDA.

2 Introduction

Typically, the application of a 2D convolution filter for an output cell update requires $n \times m$ multiplications, where n and m are the filter dimensions. Some filters are separable, meaning they can be expressed as 2 one dimensional filters, one that is applied to the rows of the image and one to the columns. For example, the application of the below Sobel filter

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

is equivalent to the application of $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ followed by the application of $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$.

The `convolutionRow` function traverses the image cell by cell and applies a row-wise convolution, storing the result in a temporary buffer as shown in fig.1

Once the buffer has been properly filled in, it is fed as input to the `convolutionCol` function that traverses the image and performs a column-wise convolution as shown in fig.2. It should be noted that there is no element under the cell with the value 14 in the second step of 2 and therefore we define the default value 0 to replace all missing values that constitute boundary conditions.

The remainder of our report is structured as follows: section 3 shortly describes the questions of our assignment and our proposed solutions.

3 Questions

3.0 Device Query

The output of the device query is contained in the deliverables. The most useful features of the Tesla K80 device of the CSL-artemis that assisted us during development are:

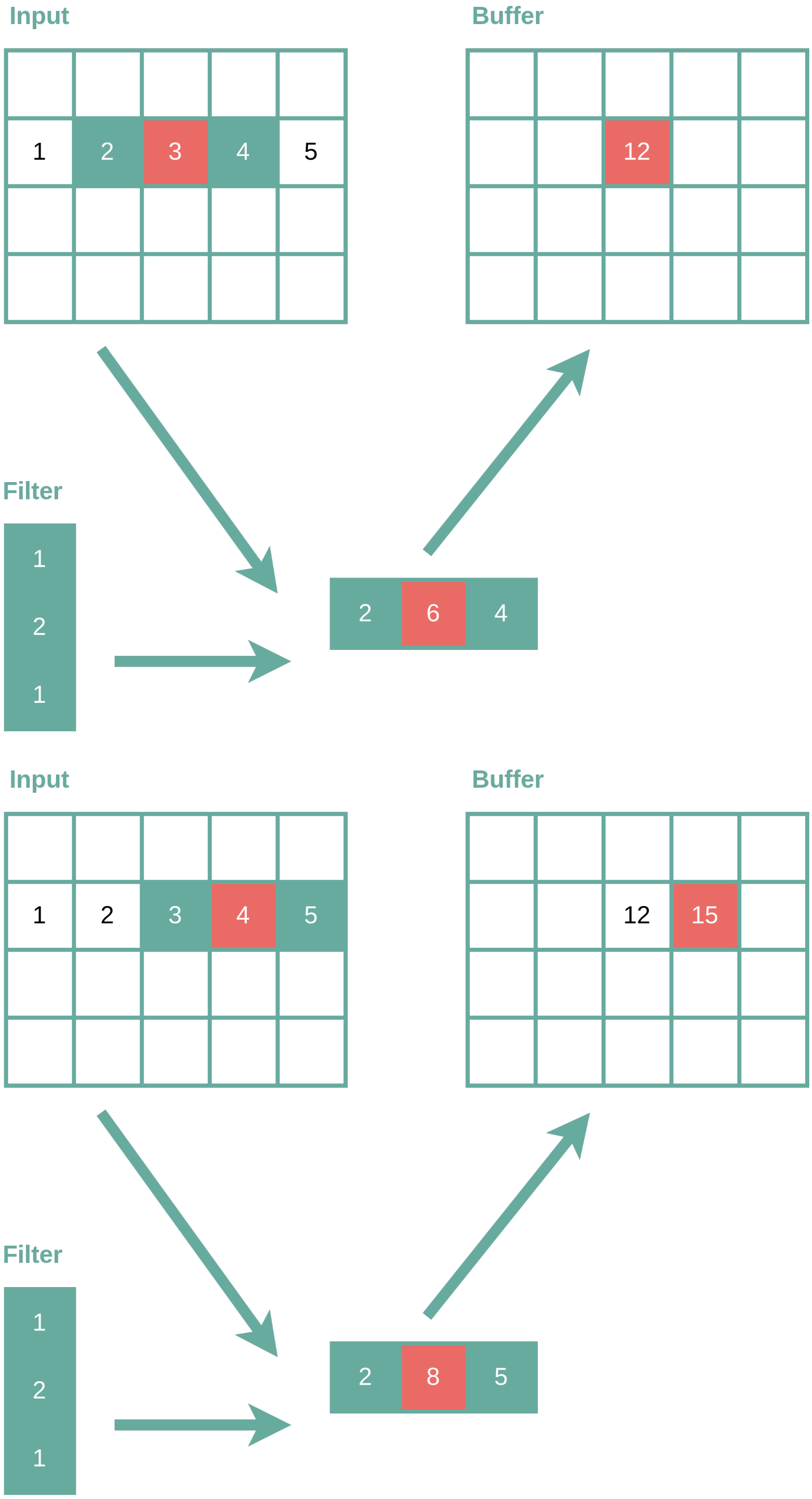

Figure 1: Two consecutive steps of row-wise convolution

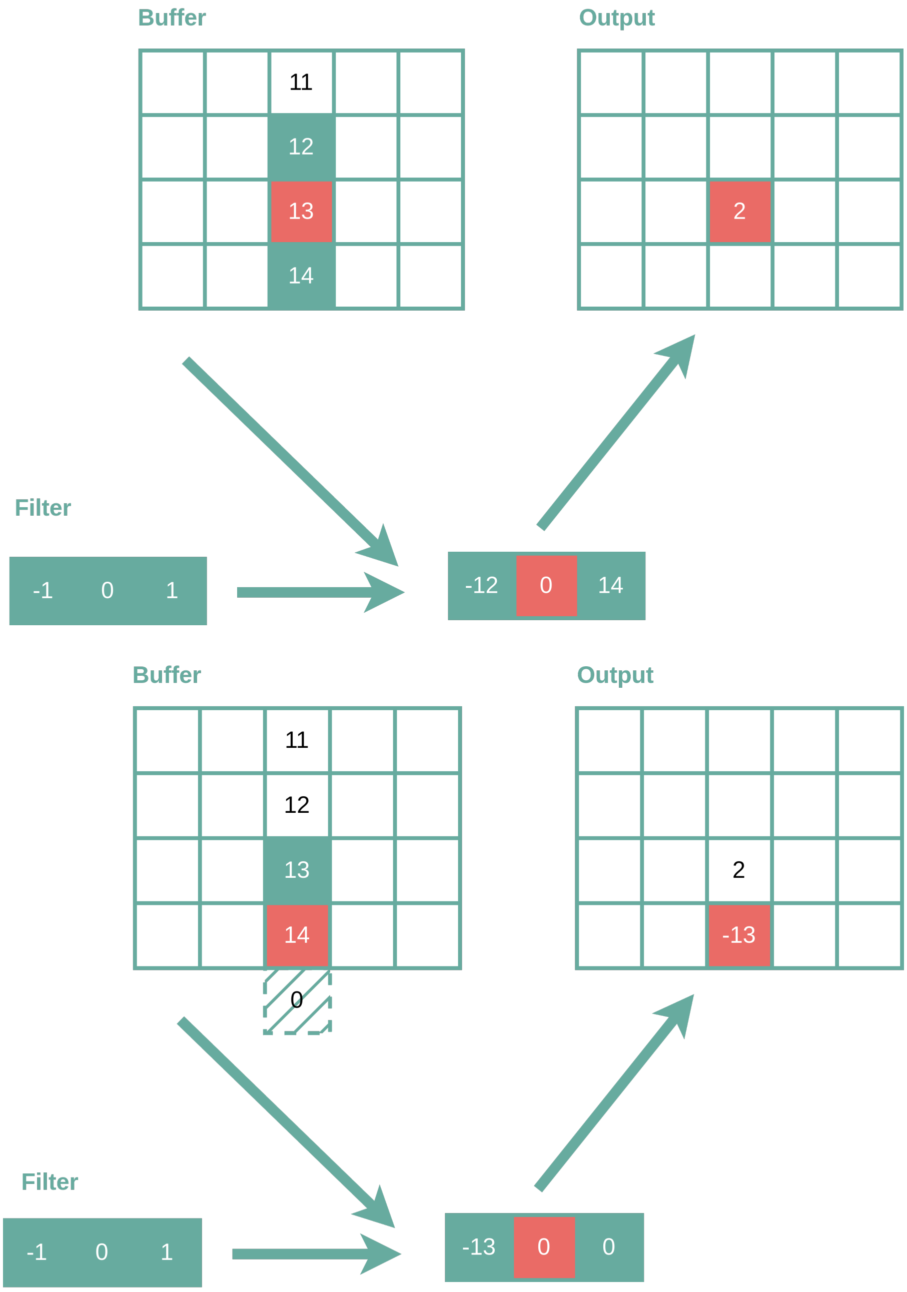

Figure 2: Two consecutive steps of column-wise convolution

```
Total amount of global memory:                 11441MBytes (11997020160 bytes)
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z):  (1024, 1024, 64)
Max dimension size of a grid size (x,y,z):     (2147483647, 65535, 65535)
```

3.1 NVCC Compiler Parameters

After running `nvcc -help`, we examined in detailed the provided compiler options. We compiled our CUDA source files with:

```
nvcc -G -g -O4 Convolution2D.cu -o Convolution2D
```

3.2 CUDA Implementation using a single block of threads

We initially allocated device pointers for the input image, the temporary buffer, the filter and the output image, as well as a host pointer where the output of the GPU is to be copied after the kernel execution has finished, in order to compare the result with the output of the CPU implementation. Then, we copied the filter and the input to the global memory of the device, executed the `convolutionRowGPU` and `convolutionColGPU` (listings 1, 2) kernels and copied the output array back to the host. We organized our threads into a 1D grid, comprised of a single block, which in turn organizes its threads into a 2D topology with dimensions equal to those of the input image, so that each thread computes one output element. To check the correctness of our implementation, we compare the absolute difference between the corresponding values of the CPU and GPU output with a certain threshold.

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src, float *
    d_Filter, int imageW, int imageH, int filterR) {
    int k;
    float sum = 0;

    for (k = -filterR; k <= filterR; k++) {
      int d = threadIdx.x + k;

      if (d >= 0 && d < imageW) {
        sum += d_Src[threadIdx.y * imageW + d] * d_Filter[filterR - k];
      }
    }
    d_Dst[threadIdx.y * imageW + threadIdx.x] = sum;
}
```

Listing 1: `convolutionRowGPU`

```
__global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, float *d_Filter, int imageW, int imageH, int filterR) {
    int k;
    float sum = 0;

    for (k = -filterR; k <= filterR; k++) {
      int d = threadIdx.y + k;

      if (d >= 0 && d < imageH) {
        sum += d_Src[d * imageW + threadIdx.x] * d_Filter[filterR - k];
      }
    }
    d_Dst[threadIdx.y * imageW + threadIdx.x] = sum;
}
```

Listing 2: `convolutionColGPU`

3.3 Experimental Evaluation

a) We can support images up to 32×32 pixels. This is due to the fact that we are confined to only $1024 = 32 \cdot 32$ threads in single block and we are using only one block.

b) We noticed that the value of the threshold for the comparison between the CPU and GPU results had to be pretty high in order for the comparison to be successful, so we started from an accuracy of 4 and each time we passed a test, we reduced the accuracy by 50% (i.e. we divided it by 2). For filters with filter radius up to 15, table 1 and figure 3 were generated:

We observe that as the filter radius increases, the value of tolerance generally increases, which makes intuitive sense, since the larger the filter radius, the more floating point operations have to be done and, therefore, the larger the error margin should be.

3.4 CUDA Implementation using multiple blocks

In order to be able to support larger image sizes, we rewrote the code of 3a), so that the threads are organized in a square grid. For the grid to be square, assuming that image height is equal to image width, block width and block height should match. To make the most out of each block, we maximize the number of threads in the x and y dimensions, so that their product is equal to the maximum number of threads per block, 1024. Our proposed geometry uses blocks with `BLOCK_DIM_X = BLOCK_DIM_Y = 32` and `GRID_DIM_X = GRID_DIM_Y = imageW/BLOCK_DIM_W`, unless the image is smaller than 32 x 32, where we use a grid that consists of a single block with `imageW` threads in each dimension x, y.

The new `convolutionRowGPU` and `convolutionColGPU` are presented in listings 3 and 4

Filter Radius	**Accuracy**
1	0.00781
2	0.0156
3	0.0625
4	0.25
5	0.25
6	0.5
7	0.5
8	1
9	1
10	1
11	2
12	1
13	2
14	1
15	1

Table 1: Accuracy values for 32x32 image

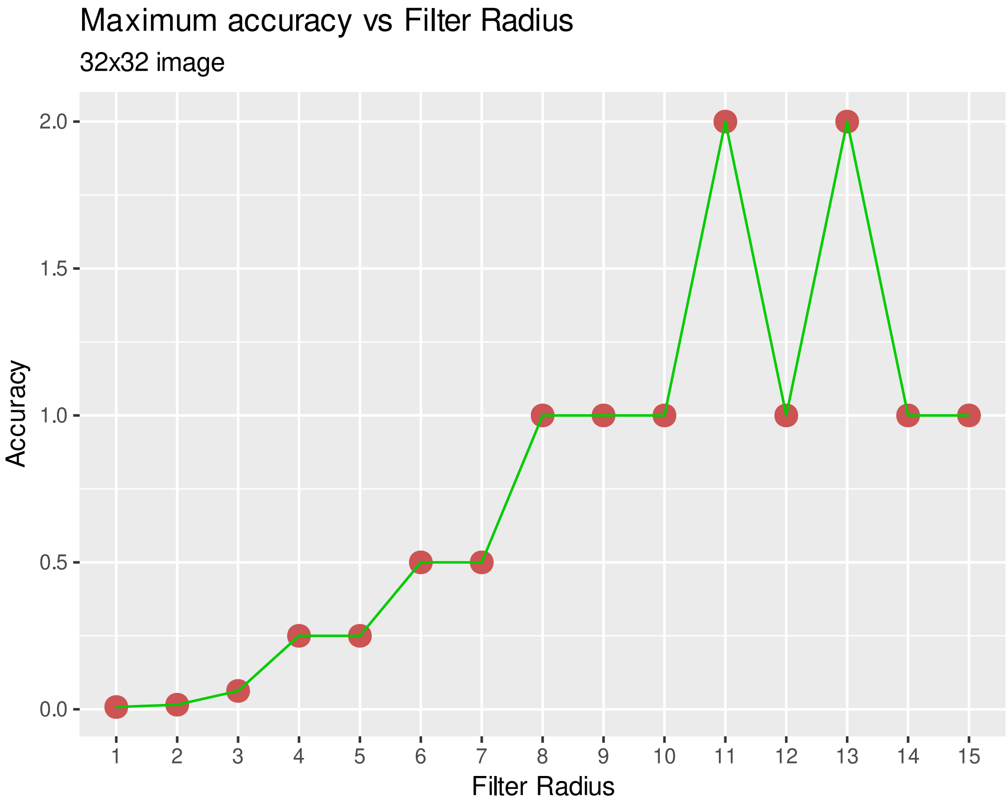

Figure 3: Accuracy values for 32x32 image

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src, int imageW, int imageH, int filterR) {
  int k;
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  float sum = 0;

  for (k = -filterR; k <= filterR; k++) {
    int d = idx + k;

    if (d >= 0 && d < imageW) {
      sum += d_Src[idy * imageW + d] * filter[filterR - k];
    }
  }
  d_Dst[idy * imageW + idx] = sum;
}
```

Listing 3: `convolutionRowGPU`

```
__global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, int imageW, int imageH, int filterR) {
  int k;
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  float sum = 0;

  for (k = -filterR; k <= filterR; k++) {
    int d = idy + k;

    if (d >= 0 && d < imageH) {
      sum += d_Src[d * imageW + idx] * filter[filterR - k];
    }
  }
  d_Dst[idy * imageW + idx] = sum;
}
```

Listing 4: `convolutionColGPU`

Observe that we no longer pass the filter as an input parameter to the kernels. We decided to declare it as global and store it in the constant memory, since its size is usually small, its contents do not change throughout the execution of the kernels and all threads need to access its elements in the same order throughout the execution of one kernel, which make it an excellent candidate for constant memory and caching. Keep in mind that constant memory is limited to 65536 bytes, so we can support filters of radius up to 8191 ($(2 \cdot 8191 + 1) \cdot 4 = 65532 < 65536$ and $(2 \cdot 8192 + 1) \cdot 4 = 65540 > 65536$).

As for the size of the image, we need 3 arrays, one for the input, one for the buffer and one for the output. With that in mind, the largest image we would be able to support is

16384 × 16384, because $16384 \times 16384 \times 4 \times 3 = 3221$MBytes $<$ 11441MBytes and $32768 \times 32768 \times 4 \times 3 = 12884$MBytes $>$ 11441MBytes. However, since we do not need the output array in the `convolutionRowGPU` kernel and the input array in the `convolutionColGPU` kernel, we could allocate space in the device global memory for only the input and buffer arrays, free the input array after the `convolutionRowGPU` kernel and allocate the output array before the `convolutionColGPU` and after the input array has been freed, which would result in a total of 2 arrays existing in the global memory at all times and, hence, the 32768 × 32768 array would fit ($32768 \times 32768 \times 4 \times 2 = 8589$MBytes $<$ 11441MBytes). We would, however, have to pay the price of calling `cudaFree` midst execution of the kernels, which would increase the execution time. To summarize, using the last trick mentioned, we could fit an image up to 32768 × 32768 with a penalty in execution time or an image up to 16384 × 16384 without the penalty.

3.5 Experimental Evaluation

a) Given that the value of tolerance that leads to successful comparisons is, again, pretty high, we start with a value of tolerance 0.5 and with every successful test, we divide it by 10. The number of zeros after the decimal point of the last value of tolerance that leaded to successful comparisons gives us the accuracy in decimal digits (0 if the test with tol=0.5 was not successful or was the last successful test, 1 if the last successful test was with a value of tol equal to 0.05 and so on). Given that a typical convolution filter is usually small, we tested filters with radius up to 16. The results are shown in fig.4. As the filter radius increases, the accuracy in decimal digits rapidly decreases. By the time we reach a filter radius of 3, the accuracy has dropped at 0 decimal points.

b) Fig. 5 presents the mean execution time in seconds for the CPU and GPU execution time. The GPU execution time includes the input and output transfers. We allocate all the necessary device memory before measuring the execution time and do not free any memory between execution of the kernels, so only images up to 16384 image width are supported. Table 2 demonstrates the standard deviation of the execution time across the diferent image sizes for both the CPU and GPU. We also provide fig 6 to quantify the difference in execution time with and without the `cudaFree`.

   Fig 7 shows the percentages for different stages of the computation out of the total execution time for both the CPU and the GPU. We can see that in the case of the CPU, the `convolutionColCPU` function generally performs worse than the `convolutionRowCPU` function, due to the column wise memory access pattern. In the case of the GPU, it is evident that the transfers take up more than half of the execution time and this percentage increases as the images get larger.

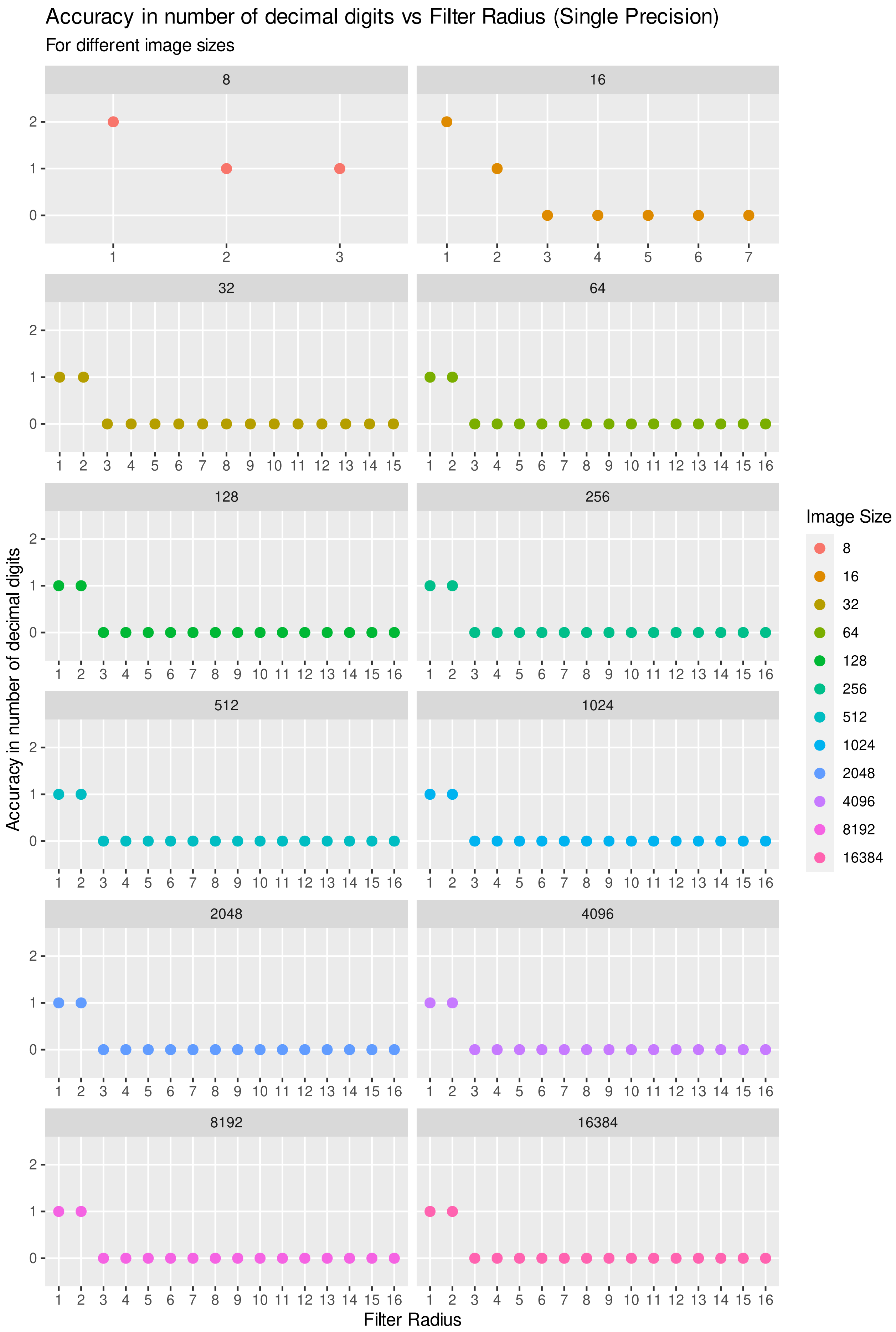

Figure 4: Accuracy in number of decimal digits vs Filter Radius for different image sizes and single precision

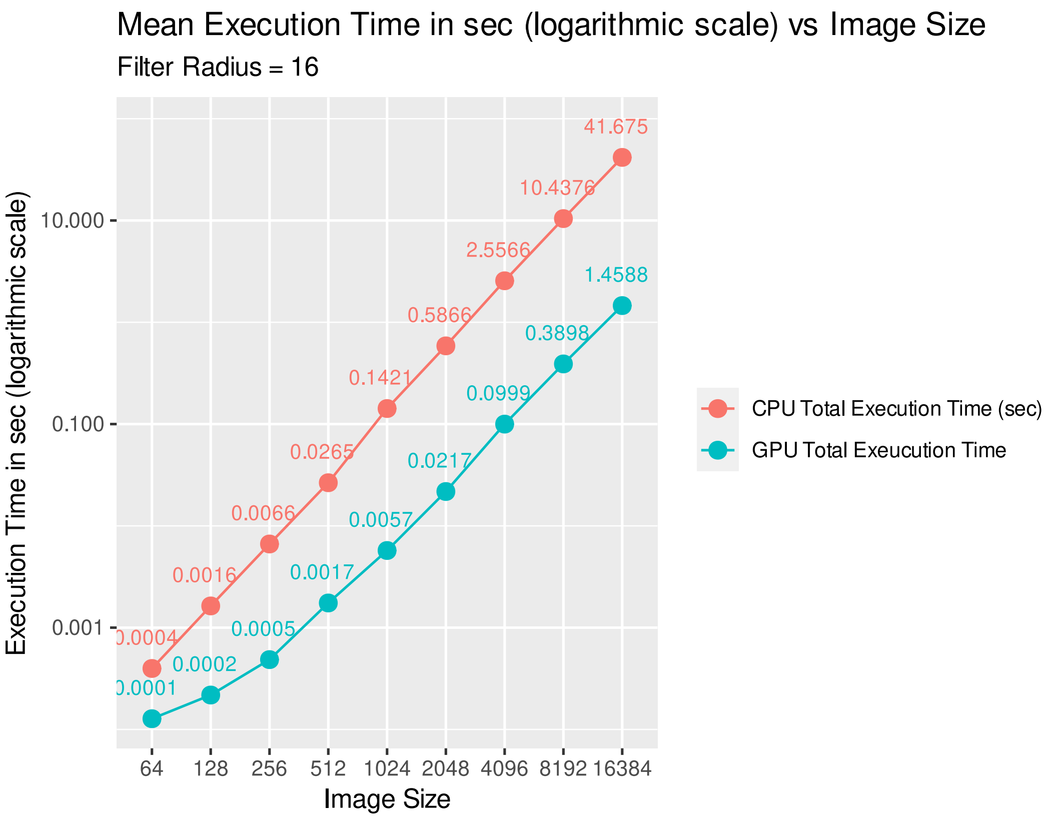

Figure 5: Mean Execution Time in seconds for CPU and GPU vs Image Size

Standard Deviation		
Image Size	CPU	GPU
64	0.0000188	0.0000079
128	0.0000448	0.0000052
256	0.0001356	0.0000055
512	0.0006619	0.0000196
1024	0.0009289	0.0000256
2048	0.0051111	0.0000735
4096	0.0054964	0.0001788
8192	0.0417840	0.0068984
16384	0.4403437	0.0375276

Table 2: Execution time standard deviation for the CPU and GPU (using floats)

3.6 Experimental Evaluation using doubles

a) In a similar fashion as in 4a), we measured the maximum value of accuracy in decimal digits for the convolution using doubles. The variable `accuracy` also had to be declared as double, so that it could capture the differences between the output values of the CPU and GPU. The results are presented in fig 8. Double precision is far better than single precision in terms of the correctness of the results, since it gives as a greater margin of error.

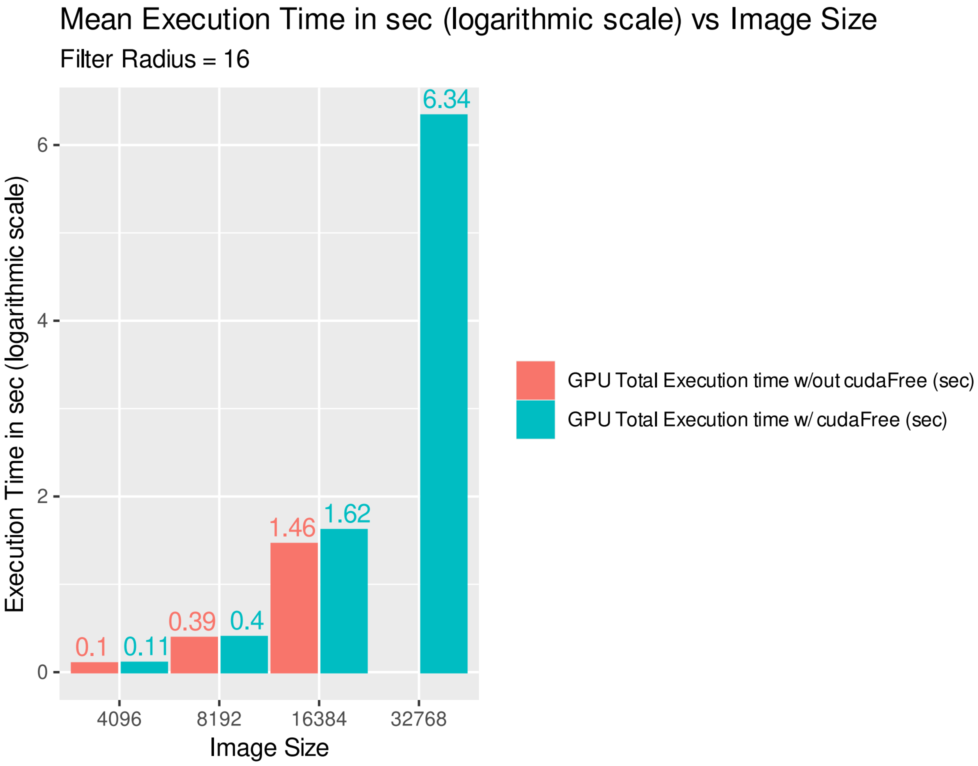

Figure 6: Mean Execution Time in seconds for GPU with and without the cudaFree call in between the `convolutionRowGPU` and `convolutionColGPU` kernels. Notice that without the `cudaFree` call, we cannot support images of 32768 × 32768, but there is extra overhead added

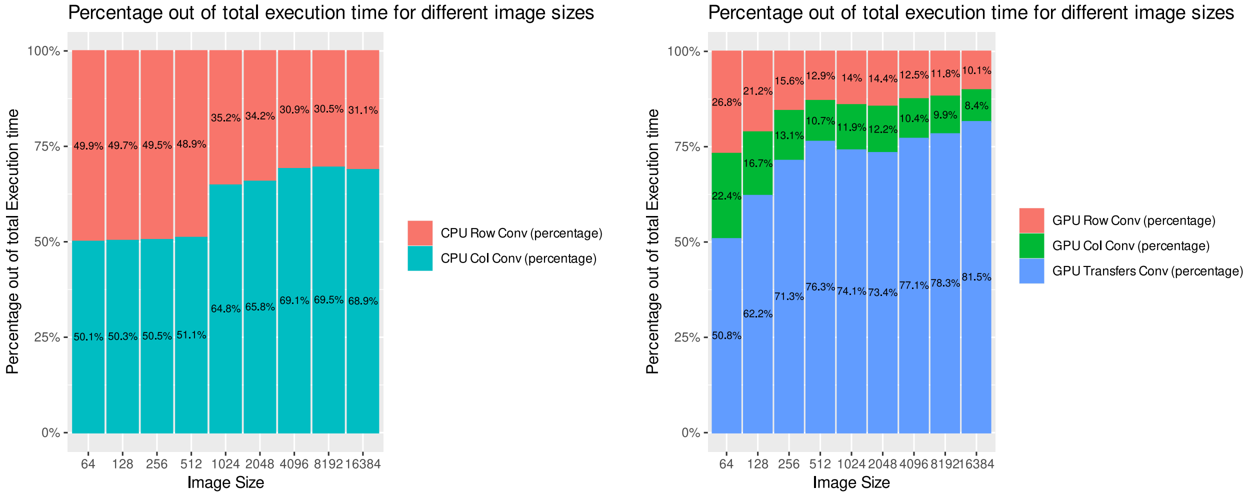

Figure 7: Percentages (%) out of Total Execution Time for CPU (left) and GPU (right) for Single Precision

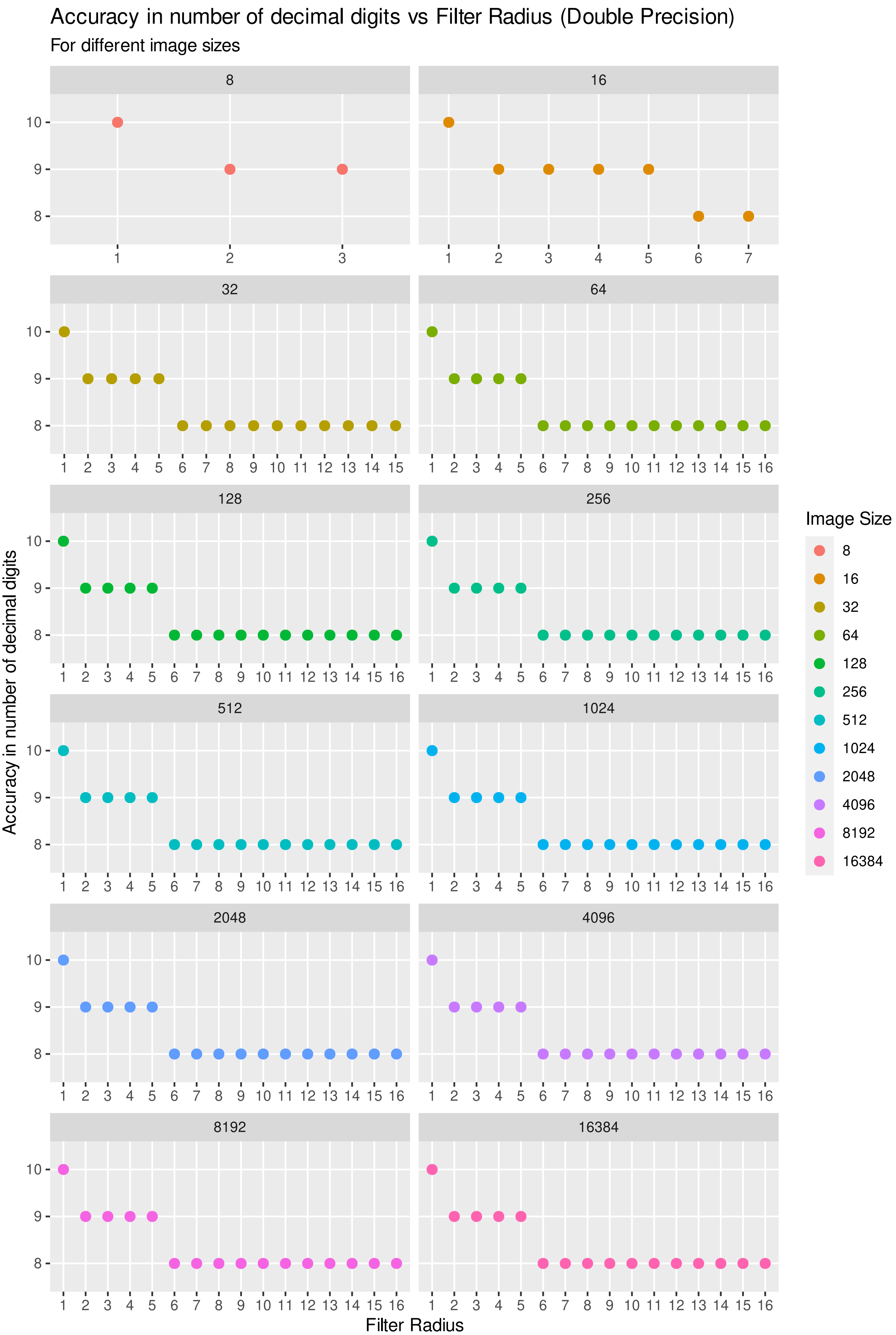

Figure 8: Accuracy in number of decimal digits vs Filter Radius for different image sizes and double precision

b) Figure 9 and table 3 present the mean execution time for the CPU and GPU kernels and the standard deviation, respectively. Although no increase in the CPU execution time was observed, the GPU execution time almost doubled compared to figure 5. Figure 10 better exemplifies the above point. Keep in mind that the axis is in logarithmic scale. The percentage for different stages of the computations remained the same as in the single precision case as can be seen from figures 7 and 11.

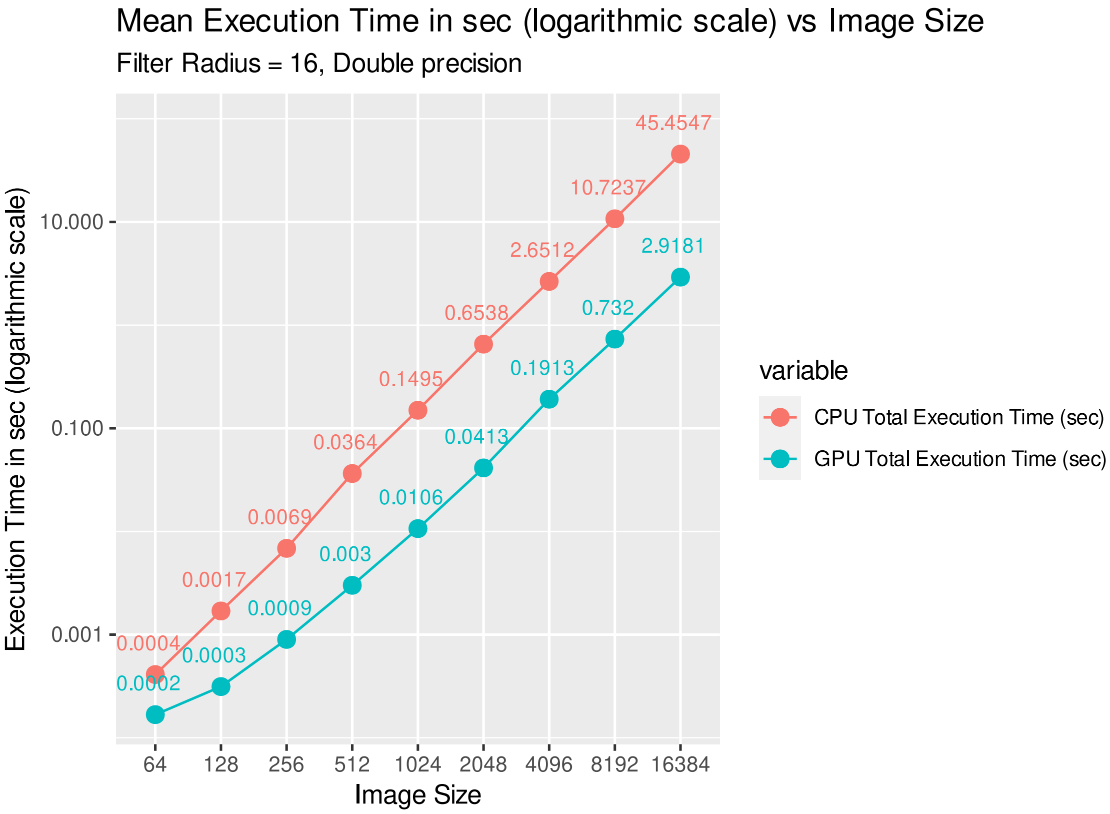

Figure 9: Mean Execution time in sec for CPU and GPU kernels

Standard Deviation		
Image Size	CPU	GPU
64	0.0000075	0.0000040
128	0.0000450	0.0000047
256	0.0001743	0.0000098
512	0.0004295	0.0000205
1024	0.0012342	0.0000344
2048	0.0028727	0.0000729
4096	0.0124157	0.0001711
8192	0.1089075	0.0262291
16384	0.7538971	0.1091101

Table 3: Execution time standard deviation for the CPU and GPU using doubles

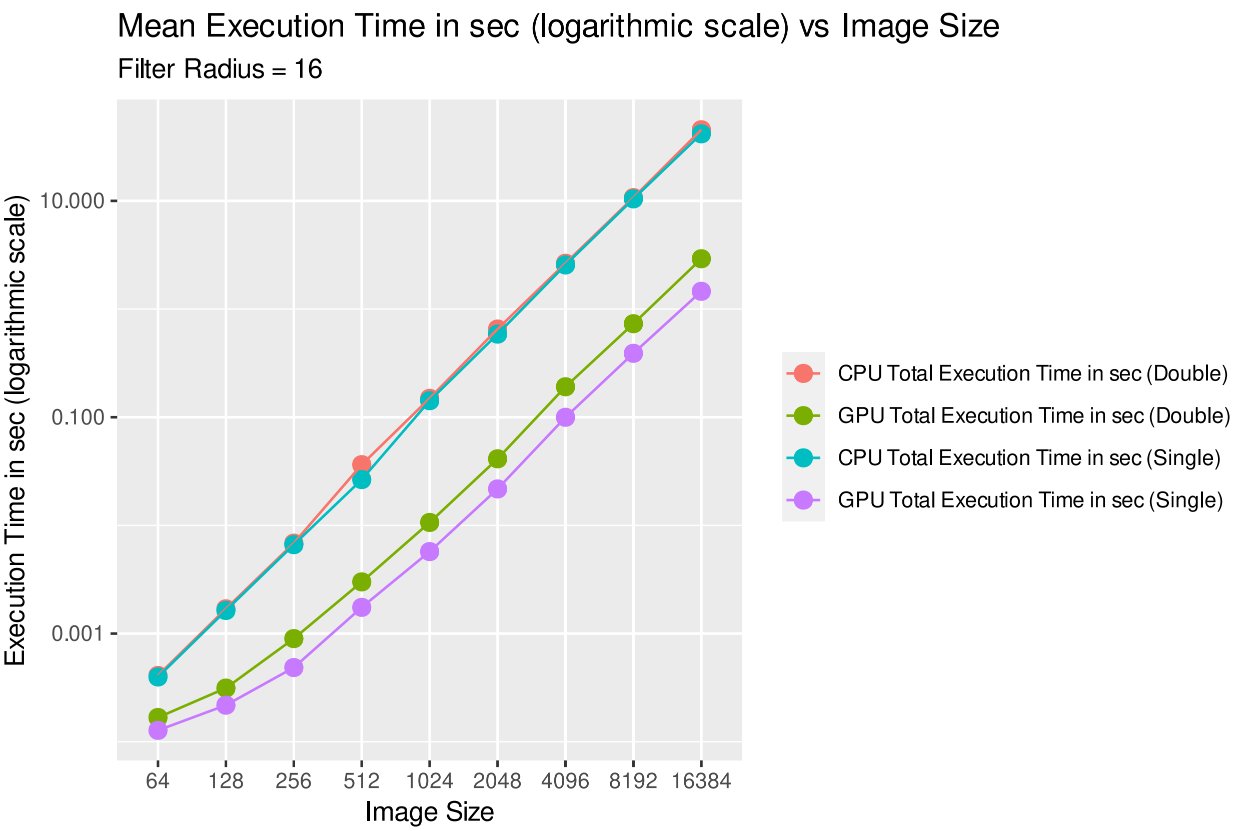

Figure 10: Mean Execution time in sec for CPU and GPU kernels

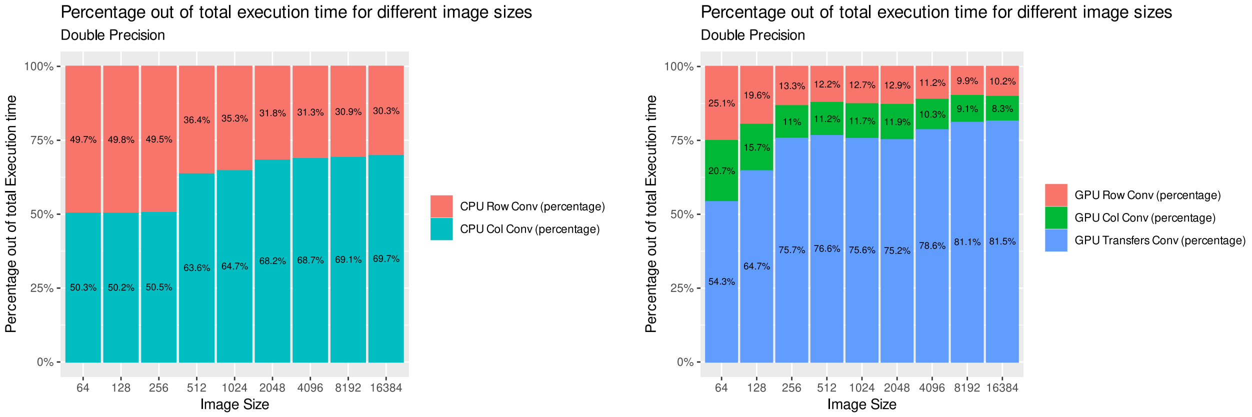

Figure 11: Percentages (%) out of Total Execution Time for CPU (left) and GPU (right) for Double Precision

3.7 Theoretical Performance Analysis

a) **How many times is every element of the input image and the filter read throughout the execution of the kernel?**

Figure 12 depicts an iteration of the row-wise convolution for a single row. The same process is repeated for each row of the input array. The filter slides across the x dimension (the horizontal axis) and accesses the elements sequentially. The elements that are within $FILTER\_RADIUS$ elements from the edges of the array, are accessed less times than the ones that are in the center. More specifically, the elements that are located in the first column (index 0) will be accessed $FILTER\_RADIUS + 1$ times, since on the first iteration, there will be in the center of the sliding filter, so, as the filter is moving, they will be accessed an additional of $FILTER_RADIUS$ times. The elements in the second column (index 1) will be accessed one additional time and so on, until we reach the column indices that are $\geq FILTER\_RADIUS$, where they will be accessed the maximum possible times, i.e. $2 \cdot FILTER\_RADIUS + 1$ times. As we approach the last columns of the input array, the inverse pattern will take place. Basically, the elements that are $FILTER\_RADIUS$ positions far from the edges will be access $2 \cdot FILTER\_RADIUS + 1$ times and the accesses for the ones left and right of those will decrease by the distance of those elements from the closest element that was accessed $2 \cdot FILTER\_RADIUS + 1$. As for the filter, the center element will be accessed once for each element across the x-axis, the elements next to it will be accessed one less time and so on.

$$f_{row,input}(x) = \begin{cases} 2 \cdot FR + 1 - (FR - x) & \text{if } 0 \leq x \leq FR \\ 2 \cdot FR + 1 & \text{if } FR \leq x \leq imageW - FR - 1 \\ 2 \cdot FR + 1 - (x - (imageW - FR - 1)) & \text{if } x > imageW - FR - 1 \end{cases}$$

where FR is the $FILTER\_RADIUS$ and x is the column index

$$f_{row,filter}(i) = imageH \cdot (imageW - \|i - FR\|), i \in 0, 2 \cdot FR$$

where i is the filter array index.

For the column-wise convolution kernel, the pattern remains the same, only now the filter slides from the top to the bottom for each column.

$$f_{col,input}(y) = \begin{cases} 2 \cdot FR + 1 - (FR - x) & \text{if } 0 \leq y \leq FR \\ 2 \cdot FR + 1 & \text{if } FR \leq y \leq imageW - FR - 1 \\ 2 \cdot FR + 1 - (y - (imageW - FR - 1)) & \text{if } y > imageW - FR - 1 \end{cases}$$

where FR is the $FILTER\_RADIUS$ and y is the row index

$$f_{col,filter}(i) = imageW \cdot (imageH - \|i - FR\|), i \in 0, 2 \cdot FR$$

where i is the filter array index.

Summing up the above equations, we get:

$$f_{input}(x,y) = \begin{cases} 2 \cdot FR + 2 + y + x & \text{if } 0 \leq x, y \leq FR \\ 4 \cdot FR + 2 & \text{if } FR \leq x, y \leq imageW - FR - 1 \\ 2 \cdot FR + 2 \cdot imageW - x - y & \text{if } x, y > imageW - FR - 1 \end{cases}$$

where FR is the $FILTER\_RADIUS$ and y is the row index

$$f_{filter}(i) = 2 \cdot imageW \cdot (imageH - \|i - FR\|), i \in 0, 2 \cdot FR$$

where i is the filter array index.

b) **What is the ratio of memory accesses to decimal point operations?**

In line 13 of listing 3, we can see that two decimal point operations take place and one element of the input image and one element of the filter are accessed. The filter is in the constant memory, so it does not count as a memory access, so we only count the accesses for the input image.

Therefore, the ratio is equal to $\frac{1}{2}$, so the kernel is expected to run at a small fraction of the peak performance.

3.8 Padding to avoid the divergence problem

As shown in figure 12, the elements that are within $FILTER\_RADIUS$ elements from the edges of the array, make use of "ghost elements" that do not exist in the array, but have a default value of 0. Those elements constitute the padding of the array and are extended $FILTER\_RADIUS$ elements from the edges of the array in every direction. We added padding for both the CPU and the GPU implementation on the input and the buffer arrays. Listings 5 and 6 present the updated kernels.

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src, int imageW, int imageH, int filterR) {
  int k;
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  float sum = 0;

  for (k = -filterR; k <= filterR; k++) {
```

```
8      int d = idx + k;
9      sum += d_Src[(idy + filterR ) * imageW + d + filterR] * filter[filterR -
       k];
10   }
11   d_Dst[ ( idy + filterR ) * imageW  + idx + filterR] = sum;
12 }
```

Listing 5: `convolutionRowGPU`

```
1  __global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, int imageW,
       int imageH, int filterR) {
2    int k;
3    int idx = blockIdx.x * blockDim.x + threadIdx.x;
4    int idy = blockIdx.y * blockDim.y + threadIdx.y;
5    float sum = 0;
6
7    for (k = -filterR; k <= filterR; k++) {
8      int d = idy + k;
9      sum += d_Src[(d + filterR) * imageW + idx + filterR] * filter[filterR -
      k];
10   }
11     d_Dst[ idy* (imageW - 2* filterR) + idx] = sum;
12 }
```

Listing 6: `convolutionColGPU`

Note that `imageW` and `imageH` are the new width and height values with the added padding. Since the output is not padded, its width has not changed, so we need to substract the added padding in line 14 of 6. To avoid having idle threads, we did not change our geometry from the previous implemention; we still have a 2D grid of `imageW / 32` blocks in each direction and the threads are organized in square blocks of width and height equal to 32 (unless the image is smaller than 32 × 32, where the block dimensions equal the image dimensions).

Figure 15 and table 4 show the mean execution time and the standard deviation for the CPU and GPU implementations with and without padding. Although padding aided the CPU implementation to reach a speedup of approximately 2, it didn't make a particular difference in the GPU and in some cases the performance was worse, as better shown in figure 14.

Solving the divergence problem created a new one; we now have to access the global memory more often since we now read the "ghost" elements from there rather than just setting them to 0. To combat this problem, we created shared arrays per block that contain all the necessary elements for the threads in this block to perform convolution. More specifically, for the row-wise convolution, the elements that are on the edges of each block, require $FILTER\_RADIUS$ elements that belong to the previous block horizontally (or padding if they are in `blockIdx.x = 0`) and $FILTER\_RADIUS$ elements from the next block horizontally (or padding if they are in the last block in the x dimension) as shown in figure 16.

Similarly, for the column-wise convolution, the elements that are on the edges of each block,

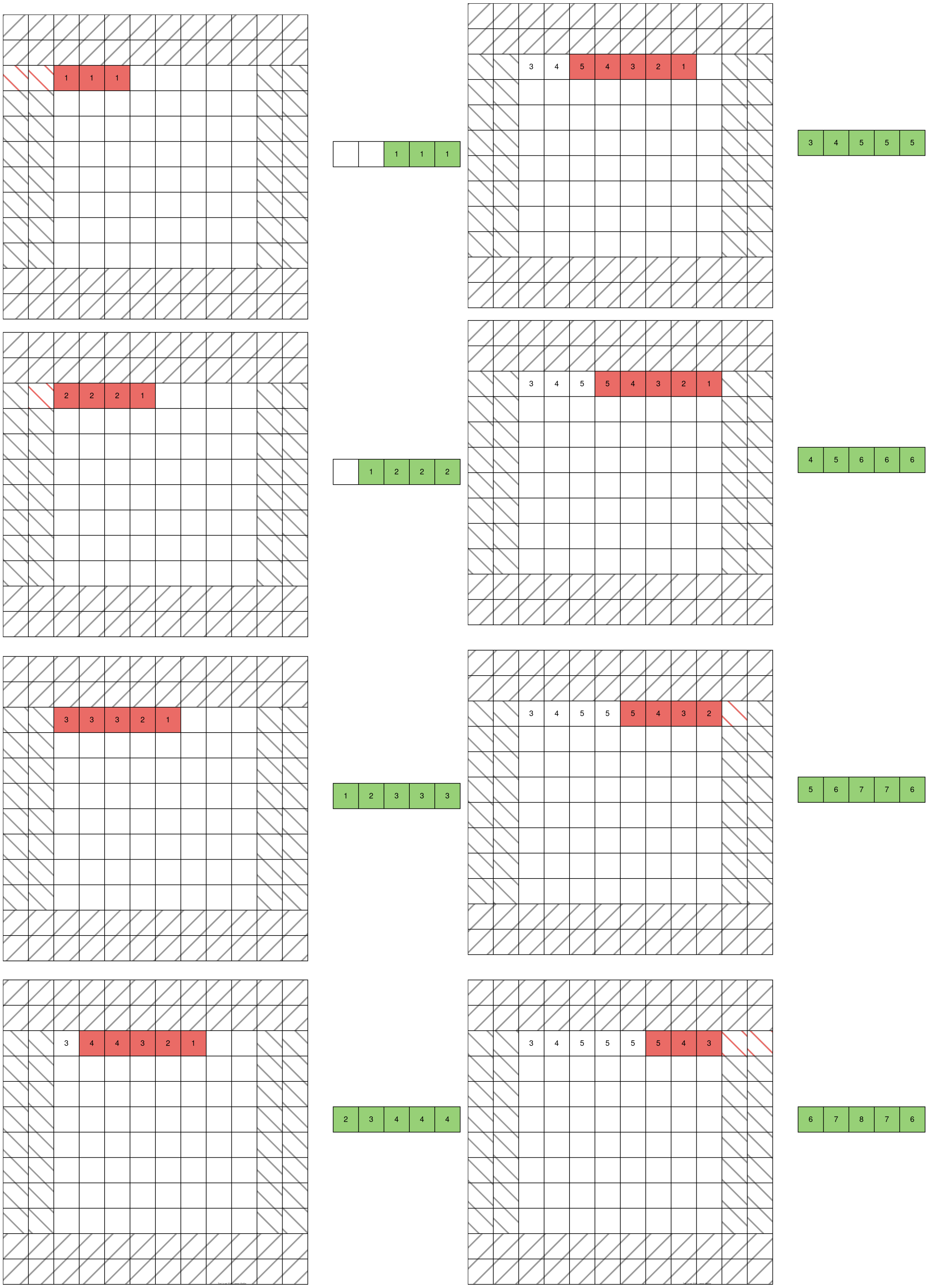

Figure 12: Runtime example of the row-wise convolution for one row. The 4 first steps are depicted in the first column and the last 4 in the second column. The input image is on the left and the filter is on the right. The colored boxes denote which elements are accessed in each iteration and the numbers denote the total number of reads for each element. The array is 8 × 8 elements and the filter has a radius of 2.

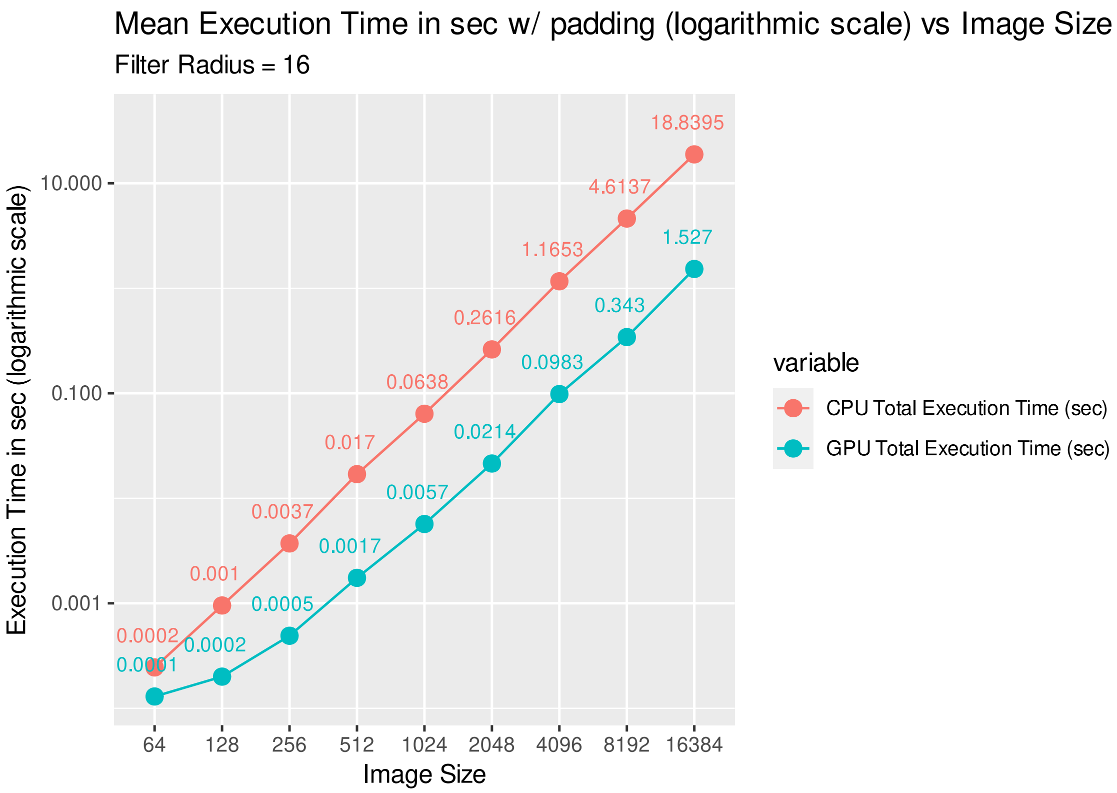

Figure 13: Mean Execution time in sec for CPU and GPU implementations using padding

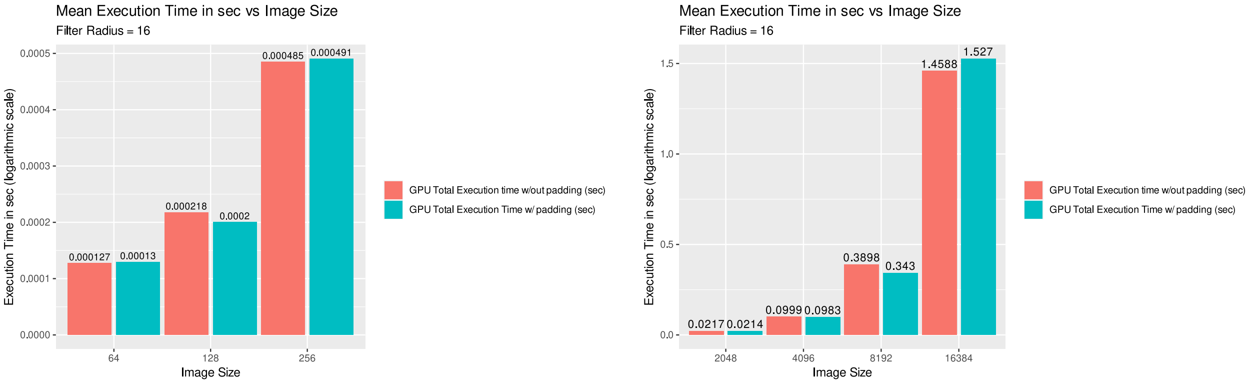

Figure 14: Mean execution time for GPU code using padding for various image sizes

require $FILTER\_RADIUS$ elements that belong to the previous block vertically (or padding if they are in `blockIdx.y = 0`) and $FILTER\_RADIUS$ elements from the next block vertically (or padding if they are in the last block in the y dimension as shown in figure 23

The process of copying the necessary elements for each block of threads is described in lines of 11-19 of listing 7 and lines 12-21 of listing 8. Lines 11-14 in 7 load the elements on the left and lines 15-20 load the elements on the right (figures 18 and 19). In a similar fashion, figures 20 and 21 depict the process of loading the elements in the shared array for the column-wise convolution. There is overlap, meaning that some elements can be loaded twice depending on the size of the array and the size of the filter (which is usually the case), but we would like to

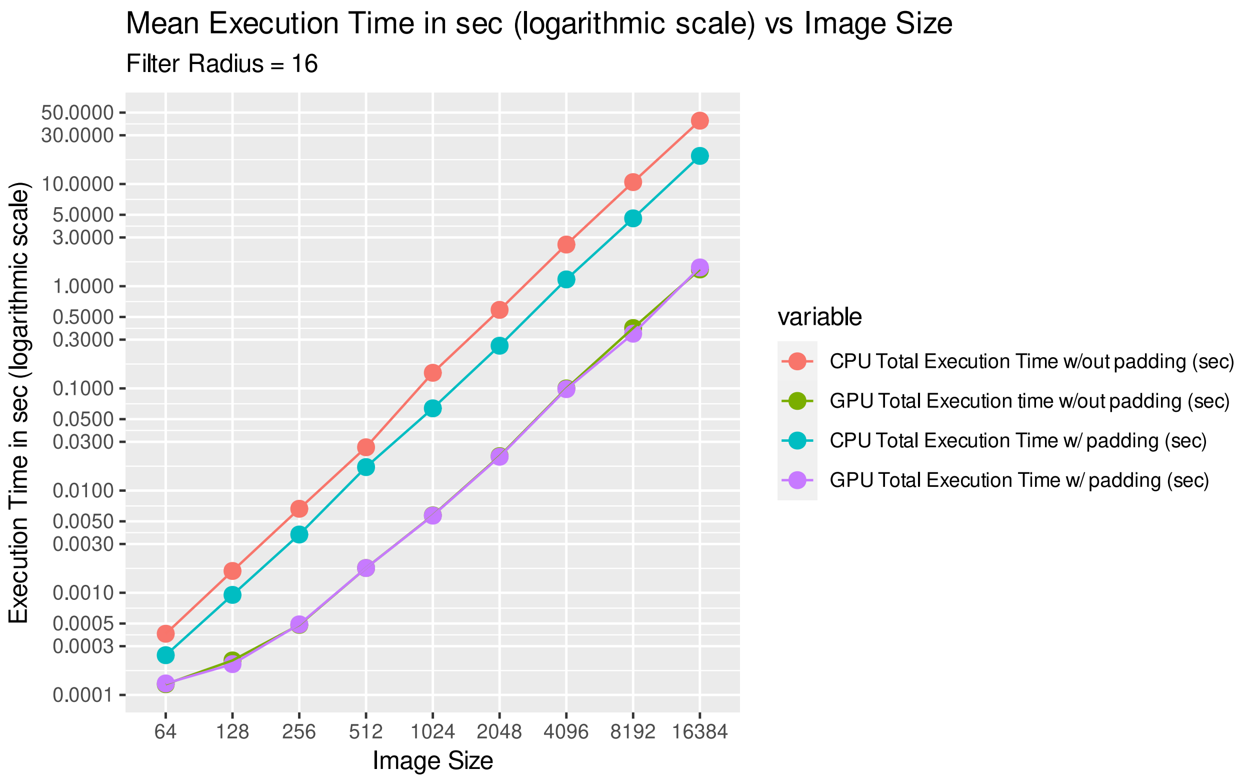

Figure 15: Mean Execution time in sec for CPU and GPU implementations with and without padding

Standard Deviation			
Image Size	CPU	GPU w/out shared	GPU w/ shared
64	0.0000062	0.0000045	0.0000054
128	0.0000357	0.0000053	0.0000109
256	0.0000344	0.0000098	0.0000127
512	0.0000297	0.0000069	0.0000148
1024	0.0002351	0.0000135	0.0000282
2048	0.0026631	0.0001013	0.0001320
4096	0.0255199	0.0007210	0.0003620
8192	0.0838625	0.0363232	0.0031750
16384	0.2597973	0.0154694	0.0306660

Table 4: Standard deviation for the execution time of the CPU and GPU with or without shared memory with padding

get rid of any if clauses that could cause divergence.

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src, int imageW,
    int imageH, int filterR) {

  int k;
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  int id = idy * imageW + idx;
```

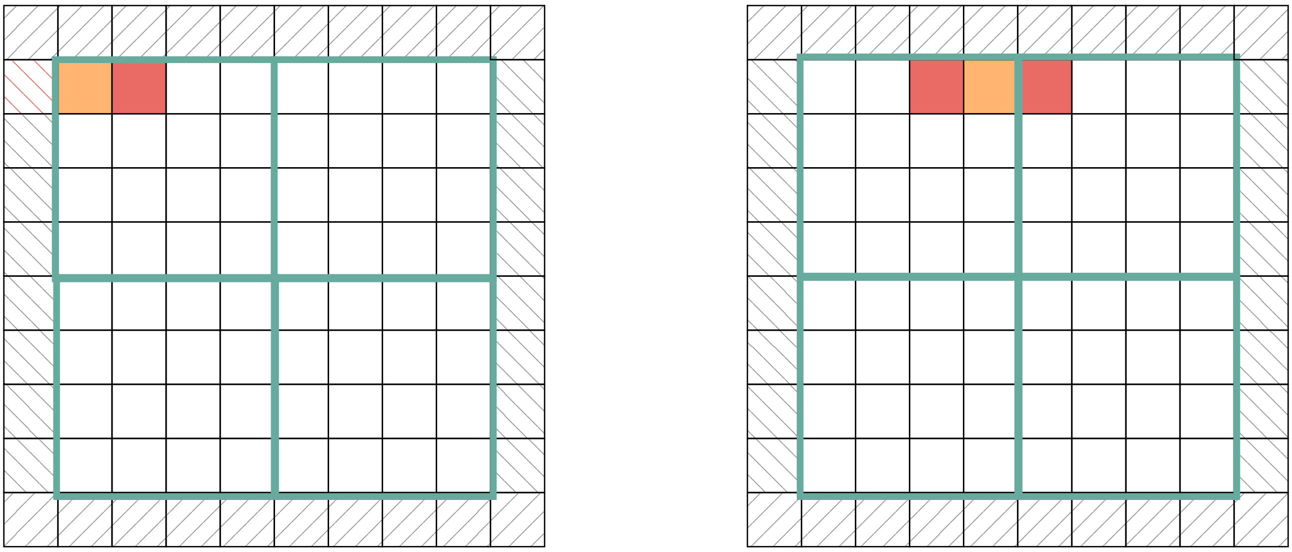

Figure 16: Row-wise convolution, square blocks of 4 threads in each dimension, image width is 8 elements and filter radius is 1

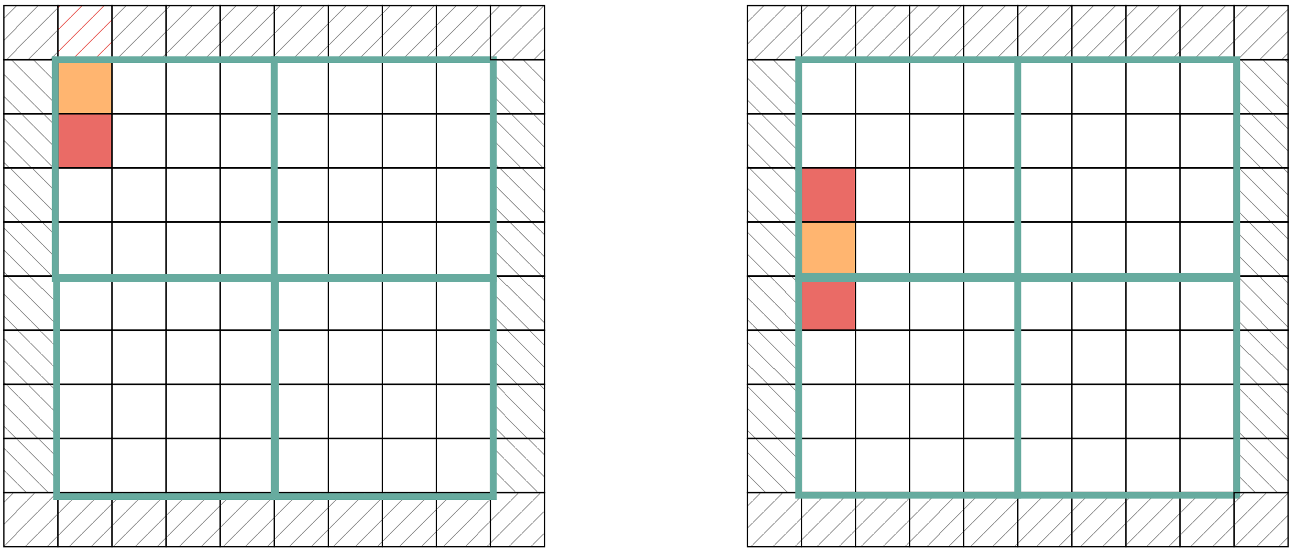

Figure 17: Column-wise convolution, square blocks of 4 threads in each dimension, image width is 8 elements and filter radius is 1

```
  extern __shared__ float shared[];

  float *Src_s = &shared[0];

  int x = idx - filterR;

  Src_s[threadIdx.x + threadIdx.y * (BLOCK_DIM_X + 2 * filterR)] = (x < 0) ? 0 : d_Src[id - filterR];

  x = idx + filterR;

  Src_s[threadIdx.x + 2*filterR + threadIdx.y * (BLOCK_DIM_X + 2 * filterR)] = (x > imageW - 1) ? 0 : \
  d_Src[id + filterR];

  __syncthreads();

  float sum = 0;

  for (k = -filterR; k <= filterR; k++) {
    int d = threadIdx.x + k;
    sum += Src_s[threadIdx.y * (BLOCK_DIM_X + 2 * filterR) + d + filterR] * filter[filterR - k];
  }

  d_Dst[id] = sum;
}
```

Listing 7: convolutionRowGPU

```
__global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, int imageW, int imageH, int filterR) {

  int k;

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int idy = blockIdx.y * blockDim.y + threadIdx.y;
  int id = idy * imageW + idx;

  extern __shared__ float shared[];
  float *Src_s = &shared[0];

  int y = idy - filterR;

  Src_s[threadIdx.y * BLOCK_DIM_X + threadIdx.x] = (y < 0) ? 0 : \
    d_Src[id - filterR * imageW];

  y = idy + filterR;
```

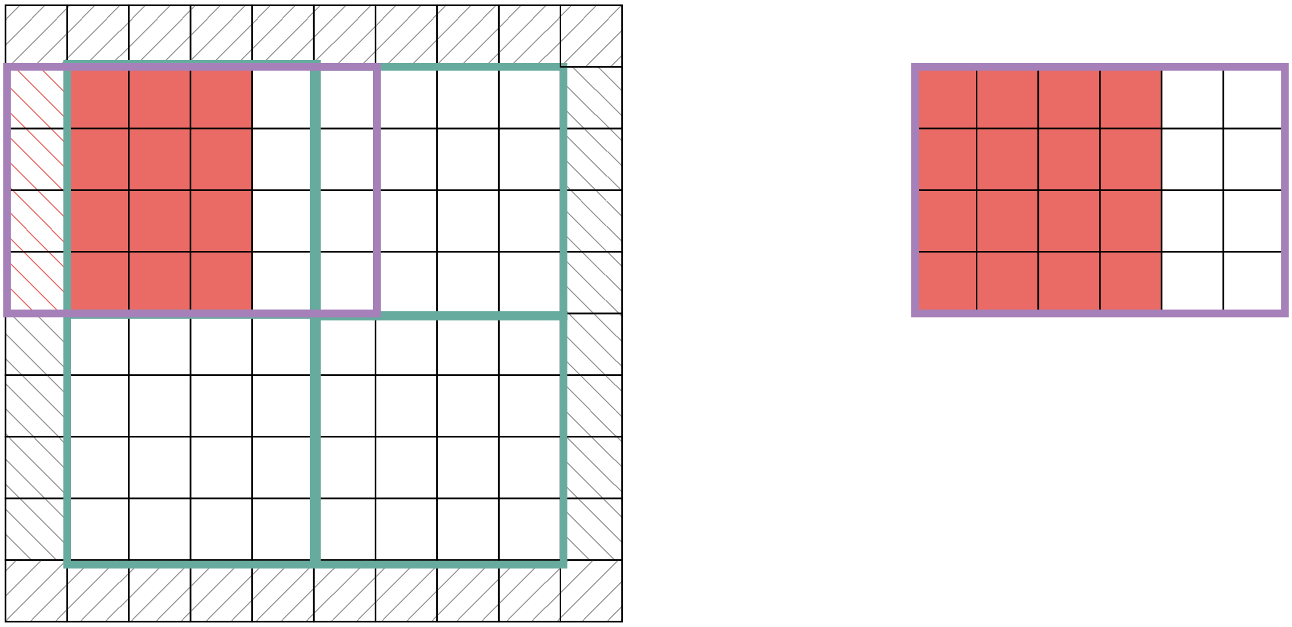

Figure 18: Loading the left elements for the shared array

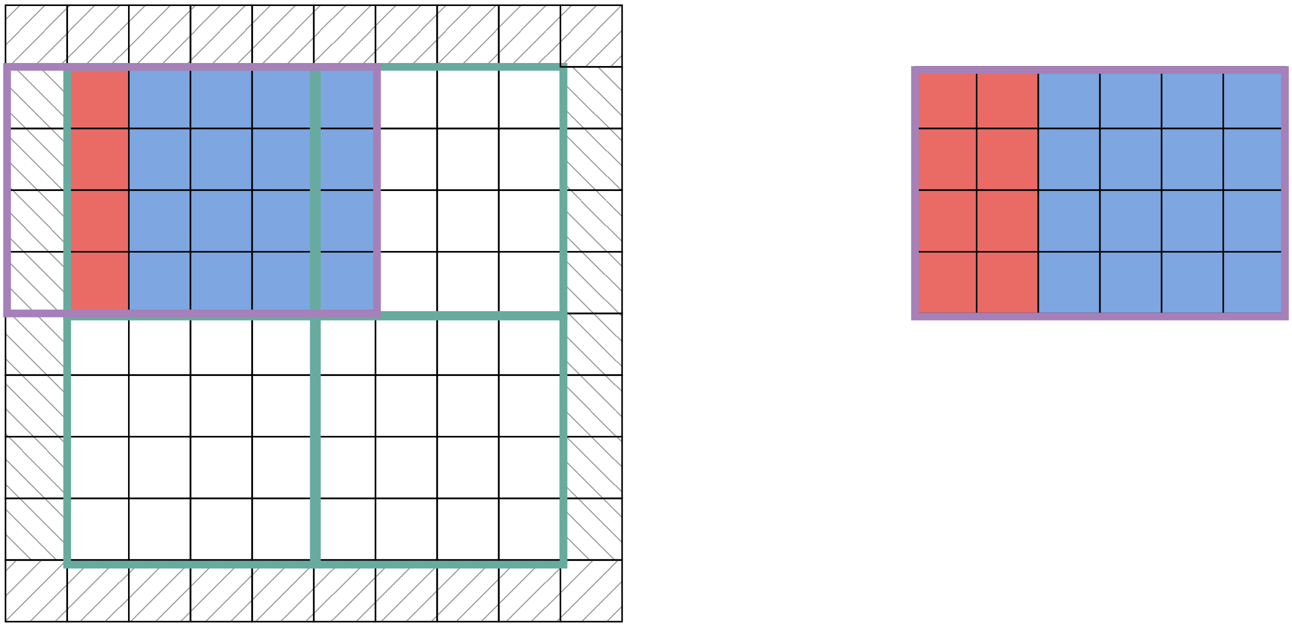

Figure 19: Loading the right elements for the shared array

```

Src_s[(threadIdx.y + 2 * filterR) * BLOCK_DIM_X + threadIdx.x] = (y >
 imageH - 1) ? 0 :\
  d_Src[id + filterR * imageW];

__syncthreads();

float sum = 0;

for (k = -filterR; k <= filterR; k++) {
```

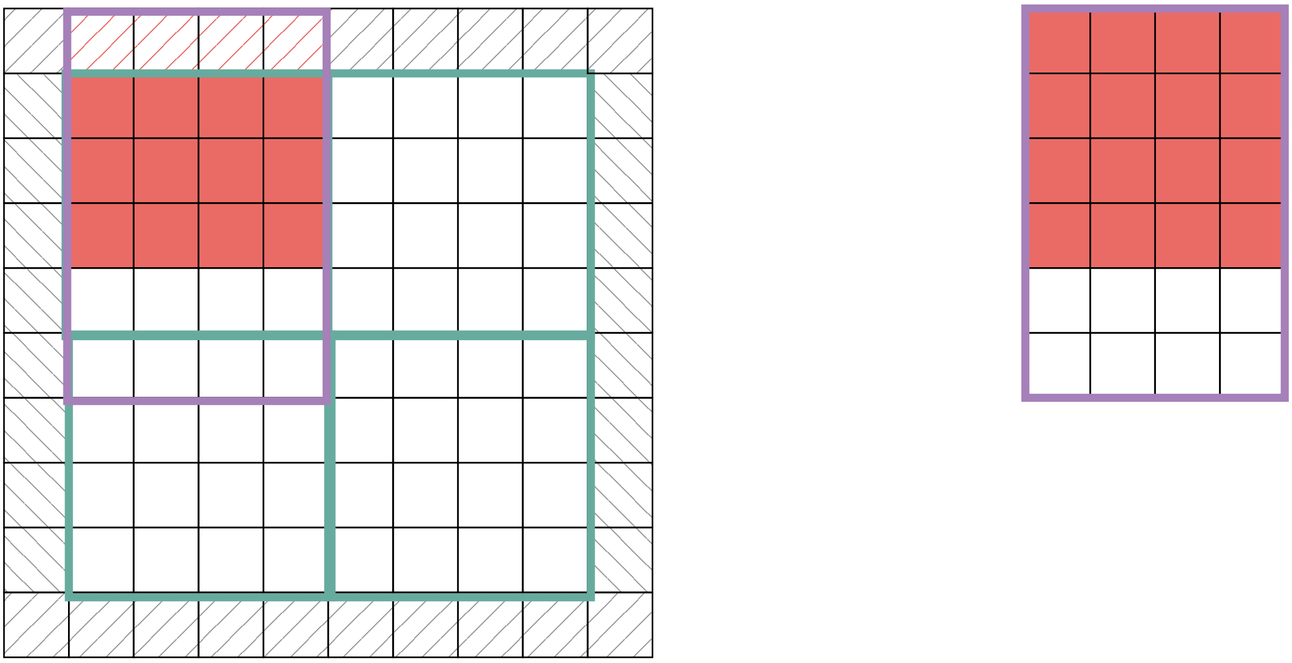

Figure 20: Loading the top elements for the shared array

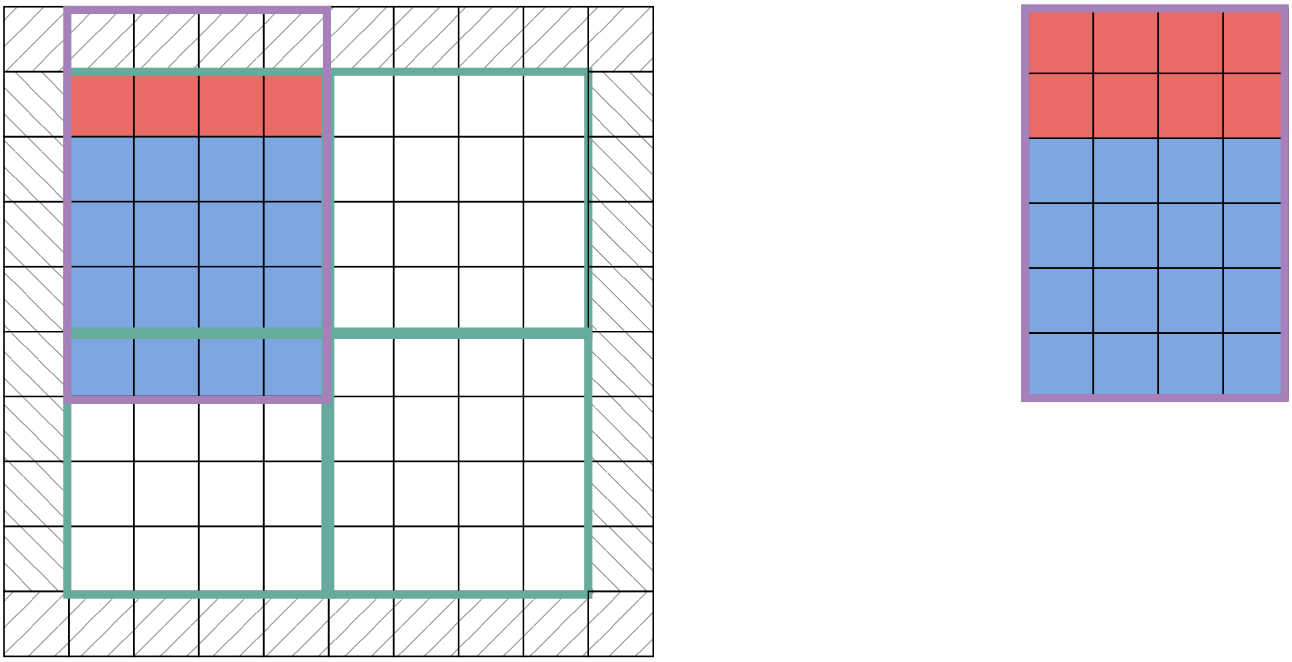

Figure 21: Loading the bottom elements for the shared array

```
        int d = threadIdx.y + k;
        sum += Src_s[(d + filterR) * BLOCK_DIM_X + threadIdx.x] * filter[filterR
        - k];
    }

    d_Dst[id] = sum;
}
```

Listing 8: `convolutionColGPU`

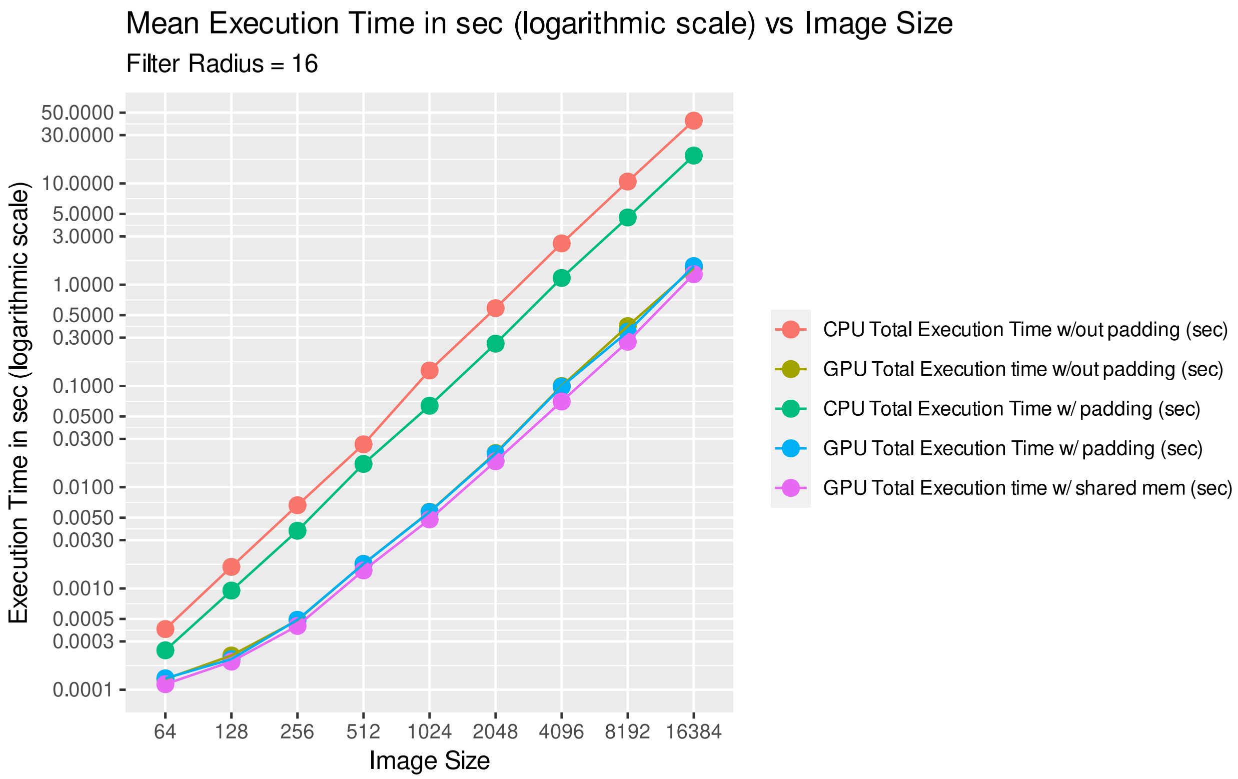

Figure 22: Mean execution for CPU and GPU implementations (single accuracy) vs Image Size

As shown in figure 22, this version outperformed both the GPU version with and without the padding.

4 Conclusions

In this assignment, we had to develop a convolution program to run on a GPU. Initially, our code used a single block of threads, and each thread calculated one pixel of the produced image. Because of the GPU limitation of 1024 threads per block this implementation could only handle images with a maximum resolution of 32×32. Also, we observed that the larger the convolution filter radius was, the lower the accuracy of our program compared to the CPU version because of the higher number of floating point multiplications the GPU performs when the radius is increased.

Using multiple blocks, enabled us to handle larger images because now we had more threads at our disposal. The speedup compared to the CPU was obvious and figure 23 is a proof of that. The speedup we got from the GPU in most cases was 30× from the baseline CPU implementation.

When coding on a GPU ideally we want to make use of all the available threads but we also have to be conscious of the memory usage. From our experiments, we showed that if we cleverly use the memory, we can double the maximum image size we can handle with our hardware. But this had a penalty at the execution time of our program. This is a classic cost benefit situation, where depending on the application we might have to chose between faster execution time or more capacity. Finally, although we expected the padding technic to have a significant improvement at the GPU, eventually it did not, especially compared to the CPU version which got almost 2× speedup.

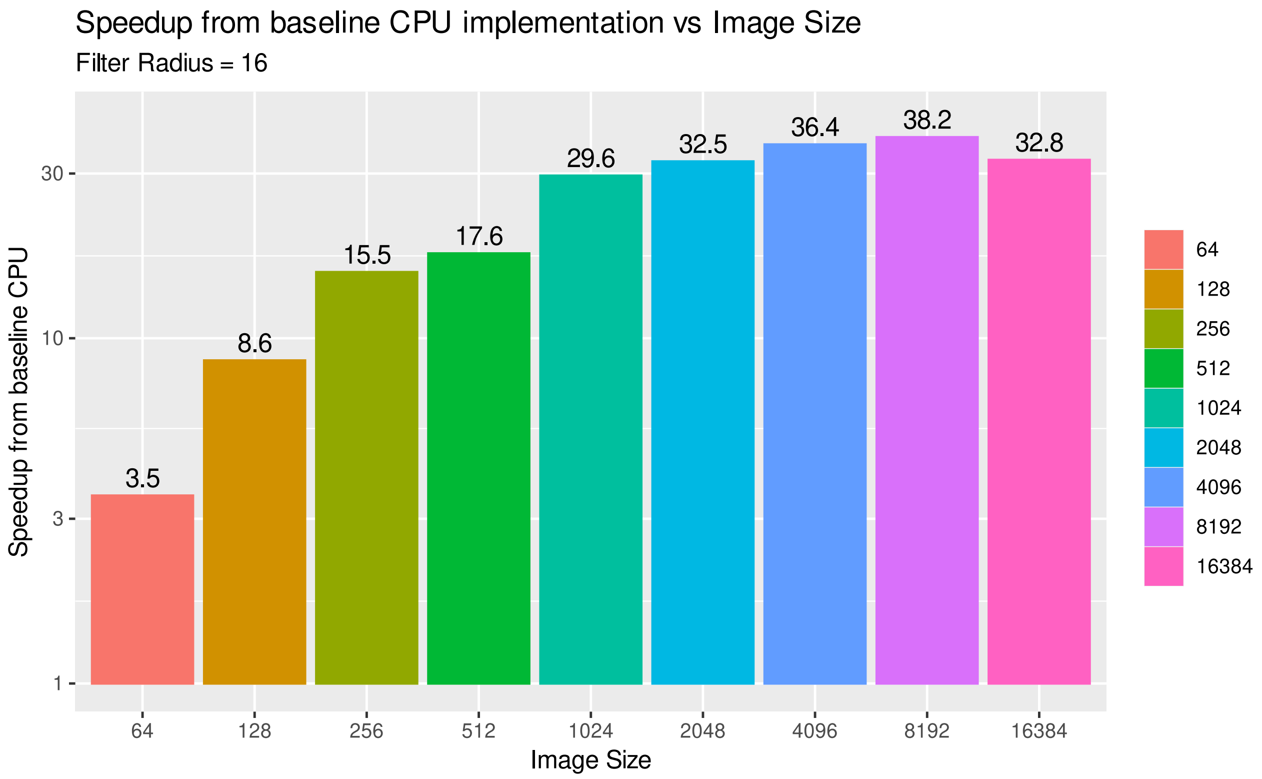

Figure 23: Speedup for GPU implementation over CPU baseline implementation per image size