ECE415: High Performance Computing Systems
Lab3 - CUDA Implementation and Performance Analysis of Separable 2D Convolution Filters

Contents

0 Device Query **2**

1 Compiler Configuration **4**

2 Single Block Kernels **4**
2.1 Memory Allocation . . . 4
2.2 Data transfer Host to Device . . . 6
2.3 GPU Kernels Implementation . . . 6
2.3.1 Kernel Launch . . . 7
2.4 Correctness Verification . . . 8
2.4.1 CPU Timing Measurement . . . 9
2.4.2 Memory Cleanup and Device Reset . . . 9
2.4.3 Accuracy Threshold Definition . . . 9
2.4.4 Comparison Implementation . . . 9
2.5 Comparison Algorithm . . . 10
2.5.1 Why Accuracy Tolerance is Necessary . . . 10

3 Maximum Image Size **11**
3.1 Maximum Numerical Accuracy . . . 11

4 Multiple Blocks Kernels **12**

5 Maximum Numerical Accuracy **14**
5.1 CPU vs GPU Execution Time . . . 15

6 Implementation Using Doubles **16**
6.1 Maximum Numerical Accuracy . . . 16
6.2 CPU vs GPU Execution Time . . . 18

7 Theoretical Questions **18**
7.1 Number of Memory Reads per Element . . . 18
7.2 Memory Accesses vs Floating-Point Operations . . . 20

8 Adding Padding (Implementation with Floats) **21**
8.1 CPU vs GPU Execution Time . . . 23
8.2 Comparison to Non-Padded Version . . . 24

Introduction

Convolution is a fundamental operation widely used in engineering and mathematics particularly in image processing where many transformation filters such as Gaussian blur, edge detection and sharpening are implemented as convolutions. Mathematically convolution quantifies the result of "overlaying" two functions through point wise multiplication and integration. In the discrete domain a 1D convolution is expressed as

$$r(i) = (s * k)(i) = \sum_n s(i-n) \cdot k(n),$$

which extends to 2D as

$$r(i,j) = (s * k)(i,j) = \sum_m \sum_n s(i-n, j-m) \cdot k(n,m).$$

In image processing this corresponds to computing a weighted sum of pixel values within a sliding window centered at each output pixel. A 2D convolution filter of size $n \times m$ typically requires $n \cdot m$ multiplications per output pixel. Separable filters can be decomposed into two 1D convolutions, one applied along rows and one along columns significantly reducing computational complexity. For example, the Sobel filter can be expressed as the composition of a vertical filter $\begin{bmatrix}1\\2\\1\end{bmatrix}$ followed by a horizontal filter $\begin{bmatrix}-1 & 0 & 1\end{bmatrix}$. In this assignment we implement such a separable 2D filter on a GPU where each thread computes one output pixel by first performing row-wise convolution, storing intermediate results in a buffer and then applying column-wise convolution to produce the final output.

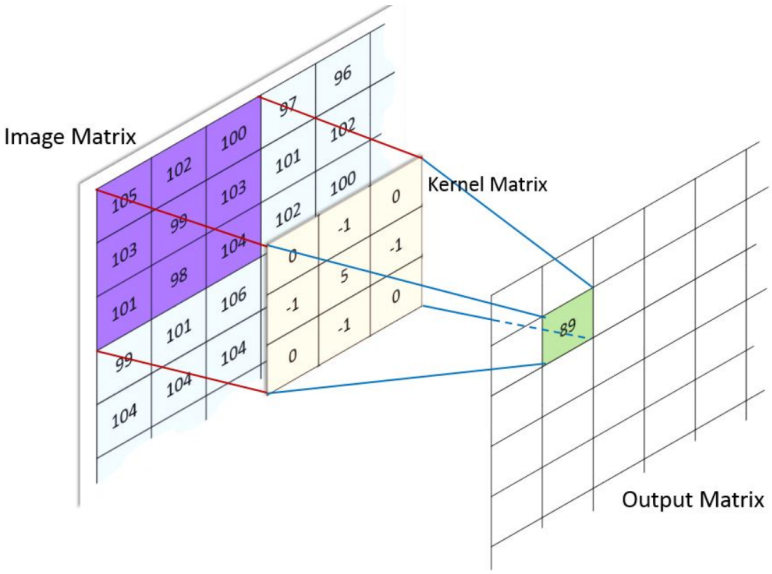

Figure 1: Illustration of a 2D convolution with a 3×3 mask applied on an input image to produce one output pixel.

0 Device Query

Before running the convolution experiments we executed the NVIDIA `deviceQuery` program to verify the CUDA installation and obtain detailed information about the available GPUs. Part of the complete output referring to one GPU can after compiling and running `./deviceQuery` is shown below:

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 4 CUDA Capable device(s)
```

```
Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
   (11997020160 bytes)
  (013) Multiprocessors, (192) CUDA Cores/MP:    2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536,
    65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
   layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
   2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        114688 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine
   (s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with
   device simultaneously) >
```

Listing 1: Output of deviceQuery on csl-venus system

The hardware characteristics that are relevant to our convolution implementation are:

- A maximum of 1 024 threads per block which limits the image size when using a single block configuration.
- 11441 MBytes of global memory
- A warp size of 32 which affects thread divergence
- 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)

1 Compiler Configuration

For compiling the CUDA code we use the NVIDIA CUDA Compiler (`nvcc`) with maximum optimization level for the host (CPU) code. The relevant Makefile configuration is shown below:

```
# Compiler
NVCC := nvcc

# Host (CPU) compiler flags
HOSTFLAGS := -O4 -I
```

Listing 2: Makefile Compiler Configuration

2 Single Block Kernels

The provided CPU code implements a separable 2D convolution through the following steps:

1. **Input Parameters**: The user provides the filter radius and image size (power of 2 greater than `FILTER_LENGTH`).
2. **Memory Allocation**: Arrays are allocated for the filter, input image, intermediate buffer and output.
3. **Data Initialization**: The filter is initialized with random integer values (0-15) and the input image with random floating point values.
4. **Convolution Computation**:
   - Row-wise convolution: `convolutionRowCPU()`
   - Column-wise convolution: `convolutionColumnCPU()`
5. **Deallocation**: All dynamically allocated memory is freed.

The mathematical formulation for the row-wise convolution is:

$$h\_Buffer[y \cdot imageW + x] = \sum_{k=-filterR}^{filterR} h\_Input[y \cdot imageW + (x+k)] \cdot h\_Filter[filterR - k] \quad (1)$$

And for column-wise convolution:

$$h\_Output[y \cdot imageW + x] = \sum_{k=-filterR}^{filterR} h\_Buffer[(y+k) \cdot imageW + x] \cdot h\_Filter[filterR - k] \quad (2)$$

Both operations include boundary checks to ensure indices remain within the image bounds.

2.1 Memory Allocation

Device arrays were created corresponding to the host arrays:

```
// Device arrays declaration
float *d_Filter, *d_Input, *d_Buffer, *d_OutputGPU;

// Error checking variables for each allocation
cudaError_t err_filter, err_Input, err_Buffer, err_OuputGPU;

// Allocate memory for filter (1D array)
err_filter = cudaMalloc((void **)&d_Filter,
                        FILTER_LENGTH * sizeof(float));

// Allocate memory for input image (NxN)
err_Input = cudaMalloc((void **)&d_Input,
                       imageW * imageH * sizeof(float));

// Allocate memory for intermediate buffer (NxN)
err_Buffer = cudaMalloc((void **)&d_Buffer,
                        imageW * imageH * sizeof(float));
```

```
18
19 // Allocate memory for output image (NxN)
20 err_OuputGPU = cudaMalloc((void **)&d_OutputGPU,
21                           imageW * imageH * sizeof(float));
```

Listing 3: Device Memory Allocation with Error Checking

Explanation:

- `d_Filter`: Stores the 1D convolution filter of length `FILTER_LENGTH = 2 * filter_radius + 1`
- `d_Input`: Stores the input image of size `imageW × imageH`
- `d_Buffer`: Stores intermediate results after row-wise convolution
- `d_OutputGPU`: Stores the final convolution result
- Each `cudaMalloc` call returns an error code stored for verification

```
1  // Check if all allocations succeeded
2  if (!IS_CUDA_MALLOC_OK(err_filter) ||
3      !IS_CUDA_MALLOC_OK(err_Input) ||
4      !IS_CUDA_MALLOC_OK(err_Buffer) ||
5      !IS_CUDA_MALLOC_OK(err_OuputGPU)) {
6
7      printf("CUDA: Memory allocation Failed!\n");
8
9      // Free host memory
10     FREE_MEM(h_Filter);
11     FREE_MEM(h_Input);
12     FREE_MEM(h_Buffer);
13     FREE_MEM(h_OutputCPU);
14     FREE_MEM(h_OutputGPU);
15
16     // Free successfully allocated device memory
17     IS_CUDA_MALLOC_OK(err_filter) ? cudaFree(d_Filter) : 0;
18     IS_CUDA_MALLOC_OK(err_Input) ? cudaFree(d_Input) : 0;
19     IS_CUDA_MALLOC_OK(err_Buffer) ? cudaFree(d_Buffer) : 0;
20     IS_CUDA_MALLOC_OK(err_OuputGPU) ? cudaFree(d_OutputGPU) : 0;
21
22     return 1;
23 }
```

Listing 4: Checking Memory Allocation Success

The macro `IS_CUDA_MALLOC_OK` is defined as:

```
1 #define IS_CUDA_MALLOC_OK(err) (((err) == (cudaSuccess)) ? (1) : (0))
```

Listing 5: Memory Allocation Check Macro

Memory Deallocation: At program termination all allocated memory is properly released:

```
1  // Free all host memory
2  free(h_OutputCPU);
3  free(h_Buffer);
4  free(h_Input);
5  free(h_Filter);
6  free(h_OutputGPU);
7
8  // Free all device memory
9  cudaFree(d_OutputGPU);
10 cudaFree(d_Buffer);
11 cudaFree(d_Input);
12 cudaFree(d_Filter);
13
14 // Reset device to clean state
15 cudaDeviceReset();
```

Listing 6: Memory Deallocation at Program End

Explanation:

- All host arrays are freed using `free()`
- All device arrays are freed using `cudaFree()`
- `cudaDeviceReset()` ensures the GPU returns to a clean state
- This cleanup occurs regardless of whether the program completed successfully or encountered errors

2.2 Data transfer Host to Device

Host to Device Transfer: After initializing the input image and filter on the host data is transferred to the device:

```
// Transfer input image from host to device
err_runtime = cudaMemcpy((float *)d_Input,
                         (float *)h_Input,
                         imageW * imageH * sizeof(float),
                         cudaMemcpyHostToDevice);
CHECK_CUDA_FAIL(err_runtime);

// Transfer filter from host to device
err_runtime = cudaMemcpy((float *)d_Filter,
                         (float *)h_Filter,
                         FILTER_LENGTH * sizeof(float),
                         cudaMemcpyHostToDevice);
CHECK_CUDA_FAIL(err_runtime);
```

Listing 7: Transferring Input Data to Device

Explanation:

- `cudaMemcpy` copies data from host memory to device global memory
- `cudaMemcpyHostToDevice` specifies the direction of transfer
- Each transfer is checked for errors using the `CHECK_CUDA_FAIL` macro
- The input image (`imageW × imageH` floats) and filter (`FILTER_LENGTH` floats) are transferred

Device to Host Transfer: After kernel execution results are copied back to the host:

```
// Transfer output image from device to host
err_runtime = cudaMemcpy((float *)h_OutputGPU,
                         (float *)d_OutputGPU,
                         imageW * imageH * sizeof(float),
                         cudaMemcpyDeviceToHost);
CHECK_CUDA_FAIL(err_runtime);
```

Listing 8: Transferring Results Back to Host

Explanation:

- `cudaMemcpyDeviceToHost` specifies transfer from device to host
- Results are stored in `h_OutputGPU` for comparison with CPU results
- Error checking ensures the transfer completed successfully

2.3 GPU Kernels Implementation

Since the convolution is separable it is implemented as two sequential 1D convolutions.

GPU Kernel Implementations: Row-wise Convolution Kernel:

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src,
                                  float *d_Filter, int imageW,
                                  int imageH, int filterR) {
    // Get thread position
    int x = threadIdx.x;
    int y = threadIdx.y;

    float sum = 0;

    // Convolve along the row
    for (int k = -filterR; k <= filterR; k++) {
        int d = x + k;

        // Boundary check
        if (d >= 0 && d < imageW) {
            sum += d_Src[y * imageW + d] * d_Filter[filterR - k];
        }
    }

    // Write result
    d_Dst[y * imageW + x] = sum;
}
```

Listing 9: Row-wise Convolution Kernel

Column-wise Convolution Kernel:

```
__global__ void convolutionColumnGPU(float *d_Dst, float *d_Src,
                                     float *d_Filter, int imageW,
                                     int imageH, int filterR) {
    // Get thread position
    int x = threadIdx.x;
    int y = threadIdx.y;

    float sum = 0;

    // Convolve along the column
    for (int k = -filterR; k <= filterR; k++) {
        int d = y + k;

        // Boundary check
        if (d >= 0 && d < imageH) {
            sum += d_Src[d * imageW + x] * d_Filter[filterR - k];
        }
    }

    // Write result
    d_Dst[y * imageW + x] = sum;
}
```

Listing 10: Column-wise Convolution Kernel

Kernel Design Explanation:

- Each thread computes exactly one output pixel
- Thread position is determined by `threadIdx.x` and `threadIdx.y`
- Row-wise kernel: iterates horizontally (varying x-coordinate)
- Column-wise kernel: iterates vertically (varying y-coordinate)
- Boundary checks prevent out of bounds memory accesses
- Filter is applied symmetrically around each pixel

2.3.1 Kernel Launch

A single block is used with dimensions matching the image size:

```
// Configure block dimensions (imageW x imageH threads)
dim3 block_dim(imageW, imageH);

// Launch row-wise convolution kernel (1 block)
convolutionRowGPU<<<1, block_dim>>>(d_Buffer, d_Input, d_Filter,
                                    imageW, imageH, filter_radius);

// Wait for kernel to complete
cudaDeviceSynchronize();

// Check for kernel execution errors
err_runtime = cudaGetLastError();
CHECK_CUDA_FAIL(err_runtime);

// Launch column-wise convolution kernel (1 block)
convolutionColumnGPU<<<1, block_dim>>>(d_OutputGPU, d_Buffer, d_Filter,
                                       imageW, imageH, filter_radius);

// Wait for kernel to complete
cudaDeviceSynchronize();

// Check for kernel execution errors
err_runtime = cudaGetLastError();
CHECK_CUDA_FAIL(err_runtime);
```

Listing 11: Kernel Execution with Single Block

Launch Configuration Details:

- **Grid dimension:** 1 block (as required by the assignment)
- **Block dimension:** `imageW` × `imageH` threads
- **Thread to pixel mapping:** One to one correspondence
- **Synchronization:** `cudaDeviceSynchronize()` ensures kernel completion before proceeding
- **Error checking:** `cudaGetLastError()` captures any kernel launch or execution errors

Execution Flow:

1. Row-wise convolution: `d_Input` → `d_Buffer`
2. Synchronization and error check
3. Column-wise convolution: `d_Buffer` → `d_OutputGPU`
4. Synchronization and error check

2.4 Correctness Verification

The GPU output is compared element-by-element with the CPU reference output:

```
// Accuracy threshold
#define accuracy 0.00005

// Compare outputs
int stop = 0;
for (i = 0; i < imageW; i++) {
    for (j = 0; j < imageH; j++) {
        if (abs(*(h_OutputCPU+(i*imageW + j)) -
                *(h_OutputGPU+(i*imageW + j))) > accuracy) {
            printf("CPU and GPU output differs!\n");
            stop = 1;
            break;
        }
    }
    if (stop) break;
}
```

Listing 12: CPU-GPU Output Comparison

This implements the requirement from Question 2b:

- Compares each element with tolerance `accuracy = 0.00005`
- Terminates comparison on first mismatch (early exit)
- Prints appropriate error message if discrepancy found

2.4.1 CPU Timing Measurement

CPU execution time is measured using `clock_gettime`:

```
struct timespec tv1, tv2;

printf("CPU computation...\n");
clock_gettime(CLOCK_MONOTONIC_RAW, &tv1);

convolutionRowCPU(h_Buffer, h_Input, h_Filter,
                  imageW, imageH, filter_radius);
convolutionColumnCPU(h_OutputCPU, h_Buffer, h_Filter,
                     imageW, imageH, filter_radius);

clock_gettime(CLOCK_MONOTONIC_RAW, &tv2);

printf("CPU: Total time = %10g seconds\n",
       (double)(tv2.tv_nsec - tv1.tv_nsec) / 1000000000.0 +
       (double)(tv2.tv_sec - tv1.tv_sec));
```

Listing 13: CPU Time Measurement

This timing will be used for performance comparisons in later questions.

2.4.2 Memory Cleanup and Device Reset

At program termination, all resources are properly released:

```
// Free host memory
free(h_OutputCPU);
free(h_Buffer);
free(h_Input);
free(h_Filter);
free(h_OutputGPU);

// Free device memory
cudaFree(d_OutputGPU);
cudaFree(d_Buffer);
cudaFree(d_Input);
cudaFree(d_Filter);

// Reset device
cudaDeviceReset();
```

Listing 14: Memory Cleanup

The `cudaDeviceReset()` call ensures the GPU returns to a clean state, which is particularly important when running multiple experiments or if the program terminated due to an error.

2.4.3 Accuracy Threshold Definition

```
#define accuracy 0.00005
```

Listing 15: Accuracy Threshold

This threshold allows for small floating-point rounding differences between CPU and GPU computations while detecting significant errors.

2.4.4 Comparison Implementation

```
// Flag to break out of nested loops
int stop = 0;

// Compare each element
for (i = 0; i < imageW; i++) {
```

```
    for (j = 0; j < imageH; j++) {
        // Calculate absolute difference
        float diff = abs(*(h_OutputCPU + (i*imageW + j)) -
                         *(h_OutputGPU + (i*imageW + j)));

        // Check if difference exceeds tolerance
        if (diff > accuracy) {
            printf("CPU and GPU output differs!\n");
            printf("Position: (%d, %d)\n", i, j);
            printf("CPU value: %f, GPU value: %f\n",
                   *(h_OutputCPU + (i*imageW + j)),
                   *(h_OutputGPU + (i*imageW + j)));
            printf("Difference: %f (tolerance: %f)\n", diff, accuracy);
            stop = 1;
            break;
        }
    }
    if (stop) break;
}

// Print success message if all elements match
if (!stop) {
    printf("SUCCESS: CPU and GPU outputs match within tolerance!\n");
}
```

Listing 16: CPU-GPU Output Comparison with Early Termination

2.5 Comparison Algorithm

Step-by-step process:

1. Iterate through all pixels in row-major order
2. For each pixel at position (i, j):
   - Calculate absolute difference between CPU and GPU values
   - Compare difference against accuracy threshold
   - If difference exceeds threshold:
     - Print error message
     - Print position of mismatch
     - Print both CPU and GPU values
     - Print the actual difference
     - Set stop flag and break from inner loop
3. Check stop flag after inner loop; if set, break from outer loop
4. If no mismatches found, print success message

Key Features:

- **Early termination:** Stops immediately upon finding first mismatch (as required)
- **Informative output:** Provides detailed information about the mismatch location and values
- **Tolerance-based:** Accounts for floating-point precision differences
- **Complete verification:** Checks every element if no errors found

2.5.1 Why Accuracy Tolerance is Necessary

Floating-point arithmetic on CPU and GPU may produce slightly different results due to:

- Different rounding modes
- Different order of operations

- Different hardware implementations
- Accumulation of rounding errors in summations

The tolerance of `0.00005` (5 decimal places) allows for these minor differences while still detecting genuine computational errors.

3 Maximum Image Size

According to deviceQuery each GPU block can contain at most 1024 threads. If we assign one thread per pixel and the entire image to be processed by a single block then the image can have at most 1024 pixels. For a square image with width W and height W, the number of pixels is

$$W^2 = 1024$$

Solving this gives

$$W = 32.$$

So the largest square image that can be processed by a single block is 32×32 pixels. That was also verified, using incrementally image sizes of the power of 2 and getting a `CUDA Error` at image size 64x64.

3.1 Maximum Numerical Accuracy

For the maximum image size (32x32) supported by our implementation, the maximum achievable accuracy without comparison errors depends strongly on the filter size (radius). Based on the experimental results we observe that as the filter radius increases the maximum possible accuracy (in decimal digits) decreases rapidly due to the accumulation of floating point rounding errors in the convolution operations. More specifically:

- For radius $r = 1$: up to **2 decimal digits** of accuracy without comparison errors.
- For radii $r = 2$–3: up to **1 decimal digit** of accuracy.
- For radii $r \geq 4$: effectively **0 decimal digits** since the observed `max_diff` values are large and comparison errors appear.

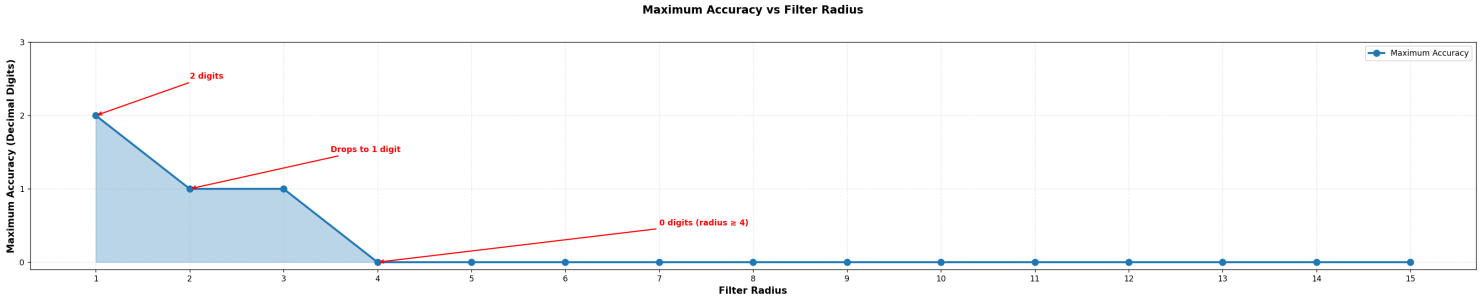

Figure 2: Maximum Accuracy vs Filter Radius

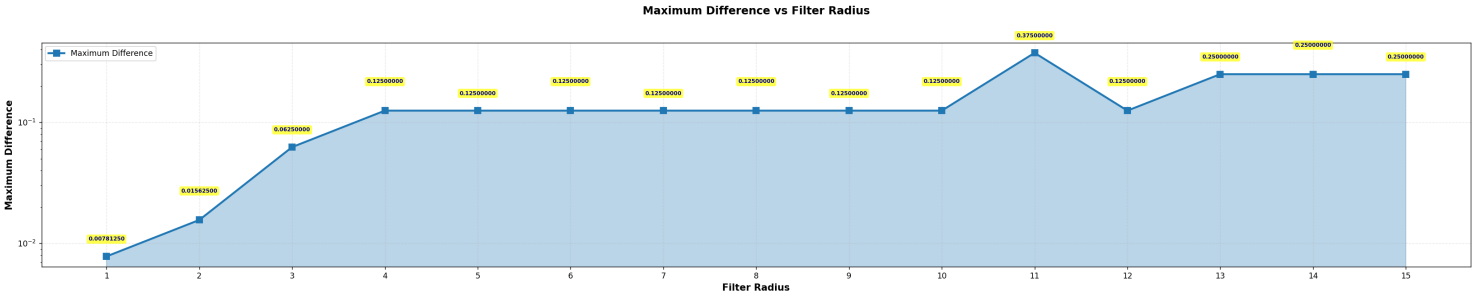

Figure 3: Maximum Difference vs Filter Radius

4 Multiple Blocks Kernels

The limitation of a single block (maximum 1024 threads) can be overcome by using multiple blocks. Our strategy is to tile the image with blocks of size 32×32 where each pixel is still handled by a single thread. If the image size (width and height) is at most 32 (`imageSize`) ≤ 32 the entire image fits into a single block since $32 \times 32 = 1024$ threads, which is the maximum. If the image size is greater than 32 we must use multiple blocks. We define:

- `imageSize`: the width (and height) of the square image
- `maxBlockLength` $= 32$: the maximum number of threads per block dimension
- `gridLength`: the number of blocks along each dimension (x and y)

For a square image we choose:

$$\text{gridLength} = \frac{\text{imageSize}}{\text{maxBlockLength}}$$

For example if the image size is 64 (the next power of two after 32) then

$$\text{gridLength} = \frac{64}{32} = 2$$

which means we launch a grid of 2×2 blocks for a total of

$$\text{gridArea} = \text{gridLength}^2 = 2^2 = 4$$

blocks. which means we launch a grid of 2×2 blocks for a total of 4 blocks.

In other words, we take the image size and determine how many 32×32 thread blocks are needed along each dimension by computing `imageSize / maxBlockLength`. If this ratio is greater than 1 (2, 3, 4, ...) its value directly gives the grid length meaning the number of blocks in both x and y directions. This allows us to scale beyond the 1024 thread limit of a single block while still assigning one thread per pixel.

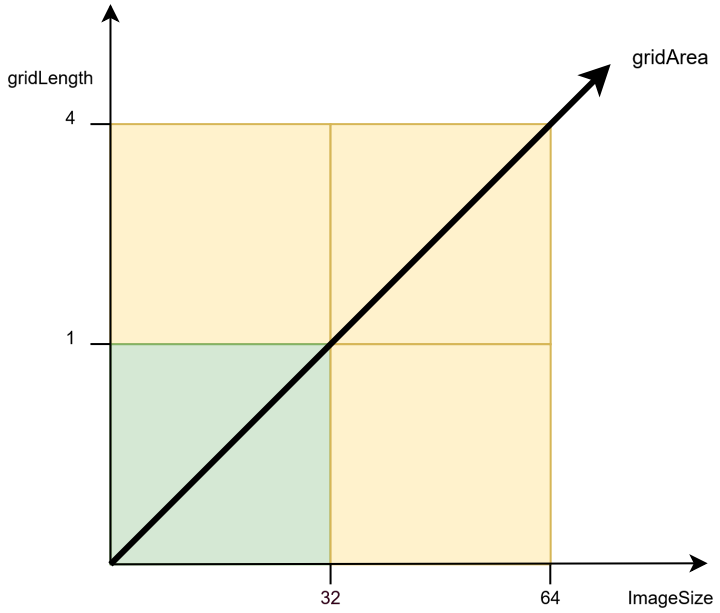

Figure 4: Image size to gridArea relationship

In general for a square image of side `imageSize` the grid area is:

$$\text{gridArea} = \text{gridLength}^2 = \left(\frac{\text{imageSize}}{\text{maxBlockLength}}\right)^2.$$

As the image size increases the total number of blocks (`gridArea`) grows with the square of the image size.

Now for the maximum supported image size: According to `deviceQuery` our GPU is a Tesla K80 with the following relevant characteristics:

- Total global memory:

$$11{,}441 \text{ MB} = 11.996.758.160 \text{ bytes} \approx 11{,}441 \text{ MB} \approx 11.17 \text{ GB}.$$

$$\text{Max grid size in } x = 2.147.483.647 \qquad \text{Max grid size in } y = 65.535$$

- Maximum number of threads per block: 1024.

We store the image in single precision floating point so each pixel requires

$$\text{bytes per pixel} = 4.$$

If we ignore all other allocations and consider only the image allocation then the maximum number of pixels that can fit in global memory is

$$N_{\text{pixels,max}} = \frac{\text{global memory (bytes)}}{\text{bytes per pixel}} = \frac{11{,}996{,}758{,}160}{4} = 2.999.189.540 \approx 2.999 \times 10^9 \text{ pixels}$$

For a square image of side length W, we have W^2 pixels. Therefore

$$W^2 \leq N_{\text{pixels,max}} \quad \Longrightarrow \quad W \leq \sqrt{2.999 \times 10^9} \approx 54.764 pixels$$

The maximum theoretical side length for a single floating point square image on this GPU is:

$$W_{\text{max,mem}} = 54.764 pixels$$

In our case we need to store three images simultaneously in global memory (input, intermediate buffer and output), so the side length x of each image must satisfy:

$$3x < 54.764 \quad \Rightarrow \quad x < 18.254.7$$

Since we have to use images that have power of two image dimensions the largest valid size is

$$x = 2^{14} = 16.384$$

which is the maximum image dimension we can use under the given memory constraints of the GPU.

For the maximum image size and our block configuration of

$$\text{blockDim} = (32, 32)$$

we need:

$$\text{gridLength} = \frac{16.384}{32} = 512$$

blocks along each dimension. Therefore, the total grid area in our implementation is:

$$\text{gridArea} = \text{gridLength}^2 = 512^2 = 262.144 \text{ blocks}$$

The next power of two image size would be

$$W = 32.768$$

which exceeds the upper limit of 18.254.7 for each image.

5 Maximum Numerical Accuracy

For a 1024×1024 image we tested filter radious that are powers of 2 (from 1 to 256) and measured the maximum accuracy (in decimal digits) that leads to successful comparisons without errors. The results are shown shown bellow:

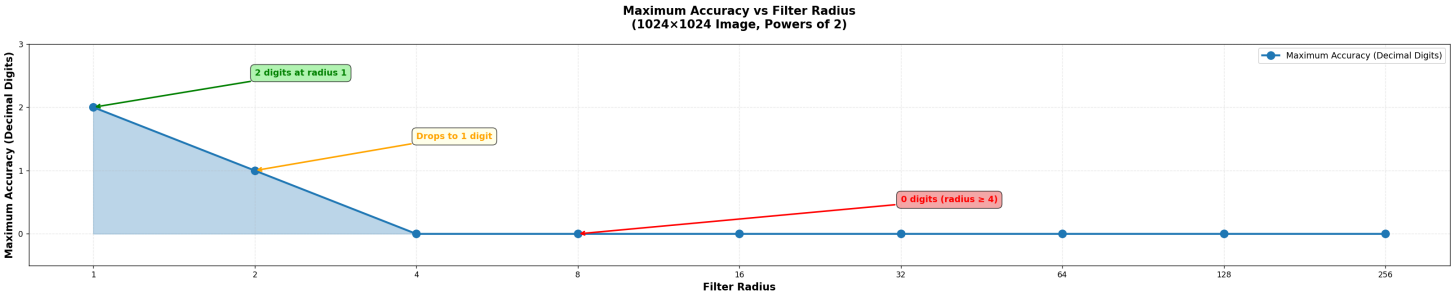

Figure 5: Maximum Accuracy vs Filter Radius

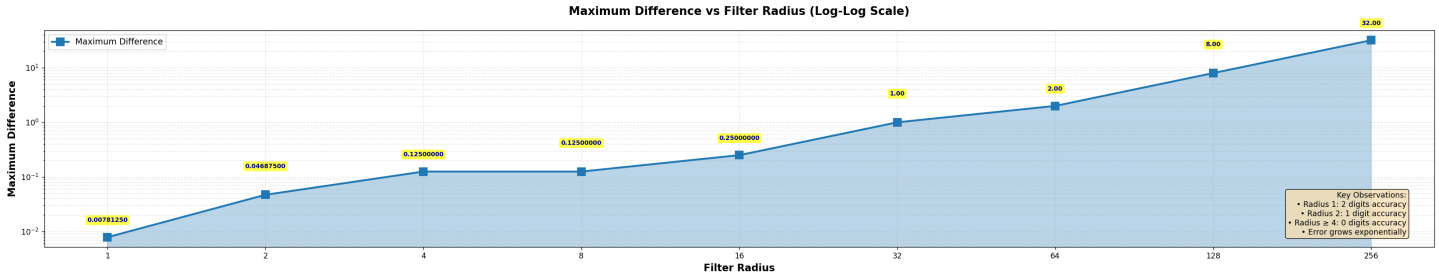

Figure 6: Maximum Difference vs Filter Radius

Observations:

- For radius $r = 1$: up to **2 decimal digits** of accuracy (max_diff = 0.0078125).
- For radius $r = 2$: up to **1 decimal digit** of accuracy (max_diff = 0.046875).

- For radii $r \geq 4$: effectively **0 decimal digits**, with max_diff growing exponentially from 0.125 (radius 4) to 32.0 (radius 256).

Why does this happen? The degradation in accuracy is due to the accumulation of floating point rounding errors during the convolution operations. For a separable 2D convolution with radius r each output pixel requires $2 \times (2r+1)$ multiply accumulate operations (one pass for rows and one for columns). As the filter radius increases:

1. The number of operations per pixel grows linearly with radius (for example the radius 256 requires 1026 operations per pixel).
2. Each floating point operation introduces a small rounding error (typically on the order of machine epsilon, $\approx 10^{-7}$ for single precision).
3. These errors accumulate with each operation leading to larger total errors.
4. The max_diff grows approximately exponentially with radius reaching 32.0 at radius 256, which means the GPU and CPU results can differ by up to 32 in pixel values.

This explains why only very small filters (radius 1-2) can maintain any meaningful accuracy while larger filters result in significant discrepancies between CPU and GPU computations.

5.1 CPU vs GPU Execution Time

Table 1: Measured execution times for separable convolution with filter radius $r = 16$ on images of size $N \times N$ in seconds. The GPU time includes host device and device host data transfers but not device memory allocation / deallocation.

Width N	**CPU [s]**	**GPU [s]**	**Speedup $T_{\mathbf{CPU}}/T_{\mathbf{GPU}}$**
64	0.00042386	0.00015594	2.72
128	0.00159320	0.00022195	7.18
256	0.00623298	0.00055987	11.14
512	0.02573890	0.00277843	9.27
1024	0.11552400	0.01002680	11.52
2048	0.33908700	0.03566720	9.51
4096	1.48560000	0.14163000	10.49
8192	7.75930000	0.60922800	12.74
16384	32.0799000	1.86726000	17.18

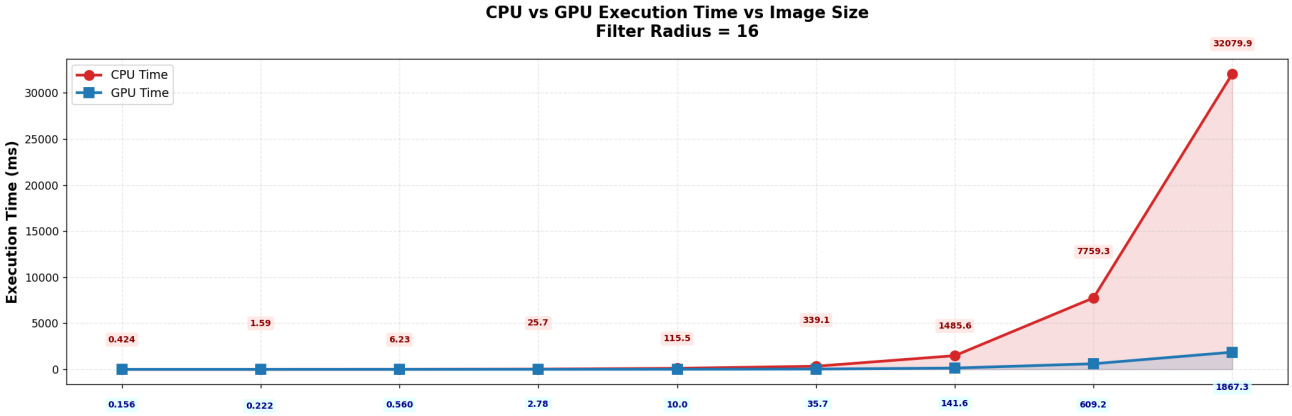

Figure 7: CPU vs GPU Execution time for different image sizes

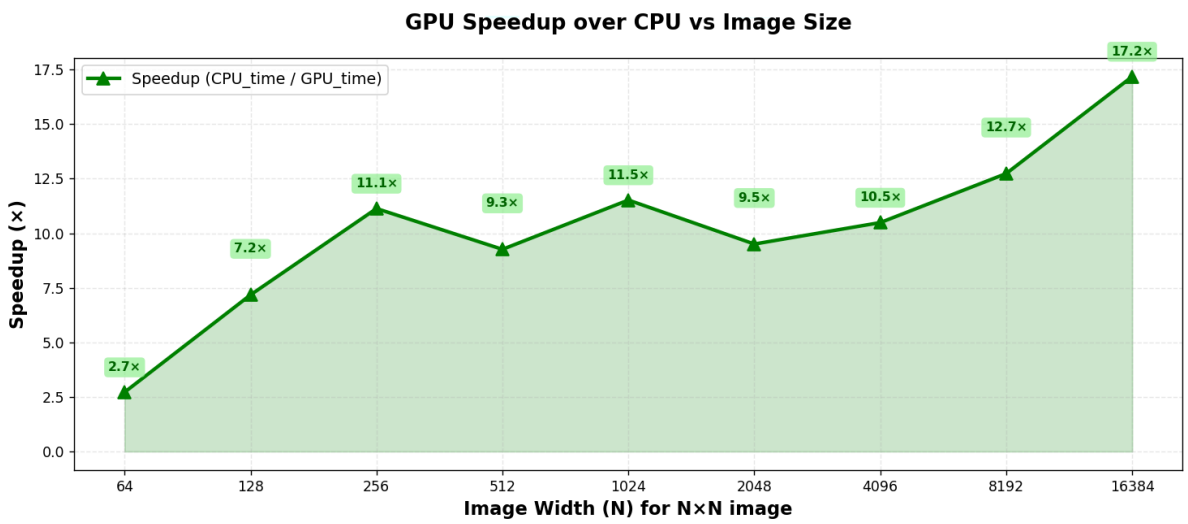

Figure 8: GPU Speedup vs CPU

6 Implementation Using Doubles

By switching from `float` (single precision) to `double` (double precision) we double the number of bits used to represent each value (from 32 to 64) and increase the available precision from about 7 decimal digits to about 15 decimal digits (see Table 2). We expect to see that the maximum absolute difference between the CPU and GPU results to be smaller and more stable (deviate less) as the problem size increases.

Table 2: Comparison of IEEE-754 `float` (single precision) and `double` (double precision).

Type	Bits	Approx. decimal digits
`float`	32	≈ 7
`double`	64	≈ 15

6.1 Maximum Numerical Accuracy

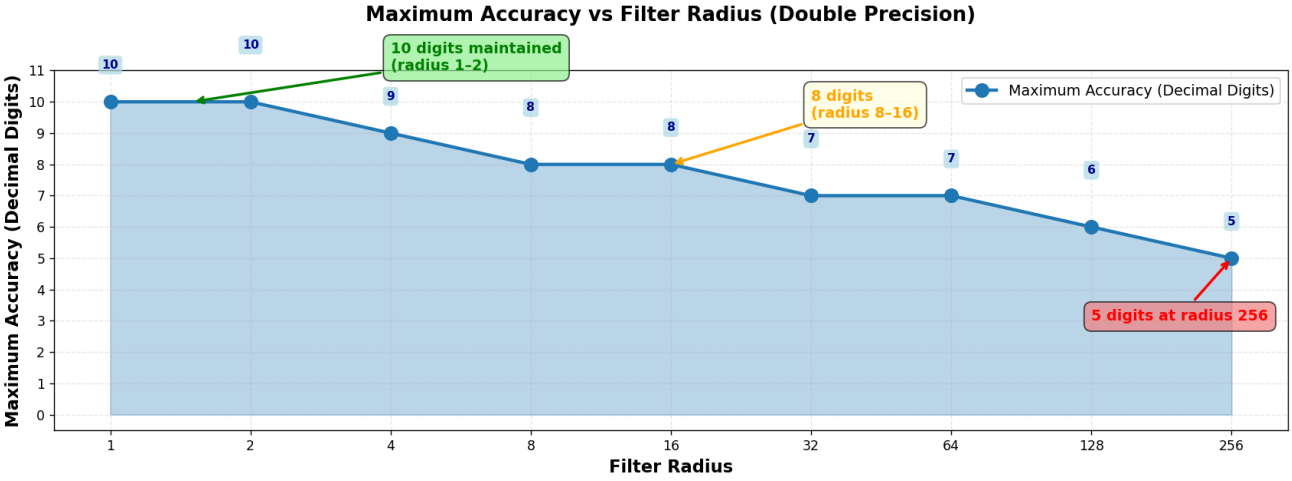

Figure 9: Maximum Accuracy in digits vs Radius

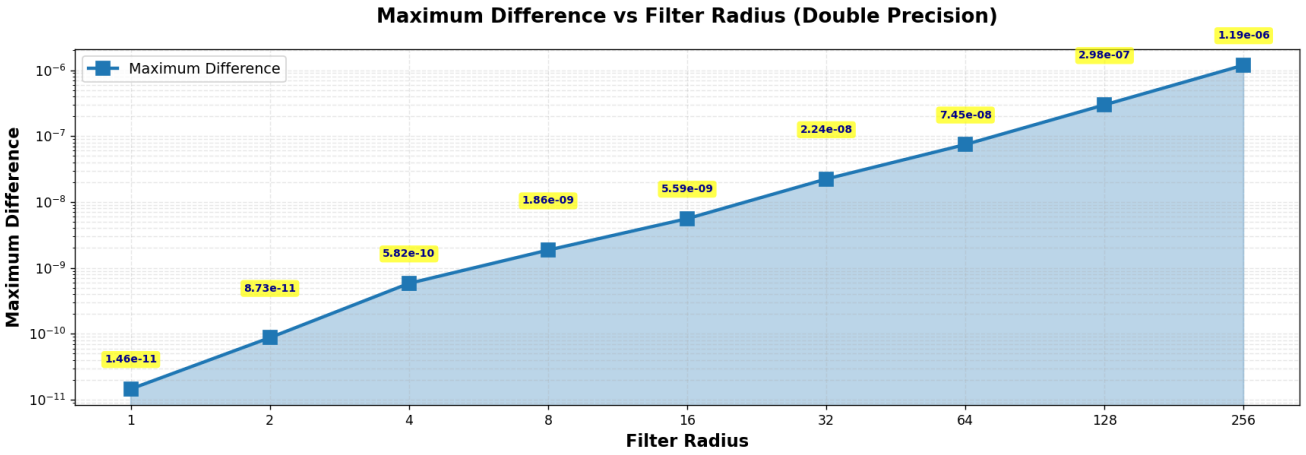

Figure 10: Maximum difference vs Filter Radious

This behaviour matches our expectations: since `double` provides about 15 decimal digits of precision (Table 2), rounding errors accumulate much more slowly and the CPU and GPU implementations produce results that are almost identical. The maximum difference between CPU and GPU results remains extremely small across all filter radius tested as shown in Table 3.

Table 3: Maximum difference and accuracy (decimal digits) for double precision as a function of filter radius (1024×1024 image).

Radius	**Digits**	**Max Difference**
1	10	1.46×10^{-11}
2	10	8.73×10^{-11}
4	9	5.82×10^{-10}
8	8	1.86×10^{-9}
16	8	5.59×10^{-9}
32	7	2.24×10^{-8}
64	7	7.45×10^{-8}
128	6	2.98×10^{-7}
256	5	1.19×10^{-6}

Even at the largest filter radius tested ($r = 256$, corresponding to a 512×512 filter kernel), the maximum absolute difference is only approximately 1.2×10^{-6}, and the results still agree to 5 decimal digits. This is in contrast to the single precision case where the maximum difference reached 32.0 and all decimal digit agreement was lost for radius ≥ 4. Switching from single to double precision makes the maximum CPU GPU difference negligible for all radius tested and show us that double precision is needed for accuracy in large convolution operations.

6.2 CPU vs GPU Execution Time

Table 4: Measured execution times for separable convolution with filter radius $r = 16$ on images of size $N \times N$ in seconds. The GPU time includes host device execution and device host data transfers but not device memory allocation or deallocation.

Width N	**CPU [s]**	**GPU [s]**	**Speedup** $T_{\text{CPU}}/T_{\text{GPU}}$
256	0.012710	0.001156	10.99
512	0.047282	0.003265	14.49
1024	0.162931	0.011865	13.72
2048	0.630413	0.044677	14.11
4096	2.396523	0.204671	11.70
8192	10.148014	0.683399	14.85
16384	41.550855	3.154930	13.17

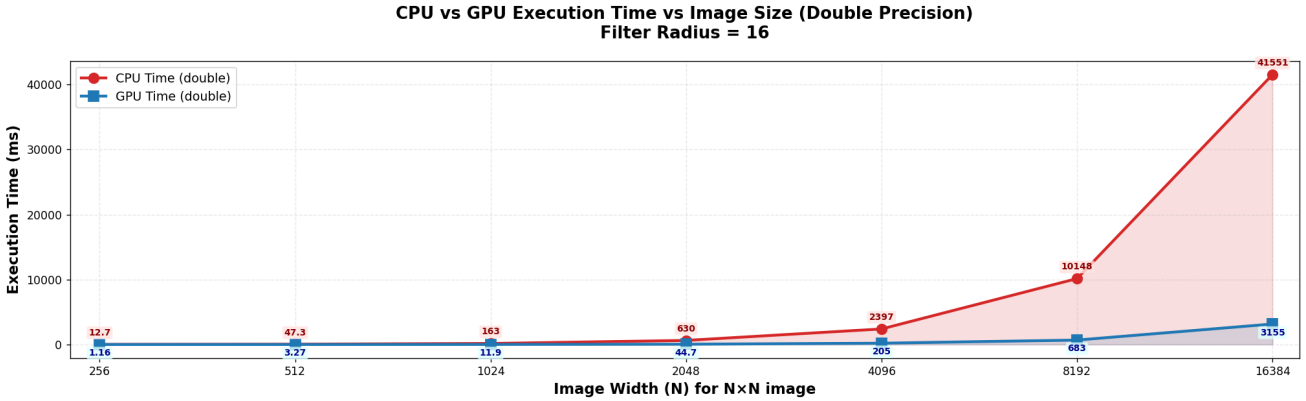

Figure 11: CPU vs GPU Execution time for different image sizes

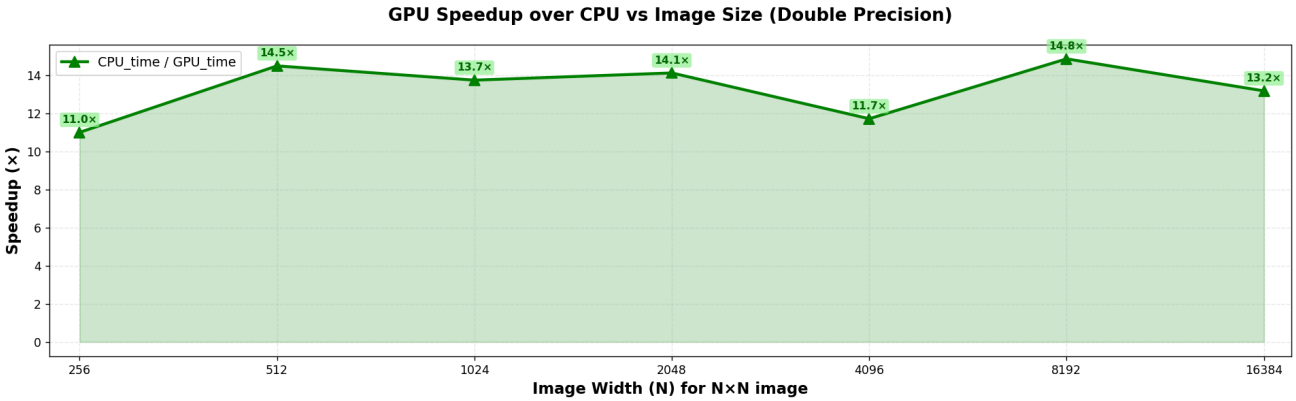

Figure 12: GPU Speedup vs CPU

7 Theoretical Questions

7.1 Number of Memory Reads per Element

We count the number of pixel accesses from d_Input and d_Buffer. Performing convolution in row wise order pixels are accessed $2 * filter\_radius + 1$ times.

Except the pixels that are near row edges. Pixels with index $x < filter\_radius$ are accessed $x + filter\_radius + 1$ times and pixels with index $x > N - filter\_radius$ (N = image width = image height) are accessed $(N - 1 - x) + filter\_radius + 1$ times. In row wise convolution all pixels in a column are accessed the same amount of times. The proccess is symmetric for column wise convolution. So the number of pixel (x, y) accesses is given by the following expression:

$$pixel\_accesses = \begin{cases} x + filter\_radius + 1, & \text{if } x < filter\_radius \\ 2 * filter\_radius + 1, & \text{if } filter\_radius < x < N - filter\_radius \\ (N - 1 - x) + filter\_radius + 1, & \text{if } x > N - filter\_radius \end{cases}$$

Where N = image width = image height

When calculating the new value of a pixel, it's neighboring filter_radius pixels from left and right are accessed for the calculation. But, near the edges the filter doesn't fit in the input array and we discard some calculations. This is the reason the pixels on the edges are accessed fewer times than the pixels near the center.

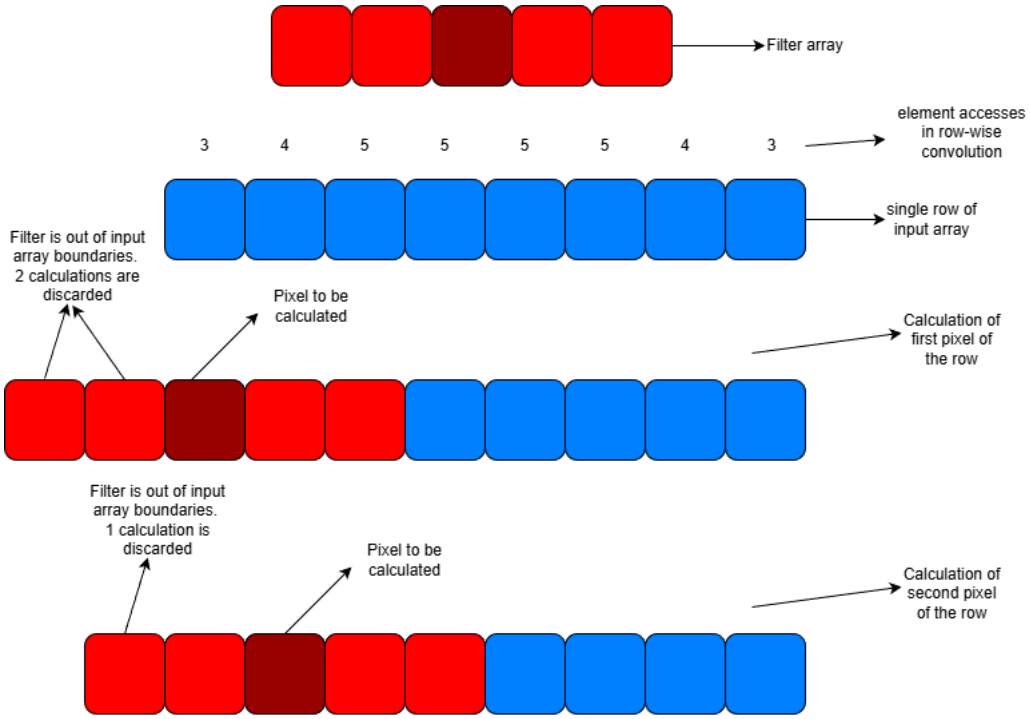

Figure 13: Example on difference of pixel accesses near edges

The elements of the filter array are accesses N times (where N is the dimensions of the square input array). Except for the elements that are located beyond the middle of the filter center. In simpler words elements that have index $x > filter\_radius + 1$. Beyond middle each element is decreased by 1. In mathematical terms it is accessed $N - (x - filter\_radius - 1)$ times. For row-wise and column-wise convolution the process is completely symmetric, so we multiply the result of row-wise convolution by 2 to find the total accesses.
Filter buffer accesses are expressed as:

$$2 * ((x < filter\_radius + 1)?N : N - (x - filter\_radius - 1))$$

Where x is the index on the filter.

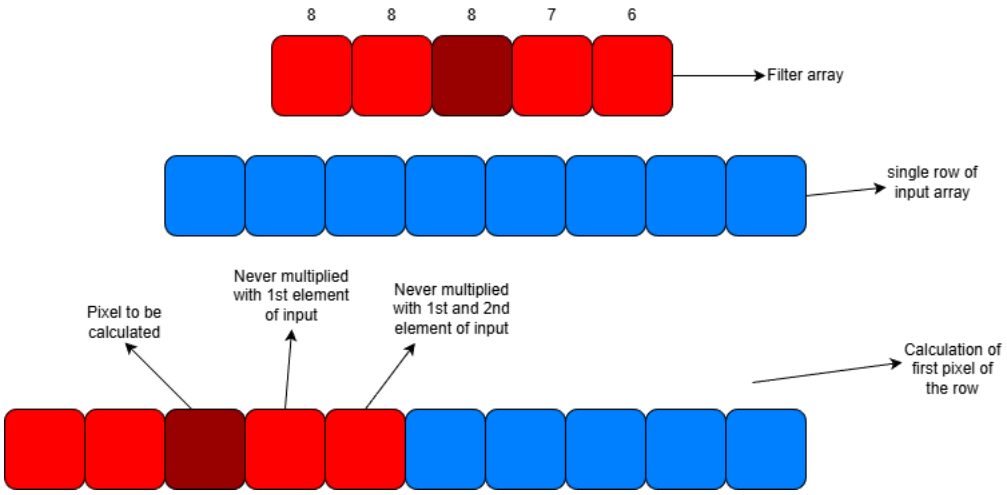

Figure 14: Example on difference of element accesses on the filter array

Below we provide an example with input dimensions N = 8 and filter_radius = 2. Inside the blocks are shown the total accesses of the elemets during execution of the 2D convolution algorithm on the GPU.

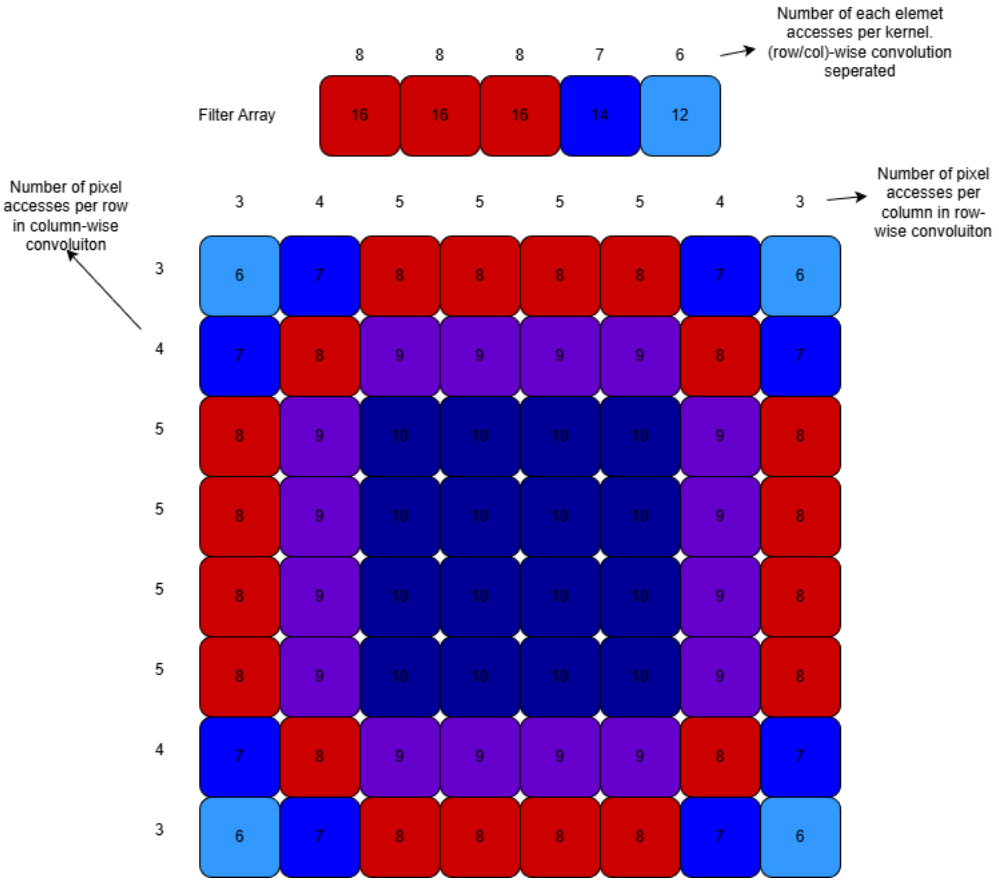

Figure 15: Example element accesses on input and filters arrays

7.2 Memory Accesses vs Floating-Point Operations

Reading the algorithm we wrote for GPU we notice that per pixel we have 2 global memory accesses (1 in d_Src and 1 in d_Filter arrays) and 2 floating point operations for calculating sum.

```
1       sum += d_Src[y * imageW + d] * d_Filter[filterR - k];
```

Per pixel the above code is calculated 2*filter_radius+1 times and there are N*N pixels. In total:

$$Ratio = \frac{\text{Total Global Memory Accesses}}{\text{Total FLOP}} = \frac{2 * N^2 * 2 * filter\_radius + 1}{2 * N^2 * 2 * filter\_radius + 1} = 1$$

8 Adding Padding (Implementation with Floats)

The divergence problem is caused from the if statement that checks if the filter exceeds the input array boundaries. To solve this problem we need to remove the if statement, this can be done by using the padding technic. For the first elemet of each row or col the filter exceed the input array by filter_radius elements. We can make the input buffer large by filter_radius elements in each direction and fill them with 0's so the output won't be affected. All arrays will need to be padded and initialized to 0's for the algorithm to work.

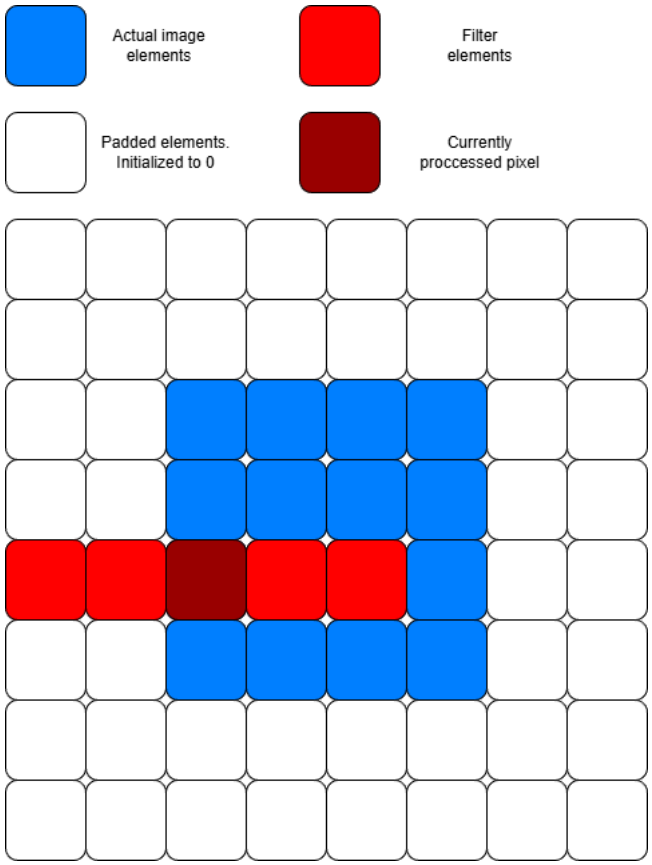

Figure 16: Padded Input array with dimensions $N = 4$, $filter\_radius = 2$

We applied padding both in GPU and CPU executions. (It was asked that we did mainly for the GPU, but the CPU had significant improvement time-wise). We allocate the input and output arrays using calloc(), so all arrays are initialized to zeroes.

```
1      h_Filter = (float *)malloc(FILTER_LENGTH * sizeof(float));
2      h_Input = (float *)calloc(paddedW * paddedH, sizeof(float));
3      h_Buffer = (float *)calloc(paddedW * paddedH, sizeof(float));
4      h_OutputCPU = (float *)calloc(paddedW * paddedH, sizeof(float));
5      h_OutputGPU = (float *)calloc(paddedW * paddedH, sizeof(float));
```

We copy them to the device arrays (including d_Buffer, so its padding is initialized to zeroes).

```
1    err_runtime = cudaMemcpy((float *)d_Input, (float *)h_Input, paddedW * paddedH *
     sizeof(float), cudaMemcpyHostToDevice);
2    CHECK_CUDA_FAIL(err_runtime);
3    err_runtime = cudaMemcpy((float *)d_Buffer, (float *)h_Buffer, paddedW * paddedH *
     sizeof(float), cudaMemcpyHostToDevice);
4    CHECK_CUDA_FAIL(err_runtime);
```

```
    err_runtime = cudaMemcpy((float *)d_Filter, (float *)h_Filter, FILTER_LENGTH *
    sizeof(float), cudaMemcpyHostToDevice);
    CHECK_CUDA_FAIL(err_runtime);
```

We fill only the image with the random floats.

```
    int index;
      for (i = filter_radius; i < imageW+filter_radius; i++) {
        for (j = filter_radius; j < imageH+filter_radius; j++) {
          // out side of image boundaries
          index = paddedW*i + j;
          if((i > filter_radius) && (i < (imageW+filter_radius)) \
              && (j > filter_radius) && (j < (imageH+filter_radius)))
            h_Input[index] = (float)rand() / ((float)RAND_MAX / 255) + (float)rand() / (
    float)RAND_MAX;
        }
      }
```

We also define an index as paddedW, that allows us to read and write in the correct positions, as we use padding.

Below is the code for GPU Row-wise Convolution:

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

  int x, y, k;
  x = threadIdx.x + blockIdx.x * blockDim.x + filterR;
  y = threadIdx.y + blockIdx.y * blockDim.y + filterR;
  int paddedW = 2*filterR + imageW;

  float sum = 0;
  for (k = -filterR; k <= filterR; k++) {
    int d = x + k;
    sum += d_Src[y * paddedW + d] * d_Filter[filterR - k];
  }

  d_Dst[y * paddedW + x] = sum;
}
```

Below is the code for GPU Column-wise Convolution:

```
__global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, float *d_Filter,
                                    int imageW, int imageH, int filterR) {
  int x, y, k;
  x = threadIdx.x + blockIdx.x * blockDim.x + filterR;
  y = threadIdx.y + blockIdx.y * blockDim.y + filterR;
  int paddedW = 2*filterR + imageW;

  float sum = 0;
  for (k = -filterR; k <= filterR; k++) {
    int d = y + k;
    sum += d_Src[d * paddedW + x] * d_Filter[filterR - k];
  }

  d_Dst[y * paddedW+ x] = sum;
}
```

The same technique was applied to the CPU execution:
Below is the code for CPU Row-wise Convolution:

```
void convolutionRowCPU(float *h_Dst, float *h_Src, float *h_Filter,
                       int imageW, int imageH, int filterR) {

  int x, y, k;
  int paddedW = 2*filterR+imageW;

  for (y = filterR; y < imageH+filterR; y++) {
    for (x = filterR; x < imageW+filterR; x++) {
      float sum = 0;

      for (k = -filterR; k <= filterR; k++) {
        int d = x + k;
        sum += h_Src[y * paddedW + d] * h_Filter[filterR - k];
```

```
      }

      h_Dst[y * paddedW + x] = sum;
    }
  }
}
```

Below is the code for CPU Column-wise Convolution:

```
void convolutionColumnCPU(float *h_Dst, float *h_Src, float *h_Filter,
              int imageW, int imageH, int filterR) {

  int x, y, k;
  int paddedW = 2*filterR+imageW;

  for (y = filterR; y < imageH+filterR; y++) {
    for (x = filterR; x < imageW+filterR; x++) {
      float sum = 0;

      for (k = -filterR; k <= filterR; k++) {
        int d = y + k;
        sum += h_Src[d * paddedW + x] * h_Filter[filterR - k];
      }

      h_Dst[y * paddedW + x] = sum;
    }
  }

}
```

8.1 CPU vs GPU Execution Time

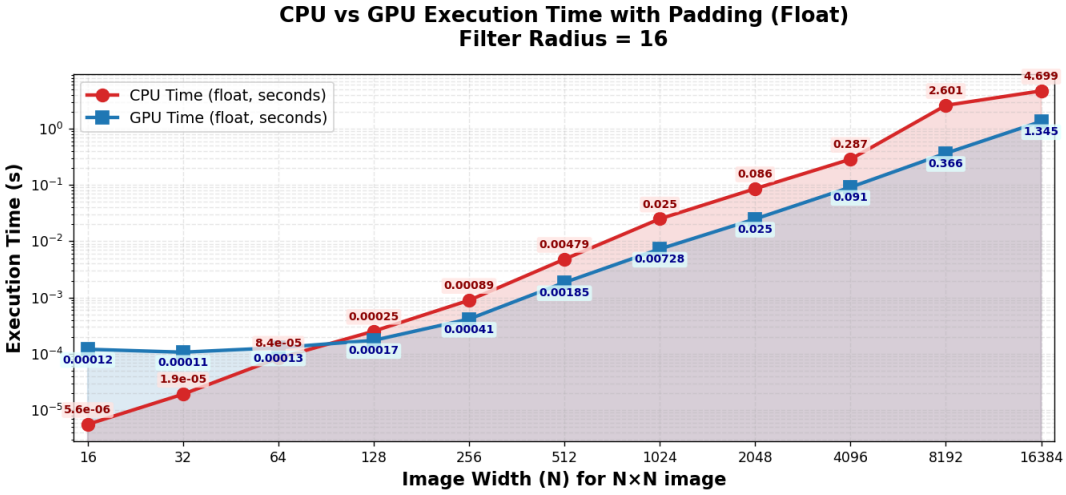

Figure 17: CPU VS GPU Padded version

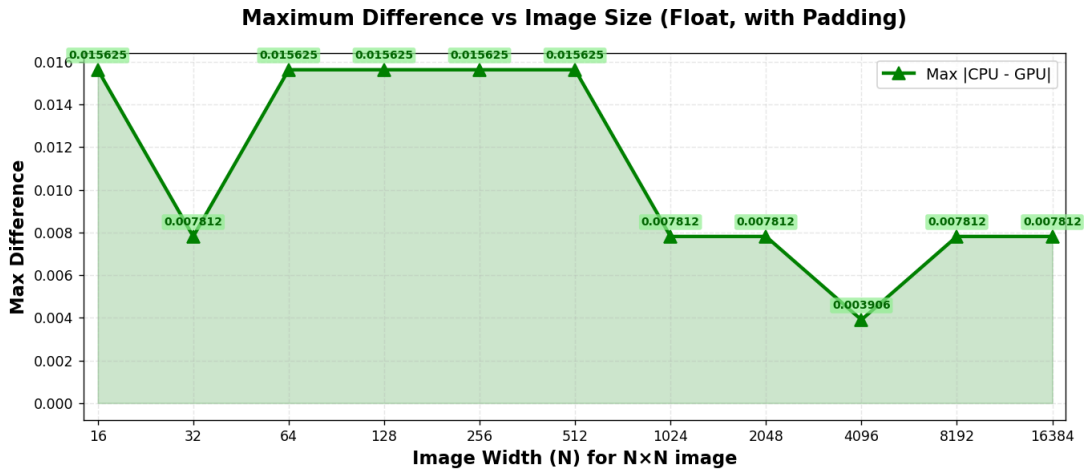

Figure 18: Maximum Difference vs Image Size

Table 5: Measured execution times for separable convolution with padding and filter radius $r = 16$ on images of size $N \times N$ in seconds. The GPU time includes host device and device host data transfers but not device memory allocation or deallocation.

Width N	**CPU [s]**	**GPU [s]**
16	5.572×10^{-6}	1.204×10^{-4}
32	1.916×10^{-5}	1.067×10^{-4}
64	8.379×10^{-5}	1.283×10^{-4}
128	2.522×10^{-4}	1.735×10^{-4}
256	8.929×10^{-4}	4.143×10^{-4}
512	4.788×10^{-3}	1.850×10^{-3}
1024	2.501×10^{-2}	7.277×10^{-3}
2048	8.586×10^{-2}	2.487×10^{-2}
4096	2.869×10^{-1}	9.118×10^{-2}
8192	2.601	3.660×10^{-1}
16384	4.699	1.345

8.2 Comparison to Non-Padded Version

When we compare the execution time between the non-padded and padded versions we notice that not only the algorithm gets a bit faster on the GPU, but also the CPU time becomes a lot faster.
The improvement in GPU and CPU is caused by different reasons. On the GPU the divergence problem is solved, so threads in the same warp execute the same instructions. Before padding same thread with in the warp could take different paths and execute different instructions. This caused threads to wait for the instruction in the warp, because each warp has one control unit and can change instruction every 2 clock cycles.
On the other side the CPU time gets significantly faster because removing the if statement results in less branch misses.

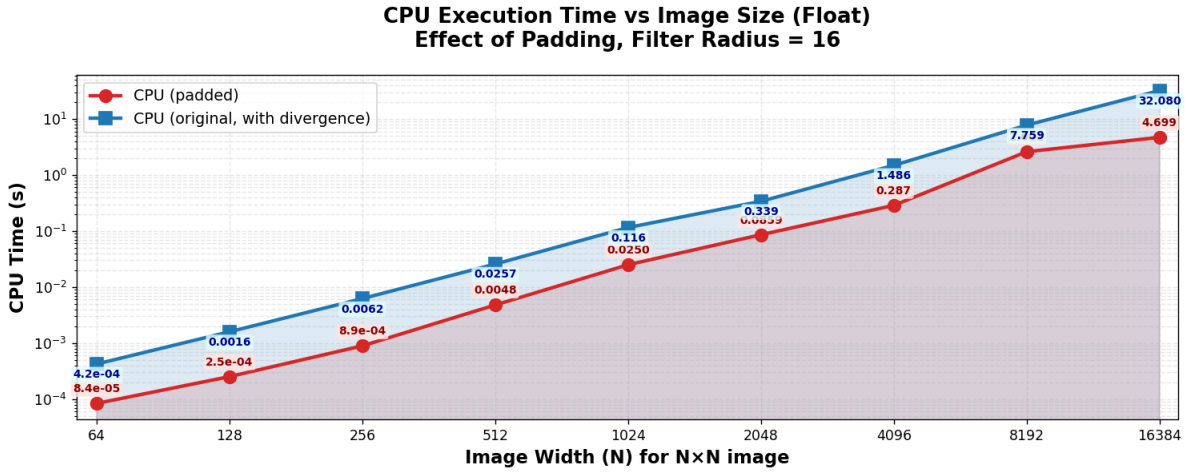

Figure 19: Execution time on CPU Padded vs Non-padded

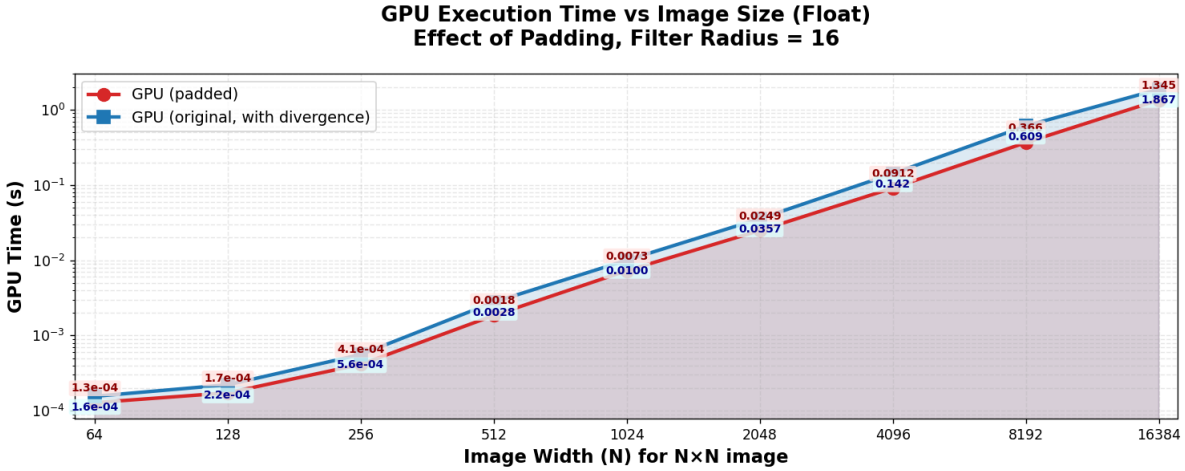

Figure 20: Execution time on GPU Padded vs Non-padded

From the graphs above, we observe that padding the image borders did not lead to significant performance improvements on the GPU. This is likely because the number of divergent threads near the image boundaries is very small compared to the total number of threads processing the large image. In contrast, on the CPU we observe a clear performance improvement when using padding. By removing the boundary checks (the if conditions), we eliminate branch mispredictions reducing execution time significantly.