

Electrical and Computer Engineering
University of Thessaly (UTH)

ECE415 - High Performance Computing (HPC)

Fall Semester — Educational year 2024-2025

Lab03

CUDA Implementation of 2-D separable Convolution filter

Dimitris Tsalapatas - AEM: 03246

Georgios Kapakos - AEM: 03165

Contents

1	Introduction	3
2	Device Query	3
3	Nvidia Compiler Parameters	4
4	Implementing the CUDA optimizations with a single block of threads	4
4.1	Experimental Study	6
5	Implementing the CUDA optimizations with multiple blocks in a grid	7
5.1	Experimental Study	9
6	Implementation with doubles	10
7	Theoretical Questions	12
8	Padding Implementation	13

1 Introduction

The implementation of a 2-D convolution filter for the computation of a output value (pixel), requires $imageW \cdot imageH$ multiplications, where imageW, imageH the image's Width and Height. A separable filter is one who can be defined as the composition of two 1-D filters. One of those two is implemented to the image's rows and the other to the image's columns. The ConvolutionRow function iterates over the image's pixels in each row, applying the filter horizontally and returns the result in a Buffer. Then the ConvolutionColumn function iterates over the image's pixels in each column, applying the filter vertically and returns the result of the convolution.

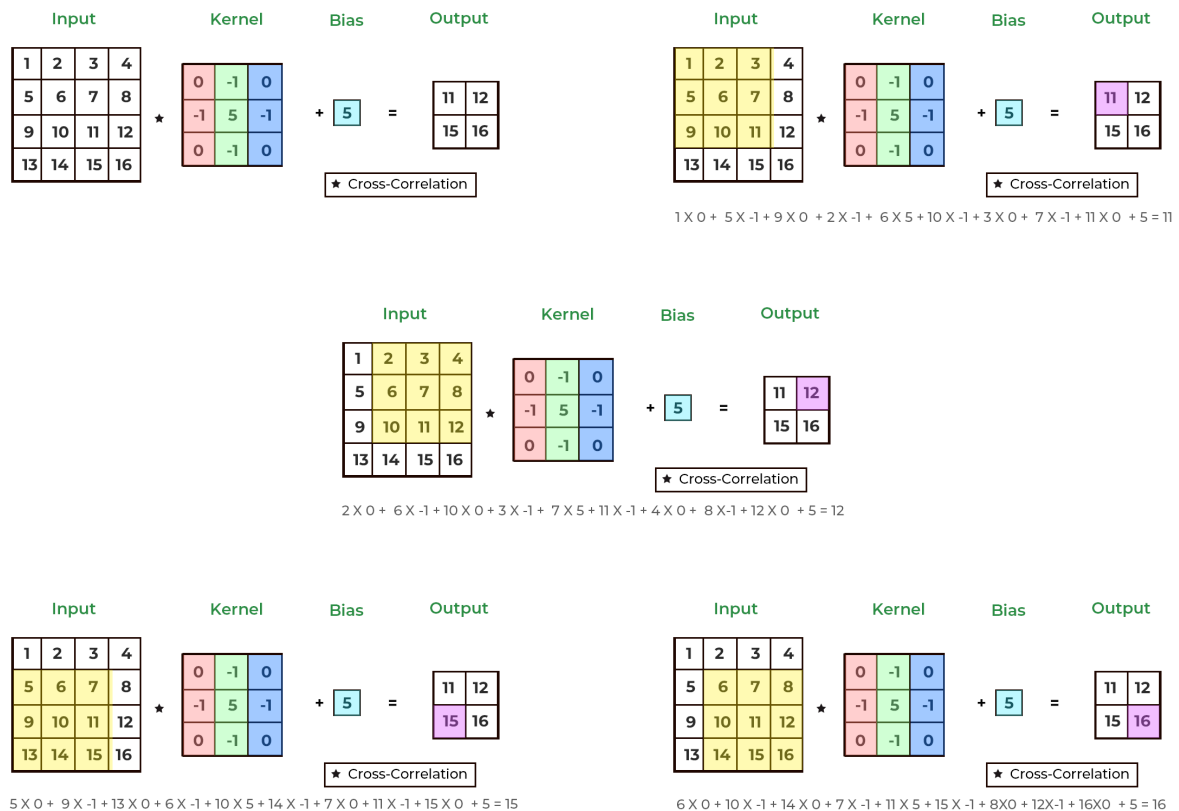


Figure 1: Visualisation for convolution filter apply

2 Device Query

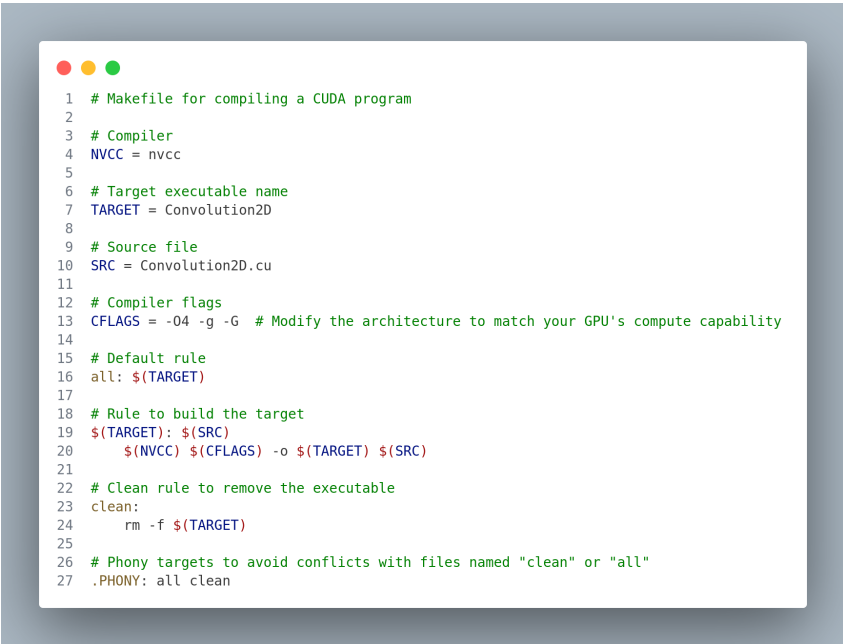
We used the DeviceQuery command and found out the Tesla K80 GPU that we use has the following specifications:

- **Device Name:** Tesla K80
- **CUDA Capability Version:** 3.7
- **Total Global Memory:** 11441 MBytes
- **Total CUDA Cores:** 2496

- **GPU Max Clock Rate:** 824 MHz (0.82 GHz)
- **Memory Clock Rate:** 2505 MHz
- **Memory Bus Width:** 384-bit
- **L2 Cache Size:** 1572864 bytes
- **Total Shared Memory per Block:** 49152 bytes
- **Total Constant Memory:** 65536 bytes
- **Total Number of Registers per Block:** 65536
- **Maximum 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 Nvidia Compiler Parameters

After running the `nvcc -help` command in the terminal we decide to compile our CUDA program with the following Makefile:



```

1  # Makefile for compiling a CUDA program
2
3  # Compiler
4  NVCC = nvcc
5
6  # Target executable name
7  TARGET = Convolution2D
8
9  # Source file
10 SRC = Convolution2D.cu
11
12 # Compiler flags
13 CFLAGS = -O4 -g -G # Modify the architecture to match your GPU's compute capability
14
15 # Default rule
16 all: $(TARGET)
17
18 # Rule to build the target
19 $(TARGET): $(SRC)
20     $(NVCC) $(CFLAGS) -o $(TARGET) $(SRC)
21
22 # Clean rule to remove the executable
23 clean:
24     rm -f $(TARGET)
25
26 # Phony targets to avoid conflicts with files named "clean" or "all"
27 .PHONY: all clean
  
```

Figure 2: Makefile

4 Implementing the CUDA optimizations with a single block of threads

These are the steps we follow for the CUDA implementation of our code with single thread blocks:

- We allocate memory for the GPU's input and output arrays, using CudaMalloc and checking if the allocations are successful.
- After initializing the CPU's inputs with random values we copy those values to the device's inputs, using CudaMemcpy.
- We call the CPU's functions to produce the host's output of the program, measuring its time. We do the same for the GPU, but first we setup the block's and grid's geometry, which include, only one block with dimensions equal to the width and height of the image.
- We use cudaMemcpy for the transfer of the device's results back to the CPU so that we can compare them with the CPU's results.
- We take the difference between the values of the CPU's values with the device's values, calculating the worse difference that they have so that we can later declare the accuracy of our program.
- We finish with freeing both the CPU and the GPU memory.

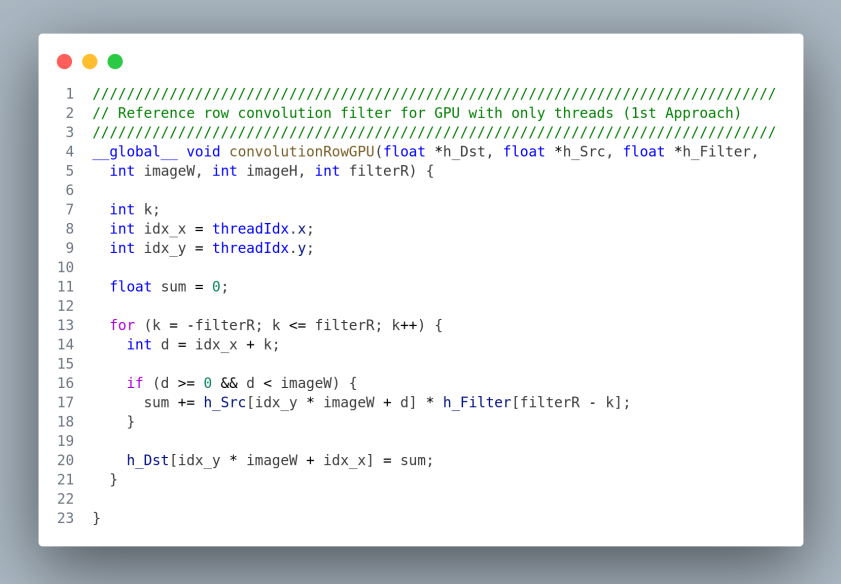
A screenshot of a code editor window with a white background and a grey border. The code is written in C++ and is a CUDA kernel for a 1D convolution. It starts with a multi-line comment on line 1, followed by a single-line comment on line 2. Line 3 is another multi-line comment. Line 4 is the function signature: `__global__ void convolutionRowGPU(float *h_Dst, float *h_Src, float *h_Filter, int imageW, int imageH, int filterR) {`. Line 5 is the opening curly brace. Line 6 is an empty line. Line 7 declares `int k;`. Line 8 declares `int idx_x = threadIdx.x;`. Line 9 declares `int idx_y = threadIdx.y;`. Line 10 is an empty line. Line 11 declares `float sum = 0;`. Line 12 is an empty line. Line 13 starts a for loop: `for (k = -filterR; k <= filterR; k++) {`. Line 14 declares `int d = idx_x + k;`. Line 15 is an empty line. Line 16 starts an if statement: `if (d >= 0 && d < imageW) {`. Line 17 is the body of the if statement: `sum += h_Src[idx_y * imageW + d] * h_Filter[filterR - k];`. Line 18 is the closing curly brace for the if statement. Line 19 is an empty line. Line 20 is the assignment: `h_Dst[idx_y * imageW + idx_x] = sum;`. Line 21 is the closing curly brace for the for loop. Line 22 is an empty line. Line 23 is the closing curly brace for the function. The code is color-coded: comments are green, keywords are blue, variables are black, and literals are red.

Figure 3: GPU Rows function implementation

```

1 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2 // Reference column convolution filter GPU with threads (1st Approach)
3 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
4 _global_ void convolutionColumnGPU(float *h_Dst, float *h_Src, float *h_Filter,
5 int imageW, int imageH, int filterR) {
6
7     int k;
8     int idx_x = threadIdx.x;
9     int idx_y = threadIdx.y;
10    float sum = 0;
11
12    for (k = -filterR; k <= filterR; k++) {
13        int d = idx_y + k;
14
15        if (d >= 0 && d < imageH) {
16            sum += h_Src[d * imageW + idx_x] * h_Filter[filterR - k];
17        }
18
19        h_Dst[idx_y * imageW + idx_x] = sum;
20    }
21 }
22 }

```

Figure 4: GPU Columns function implementation

4.1 Experimental Study

- From our experimental study for images with a radius of 4, we observe that the maximum height/width we can support is 32 pixels (32x32 pixel image). The problem with supporting bigger images lies in the fact that a single block only has 32x32 threads. For the comparison of the values between the CPU and the GPU, we remove the -g and -G flags, which are used for debugging. These flags slow down the performance of our program, but especially the -G flag also seems to improve the accuracy of our program.
- For a 32x32 image we will start from an image with a radius of 1 and finish with an image with a radius of 15. Comparing the CPU with the GPU results will determine the accuracy of our model.

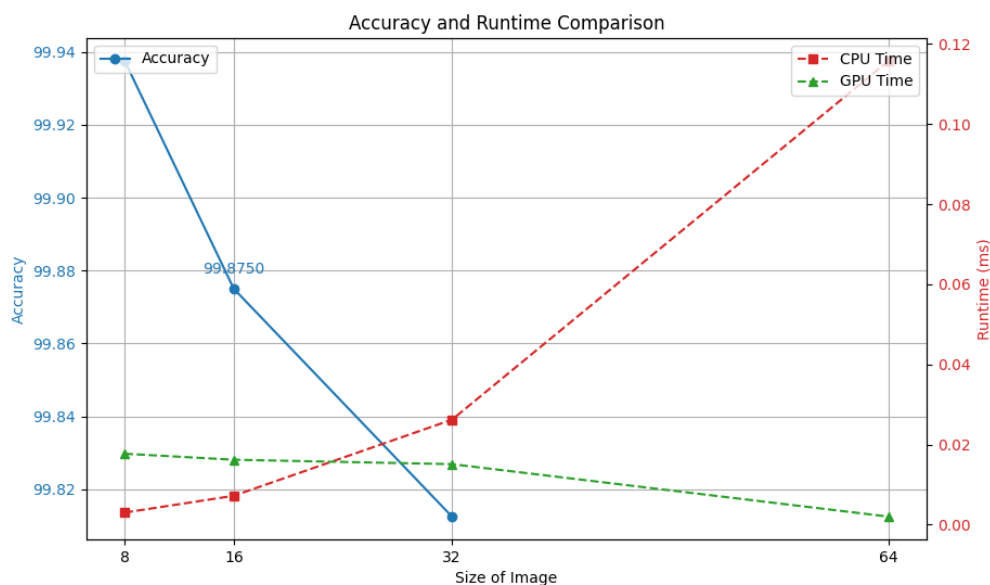


Figure 5: Radius=4, different images size

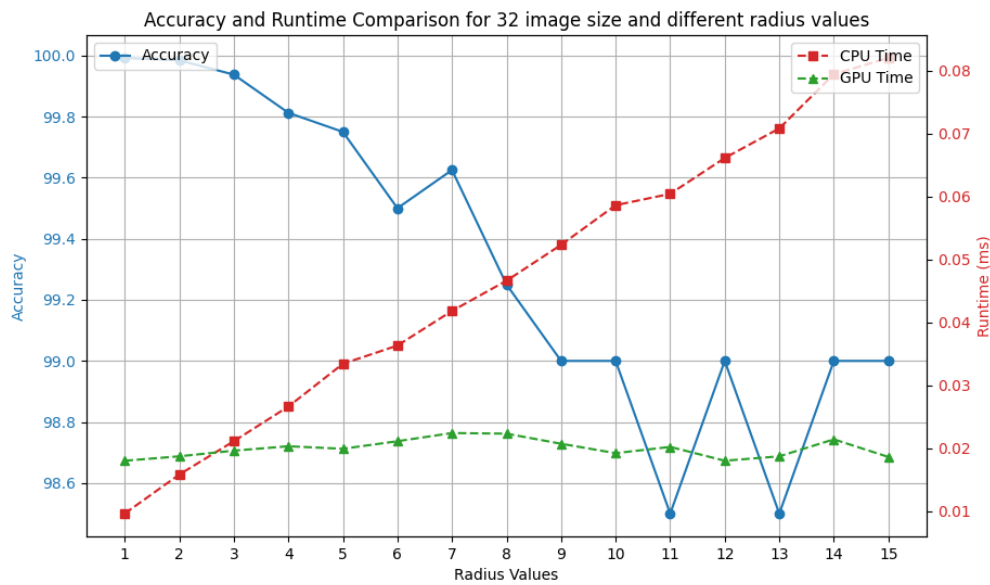


Figure 6: Image size = 32, radiuses from 1 - 16

Radius	Accuracy (in decimals)	CPU Runtime	GPU Runtime
1	3	9.7e-06	1.62e-05
2	2	1.55e-05	1.63e-05
3	2	2.09e-05	1.65e-05
4	1	2.61e-05	1.66e-05
5	1	3.11e-05	1.67e-05
6	1	3.68e-05	1.55e-05
7	1	4.04e-05	1.6e-05
8	1	4.6e-05	1.56e-05
9	0	5.21e-05	1.58e-05
10	0	5.73e-05	1.68e-05
11	0	5.94e-05	1.54e-05
12	0	6.53e-05	1.65e-05
13	0	6.94e-05	1.68e-05
14	0	7.6e-05	1.69e-05
15	0	8.17e-05	1.57e-05

Table 1: Accuracy on different Radius

5 Implementing the CUDA optimizations with multiple blocks in a grid

The previous code, only takes advantage of one block of threads where now we will consider having multiple blocks of threads, fixing the previous problems. This optimization leads to:

- **Scalability to larger data sets:** A single block of threads can accommodate for just

32x32=1024 pixels, where a grid of blocks can extend the computation of larger data sets.

- **Better Utilization of the GPU Hardware:** Multiple blocks of threads leads to a better parallelization of the code, maximizing the parallel execution and speeding up the overall computation. To maximize the performance in our program we use the maximum number of threads in each block, this is equal to 1024, setting a TILE_WIDTH of 32. Where the grid uses dimensions of imageW/TILE_WIDTH and imageH/TILE_WIDTH.

The maximum size that we are able to implement for a radius of 4 is going to be for an image with the size of 16384x16384. The calculations for this problem are the following:

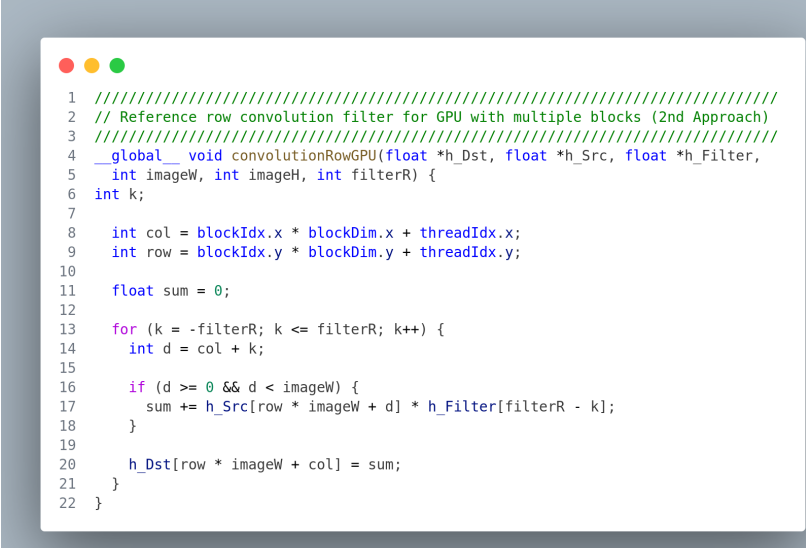
- **Image size:** The image's size $Width \cdot Length = 16384 \cdot 16384$.
- **Bytes per pixel:** Because the arrays are floats, each float is 4 bytes, so each pixel has 4 bytes as each pixel is represented by a float.
- **Memory Allocation arrays:** For the device's memory allocation we allocate the memory for 3 arrays, which are: Input, Buffer and Output array.

So the result is calculated by this formula:

$$Image_size \cdot bytes_per_pixel \cdot num_memory_allocations$$

For the upper specifications we have: $16384 \cdot 16384 \cdot 4 \cdot 3$, which is equal to 3,321.22546 MBytes, which is less than the total global memory of the device as $3,321.22546 < 11,441$ MBytes.

We then try for the next possible image size of 32768x32678, but it doesn't fit in the GPU as it's value is: $12,884.9019 > 11,441$ MBytes. In the calculations we excluded the calculation of the filter, because the filter size is insignificant compared to the image size.



```

1  //////////////////////////////////////////////////
2  // Reference row convolution filter for GPU with multiple blocks (2nd Approach)
3  //////////////////////////////////////////////////
4  __global__ void convolutionRowGPU(float *h_Dst, float *h_Src, float *h_Filter,
5  int imageW, int imageH, int filterR) {
6  int k;
7
8  int col = blockIdx.x * blockDim.x + threadIdx.x;
9  int row = blockIdx.y * blockDim.y + threadIdx.y;
10
11  float sum = 0;
12
13  for (k = -filterR; k <= filterR; k++) {
14      int d = col + k;
15
16      if (d >= 0 && d < imageW) {
17          sum += h_Src[row * imageW + d] * h_Filter[filterR - k];
18      }
19
20      h_Dst[row * imageW + col] = sum;
21  }
22  }

```

Figure 7: GPU Rows function implementation


```

1 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2 // Reference column convolution filter GPU with multiple blocks (2nd Approach)
3 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
4 __global__ void convolutionColumnGPU(float *h_Dst, float *h_Src, float *h_Filter,
5   int imageW, int imageH, int filterR) {
6
7   int k;
8
9   int col = blockIdx.x * blockDim.x + threadIdx.x;
10  int row = blockIdx.y * blockDim.y + threadIdx.y;
11
12  float sum = 0;
13
14  for (k = -filterR; k <= filterR; k++) {
15    int d = row + k;
16
17    if (d >= 0 && d < imageH) {
18      sum += h_Src[d * imageW + col] * h_Filter[filterR - k];
19    }
20
21    h_Dst[row * imageW + col] = sum;
22  }
23 }

```

Figure 8: GPU Columns function implementation

5.1 Experimental Study

- Starting from a small radius, going all the way up to the maximum radius it can support we observe that the accuracy values are increasing, but each and every one of them, is within a respectful threshold. The thresholds are being modified by few decimal points as the radius increases.
-

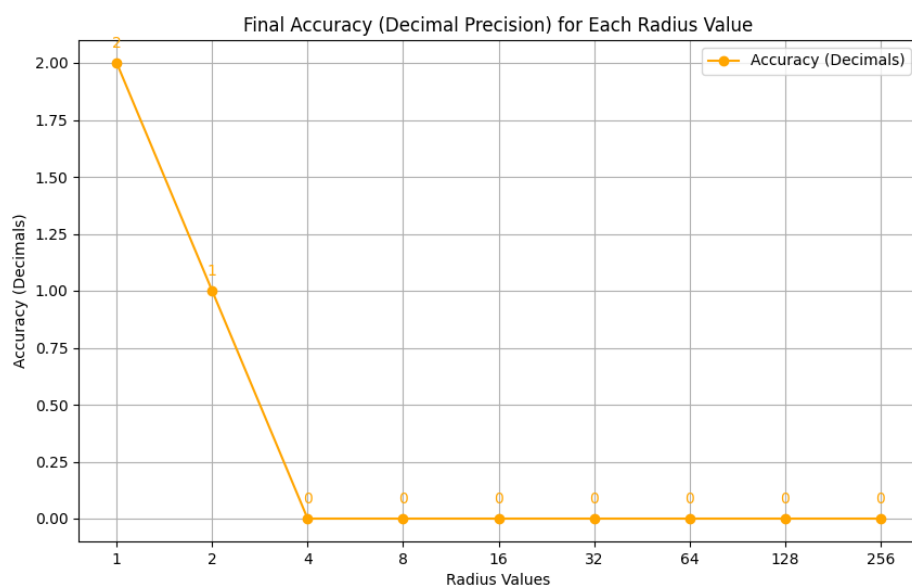


Figure 9: Accuracy on Image size 1024 and different Radius values

Radius	Accuracy in decimals
1	2
2	1
4	0
8	0
16	0
32	0
64	0
128	0
256	0

Table 2: Accuracy in decimals for various radius values

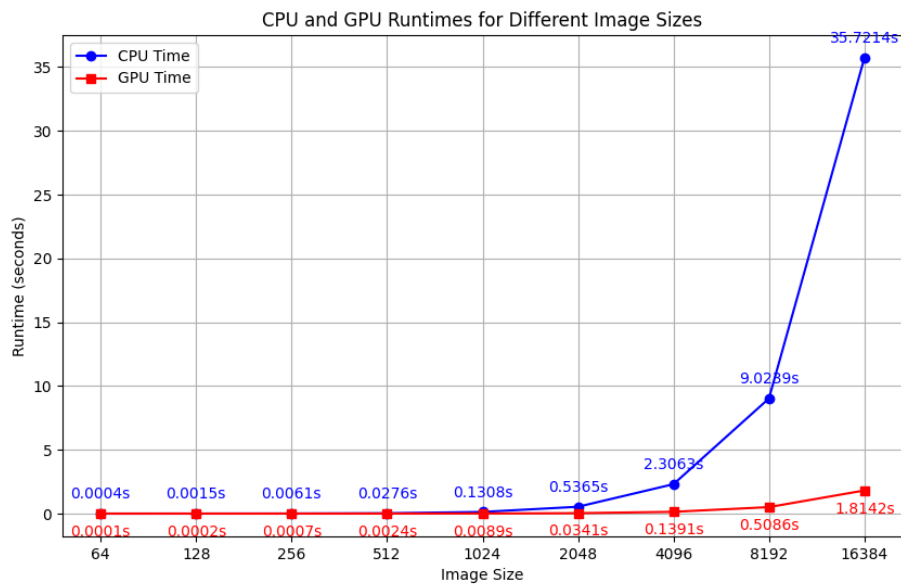


Figure 10: Accuracy on Image size 1024 and different Radius values

6 Implementation with doubles

In the same spirit as the previous code implementation we will change all the floats in the program to doubles. Doubles have a precision of 15 decimal points in comparison to the floating point numbers. That means that the accuracy of the program is due to be improved by a lot, where the performance is going to be affected negatively and also the area that the program occupies is going to rapidly increase, as doubles occupy more memory.

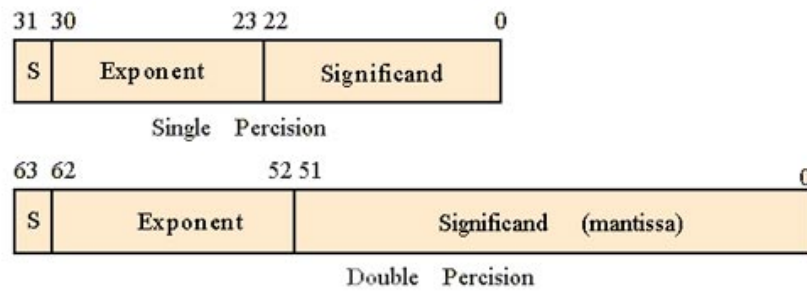


Figure 11: Accuracy on Image size 1024 and different Radius values

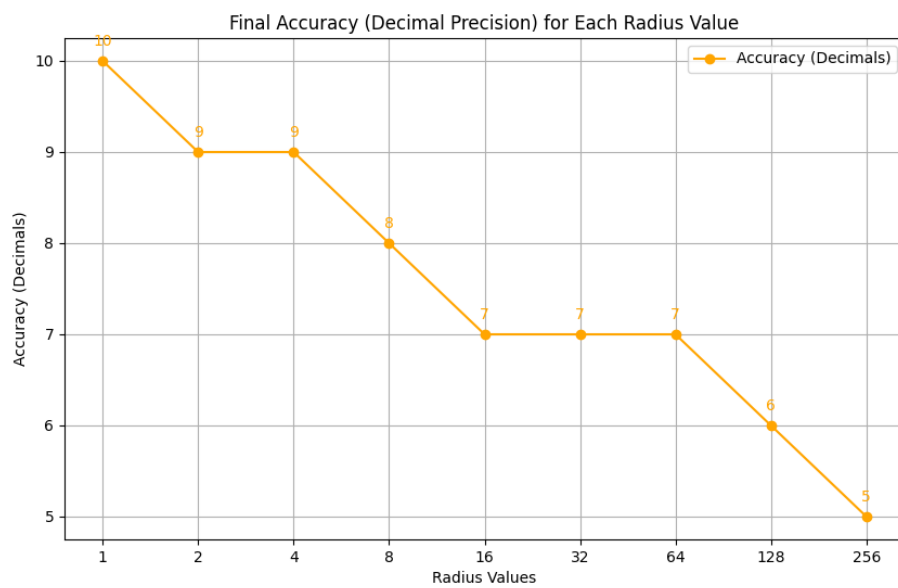


Figure 12: Accuracy on Image size 1024 and different Radius values

Radius	Accuracy in decimals
1	10
2	9
4	9
8	8
16	7
32	7
64	7
128	6
256	5

Table 3: Accuracy in decimals for various radius values (doubles Implementation)

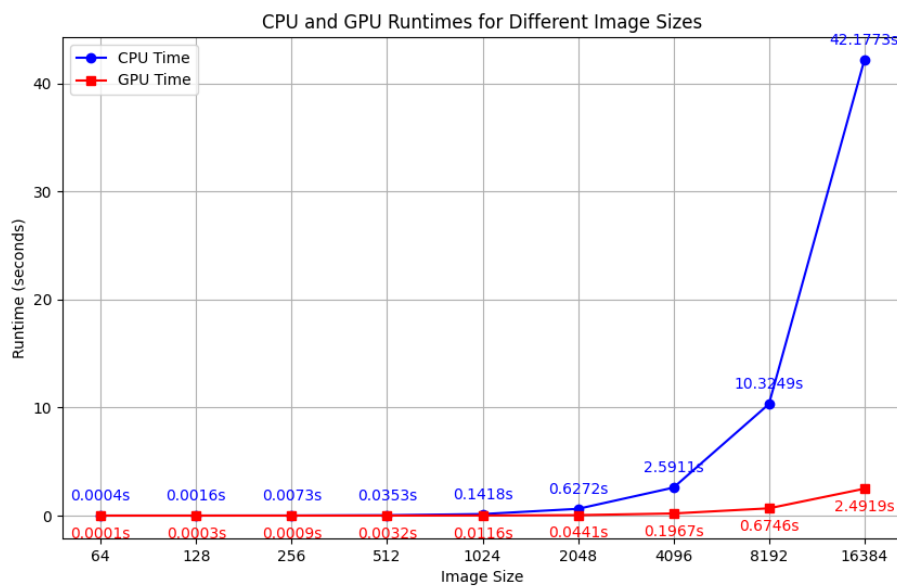


Figure 13: Accuracy on Image size 1024 and different Radius values (doubles Implementation)

7 Theoretical Questions

- **How many times does a image pixel accessed from kernel:** The ConvolutionRow filter traverses the rows of the image and the ConvolutionColumn the columns of the image.

For the rows: we apply the 1-D filter horizontally, to each pixel. The filter goes from left to right and it's center being applied in the pixel of interest. The filter then slides till the end of the row sequentially, accessing all the elements in each row. The first and last elements of each row is being accessed the least times as the filter passes through those

elements radius + 1 times. . Where the other elements which are next to the first and last element will be accessed + 1 times till the center of the row. The center is accessed $2 * \text{radius} + 1$ times, which is the maximum in each row.

For the columns: we will follow the same procedure vertically and apply the 1-D filter through the image. Now we will traverse the values from top to down and the edge values will be accessed only radius + 1 times, where the middle value will be accessed the most ($2 * \text{radius} + 1$). Where the values from the edges to the middle point will be accessed increasing times as the reach the middle.

For the filter: From the middle point of the filter which gets accessed the most times, which is the whole size of the image ($\text{width} * \text{height}$), to the edges which are accessed $\text{width} * \text{height} - \text{filter_size}$ times, with the intermediate points getting accessed in a range between $\text{width} * \text{height} - \text{filter_size}$ to $\text{width} * \text{height}$.

- **What is the ratio of memory accesses to decimal point operations?:** In the previous code we observe that there are two decimal point operations (a multiplication and an addition) and two memory operations (d_Src and d_Filter is called). An element from the input array and an element from the filter is accessed in memory. Both of those are stored in the global memory. Therefore, the ratio is equal to 1:1 , this creates a bottleneck in the performance.

8 Padding Implementation

The elements near the edges of the array, within a distance of FILTER_RADIUS, depend on values that do not exist in the actual array. To address this, padding is added to the array, where these extra elements are assigned a default value of 0. This padding extends outward from the edges by FILTER_RADIUS elements in all directions. Padding is applied to both the input and buffer arrays in both the CPU and GPU implementations. Padding was applied to both the input and buffer arrays for the CPU and GPU implementations. Here we can see these implementations.




```

1 //////////////////////////////////////////////////
2 // Reference row conon filter
3 //////////////////////////////////////////////////
4 void convolutionRowCPUPadded(float *h_Dst, float *h_Src, float *h_Filter,
5                             int imageW, int imageH, int filterR) {
6
7     int x, y, k;
8
9     for (y = 0; y < imageH; y++) {
10         for (x = 0; x < imageW; x++) {
11             float sum = 0;
12
13             for (k = -filterR; k <= filterR; k++) {
14                 int d = (x+filterR) + k;
15
16                 sum += h_Src[(y + filterR) * (imageW + filterR*2) + d] * h_Filter[filterR - k];
17
18                 h_Dst[(y+filterR) * (imageW + filterR*2) + (x+filterR)] = sum;
19             }
20         }
21     }
22 }
23

```

Figure 14: CPU Padding implementation Rows



```

1 void convolutionColumnCPUPadded(float *h_Dst, float *h_Src, float *h_Filter,
2                                 int imageW, int imageH, int filterR) {
3
4     int x, y, k;
5
6     for (y = 0; y < imageH; y++) {
7         for (x = 0; x < imageW; x++) {
8             float sum = 0;
9
10            for (k = -filterR; k <= filterR; k++) {
11                int d = (y+filterR) + k;
12
13                sum += h_Src[d * (imageW + filterR*2) + (x+filterR)] * h_Filter[filterR - k];
14
15                h_Dst[(y+filterR) * (imageW + filterR*2) + (x+filterR)] = sum;
16            }
17        }
18    }
19 }
20

```

Figure 15: CPU Padding implementation Columns

We have to allocate memory with size $imageH + filter_radius * 2$ for each input variable as we can see on code block.

```

1  /* Allocate Memory for the device(GPU) */
2  // Check allocation of device memory
3  if (cudaMalloc((void**) &d_Filter, FILTER_LENGTH * sizeof(float)) != cudaSuccess) {
4      printf("Error allocating memory for d_Filter on the device: %s\n", cudaGetErrorString(cudaGetLastError()));
5  }
6
7  if (cudaMalloc((void**) &d_Input, (imageW + filter_radius*2) * (imageH + filter_radius*2) * sizeof(float)) != cudaSuccess) {
8      printf("Error allocating memory for d_Input on the device: %s\n", cudaGetErrorString(cudaGetLastError()));
9  }
10
11 if (cudaMalloc((void**) &d_Buffer, (imageW + filter_radius*2) * (imageH + filter_radius*2) * sizeof(float)) != cudaSuccess) {
12     printf("Error allocating memory for d_Buffer on the device: %s\n", cudaGetErrorString(cudaGetLastError()));
13 }
14
15 if (cudaMalloc((void**) &d_OutputGPU, (imageW + filter_radius*2) * (imageH + filter_radius*2) * sizeof(float)) != cudaSuccess) {
16     printf("Error allocating memory for d_OutputGPU on the device: %s\n", cudaGetErrorString(cudaGetLastError()));
17 }
18
19 // to 'h_Filter' apotelei to filtro me to opoio ginetai to convolution kai
20 // arxikopoieitai tuxaia. To 'h_Input' einai h eikona panw sthn opoia ginetai
21 // to convolution kai arxikopoieitai kai auth tuxaia.
22
23 srand(200);
24
25 for (i = 0; i < FILTER_LENGTH; i++) {
26     h_Filter[i] = (float)(rand() % 16);
27 }
28
29 for (i = 0; i < (imageW) * (imageH); i++) {
30     h_Input[i] = ((float)rand() / ((float)RAND_MAX / 255) + (float)rand() / (float)RAND_MAX);
31 }
32
33 for (i = 0; i < (imageH); i++) {
34     for(j = 0; j < (imageW); j++) {
35         h_InputPadded[(i+filter_radius)*(imageW+filter_radius*2) + (j + filter_radius)] = h_Input[i*imageW + j];
36     }
37 }

```

Figure 16: CPU Padding implementation Columns


Also, on side of GPU we try to make same implemenation, in order to avoid the if condition and use padding. Here is the trial implementation.

```

1  __global__ void convolutionRowGPU(float *d_Dst, float *d_Src, float *d_Filter,
2  int imageW, int imageH, int filterR) {
3
4  int row = blockDim.y * blockIdx.y + threadIdx.y;
5  int col = blockDim.x * blockIdx.x + threadIdx.x;
6  int k;
7
8  row = row + filterR;
9  col = col + filterR;
10
11 float sum = 0.0;
12
13 for (k = -filterR; k <= filterR; k++) {
14     int d = (col) + k;
15     sum += d_Src[(row) * (imageW + filterR * 2) + d] * d_Filter[filterR - k];
16 }
17 d_Dst[(row) * (imageW + filterR*2) + (col)] = sum;
18 printf("%f\n", d_Dst[(row) * (imageW + filterR*2) + (col)]);
19 }

```

Figure 17: GPU Padding implementation Rows



```
1  __global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, float *d_Filter,
2  int imageW, int imageH, int filterR) {
3
4  int row = blockDim.y * blockIdx.y + threadIdx.y;
5  int col = blockDim.x * blockIdx.x + threadIdx.x;
6  int k;
7
8  row = row + filterR;
9  col = col + filterR;
10
11 float sum = 0.0;
12 for (k = -filterR; k <= filterR; k++) {
13     int d = (row) + k;
14
15     sum += d_Src[d * (imageW + filterR * 2) + (col)] * d_Filter[filterR - k];
16
17 }
18 d_Dst[(row) * (imageW + filterR*2) + (col)] = sum;
19 }
20
```

Figure 18: GPU Padding implementation Columns

Unfortunately, the GPU implementation does not have the correct accuracy and the results are wrong, and we did not get measurements for the runtime and accuracy.