High Performance Computing Systems
Project 3

Tsongas Panagiotis Nikas Ioannis Iason Tsiantos Dimitrios

November 2025

Contents

1 System Information **3**

2 Experimental Study: Maximum Image Size and Accuracy **3**
- 2.1 Image Size Limits . . . 3
- 2.2 FMAD . . . 3
- 2.3 Calculation Accuracy . . . 4

3 Implementation of Multiple Thread Blocks **5**
- 3.1 Image Size Limits . . . 5

4 Experimental Study with Floats **6**
- 4.1 Maximum Difference . . . 6
- 4.2 CPU & GPU Times per Image Size . . . 6

5 Experimental Study with Doubles **8**
- 5.1 Maximum Difference . . . 8
- 5.2 CPU & GPU Times per Image Size . . . 8

6 Memory Accesses and FLOPS **10**
- 6.1 Image and Filter Reads . . . 10
- 6.2 Memory Accesses to FLOPs Ratio . . . 10

7 Eliminating Divergence **10**

A Appendix **12**

1 System Information

By running the device query from the cuda-samples, we can see various pieces of information regarding the system's graphics cards. We observe that four Tesla K80 devices are connected (two per board) with a total global memory size of `11441 MBytes` per device instead of `12 Gbytes` (`24 Gbytes` per board). This occurs because ECC (Error Correction Code) is enabled and occupies ∼6.25% of the total memory, leaving ∼`11.25 GBytes` per device available for the user.

Memory Type	**Size**
Global Memory (VRAM)	$11\,441\,\mathrm{MB}$ (Approx. $11.2\,\mathrm{GB}$)
L2 Cache Size (Global)	$1536\,\mathrm{kB}$ (Approx. $1.5\,\mathrm{MB}$)
Shared Memory per Block	$48\,\mathrm{kB}$
Total Constant Memory (Device)	$64\,\mathrm{kB}$
Registers per Block	$65\,536$ (32-bit registers)

2 Experimental Study: Maximum Image Size and Accuracy

2.1 Image Size Limits

By using a single block for the kernel implementation, which includes 1024 threads mapped one-to-one with the pixels, we can predict (based on the deviceQuery data A) that the maximum image dimension we can process is $W \times H = 1024$. Given that $W = H$, for a simpler implementation, it follows that $W = H = \sqrt{1024} = 32$. This is also observed experimentally, as with a width and height greater than 32, we receive the following error message:

```
CUDA Error:  invalid argument in Convolution2D.cu, line 235
```

2.2 FMAD

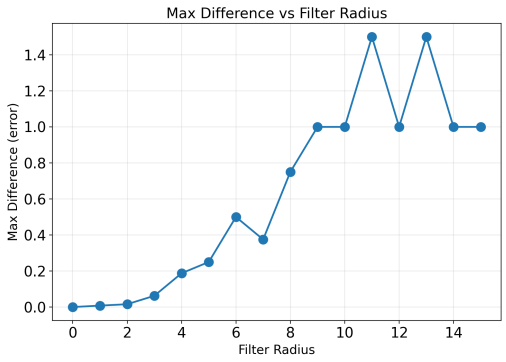
In our tests, a significant deviation was observed between the Host and Device results. This deviation disappeared when using the -G flag during compilation. From nvcc –help, we confirmed that this specific flag disables optimizations, including the use of the fused multiply–add (FMAD) operation. This operation allows the Device to execute a multiplication and an addition in a single instruction, with greater accuracy due to single rounding. Paradoxically, the difference we observed is ultimately due to the numerical inaccuracy of the Host and not a Device error when FMAD is enabled.

2.3 Calculation Accuracy

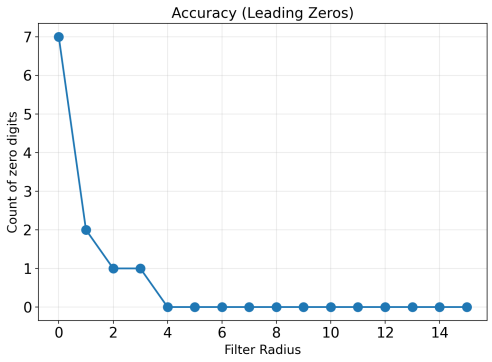
For an image with dimensions 32×32 and testing filters with radii from 0 to 15, we made the following measurements for the program's accuracy:

Filter Radius	Maximum Difference
0	0
1	0.0078125
2	0.015625
3	0.0625
4	0.1875
5	0.25
6	0.5
7	0.375
8	0.75
9	1.0
10	1.0
11	1.5
12	1.0
13	1.5
14	1.0
15	1.0

We conclude that as the filter size increases, the accuracy decreases significantly.



(a) Maximum difference as a function of radius



(b) Maximum difference in digits

Figure 1: Error analysis as a function of filter radius

3 Implementation of Multiple Thread Blocks

For the implementation with multiple blocks, we use the structure shown in Figure 2 for thread organization. Each thread identifies the pixel it will work on through the following indexing:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

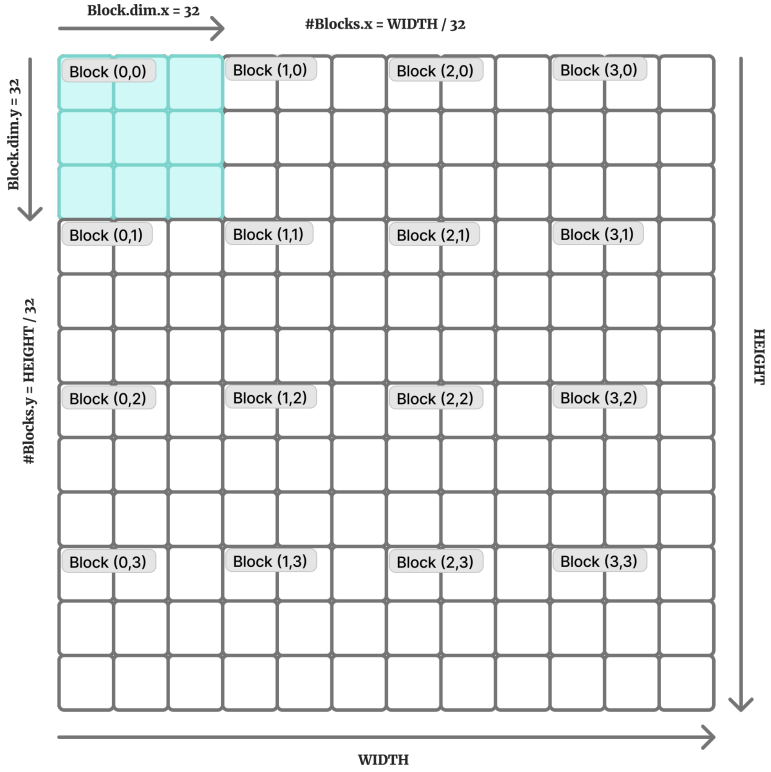


Figure 2: Grid Layout

3.1 Image Size Limits

Theoretically, the use of multiple blocks allows us to request the execution of a kernel with a grid up to $int_max \cdot 65536$, which is the arbitrary limit of CUDA. Each block has dimensions $block_width = 32$, hence $32 \times 32 = 1024$ threads, as mentioned previously. Therefore, the theoretical maximum image size that could be given as input is:

$$(block_width \cdot int_max) \times (block_width \cdot int_max).$$

However, in practice, the program stops working much earlier, as the real bottleneck is not the grid limits, but the available memory. Specifically, on the Device, when using the arrays d_Input, d_Buffer, and d_Output, the total allocated memory amounts to $3 \cdot W \cdot H \cdot$sizeof(pixel), which directly limits the maximum allowed image size. Given a memory of 11997020160 bytes, from deviceQuery A, we have:

$$\text{TotalAlloc}_{\text{bytes}} = 3 \cdot \text{Width} \cdot \text{Height} \cdot \text{sizeof(PixelScalar)}_{\text{bytes}}$$

$$\text{Width} \cdot \text{Height} = \frac{11997020160_{\text{bytes}}}{4_{\text{bytes}} \cdot 3}$$

$$\text{Width} \cdot \text{Height} = 999751680, \quad (\text{Width} = \text{Height} = \text{side})$$

$$\text{side} = \left\lfloor \sqrt{999751680} \right\rfloor = 31618$$

Later, we applied an optimization where we use only the d_Input and d_Buffer arrays, while the unused d_Input array is reused as the output array, saving significant memory space. Thus, the new limit is:

$$\text{TotalAlloc}_{\text{bytes}} = 2 \cdot \text{Width} \cdot \text{Height} \cdot \text{sizeof(PixelScalar)}_{\text{bytes}}$$

$$\text{Width} \cdot \text{Height} = 1499627520, \quad (\text{Width} = \text{Height} = \text{side})$$

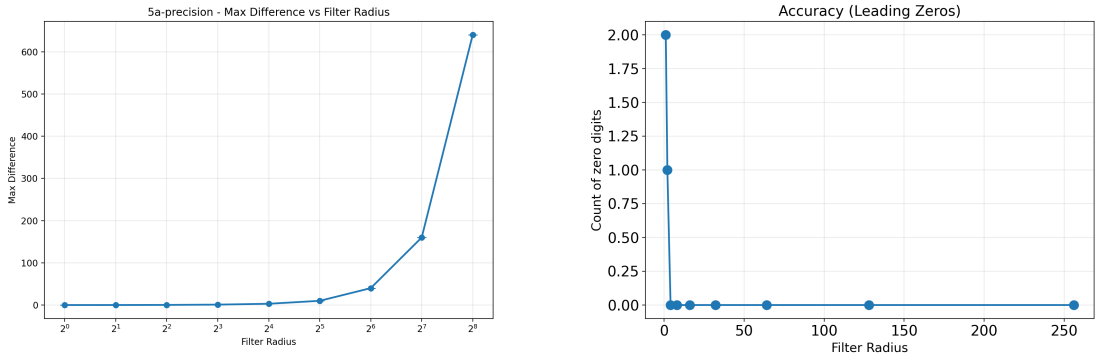
$$\text{side} = \left\lfloor \sqrt{1499627520} \right\rfloor = 38725$$

In reality, on the Device, we can run the algorithm simultaneously for $\#\text{MPs} \cdot \#\text{ThreadsMPs}$, which is $13 \cdot 2048 = 26624_{pixels}$.

4 Experimental Study with Floats

4.1 Maximum Difference

In the program, we observe a large deviation in the maximum difference of the result between the host and the device. This is due to the different way instructions are executed in the two architectures. The CPU runs instructions serially per pixel, performing rounding after each floating point operation. Conversely, the graphics card executes a multiplication and an addition in one instruction (FMAD 2.2), without losing much precision.



(a) Maximum difference as a function of radius.

(b) Maximum difference in digits.

Figure 3: Error analysis as a function of filter radius.

Filter Radius	1	2	4	8	16	32	64	128	256
Max Difference	0.0078125	0.046875	0.25	1	3	10	40	160	640

4.2 CPU & GPU Times per Image Size

As the image size increases, the execution time on both the CPU and the graphics card increases exponentially. Although both have a similar growth rate, the graphics card has a much lower execution time.

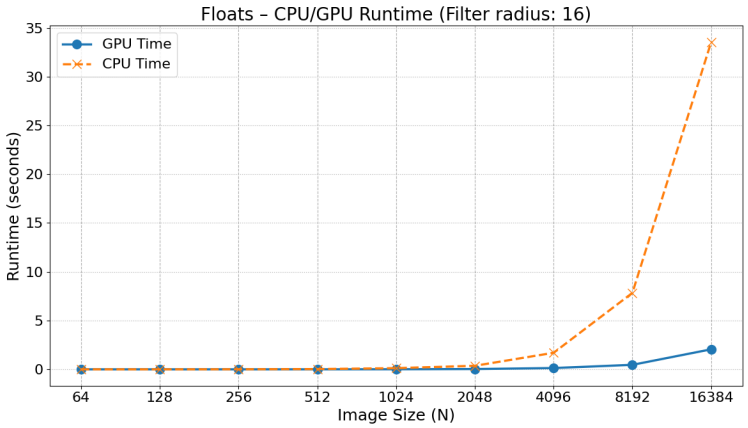


Figure 4: Execution time with floats per image size.

Image Size	GPU avg (s)	GPU Std dev (s)	CPU avg (s)	CPU Std dev (s)	Speedup
64×64	0.0001489	7.3401E-06	0.0004886	2.1329E-05	3.28×
128×128	0.0002829	6.5735E-06	0.0019377	6.0046E-05	6.85×
256×256	0.0007317	9.2622E-06	0.0074893	9.9947E-05	10.24×
512×512	0.0026406	2.3566E-05	0.0299085	0.0002121	11.33×
1024×1024	0.0094353	4.7866E-05	0.1218538	0.0029249	12.91×
2048×2048	0.0334919	0.0004511	0.3646621	0.0161425	10.89×
4096×4096	0.1328186	0.0011445	1.6972503	0.1270327	12.78×
8192×8192	0.4616295	0.0368957	7.8352702	0.2023358	16.97×
16384×16384	2.0517052	0.0949082	33.5328534	1.4619840	16.34×

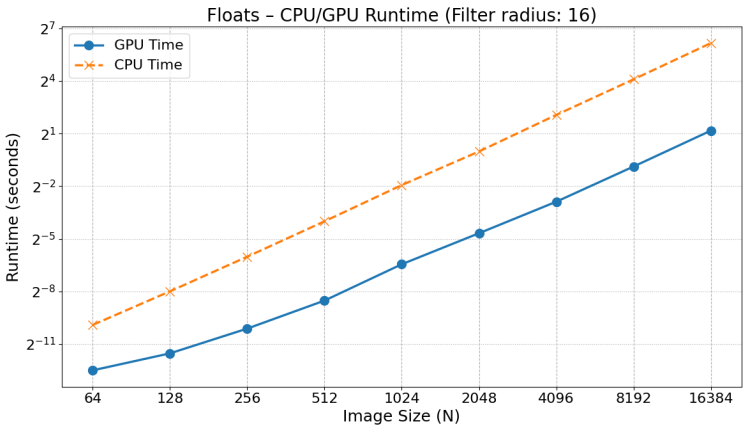
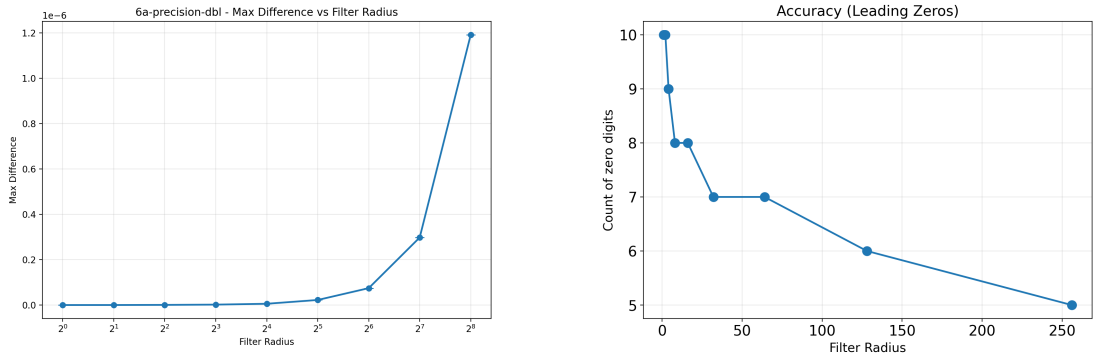


Figure 5: Execution times on a logarithmic scale.

5 Experimental Study with Doubles

5.1 Maximum Difference

Changing the pixel type from float to double, we see a significant reduction in the maximum difference, as double has better precision. Specifically, while float uses 32 bits to represent a floating-point number, double uses 64 bits. This means that operations involving multiplications and additions are less affected by rounding errors. The use of double is appropriate for cases where high accuracy is required.



(a) Maximum difference as a function of radius.

(b) Maximum difference in digits.

Figure 6: Error analysis as a function of filter radius.

Filter Radius	Maximum Difference
1	1.4552E-11
2	8.7311E-11
4	5.82077E-10
8	1.862645E-09
16	5.587935E-09
32	2.2351742E-08
64	7.4505806E-08
128	2.98023224E-07
256	1.192092896E-06

5.2 CPU & GPU Times per Image Size

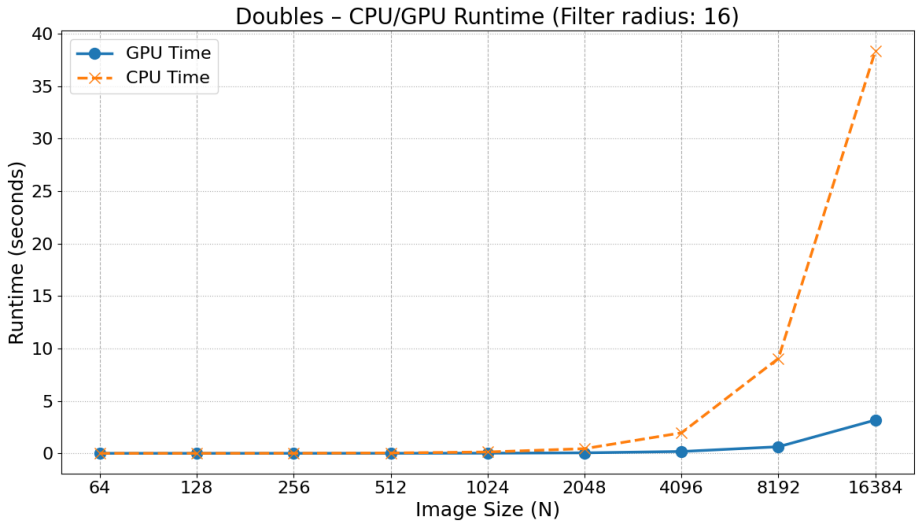


Figure 7: Program execution time on CPU and GPU using doubles.

Image Size	GPU avg (s)	GPU Std dev (s)	CPU avg (s)	CPU Std dev (s)	Speedup
64×64	0.0001738	2.4738E-05	0.000525	1.4184E-04	3.02×
128×128	0.0003289	5.8585E-06	0.0019992	4.6557E-05	6.08×
256×256	0.0011041	2.4118E-05	0.0082182	1.44198E-04	7.44×
512×512	0.0037084	2.2136E-04	0.034054	0.0033225	9.18×
1024×1024	0.0131907	8.6974E-04	0.1278087	0.0151046	9.69×
2048×2048	0.0464747	2.2383E-03	0.4439873	0.0413447	9.55×
4096×4096	0.1740196	1.4832E-03	1.9524106	0.101432	11.22×
8192×8192	0.6261251	3.5972E-02	9.0471454	0.426256	14.45×
16384×16384	3.172902	0.442676	38.3644301	1.369290	12.09×

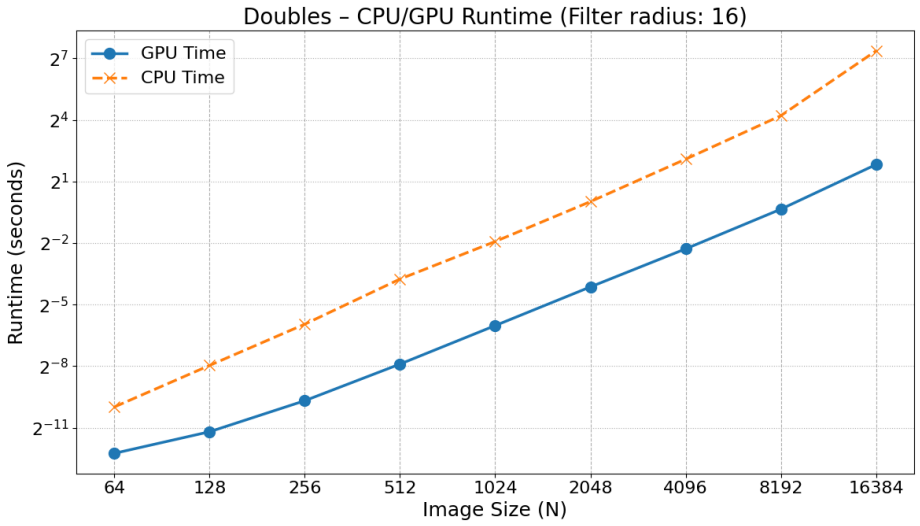


Figure 8: Execution times on a logarithmic scale.

6 Memory Accesses and FLOPS

6.1 Image and Filter Reads

The image elements located at a distance greater than the filter radius from the edges are read as many times as there are pixels in the filter. This happens because, during the convolution process, every time the filter is applied to a pixel, all elements within distance filter_radius from the center of the filter are accessed. Thus, each internal pixel participates in the total number of accesses as many times as the area of the kernel.

The elements of the filter, specifically its center, are accessed as many times as there are pixels in the image, as they participate in every filter application. However, the other elements of the filter beyond the center are not accessed when the filter is at a distance less than filter_radius from the image edges. It is worth noting that cases with fewer accesses do not apply in the padding implementation.

6.2 Memory Accesses to FLOPs Ratio

Looking at the code, we observe that during the convolution execution, we have two memory accesses, one for the pixel value and one for the filter value. Then we multiply these two floating-point numbers and add the result to the sum.

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

    if (d >= 0 && d < imageW)
        // two accesses, two FLOPs
        sum += h_Src[y * imageW + d] * h_Filter[filterR - k];
}
```

Since we have two memory accesses and two floating-point operations, the ratio of memory accesses to FLOPs is 1.

7 Eliminating Divergence

To resolve divergence, we used padding as shown in Figure 9. Below we see the executions for padded and non-padded versions.

In the implementation of the padded code, to save memory, we initially used a cudaTransferPadded function which had considerable overhead due to multiple cudaMemcpy calls. Later, we replaced this function with cudaMemcpy2D, which can be used for this operation.

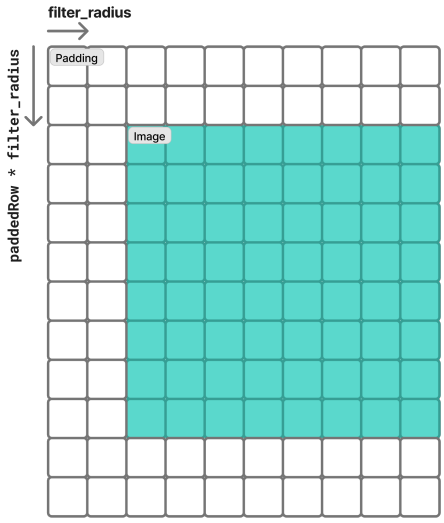


Figure 9: Padded image for filter_radius = 2

```
void cudaTransferPadded (int xPad, int yPad, PixelScalar *device,
                         PixelScalar *host, int arrayWidth,
                         int arrayHeight, cudaMemcpyKind kind)
{
    int paddedRow = (xPad << 1) + arrayWidth;
    int startOffset = (paddedRow * yPad) + xPad;
    PixelScalar *curRowStartD = device + startOffset;
    PixelScalar *curRowStartH = host;
    for (int i = 0; i < arrayHeight; i++) {
        if (kind == cudaMemcpyHostToDevice)
            cudaMemcpy(curRowStartD, curRowStartH, arrayWidth * sizeof(PixelScalar), kind);
        else
            cudaMemcpy(curRowStartH, curRowStartD, arrayWidth * sizeof(PixelScalar), kind);
        curRowStartD += (xPad << 1) + arrayWidth;  //* Skips side padding
        curRowStartH += arrayWidth; //* Go next line on host
    }
}
```

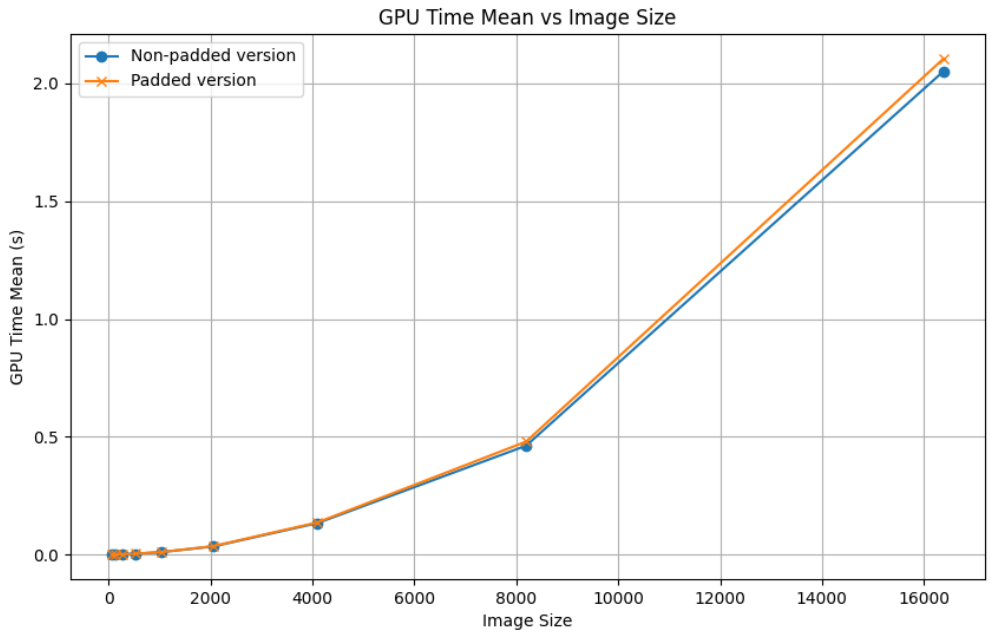


Figure 10: Comparison of execution time with and without padding.

We observe that the execution times do not differ greatly. This is likely due to the fact that although we gain from eliminating divergence, we lose from the more complex transfers required by the padded array. After additional tests on our own system, the desired difference between times was observed.

Filter Radius	GPU Time (Non-padded)	GPU Time (Padded)
128	0.989189	0.920196
256	1.354624	1.228230
512	2.150286	1.900903
1024	3.711623	3.193815
2048	6.827488	5.756322
4096	13.053341	10.914864

Table 1: Execution on a system with `RTX 3060 12GB` on a 16384 array.

Filter Radius	GPU Time (Non-padded)	GPU Time (Padded)
128	3.823996	3.604211
2048	27.961308	23.122805

Table 2: Execution on a system with `RTX 3060 12GB` on a 32768 array.

A Appendix

Result of deviceQuery

```
./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)
  (13) 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 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 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) >

Device 1: "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)
  (13) 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 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 Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 7 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
     simultaneously) >

Device 2: "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)
  (13) 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 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 Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 132 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
     simultaneously) >

Device 3: "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)
  (13) 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 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 Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 133 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
     simultaneously) >
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU2) : No
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU3) : No
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU2) : No
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU3) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU0) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU1) : No
> Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU3) : Yes
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU0) : No
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU1) : No
> Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU2) : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4,
CUDA Runtime Version = 11.5, NumDevs = 4, Device0 = Tesla K80,
Device1 = Tesla K80, Device2 = Tesla K80, Device3 = Tesla K80
Result = PASS
```