High Performance Computing
GPU Programming with CUDA

Στολτίδης Αλέξανδρος, Κουτσούκης Νικόλαος

December 11, 2021

Contents

1 Introduction

High Performance Computing must harvest any available

2 Image - Kernel Convolution

Convolution Kernel are used with images for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between a kernel and an image.

2.1 General Idea

The filter is taking values from around the pixel of interest. These locations range from $(x-1, y-1)$ to $(x+1, y+1)$. It is multiplying those values by the corresponding value in the kernel matrix and then summing up these values to give the new pixel value. This process is performed for almost all the pixels in the image. A major problem for might occur when trying to calculate the value of the pixels located in the edges. When solving this problem a processing overhead play a big role in computation time.

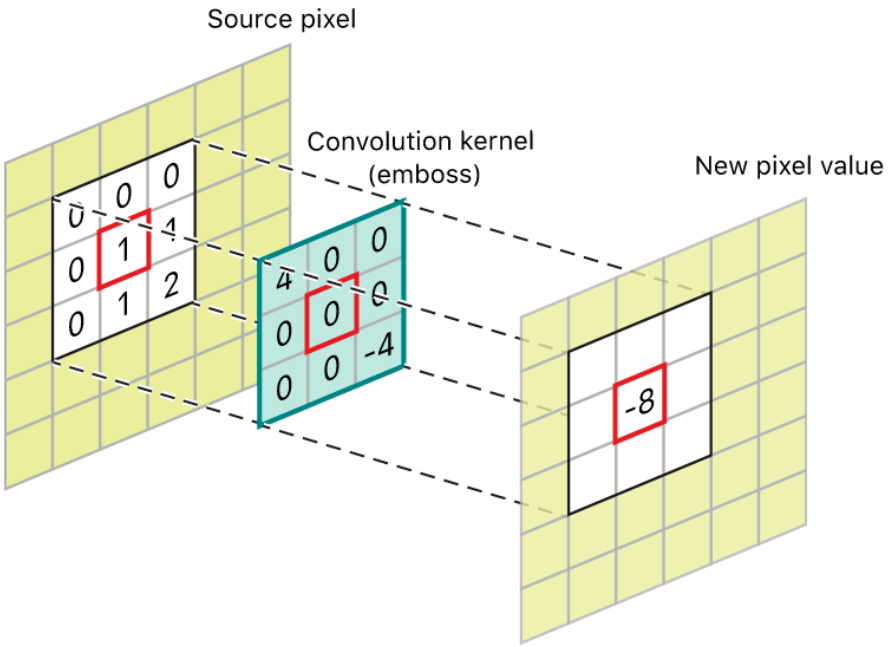


Figure 1: Image Cells that the Convolution Affects

2.2 Data Structures

2.2.1 Image Array

An image can be visualized as a 2D matrix of pixels. This matrix can be stored in a 1D array in memory. The position of rows and columns can be easily calculated by the use of pointer arithmetics.

$$position = row * width + column \tag{1}$$

2.2.2 Kernel - Filter

The filter can be also stored in a 1D array and it's length can be easily calculated from the following formula

$$length = 2 * radius + 1 \tag{2}$$

2.3 Gaussian Blur

Gaussian blur is the result of blurring an image by a Gaussian function. The convolution between an image and a kernel, when both are stored in a matrix, is shown below.

$$r(i,j) = \sum_n \sum_m s(i-n, j-m)k(n,m) \tag{3}$$

When the image and the kernel are stored in single dimensional arrays the convolution is as follows.

$$r(i) = \sum_n \sum_m s(i-n)k(n,m) \tag{4}$$

The convolution is executed as shown in the images below.

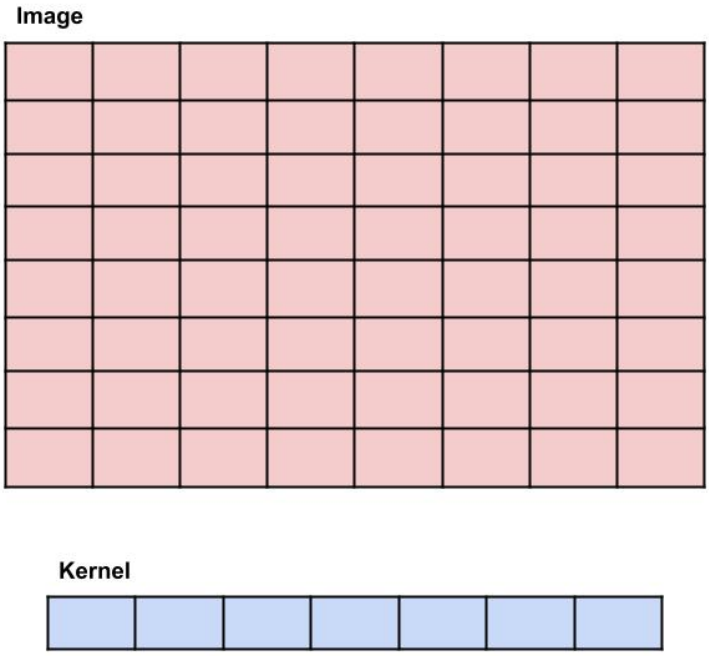


Figure 2: Image and Kernel from Programming Perspective

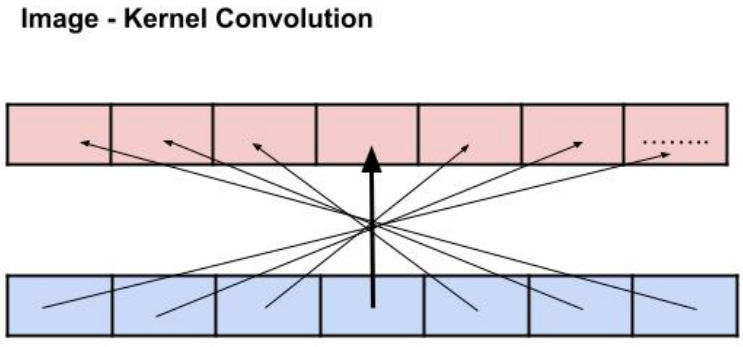


Figure 3: Image and Kernel Memory Allocation and Convolution

3 Single Block - Multiple Threads

Each GPU thread is responsible to calculate the convolution of each image pixel with the kernel. Threads are organized in a two dimensions. There are N vertical and M horizontal GPU Threads, where N and M are equal to the height and the width of the image respectively.

3.1 Maximum Supported Image Size

Each GPU block can handle at most 1024 threads. This information can be easily extracted from the "deviceQuery" command. Since each thread is responsible for it's own image cell, the maximum number of cells must also be at most 1024. Moreover, the image is a two dimensional array and the height is the same as the width. Thus, $width * height \leq 1024 \Rightarrow width = height \leq 32$. For images larger than $32 * 32 = 1024$ cells an "invalid argument" error will occur since a single block can not handle that many threads.

3.2 Accuracy for Different Kernel Size

The maximum image size that a block can support is $32 * 32 = 1024$ cells. For different filter lengths we observe how the accuracy varies with the graph bellow.

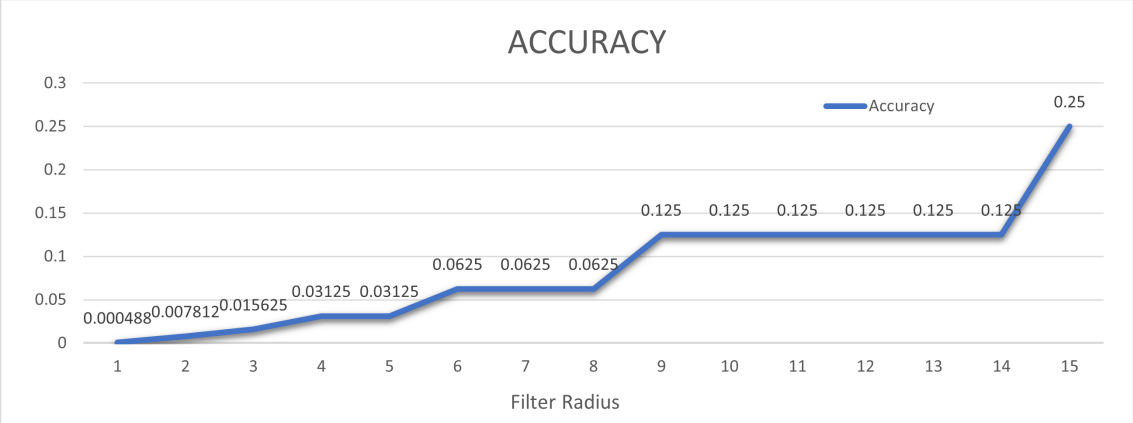


Figure 4: Accuracy with Maximum Image Supported and Variable Kernel Sizes

As it can be observed the accuracy criteria are not met for filter sizes larger than one. This can be attributed to the fact that floats are used in this instance for calculations. Floats provide fast single precision arithmetics, but do not guarantee accuracy. Later into the project the float data type is replaced by double, which handles double precision calculations.

4 Multiple Blocks - Multiple Threads

Images with 1024 pixels are not that large and can be easily supported by the CPU. Larger image sizes must be supported by the GPU in order to see a significant performance boost compared to the CPU. To support larger images we implement a two dimensional grid of blocks. Each block contains at most 1024 threads. These threads are also organized in two dimensions.

4.1 Block and Grid Geometries

4.1.1 Block of Threads Dimensions

$$block(x) \Rightarrow \begin{cases} x = width,\ width \leq 1024 \\ x = 1024,\ width > 1024 \end{cases} \qquad block(y) \Rightarrow \begin{cases} y = height,\ height \leq 1024 \\ y = 1024,\ height > 1024 \end{cases}$$

4.1.2 Grid of Blocks Dimensions

$$grid(x) \Rightarrow \begin{cases} x = 1,\ width \leq 1024 \\ x = \lceil \frac{width}{1024} \rceil,\ width > 1024 \end{cases} \qquad grid(y) \Rightarrow \begin{cases} y = 1,\ height \leq 1024 \\ y = \left\lceil \frac{height}{1024} \right\rceil,\ height > 1024 \end{cases}$$

Since a ceiling operation is used when calculating grid dimensions, the total amount of threads might be more than the actual image pixels. For example, if an image has 1025 cells, the total number of threads per block is 1024 and the total blocks are two. So $2 * 1024 = 2048$ threads are available. However the image only has 1025 pixels and $2048 - 1025 = 1023$ threads must not be used and should instantly return to avoid invalid memory accesses. A condition at the start of each kernel checks if the current thread is not outside of the image array. This amplifies the divergence since many threads must execute different instructions at the same time.

4.2 Maximum Supported Image Size

Each GPU block can handle at most 1024 threads. The maximum number of blocks is 2,147,483,647. This information can be easily extracted from the "deviceQuery" command. Since each thread is responsible for it's own image cell, the total amount of threads that the GPU can support is $2147483647 * 1024 = 2,199023255 * 10^{12}$ which is equal to the total image pixels supported by the GPU. To find maximum width and maximum height we just calculate the square root of the total pixels. In conclusion, $width_{max} = height_{max} = \sqrt{2,199023255 * 10^{12}} = 1,482,910,400,192,810$.

4.3 Accuracy for Different Kernel Size

For different filter sizes the accuracy can be observed bellow.

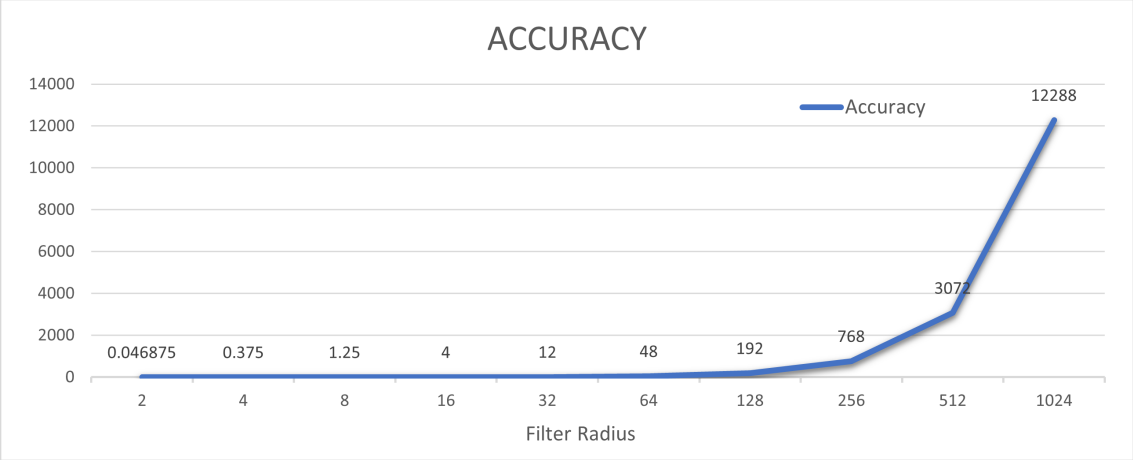
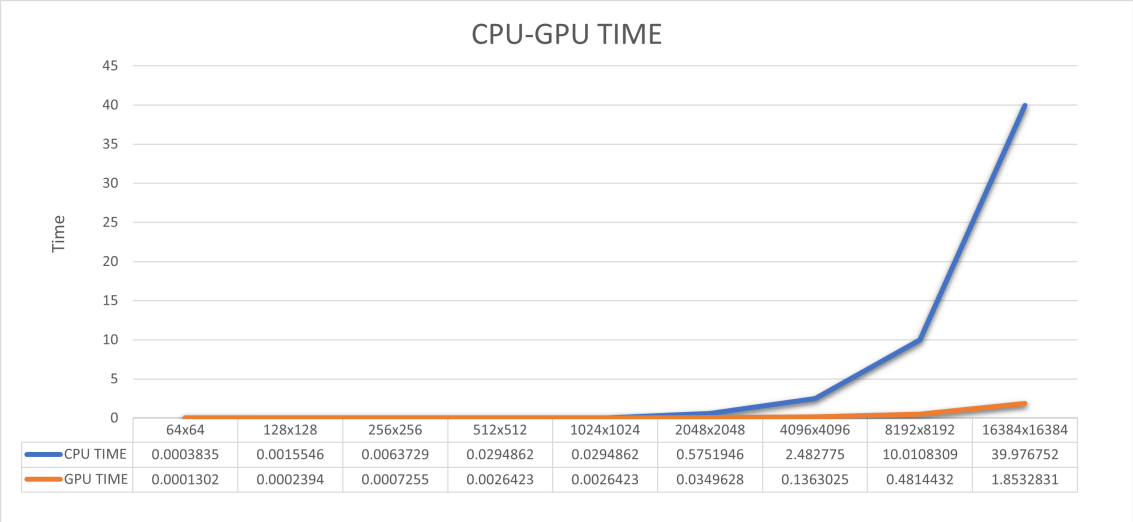


Figure 5: Accuracy with Maximum Image Supported and Variable Kernel Sizes

When the filter size increases the accuracy decreases dramaticaly. This can be attributed to the fact that floats support single point precision operations. When the float datatype is replaced by double the accuracy must remain the same.

4.4 CPU vs GPU

Our hypothesis was that with larger images the performance boost that the GPU offers is immense compared to the CPU. When executing on CPU we only measure the duration of the Gaussian Blur's computation. When executing on GPU we measure the duration of both memory transfers and actual computations. Finally, for both cases, we do not measure the duration of memory allocations on the host's main memory.



	64x64	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096	8192x8192	16384x16384
CPU TIME	0.0003835	0.0015546	0.0063729	0.0294862	0.0294862	0.5751946	2.482775	10.0108309	39.976752
GPU TIME	0.0001302	0.0002394	0.0007255	0.0026423	0.0026423	0.0349628	0.1363025	0.4814432	1.8532831

Figure 6: CPU vs GPU for Variable Image Sizes and Kernel Radius 16

As it can be observed, the GPU outperforms the CPU in all cases. Here we must also note that the CPU code runs completely sequentially and thus it is not optimized at all. If CPU optimizations are applied the GPU would still be faster since more threads are available and the actual computation includes mostly arithmetical operations and not logic, where actually the CPU excels.

5 From Single to Double Point Precision

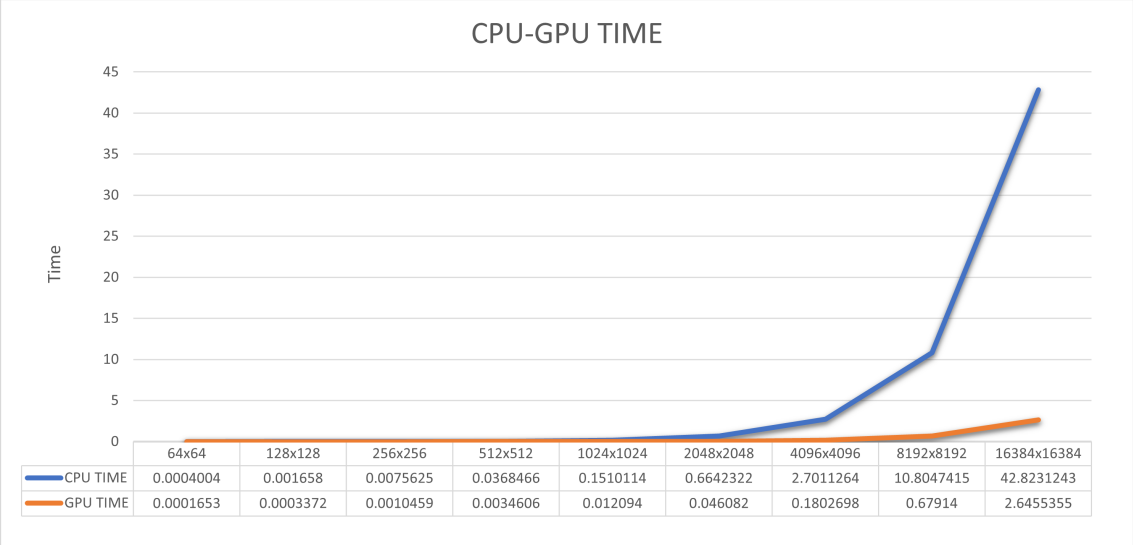
When replacing the floats with doubles we increase the accuracy that the ALUs can support. Doubles take up twice the space that floats do and thus their precision is increased.

5.1 Accuracy for Different Kernel Size

Indeed when using doubles the accuracy remains steadily at 0.00005 which is the accuracy we wanted to achieve.

5.2 CPU vs GPU

Both CPU and GPU must handle calculations with doubles, which are twice the size of a float. Using doubles instead of floats is a trade-off between processing speed and accuracy. A small overhead in performance can be seen below.



	64x64	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096	8192x8192	16384x16384
CPU TIME	0.0004004	0.001658	0.0075625	0.0368466	0.1510114	0.6642322	2.7011264	10.8047415	42.8231243
GPU TIME	0.0001653	0.0003372	0.0010459	0.0034606	0.012094	0.046082	0.1802698	0.67914	2.6455355

Figure 7: CPU vs GPU for Variable Image Sizes and Kernel Radius 16

When the image size increases the total multiplications and additions between doubles also increase. The more operations executed the bigger the overhead is. When the image size is $16384 * 16384$ and floats are replaced by doubles, the GPU run-time increases by almost 30% compared to the previous results. This overhead however is partly attributed to the actual computation. As it can be observed, most of the time is spent in memory operations and more specifically, operations that copy data from host to device and back.

6 Memory Reads and FLOP/Memory Reads

6.1 Image Memory Reads

Both row and column convolutions do the same calculations on different dimensions. Thus, we can calculate the memory accesses only for one type of convolution and multiply the result by two to find the actual memory accesses. An instance of row convolution can be seen bellow.

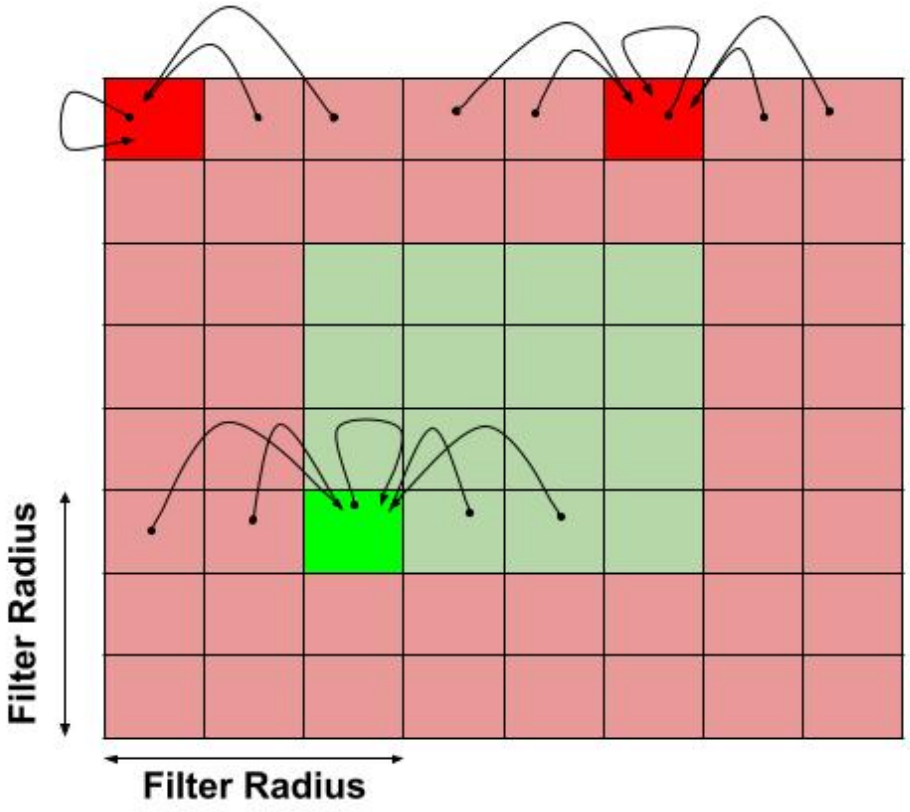


Figure 8: $radius < x, y < size - radius$ $x, y < radius$ or $x, y > size - radius$

In conclusion the image memory accesses, both from row and column convolution, are

$$convolution(x) \Rightarrow \begin{cases} Filter\ Length = 2 * Filter\ Radius + 1,\ radius < x < width - radius \\ x + \left\lceil \frac{Filter\ Length}{2} \right\rceil,\ x < radius \\ (width - x - 1) + \left\lceil \frac{Filter\ Length}{2} \right\rceil,\ x > width - radius \end{cases} \quad (1)$$

$$convolution(y) \Rightarrow \begin{cases} Filter\ Length = 2 * Filter\ Radius + 1,\ radius < y < height - radius \\ y + \left\lceil \frac{Filter\ Length}{2} \right\rceil,\ y < radius \\ (height - y - 1) + \left\lceil \frac{Filter\ Length}{2} \right\rceil,\ y > height - radius \end{cases} \quad (2)$$

From (1) and (2) we can calculate the convolution(x,y)

6.2 Kernel Memory Reads

Kernel memory reads can be calculated with the same formulas as the image memory reads. Thus, we must calculate the average memory reads in order to calculate the FLOP/Memory Reads.

6.3 FLOP/Memory Reads

In order to calculate the correct sum at each iteration, we must first fetch both an image pixel and the correct filter cell. Then we multiply them together and store them into a register. To do one multiplication we fetch two elements from memory. Therefore, the FLOP/Memory Reads is equal to 1/2.

7 Padding Image and Reduce Divergence

Each streaming multi-processor has 16 ALUs and can execute 32 operations in 2 Clock Ticks. Each SM is responsible to execute arithmetical operations for one warp at a time. These warps must execute the exact same instruction with different input data. Warps can not execute different instructions at the same time. If they try, they must wait for their turn and thus divergence occurs. In order to solve the divergence problem we must get rid of conditional statements or make sure that our warps follow the same path on each conditional statement and execute the same instructions. In our case we completely remove conditional statements by adding some padding to the image array.

7.1 Before Padding

We start from [0][0] and we finish at [height - 1][width - 1].

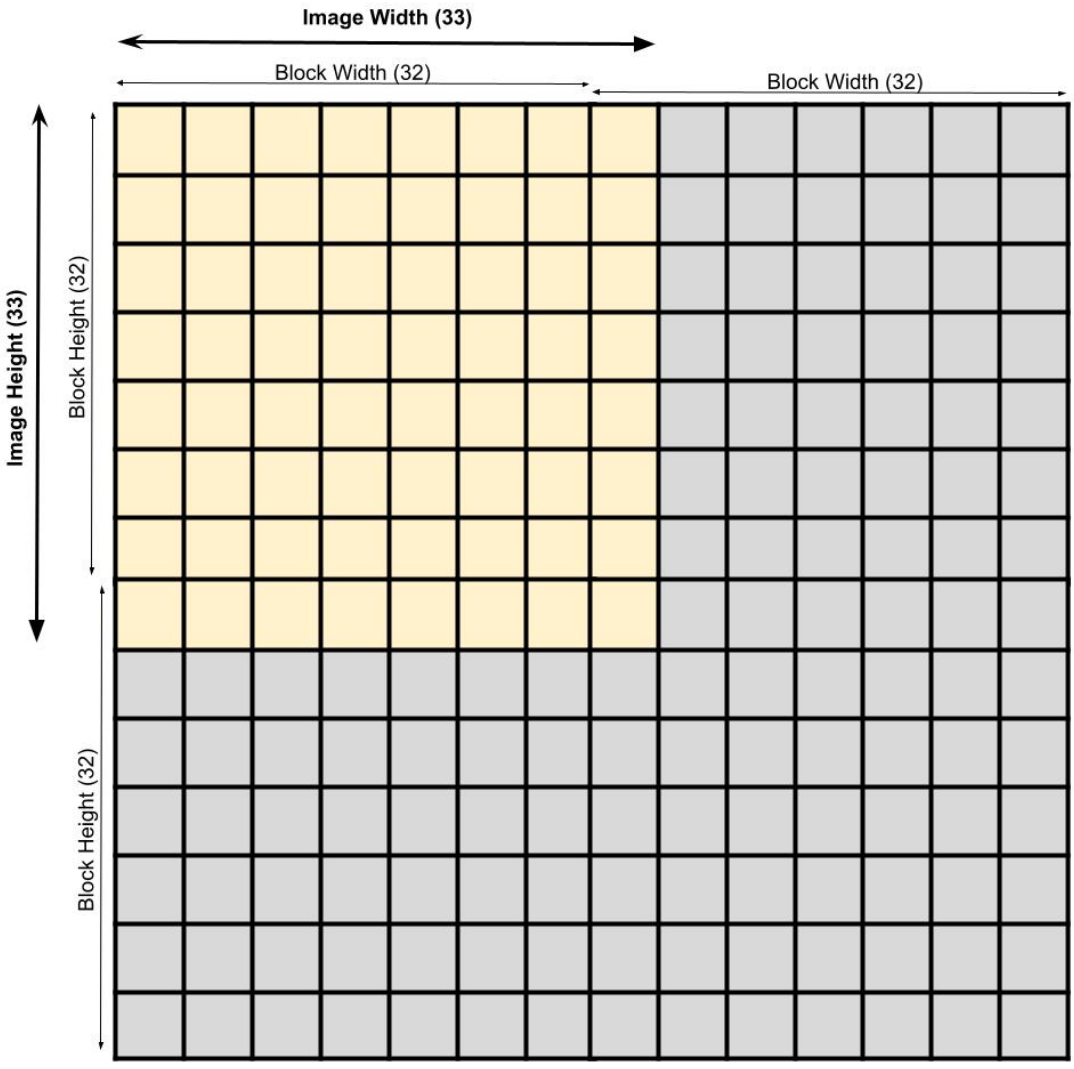


Figure 9: Image 33x33 Pixels, 4 Blocks of 1024 Threads with Grey Threads Unused

Each thread checks it's position inside the image. If they are out of bounds then they must return immediately. This happens when the $width \mod 1024 \neq 0$ or $height \mod 1024 \neq 0$ and an additional block must be created to handle the remaining cells.

In addition, filter convolution must insure that the correct filter cells are combined with their corresponding image pixel. Thus, the actual filter convolution with the image must avoid accessing invalid memory addresses outside of the image.

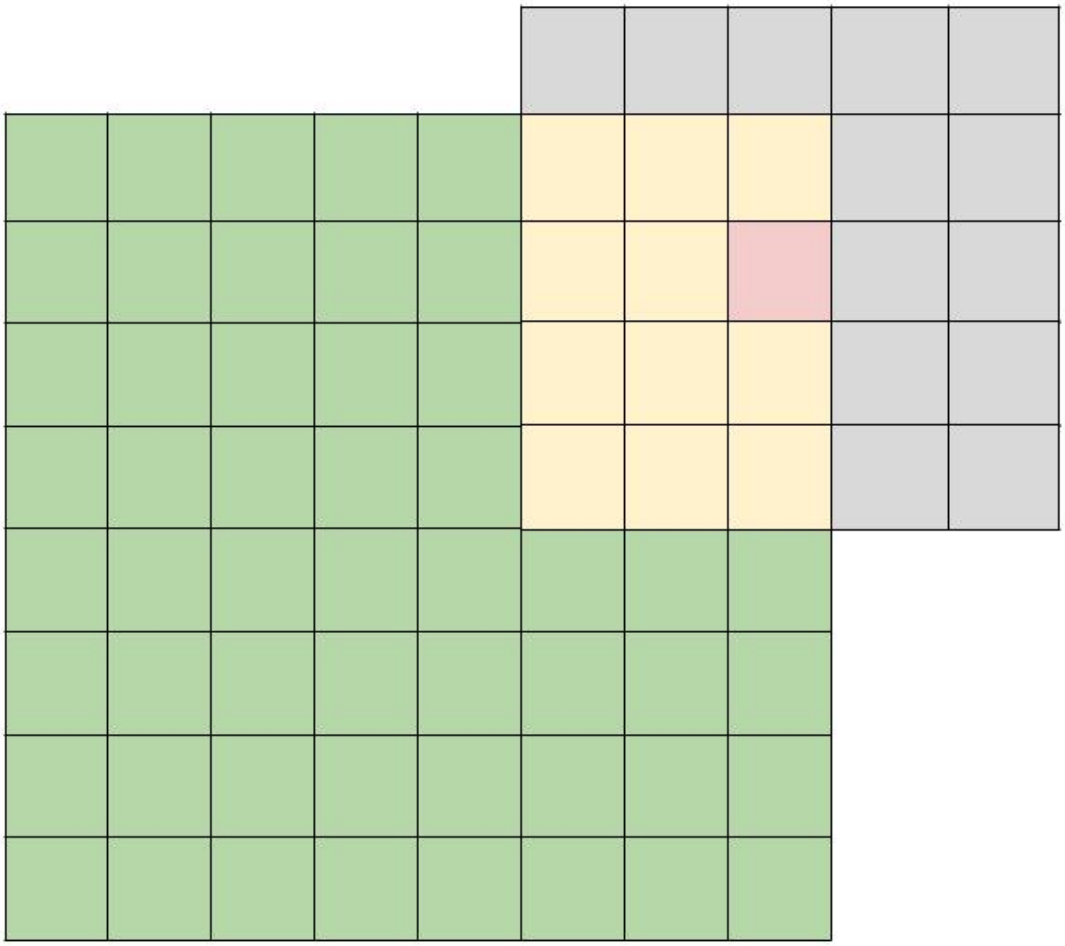


Figure 10: Convolution to Calculate the Pink Pixel

Neighboring Cells Read by Filter to Calculate the Convolution

Unused Filter Cells Outside of the Image

As we can see additional conditional statements must be inserted in order to insure that the filter always remains inside the correct memory bounds. This increases the divergence even more and must also be avoided.

7.2 After Padding

To reduce divergence and avoid a few of these conditional statements above we add a little padding to the image. We populate the padded section with zeroes and keep the image intact. The geometry of the new padded image depends mainly on filter's dimensions. In addition, if more threads are created than the actual image pixels then the padding must modified to support the additional blocks.

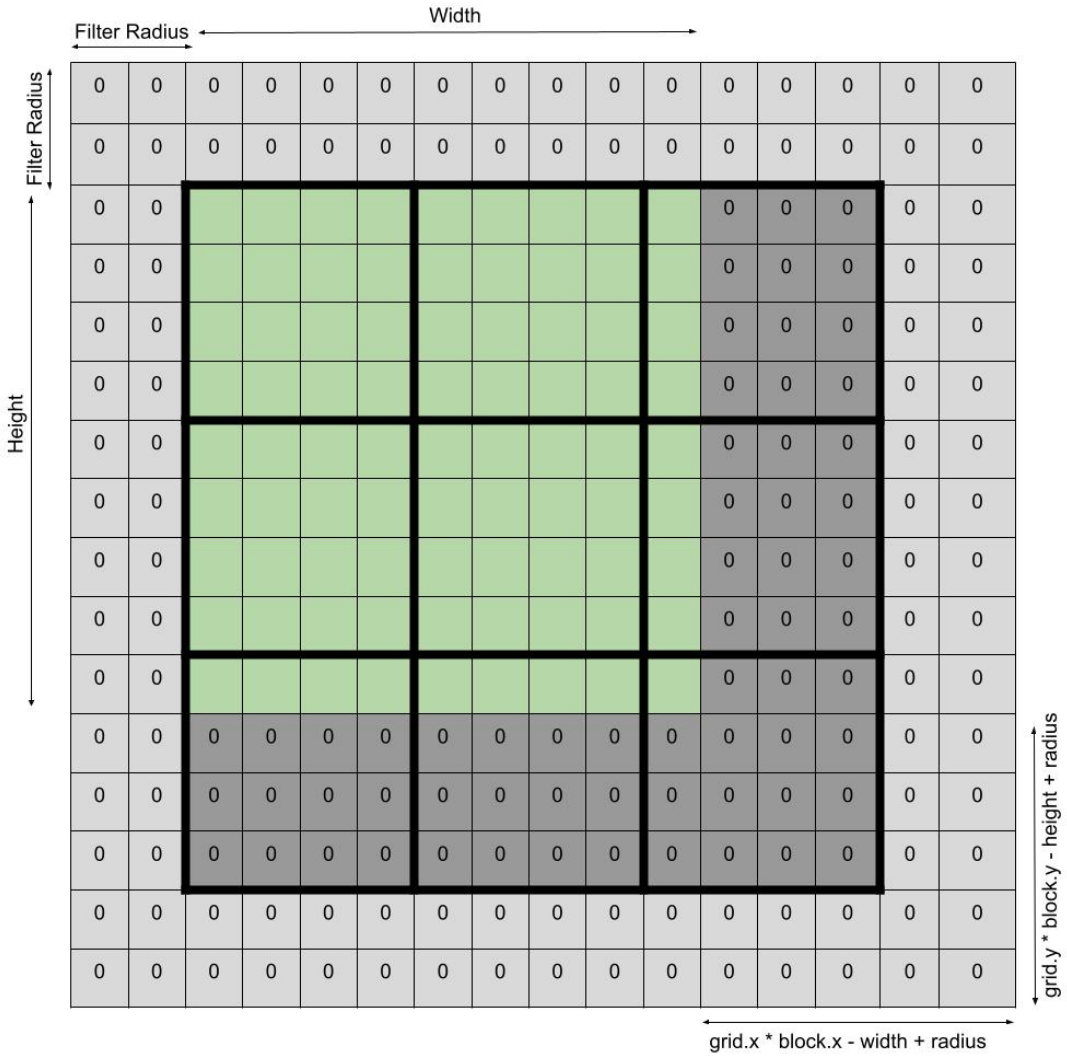
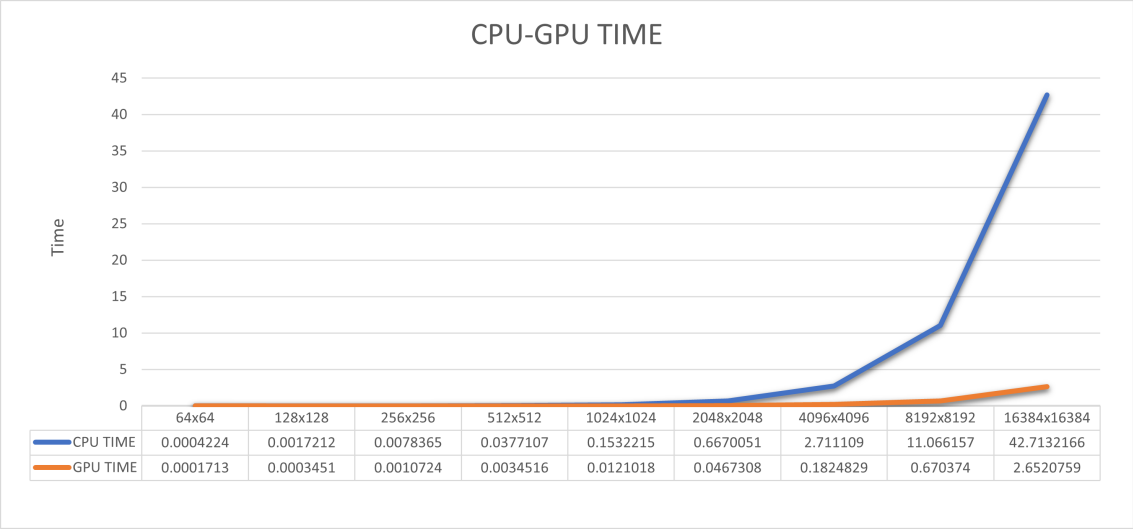


Figure 13: Blocks of Threads Applied both on Green and Gray Padded Pixels

Image is consisted only by the green pixels. When the filter is applied, on an edge pixels of the image, the conditional statements are avoided since multiplications with zero cells of padding are allowed and do not alter the final result. When more threads than existing pixels are spawned they perform calculations outside of the image using only the padding cells. This is not a big problem since it allows us to avoid the additional conditional statements that insure that the threads remain in bounds. It is a trade-off between more computation and less divergence.

7.2.1 Analysis

Finally, we benchmark our code to understand if these optimization actually are able to give any significant performance boost.



	64x64	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096	8192x8192	16384x16384
CPU TIME	0.0004224	0.0017212	0.0078365	0.0377107	0.1532215	0.6670051	2.711109	11.066157	42.7132166
GPU TIME	0.0001713	0.0003451	0.0010724	0.0034516	0.0121018	0.0467308	0.1824829	0.670374	2.6520759

Figure 14: CPU vs GPU for Variable Image Sizes and Kernel Radius 16

As we can see after all these optimization the execution is actually a little slower. This can be attributed to the fact that even with some conditional statements the warps of 32 threads find the correct combination to execute the same instructions in parallel. In conclusion, since divergence does not only depend on the amount of conditional statements but also on the combination of warps that execute the same instructions, avoidance of conditional statements does not guarantee performance optimizations.