HPC - Lab3

Zerdalis Christodoulos 03531 - Malakoudis Charalampos 03516

Intro

In this lab we are focusing on the cuda environment and more specifically trying to parallelize and run a convolution function over some artificial “images” on the gpu(’s) of the Csl-venus machines.

Step 0

The first step of the lab requires us to run the deviceQuery function that displays useful architecture info about the GPU we are using. To do that we cloned the official NVIDIA repo of cuda samples https://github.com/NVIDIA/cuda-samples?tab=readme-ov-file , since the files usually located locally on the machine were either missing or corrupted. We built the necessary files and run the deviceQuery. The results are in the deviceQuery.txt file in our zip folder.

Step 1

In this step we are asked to study the nvcc execution parameters. Later on we decide to use the parameters **--ptxas-options="-v"** and **-G** in order to get valuable information about our program and how the hardware runs it. Also -G is used to test the program's functionality as it deactivates the optimizations and thus the rounding of the results. In the previous step we found out that the Computational Capability of our GPU is 3.7 and with cross references from the NVIDIA documentation and stack Overflow decided that we should add the flag **-gencode arch=compute_37,code=sm_37** which finetunes the compilation for our specific GPU. From our understanding we could also simply use **-arch=sm_37** as it would have the same effect (we actually did only use this parameter). We also used -O4 as instructed.

Step 2

In this step we implemented the first edition of the algorithm (**v1**) that runs in the GPU. We were asked initially to make the implementation in a way that the whole image will

be processed by **one** CUDA block and every thread will compute the value of one pixel. Thus we create a block with dimensions (**imageW, imageH**) and a grid(**1,1**). Imitating the convolutionCPU functions we implemented **two kernels**: one for the convolution per row and one per column. Since each thread does calculations for one final pixel, we don't write to overlapping memory and thus we can implement the kernels without any synchronization between the threads. We move the data back to the GPU, run the two kernels, copy the results back and compare with the CPU results to see if they match

For testing purposes the program doesn't stop if differences were found.
The compilation was done with the parameters we picked in the previous step.

Step 3

A)

Although this step is the one we should do the experiments with, the issues became evident before we reached it. It seems that regardless of the filter radius , the max image size this version can support is **32 * 32**. By putting our thinking hats on and taking a look at the specifications for the Tesla K80 GPU we found the following sign :

"***Maximum number of threads per block: 1024***"

Since we are using a thread for each resulting pixel we cannot have more than 1024 resulting pixels. For the 4th 5th and 6th power of 2 the resulting image sizes come out to 256, 1024 and 4096 pixels. That means that for any image **size > 32** we exceed the 1024-thread/block limit, and CUDA refuses to launch the kernel with the message "**invalid configuration argument**"

B)

In order to find the maximum accuracy value (in digits) for which our program can currently run flawlessly, we need to calculate the largest differences produced by each combination of filter radius and the max size (32). Thus we got the following results for filter radius of 1->15:

Index	Value	Index	Value
1	0.007 812	9	1.000 000
2	0.015 625	10	1.000 000
3	0.062 500	11	1.000 000
4	0.187 500	12	1.000 000
5	0.250 000	13	1.500 000
6	0.375 000	14	1.000 000
7	0.500 000	15	1.000 000
8	0.750 000		

From these almost naturally sorted values we can see that for really high filter radius values the largest differences are up to 1.5 , which is a huge number. Thus we can conclude that the maximum usable accuracy threshold is as follows: for filter value of 1-> **two digit** accuracy, from 2-3 -> **one digit**, from 4-8 **hardly** one digit and the rest **zero digits**. After some research we came to the conclusion that these large deviations are due to some fundamental differences between the way each piece of hardware handles the computations. On GPUs, floating-point operations are often encoded with the **rounding mode** directly in each instruction, and the hardware lacks the same floating-point status/control word mechanism as x86 CPUs. Also, GPUs may use **multiply-add (MAD)** or combine operations in a way CPUs do not, changing the intermediate values and thus the final result.

https://docs.nvidia.com/cuda/floating-point/index.html
https://forums.developer.nvidia.com/t/cpu-and-gpu-floating-point-calculations-results-are-different/18175/2

Step 4

In this step we are asked to fix the main limitation of the **v1** implementation. As we saw in Step 3, version 1 can only support image sizes up to **32×32** because the entire image was assigned to a single CUDA block, and the **Tesla K80** only allows 1024 threads per block.

To remove this restriction, in **v2** we change the way we launch the kernels. Instead of forcing all the threads into one block, we split the image into multiple blocks arranged in a 2D grid as instructed. Each block now processes only a small part of the final image. The block size is chosen to be **(32,32)** as this results in the maximum allowed threads, and the grid dimensions are computed from the image size. This way the number of threads per block

stays within the hardware limit, while the total number of threads (block count × block size) can now scale to much larger images (128, 256, 512, etc).

The kernels themselves do not change much logically. Each thread still computes one output pixel using the same convolution formula. Since we now launch multiple blocks and each block is a fixed size, the last block in each row/column may contain threads that point outside the valid image range. Thus we had to add a **boundary check**, in order to prevent those threads from trying to read or write beyond the array bounds, causing invalid memory accesses and undefined behavior.

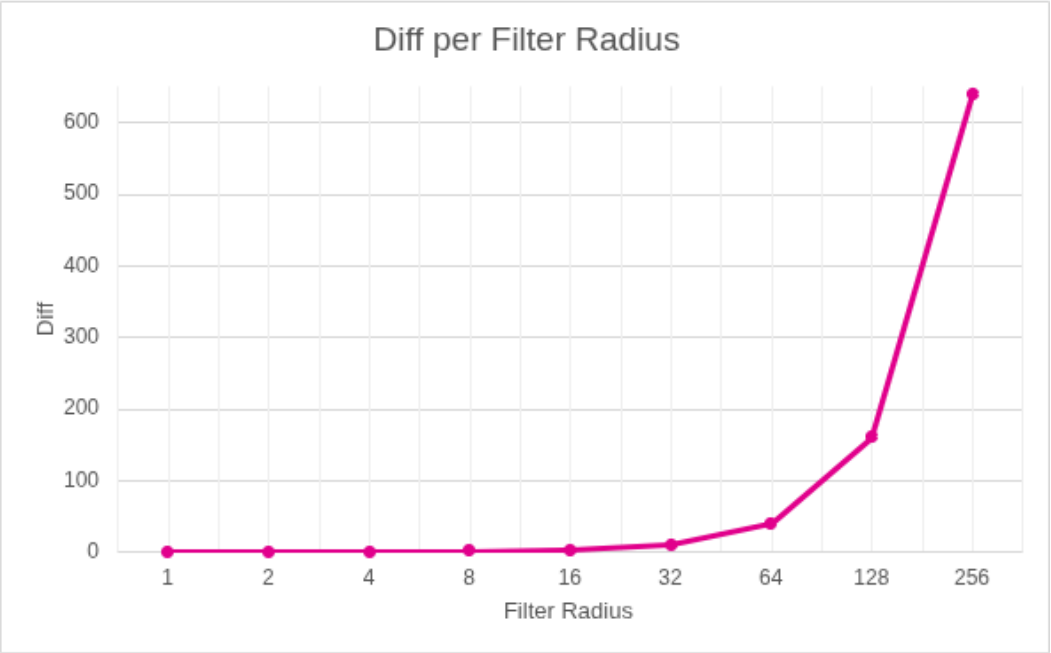
During our experiments we tested how big of an image our program could handle and for a filter radius of **4** we were able to reach a size of **16384**! After this number, using **32768** (which is the next power of 2) produces an out of memory error for CUDA. The Tesla K80 GPU has a global memory of **~ 11.5 Gigabytes**. For the GPU we allocate 3 float arrays , each of size imageH*imageW* sizeOf(float). Since a float is 4 bytes and our imageSize is 32768 we get 32768 * 32768 * 4 = **4294967296** bytes. 1 Gigabyte translates to 1073741824 bytes and thus each array takes up almost **4GB of space**! For 3 arrays that means we need **at least 12GB** of space for the arrays alone, which is not possible with our current GPU. Of course any size larger than 8k is already way too big for realistic images, so the 32k size we tested is already quite far fetched!

Step 5

A)

In this step we are asked to make a graph of the maximum digit accuracy we can have so that our program runs smoothly. Just like step 3 question b, all we have to do is find the maximum difference that is produced for each combination of image size and filter radius. In this case we have a standard image size of **1024** and varying filter radius.

Because the value of the diffs are so large, it's more understandable to plot the values as the actual difference, rather than digit precision. The higher the line becomes, the accuracy lowers!



Just like step 3 the precision pivots from double digit for filter radius of 1, single for 2, barely single for 4 and the rest have max differences way larger than 1. What is interesting is that by taking a look at the numbers , or at the end of the graph where it is more visible , we can see that with each increase in filter radius , the difference becomes **roughly 4 times** larger!

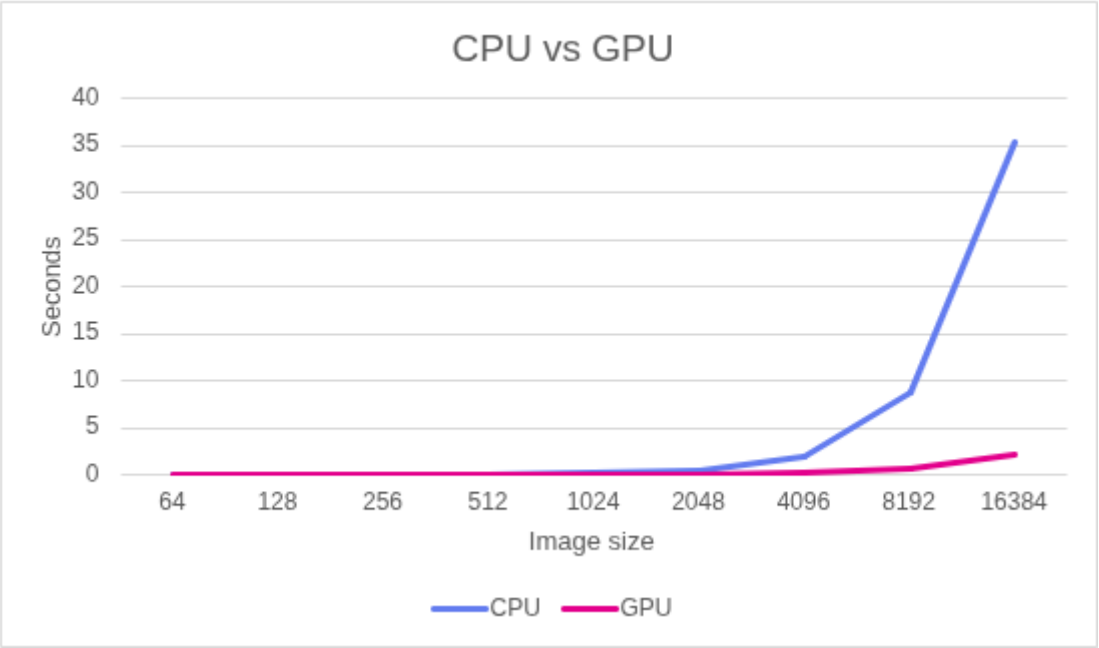
0.007812	0.046875	0.25	1	3	10	40	160	640

This might be happening because doubling R roughly doubles the number of additions, twice (row pass + column pass), and the rounding error roughly multiplies by 4.

B)

We performed the time calculations as instructed , we used the ready helper functions in order to time the GPU and simple timing for the CPU. We ran each image size about 12 times before getting the average of the results. We got the following graph :

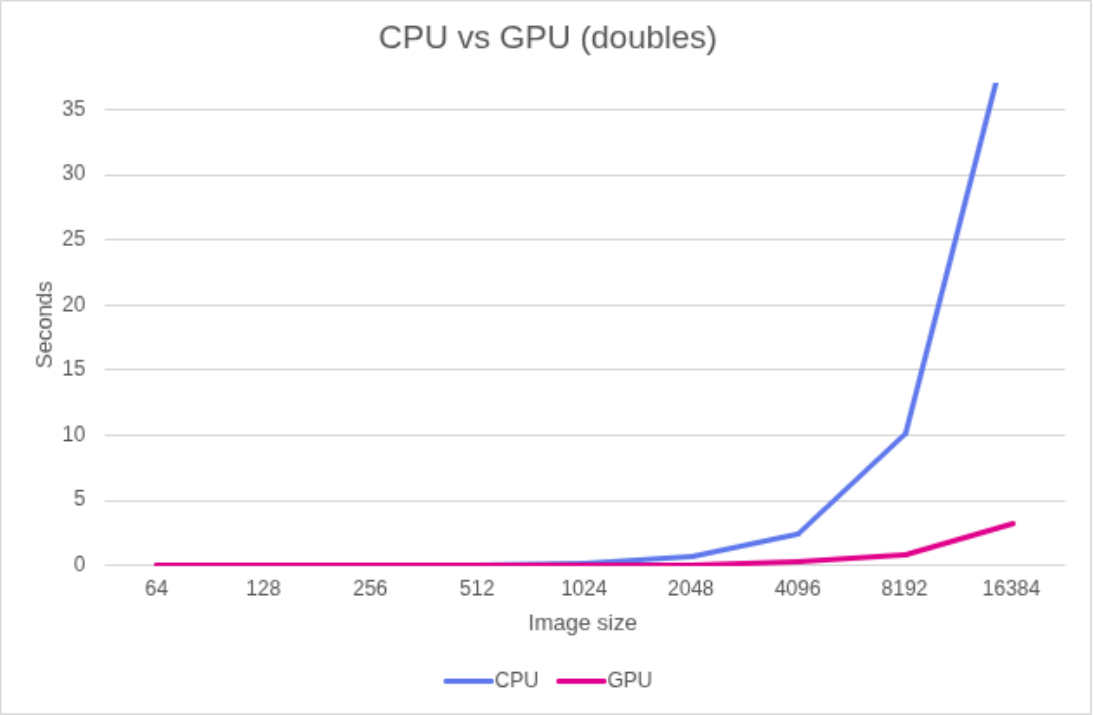
We can see that the CPU performed relatively similarly to the GPU (The GPU was 10 times faster but for small numbers we can't really notice it in the graph) and as the image size rises we can see how much the GPU **outscales** the CPU in terms of speed!



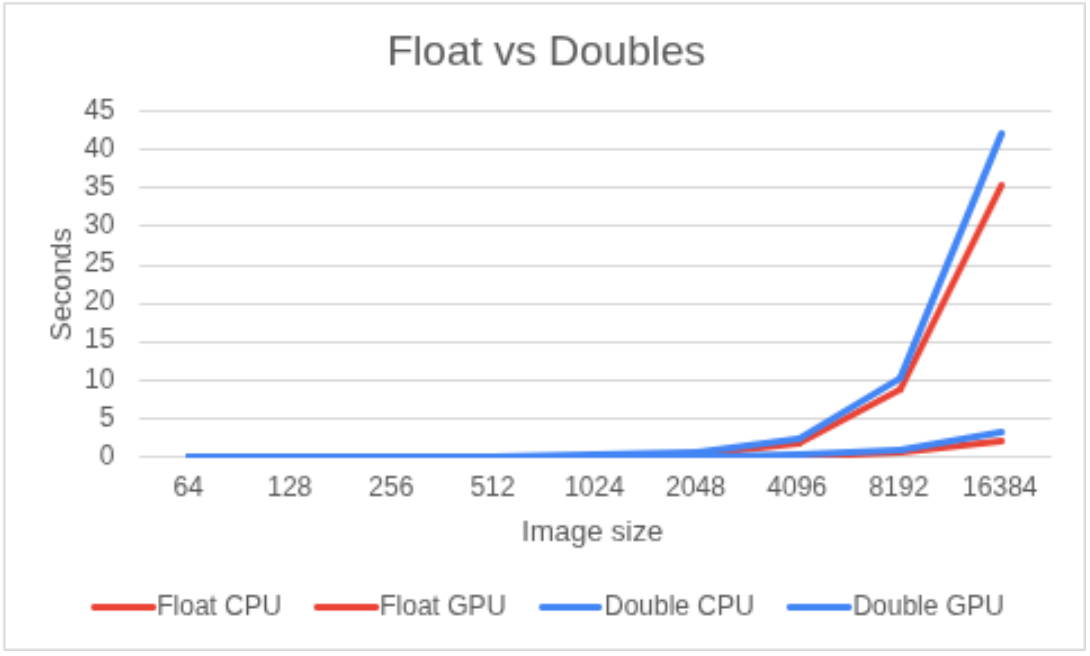
Step 6

In this step we are asked to replace the type of the internal variables from floats to doubles. The code for this step is **v3**. For the first experiment that we were asked to do in step 5 we ran the tests all over again and each run had **0 number of errors**! This is due to the fact that double precision numbers carry much more accuracy, around **15 decimal digits** instead of **float's 7**. In practice, this means that all the tiny rounding differences that normally appear when the CPU and GPU add numbers in a different order simply disappear, because double precision has enough "room" to represent the intermediate values accurately. The CPU and GPU may still execute the operations in slightly different sequences, but with doubles those differences no longer matter.

For the second experiment we got the following chart :



While it's not necessarily visible from the chart itself , the new times we recorded were **roughly 20%** higher than the ones with the float variables. This is normal and expected in our case since the GPU and CPU have to do twice the amount of work for every operation. A double is 64 bits instead of 32, so the hardware needs more cycles to load, move, and compute with the numbers.



Step 7

A)

In our implementation every thread is responsible for computing exactly one output pixel, and to do that it must read all the neighboring values inside the filter window. Because our convolution is separable, this happens in two passes: one along the rows and one along the columns. In the row kernel each thread reads at most **(2R + 1)** input elements, and the same happens again in the column kernel. That means that for each output pixel the thread reads up to **2·(2R + 1)** values from the input image in total. The same logic applies to the filter: each thread needs the entire one-dimensional filter for every pixel it computes, so the filter is effectively read (2R + 1) times in the row pass and (2R + 1) times again in the column pass.

For pixels near the edges of the image, the situation changes. Because part of the filter falls outside the image boundaries, fewer valid convolution windows read these border pixels. For a pixel at position (x, y) we can see that:

- **Horizontally**, it participates in windows centered from $x - R$ to $x + R$, but clipped to the image range $[0, N - 1]$.
- **Vertically**, it participates in windows centered from $y - R$ to $y + R$, also clipped to the same range.

Thus, the number of times a pixel (x, y) is actually read is:

For the x axis :

$If\ (x <= R) -> 2R + 1 - x$

$If\ (x+R >= N) -> 2R + 1 + x - N$

For the y axis :

$If\ (y <= R) -> 2R + 1 - y$

$If\ (y+R >= N) -> 2R + 1 + x - N$

For central pixels:

$(2R+1)^2$

The same logic applies to the filter coefficients: the central coefficients $(k = -R \dots +R)$ are used in every convolution window, while the coefficients near the edges of the filter (those with |k| close to R) are used fewer times because they fall outside the image more often.

Let's assume $n \in [-R, R]$:
For a single row the central element ($n = 0$) of the filter array is read N times,
The next element ($n = 1$) is read $N - 1$ times, because it gets clipped at the end.
The next is read $N - 2$ times and so on until the last which is read $N - n$ times.

This pattern is the same for the other direction so its safe to say that for a single row the number of times that a filter element has been read is $R(n) = N - |n|$. Since the image is NxN

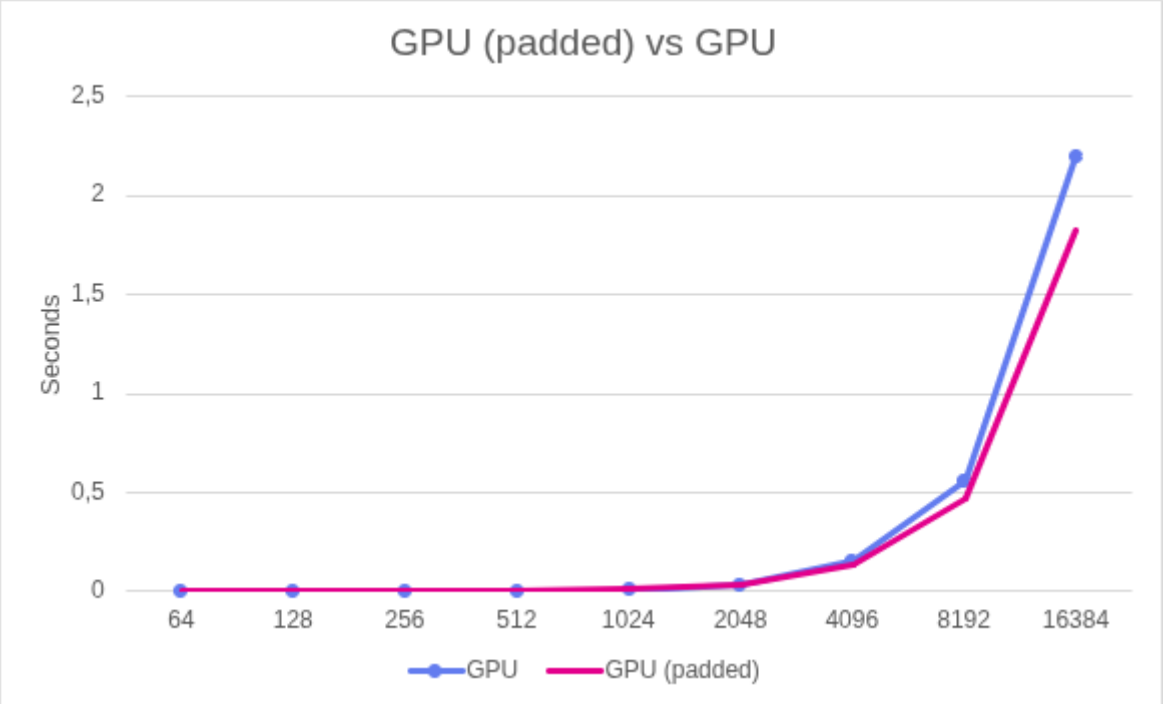
size, we can generalize that for all the rows its $R`(n) = (N- |n|) * N$ and since this behavior is exactly the same but for the y axis we can confidently say that the number of times the filter array is read is $R``(n) = 2 * [(N - |n|) * N]$ with $n \in [-R, R]$!

B)

In our implementation each thread computes exactly one output pixel, and for every value it needs from memory it performs one multiplication and one addition. Basically , for every pixel and filter fetch we perform one multiply and one addition, so the number of global memory reads matches the number of floating-point operations, giving us a memory-to-compute ratio of 1.

Step 8

In this step we tried to eliminate the warp divergence that appears in the previous implementations. The code for this step is in **v4**. To fix this problem, we used padding: we created a larger temporary image where we surrounded the original image with a border of zeros whose width equals the filter radius. This way, every thread no matter where it is in the image, can safely read all filter elements without performing any boundary checks, since the padded region provides valid values. The convolution kernel now sees a perfectly regular grid of data and never needs an if to check bounds. This removes divergence inside the warps and allows the GPU to execute the convolution more efficiently.



Although padding removes the branch divergence at the borders of the image, the performance difference compared to the previous version is not huge. The reason is that our convolution kernel is memory-bound: the GPU spends most of its time performing global memory reads, not arithmetic. Therefore, removing the if conditions at the borders does not reduce the number of memory accesses and has almost no impact on runtime. As a result, the padded version (v4) is just slightly faster than the non-padded version. Maybe for more computationally expensive tasks, we could really see the difference!