Optimization for 2D-3DConvolution Algorithms in CUDA

Burak Topçu
Computer Engineering Department
Izmir Institute of Technology
buraktopcu@iyte.edu.tr
GitHub: https://github.com/topcuburak/ceng443_FinalProject

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Convolution is one of the most widely used filtering algorithms, especially for images. In the last decade, the importance and usage of convolution algorithms increased because of the popularity of deep neural networks. Most of the contemporary deep neural network architecture includes convolutional layers to be used for different purposes such as filtering for the noise etc. Since the data processed by the convolution algorithms increases day by day, developers must think to optimize the belonging convolution algorithms with respect to the hardware.

In this work, I tried to optimize 2DConvolution and 3DConvolution algorithms which can be found in the PolyBench benchmark suite [1]. These algorithms which were implemented in a way were created on the CUDA environment to execute them on GPUs. To optimize these algorithms, lots of methods are taken into account such as including shared memory usages, putting masking arrays into a constant memory, benefiting from stream version to decrease memory copy latency etc. I will give more detail for each optimization approach in the Proposed Solution section.

The remaining part of the paper includes the following sections. The problem definition section includes the detailed and multi-perspective approaches of the possible problems, 3. Proposed Solutions and corresponding implementation details illuminate the optimization patterns with different configurations techniques, 4. The Experimental Work part gives the experiment results, comparison among them and interpretations. The Future Work section shares possible optimization approaches that will be focused on near future, and I finalize the paper with the conclusion section.

II. Problem Definition

To see the possible bounds of 2DConvolution and 3DConvolution algorithms, I profiled each of them by using the Nsight Compute tool to collect the GPU kernel metrics such as streaming multiprocessor utilization, memory coalescing, bandwidth utilization and Nsight Systems tool to determine other performance bounds that can be caused by data transfers between CPU and GPU. Before examining the profiling details, I want to mention that whatever is problematic for 2DConvolution is probably problematic for 3DConvolution because the 3DConvolution algorithm is obtained sequential launches of a kernel that is very similar to the 2DConvolution kernel.

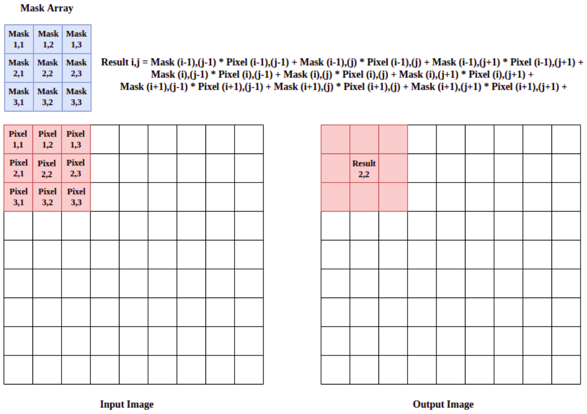

Mask Array
Mask 1,1
Mask 1,2
Mask 1,3
Mask 2,1
Mask 2,2
Mask 2,3
Mask 3,1
Mask 3,2
Mask 3,3
Result i,j = Mask (i-1),(j-1) * Pixel (i-1),(j-1) + Mask (i-1),(j) * Pixel (i-1),(j) + Mask (i-1),(j+1) * Pixel (i-1),(j+1) +
Mask (i),(j-1) * Pixel (i),(j-1) + Mask (i),(j) * Pixel (i),(j) + Mask (i),(j+1) * Pixel (i),(j+1) +
Mask (i+1),(j-1) * Pixel (i+1),(j-1) + Mask (i+1),(j) * Pixel (i+1),(j) + Mask (i+1),(j+1) * Pixel (i+1),(j+1) +
Pixel 1,1
Pixel 1,2
Pixel 1,3
Pixel 2,1
Pixel 2,2
Pixel 2,3
Pixel 3,1
Pixel 3,2
Pixel 3,3
Result 2,2
Input Image
Output Image

Fig. 1: 2DConvolution on an Image

Figure 1 represents the theoretical approach for the 2DConvolution for an arbitrary image. For filtering each image pixels, target and surrounding pixels 3x3 2D-array in our case are multiplied with the mask array which is also 3x3 constant 2D-array, and the result is obtained with summation of multiplication results. During these convolution operations on each pixel, there is a huge amount of repeated memory access for the surrounding pixels. Since the naive version does not include any optimization to load each of those repeated data accesses in a more effective way, these repeated accesses for the same data requires lots of global memory access which is the main limiting factor for the performance of the 2DConvolution algorithm. For example, I have Nvidia's GeForce GTX 1650 GPU on my computer which has 128 GB/s [2] and naively implemented 2DConvolution algorithm result in 90.41 GB/s [3] data transfer rates between SMs and global memory for the case where each TB (thread block) includes 1024 thread and problem size is 8192*8192 bytes.

Another problem that can be easily noticed is that the naive version stores 3x3 constant memory in the register field. By depending on the belonging GPU architecture, can create problems especially if the user tries to launch its kernel with more threads. Each GPU has a limited amount of registers in its SM units and these registers can be used by multiple registers. Because of the limited amount of registers for each thread, if each thread uses some of its available register resources to store mask array, insufficiency of registers for threads can limit the performance at the kernel launch time. That is, because of the insufficiency of register resources multi-threaded performance of each SM will decrease. As a result, this affects performance dramatically especially if belonging GPU has a limited amount of register resources to store mask arrays in the registers of each thread. In addition to kernel optimization, memory transfers between the GPU device and the host device is another performance boundary factor. Figure 2 which is obtained from Nsight System shows elapsed time during data transfers from both host to device and device to host and kernel execution. As one can confirm, most of the elapsed time is elapsed for data transfers between host and GPU device.

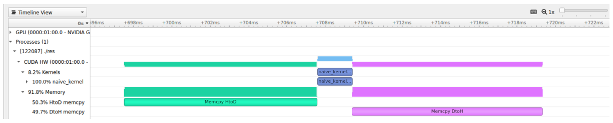


Fig. 2: Nsight System profiling for Naive version of 2DConvolution algorithm

These problems are valid for the 3DConvolution algorithm. There 2 main differences between these convolution algorithms such that,

1) 3DConvolution is an algorithm obtained by the sequential launch of the 2DConvolution algorithm for the 3D data. However, each 2D portion of corresponding 3D data should be small such that 2DConvolution works with bigger 2D arrays such as 4096*4096 while the 3DConvolution algorithm operates with 256*256*256 type 3D arrays.
2) Calculation of convolution is a bit different for the 3DConvolution algorithm compared to the 2DConvolution algorithm. 3DConvolution algorithm implemented in PolyBench benchmark calculates one target by examining 15 different elements from different dimensions of 3D data while 2DConvolution algorithm operates with 9 elements of 2D data.

There may be some other bound points such as non-coalesced memory accesses. Moreover, some other problems can arise if a developer tries to optimize the naive version. Assume that, a developer optimize these convolution algorithms by benefiting from the shared memory. To fulfil the boundaries of the shared memory is implemented with carefully structured if statements and this can create some branch divergence. We can enlarge the possible problems for the 2D-3DConvolution algorithms. However, we focused on the mentioned 3 main problems in this paper.

III. Proposed Solution

To solve the problems mentioned in the Problem Definition section, I try to implement different optimization techniques to see the effect of each of them and try to join them at the final stage. Each of these optimizations is developed for the 2DConvolution algorithm and enlarged to the 3DConvolution algorithm.

A. *Constant Memory Usage*

Constant memory is highly beneficial for the algorithms executed on GPU where each thread of a kernel read the same element nearly the same time during the execution process. In our convolution algorithms, each of the threads needs to read elements of the mask array. However, the naive version assigns a copy of the mask array to each thread which consumes the register resource of belonging GPU. Instead of storing the mask array in the registers for the convolution algorithms, we can hold this mask array in our constant memory. Since convolution kernels are not too complicated such that they do not include lots of atomic or branch operations, accesses to the mask array which is held in constant memory occur nearly the same time. In this way, we can prevent inefficient use of registers and performance does not decrease because these convolution algorithms execute to benefit from constant memories in the most effective way.

B. *Shared Memory Usage*

Shared memory is another utility of the GPUs used to decrease DRAM accesses. In addition, loading from or storing to shared memory is dramatically faster compared to the GPU global memory. In my work, each thread accesses global memory nine times for 2DConvolution and fifteen times for 3DConvolution algorithms. Apart from the boundaries of the 2D and 3D data, each element of those matrices are accessed multiple times which decrease the performance of execution because of global memory access latency, increased traffic on bandwidth. To decrease this, I manipulated the naive version such that the data used for each thread block is moved from global memory to shared memory. For example, a 2D thread block including the 32*32 threads moves the 34*34 data portion of main data from global memory to shared memory. The reason why such a thread block moves the 34*34 data instead of the 32*32 is that we need to include the surroundings of our data bounds. Then, convolution operations are carried out by using the data stored in shared memory. As a result of shared memory usage, one can expect the following:

- Shared memory usage decreases data transfer on the global memory bandwidth.
- Utilization on SMs increase since latency resultant from global memory accesses decrease.
- Performance of both 2D/3DConvolution increases.

C. Benefiting from The Streams

As mentioned in the Problem Definition section, most of the elapsed time is consumed during the data transfer between host and device memories. Instead of waiting for the completion of all data elements to launch a kernel, we can utilize stream implementation [4] which decreases the effect of memory-copy operations and increases overall performance.

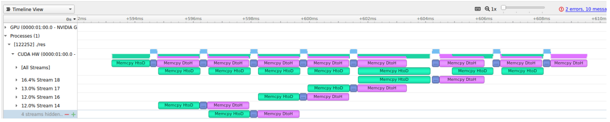


Fig. 3: Nsight System profiling for Naive version of 2DConvolution algorithm

With this implementation, one can hide the latency of memory-copy operations. Figure 2 shows the result for naive memory copy operations from host to device (HtoD) and device to host (DtoH). Figure 3 shows the profiling result on a stream version implementation with 8 streams. By comparing these two figures, we can verify that most of the HtoD latency overlaps with the DtoH which hide the effect of repeated DtoH latency. As a result, we can increase the overall performance by implementing 2D/3DConvolution algorithms working with partitioned data and streams.

IV. Experimental Work

A. Experiment Setup

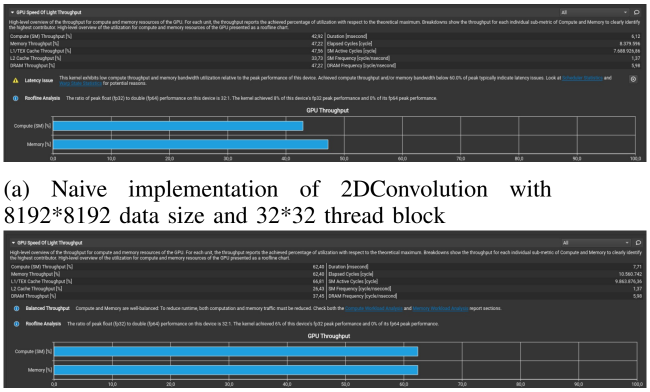
In this work, I completed the experiments on two different environments:

1) Nvidia's GeForce GTX 1650 and CUDA 11.4. Also, I profiled each experiment on GeForceGTX 1650 GPU.
2) NVIDIA's Tesla K80 GPU provided by Google Colab [5] and CUDA 9.1.

For the 2DConvolution algorithm, I completed 7 different experiments to observe the effect of streams, constant memory and shared memory usages. The problem size for those experiments varies as 4096*4096, 8192*8192 and 16384*16384. The number of streams used for these experiments varies as 8, 16 and 32, and I created square thread blocks as 8*8, 16*16 and 32*32. For the 3DConvolution algorithm, I completed 6 different experiments where stream sizes are 4, 8 and 16; problem sizes are 128*128*128, 256*256*256 and 512*512*512; and thread blocks are 16*16 and 32*32. The data and constant mask arrays are created randomly in the naive versions.

B. Experiment Results

In this work, I tried to optimize 2D/3DConvolution algorithms by utilizing shared memory, constant memory and stream implementation. To see the effect of constant memory and shared memory usage, I profiled these algorithms with Nsight Compute. Profiling on Nsight Compute shows a detailed hardware usage of the corresponding kernel.


GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	42,92	Duration [msecond]	6,12
Memory Throughput [%]	47,22	Elapsed Cycles [cycle]	8.379.596
L1/TEX Cache Throughput [%]	47,56	SM Active Cycles [cycle]	7.688.926,86
L2 Cache Throughput [%]	33,73	SM Frequency [cycle/nsecond]	1,37
DRAM Throughput [%]	47,22	DRAM Frequency [cycle/nsecond]	5,98

Latency Issue: This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

Roofline Analysis: The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 8% of this device's fp32 peak performance and 0% of its fp64 peak performance.

(a) Naive implementation of 2DConvolution with 8192*8192 data size and 32*32 thread block

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

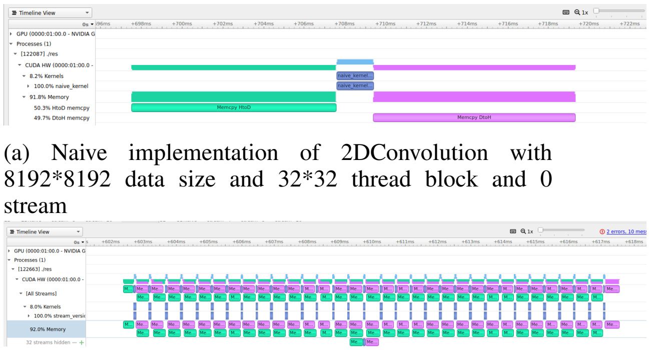
Compute (SM) Throughput [%]	62,40	Duration [msecond]	7,71
Memory Throughput [%]	62,40	Elapsed Cycles [cycle]	10.560.742
L1/TEX Cache Throughput [%]	66,81	SM Active Cycles [cycle]	9.863.876,36
L2 Cache Throughput [%]	26,43	SM Frequency [cycle/nsecond]	1,37
DRAM Throughput [%]	37,45	DRAM Frequency [cycle/nsecond]	5,98

Balanced Throughput: Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis report sections.

Roofline Analysis: The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 6% of this device's fp32 peak performance and 0% of its fp64 peak performance.

(b) Shared memory included implementation of 2DConvolution with 8192*8192 data size and 32*32 thread block

Fig. 4: Profiling results of 2DConvolution with Nsight Compute Tool



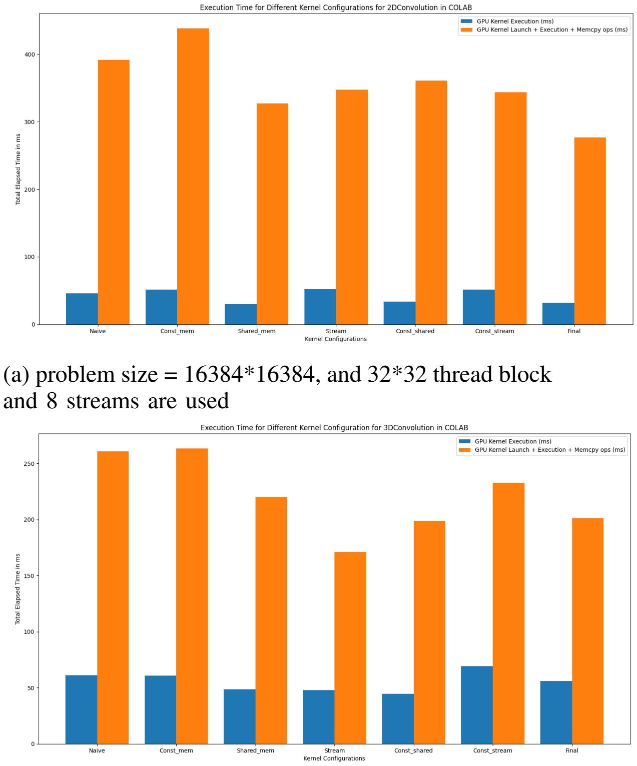
(a) Naive implementation of 2DConvolution with 8192*8192 data size and 32*32 thread block and 0 stream

(b) Stream included implementation of 2DConvolution with 8192*8192 data size and 32*32 thread block and 32 streams

Fig. 5: Profiling results of 2DConvolution with Nsight Systems Tool

Figure 4a and 4b shows the effect of shared memory usage. By optimizing the naive version of the 2DConvolution algorithm using shared memory, I could increase SM utilization and memory throughput. The reason why memory throughput increases are that since global memory bandwidth is not wasted with the repeated data, the remaining capacity is used effectively-remaining part of the kernel such as storing convolution result to the memory. Overall performance increases by 1.4 times for SM utilization and 1.28 times for the memory throughput.

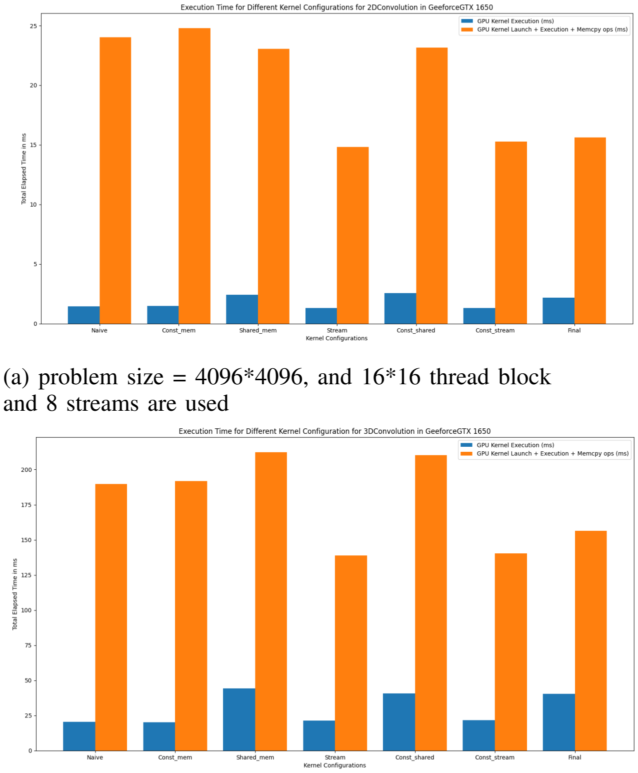
To see the effect of the streams to the naive convolution algorithms, I profiled the results with Nsight System profiler and results are shared in Figure 5a and 5b for Naive version and a version which uses streams respectively. This profiler captures memory-copy operations between host and device. Completion of the execution including memory copy operations between host and GPU device is around 15ms with 32 streams while the same process occurs in 23ms for the naive version. Overall performance of the 2DConvolution algorithm



(a) problem size = 16384*16384, and 32*32 thread block and 8 streams are used

(b) problem size = 512*512*512, and 16*16 thread block and 8 streams are used

Fig. 6: Performance results on NVIDIA's Tesla K80 GPU for different configurations of 2D/3DConvolution algorithms



(a) problem size = 4096*4096, and 16*16 thread block and 8 streams are used

(b) problem size = 512*512*512, and 32*32 thread block and 8 streams are used

Fig. 7: Performance results on NVIDIA's GeForce GTX 1650 GPU for different configurations of 2D/3DConvolution algorithms

increases 1.5 times with the help of the streamed version.

For different configurations including different optimization techniques, I completed lots 7 experiments for 2DConvolution and 6 experiments for 3DConvolution. The experiments record both GPU kernel execution time, summation of GPU kernel execution time and memory copy operations.

I can observe enhancement for both 2D/3DConvolution algorithms with the help of the shared memory usage and streams as shown from the figure 6a and 6b respectively. By using shared memory, total execution time decreases from 391.313ms to 327.321ms and GPU kernel execution time decreases from 46.0393ms to 29.9667ms. For the addition of the streams to the naive version, total execution time decreases from 391.313ms to 347.486ms and GPU kernel execution time increases from 46.0393ms to 51.9297ms. The increase in GPU kernel execution time is because of the overheads of multiple kernel launches and streams. Even if GPU kernel execution time gets worse, overall performance increases 1.15 times. The addition of constant memory usage affects overall performance in a negative way because access to constant memory is more expensive compared to access to registers. Hence GPU kernel and overall performance get worse with the addition of the constant memory usage to the naive version. For the final configuration which includes constant memory, shared memory and streams overall performance decreases from 391.313ms to 276.871ms and GPU kernel execution time decreases 46.0393ms to 32.1359ms. The final implementation provides 1.42 times faster execution capability.

In my local device which is GeForce GTX 1650 GPU, shared memory usage does not increase the GPU kernel performance. The reason why this is so is explained that GeForce GTX 1650 GPU does not have a shared memory space as mentioned in [6] even if Nsight Compute profiler generates shared memory results. I do not know how Nsight Compute tool collects those metrics but these metrics are not reliable for shared memory usage. I can observe the positive effect of shared memory usage on Colab's Tesla K80 GPU while I cannot see a similar performance enhancement on my GeForceGTX 1650 GPU device. Adding constant memory for the mask array does not decrease performance for GeForceGTX 1650 as in Tesla K80. By using constant memory for the mask array, GPU kernel performance increases from 2.6772ms to 2.8291ms and from 18.5337ms to 24.7836ms for 2D and 3DConvolution algorithms respectively. The decrease in performance by adding constant memory to naive implementation is not as big as in Tesla K80. Furthermore, the streams decrease GPU performance from 18.5337ms to 19.1869ms and increase overall performance from 184.588ms to 138.99ms for 3DConvolution as shown in

Figure 7b. Similarly, GPU kernel performance is around the same and the overall performance increases from 24.0375ms to 14.814ms. The final implementation does not increase performance as I expected because of the misconception on shared memory for GeForce GTX 1650 GPU. Finally, overall performance increases from 24.0375ms to 15.616ms and GPU kernel performance is decreases from 1.4427ms to 2.1913ms due to the stream overheads and shared memory problem. For the remaining profiling results and experiments, one can reach out to them from the GitHub link [1].

V. Conclusions

To conclude, including shared memory in the naive versions of both 2D and 3DConvolution algorithms result in an enhancement in performance. However, the quantity of the performance enhancement depends on the belonging GPU architecture. Similarly, holding mask array in the constant memory instead of registers does not decrease performance and if you have big mask arrays such as 7*7 which requires 49 registers, holding mask array in the constant memory will absolutely increase the GPU kernel performance because it reduces the inefficient use of registers. Furthermore, adding streams to the naive versions of these algorithms increases performance by reducing the elapsed time for memory copy operations.

VI. Future Work

For future work, the problems that arise after the implementation of the mentioned optimizations can be examined. For example, the decision of the most efficient stream amount depends on the data and belonging architecture in terms of peripheral between host and GPU device. Similarly, the convolution can operate on bigger arrays such as 7*7 or 8*8 instead of 3*3. This will increase the tile size of shared memory. For such a scenario, the user needs to re-implement shared memory usage with respect to the Convolution algorithm features and GPU shared memory resource. This also depends on the GPU architecture resources and the data which will be processed.

References

[1] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. Proceedings of Innovative Parallel Computing (InPar '12), 2012

[2] G. Database and G. Specs, ”NVIDIA GeForce GTX 1650 Specs”, TechPowerUp, 2022. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gtx-1650.c3366. [Accessed: 02- Jan- 2022]

[3] B. Topçu, ”ceng443_FinalProject/2DConvolution-Naive-8192-32.pdf at main · topcuburak/ceng443_FinalProject”, GitHub, 2022. [Online]. Available: https://github.com/topcuburak/ceng443_FinalProject/blob/main/Final_Versions/2DConvolution/2DConv-NsightCompute/2DConvolution-Naive-8192-32.pdf. [Accessed: 02- Jan- 2022]

[4] ”Streams and Concurrency”, Developer.download.nvidia.com, 2022. [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. [Accessed: 02- Jan- 2022]

[5] ”Google Colab”, Research.google.com, 2022. [Online]. Available: https://research.google.com/colaboratory/faq.html. [Accessed: 02- Jan- 2022]

[6] ”NVIDIA GeForce GTX 1650 (Laptop) GPU”, Notebookcheck, 2022. [Online]. Available: https://www.notebookcheck.net/NVIDIA-GeForce-GTX-1650-Laptop-GPU.416044.0.html. [Accessed: 02- Jan- 2022]

[1] https://github.com/topcuburak/ceng443_FinalProject