PC LAB 4

CONVOLUTION

NAME	SEC	BN
Abdalhameed emad	1	34
Mahmoud Abdalhameed	2	22

Changing image Size and Batch Size

- **Using Shared Memory Mask**

1)Image : width = **2482**, height = **1718**, comp = 3
Kernel size = **3** **,**Batch = B

	TIME B=**1**	**TIME** B=**2**	**TIME** B=**3**
K1 (no tile)	**958.13us**	**1.9089ms**	**2.8589ms**
K2 (input tile)	**741.27us**	**1.7399ms**	**2.6105ms**
K3 (output tile)	**872.60us**	**1.6010ms**	**2.3977ms**
Python	**61.656 ms**	**128.2958ms**	**149.22380 ms**

2)Image : width = **1980**, height = **1289**, comp = 3
Kernel size = 3 , Batch = B , **(K=3)**

	TIME (B=1)	**TIME (B=2)**	**TIME (B=3)**
K1 (no tile)	**572.47us**	**1.1372ms**	**1.7019ms**
K2 (input tile)	**515.07us**	**1.0242ms**	**1.5304ms**
K3 (output tile)	**468.92us**	**930.84us**	**1.3887ms**
Python	**25.94558 ms**	**60.10794 ms**	**89.3836ms**

3)Image : width = **5000**, height = **5000**, comp = 3
Kernel size = 3 ,Batch = b, **(K=9)**

	TIME (B=1)	**TIME (B=2)**	**TIME (B=3)**
K1 (no tile)	**49.203ms**	**98.389ms**	**133.73ms**
K2 (input tile)	**36.768ms**	**72.873ms**	**108.98ms**
K3 (output tile)	**27.585ms**	**55.149ms**	**82.717ms**
Python	**357.4313ms**	**788.333ms**	**1667.3336ms**

Changing kernel Size
Shared Memory Mask

Image : width = 1980, height = 1289, comp = 3
Kernel size = K , Batch = 3

	Kernel TIME (K=5)	**Kernel TIME (K=7)**	**B=3 (K=9)**
K1 (no tile)	4.6875ms	**9.1304ms**	15.084ms
K2 (input tile)	3.0413ms	5.1118ms	11.272ms
K3 (output tile)	**2.8885ms**	5.1648ms	8.4618ms
Python	**85.332 ms**	**97.6677 ms**	**117.990ms**

-Using Constant Memory Mask

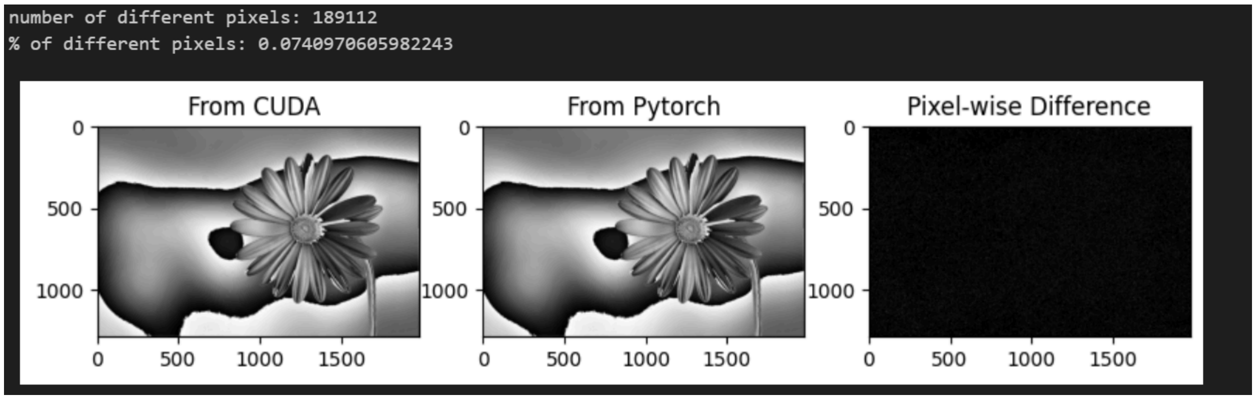
Image :width = **1980,** height = **1289,** comp = 3 , B=3

	TIME k=5	**TIME (K=7)**	**TIME (K=9)**
K1 (no tile)	**4.6944ms**	**9.1363ms**	**15.115ms**
K2 (input tile)	**2.5046ms**	**4.9290ms**	**9.8807ms**
K3 (output tile)	**2.6136ms**	**5.0517ms**	**8.3390ms**
Python	**85.332 ms**	**97.6677 ms**	**117.990ms**

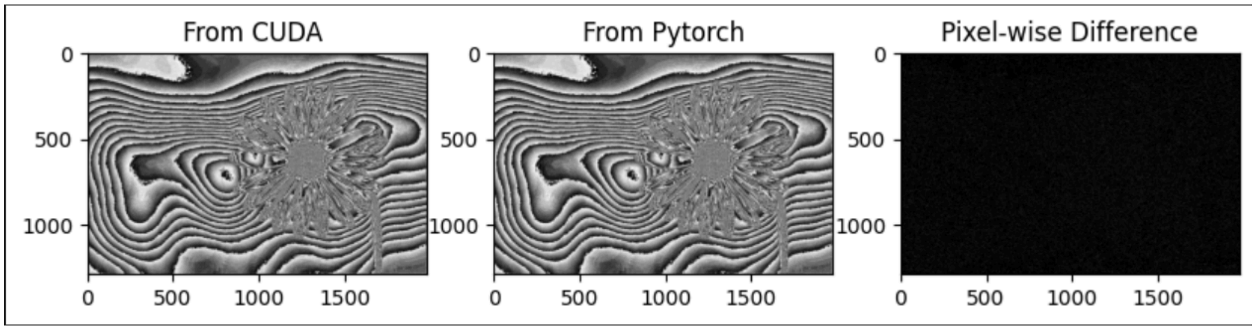
Image :width = **5000**, height = **5000**, comp = 3, B=3

	TIME (K=5)	**TIME (K=7)**	**TIME (k=9)**
K1 (no tile)	**45.876ms**	**89.346ms**	**147.87ms**
K2 (input tile)	**24.507ms**	**45.245ms**	**95.407ms**
K3 (output tile)	**25.573ms**	**49.344ms**	**81.441ms**
Python	**921.9997 ms**	**1277.66 ms**	**1647.665ms**

RESULTS



```
number of different pixels: 189112
% of different pixels: 0.0740970605982243
```



Comments:

Changing Image Size and Batch Size:

For smaller image sizes (2482x1718 and 1980x1289), the execution times for the CUDA kernels (K1, K2, and K3) are in the range of microseconds to a few milliseconds, even with batch sizes up to 3.

As the image size increases (5000x5000), the execution times for the CUDA kernels increase significantly, reaching up to tens of milliseconds for batch size 3.

Generally, the execution time increases with larger image sizes and batch sizes, as expected due to the increased computational workload.

The Python code using PyTorch has significantly higher execution times (in the range of tens to hundreds of milliseconds) compared to the CUDA kernels, likely due to the overhead of Python and the PyTorch framework.

Changing Kernel Size:

For both shared memory and constant memory scenarios, the execution time increases as the kernel size (K) increases from 5 to 7 to 9.

The K3 kernel (output tiling) generally performs better than K2 (input tiling) and K1 (no tiling) for most cases, suggesting that output tiling is more efficient for the given workloads.

The performance gap between the CUDA kernels and the Python code widens as the kernel size increases, with the CUDA kernels being significantly faster.

Shared Memory vs. Constant Memory:

For small kernel sizes (5 and 7), the shared memory implementation (K3) outperforms the constant memory implementation for both small and large image sizes.

For larger kernel sizes (9), the constant memory implementation (K3) performs better than the shared memory implementation for the larger image size (5000x5000), potentially due to the increased shared memory requirements.

Overall, the results demonstrate the performance benefits of using CUDA kernels with tiling optimizations (K2 and K3) compared to the basic convolution implementation (K1) and the Python code. The choice of tiling strategy (input or output) and memory type (shared or constant) can impact performance, and the optimal choice depends on factors such as image size, kernel size, and available memory resources.

CONCLUSION:

K1 (No Tiling):

- K1 is the basic convolution implementation without any tiling optimizations.
- It generally has the slowest execution times among the three kernels, especially for larger image sizes and kernel sizes.

- The lack of tiling leads to poor memory access patterns and increased global memory traffic, resulting in lower performance.

K2 (Input Tiling):

- K2 implements input tiling, where the input data is divided into tiles that fit into the shared memory or registers of the GPU.
- This approach reduces global memory accesses for the input data, leading to better performance compared to K1.
- However, it still suffers from inefficient memory access patterns for the output data, which can limit its performance gains.

K3 (Output Tiling):

- K3 implements output tiling, where the output data is divided into tiles that fit into the shared memory or registers of the GPU.
- This approach ensures coalesced memory access patterns for both input and output data, resulting in the best performance among the three kernels in most cases.
- By reducing global memory traffic and optimizing memory access patterns, K3 can achieve significant performance improvements, especially for larger image sizes and kernel sizes.

Based on the results, K3 (output tiling) generally outperforms K2 (input tiling), which in turn outperforms K1 (no tiling). The performance gap between K3 and the other kernels becomes more pronounced as the image size and kernel size increase, indicating the importance of optimized memory access patterns for efficient convolution operations on GPUs.