# Implementation of a CNN on a GPU to Optimize Performance

Aditya Chavan

## Introduction:

This assignment includes implementing a CNN on the Nvidia Pascal GPU. The CNNs take a input image as a text file and perform OCR on it. The recognized characters are returned as the output. The goal of this project is to reduce execution time as much as possible. We focus on two ways to optimize performance in this assignment: use of shared and constant memory to reduce memory bandwidth required and allocating kernels in a way that uses the hardware as efficiently as possible.

To keep memory accesses to the CPU ram to a minimum, we use memory within the GPU, namely the constant and the shared memory. This memory is fast but limited and cannot fit the larger matrices we use in the process (matrices of size 6272 x 512). The convolution process needs the weight matrices at every step and hence these are put in the shared memory. This keeps accesses to the main memory as little as possible.

For the kernels, the GPU uses a SIMD architecture, namely Single Snstruction, Multiple Data. Hence each kernel has code that thousands of threads use with only the data being different. We keep these statements as efficient as possible avoiding using sync statements and data corruption. Instead of writing one kernel to handle everything, splitting the code in 4 kernels allowed only the necessary amount of resources being allocated to each kernel, which allowed for efficient use of the GPU cluster. The computationally intensive kernels which multiplied large matrices were allocated higher resources than the ones working on smaller data. This was found to also increase execution speed.

## CNN Used:

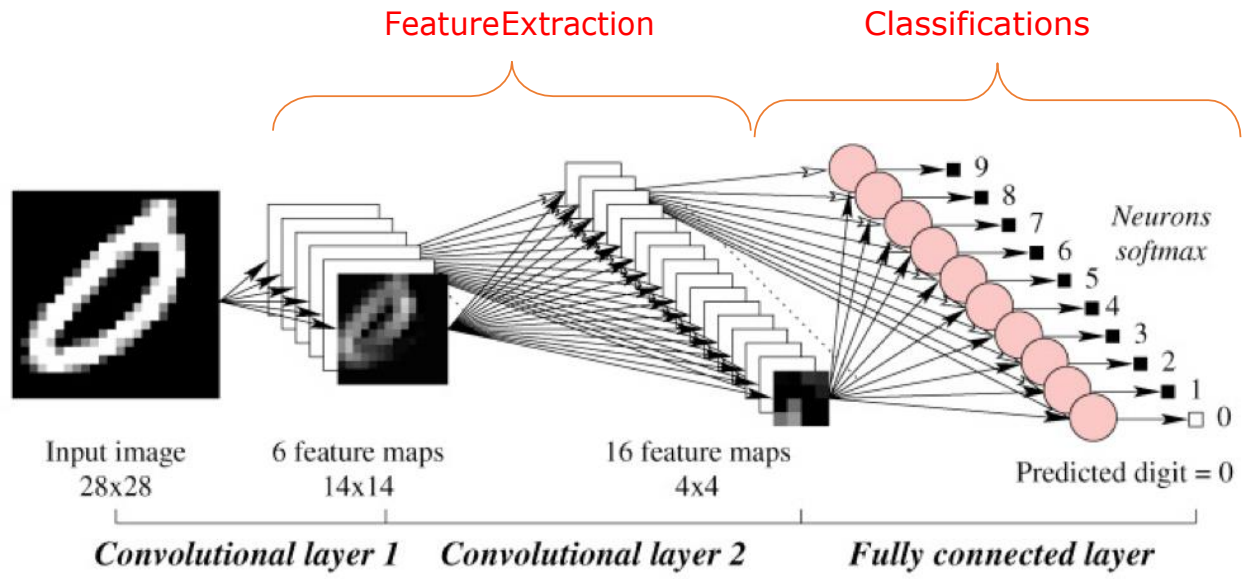



Figure from: Salomon, M., et al. "Steganalysis via a convolutional neural network using large convolution filters for embedding process with same stego key: A deep learning approach for telemedicine." *European Research in Telemedicine/La Recherche Européenne en Télémédecine* 6.2 (2017): 79-92

## Implementation:

We create a matrix for the image and the filters. The weights are also provided by a matrix. As GPUs are very optimized to perform parallel calculations, matrix multiplication is extremely efficient to perform. We create a separate kernel to handle each multiplication, to reduce execution time further.

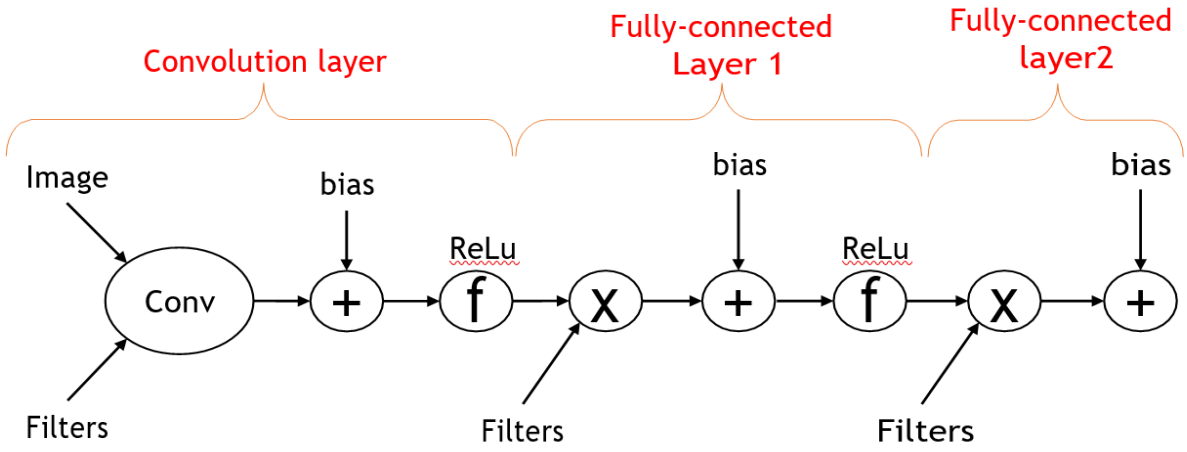
## Calculations:

- $A_{28\times28\times8} = Input_{28\times28\times1} * W^{(1)}_{10\times10\times1\times8}$ (Note: "*" is convolution as in equation 1)
- $B_{28\times28\times8} = A_{28\times28\times8} + b_{1\times8}{}^{(1)}$ (Note: the scalar bias for each filter is added to all elements of the matrix)
- $C_{28\times28\times8} = ReLu(B_{28\times28\times8})$
- $D_{1\times6272} = Reshape(C_{28\times28\times8})$
- $I_{1\times512} = D_{1\times6272}\times W^{(2)}_{6272\times512}$
- $J_{1\times512} = I_{1\times512} + b_{1\times512}{}^{(2)}$
- $K_{1\times512} = ReLu(J_{1\times512})$
- $L_{1\times10} = K_{1\times512}\times W^{(3)}_{512\times10}$
- $Output_{1\times10} = L_{1\times10} + b_{1\times10}{}^{(3)}$

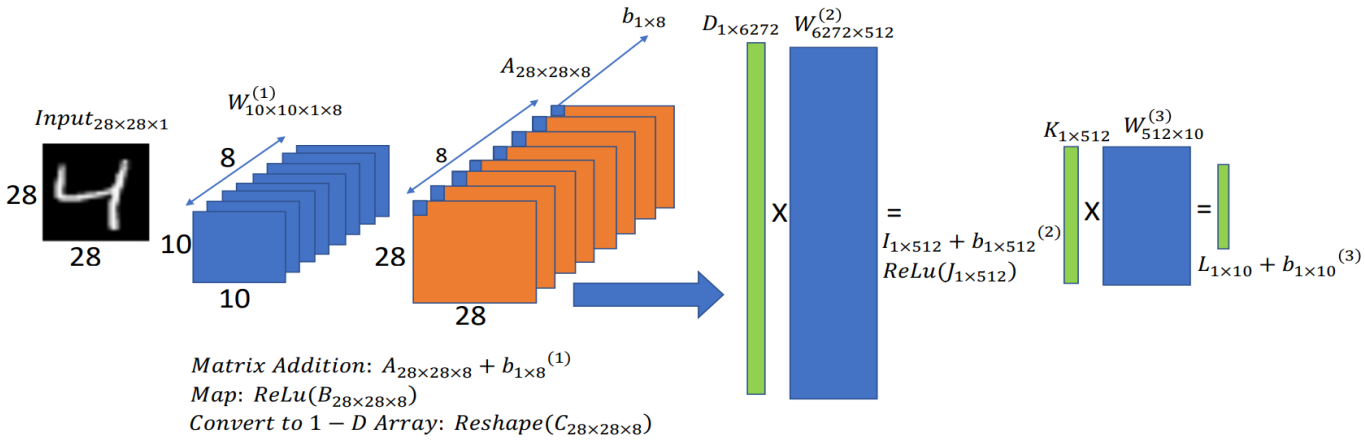
# Layer-by-layer parameters

| Tensor_name | Layer name | Filter size(FxF) | # of channels (C ) | # of filters (K) | Matrix dimensions | File name |
|---|---|---|---|---|---|---|
| Variable | Conv1 | 10x10 | 1 | 8 | $W_{10\times10\times1\times8}^{(1)}$ | Conv_w |
| Variable_1 | Conv1 | | | | $b_{1\times8}{}^{(1)}$ | Conv_b |
| Variable_2 | Fc1 | 1x1 | 6272 | 512 | $W_{6272\times512}^{(2)}$ | Fc1_w |
| Variable_3 | Fc1 | | | | $b_{1\times512}{}^{(2)}$ | Fc1_b |
| Variable_4 | Fc2 | 1x1 | 512 | 10 | $W_{512\times10}^{(3)}$ | Fc2_w |
| Variable_5 | Fc2 | | | | $b_{1\times10}{}^{(3)}$ | Fc2_b |

# Implementation:




# CNN Model:



# Report:

- All data was sent to constant memory except the conv_w and fc1_w matrices, which are too big to fit in constant memory.
- **Kernel 1**: The convolution, reshaping, bias and relu on the padded image were done in a 28*28 *8 kernel.
- **Kernel 2**: 28*28 Block Size and 512*1 grid size was used for multiplying the 2 matrices, D and W. (As specified in the convolution pdf)
- **Kernel 3**: 32*16 Block Size was used for adding bias and relu of 512*1 matrix.
- **Kernel 4**: 10*1 Block Size was used for the final biasing and addition.
- Intermediate data used multiple times, (like for eg. for multiplication) were stored in shared memory.

**Final Execution time:** (on average using nvprof)

| | | |
|---|---|---|
| Kernel 2: | 242.7 | usec |
| Kernel 1: | 213.1 | usec |
| Kernel 3: | 7.7 | usec |
| Kernel 4: | 2.8 | usec |
| Total: | 465.2 | usec |

**Commands to execute:**

**qsub -I -q coc-ice -l nodes=1:gpus=1:teslap100,walltime=01:00:00**

**…….**(Using Nvidia Tesla Pascal)

<navigate to executable>

**nvprof ./mnist images/xaa labels/xaa**

# Conclusion

The kernels were tested to be working correctly and were able to successfully classify the input images. The initial execution time was about 47 ms using the CPU which was reduced to about ~500 usec using the GPU, which represented a significant reduction (~90%) in testing time.