Computer Vision + ML using GPU/CUDA

CuCNN **[GitHub]**

CSN- 291: Computer Architecture and Microprocessor (CAM)

CuCNN

(Cuda Convolutional Neural Network)

- Jitesh Jain

---

1. Problem Statement

Implement a simple ***3-layer CNN***[5][6] (Convolutional Neural Network) for Image Classification on ***MNIST***[1][2] using ***CUDA***[3][4] ***Programming*** and observe the *effect of different kernel settings on the performance of the model*.

2. Novelty

This work presents a study about the *effect of using different Grid Size and Block Size on a CNN* model's performance, which is supported by the experiments.

It also presents forward a *simple 3-layer* CNN architecture that attains a *test accuracy in access of 97%* along with a *training time of only 5* minutes on a **Tesla T4** (available for free on *GoogleColab*[7]) system.

3. Evaluation Parameters

We use the **MNIST**[1][2] dataset for training and testing of our CNN model. **MNIST**[1][2] is an extensive database of handwritten digits

(0-9) containing **60000 images of size *28x28***. Due to its ***wide usage*** for *testing and training of classification models in the field of Machine Learning*, we consider it as a **benchmark** for evaluating our model's performance.

We evaluate our model based on three parameters:

- ***Accuracy:*** The *ratio of the number of correctly classified images to the total number of images*.
- ***Train Error:*** The *total error* calculated during the training as the *sum of the euclidean norm* (implemented in CUDA using *cublasSnmr2* in the *cuBLAS*[8] library) of the vector containing the sum of the differences between the target and predicted probability for each digit of a given image *divided by the number of images*.
- ***Training Time:*** The *time taken to train* the CNN model on a Tesla T4 GPU for 50 epochs.

4. Methodology

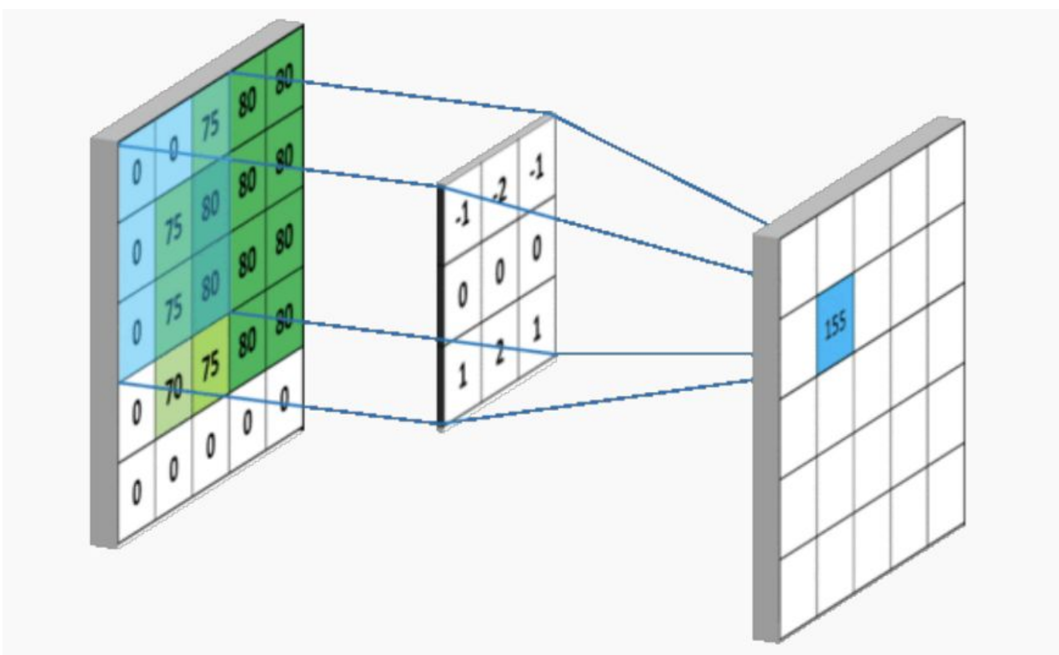
In this section, we first provide an overview of CNN in ***Section 4.1***. We briefly explain our 3-layer CNN architecture in ***Section 4.2***, followed by introducing the terms in CUDA programming in ***Section 4.3***. Finally, in ***Section 4.4***, we present the implementation details.

4.1 Convolutional Neural Network (CNN)

A **CNN**[5][6] is a Neural Network specifically designed for computer vision applications like image classification, semantic segmentation, etc.

The basic building blocks of a CNN are:

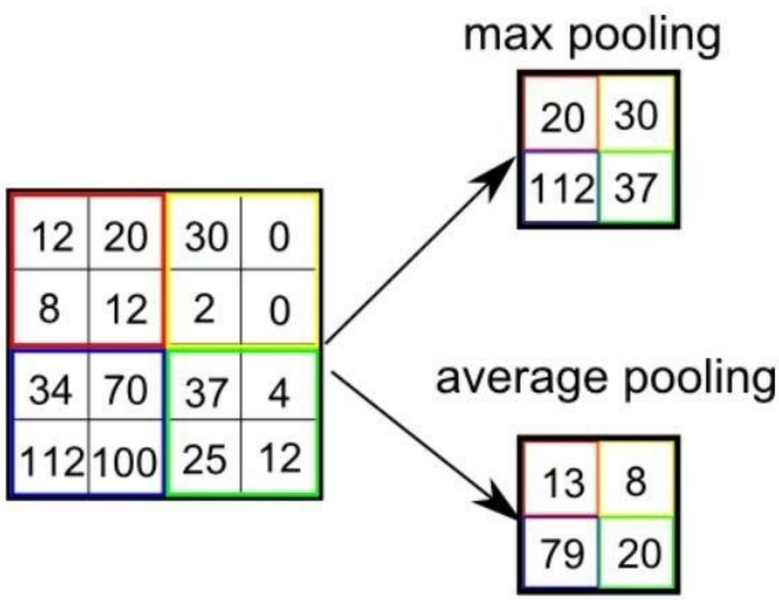
1. **Convolution layer**: It consists of *sliding window* operations where a ***filter*** (also called the ***kernel***, the *middle one in the image below*) *slides* over the ***input feature map*** (*left one in the image below*) and *outputs one value corresponding to* each of the overlaps. It works using *parameter sharing* (as only the kernels have the learnable parameters), thus reducing the parameters by a considerable margin compared to a *linear deep neural network*[9].



Convolution Operation

[Source]

2. **Pooling Layer:** It is used to reduce the spatial size of the feature map output by a convolution layer. There are two types of pooling: Average Pooling and Max Pooling (see the image below). We don't use a pooling layer in our architecture because the input image already has a small size (28 x 28).



Pooling Operation
[Source]

3. **Activation Function:** It is generally used to induce non-linearity in the network to make the model more robust to variations in the data. We use ***sigmoid*** as our activation function.

$$S(x) = 1 / (1 + e^{x})$$

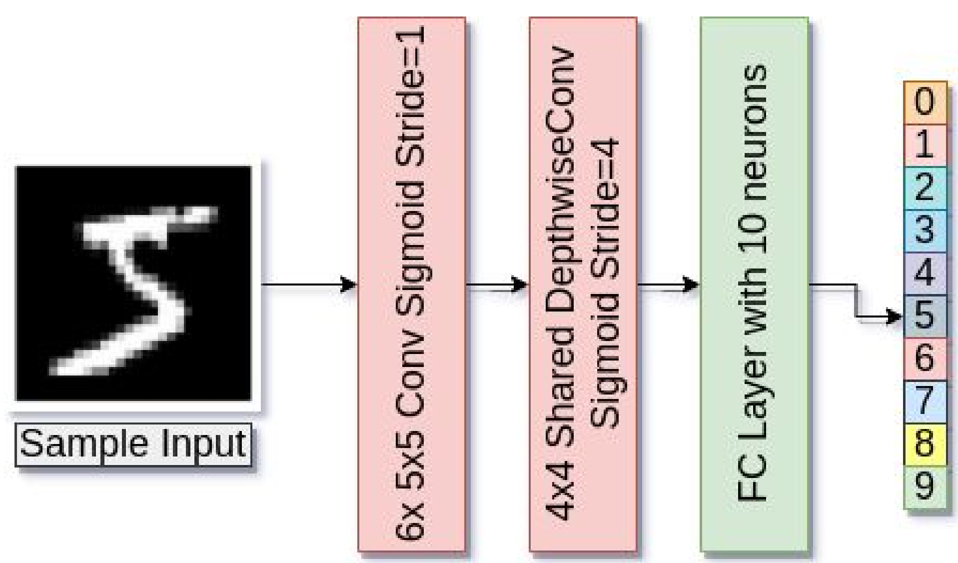
Sigmoid Function

4. **Fully-Connected Layer:** It is a linear layer with multiple nodes at the end stages of the network to predict the probability of each category. We use an FC layer with *10* nodes in our architecture.

4.2 The CNN Architecture

The 3-Layer CNN consists of:

- ***Convolution Layer:*** Applies a **Convolution Operation** with *6* kernels of size *5 x 5* with *stride=1* on the input image (size= *28 x 28*) to output a map of shape=*24 x 24 x 6*.
- ***Shared Depthwise Convolution Layer:*** Applies a *shared Depthwise* (the same kernel applied to different channels of the input from the previous layer) *Convolution Operation* with a *4 x 4* kernel with *stride = 4* on each channel of the previous Convolution layer's output feature map to output a map of shape = *6 x 6 x 6*.
- **Fully Connected Layer:** *Flattens* the output from the previous layer to a layer with *10 nodes,* with each node's value representing the *probability* of a *digit from 0-9*.



3-layer CNN Architecture (made using draw.io)

4.3 CUDA Programming

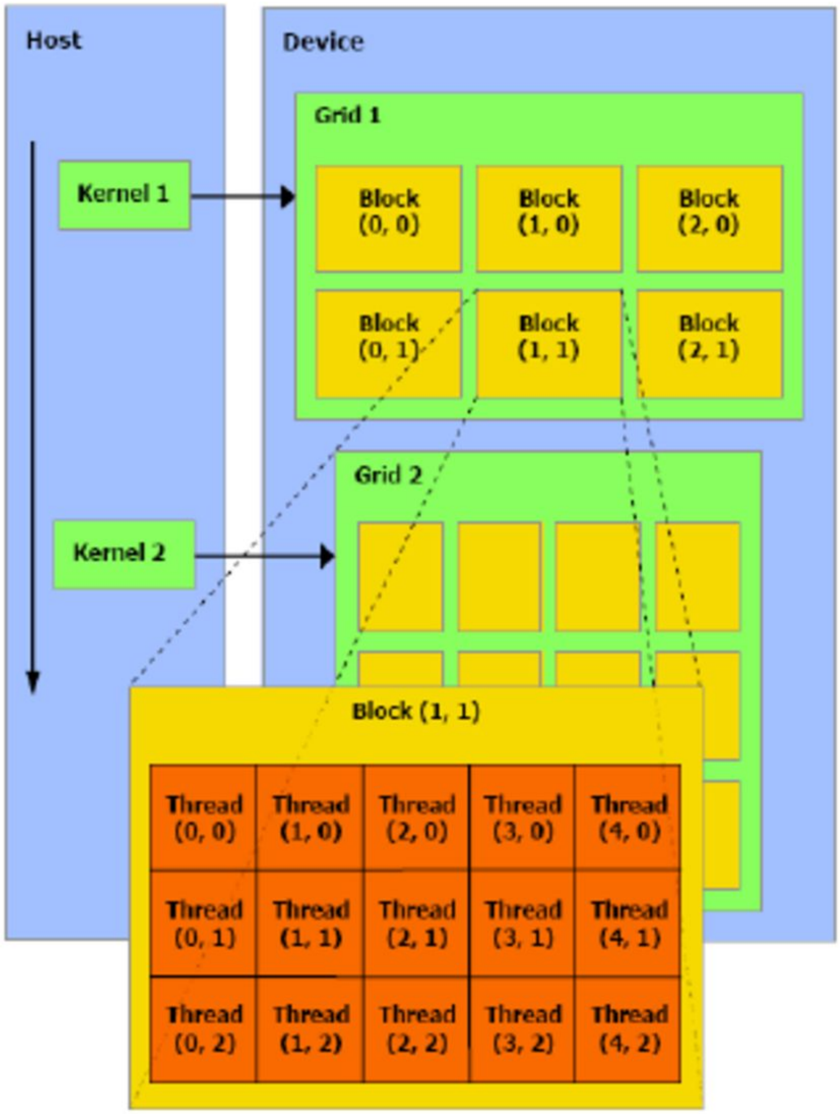
CUDA[3][4] is a parallel computing platform and programming model developed by Nvidia for general computing on its GPUs. CUDA enables developers to speed up compute-intensive applications by harnessing GPUs' power for the computation's parallelizable part.

Some technical terms:

- **Host:** The *CPU* that makes calls to the kernels.
- **Device:** The *GPU*.
- **__global__ Function:** Called by the *host* and executed on the *device*.
- **__device__ function:** Called by the *device* and executed on the *device*.
- **Block Size:** Number of threads inside a block.
- **Grid Size:** Number of blocks inside a grid.

4.4 Implementation Details

We implemented specific feed-forward *and backpropagation* functions for our CNN model along with activations and error functions, all using the **CUDA** framework. Some important details are:



High-level Overview of CUDA

[Source]

- We used simple ***float multi-dimensional arrays*** to implement the *parameter structures* (weights and biases) and *outputs* for the CNN.

- The functions called during feedforward propagation and backpropagation were specified as **__global__** as they are called during training from the **host (CPU).**
- We train our model for ***50 epochs (or iterations***, in different settings) and 100 epochs (or iterations, in one setting).
- The **best performing** *kernel sizes* are: **Grid Size = 64**, **Block Size = 64**, i.e, ***kernel <<<64,64>>>.***

5. Results/Experiments

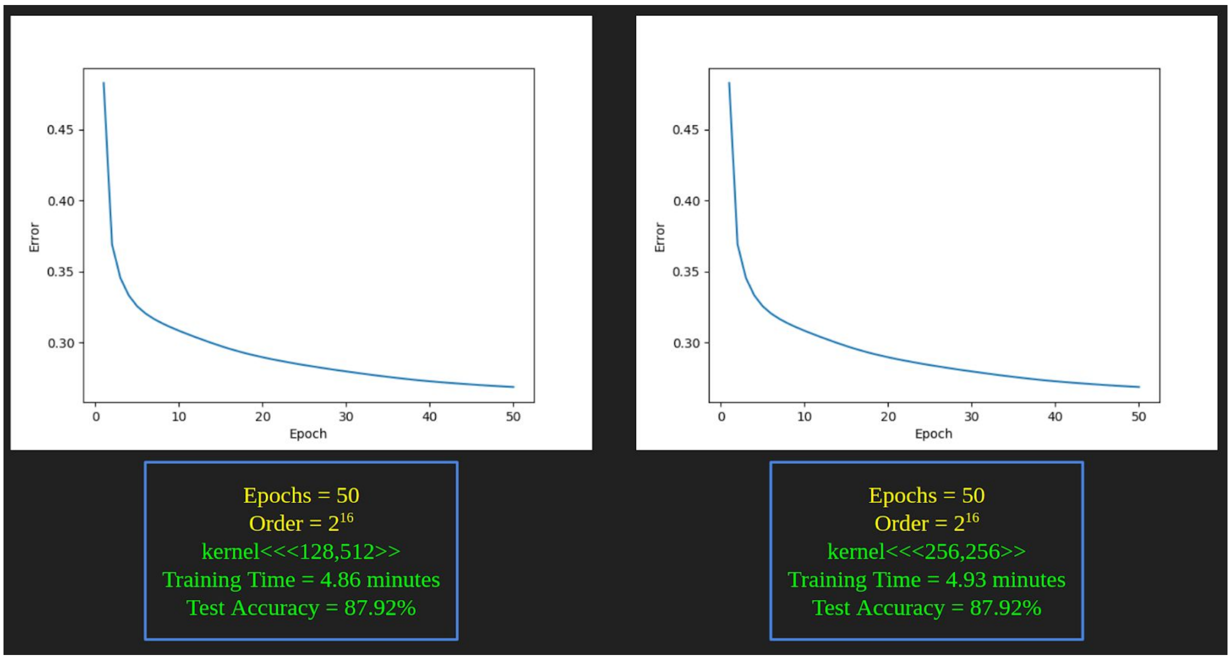
We perform experiments on *different Block Sizes* and *Grid Sizes* settings for the kernel and make *useful observations*. All experiments were performed on a **Tesla T4 GPU.**

Grid Size	Block Size	Order (2)	Epochs	Test Accuracy	Training Time (Minutes)
64	**64**	**12**	**50**	**97.12**	**4.54**
64	64	12	100	**97.41**	9.10
128	512	16	50	87.92	4.86
256	256	16	50	87.92	4.93
16	16	8	50	97.12	10.02

Results

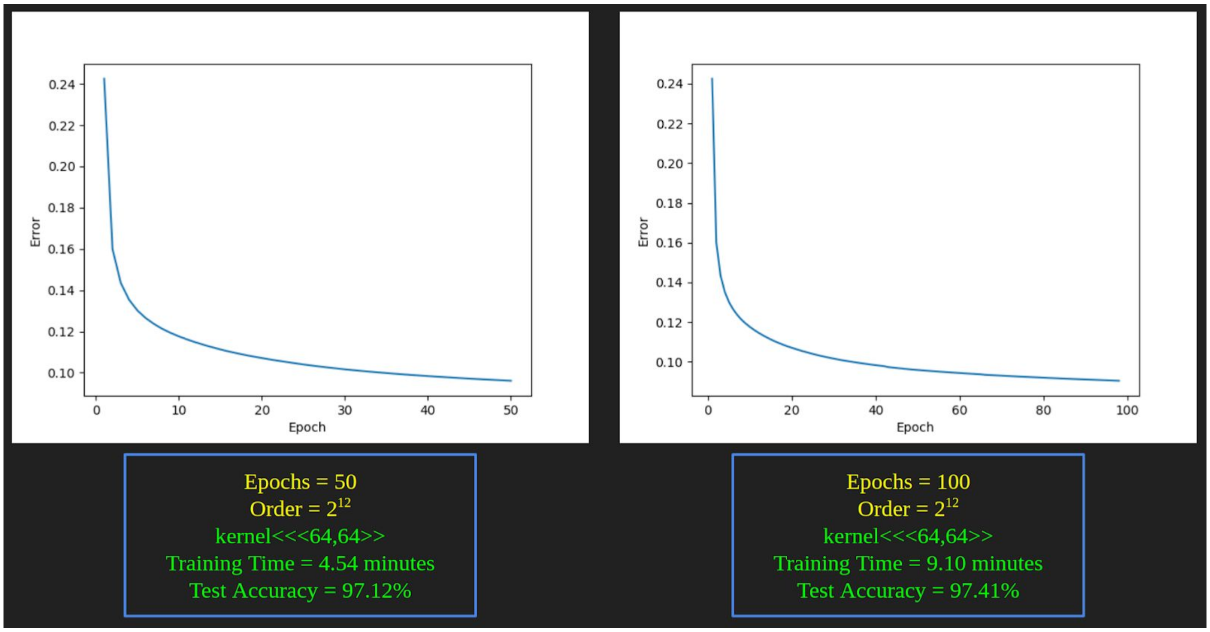
Observations

- The model’s performance depends on the ***Grid Size*** (number of Blocks) and ***Block Size*** (number of Threads).
- Products of the same order give almost the same results (in terms of time and accuracy).



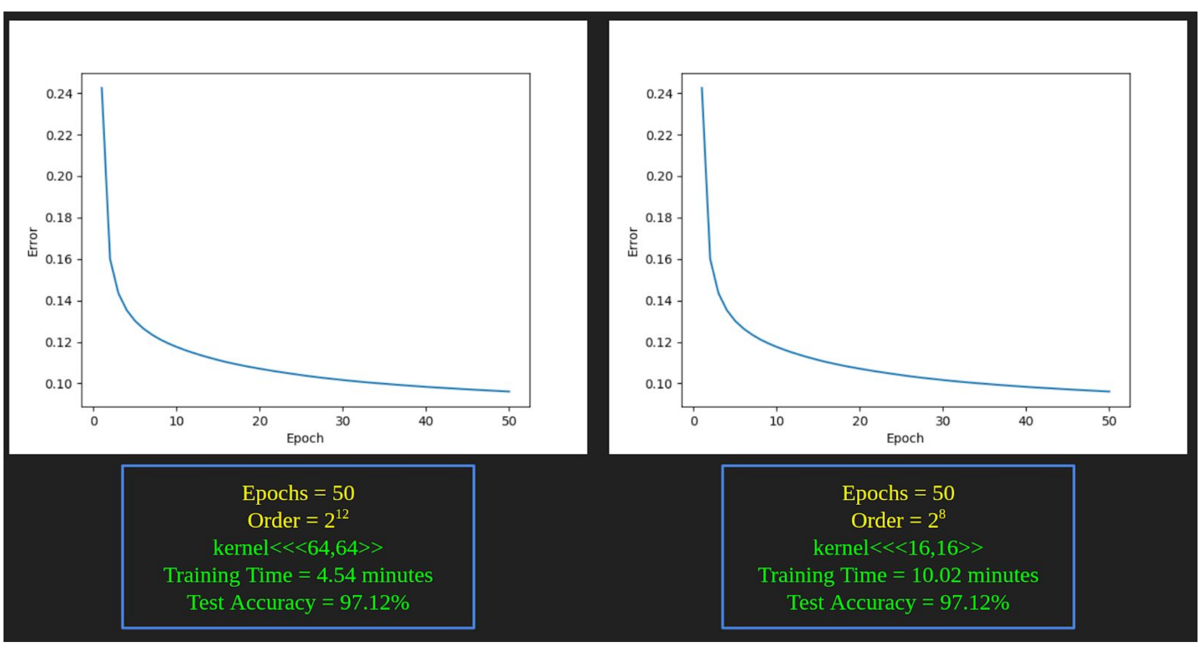
Similar Order (product) give similar results

- Higher-order (product, 2^{16}) gives *inaccurate results* compared to those of lower-order (2^{12}) *of 2*.



The training doesn't improve accuracy a lot after 50 epochs

- As the product's order *becomes smaller*, training time increases even if accuracy remains the same (2^8 *order* v/s 2^{12} *order*).



Training time increases although the accuracy is the same

6. Conclusion

Here, we present a simple 3-layer *CNN*[5][6] implemented using *CUDA*[3][4]. We call the resulting implementation ***CuCNN.*** Along with successful training and inference on the *MNIST*[1][2] dataset, we also present some useful insights into the implementation of ML algorithms on CUDA.

Implementation and Report by:

Jitesh Jain (Enrolment Number: **19114039**)

References

1. MNIST handwritten digit database
2. https://github.com/projectgalateia/mnist#mnist
3. Getting Started with CUDA
4. About CUDA
5. ImageNet Classification with Deep Convolutional Neural Networks
6. https://towardsdatascience.com/an-introduction-to-convolutional-neural-networks-eb0b60b58fd7?gi=31c7f978cb3b
7. Google Colab Introduction
8. cuBLAS Library
9. Neural networks and deep learning
10. CS334 on Udacity