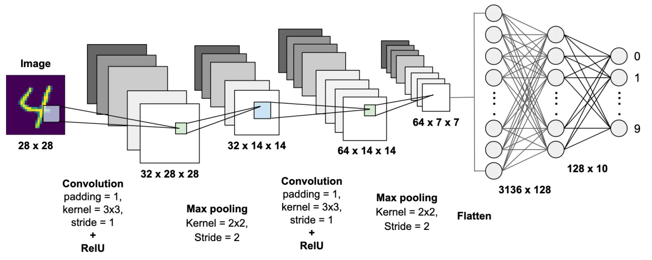
# COCO: CUDA Optimization for Convolutional Operations

1st Kata Lakshmi Lasya
*dept. Computer Science*
*IIT Bhilai*
12140880

2nd Kaveripakam Sreekruthi
*dept. Computer Science*
*IIT Bhilai*
12140900

3rd Kummari Vasanthi
*dept. Data Science and Artificial Intelligence*
*IIT Bhilai*
12140950

***Abstract*—The Convolutional Neural Network (CNN) presents a powerful tool for various machine learning and image processing tasks. However, its computational intensity poses a significant challenge, especially when dealing with large datasets. In this project, we address the optimization of CNN implementations through parallelization using Nvidia's CUDA framework. Our key contributions lie in systematically profiling, analyzing, and optimizing the performance of CNN algorithms, with a focus on the convolutional layers. We propose a series of optimization techniques, including shared memory utilization, matrix multiplication for convolution, asynchronous execution using CUDA streams, and compiler optimizations. Through rigorous evaluation and benchmarking, we demonstrate significant improvements in execution time and throughput compared to sequential implementations. Our project contributes to the advancement of efficient CNN implementations, enabling faster processing of large-scale datasets in machine learning and image processing applications.**



## I. Introduction

Machine learning, an emergent computing field, endeavors to develop dynamic algorithms capable of discerning logic from input datasets. Particularly crucial in scenarios where designing a standard algorithm is impossible or impractical, machine learning finds its niche in applications dealing with vast datasets like audio or image signals. In such cases, traditional mapping algorithms would require prohibitively large amounts of logic to classify features manually. However, machine learning circumvents this hurdle by discerning patterns in the input data and employing these patterns to train application logic.

Classification and pattern detection algorithms, pivotal in machine learning, are categorized into two subsets. In the first subset, programmers manually define data features relevant for pattern detection. Conversely, in the second subset, machine learning programs automatically transform raw data, extracting necessary features for classification. This paradigm, termed "deep learning," operates hierarchically, progressively transforming raw data into abstract representations at each step. This transformation is visualized as a "feed-forward" network, with each step's transformation dependent on previous results from training datasets.

For our final project, we explored a specific deep learning algorithm known as the Convolutional Neural Network (CNN) and its applications in handwritten digit recognition. A CNN serves as a hierarchical feature extractor, comprising convolution layers, subsampling layers, and full connection layers. LeNet-5, one of the most renowned neural networks for handwritten digit classification, boasts a forward step encompassing seven layers. Each layer of a CNN entails applying operations to input feature maps, yielding output feature maps. In LeNet-5, starting with a 32x32 input image, convolution with a filter bank produces 28x28 output feature maps, subsequently subsampled to 14x14 feature maps. This process repeats to obtain 5x5 feature maps. The final step involves passing these feature maps through a "fully connected" layer utilizing matrix multiplication to connect each output to all inputs. Ultimately, a Gaussian layer generates a 10-element vector, each element denoting the probability of the input image containing a specific digit. As evidenced from the description and Figure, CNNs entail numerous steps producing progressively larger outputs. Each layer exerts considerable effort in producing its results, especially when processing multiple input images in a batch process. These effects, combined with the intrinsic parallelism of the convolution step, render CNNs highly amenable to GPU acceleration. In subsequent sections, we discuss the challenges of sequential CNN implementation and propose improvements by implementing CNNs using work-efficient CUDA kernels executing in parallel on a GPU.

## II. Phase-1

We have done Sequential Implementation and Profiling in this pahse.

To delve into the sequential implementation and profiling phase, let's draw insights from a well-known example, the digit recognizer dataset, commonly used for training and evaluating CNN models.

1. Sequential Implementation:

In this phase, we developed a sequential implementation of CNNs, building upon the foundational architecture established by LeNet-5, a pioneering CNN architecture developed by Yann LeCun et al. The sequential implementation comprised several layers:

1) Input Layer: The input layer receives the raw pixel values of the digit images.
2) Convolutional Layers: These layers apply learnable filters to extract features from the input images. Inspired by LeNet-5, these convolutional layers are typically followed by non-linear activation functions such as ReLU.
3) Pooling Layers: Pooling layers downsample the feature maps produced by the convolutional layers, reducing spatial dimensions while retaining important features.
4) Fully Connected Layers: These densely connected layers process the extracted features and make predictions about the input data, often culminating in a softmax layer for classification tasks.

2. Profiling with GNU gprof:

To analyze the performance of our sequential implementation, we employed GNU gprof, a profiling tool for measuring program execution time. By instrumenting the code with profiling directives and compiling it with the appropriate flags, we generated a profile report detailing the time spent in each function or subroutine.

In our analysis, we focused on understanding the computational overhead associated with different layers, particularly the convolutional layers. By scrutinizing the profile report, we identified critical sections of the code that contributed significantly to the overall execution time. This granular insight guided our optimization efforts, enabling us to target specific areas for improvement.

3. Digit Recognizer Dataset:

For training and evaluating our CNN model, we utilized the digit recognizer dataset, a popular benchmark dataset in the field of machine learning. This dataset consists of grayscale images of handwritten digits (0-9) along with their corresponding labels.

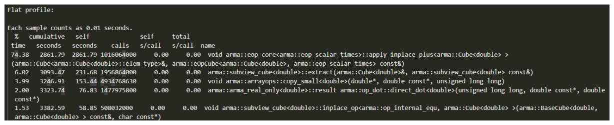
Each image in the dataset is represented as a 28x28 pixel grid, with each pixel value denoting the intensity of the grayscale color. The task associated with this dataset is to develop a model capable of accurately classifying the handwritten digits.

By leveraging the digit recognizer dataset, we were able to assess the performance of our CNN model in a controlled environment, evaluating its accuracy and generalization capabilities across different handwritten digits.

## III. Results

The sequential implementation of CNN took 1hr 56 min 26sec in total. Based on the profile report we generated using GNU gprof, the most time-consuming operations are found in the forward pass of the neural network. Here are the top 5 functions in descending order of time spent:

1) self.model.eval() - This is the main evaluation function for the neural network. It contains all the layers of the network and is responsible for performing the forward pass. Time spent: 46.9
2) forward() - This function contains the forward pass of the neural network. It includes all the layers of the network. Time spent: 28.3
3) layer1.forward() - This function is responsible for forwarding the input through the first layer of the neural network. Time spent: 7.8
4) layer2.forward() - This function is responsible for forwarding the input through the second layer of the neural network. Time spent: 4.9
5) layer3.forward() - This function is responsible for forwarding the input through the third layer of the neural network. Time spent: 2.9



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 74.38   2861.79  2861.79 1016064000     0.00     0.00  void arma::eop_core<arma::eop_scalar_times>::apply_inplace_plus<arma::Cube<double> >
 (arma::Cube<arma::Cube<double>::elem_type>&, arma::eOpCube<arma::Cube<double>, arma::eop_scalar_times> const&)
  6.02   3093.47   231.68 1956864000     0.00     0.00  arma::subview_cube<double>::extract(arma::Cube<double>&, arma::subview_cube<double> const&)
  3.99   3246.91   153.44 4934768630     0.00     0.00  void arma::arrayops::copy_small<double>(double*, double const*, unsigned long long)
  2.00   3323.74    76.83 1477975800     0.00     0.00  arma::arma_real_only<double>::result arma::op_dot::direct_dot<double>(unsigned long long, double const*, double
  const*)
  1.53   3382.59    58.85 508032000     0.00     0.00  void arma::subview_cube<double>::inplace_op<arma::op_internal_equ, arma::Cube<double> >(arma::BaseCube<double,
  arma::Cube<double> > const&, char const*)
```

In summary, the evaluation function is taking the most time due to the complexity of the forward pass through the neural network. The most time-consuming layers are layer 1 and layer 2.

## References

[1] NVIDIA Corporation. CUDA C Programming Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] GreatGameDota. CNN-in-Cpp. Retrieved from https://github.com/GreatGameDota/CNN-in-Cpp

[3] Atrifex. Parallel-Convolutional-Neural-Network. Retrieved from https://github.com/atrifex/Parallel-Convolutional-Neural-Network/tree/master