COP 290 Design Practices Report

Kanish Garg

kanishgarg428@gmail.com

Harshit Jain

harshitjain1371999@gmail.com

February 17, 2019

1 Image processing library

We created our own image processing library, and used it to implement a Convolution Neural Network(LeNet) for handwritten digit recognition.

1.1 Functions Implemented

- **convolution** of a square input matrix and a square kernel, both matrices of any size and the kernel smaller than the input.
  - with and without input padding to maintain or reduce the size of the input matrix.
  - implemented the function both as convolution and as matrix multiplication (using Teoplitz matrix).
- non-linear activations of an input matrix of any size with **relu** and **tanh** functions.
- sub sampling of square input matrices of any size with **max pooling** and **average pooling** functions
- converting a vector of random floats to a vector of probabilities with **softmax** and **sigmoid** functions

Note: All the functions are implemented using 32 bit float as datatype. Also some other sub-functions are implemented as a part of these functions and can be used independently.
Makefile: We wrote a makefile which simplifies making program executables from a project module having its dependencies at different locations making use of linker flags and other parameters.

2 Accelerating Matrix Multiplication

The matrix multiplication that we are doing is an $O(n^3)$ algorithm. Now, to accelerate the performance of convolution, we used some linear algebra libraries which implement the optimized versions of functions like matrix multiplication specifically for Intel processors leveraging the parallel computational capability of modern processors and the optimized cache usage.
The libraries which we have used are namely **Intel MKL and OpenBLAS**.

Also, we ourselves tried to make the matrix multiplication faster by using multiple threads so that there will be some level of parallelism in calculating the multiplication and hence would enhance the performance. We used **pthreads** for implementing the optimized version. To optimize our results more we used flattened arrays so as to reduce cache misses.

3 Performance Comparison

We implemented matrix multiplication using all the above mentioned approaches and compared their performances for different matrix sizes. We convoluted nxn matrices with nx1 matrices for various values of n and the results obtained are shown in the figure below.

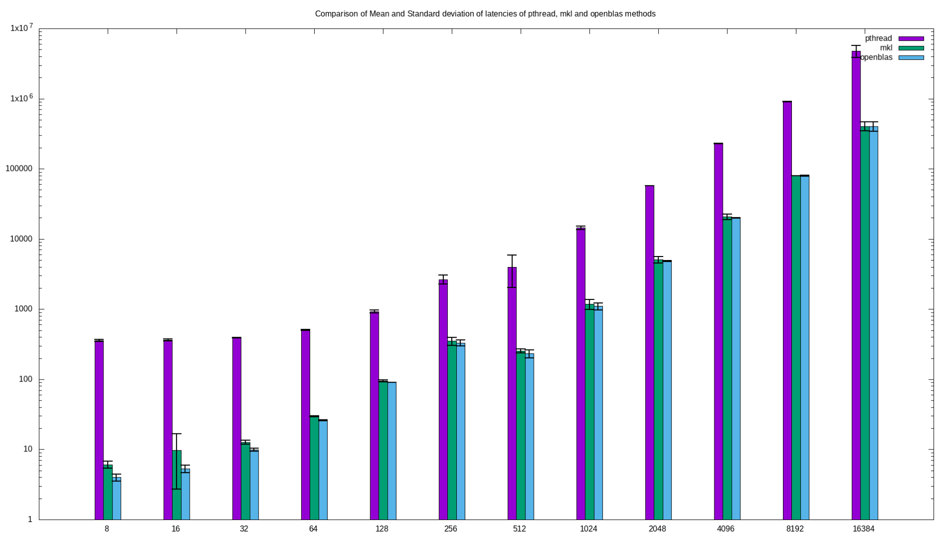


Figure 1: Performance comparison of MKL, OpenBLAS and pthreads

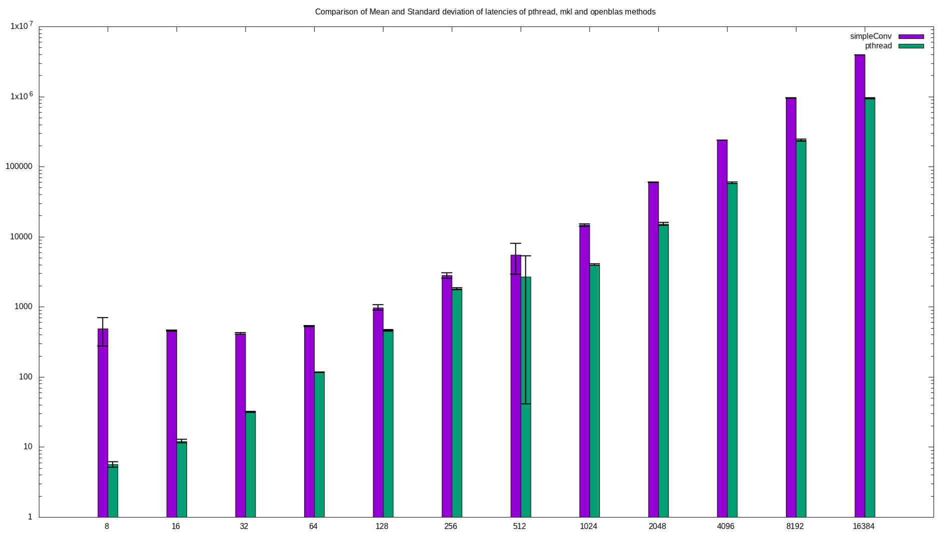


Figure 2: Performance comparison of pthreads and Normal method

Time Statistics (μs)				
Matrix size(n)	MKL time	OpenBLAS time	pthread time	Normal time
8	6.1	4.0	358.45	850.0
16	9.7	5.3	366.45	905.47
32	12.7	9.9	391.7	927.89
64	29.65	26.05	501.9	1012.24
128	95.1	90.2	930.55	1208.0
256	349.35	330.3	2656.25	3284.35
512	253.5	232.6	3979.6	5506.5
1024	1180.8	1100.95	14402.2	20891.45
2048	5067.35	4846.7	57628.4	69864.2
4096	20638.3	19993.5	228283	30864.4
8192	79609.1	79533.8	905226	1124986.5

Table 1: Time Comparison of Matrix multiplication $\left(\left[n \times n\right] \text{ with } \left[n \times 1\right]\right)$

The above results show us that the linear algebra libraries (MKL and OpenBLAS) indeed have a better performance than even our optimized implementation using pthreads. Also, OpenBLAS does seem to perform better than MKL for smaller matrix sizes but as the matrix sizes increase further, MKL starts performing essentially better than OpenBLAS. Moreover, our implementation with pthreads also starts to compete more closely with the other two as the matrix sizes increase. Moreover, the data also tells us a very subtle point that pthread seems to have an extra overhead i.e. even for matrix size of 8, the time is $358.45\mu s$, which suggests that creating multiple threads does cost an initial loss in performance which although becomes less significant if we go on increasing matrix sizes. Also, we can see that single threaded matrix multiplication performs worst especially for larger matrix sizes.

4 LeNeT: MNIST Handwritten digit Recognition

The last part of the assigment was to integrate the above implemented functions into a CNN (LeNet). The architecture of the neural network was given to us with pretrained weights. So, we built together layer by layer using the above functions and actually made a Neural network which takes an image (28x28) as the input and gives the top five probable digits with respective probablities as the output.

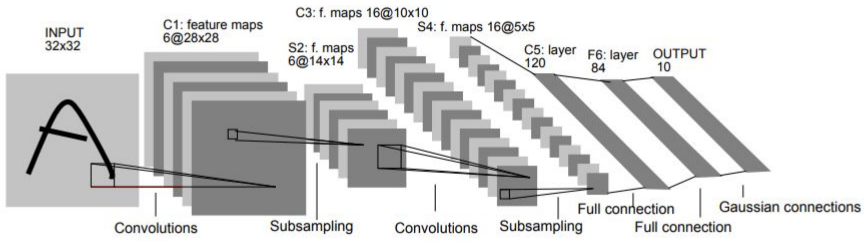
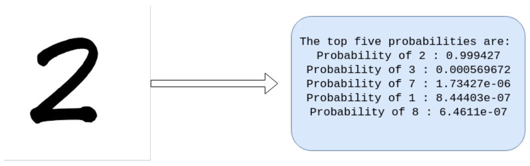
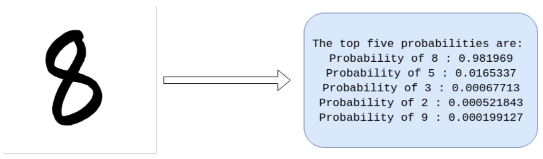


Figure 3: Architecture of LeNeT

Usage example:



```
The top five probabilities are:
  Probability of 2 : 0.999427
 Probability of 3 : 0.000569672
 Probability of 7 : 1.73427e-06
 Probability of 1 : 8.44403e-07
  Probability of 8 : 6.4611e-07
```



```
The top five probabilities are:
  Probability of 8 : 0.981969
  Probability of 5 : 0.0165337
 Probability of 3 : 0.00067713
 Probability of 2 : 0.000521843
 Probability of 9 : 0.000199127
```

References

[1] Colby College. A simple makefile tutorial.

[2] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[3] Muhammad Rizwan. Lenet-5 – a classic cnn architecture, 2012.