COP290 (Design Practices)

Performance Report : Task 1

Kshitij Alwadhi (2019EE10577)[1] and Vaibhav Verma (2019EE10543)[2]

[1]Department of Electrical Engineering, IIT Delhi
[2]Department of Electrical Engineering, IIT Delhi

12 February 2022

1 Introduction

In the assignment given to us in Task 1, we make a library for classifying a given audio clip into various classes. This library has been broken down into various functions which we will finally stitch together at the end. One of the components used in this library is the Fully Connected Layer. This component comprises of a **matrix multiplication** step (multiplication of the weight matrix with the input matrix). Now there are various ways in which we can perform this matrix multiplication and we explore their performance in this report.

1.1 Naive Method

The naive method for matrix multiplication is the $O(n^3)$ implementation which we use as the baseline for making our comparisons. Little to none optimization has been done in this implementation so as to get the complete picture of how much improvement we can get from using other libraries.

1.2 Threading (using pthreads)

To make the naive method of matrix multiplication faster, we bring about parallelism in the other for loop of the code. To implement this, we make use of **pthreads** in C++. We restrict the number of threads to 4 for our system in which we perform these experiments.

1.3 Linear Algebra Libraries

We also try out various libraries which are highly optimized for linear algebra tasks (matrix multiplication in our case). These libraries include **OpenBlas** and **Intel MKL**. We use these libraries as a black box but under the hood they have various levels of optimizations such as utilising multiprocessing and better caching.

2 Experimental Setup

To perform our experiments, we perform matrix multiplication of two matrices each of size $n \times n$. These matrices are randomly generated by the code. We vary the parameter n across a range of values and at every iteration, we multiply the previous value of n by 2. Moreover, to get stable results, at each iteration, we perform this calculation 30 times and take the mean/standard deviation.

3 Performance Comparison

3.1 Performance vs Number of Threads

In this set of experiments, we observe the performance of implementation as we vary the number of threads. We vary the number of threads from 1 to 4.

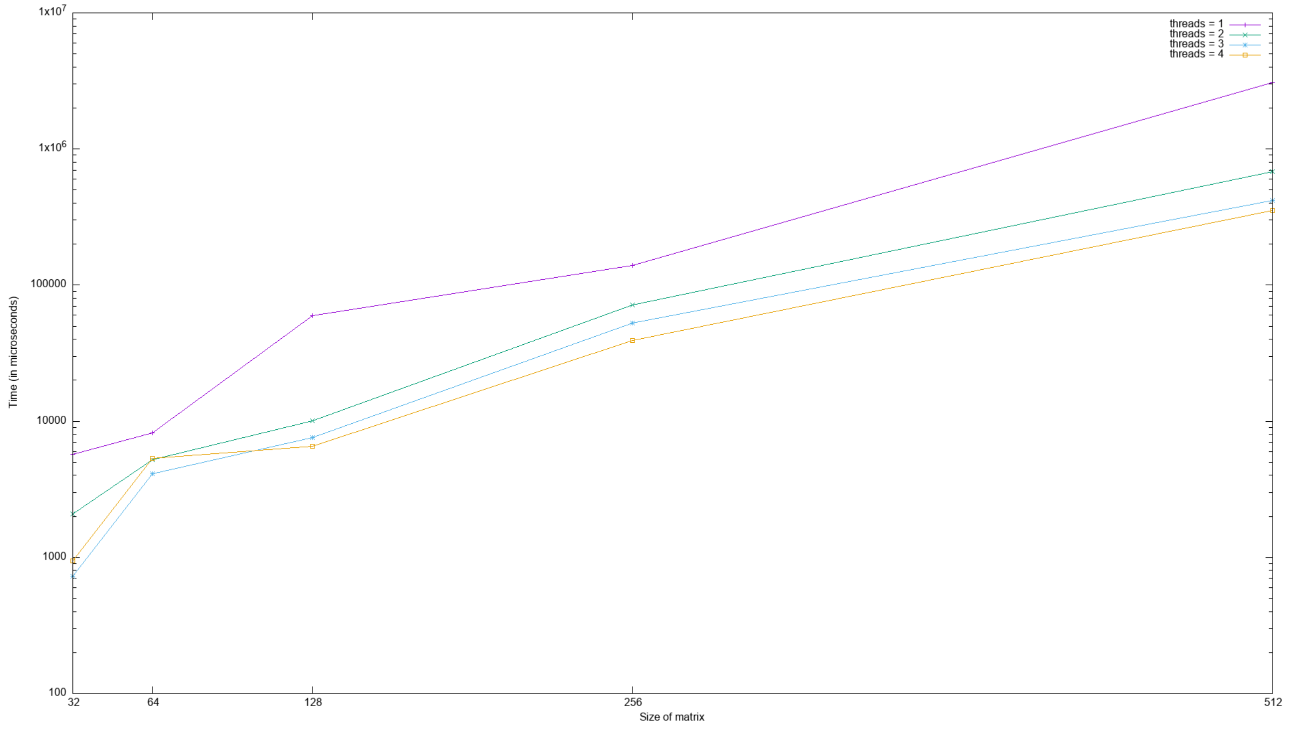

Figure 1: Time vs. Size of Matrix for Fixed Number of Threads

As seen on the X-axis, we have chosen the range [32, 64, 128, 256, 512] for the size of the matrices. We also tried running our implementation for values beyond this range, however, the matrices were too large to fit into the memory. Hence the upper limit of n has been chosen as 512.

n	**num_threads**	**time(μS)**
32	1	5710.13
32	2	2073.8
32	3	730.433
32	4	943.9
64	1	8247.97
64	2	5193.43
64	3	4097.07
64	4	5336.47
128	1	59544.5
128	2	10092.8
128	3	7571.97
128	4	6572.23
256	1	138898
256	2	71755.1
256	3	52945.7
256	4	39400.5
512	1	3074930
512	2	683593
512	3	418104
512	4	352216

Table 1: Performance Comparison across different number of threads

3.2 Performance across various implementations

Here, we bring about the comparison across all the implementations that we have in our code. We present the observed data through boxplots for $n = [32, 64, 128, 256, 512]$. Note that the number of threads in the pthread implementation has been set at 4 in these experiments and the time reported is in μS.

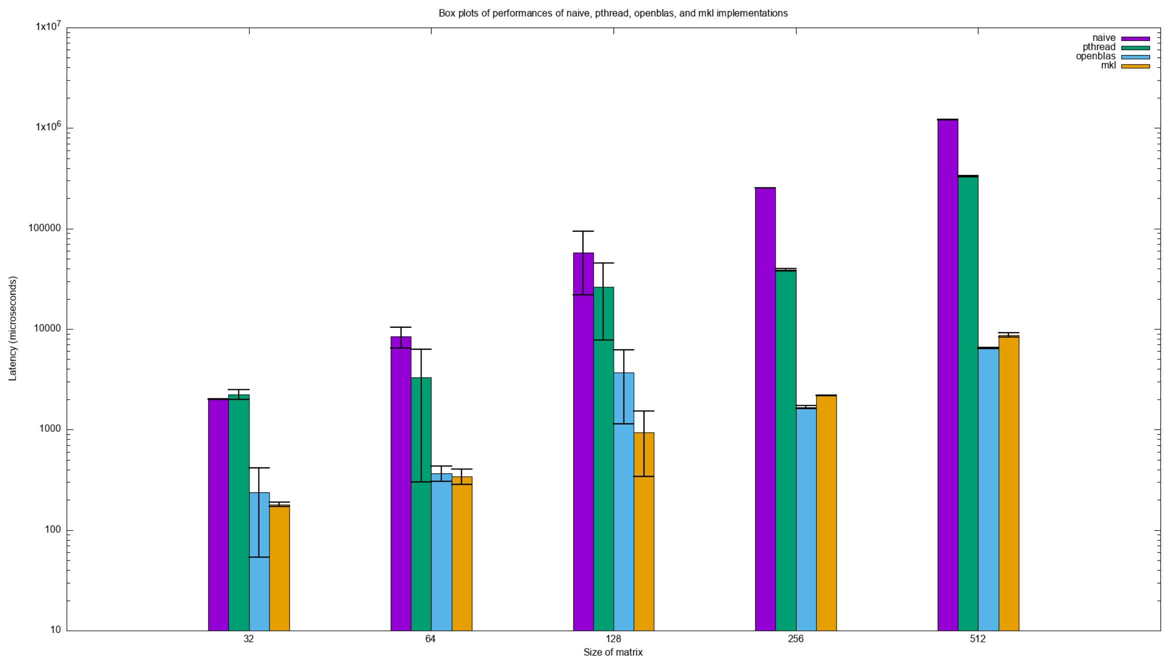

Figure 2: Performance across different implementations

n	**Naive**	**pthread**	**openBLAS**	**MKL**
32	2009.23	2239.87	235.833	180.333
64	8449.83	3309.23	368.9	343.1
128	57857	26450.8	3674.37	939.7
256	255061	38711.1	1672.27	2191.37
512	1219320	330361	6461.6	8742.47

Table 2: Performance Comparison across different implementations

4 Conclusion

From figure 1, we conclude that having more number of threads leads to better performance, however, the results don't necessarily scale linearly with the increase in number of threads. From figure 2, we observe that we are able to get a significant increase in performance when we move from our implementations to the linear algebra libraries. The performance of both the libraries (openBLAS and MKL) is similar with openBLAS performing slightly better for bigger inputs. Moreover, we do see an improvement in performance when we switch to pthreads from the naive approach. However the performance of the libraries is superior as they are much more optimized. However, when the input size is small (32), the pthreads approach performs poorer as more time is spent in initializing the threads as compared to time spent in the actual computation.