Image Processing Libraries

Priyanshu Gautam
cs1170362@cse.iitd.ac.in

Sameer Vivek Pande
cs1170371@cse.iitd.ac.in

2019/02/16

1 Introduction

Image processing has been on the rise with the advent of neural networks and machine learning in the past decade. It has therefore become almost absolutely necessary to have image processing libraries to perform convolution, to calculate correlation, linear activations and all the other functions which are the building blocks of any image recognition API. In our assignment one, we had to implement this only. In our first subtask we made our own functions for convolution, relu activation, tanh activations, convolution using matrix multiplication, convolution using iteration and so on. In the subtask 2, what we were required to do was to make this process more efficient via exploring

- Already available Linear Algebra libraries having all the optimisation techniques like mkl and openBlas
- Exploring multi threading for making the process of matrix multiplication parallel and more efficient.

In this subtask we shall go through each library and compare their performances on different sets of inputs.

2 MKL and Open library

IntelMKL [1] and OpenBlas [2] have been used to boost matrix multiplication. Both use a function named "cblas_sgemm" to peform accelerated matrix multiplication. The function takes two input matrices as float* and their specifications (ex. row-major or column-major etc.) and returns computed answer as a float* with specified order (row-major or column-major). We have observed that when the input size and kernel sizes are smaller, the MKL and OpenBlas is outperformed by regular convolution but as size of input and filter increases, these libraries start showing significant improvements over the regular convolution methods.

3 Pthread

To implement multi threading, we made use of pthread module of C++. This allowed us to perform parallel computations in matrix multiplication, as to fully utilise our CPU resources. But for this implementation to be any faster, our number of threads must be equal to the number of cores in the processor, otherwise this technique renders itself to be useless. If number of threads are more, waiting time exists, and the overhead time cost of creating a thread makes the process even slower. Therefore in order to optimise our function we set the number of threads to be 2 which will in most cases decrease the time. For faster time, you can pass the argument as a parameter. The overhead time cost is negligible for large and small values of filters and inputs but for intermediate values, it becomes considerable enough

and the normal multiplication becomes as fast as pthread implementation. It can be seen in the tables 2 and 1 and also in figures 1 and 2.

4 Comparison

To compare how the publicaly available libraries - mkl, and openBlas, and our multi threading implementation fared against our original matrix multiplication, we did a comparative study of all of these by running them on a random set of inputs.

For doing this comparative study, we calculated the time taken by each of them to multiply two random matrices (one input matrix, and the other kernel), with the size of the input matrix being varied from 32 to 1024, and the size of kernel taken as 3 and 5 for all the input matrices. To calculate the time elapsed we made use of **high_resolution_clock** class present in **chrono** library. All the data was calculated in seconds, for kernel size of 3, the data is shown in table 1 while for the kernel of size 9, it is shown in table 2

Table 1: Comparative Data for kernel size 3

Size of input	MKL	OpenBlas	Pthread	Normal
32	4.481424×10^{-3}	3.480212×10^{-3}	3.102635×10^{-3}	6.335516×10^{-5}
64	3.220144×10^{-2}	3.823522×10^{-2}	2.418458×10^{-3}	2.683123×10^{-4}
128	3.878288×10^{-2}	4.375075×10^{-2}	3.697864×10^{-3}	1.192761×10^{-3}
256	3.580013×10^{-2}	3.660659×10^{-2}	7.239761×10^{-3}	4.761522×10^{-3}
512	5.115314×10^{-3}	5.414858×10^{-3}	6.146626×10^{-3}	9.997924×10^{-3}
1024	2.99761×10^{-2}	1.040341×10^{-2}	1.900039×10^{-2}	3.13613×10^{-2}

Table 2: Comparative Data for kernel size 9

Size of input	MKL	OpenBlas	Pthread	Normal
32	3.455654×10^{-2}	3.61308×10^{-2}	2.772176×10^{-3}	3.512465×10^{-4}
64	3.818058×10^{-2}	4.121977×10^{-2}	4.273913×10^{-3}	1.845449×10^{-3}
128	3.791414×10^{-2}	4.48504×10^{-2}	8.704772×10^{-3}	8.639242×10^{-3}
256	7.492658×10^{-3}	7.079206×10^{-3}	1.154506×10^{-2}	1.860959×10^{-2}
512	3.504662×10^{-2}	1.527943×10^{-2}	4.453151×10^{-2}	7.366425×10^{-2}
1024	6.969363×10^{-2}	6.960464×10^{-2}	1.602464×10^{-1}	2.668867×10^{-1}

From these tables we see that for smaller values of kernel and input size, our normal multiplication was faster than MKL and openBlas libraries. Our multi threading version was almost twice as fast as normal at these small sizes. As we begin to increase the size, we can see that pthread becomes slower. It is due to the overhead cost of creating threads. And as our sizes become ≈ 256, MKL and openBlas emerge out to be the fastest. As our normal matrix mutiplication of the order of $\mathcal{O}(n^3)$, it begins to diverge while the optimisations in the libraries make them the fastest at this point.

In the figure 1 we can see this variation better via means of box plots with error bars how this varies, wherein normal multiplication if the fastest for smaller value and the fastest for larger values for filter size 3.

While figure 2 shows the dependence for filter size of 9.

5 Conclusions

After implementing each of the above models, we see that for small values of filter sizes, normal multiplication and pthread are the fastest (pthread is the faster of the two), and as we increase the sizes MKL

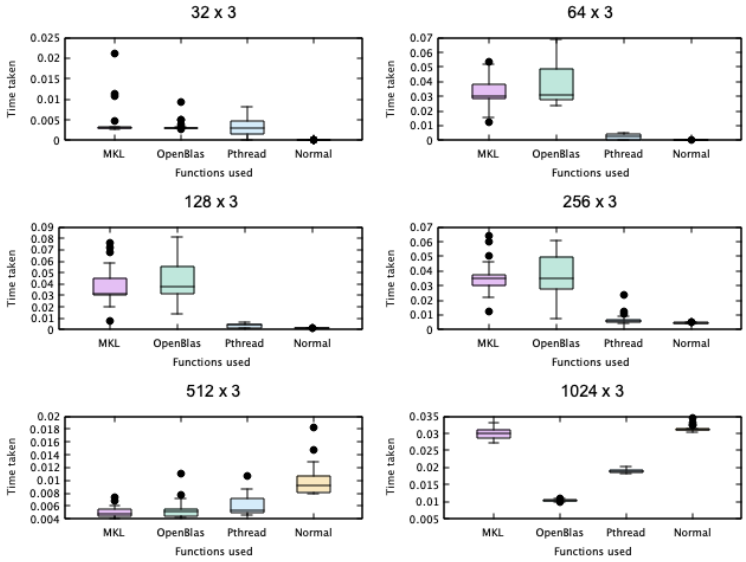

Figure 1: Box plots for all 4 implementation for all filters and input matrix

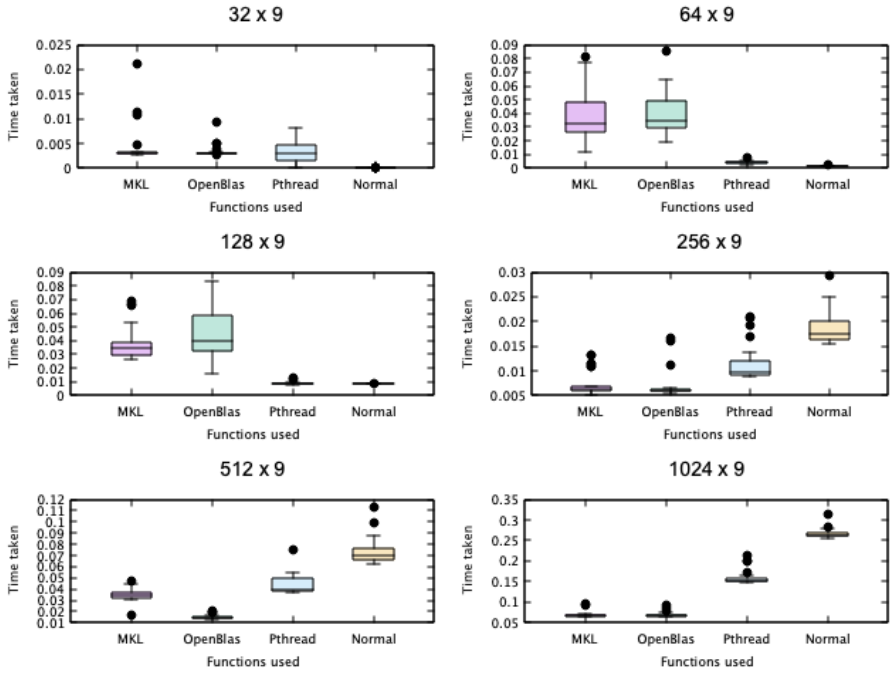

Figure 2: Box plots for all 4 implementation for all filters and input matrix

and OpenBlas become the fastest with OpenBlas a tad faster than MKL.

References

[1] Intel. Mkl library. *Link to documentation.*

[2] Open source. Openblas. *Link to documentation.*