COP-290
ACCELERATING MATRIX MULTIPLICATION

Mohammad Kamal Ashraf
E-mail : cs1170589@cse.iitd.ac.in

Mayur Solanki
E-mail: cs1170349@cse.iitd.ac.in

February 17, 2019

1 Introduction

Linear Algebra has always been central to the development of numerous fields in Computer Science. From computer vision to machine learning to parallel computing, everywhere most of the modelling and computation follows models arising from Linear Algebra.

The intensive and extensive use of linear algebra operations in computing necessitates the development of efficient algorithms and optimizations that could allow us to exploit the computing resources for development of better and effective applications. Most algorithms based on linear algebra extensively use vector and matrix operations. Thus a very high level of speed-up can be achieved by optimizing vector and matrix operations.

In this report we present the results obtained through the use of Linear Algebra Libraries and our own effort at using parallelism through POSIX pthreads to optimize Matrix Multiplication.

2 Linear Algebra Libraries

As the linear algebra operations are such central to a lot of algorithms, a lot of optimized libraries are available for such tasks. Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector additions, scalar multiplications, dot products, linear combinations, and matrix multiplications.

We used two implementations of the BLAS provided by Intel®Math Kernel Library(MKL) and an open source library OpenBLAS and compared the results obtained from these. In both the implementations we have used the following function form BLAS: cblas_segmm - matrix multiplication for short point precision.[1]

void cblas_sgemm(const enum CBLAS_ORDER __Order, const enum CBLAS_TRANSPOSE __TransA, const enum CBLAS_TRANSPOSE __TransB, const int __M, const int __N, const int __K, const float __alpha, const float *__A, const int __lda, const float *__B, const int __ldb, const float __beta, float *__C, const int __ldc);

2.1 Intel®Math Kernel Library(MKL)

Intel MKL has been developed by Intel Corporation and these libraries have been optimized specially for Intel processors. In fact these libraries work the best with Intel's own C\C++ compilers. The MKL library provides implementations for BLAS, LAPACK, and FFTW functions.[2]

The basic functions for linear algebra are provided by BLAS and here we implemented this library for matrix-vector multiplication using the GNU g++ compiler for C++. We used the cblas_sgemm function for performing matrix multiplication on matrices stored in contiguous storage.

This library was significantly slower than OpenBLAS but on increasing the input matrix size the MKL tends to come out better. We present the latency in milliseconds for increasing matrix sizes with MKL implementation.

2.2 OpenBLAS

OpenBLAS is an open source implementation of the Basic Linear Algebra Sub-programs(BLAS). It has been developed by the Lab of Parallel Software and Computational Science, ISCAS and claims to achieve performance comparable to that of MKL.[4]

Here again we used the cblas_segmm function for performing matrix multiplication. however due to the conflict between MKL and OpenBLAS, we kept the OpenBLAS functionalities in different folder with a different make. In our implementation the OpenBLAS was significantly faster than any other implementation though in the higher matrix sizes the difference between MKL and OpenBLAS seems to become narrow.

3 Performance Comparison

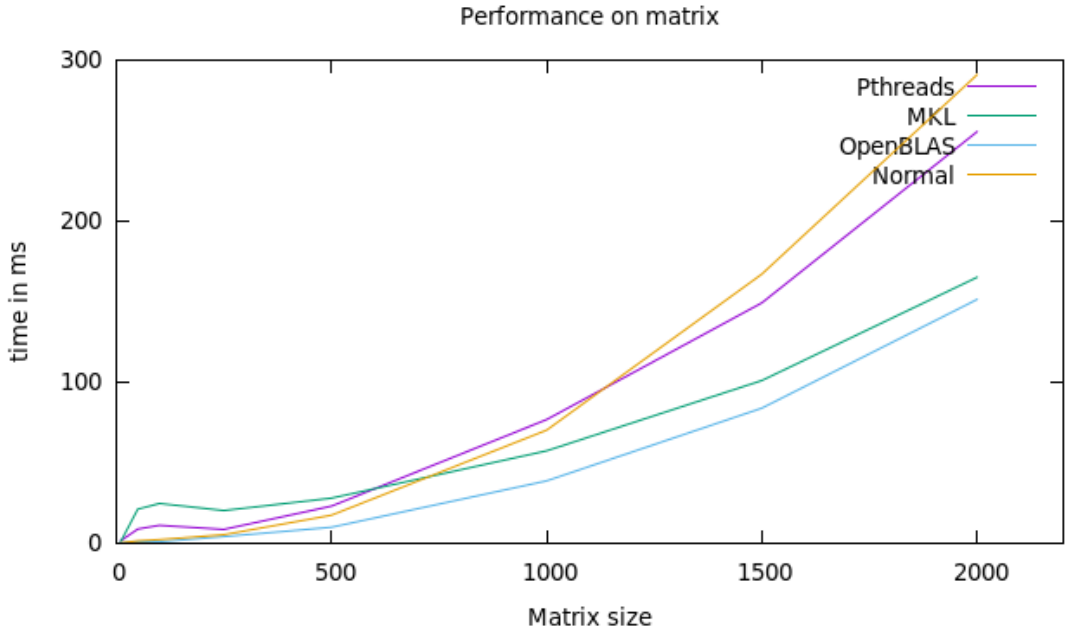
3.1 Compariosn Over Matrix Size

For the comparison of performance we used multiplication of a square matrix input converted to a Toeplitz matrix with a vector as the common run case as this was what our algorithm for convolution for images required. We ran the test on a kernel of order 2 over different matrix sizes with the same dataset for each test.

The latency in milliseconds was calculated using standard C++ "crono" library and the results were as follows:

Comparison of Performance Across Input Matrix Sizes

Matrix Size (Matrix Order)	Pthreads Latency (ms)	Math Kernel Library (MKL) Latency (ms)	OpenBLAS Latency (ms)	Normal Latency (ms)
10	1.3489	0.6436	0.1758	0.042493
50	8.675	21.032	0.9	1.36251
100	10.98	24.5025	0.7442	2.11072
250	8.45	20.1742	3.954	5.1453
500	22.8862	27.8571	9.7985	17.2756
1000	76.6745	57.2758	38.63	70.1492
1500	149.212	100.963	83.82	167.072
2000	255.471	165.051	151.265	290.861

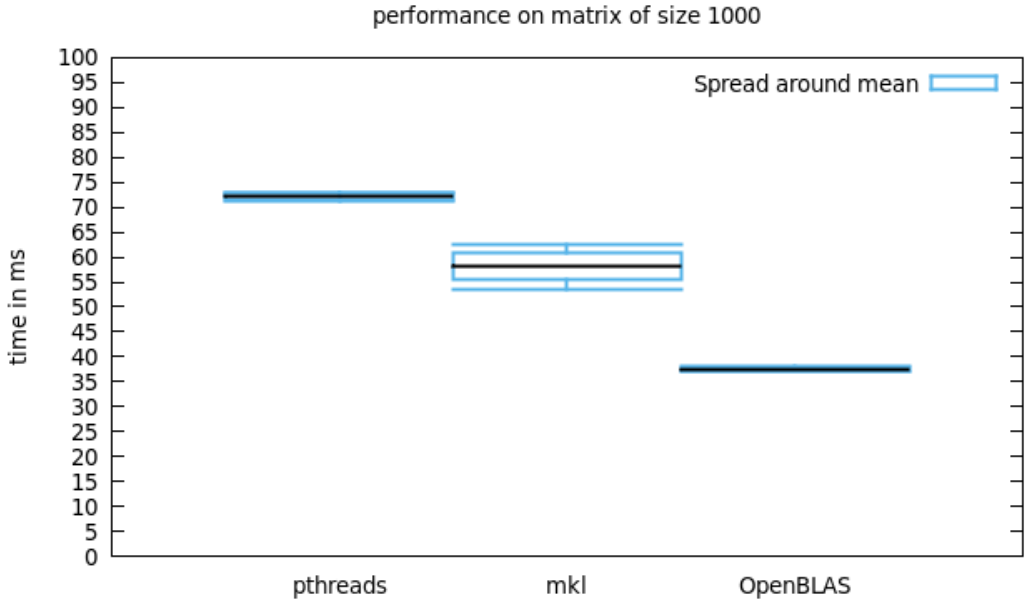


3.2 Mean and Standard Deviations

For the boxplot, we ran our three optimized implementations several times over input matrix size of 1000 and took the mean and standard deviation of the latency in milliseconds(ms). In the boxplot we have provided the mean and the standard deviation spread around mean.

Mean and Standard Deviations

Implementation	Mean Latency (ms)	Standard Deviation
Pthreads	72.18661	0.9157
MKL	58.2962	5.9971
OpenBLAS	37.45367	0.5010



4 Acceleration with Pthreads

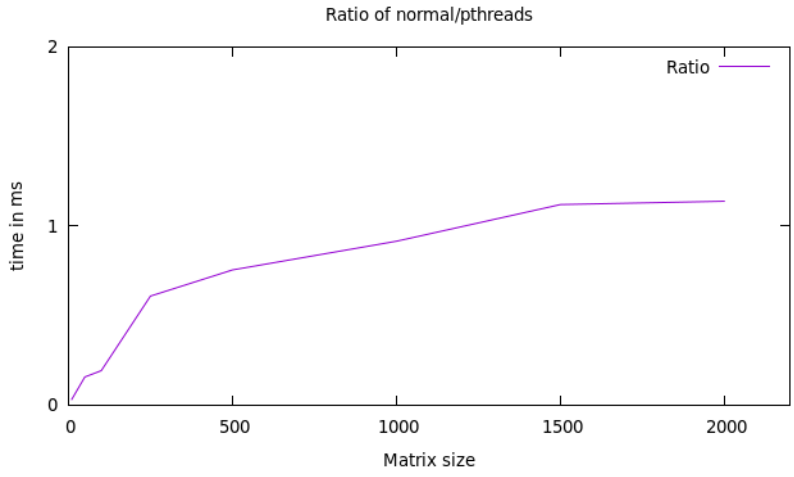
4.1 Implementation strategy for Pthreads

For the Pthreads implementation we used multi-threading[3] for optimizing the operations as far as possible. We maintained an approach such that the number of threads scale as the input dimensions scale.

- In our implementation we have optimized the matrix multiplication using number of threads equal to the number of rows in the Toeplitz Matrix.
- Earlier we used the number of threads equal to the number of elements in the output matrix but we realised that doing so will create a large number of threads that may adversely affect the performance and so we decided to reduce the number of threads.
- We also tried to implement threading in the functions for conversion of input image to Toeplitz matrices but we did not do this as the implementations from MKL and OpenBLAS also use this code and so we maintained uniformity and made comparison only on the basis of matrix multiplication sections of code.

4.2 Graphical comparison with Normal implementation

The following graph provides the ratio of the latency in the normal implementation and the threaded implementation over different matrix sizes.



Observations

- In the lower region, with low matrix dimensions, the threaded performance is significantly poor compared to the norml performance.
- In the higher values of matrix dimensions, the threaded performance overtakes the normal performance and there is significant improvisation in performance continually.

References

[1] Learnig about blas. `https://software.intel.com/en-us/mkl-developer-reference-c-cblas-gemm`.

[2] Math kernel library download. `https://software.intel.com/en-us/mkl`.

[3] Mulithreading tutorial. `https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm`.

[4] Openblas download. `https://www.openblas.net`.