Report

Exercise 01:

Computing of Matrix Multiplication using C++

Group: ex01_group02

1. Eibl, Sebastian
2. Schottenhamml, Julia
3. Sen, Karnajit

Contents

1. Objectives:

Objective of this report is to implement a matrix-matrix multiplication C = A X B, where A is a M X K matrix, B is a K X N matrix, and C is aM X N matrix.

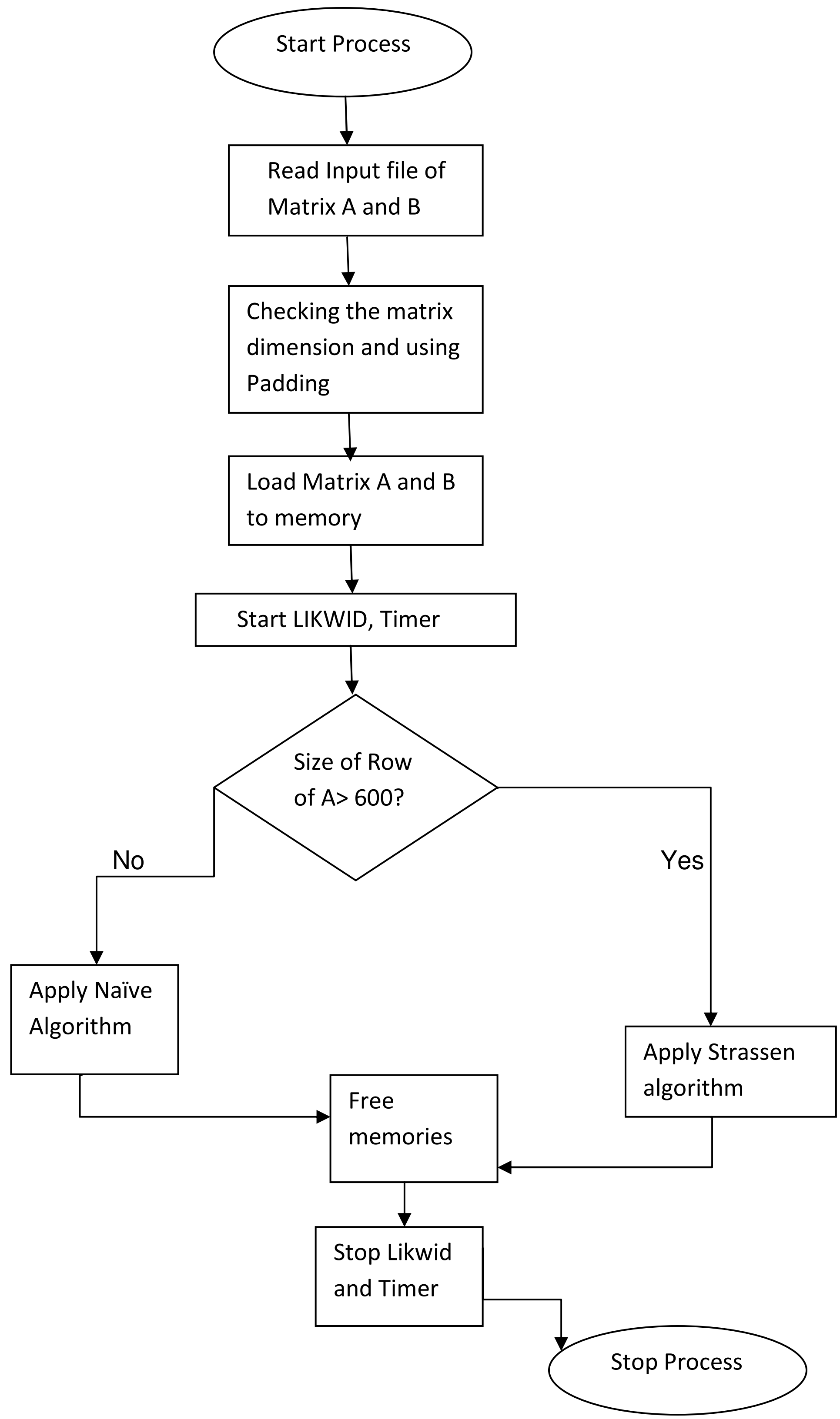
2. Description of the task:

Refer to the attached document here.

ex01.pdf

3. Design/Flowchart:

Following diagram is the flow chart of the implemented C++ program.


Start Process
Read Input file of Matrix A and B
Checking the matrix dimension and using Padding
Load Matrix A and B to memory
Start LIKWID, Timer
Size of Row of A> 600?
No
Yes
Apply Naïve Algorithm
Apply Strassen algorithm
Free memories
Stop Likwid and Timer
Stop Process

4. Methodologies:

These are the key points used in the implementation.

1. A separate class "matrix" (implemented in matrix.hpp) is used for all the basic operations and properties of a Matrix. It is kind of a wrapper class around the native double array. It also handles access to sub blocks of the matrix.
2. For the transposition a blocking size of 128 is used.
3. All the matrix rows are padded by 64 to avoid cache threshing.
4. **Advanced Vector Extensions** (**AVX**) Instruction set (a new 256 bit Instruction set features provided by Intel processor) is used while fetching data from memory in matrix multiplication operation both in naïve and Strassen modules.
5. In naïve multiplication, block size of 4 X 4 is considered.
6. AVX methodologies like _mm256_blend_pd(), _mm256_hadd_pd(), _mm256_permute2f128_pd() and _mm256_add_pd() are used for the addition of multiplied components of A with B.
7. More optimized recursive Strassen algorithm is implemented to measure the multiplication of matrix size more than 600.
8. Compare.cpp is written to compare the output matrix of the program to the standard output files. It will confirm us about the correctness of the result.
9. The program is compiled with a Makefile.
10. Double precision floating-point operations are used for the multiplication.
11. 5 stages of optimization techniques as opt1, opt2, opt3, opt4, opt5 has been done. Opt5 is the final version.

5. Optimization stages:

All stages include optimizations of previous stages.

Naïve: Basic implementation of matrix matrix multiplication.

Opt1: Transpose B before doing matrix multiplication to have better memory layout for B.

Opt2: Using AVX instructions to increase performance of naïve matrix matrix multiplication.

Opt3: Loop unrolling for naïve multiplication.

Opt4: Introduced blocking in matrix transposition.

Op5: Implementation of Strassen algorithm.

6. Source Code:

Find the attached files below for the matmult.cpp and Matrix.hpp.

matmult.cpp matrix.hpp

7. Discussion on the Result and Graphical Representations:

Performance and result of the implementation has been measured in all the optXX versions. And the results attached in the Package.

Following is the table for the final opt5 version which will give the different likwid measurement of time taken by the program. It is taking only 0.781054 sec for 2048 X 2048 matrix multiplication. All the time measurements are lesser than the tutor implementation except for 32 X 32 implementation.

Opt 5 Results:

dim1	dim2	dim3	time	L2Bandwidth[MBytes/s]	L2RequestRate	L2MissRate	L2MissRatio	MFlops/s	AVXFlops/s	PackedMUOPS/s	ScalarMUOPS/s
32	32	32	4.12E-05	346.805	0.186532	0.0542	0.290567	4.29474	69.3809	18.4194	2.14638
64	64	64	8.41E-05	1072.01	0.140917	0.038137	0.270631	14.3908	470.286	121.169	7.1954
128	128	128	0.000356	3364.93	0.110735	0.028097	0.253733	37.4148	2455.4	623.199	18.7175
256	256	256	0.002151	10817.6	0.151221	0.048412	0.320139	47.7533	6298.87	1586.65	23.8918
512	512	512	0.015691	24061.5	0.171358	0.048412	0.28252	52.822	14044.5	3524.34	26.4128
1024	1024	1024	0.111356	26653.6	0.180665	0.056845	0.314642	64.9698	17279.2	4336.03	32.4875
2048	2048	2048	0.781054	27568.4	0.183776	0.057751	0.314248	65.5946	17559.6	4406.31	32.7972

Opt 4 Results:

dim	din	dim3	time	L2Bandwidth[MBytes/s]	L2Request Rate	L2MissRate	L2MissRatio	MFlops	32b AVX MFlops/s	PackedMUOPS/s	ScalarMUOPS
32	32	32	0.000181351	390.851	0.0325097	0.00819859	0.252189	155.456	0	0	0
64	64	64	0.000429757	1090.72	0.0163535	0.00232315	0.142059	598.818	0	0	0
128	128	128	0.00209118	4917.15	0.0331493	0.00143158	0.043186	1116.67	0	0	0
256	256	256	0.0142631	7351.06	0.056958	0.0185848	0.326289	1922.7	0	0	0
512	512	512	0.111144	10074.8	0.0556738	0.0182398	0.327619	1740.58	0	0	0
1024	1024	1024	0.958655	7239.13	0.0739376	0.0229468	0.310354	2281.75	0	0	0
2048	2048	2048	8.17724	9050.51	0.0832589	0.0253613	0.304608	2165.75	0	0	0

Opt 3 Results:

dim	dim	din	time	L2Bandwidth[MBytes/s]	L2RequestRate	L2MissRate	L2MissRatio	MFlops/	32bAVXFlops/s	PackedMUOPS/s	ScalarMUOPS/s
32	32	32	0.000194571	228.373	0.134529	0.0357023	0.265388	4.13794	61.5409	16.4232	2.06188
64	64	64	0.000396257	924.84	0.11225	0.0174439	0.155402	11.1243	341.228	88.0963	5.54552
128	128	128	0.000832563	3318.77	0.097433	0.0124141	0.127412	27.5898	1731.08	439.677	13.7752
256	256	256	0.00323364	8706.62	0.230955	0.108098	0.468045	57.2609	7440.71	1874.5	28.6233
512	512	512	0.01582	24941.3	0.22226	0.101927	0.458594	57.4173	14970.7	3757.02	28.7071
1024	1024	1024	0.207343	31869	0.300429	0.101442	0.337657	22.1342	11805.1	2956.8	11.0649
2048	2048	2048	1.68029	31753.4	0.312186	0.102739	0.329095	11.3815	12221.8	3058.3	5.69029

Opt2 Result:

dim1	dim2	dim	Time	L2Bandwidth[MBytes/s]	L2Request Rate	L2MissRate	L2MissRat	MFlops	32bAVXMFlops/s	PackedMUOPS/s	ScalarMUOPS
32	32	32	0.0002284	194.746	0.100936	0.029771	0.294943	3.52274	71.9262	18.8616	1.76266
64	64	64	0.00048	1959.67	0.12618	0.010585	0.0838857	10.0233	420.503	107.63	5.01409
128	128	128	0.0012487	6155.21	0.125442	0.011412	0.0909713	17.921	1510.39	382.077	8.96093
256	256	256	0.0068602	12043.4	0.371693	0.150266	0.404274	30.5772	6044.33	1518.73	15.2888
512	512	512	0.0471417	23008.3	0.336972	0.130501	0.387276	20.0061	7712.9	1933.23	10.003
1024	1024	1024	0.522609	17081.3	0.387659	0.142411	0.367363	8.09231	6952.78	1740.22	4.04595
2048	2048	2048	4.64301	17782	0.435596	0.147877	0.339482	3.60999	6987.21	1747.7	1.805

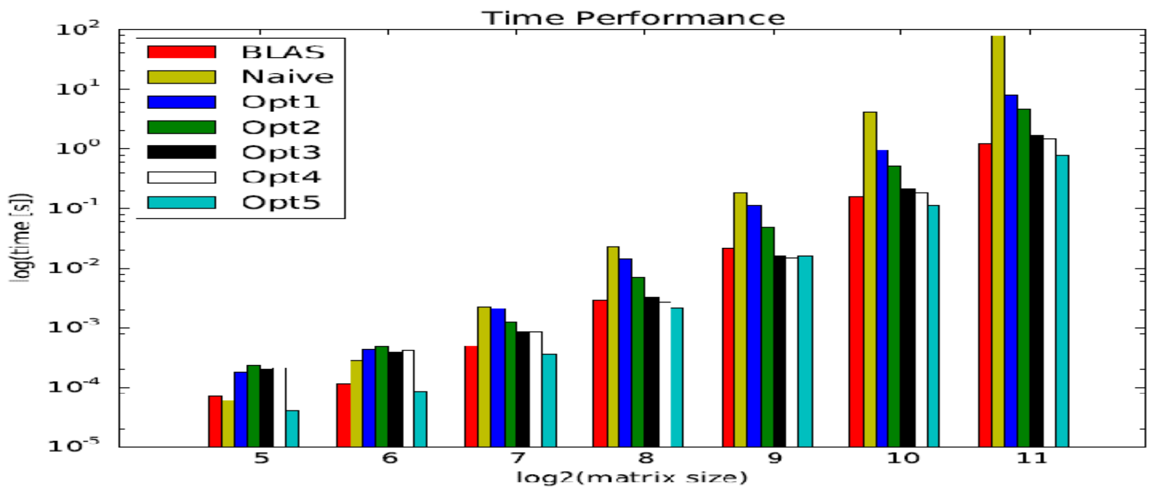
Opt 1 Result:

dim 1	din	din	time	L2Bandwidth[MBytes/s]	L2RequestRate	L2MissRate	L2MissRatio	MFlops/s	32b AVX MFlops/s	PackedMUOPS/s	ScalarMUOPS
32	32	32	0.000181351	390.851	0.0325097	0.0081986	0.252189	155.456	0	0	0
64	64	64	0.000429757	1090.72	0.0163535	0.0023232	0.142059	598.818	0	0	0
128	128	128	0.00209118	4917.15	0.0331493	0.0014316	0.043186	1116.67	0	0	0
256	256	256	0.0142631	7351.06	0.056958	0.0185848	0.326289	1922.7	0	0	0
512	512	512	0.111144	10074.8	0.0556738	0.0182398	0.327619	1740.58	0	0	0
1024	1024	1024	0.958655	7239.13	0.0739376	0.0229468	0.310354	2281.75	0	0	0
2048	2048	2048	8.17724	9050.51	0.0832589	0.0253613	0.304608	2165.75	0	0	0

Naïve Result:

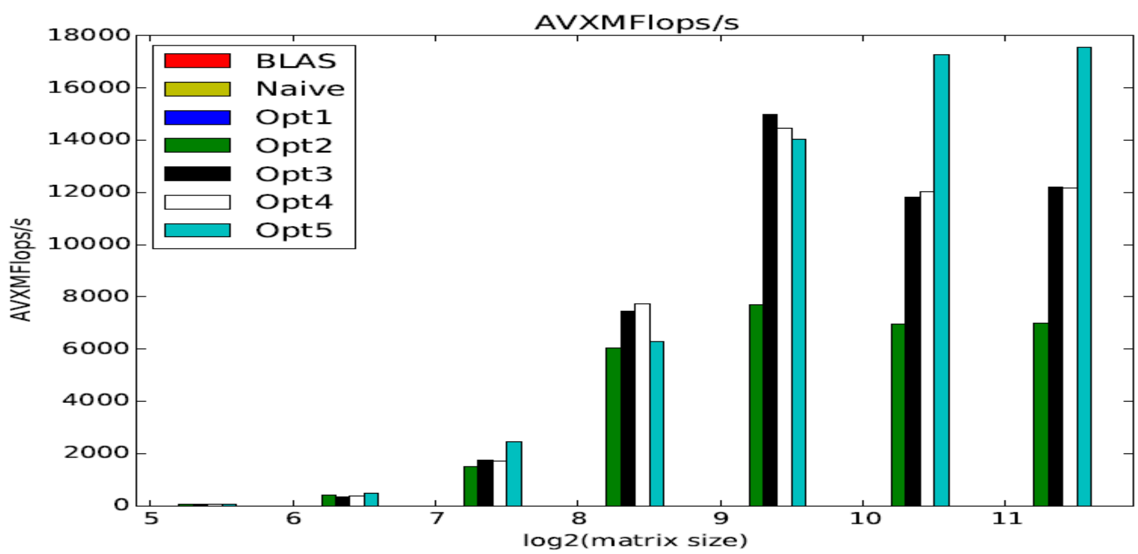
dim	dim	dim	time	L2bandwidth[MBytes/	L2RequestRate	L2MissRa	L2MissRat	MFlops	32b AVX MFlops/s	PackedMUOPS/	ScalarMUOPS/s
32	32	32	5.96E-05	686.518	0.0269837	0.00603804	0.223766	641.44	0	0	0
64	64	64	0.000277684	13407.6	0.0418062	0.00150309	0.0359539	849.564	0	0	0
128	128	128	0.00224836	27509.6	0.133442	0.00087326	0.00654411	1846.53	0	0	0
256	256	256	0.0229925	40590.2	0.224346	0.111569	0.497306	2346	0	0	0
512	512	512	0.184339	45718.5	0.223317	0.111338	0.498567	2690.98	0	0	0
1024	1024	1024	4.03793	16704.9	0.242358	0.111242	0.458997	996.826	0	0	0
2048	2048	2048	78.9123	6917.62	0.33396	0.111181	0.332916	978.597	0	0	0

Plot:Time Performance:



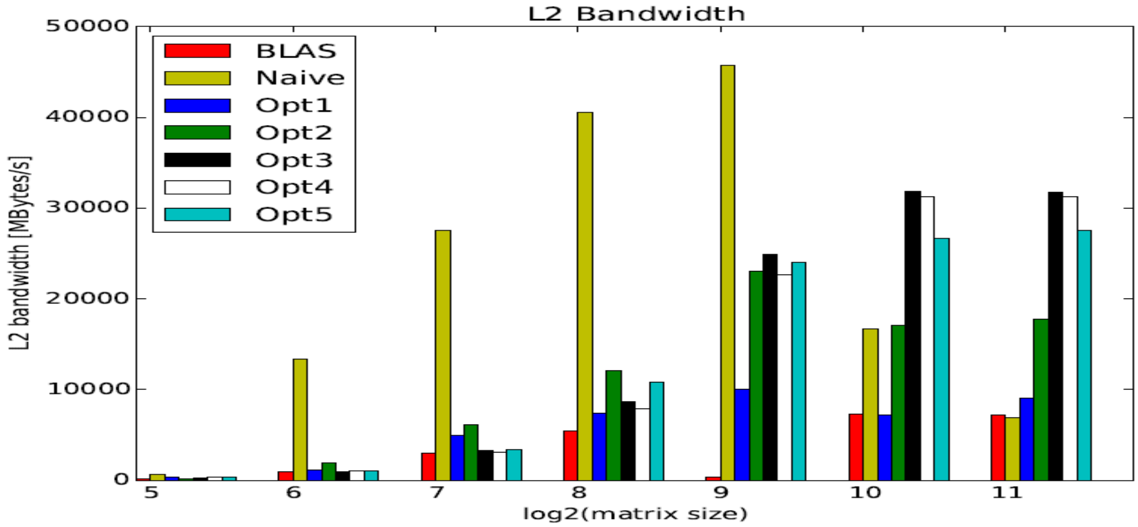
From the above diagram it is clear that for the final opt5 version, time taken is always lesser than BLAS irrespective of the matrix size.

Plot of AVXMFlops/

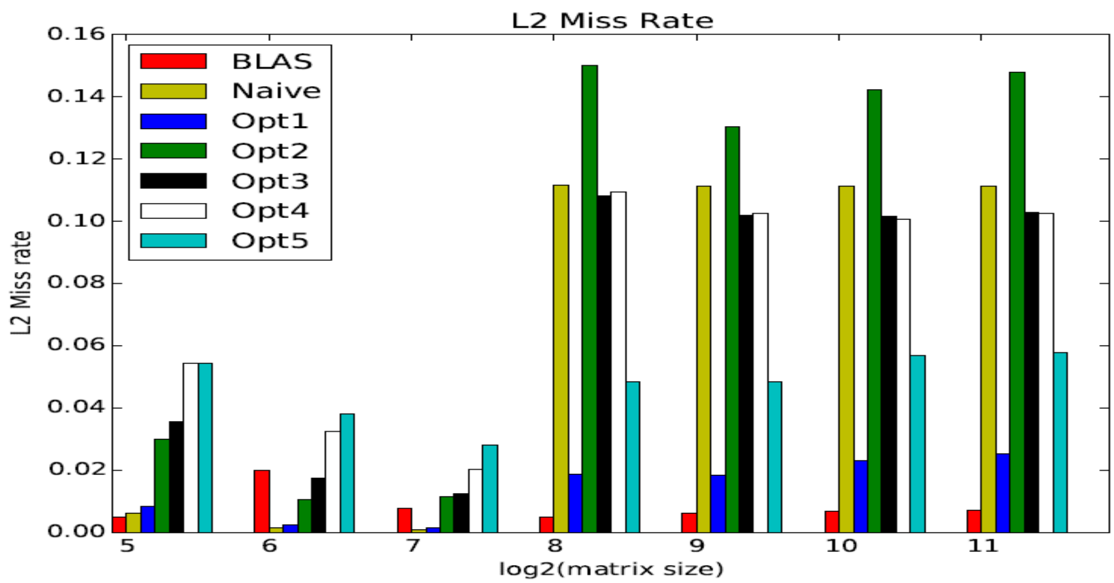


You can clearly see the introduction of AVX instructions in Opt2. BLAS seems to not use AVX instructions at all.

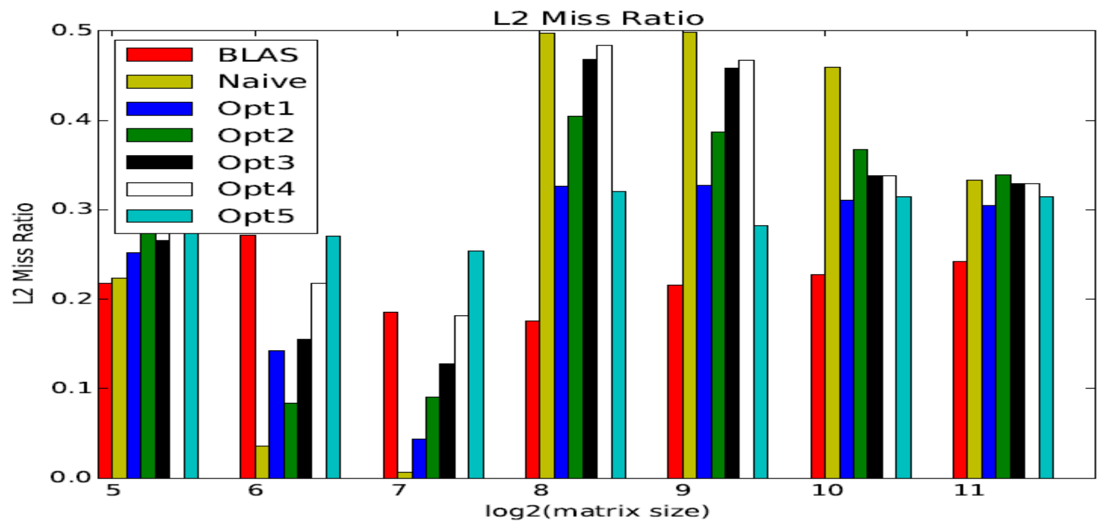
Plot: L2 Bandwidth:



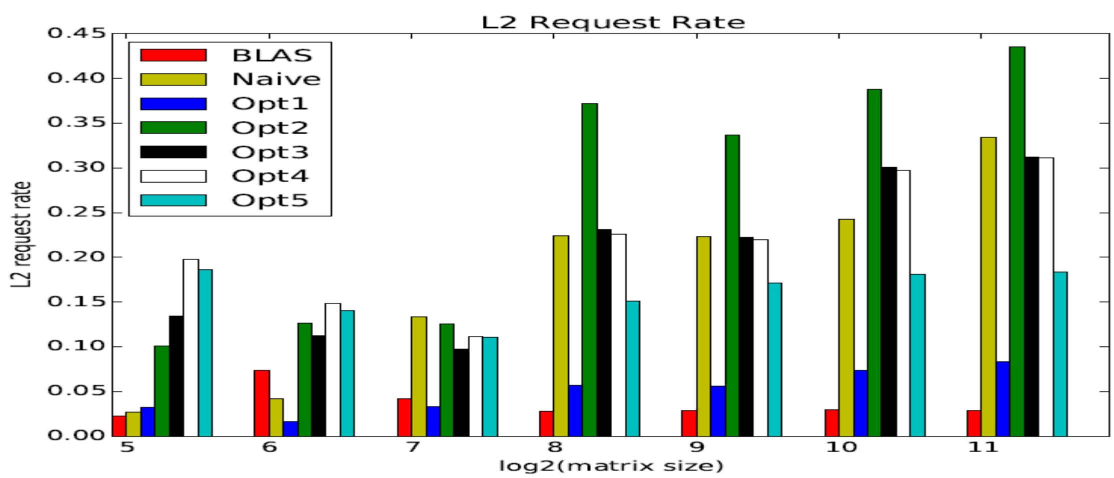
Plot: L2 Miss Rate:



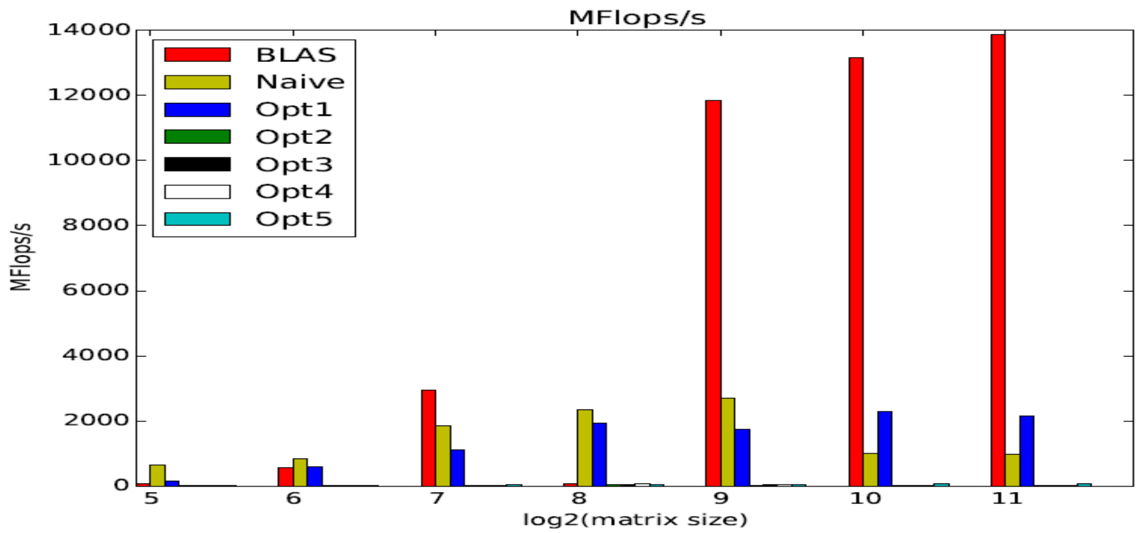
Plot: L2 Miss Ratio:



Plot: L2RequestRate:



Plot:MFlops:



Plot: PackedMUOPSs: