Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

Empirical Analysis of Cache-Oblivious Matrix Multiplication on Multicore Processor Systems

[1]Riaz Ahmed and [2]Lalit Sen Sharma

[1]Department of Computer Sciences, Govt. Degree College Poonch, J&K, India

[2]Department of Computer Sciences & IT, University of Jammu, J&K, India

E-mail: [1]riaz.mirza.11@gmail.com, [2]lalitsen@yahoo.com

ABSTRACT

In this study the performance of cache-oblivious matrix multiplication algorithm on multicore machines has been evaluated using experimental method. The evaluation has been made against the naive-iterative and cache-aware variants of matrix multiplication algorithm. The performance bottleneck in cache-oblivious matrix multiplication was identified using experimental study and a conceptual framework was designed using poly-algorithmic approach for its optimization in which recursion of cache-oblivious matrix multiplication was stopped at optimized base case and then iterative method was applied for completion of computation. The optimized cache-oblivious matrix multiplication was further analysed against iterative and cache-aware variants of matrix multiplication algorithm with varying matrix dimensions. The experimental results showed that using poly-algorithmic approach has improved the performance of algorithm. The performance of cache-oblivious algorithm remained better than traditional iterative method of matrix multiplication and remained competitive with cache-aware matrix multiplication algorithm. The study indicates that cache-oblivious approach is very beneficial for present complex multicore architecture because of its simplicity, portability and competitive performance.

Keywords: *Cache-aware, cache-misses, cache-oblivious, cilkplus. multicore, matrix multiplication*

1. INTRODUCTION

Multicore processor systems are now common across the world, even small hand held devices like mobile phones having quad- and octa-core processors are easily available in the market due to paradigm shift of almost all chip manufacturing companies towards producing chips containing multiple processors or cores [1]. The appearance of multicore processors has provided huge computational power which could only be utilized by using efficient algorithms and data structures. Therefore, the importance of efficient and optimized algorithms and their implementations has increased many folds in order to achieve the expected performance of multicore hardware by utilizing its computing resources. One of the keys to exploit the performance potential of multicore systems is the best use of cache hierarchy of multicore processors. The cache-oblivious algorithm [2] [3] was designed to avoid some of the difficulties of taking advantage of memory hierarchy of single core microprocessor machines for optimization of applications. In cache-oblivious algorithm variables are not depend upon the hardware parameters such as cache size, cache line length i.e. these variables are not required to be tuned for performance improvement. These algorithms are based on divide-and-conquer strategy in which each division step creates sub-problems of small size until it fits in some level of memory hierarchy where computation is performed without suffering cache-misses at that level and automatically adopts to all level of memory hierarchy. The cache-oblivious matrix multiplication(COMM) algorithm was originated in [4] where it was described in different frameworks and lateron extended to rectangular matrix in [2] [3] and described in cache-oblivious framework for machines having single core microprocessors and two level of cache hierarchy. In order to achieve the expected performance of multicore hardware the oblivious approach in [2] [3] need to be extended for multicore platform.

This paper evaluates the performance of cache-oblivious matrix multiplication algorithm on machines having multicore processors. The evaluation has been made in contrast to traditional iterative and cache-aware variants of matrix multiplication algorithm. A conceptual framework for optimization of cache-oblivious algorithm was designed by using poly-algorithmic approach and implemented in which matrices were recursively divided upto optimized base case and then iterative algorithm was used for completion of computation. The improved variant of cache-oblivious algorithm was further examined by performing experimental study.

The rest of the paper is organized as follows: Section 2 briefed about the related work of earlier researchers. The experimental setup and methodology adapted in this study has been explained in section 3. Section 4 presents the results and analysis of our experiments and conclusion of the study has been drawn in section 5 along with future scope of research.

2. RELATED WORK

The first study on COMM (cache-oblivious matrix multiplication) algorithm was appeared in a landmark paper of Frigo et al. [2] [3] in which authors had designed and analysed matrix multiplication algorithm in cache oblivious settings on a machine having 450 *MHZ* (megahertz) AMD processor, *32 KB* (kilobyte) *L1* cache with 32 byte cache line size. The study indicated that per-integer average time used by recursive

Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

algorithm was almost constant and took *50%* less time to execute than the time taken by iterative algorithm.

Another experimental study on recursive and iterative programs of matrix multiplication was performed by Yotov et al. [5] on three modern architectures namely *IBM* Power 5, *Sun UltraSPARC-IIIi* and *Intel Itanium-2*. The results of their study indicated that even highly optimized COMM algorithm performed poorly than cache-conscious program for the same problem. The authors pointed out that less performance of cache-oblivious algorithm was due to memory latency and due to the fact that recursive code might not exploited processor pipelines efficiently. They also studied the performance of algorithms by using microkernels. They found that iterative kernels performed better than recursive kernels. Their results showed that high performance codes require microkernels that must be optimized for *L1* instruction cache, registers and processor pipelines.

The authors in [6] evaluated the performance of parallel COMM which is based on Peano Curve [7] on shared memory multicore platforms uisng *OpenMP* concurrency platform. Algorithm was implemented using block recursive approach in which resursion was stopped on matrix blocks having size equivalent to the size of *L1* cache of underlying processor. The performance was evaluated against the well established libraries such as *GotoBLAS* and *MKL* [8] on two multicore platforms namely *Intel Xeon Server* having Intel Xeon X7350 quad-core processors and *AMD Optron Server* having *2* AMD Optron 2347 processors. The results of their study indicated that matrix multiplication based on Peano Curve showed better scalability than by *Intel's MKL* and *GotoBLAS* and. The results also pointed out that the kernel optimized according to the *L1* cache size is very beneficial for the performance improvement of COMM algorithm.

A cache-aware, adaptive and efficient multithreaded implementation of dense matrix multiplication was appeared in [9] in which authors used several techniques such as Z-*Morton* ordering layout of arrays, basic kernel operation at low-level, software data prefetching, task creation mechanism for recursion and thread pool for synchronizing shared memory writes. The performance of optimized task based matrix multiplication algorithm was compared against efficient matrix multiplication libraries such as *GotoBLAS*, *AMCL* (AMD Core Math Library) and *IMKL* (Intel Math Kernel Library) on three multicore systems namely *Egypt*, *Barcelona*, and *Clovertown*. As per the results the efficiency of their optimized task based matrix multiplication remained better than the well-established libraries and achieved *85%* of peak performance on AMD and Intel platforms.

In [10] authors extended COMM algorithm based on Peano Curve [6] for modern architectures and analysed on *4* different architectures namely *SGI Altix*, *Intel Sandy Bridge Architecture*, *AMD Bulldozer architecture* and *Intel Many Integrated Core Architecture*. Their results indicated that COMM using Peano Curve remained equally efficient against the optimized and architecture-specific libraries namely *MKL* and *AMD CML*. The authors pointed out that on future systems remarkable performance could only be achieved by using vector instructions because auto vectorization would become a crucial point.

In another study on COMM using sequential access processing [11] authors analysed the performance in terms of cache misses using *cachegrind* tool of the *valgrind* with different cache and matrix sizes. The results indicated that the recursive COMM has lesser number of cache miss ratios when compared with the naïve matrix multiplication program.

3. METHODOLOGY

The present study comprised of seven experiments performed on two fundamentally different machines. Initially five experiments were performed to determine the performance bottlenecks in COMM algorithm. After the analysis of first five experiments a conceptual framework was designed for improvement of COMM using poly-algorithmic approach in which recursion of COMM was stopped at optimized base case and then applied iterative algorithm to complete the computation. Two more experiments were performed in which performance of improved COMM was compared with cache-aware and naïve-iterative variants of matrix multiplication algorithm. All experiments were performed on two multicore machines with different specifications as shown in Table 1:

Table 1: Specifications of Experimental Machines

	Machine-1	**Machine-2**
Model	Intel Core i-3-240M	Intel Core i-5-240M
CPU	2.50 GHz	3.40 GHz
No. of Cores	2	4
No. of Threads	4	8
L1 Cache	32 KB	256 KB
L2 Cache	256 KB	1024 KB
L2 Cache	3072 KB	8192 KB
Memory	4 GB	4 GB
O.S.	Linux 64-bit	Linux 64-bit

3.1 Concurrency Platform: *Intel-Cilk-Plus* concurrency platform was used for implementation of all algorithms. *Intel Cilk Plus* [12] [13] is a language extension to *C* and *C++* languages and was developed to ease the programming efforts for shared memory multiprocessor systems. The parallelism in the programs has been expressed using three keywords of *Intel Cilk plus* namely *cilk_spawn, cilk_sync and cilk_for* which explain the logical structure of parallel program and the runtime system of *Intel Cilk Plus* schedules the parallel work dynamically onto the available cores. A parallel loop has been expressed by *cilk_for* keyword which allows the iteration of the *for* loop body to run in parallel.

3.2. Performance Metrics: The metrics of latency, GFlops, speedup, efficiency and cache miss-ratio were

Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

evaluated using experiments. The *gettimeofday* command of *C* language was used to measure the execution time of algorithms. In scientific computation FLOPS (floating point operations) has been used to measure the performance of applications on computer and the more accurate metric used in modern computers is GFLOPS:

$$GFlops = \frac{n^2}{time * 2^2} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

The speedup is calculated by dividing Serial Execution Time by Parallel Execution Time:

$$SU = \frac{T1}{Tn} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

The efficiency is calculated by dividing speedup by number of processors:

$$E = \frac{SU}{Pn} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3)$$

Perfsuit tool also known as the performance counter for Linux was used to evaluate the hardware events like cache misses by entering a ***perf stat*** command. The *L1 cache miss ratio* was measured as below:

$$miss - ratio = \frac{L1-dcache-loads}{\text{L1-dcache-load-misses}} \ldots\ldots\ldots\ldots (4)$$

3.3. **Data-Set used for Experimentation:** The double precision (*8* bytes), randomly generated data set has been used in all experiments. The random values of matrices were generated using *srand* function of *C* library. The matrices were stored in row-major order. Three types of matrix dimensions were used for experimentation namely; the dimensions as exact power of *2* (*n*= 8, 16, 32, 64, 128, 256, 512, 1024), matrices with not power of *2* but multiple of *100* (*n* = 100, 300, 600, 900) and matrix with length of matric row is multiple of cache line length (384, 640, 896, 1152).

4. RESULTS AND ANALYSIS

The experimental results along with discussions has been presented in this section.

Experiment-1: The first experiment was performed for two variants of matrix multiplication viz. naïve-iterative and cache-oblivious (CO) on a dual-core machine with matrix dimensions exact power of *2*. The algorithms were executed sequentially as well as in parallel and execution time was recorded as performance metric and shown in Table 2. It has been observed that for small matrix sizes up to *128* both algorithms with sequential access performed better and thereafter the performance of parallel algorithms remained better than their sequential variants. It has also been observed that for small matrix dimensions naïve matrix multiplication algorithm outperformed all implementations. For matrix size *32* and above cache oblivious matrix multiplication outperformed the naïve matrix multiplication algorithm. The results were further analysed by calculating their means and their computation times as shown in *Fig 1(a)* and *Fig 1(b)* respectively. The graphs in *Fig 1(a)* and *Fig 1(b)* indicate that parallel COMM algorithm took least amount of time among all the algorithms whereas sequential naïve algorithm remained the most expensive one. The mean results were further analyzed by evaluating the speedup and efficiency. The results show that parallel COMM and parallel naïve algorithms remained *91%* and *59%* more efficient than their sequential variants and the parallel COMM algorithm remained *120%* more efficient when compared with the parallel naïve MM algorithm.

Table 2: Execution time (in sec) of MM Algorithms with matrix dimensions power of 2 on a dual-core machine

Matrix Size (n)	Naive		Cache Oblivious	
	Sequential	Parallel	Sequential	Parallel
16	0.00016	0.00073	0.00017	0.00018
32	0.00109	0.01038	0.00050	0.00066
64	0.00771	0.02214	0.00279	0.00303
128	0.02798	0.03207	0.00930	0.00519
256	0.15906	0.11708	0.07015	0.03965
512	0.81980	0.67428	0.57049	0.30804

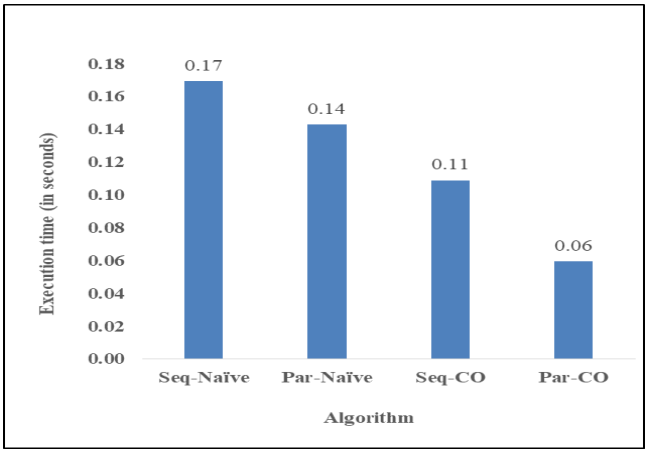

Fig 1(a)

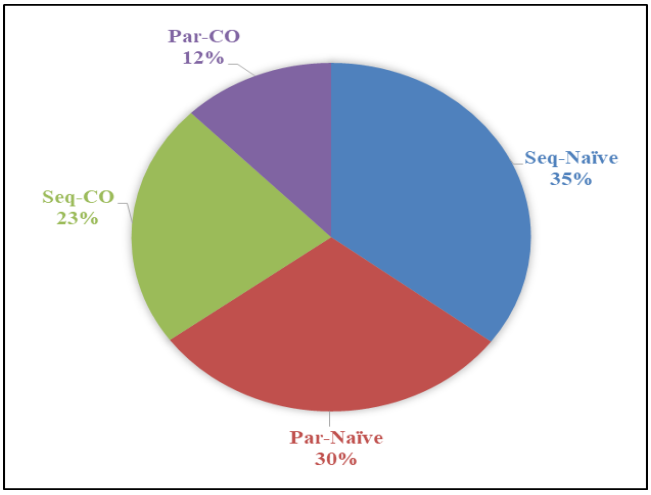

Fig 1(b)

Fig 1(a) shows mean execution time, **Fig 1(b)** shows percentage of computation time, of MM Algorithms with matrix dimensions exact power of 2 on a dual-core machine

Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

Experiment-2: The second experiment was also performed for both naïve and COMM algorithms on a dual-core machine but the input matrix dimensions were multiple of *100*. The algorithms were executed sequentially as well as in parallel and execution time was recorded as performance metric as shown in Table 3.

Table 3: Execution time (in seconds) of MM Algorithms with matrix dimensions multiple of *100* on a dual-core machine

Matrix Size (n)	Naive		Cache Oblivious	
	Sequential	Parallel	Sequential	Parallel
100	0.00544	0.00532	0.00626	0.00471
300	0.11652	0.06705	0.12657	0.10334
600	1.84589	1.04648	1.00553	0.76168
900	7.76429	4.34658	3.30237	2.04493
1200	18.68961	11.45708	8.11196	6.71168

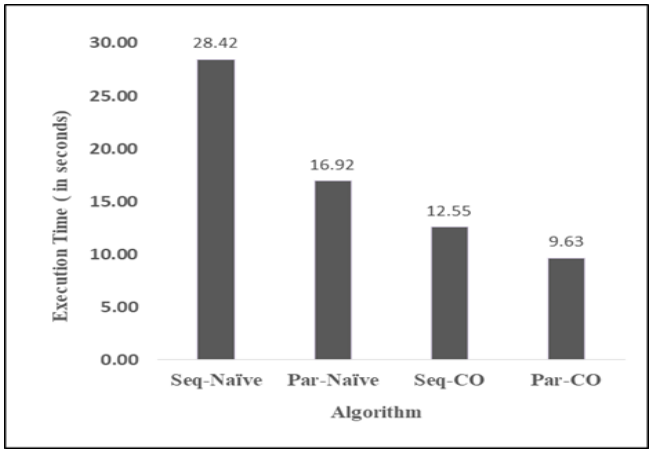

Fig 2(a)

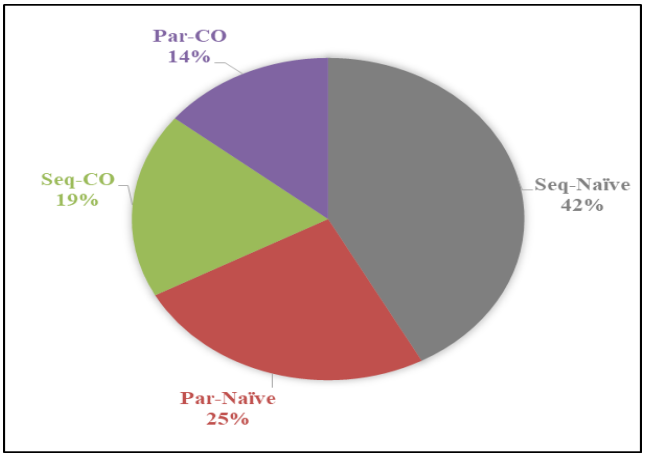

Fig 2(b)

Fig. 2(a) shows mean execution time, **Fig. 2(b)** shows percentage of computation time, of MM Algorithms with matrix dimensions multiple of *100* on a dual-core machine

The results indicate that parallel variants of both algorithms outperformed their sequential variants for all matrix dimensions. For small matrix sizes the performance of sequential naïve was better than sequential COMM and thereafter COMM remained best. The performance of parallel cache oblivious algorithm remained better in all cases except for matrix size *300* where as naïve variant performed better.

The mean of all results has been calculated and shown in Fig 2(a) and the percentage of computation time is calculated and shown in Fig 2(b). The results in Fig.2 showed that parallel cache-oblivious algorithm took least amount of time among all the algorithms whereas sequential naïve algorithm remained the most expensive one.

The mean results were further analyzed by evaluating the speedup and efficiency. The results showed that parallel cache-oblivious and naïve algorithms remained *65%* and *83%* more efficient than their sequential variants respectively and the cache-oblivious algorithm remained *88%* more efficient when compared with the naïve algorithm.

Experiment-3: The third experiment was performed on a quad-core machine for two variants of matrix multiplication viz. naïve and cache oblivious. Both sequential and parallel variants were executed with matrix dimensions multiple of *100* and results have been shown in *Table 4.*

Table 4: Execution time (in seconds) of Matrix Multiplication Algorithms with matrix dimensions multiple of *100* on a quad-core machine.

Matrix Size (n)	Naive		Cache Oblivious	
	Sequential	Parallel	Sequential	Parallel
100	0.004353	0.003876	0.004886	0.006785
300	0.04625	0.02508	0.086255	0.039031
600	0.704802	0.316927	0.690731	0.310163
900	3.246513	1.255017	2.272097	0.779073
1200	15.23976	3.855156	5.582139	2.422053

It has been observed that algorithms have similar behavior on quad-core machine as was observed on dual-core. The results indicated that the parallel variants of both algorithms outperformed their sequential variants for all matrix dimensions on quad-core machine. For small matrix sequential naïve was better than sequential cache oblivious. Parallel cache oblivious algorithm remained better in all cases except for matrix size *100* & *300* where naïve performed better. The mean

Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

of all results has been calculated and shown in Fig 3(a) and the percentage of computation time is calculated and shown in Fig 3(b). The results show that parallel cache oblivious algorithm took least amount of time among all the algorithms whereas sequential naïve algorithm remained the most expensive one. The mean results were further analyzed by evaluating the speedup and efficiency. The results show that parallel cache-oblivious and naïve algorithms remained 88% and 60% more efficient than their sequential variants respectively and the cache-oblivious algorithm remained 38% more efficient when compared with the naïve algorithm.

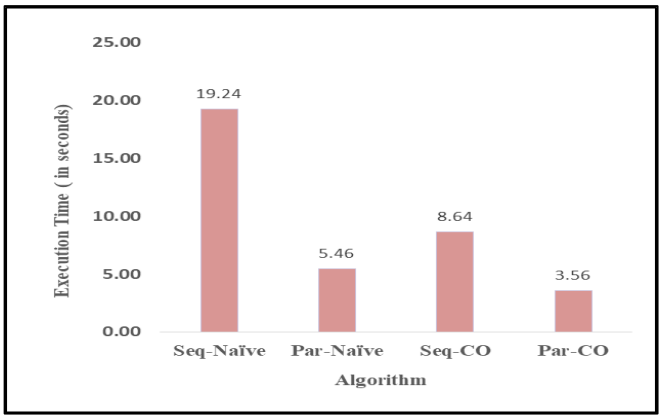

Fig 3(a)

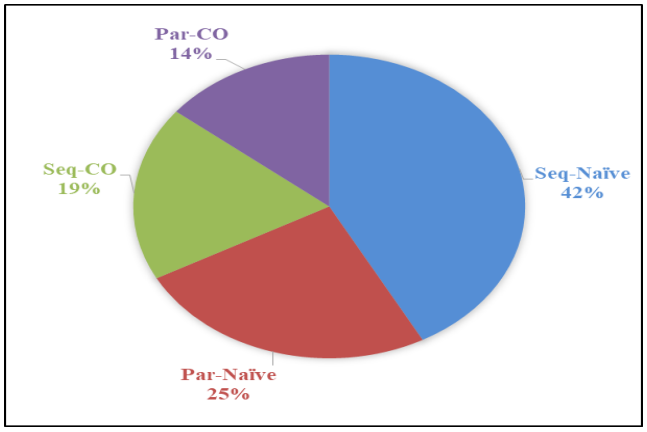

Fig 3(b)

Fig 3(a) shows mean execution time, **Fig 3(b)** shows percentage of computation time of MM Algorithms with matrix dimensions multiple of 100 on a quad-core machine

Experiment-4: The fourth experiment was performed on a dual-core machine by evaluating GFlops/sec. This experiment was performed for three variants of matrix multiplication namely naïve, cache oblivious (CO) and cache aware (CA) with matrix dimensions exact power of 2. All algorithms were executed both sequentially and parallel and results have been shown in Fig 4(a-c). The results indicate that sequential variants of all algorithms outperformed their parallel counterparts for small data set. For matrix dimensions up to *16* there was abrupt increase in the performance of sequential algorithms and then almost linear. The parallel variants outperformed for matrix sizes *128* and above.

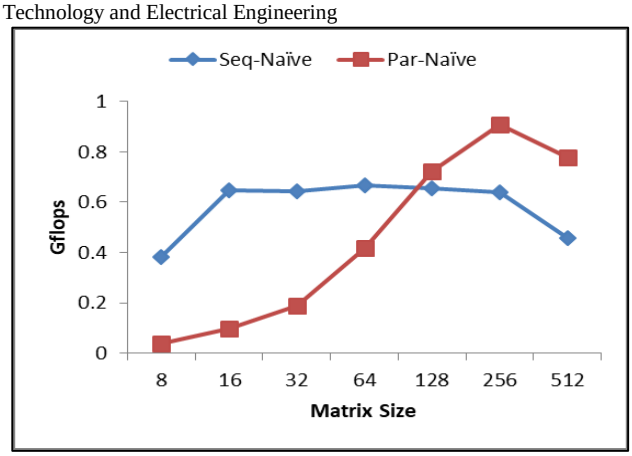

Fig 4(a)

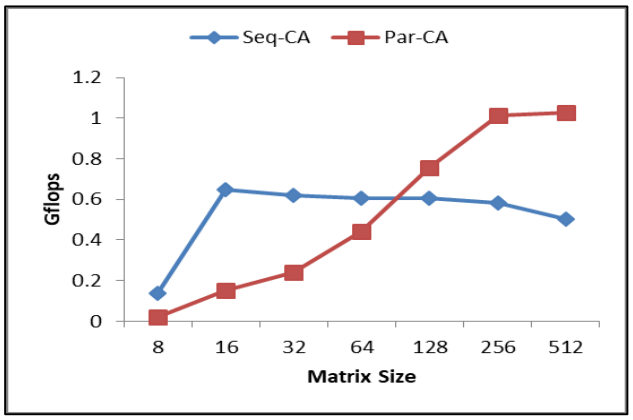

Fig 4(b)

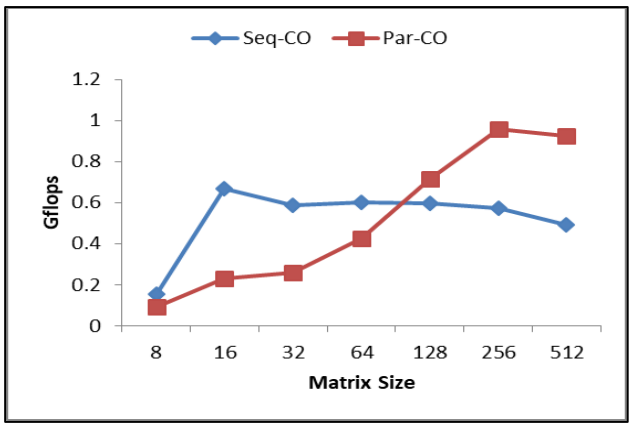

Fig 4(c)

Fig 4 shows **c**omparative Performance (in GFLOPS) of **a)** Naïve, **b)** Cache-Aware and **c)** COMM Algorithms with matrix dimensions power of 2 on a dual-core machine

Experiment-5: The performance of matrix multiplication algorithms was further analysed on both dual-core and quad-core machines by measuring their cache misses in fifth experiment. The ratios of cache misses were evaluated and shown in Table 5 and Table 6 respectively. It has been observed that for small matrix sizes sequential naïve algorithm suffered with small cache penalties but when the size of matrix grows it suffered from high cache penalty. The cache oblivious algorithm showed different behaviour as compared to naïve

Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

algorithm. For small matrix sizes cache-oblivious matrix multiplication showed high ratio of cache misses whereas for large matrix sizes it showed lesser number of cache misses. Less cache-miss ratio of cache-oblivious algorithm for large matrix dimensions indicate that cache oblivious algorithm used cache memory efficiently.

Further the impact of cache misses on the performance of algorithm has been evaluated by calculating the correlation between cache misses and performance of an algorithm. The calculated value of *r (correlation coefficient)* was *0.63*, which shows strong positive correlation between algorithmic execution time and cache-miss ratio. It has been observed that the cache miss ratio of algorithms was directly proportional to their execution time as the performance of algorithms increased with the decrease of cache misses. So, the rate of cache misses has tremendous impact on the performance of algorithm with sufficiently large matrix sizes. The performance of cache oblivious algorithm remained better for large matrix sizes because of lesser number of cache misses.

Table 5: Cache Miss Ratio of Matrix Multiplication Algorithms with matrix dimensions multiple of *100* on a quad-core machine

Matrix size	On dual-core machine			
	Naïve		Cache Oblivious	
	Sequential	Parallel	Sequential	Parallel
100	0.44	1.1	0.67	1.13
300	1.7	0.36	1.66	0.93
600	10.01	0.37	1.49	0.77
900	10.58	0.26	1.81	0.52

Table 6: Cache Miss Ratio of Matrix Multiplication Algorithms with matrix dimensions multiple of *100* on a quad-core machine

Matrix size	On quad-core machine			
	Naïve		Cache Oblivious	
	Sequential	Parallel	Sequential	Parallel
100	3.83	2.15	1.38	2.08
300	1.32	0.38	2.77	1.73
600	8.55	0.34	0.86	0.84
900	9.65	0.28	0.56	0.56

Inferences (Experiment 1 to 5): The following observations has been drawn from above five experiments:

- The results indicated that the COMM faced lesser number of cache penalties for large matrix sizes and thus remained better than naïve iterative matrix multiplication algorithm. The results showed that the cache penalties had significant impact on the performance of algorithms for large matrix sizes. For small matrix sizes the difference in cache miss ratio of cache-oblivious matrix multiplication and naïve iterative matrix multiplication was very small which indicated that cache penalties have negligible impact on the performance of algorithms for small matrix sizes. Therefore it was observed that using cache-oblivious approach for multiplication of large matrix sizes could be advantageous.

- It has been seen in the results that the performance of cache-oblivious matrix multiplication did not scale to the performance of naïve-iterative matrix multiplication for small matrix sizes. The less performance of COMM was due to overhead caused by thread management and overhead due to excessive stack operations caused by large number of recursive calls made by COMM algorithm. Therefore, using iterative approach for multiplication of small matrix sizes could be more beneficial.

- It has also seen that the sequential algorithms for naïve iterative, cache-aware and cache-oblivious matrix multiplication outperformed their parallel variants for small matrix sizes. The less performance of parallel variants was due to the overhead caused by thread management. Therefore, using sequential algorithms for small matrix sizes could be more advantageous.

The careful analysis of above inferences indicated that there was no single solution for designing optimized algorithm for all matrix dimensions and on different platforms. That reason compelled for adapting a poly- algorithmic approach for implementation of cache-oblivious matrix multiplication so that it could performed well for all matrix sizes and on different machines. In that approach the recursion of COMM algorithm was stopped at base case of *16* and then naïve-iterative routine was used to complete the matrix multiplication so that the issues of overhead caused by thread management and the overhead due to excessive stack operations caused by large number of recursive calls made by cache-oblivious matrix multiplication algorithm could be addressed. The performance of improved variant of COMM was further analysed against naïve and cache-aware variants on both the experimental machines and results has been shown below in experiment 6 and 7.

Experiment-6: Two sub-set of experiments were performed for naïve, cache-oblivious and cache-aware matrix multiplication (CAMM) on a dual-core machine. First set of experiments was performed with matrices having row length exact power of *2* and results has been shown in *Fig 5(a & b)*. The second set of experiments was performed with matrices having row length multiple of cache line length and results has been shown in *Fig 6(a & b)*.

Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

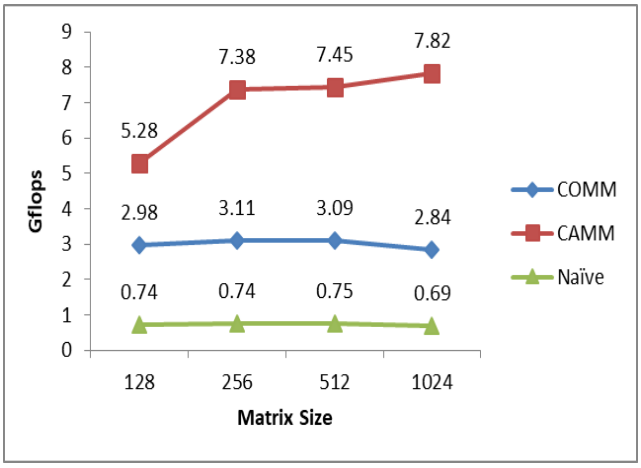

Fig 5(a)

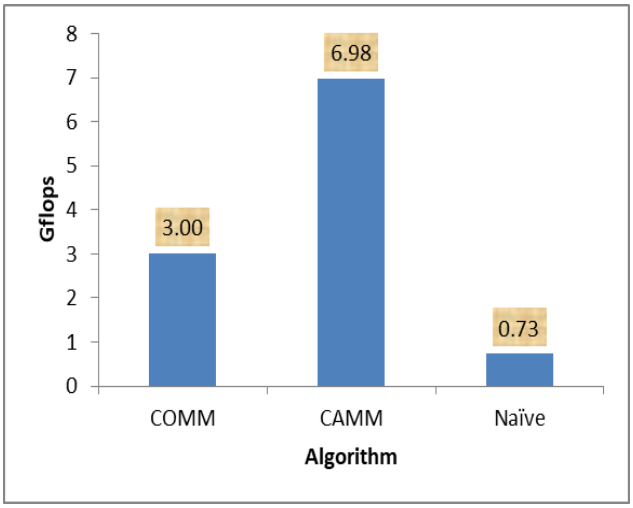

Fig 5(b)

Fig 5 (a & b): Comparative Performance (in GFLOPS) of Naïve, Cache-Aware and Cache-Oblivious Matrix Multiplication Algorithms with matrix having row length power of 2 on a dual-core machine

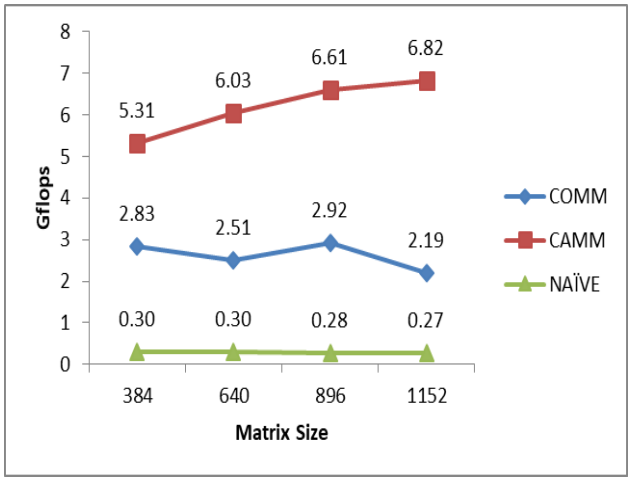

Fig 6(a)

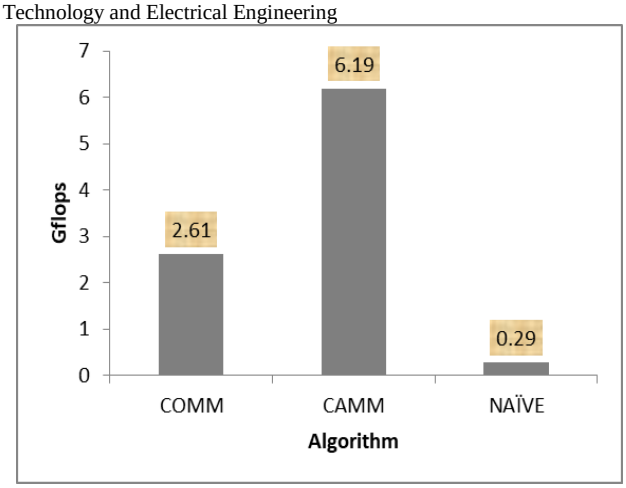

Fig 6(b)

Fig 6(a & b): Comparative Performance (in GFLOPS) of Naïve, Cache-Aware and COMM Algorithms with matrix having row length multiple of cache line length on a dual-core machine

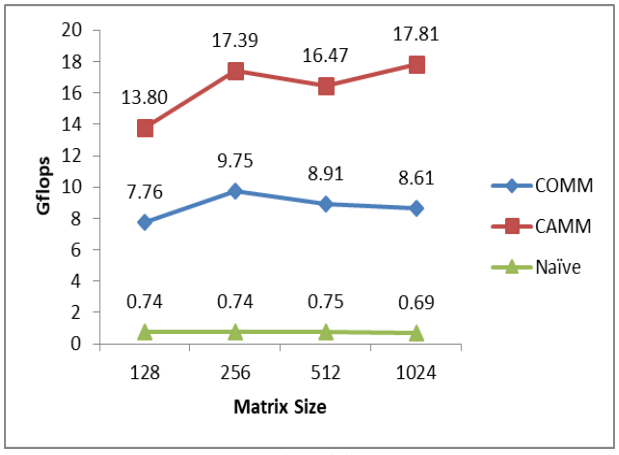

Fig 7(a)

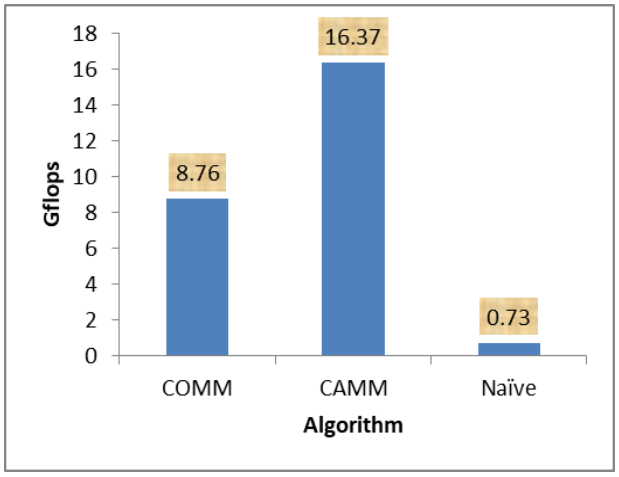

Fig 7(b)

Fig 7(a & b): Comparative Performance (in GFLOPS) of Naïve, Cache-Aware and Cache-Oblivious Matrix Multiplication Algorithms with matrix having row length power of 2 on a quad-core machine

Volume 10, Issue 1
February 2021

ITEE Journal
Information Technology & Electrical Engineering

ISSN: - 2306-708X

Experiment-7: Two sub-set of experiments were performed for naïve, cache-aware and cache-oblivious matrix multiplication on a quad-core machine. First set of experiments was performed with matrices having row length exact power of *2* and results has been shown in *Fig 7(a & b).* The second set of experiments was performed with matrices having row length multiple of cache line length and results has been shown in *Fig 8(a & b).*

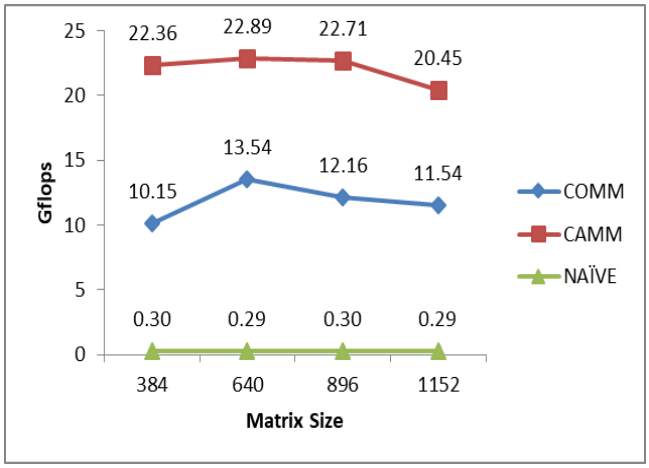

Fig 8(a)

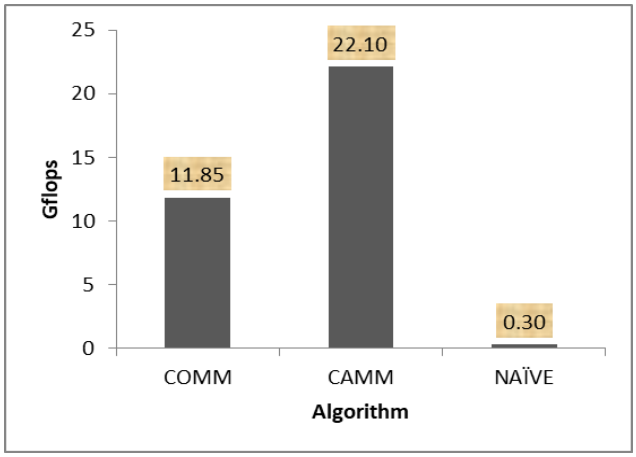

Fig 8(b)

Fig 8(a & b): Comparative Performance (in GFLOPS) of Naïve, Cache-Aware and Cache-Oblivious Matrix Multiplication Algorithms with matrix having row length multiple of cache line length on a quad-core machine

Observations (Experiment 6 & 7):

- It has been observed that performance of cache-aware matrix multiplication remained best on both the experimental machines.
- The cache-oblivious matrix multiplication performed better than naïve algorithm and remained at top two best performing algorithms.
- Although using poly-algorithmic approach improved the performance of COMM but it did not performed better than the cache aware.
- The mean results showed that cache-oblivious algorithm remained *23%* more efficient than naïve algorithm but remained *38%* less efficient when compared with cache aware algorithm on dual-core machine.
- On a quad-core machine cache-oblivious algorithm remained *49%* more efficient than naïve algorithm but remained *45%* less efficient when compared with cache aware algorithm.

5. CONCLUSION

In the present research work we have evaluated the performance of cache-oblivious matrix multiplication on multicore platforms. Initially, performance bottleneck in cache-oblivious matrix multiplication was identified using experimental study and then a conceptual framework was designed for its optimization using poly-algorithmic approach. In that approach recursive divide-and-conquer algorithm was used as overall approach and iterative algorithm at low level as microkernel.

The optimized cache-oblivious matrix multiplication was further analysed against iterative and cache-aware variants of matrix multiplication with varying matrix dimensions. The experimental results showed that using poly-algorithmic approach has improved the performance of algorithm. The performance of cache-oblivious algorithm remained better than traditional iterative method of matrix multiplication and remained competitive with cache-aware matrix multiplication algorithm.

The study indicates that cache-oblivious approach is very beneficial for present complex multicore architecture because of its simplicity, portability and competitive performance.

FUTURE SCOPE: The present research work can be extended by performing analysis on many core platforms. Further the microkernel used at low level could be optimized using vector instructions or transposition which can improve the performance of cache-oblivious matrix multiplication.

REFERENCES

[1] G. Blelloch, R. A. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proceedings of nineteenth annual ACM symposium on Discrete algorithms*, San Francisco, California, 2008.

[2] M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science* , 1999.

[3] M. Frigo, M. Leiserson, H. Prokop and S. Ramachandran, "Cache-Oblivious Algorithms," *ACM Transactions on Algorithms*, vol. 8, no. 1, pp. 1-22, January 2012.

[4] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson and K. H. Randall, "An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms," in *Procedings of Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2004)*, Padus, Italy, 1996.

[5] K. Yotov, T. Roeder, K. Pingali, J. Gunnels and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proceedings if nineteenth annual ACM symposium on Parallel Architecture*, San Diego, Calofornia, USA: ACM New York, NY, USA, 2007.

[6] A. Heinecke and M. Bader, "Parallel matrix multiplication based on space-fitting curves on shared memory multicore platform," in *Proceedings of the 2008 workshop on Memory access on furure processors: a solved problem ?*, Ischia, Italy, 2008.

[7] M. Bader and C. Zenger, "Cache oblivious matrix multiplication using an element ordering based on a Peano Curve," *Linear Algebra and its Applications*, vol. 417, no. 2-3, pp. 301-313, 2006.

[8] Intel Corporation, 2007. [Online]. Available: http://intel.com/cd/software/products/asmo-na/eng/perflib/mkl/. [Accessed 5 June 2017].

[9] G. Runger and M. Schwind, "Fast recursive matrix multiplication for multi-core architectures," *Procedia Computer Science: International Conference on Computational Science, ICCS 2010*, vol. 1, no. 1, p. 67–76, 2010.

[10] A. Heinecke and C. Trinitis, "Cache-oblivious matrix algorithms in the age of multi- and many-cores," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 9, 20 December 2012.

[11] C. S. Kumar and B. S. Pattnaik, "Miss rate analysis of cache oblivious matrix multiplication using sequential access recursive algorithm and normal multiplication algorithm," in *2013 International Conference on Emerging Trends in Communication, Control, Signal Processing and Computing Applications*, Bangalore, India, 2013.

[12] "cilk-plus-tutorial," [Online]. [Accessed 05 January 2016].

[13] Intel Corporation, "intel Cilkplus Tutorial".

AUTHOR PROFILES

Riaz Ahmed received the Master of Computer Applications from Indira Gandhi National Open University, New Delhi in 2003. He is a research student of University of Jammu. Currently, he is an Assistant Professor (selection grade) at Govt. Degree College Poonch, J&K, India.

Lalit Sen Sharma MCA, M. Sc. (Mathematics), B,Sc. Ph. D. (Computer Science) is a professor at University of Jammu.