Ain Shams Engineering Journal 11 (2020) 1179–1190

Contents lists available at ScienceDirect

Ain Shams Engineering Journal

journal homepage: www.sciencedirect.com

Electrical Engineering

Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor

A.M. Hemeida [a,*], S.A. Hassan [b], Salem Alkhalaf [c], M.M.M. Mahmoud [b], M.A. Saber [b], Ayman M. Bahaa Eldin [d], Tomonobu Senjyu [e], Abdullah H. Alayed [f]

[a] Department of Electrical Engineering, Faculty of Energy Engineering, Aswan University, Aswan 81528, Egypt
[b] Department of Electrical Engineering, Faculty of Engineering, Aswan University, Aswan 81542, Egypt
[c] Department of Computer Science, Alrass College of Science and Arts, Qassim University, Qassim, Saudi Arabia
[d] Department of Computer Engineering, Faculty of Engineering, Ain Shams University, 11517 Cairo, Egypt
[e] Department of Electrical and Electronics Engineering, Faculty of Engineering, University of the Ryukyus, Japan
[f] Faculty of Education, Qassim University, Qassim, Saudi Arabia

ARTICLE INFO

Article history:
Received 24 July 2019
Revised 4 January 2020
Accepted 6 January 2020
Available online 30 January 2020

Keywords:
Intel's AVX
Intel MKL SGEMM
Matrix-matrix multiplications
Optimization
Multicore

ABSTRACT

This paper is focused on Intel Advanced Vector Extension (AVX) which has been borne of the modern developments in AMD processors and Intel itself. Said prescript processes a chunk of data both individually and altogether. AVX is supporting variety of applications such as image processing. Our goal is to accelerate and optimize square single-precision matrix multiplication from 2080 to 4512, i.e. big size ranges. Our optimization is designed by using AVX instruction sets, OpenMP parallelization, and memory access optimization to overcome bandwidth limitations. This paper is different from other papers by concentrating on several main technique and the results therein. Making parallel implementation guidelines of said algorithms, where the target architecture's characteristics need to be taken into consideration when said algorithms are applied are presented. This work has a comparative study of using most popular compilers: Intel C++ compiler 17.0 over Microsoft Visual Studio C++ compiler 2015. Additionally, a comparative study between single-core and multicore platforms has been examined. The obtained results of the proposed optimized algorithms are achieved a performance improvement of 71%, 59%, and 56% for C = A.B, C = A.B^T, and C = A^T.B separately compared with results that are achieved by implementing the latest Intel Math Kernel Library 2017 SGEMV subroutines.

1. Introduction

Intel's AVX is a type of parallel processing wherein single instruction is applied to several data streams simultaneously or in parallel. It has been created to help simplify efficacy application through beamy spectrum of different range of line parallelism in terms of software structures, alongside data of vector lengths. Intel's AVX upholds 256-bit wide vectors and has 8 SIMD registers (YMM0-YMM7) in 32-bit operating mode or 16 registers (YMM0-YMM15) in 64-bit mode. In addition, Intel's AVX prescripts spread the former SIMD characteristics by modifying the features and prescript [1,2].

Dense matrix-matrix multiplication is applying in different areas such as scientific, numerical problems, physical phenomena, and other fields. Therefore, it is necessary to be optimized and accelerated. Basic Linear Algebra Subprograms (BLAS) is a gauge implementation programming moderator to carry out the matrix and vector operations such as multiplication.

Nowadays, we have different fast rendering libraries, which can use for dense matrix-matrix multiplication, but that applications are not as well as the basics of the straightforward matrix-matrix multiplication such as ATLAS [3,4], LAPACK [5], and PHiPAC [6], and the improvement algorithms were applied in these packages are totally varied from those implemented in this study, and they were not using AVX prescript for their applications.

* Corresponding author at: Faculty of Energy Engineering, Aswan University, 81528 Aswan, Egypt.
E-mail address: ashraf@aswu.edu.eg (A.M. Hemeida).
Peer review under responsibility of Ain Shams University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.asej.2020.01.003

Nomenclature

List of symbols

A	the (n × n) matrix
A^T	the transposed matrix of A
acc	the scalar accumulator
B	the (n × n) matrix
BS	the Block Size in blocking
C	the (n × n) matrix
op (A)	the (n × n) matrix is given by A or A^T
op (B)	the (n × n) matrix is given by B or B^T

Historically, LAPACK was unchained as initiating a building structure of the first BLAS [5]. LAPACK is a rendering ambulant library that was implemented for dense linear algebra. BLAS libraries were tuned by many hardware vendors on their different own hardware architecture such as Intel MKL and AMD CML for the NVIDIA GPUs [12,13].

1.1. Related work

All hardware vendors have optimized BLAS code especially to the underlying hardware to get the best performance. Intel's AVX instructions have been already used in previous works to speed up matrix-matrix multiplications.

In [7], matrix-matrix multiplication (DGEMM) in double-precision has been tuned and optimized by implementing AVX prescript on Intel Xeon Phi coprocessor. An improvement of 33% performance is obtained when comparing the proposed algorithm to that existed with the Intel Math Kernel Library (MKL) 11.0 and is utilized the Intel Composer XE 2013 compiler. The mentioned pervious work was implemented and is focused only on one algorithm C = A.B.

Hadizadeh et al. proposed a new architecture based on the MIPS processor was provided. That new parallel processing architecture permitted parallel implementation of instructions by presenting new concepts for the issuance of commands in parallel such as clever discovery of restricted jumps and memory organization [8].

In [9], matrix-matrix multiplication was implemented depended on blocking, data prefetching, loop unrolling, and the Intel AVX-512 for optimizing matrix multiplication. A single core of the Knights Landing (KNL) was used and achieved up to 98% of SGEMM and 99% of DGEMM using the Intel MKL.

1.2. General review of Intel's AVX

Intel's AVX is 256-bit prescript sets spreads to Intel Streaming SIMD Extensions (SSE) and is developed for the application of intensive floating-point [11]. Intel's AVX is a good model of a single instruction multiple data streams and provides a good performance to wider 256-bit vectors. The prescript upholding different types of operands which can be implementing to raise the efficiency of the prescript programming resilience, restful memory bias for processes and permit for non-wasteful exporter code [12,13]. AVX prescripts has the ability to support many processes, such as cryptography, functions, convert, comparison, and other prescripts. These prescripts run on eight elements (32-bit) floating-point data or four (64-bit) floating-point data. Intel's AVX adds 16 registers (YMM0-YMM15), each of 256-bits wide which aliased over 128-bit (XMM0-XMM15) old registers [14,15]. Fig. 1 shows a comparison between scalar and SIMD summing operations when applying Intel AVX. Intel's AVX instructions work on eight elements single-precision floating-point data or four double precision floating-point data while scalar instructions work on one element.

Fig. 2 presents SIMD 256-bit wide register."

Although the simplicity of the sequential algorithms of matrix multiplication, these algorithms suffer from poor locality, poor performance and low efficiency (i.e. time spent for doing computations against data) due to bandwidth limitations. These problems are solved by using optimization techniques such as blocking, prefetching, and unrolling.

Our study presents a novelty of adjusting single-precision matrix-matrix multiplication (SGEMM) on Intel Core i7 platform in a Broadwell framework. Implementation using square matrices, for a huge sizes, by implementing the AVX prescripts, memory access optimization, low-level code optimization and parallelization optimization.

In this work, we create the optimization techniques manually to achieve significant high performance of the overall algorithm while Intel's advanced vector extensions provide powerful improvement for memory access and data computing. the performance of the proposed optimized algorithms of single-precision dense matrix-matrix multiplications is evaluated in this study. The comparison study between the performance of the proposed algorithms and the latest Intel Math Kernel Library 2017 SGEMM subroutines is introduced [10]. The latest version of Intel Math Kernel Library subroutines presents newer version of Intel MKL SGEMM subroutine which implements modern benchmark extensions and Intel AVX instructions widely. The obtained results of the proposed optimized algorithms are compared with Intel MKL SGEMM subroutines to be competitive, since they also consider the same optimization methods. This proposed optimized SGEMM algorithm is specialized for the large size data and has the best utilization of the memory bandwidth. The results are measured also on single-core and multicore platforms. Finally, a comparative study between Intel C++ 17.0 compiler [23] in compared to Microsoft Visual Studio C++ 2015 [26] compiler is given in details, to show the effects of two widely used compilers."

The rest of paper has been organized in the following way: Section 2, presents matrix-matrix multiplication algorithm. Implementing Intel AVX's is outlined in Section 3. In Section 4, optimization techniques are introduced. The experimental results are outlined and analyzed in Section 5. Finally, Section 5 addresses conclusion and future work for this work.

2. Matrix multiplication algorithm

BLAS of level 3 can be defined as a set of subprograms to carry out matrix-matrix multiplication. They provide portable and efficient structures blocks to deal with linear algebra solution techniques. Matrix-matrix multiplication is viewed as an operation related to linear algebra operations, largely and smoothly applied to various applications. The Level 3 BLAS is limited by the number of CPU FLOPs (CPU bound) on most hardware and with blocked algorithms. Matrix- matrix multiplication is usually applied to numerical problems, scientific, and digital signal processing, so it is essential to speed up application of matrix-matrix multiplication [16–18]. It constitutes the fundamentals of the level-3 BLAS processes, which covers ($2\ N^3$) mathematical processes, but makes and expends ($3\ N^2$) data worth. Tuning matrix multiplication pro-

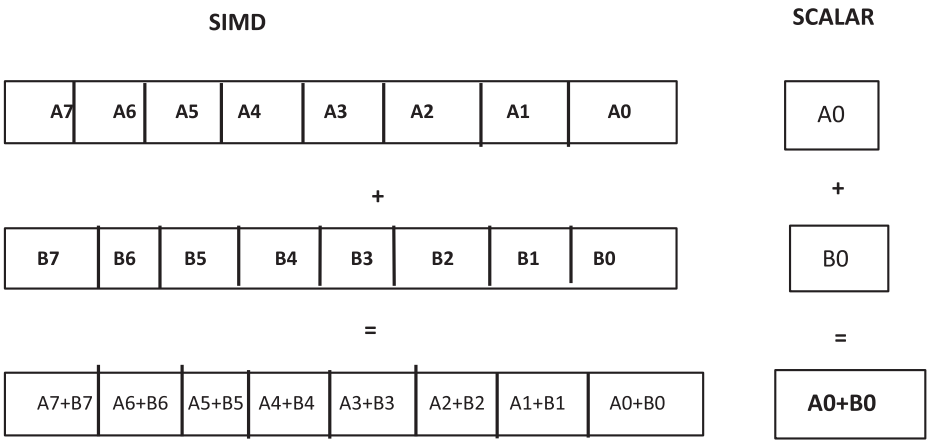

Fig. 1. Scalar and parallel summing operation.

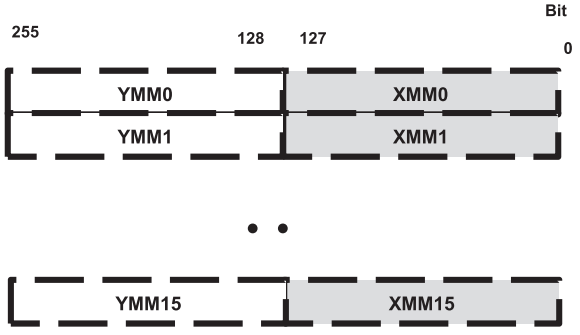

Fig. 2. SIMD 256-bit wide register.

poses to develop the most powerful available GEMM routines, to be used for any exist architecture. BLAS SGEMM takes three matrices (A, B, C) and modifies C according to the matrix operation:

Where C is (N × N) matrix, op (A) is the (N × N) matrix given by A or A^T and op (B) is the (N × N) matrix is given by B or B^T. The implementation of matrix-matrix multiplication is three nested loops as shown in Fig. 3 [19,20].

3. Implementing Intel's AVX

This paper aims to implement matrix-matrix multiplication algorithm presented in Section 3 using AVX instruction sets and optimization methods. In current architectures, single Instruction, Multiple Data (SIMD) is the core of controlling the execution and is developing best rendering with instructions that are fixed by prescript throughput. Intel's AVX prescripts sets permitting one floating-point prescripts to be applied on eight pairs of (32-bit) floating- point numbers or four pairs of (64-bit) floating- point

```
for (i=0 ;i<N ; i++){
for(k=0;k<N;k++){
acc=a[i][k];
for(j=0; j<N; k++){
c[i][j]+= acc* b[k][j];
}
}
}
```
(a) C=A*B

```
for(i=0; i<N; i++){
for(j=0; j<N; j++){
sum=0;
for(k=0 ;k<N; k++){
sum+= a[i][k]* b[j][k];
}
c[i][j]+=sum;
}  }
```
(b) C= A* B^T

```
for(k=0; k<N; k++){
  for(i=0; i<N; i++){
    acc=a[k][i];
     for(j=0 ;j<N; j++){
c[i][j]+= acc* b[k][j];
     }  }  }
```
(c) C= A^T*B

Fig. 3. Conventional matrix-matrix multiplication kernels.

numbers simultaneously. A distinct destination argument is exist in Intel Advanced Vector Extension that produces in fewer register copies, better register use, and lower size of code [1,2,11,14,15].

We implement a manual tuning of the algorithms for big square matrices, which ranges from 2080 to 4512. The implementation is validated on Lenovo Laptop computer with an Intel Core i7 platform [21,22] of 2.6 GHz with 128 KB L1 cache 512 KB L2 cache, and 4 MB L3 cache rendering on windows 10O.S. It is essential to utilize a suitable compilers that can significantly implement the updated characters of the developing processors. The programs are compiled with Intel C++ compiler 17.0 in Intel Parallel Studio XE 2017 [23–25] and Microsoft Visual C++ 2015 compiler [26]. Intel core i7 processors support two cores and up to four threads with Intel Hyper-Threading Technology (Intel HT Technology). This technology delivers two processing threads per physical core. Applications that are highly threaded can get more work done in parallel and completing tasks sooner. The implementation is coded using Intel intrinsic functions for AVX code. The measuring metric for performance is Giga flops per seconds (GFLOPs) of each kernel. The amount of floating-point operations in matrix-matrix multiplication is ($2n^3$ FLOPs). The clock cycles were measured using RDTSC command [27]. With Intel AVX instructions we can process 512 bits of data per cycle with vaddps and vmulps simultaneously. Fig. 4 shows an example of summing two vectors using Intel AVX instructions [11]. Using intrinsic functions helps for easy programming way utilizing the standard C/C++ language, improving code readability, less errors, and eliminating function call overhead. On the other hand, inline assembly is the easiest way to handle of every part of a program using low level assembly instructions [28].

4. Optimization techniques

The value of effective implementation of the memory hierarchy is increased as the gap between CPU and memory performance continues to grow. This is clear in determining dense algorithms that able to execute a huge number of date, such as the numerical problems. In order to determine the effectiveness of techniques that aim at improving memory utilization, a classical benchmark is established, which known as the key problem of dense matrix multiplication. We attempt to reach optimization through use of the cache hierarchy in an efficient and effective manner. Optimizing matrix multiplication algorithms in the context of their performance need a significant comprehension of how memory hierarchy functions and how it should be manipulated to augment data reuse, number of floating point units, size of cache levels, and

```
For(i=0; i,<n; i+=8){
__asm{
vmovaps ymm0,ymmword [eax]
vmovaps ymm1,ymmword[ebx]
vaddps ymm0,ymm1,ymm0
vmovaps ymmword[ecx],ymm0
}
}
```
(a) inline assembly language

```
For(i=0; i,<n; i+=8){
__m256 a1 = _mm256_load_ps(&a[i]);
__m256 b1 = _mm256_load_ps(&b[i]);
__m256 c1 = _mm256_add_ps(a1, b1);
__mm256_store_ps(&c[i], c1);
}
```
(b) intrinsic functions

Fig. 4. Summing two vectors using AVX code.

other important elements has to be considered [29–32]. The objective of the current section revolves around introducing efficient application of matrix-matrix multiplication kernels found in Section 2 for triples its cases (SGEMM) using AVX instruction sets and optimization techniques.

4.1. Loop unrolling

The main idea of unrolling is by increasing the inner loop complexity to augment the quantity of floating point processes as opposed to the quantity of load/store memory access. "Reusing data by storing it in registers as much as possible, so that it does not have to be repeatedly loaded from the caches. In addition, accumulating results in registers as long as possible, so that it reduces the times of storing data into the cache memory. Loop unrolling can be combined with prefetching technique to process elements as much as the cache line contains [33–36]. Listing 1 shows solution steps of matrix multiplication algorithm with unrolling."

4.2. Blocking

This is a renowned technique for optimization which helps improve and optimize memory hierarchy's efficacy. When the gap between fast register and slow main memory expands, the cache hierarchy comes into the mix to address the different and augment performance. This is an efficient manner through which the bandwidth of memory can be augmented, where a large matrix is broken down into smaller fixed size matrices. The bottleneck then is finding a suitable size for the block, which is big enough to steer clear of many load and store processes, but is tiny enough to easily fit into the cache itself. It helps to arrange the matrix-matrix multiplication kernel so that data locality can be achieved and cache misses are avoided. Data locality works very well for level3 BLAS operations such as matrix multiplication. Therefore, the matrix-matrix multiplication is carried out as blocks [37–40].

In Fig. 5 and Listing 2 the blocking algorithm is presented where BS is (block_size). Fig. 6 shows the unblocked matrix multiplication in which the element for matrix A is used by all iterations of the innermost loop so that it is fetched from memory once and can be stored in a register. In C language, the matrix is arranged in a row major order so that the data is accessed in a consecutive order in B and C matrices [41,42] . From Fig. 7 the basic idea of blocking algorithm is to divide matrix A and C into 1xBS row in dark gray area and to divide matrix B into BSxBS blocks. The innermost (k, j) loops multiply a dark gray area of A by a block of B and accumulate the result into a dark gray area of C. The loop (i) iterates using the same block in B through N row dark gray area of A and C. The key idea is loading the block of matrix B into the cache, uses it up, then rejects it, and the same row of B is reused in the outermost loop. Each dark gray area is accessed with a stride of one in matrix A, so it has a good spatial locality. The entire dark gray area is referenced BS times in succession, "so there is also good temporal locality. The entire BSxBS block is accessed N times in succession, so matrix B has a good temporal locality. In addition, matrix C has good spatial locality because each element of the dark gray area is written in succession and the same row of C is reused in the next iteration of the outer loop. The blocking size, BS is chosen so that the NbxNb submatrix of B and a row of length Nb of C can fit in the cache. Hence, both B and C are reused Nb times each time

```
Step1: load packed single precision floating point eight values from matrix A in the first row.
Step2: load next packed single precision floating point eight values from matrix A in the first row.
Step3: load packed single precision floating point eight values from matrix B.
Step4: blend first packed single precision floating point eight values of matrix A with zero.
Step5: multiply the blended results of matrix A by the packed elements of matrix B.
Step6: add the packed elements resulted in step5 to the accumulator.
Step7: blend the packed elements of matrix A with 9F50D1.
Step8: load next packed elements of matrix B.
Step9: blend the next packed eight elements of matrix A with zero.
Step10: multiply the blended results of matrix A in step9 by the next packed elements of matrix B in step8.
Step11: add the packed elements resulted in step10 to the accumulator.
Step12: blend the next packed elements of matrix A with 9F50D1.
Step13: repeat the steps from step3 to step12 for the next row of matrix B.
Step14: load the next packed elements of matrix A.
Step15: repeat the steps from step1 to step13.
Step16: store the packed eight elements of matrix C.
Step17: store the next packed eight elements of matrix C.
Step18: load the packed eight elements of matrix A in the next row.
Step19: repeat the steps from step3 to step13.
```

Listing 1. Unrolled solution steps of matrix multiplication.

Unblocked matrix multiplication	Blocked matrix multiplication

for (i = 0; i<N; i++){
for (k = 0; k<N; k++){
acc = a[i][k];
for (j = 0; j<N; j++){
c[i][j] += acc * b[k][j];
}}}
	for (kb = 0; kb <N; kb += BS){
	for (jb = 0; jb < N; jb += BS){
	for (i = 0; i <N ; i ++){
	for (k = kb; k < kb + BS; k ++){
	acc1 = a[i][k];
	for (j = jb; j<jb + BS; j++){
	c[i][j] += acc1*b[k][j];}}}}}

Fig. 5. Blocked vs. Unblocked matrix-matrix multiplication.

the data are brought. However, the performance is decreased if the ratio of memory fetches to numerical operations is increased [33]."

4.3. Prefetching

Matrix multiplication has been a tricky kernel to optimize for cache prefetching because it exhibits temporal locality in addition to the normal spatial locality [35]. Recent Intel processors families use many prefetching systems to augment the code's speed and aid performance [36]. Two types of prefetching may be deployed, software and hardware. The former can be created automatically or manually and requires the use of the PREFETCH instructions. Through this a compiler or programmer inserts prefetch code into the system.

The merit of software-controlled strategies is the higher possibility of the developers or a compiler that have better knowledge of the application's data requirements to gain more from software prefetching [43,44]. However, the implementation should be carefully designed to avoid cache pollution by the prefetching data. Prefetching attempts to hide memory latencies by initiating a prefetching instruction sufficiently early before the data item is used as shown in Fig. 8. Memory hierarchy consists of several levels including main memory, cache memory, and storage. Prefetching data can be implemented at different levels of memory hierarchy. Fig. 9 presents the memory hierarchy layout and Fig. 10 illustrates prefetching instruction in matrix multiplication algorithm.

4.4. OpenMP

(Open Multi-Processing) API makes it possible to add parallelism into the cost that already exists without substantially rewriting anything. OPenMP is a helpful tool for loop parallelization. It can control the parallel process, receive great performance increase. Furthermore, it can be easily applied within compiler directives, environment variables, and library routines. This makes it possible for users to develop and run parallel programs while permitting portability. The users insert directives to assist the compiler into generating threads for the parallel processor platform [43–45]. The OPenMP Parallelization example of matrix multiplication are presented in Fig. 11.

We can use (***omp parallel for***) in different loops order, the innermost loop k, the outer loops i, j and jb and the outermost loop kb. Parallelizing loop i will lower thread creation overhead since it will only create the threads once. Every thread executes statements in parallel block in parallel. Using data in cache and providing bigger context for compiler optimization have more opportunities to be offered in large parallel regions. Fig. 12 presents loops parallelization. Thread starting time, synchronization, software overhead, and thread terminating time are the parallel overhead factors using OpenMP. When the parallelization region is outside the loop nest, the overhead of parallelization is minimized. The merits of OpenMP include:"

- Allows the increment parallelization of the program and requires very little programming effort.
- It is supported by a large number of compilers (portable).
- It has a good performance.
- It is widely used and available, lightweight, mature, and suited ideally for multi-core architecture.

5. Experimental results

This section explores the results of the study, i.e. pertaining to a hand-tuned implementing nxn square matrix-matrix multiplication "as the objective issue for large matrices ranges from 2080 to 4512 step 128 of single precision floating-point data as an example of large matrices that are needed in different applications. We have implemented three single-precision kernels: C = A. B, C = A. B^T, and C = A^T. B utilizing Intel's AVX instructions and the previous mentioned optimization techniques in Section 4 to overcome the performance achieved by the Intel MKL SGEMM. In addition, the comparison between ICC 17.0 compiler over MSVC++ 2015 compiler and single-core over multicore are introduced."

The performance of the optimization techniques on single thread is presented from Tables 1–7.

From Tables 1 and 2, we can observe that unrolling works better in larger matrices with ICC compiler than with MSVC++ compiler. Additionally, MSVC++ compiler outperforms ICC compiler in larger matrices using blocking. Prefetching works better in larger matrices with ICC compiler than MSVC++ compiler and fluctuates values in different sizes of matrices.

```
Step1: load packed single precision floating point eight values from matrix A in the first row.
Step2: load packed single precision floating point eight values from matrix B.
Step3: blend first packed single precision floating point eight values of matrix A with zero.
Step4: multiply the blended results of matrix A by the packed elements of matrix B.
Step5: add the packed elements resulted in step5 to the accumulator.
Step6: blend the packed elements of matrix A with 9F50D1.
Step7: load first packed elements of matrix B in the second row.
Step8: repeat the steps from step4 to step7.
Step9: repeat step8 for eight rows of matrix B.
Step10: repeat the steps from step1 to step10 for the block size (kb) of matrix B.
Step11: store the packed eight elements of matrix C.
Step12: load the packed eight elements of matrix A in the second row.
Step13: repeat steps from step2 to step12.
Step14: repeat steps from step13 and step14 for the block size (ib) of matrix A and C.
```

Listing 2. Solution steps of blocking matrix-matrix multiplication.

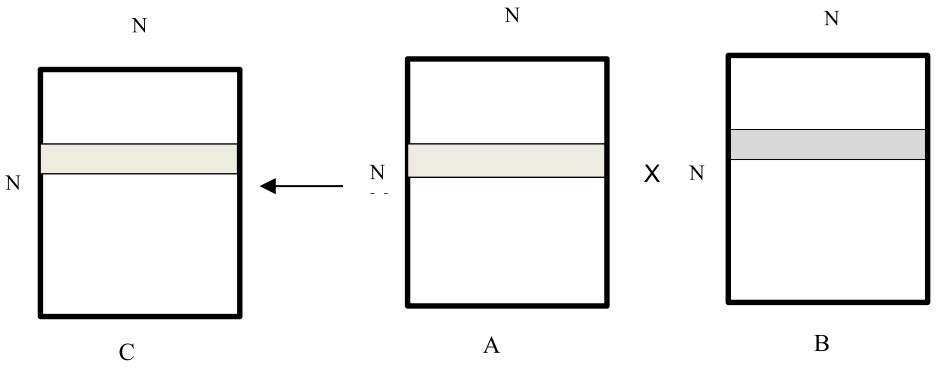

Fig. 6. Accessing data for unblocked matrix multiplication.

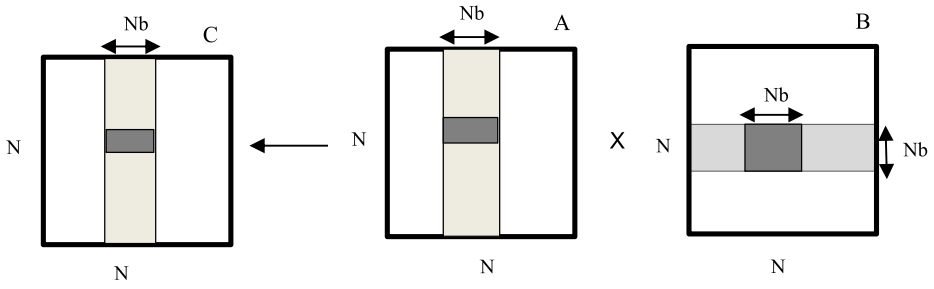

Fig. 7. Accessing data for Blocked matrix multiplication.

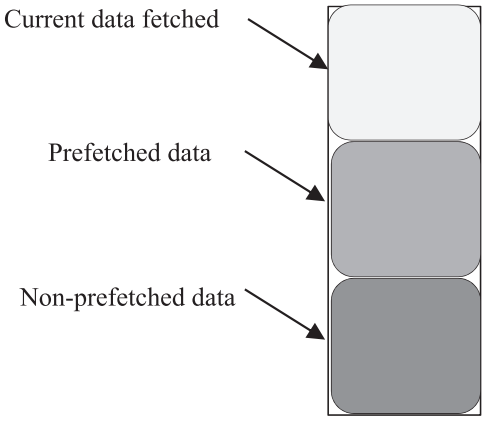

Fig. 8. Data prefetching in cache.

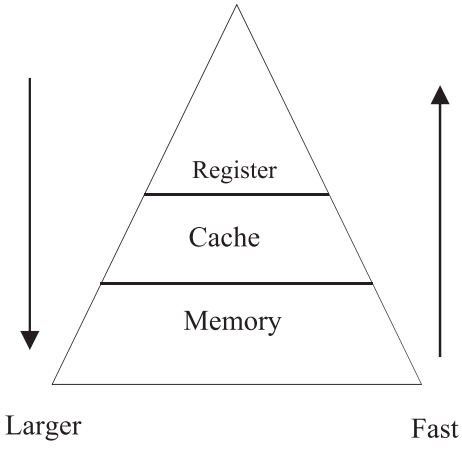

Fig. 9. Memory hierarchy layout.

From Tables 3 and 4, we can observe that unrolling gives variable values depending on the size of the matrices with ICC and MSVC++ compilers. Blocking also works well in large matrices with ICC compiler and gives variable values depending on the size of the matrices with MSVC++ compiler. Prefetching works better in larger matrices with ICC and MSVC++ compilers.

From Tables 5 and 6, we can observe that unrolling, blocking, and prefetching give variable values depending on the size of the matrices with ICC and MSVC++ compilers.

We evaluated the performance of the proposed optimized matrix-matrix multiplication (New_SGEMM) in three forms (C = A. B, C = A. B^T, and C = A^T. B) Compared to the results of the latest version of Intel MKL SGEMM subroutines. The performance of the optimized matrix-matrix multiplication compared to Intel MKL SGEMM using ICC and MSVC++ compilers on single thread are presented in Figs. 13–18.

```
for (kb = 0; kb < N; kb += BS){
for (ib = 0; ib < N; ib += BS){
        _mm_prefetch; // prefetch A
for (j = 0; j < N; j+=vector_length){
    _mm_prefetch; // prefetch B
    _mm_prefetch;  // prefetch C
for (i = ib; i < ib + BS; i++){
for (k = kb; k < kb + BS; k += vector_length){
…… // AVX code        }
}
}
}}
```

Fig. 10. Prefetching algorithm.

```
#pragma omp parallel
for (kb = 0; kb < N; kb += BS){
    for (jb = 0; jb < N; jb += BS){
   for (i = 0; i < N; i ++){
     for (j = jb; j < jb + BS; j++){
                . . . . . .. // AVX code
      for (k = kb; k < kb + BS; k +=8){
              . . . . . .
             // AVX code. . }
               . . . . . . . . }
               . . . . . . . .}
                   }
                 }
```

Fig. 11. OPenMP Parallization.

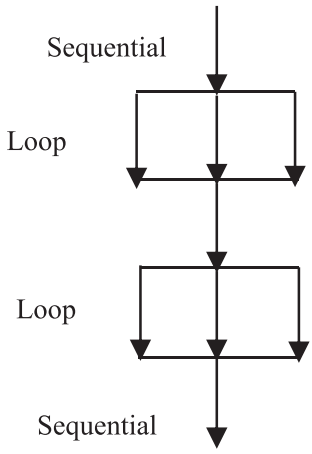

Fig. 12. Loop parallelization using OPenMP.

Table 1
Optimization performance for C = A. B with Intel compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)
2208 × 2208	8.44%	39.4%	15.6%
2592 × 2592	54.3%	14.1%	8.04%
3488 × 3488	31.1%	2.18%	8.07%
4128 × 4128	24.9%	9.81%	24.9%

Table 2
Optimization performance for C = A. B with MSVC++ compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)
2208 × 2208	27.6%	19.8%	13.7%
2592 × 2592	7.24%	50.8%	30.4%
3488 × 3488	12%	46.6%	5.05%
4128 × 4128	3.66%	41.3%	13%

Table 3
Optimization performance for C = A. B^T with Intel compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)
2080 × 2080	41.5%	7.26%	30%
2976 × 2976	14.9%	1.9%	7.06%
3360 × 3360	43.3%	11.8%	4.81%
4000 × 4000	4.08%	11.8%	40.3%

Table 4
Optimization performance for C = A. B^T with MSVC++ compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)
2080 × 2080	12.2%	37%	19.7%
2976 × 2976	27.4%	60.2%	13.9%
3360 × 3360	17.1%	18.9%	29%
4000 × 4000	8.31%	18%	31.7%

Table 5
Optimization performance for C = A^T. B with Intel compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)
2080 × 2080	32%	8.99%	1.94%
2592 × 2592	12.8%	31%	13.8%
3488 × 3488	37.1%	25%	5.36%
4512 × 4512	13.5%	16.8%	23.4%

Table 6
Optimization performance for C = A^T. B with MSVC++ compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)
2080 × 2080	18.5%	45.5%	0.183%
2592 × 2592	19.6%	51.3%	27.1%
3488 × 3488	3.44%	34.6%	22.4%
4512 × 4512	19.5%	34.4%	0.523%

Our experimental results show that the optimized (New_SGEMM-1 T) achieves the average performance improvement of 49%, 54%, and 40% compared to the results achieved using Intel MKL SGEMM (Intel MKL-1 T) subroutines on single thread with Intel compiler and 45%, 42%, and 50% with MSVC++ compiler for C = A . B, C = A. B^T, and C = A^T. B respectively as shown in Figs. 13–18.

From Figs. 13–18, we can find that ICC compiler works better than MSVC++ compiler in C = A. B, C = A. B^T, and MSVC++ compiler works well than ICC compiler in C = A^T. B. Additionally, Intel MKL-1 T works close to New_SGEMM-1 T in some values of matrices.

The performance of matrix-matrix multiplication using optimization techniques on four threads with different two compilers are shown in Tables 7–12.

From the results that are shown in Tables 7–12, "we can observe that the performance of the cache is highly dependent on the problem size and the block size. The same block size can give rising to widely varying cache miss rates for very similar problem sizes. Prefetching works well with different sizes of matrices, which the data in the matrix can be stored directly into the L1 cache and there is no frequent data transfer between cache and main memory.

OpenMP parallelization works very well with matrices of large sizes. This may be due to the big amount of work in them and they have enough computational cost to justify multithreading. Basically, OpenMP is gainful for doing large and expensive loops."

The performance of the optimized matrix-matrix multiplication compared to Intel MKL SGEMM using ICC and MSVC++ compilers on four threads are presented in Figs. 19–27.

From Fig. 19, the optimized (New_SGEMM-4T) achieves the average performance improvement of 71% compared to the results achieved using Intel MKL SGEMM (Intel MKL-4T) subroutine with Intel compiler.

The optimized (New_SGEMM-4T) achieves the average performance improvement of 70% compared to the results achieved using Intel MKL SGEMM (Intel MKL-4T) subroutine with MSVC++ compiler as shown in Fig. 20.

Table 7
Optimization performance for C = A. B with Intel compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)	With OpenMP (%)
2208 × 2208	63.3%	17.9%	56.7%	3.54%
2592 × 2592	22%	20.3%	21.7%	47.9%
3232 × 3232	20.7%	19%	41.4%	41.4%
4128 × 4128	11.5%	22.9%	29.7%	42.6%

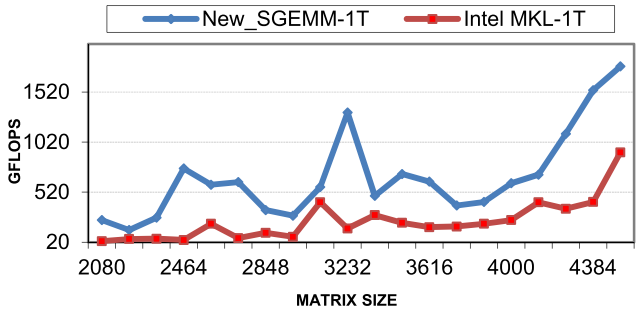

Fig. 13. C = A. B on Single Thread with Intel compiler.

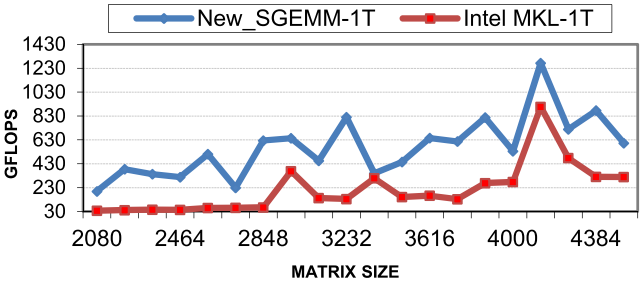

Fig. 14. C = A. B on Single Thread with MSVC++ compiler.

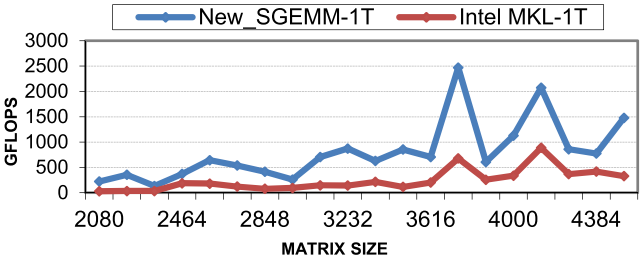

Fig. 15. C = A. B^T on Single Thread with Intel compiler.

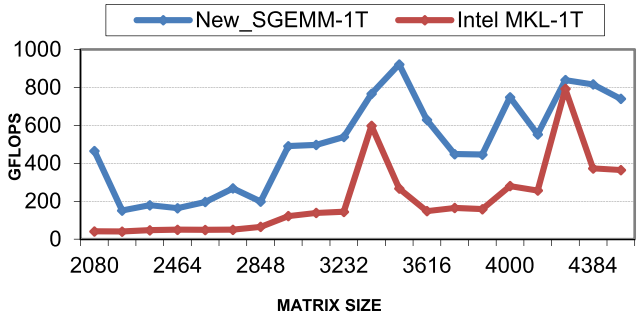

Fig. 16. C = A. B^T on Single Thread with MSVC++ compiler.

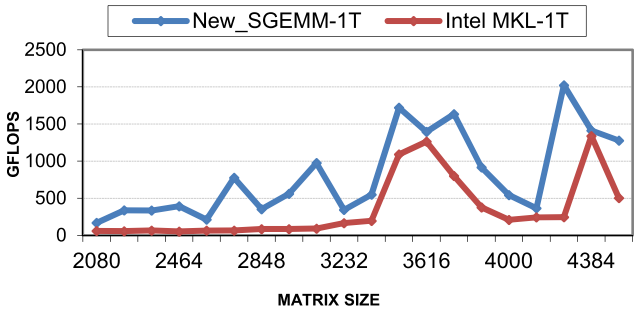

Fig. 17. C = A^T. B on Single Thread with Intel compiler.

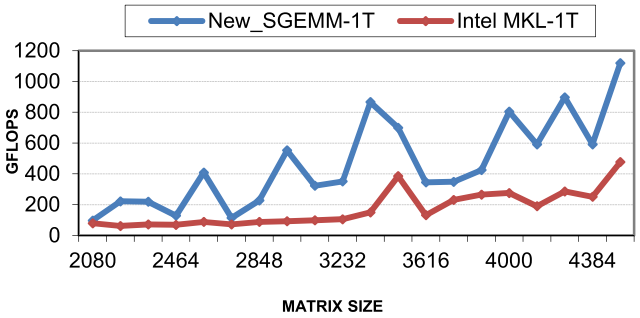

Fig. 18. C = A^T. B on Single Thread with MSVC++ compiler.

The average speedup (the ratio of the execution time using Intel compiler to the execution time using MSVC++ compiler) is introduced in Fig. 21. The average speedups are 3.19 and 1.36 for C = A. B.

In Fig. 22, the optimized (New_SGEMM-4T) achieves the average performance improvement of 59% compared to the results achieved using Intel MKL SGEMM (Intel MKL-4T) subroutine with Intel compiler.

The optimized (New_SGEMM-4T) achieves the average performance improvement of 56% compared to the results achieved using Intel MKL SGEMM (Intel MKL-4T) subroutine with MSVC++ compiler as shown in Fig. 23.

The average speedups are 1.67 and 1.32 for C = A. B^T of using Intel compiler against MSVC++ compiler.

The optimized (New_SGEMM-4T) achieves the average performance improvement of 56% compared to the results achieved using Intel MKL SGEMM (Intel MKL-4T) subroutine using Intel compiler.

The optimized (New_SGEMM-4T) achieves the average performance improvement of 53% compared to the results achieved using Intel MKL SGEMM (Intel MKL-4T) subroutine using MSVC++ compiler.

The average speedups are 2.25 and 1.54 for C = A^T. B using Intel compiler against MSVC++ compiler.

We have presented the evaluation study for different matrix sizes, various optimization parameters, parallelization by OpenMP directives and two different compilers that impress the perfor-

Table 8
Optimization performance for C = A. B with MSVC++ compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)	With OpenMP (%)
2208 × 2208	22.0%	65.5%	14.1%	15.5%
2592 × 2592	27%	5.27%	30.8%	31.4%
3232 × 3232	17.4%	6.98%	32.6%	33%
4128 × 4128	15.6%	66.6%	13.5%	48.2%

Table 9
Optimization performance for C = A. B^T with Intel compile.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)	With OpenMP (%)
2208 × 2208	57%	21.9%	9.71%	19.7%
2720 × 2720	8.69%	33.1%	11.9%	15.3%
3232 × 3232	9.35%	12%	22.4%	39.3%
4000 × 4000	12.7%	39.4%	15.8%	10.2%

Table 10
Optimization performance for C = A. B^T with MSVC++ compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)	With OpenMP (%)
2208 × 2208	18.6%	3.39%	11.0%	33.5%
2720 × 2720	6.7%	18.8%	33.3%	32.0%
3232 × 3232	16.6%	3.67%	22.7%	33.7%
4000 × 4000	26.6%	5.50%	9.03%	21.3%

Table 11
Optimization performance for C = A^T. B with Intel compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)	With OpenMP (%)
2080 × 2080	30.9%	20.5%	8.39%	6.26%
2592 × 2592	2.88%	19.9%	15.8%	6.29%
3232 × 3232	17%	7.25%	9.57%	6.23%
4128 × 4128	13.6%	7.67%	6.7%	17.5%

Table 12
Optimization performance for C = A^T. B with MSVC++ compiler.

Size of matrix	With unrolling (%)	With blocking (%)	With prefetching (%)	With OpenMP (%)
2080 × 2080	47.8%	2.27%	6.95%	6.00%
2592 × 2592	6.29%	33.8%	33.5%	5.51%
3232 × 3232	20.7%	14.4%	13.5%	11.2%
4128 × 4128	20.3%	9.69%	12.3%	25.2%

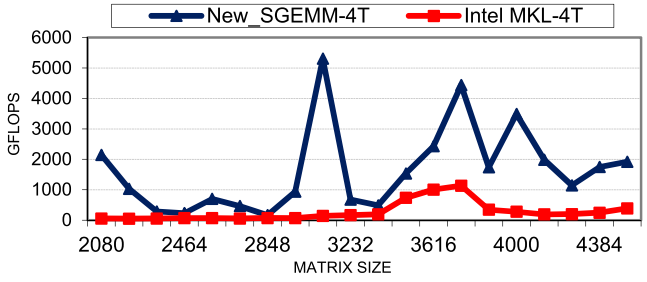

Fig. 19. Matrix Multiplication Performance C = A. B on Four Threads with Intel compiler.

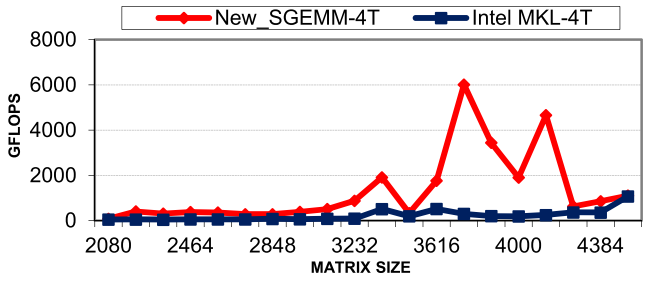

Fig. 20. Matrix Multiplication Performance C = A. B on Four Threads with MSVC++ compiler.

mance. In addition, the evaluation presents detailed steps of optimization techniques. We have tested different block sizes 128, 256, 512, and 1024, and find the best block size is 1024 which achieves high performance. This may be due to the number of blocks of divided matrices which are loaded into the cache. The less number of fetched blocks, the better performance of the choosing block size. "Intel core i7 employs several prefetching mechanisms to accelerate the movement of data or code and improves performance. Intel core i7 Broadwell has shown an accomplished implicit to speed up the algorithm through the optimization process. From the previous results, we found that ICC compiler has better performance in most different sizes of matrices than MSVC++ compiler for compiling AVX code. Intel compiler 17.0 offers simplified vectorization, guided auto parallelization support and high perfor-

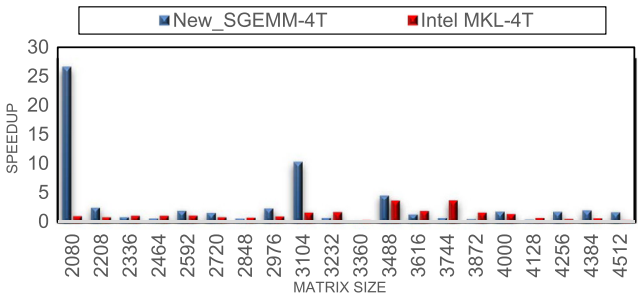

Fig. 21. Speedup of Intel compiler vs. MSVC++ compiler for C = A. B.

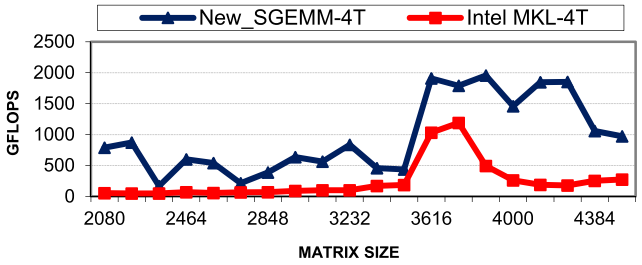

Fig. 22. Matrix Multiplication Performance C = A. B^T on Four Threads with Intel compiler.

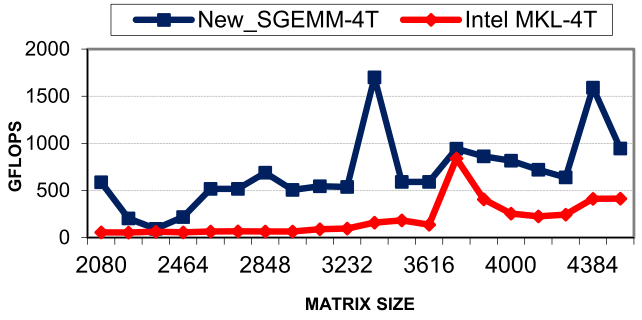

Fig. 23. Matrix Multiplication Performance C = A. B^T on Four Threads with MSVC++ compiler.

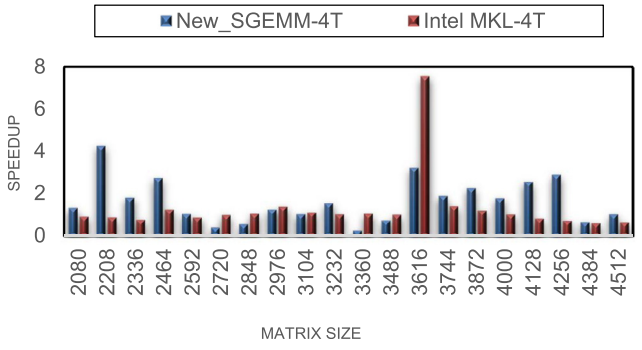

Fig. 24. Speedup of Intel compiler vs. MSVC++ compiler for C = A. B^T.

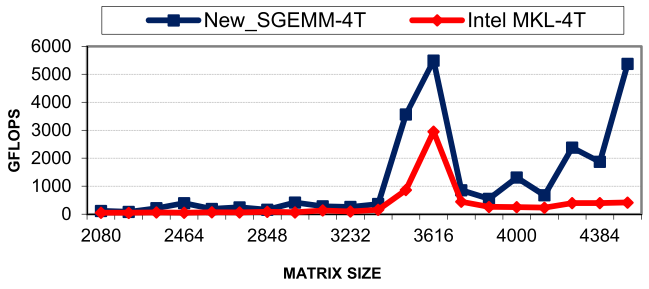

Fig. 25. Matrix Multiplication Performance C = A^T. B on Four Threads with Intel compiler.

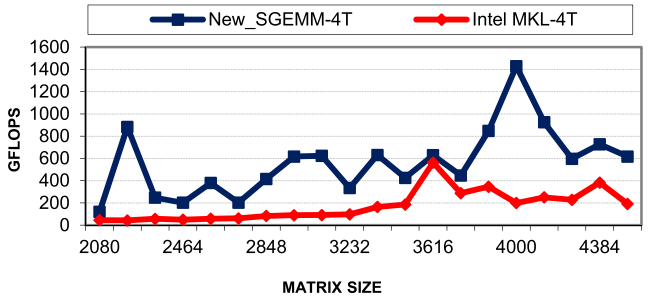

Fig. 26. Matrix Multiplication Performance C = A^T. B on Four Threads with MSVC++ compiler.

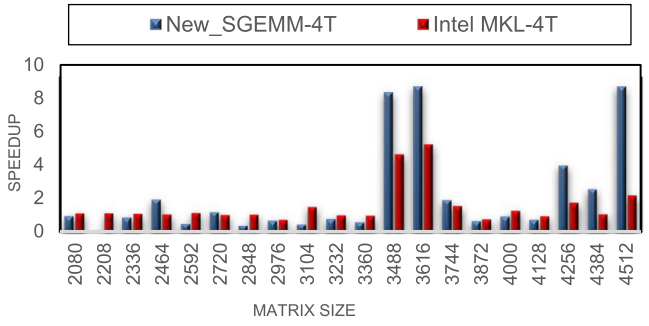

Fig. 27. Speedup of Intel compiler vs. MSVC++ compiler for C = A^T. B.

mance parallel optimizer. It is designed to produce code that executes in few number of cycles, it has the best support for SIMD instructions, it can exploit its processor and the system, and it supports both automatic parallelization, as well as manual, and does both very well. MSVC++ works well and very close to Intel compiler performance. Intel compiler is almost always faster than MSVC++ compiler and sometimes it is slower than it. However, this depends on the kind of compiled software. MSVC++ compiler has new improvement capabilities in vectorization, but there are still several classes of vector optimization which MSVC++ misses. Hence, both compilers are good pieces of development programs'. Intel core i7 processors also support two cores and up to four threads with Intel Hyper-Threading Technology (Intel HT Technology). This technology delivers two processing threads per physical core. The implementation also compares single-core against multicore platforms. Highly threaded applications can get more work to do in parallel and completing tasks sooner since more than one thread can execute in parallel. The results provide the increasing efficiency of multithreads over single-thread OpenMP implementation is gainful if there is big amount of work to avoid the overhead, but the overhead may increase for the small amount of work. With OpenMP, increasing the number of threads increase the speed up over using only four threads. Multithreads can increase the efficiency of specific task by using multiple threads in parallel which can result in significant performance gains.

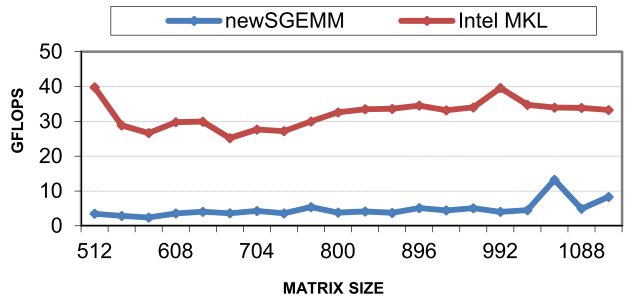

Fig. 28. Performance of C = A. B in Small Matrices Sizes.

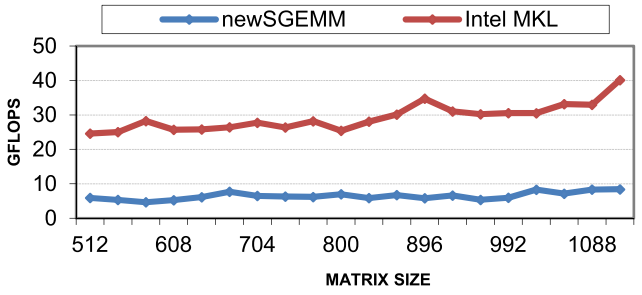

Fig. 29. Performance of C = A. B^T in Small Matrices Sizes.

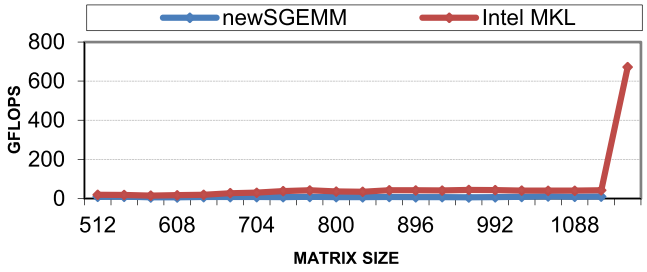

Fig. 30. Performance of C = A^T. B in Small Matrices Sizes.

Figs. 28–30 show the performance of the proposed New_SGEMM in small size of matrices compared to Intel MKL SGEMM.

From Figs. 28–30, we can observe that the proposed algorithm New_SGEMM is not valuable with small sizes of matrices in compared to Intel MKL subroutines.

6. Conclusion and future work

In this paper, we implement the optimization techniques manually to find substantial overall algorithm based high performance. Moreover, three algorithms for matrix-matrix multiplication using Intel Advanced Vector Extension instructions, memory access optimization, and OpenMP parallelization running on Intel Core i7 processor are evaluated. In the evaluation study we have tested the proposed optimized algorithms (New_SGEMM) and compared them against the latest Intel MKL (SGEMM) subroutines. The results clearly show that our optimized algorithms (New_SGEMM) provide 71%, 59%, and 56% on multithreads platform for C = A. B, C = A. B^T, and C = A^T. B respectively." In addition, matrix-matrix multiplication algorithms which compiled with ICC compiler are quicker than those compiled with MSVC++ compiler. The results turn out that multithreads achieve significantly higher average performance improvement than single thread by 57.3%. This optimized algorithms are extremely helpful for many applications that are based on matrix-matrix multiplication having to do with large matrices, and eat up a good amount of time to be run out.

In future work, we aim to Exploit GPU together with a CPU to accelerate BLAS subroutines, measure the performance of the proposed algorithms using cache miss rate and memory bandwidth limitation metrics. Extending sizes of matrices and blocking sizes in implementing BLAS subroutines using the proposed algorithm.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Intel Advanced Vector Extensions Programming Reference, Ref. #319433-005, www.intel.com, January 2009.
[2] Intel Architecture Instruction Set Extensions Programming Reference, Ref. #319433-14, www.intel.com, August, 2012.
[3] Whaley RC, Dongarra JJ. Automatically tuned linear algebra software, Technical report, Computer Science Department, University of Tennessee, 1997. http://www. netlib.org/utk/projects/atlas/.
[4] Demmel J, Dongarra J, Eijkhout Victor, Fuentes Erika, Petitet Antoine, Vuduc Rich, Whaley Clint, Yelick Katherine. Self adapting linear algebra algorithms and software. Proc. IEEE special issue on Program Generation, Optimization, and Adaptation, 93 (2). doi: https://doi.org/10.1109/JPROC.2004.840848.
[5] Chen Z, Dongarra J, Luszczek P, Roche K. Self-adapting software for numerical linear algebra and LAPACK for clusters. Parallel Computing 2003;29(11–12):1723–43. doi: https://doi.org/10.1016/j.parco.2003.05.014.
[6] Gunnels JA, Henry GM, van de Geijn RA. A family of high-performance matrix multiplication algorithms May 28-30. In: Proceedings of the International Conference on Computational Sciences-Part I. p. 51–60.
[7] Gepner P, Gamayunov V, Fraser DL. Effective implementation of DGEMM on modern multicore CPU. Procedia Comput Sci 2012;9:126–35.
[8] Hadizadeh Ali, Tanghatari Ehsan. Parallel processor architecture with a new algorithm for simultaneous processing of MIPS-based series instructions. Emerg Sci J 2017;1(4). doi: https://doi.org/10.28991/ijse-01126.
[9] Lim Roktaek, Lee Yeongha, Kim Raehyun, Choi Jaeyoung. An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512. Cluster Comput 2018;21(4):1785–95. doi: https://doi.org/10.1007/s10586-018-2810-y.
[10] Intel® Math Kernel Library Parallel Studio XE 2017 for Windows* Developer Guide, Intel® MKL 2017-Windows*, Revision 054, ww.Intel.com.
[11] Intel Intrinsic Guide, Jul 15, 2019, https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
[12] Jeong H, Lee W, Kim S, Myung S-H. Performance of SSE and AVX Instruction Sets. arXiv preprint arXiv: 1211.0820, 2012.
[13] Lomont C."Introduction to Intel Advanced Vector Extensions; 2011, https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.
[14] Intel 64 and IA-32 Architectures Software Developers Manual. Vol. 2 (2A, 2B&2C), www.intel.com, June 2015.
[15] Intel 64 and IA-32 Architectures Software Developers Manual. Vol. 3 (3A, 3B&3C), www.intel.com, June 2015.
[16] Dongarra JJ, Croz JD, Hammarling S, Duff I. A set of level 3 basic linear algebra subprograms. ACM Trans Mathem Softw 1990;16(1):1–17.
[17] Basic linear algebra subprograms (BLAS); 2001, http://www.netlib.org/blas.
[18] Watkins DS. Fundamentals of matrix computations, 3rd ed., New York (2010), ISBN: 978-0-470-52833-4.
[19] Higham NJ. Exploiting fast matrix multiplication within the level 3 BLAS. ACM Trans Mathem Softw 1990;16(4):352–68. doi: https://doi.org/10.1145/98267.98290.
[20] General Matrix Multiply Sample. User's Guide, 2013 Intel Corporation, Document Number: 325264-003US.
[21] 5th Generation Intel Core™ Processors based on the Mobile U-Processor Line, (Intel Core™ i7-5650U Processor, Intel Core™ i5-5350U Processor, Intel Core™ i3-5010U Processor), 2015, ww.intel.com.
[22] Intel Announces New 5th Gen Intel Core™ Processors and Latest Intel Xeon Processors at Computex 2015, June 2, 2015, ww.intel.com.
[23] Intel® Math Kernel Library Parallel Studio XE 2017 for Windows* Developer Guide, Intel® MKL 2017-Windows*, Revision 054, ww.intel.com.
[24] Intel® C++ Compiler in Intel Parallel Studio XE, Optimization Notes, August 16, 2012, last updated on February 2, 2016, ww.intel.com.
[25] Intel® C++ Compiler 16.0 Update 4 User and Reference Guide (PDF), 08/24/2015, www.intel.com.
[26] Visual C++ in Visual Studio 2015, https://msdn.microsoft.com/.
[27] Using The RDTSC Instruction for Performance Monitoring, www.intel.com, Intel Corporation; 1997.
[28] Hassan SA, Hemeida AM, Mahmoud MMM. Performance evaluation of matrix-matrix multiplication using intel's advanced vector extension. Microprocess

Microsyst 2016;47(PB):369–74. doi: https://doi.org/10.1016/j.micpro.2016.10.002.
[29] Gunnels J, Henry G, van de Geijn R. A family of high-performance matrix multiplication algorithms. In: Proceedings of the International Conference on Computational Sciences-Part I, ICCS 2001. London, UK: Springer-Verlag; 2001. p. 51–60.
[30] Aiken A, Banerjee U, Kejariwal A, Nicolau A. Instruction level parallelism. US: Springer; 2016. 10.1007/978-1-4899-7797-7.
[31] Peter G, Nikil D, Alexandru N. Memory architecture exploration for programmable embedded systems. US: Springer; 2002. 10.1007/b101865.
[32] Higham N. Accuracy and stability of numerical algorithms SIAM. Soc Indus Appl Mathem 2002. doi: https://doi.org/10.1137/1.9780898718027.
[33] Lam MS, Rothberg EE, Wolf ME. The cache performance and optimizations of blocked algorithms April 9-11. In: Fourth Intern. Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), Santa Clara, California, USA. p. 63–74.
[34] Anchev N, Gusev M, Ristov S, Atanasovski B. Some optimization techniques of the matrix multiplication algorithm. Proceedings of the ITI 2013 35th Int. Conf. on Information Technology Interfaces, Cavtat, Croatia, June 24-27, 2013.
[35] Kowarschik M, Wei C. An overview of cache optimization techniques and cache - aware numerical algorithms. In: Meyer U, Sanders P, Sibeyn J, editors. Algorithms for Memory Hierarchies. Lecture Notes in Computer Science, Vol. 2625. Berlin, Heidelberg: Springer; 2003. doi: https://doi.org/10.1007/3-540-36574-5_10.
[36] Ding X, Wang K, Zhang X. ULCC: a user-level facility for optimizing shared cache performance on multicores. ACM SIGPLAN Notices 2011;46(8).
[37] Ristov S, Gusev M, Velkoski G. Optimal Block Size for Matrix Multiplication Using Blocking, MIPRO 2014, 26-30 May 2014, Opatija, Croatia, 295–300, (2014).
[38] Frens JD, Wise DS. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In: Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) 206-216, jfrens, dswise@cs.indiana.edu, 1997.
[39] Schreiber R, Dongarra J. Automatic blocking of nested loops. Technical Report CS-90-108, Department of Computer Science, University of Tennessee, May 1990.
[40] Shaw Z, Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (like C). July 6, 2015.
[41] Hassan SA, Mahmoud MMM, Hemeida AM, Saber MA. Effective implementation of matrix-vector multiplication on Intel's AVX multicore processor. Comp Lang, Syst Struct 2017;51(C):158–75. doi: https://doi.org/10.1016/j.cl.2017.06.003.
[42] Elmroth E, Gustavson F, Jonsson I, Kagstrom B. Recursive blocked algorithms and hybrid data structures for dense matrix library software. Soc Indus Appl Mathem 2004;46(1):3–45. doi: https://doi.org/10.1137/S0036144503428693.
[43] Optimizing Software for Multi-core Processors, www.intel.com, 2007.
[44] OpenMP Application Program Interface Examples, Version 4.5 - November 2015, https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.
[45] Yang Y, Zhou H. The implementation of a high performance GPGPU compiler. Int J Parallel Program 2013;41(6):768–81. doi: https://doi.org/10.1007/s10766-012-0228-3.