ENHANCING THE MATRIX TRANSPOSE OPERATION USING INTEL AVX INSTRUCTION SET EXTENSION

Ahmed Sherif Zekri[1,2]

[1]Department of Mathematics & Computer Science, Alexandria University, Egypt
[2]Department of Mathematics & Computer Science, Beirut Arab University, Lebanon

ABSTRACT

General-purpose microprocessors are augmented with short-vector instruction extensions in order to simultaneously process more than one data element using the same operation. This type of parallelism is known as data-parallel processing. Many scientific, engineering, and signal processing applications can be formulated as matrix operations. Therefore, accelerating these kernel operations on microprocessors, which are the building blocks or large high-performance computing systems, will definitely boost the performance of the aforementioned applications. In this paper, we consider the acceleration of the matrix transpose operation using the 256-bit Intel advanced vector extension (AVX) instructions. We present a novel vector-based matrix transpose algorithm and its optimized implementation using AVX instructions. The experimental results on Intel Core i7 processor demonstrates a 2.83 speedup over the standard sequential implementation, and a maximum of 1.53 speedup over the GCC library implementation. When the transpose is combined with matrix addition to compute the matrix update, $B + A^T$, where A and B are squared matrices, the speedup of our implementation over the sequential algorithm increased to 3.19.

KEYWORDS

Matrix Transpose, vector instructions, streaming and advanced vector extensions, data-parallel computations

1. INTRODUCTION

Matrix transpose is a main operation in many matrix- and vector-based computations of image, video, and scientific and image/signal processing applications. For example, transforming an image from the time domain to the frequency domain requires the image data first accessed along the rows then along the columns.

Due to the high processing capabilities required for the previous applications, vendors of general-purpose (GP) CPUs extended the scalar instruction sets of their CPUs with short-vector registers and enough execution units in order to process more than one piece of data in parallel. This parallel execution mode boosted the performance of many data-parallel applications which are characterized by applying the same operation on multiple data. Intel MMX, SSE, AVX, AMD 3D!Now, Motorola's AltiVec, and Sun's VIS, are examples of short-vector instruction set extensions (ISAs) [11].

For solving large scale applications on parallel, distributed, and HPC machines, the data and processes are distributed among the processors or the nodes of the system so that each part of the whole application is performed on single node. Therefore, optimizing the kernel operations such as matrix transpose on nodes or cores (for current multi-core CPUs) is of great importance and leads to the overall performance enhancement of applications.

DOI:10.5121/ijcsit.2014.6305

Many accelerated algorithms for the matrix transpose operation on vector, parallel, SIMD, distributed, etc., have been devised in literature. In these algorithms, the matrix transpose operation have been studied as a communication problem, where different routing schemes are employed [1],[6],[10],[13]. The transpose operation is also treated as a special case of matrix permutations [7],[8],[12],[14]. Current one-dimensional implementations of matrix transpose on CPUs with instruction set extensions use inter- and intra-register data shuffle instructions in a way similar to Eklunde's original algorithm [2] for out-of-core matrix transpose. Few implementations of the matrix transpose on modern CPUs with SSE/AVX extensions have been devised [3],[5],[9].

In this paper, our first contribution is a new vector-based matrix transpose algorithm that is amenable for direct implementations on contemporary CPU architectures with short-vector instructions extensions. The algorithm enhances the performance of the kernel matrix transpose operation where the matrix data can be placed into the vector registers of the underlying processor. For larger matrix sizes, the blocking approach is applied so that optimized implementations of the kernel operation is easily deployed to enhance the performance of transposing a larger matrix.

The second contribution in this paper is a detailed implementation using the new Intel 256-bit AVX vector instruction extension. We evaluated the performance of our AVX implementation and compared its performance with: (1) a standard serial implementation, (2) a GCC library using an SSE implementation of Eklunde's Algorithm, (3) another variant of Eklundh SSE implementation, and (4) an SSE implementation of our proposed algorithm. The experimental results on Intel Core i7 processor using the AVX instructions demonstrates a 2.83 speedup over the standard sequential implementation, and a maximum of 1.53 speedup over the GCC SSE library implementation. In addition, when the transpose operation is combined with matrix addition to compute the matrix update $B + A^T$, the speedup of our AVX implementation over the sequential algorithm increased to 3.19.

The rest of the paper is organized as follows. Section 2 gives an overview of short-vector instructions, especially Intel AVX instruction set extension. Section 3 presents an overview of the standard matrix transpose algorithm and presents a new parallel algorithm suitable for CPUs with short-vector extensions. Section 4 presents our detailed implementation of the proposed algorithm using AVX instructions together with a comparison of our implementation with other SSE implementations. Section 5 shows the experimental results and the performance evaluation of running our implementations on an Intel Core i7 CPU. Section 6 concludes the paper and outlines future works.

2. SHORT-VECTOR SIMD ISA EXTENSIONS

SIMD (Single Instruction Multiple Data) instructions are extensions to scalar instruction sets of GP-CPUs. These instruction extensions allow the parallel processing of multiple pieces of data which speeds up throughput for many tasks in video encoding and decoding, image processing, data analysis, physics simulations, to name a few. The set of instructions added to the processor micro-architecture supports the following operations: memory data movement, arithmetic and logical, comparison, conversion, permutation, state management, and cacheability control.

2.1 AVX instructions

The AVX instructions are introduced in the Intel 64-bit Sandy Bridge processors to extend the capabilities of the former 128-bit SSE instructions in previous processors to 256-bit allowing for performance increase.

The AVX instruction extension has sixteen 256-bit registers named YMM0 .. YMM15 which can hold thirty two 8-bit integers, sixteen 16-bit integers, eight 32-bit integers, four 64-bit integers, eight 32-bit floating-point numbers, or four 64-bit floating-point numbers. Each YMM register is logically viewed as two lanes with 128-bit each. In order to use an AVX instruction, data elements must be first loaded into appropriate AVX registers. Then, apply an AVX instruction on the vector registers. Finally, the output in AVX registers are stored back to memory if the data is no longer required.

To use the AVX instructions, one can directly work with assembly language programming or use compiler intrinsics inside traditional programming languages such as C/C++. In our implementations in Section 4, we used the latter approach due to its simplicity. Now, we present some C language intrinsics that are used in our implementation. The reader is encouraged to check [4] for more details on intrinsics programming using Intel AVX extension.

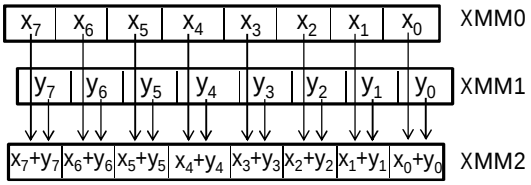


Figure 1. The eight-element single precision floating-point vectors *x* and *y* are loaded into two 256-bit vector registers YMM0 and YMM1. One AVX assembly instruction "VADDPS YMM2, YMM0, YMM1" is applied to add their corresponding elements in parallel to produce the result in register YMM2.

Arithmetic instructions. Figure 1 shows the addition of two 256-bit vector registers YMM0 and YMM1 holding eight single precision floating-point numbers. Using one C intrinsic _mm256_add_ps(), one AVX hardware instruction is applied to add the eight elements in parallel. The result eight-element vector is held in register YMM2 as shown in the figure. The single precision floating point multiplication intrinsic _mm256_mul_ps() multiplies the contents of two vector registers and stores the result into a third one. The other arithmetic instructions such as division, subtraction, ...etc, are treated similarly.

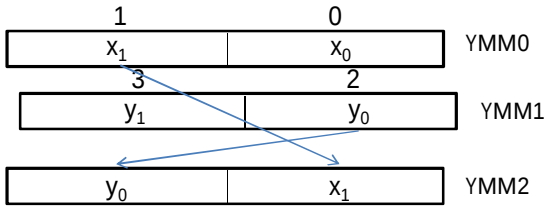


Figure 2. The eight-element single precision floating-point vectors *x* and *y* are loaded into two 256-bit vector registers YMM0 and YMM1. One AVX assembly instruction "VPERM2F128 YMM2, YMM0,YMM1,0X21" is applied to move the second 128-bit of YMM0 to the first 128-bit of YMM2, and the first 128-bit of YMM1 to the second 128-bit of YMM2. This order is specified in the immediate operand 0X21 where 1 is the second half index of YMM0 and 2 is the first half index of YMM1.

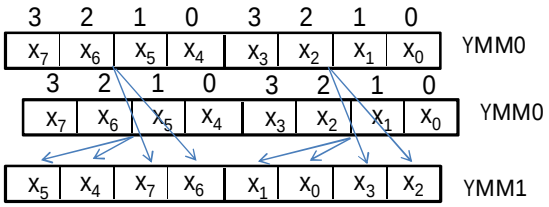


Figure 3. The AVX assembly instruction "VSHUFPS YMM1, YMM0, YMM0, 0X4E" is applied to shuffle, circular shift, both 128-bit lanes of register YMM0 by two elements to the right. The immediate operand 0X4E specifies the order of the four elements in each lane. Register YMM1contains the result after shuffling.

Data permutation instructions. There are a number of AVX instructions for rearranging data inside the vector registers based on a control constant field used to select the parts of the registers that will be collected in the destination register. We discuss here two types of permutations which are used in our implementations.

The _mm256_permute2f128() intrinsic permutes 128-bit between two input registers using an immediate control constant, and stores the result in an output register. Figure 2 shows an example of permuting the two 128-bit lanes of registers YMM0 and YMM1 to produce YMM2. Any 128-bit permutations can be obtained by adjusting the appropriate immediate filed in the instruction/intrinsic.

The _mm256_shuffle_ps() intrinsic performs the same permutation to the floating-point numbers in each YMM lane. Figure 3 shows how the elements in the two lanes of register YMM0 is circularly shifted to the right by one element. This type of data shuffling is used in our proposed matrix transpose algorithm.

3. MATRIX TRANSPOSE ALGORITHM

In this section, we present a new vector-based matrix transpose algorithm which is suited for direct implementation using short-vector extensions to contemporary CPU instruction sets.

In the matrix transpose operation, each row of an n x n matrix X is converted to a column so that $x_{ij} = x_{ji}$. The standard sequential operation is performed as in the following algorithm.

SEQUENTIAL ALGORITHM:

```
01. for  i = 0 to n-1
02.       for  j = i+1 to n-1
03.            temp =  x_ij
04.            x_ij  = x_ji
05.            x_ji  = temp
06.       end for
07. end for
```

3.1 Vector-based Algorithm

Here, we present a new algorithm for the kernel n x n matrix transpose assuming that the size of each vector register is n. For matrices with larger sizes, the blocking approach can be employed. Let each row of matrix X is denoted by X_i where i = 0,1,...,n-1 is the row index. The unit vectors e_i are the rows of the Identity matrix where i = 0,1,...,n-1 is the row index. The rows of the output

matrix $T = X^T$, X_i where i = 0,1,...,n-1. The following parallel algorithm describes how the transpose of matrix X is performed.

PARALLEL ALGORITHM:

01. Load X_i, $i = 0,1, ..., n-1$ into vector registers
02. Set e_i, $i = 0,1, ..., n-1$ into vector registers
03. **for** $i = 0,1, ..., n-1$
04. **for** $j = 0,1, ..., n-1$
05. Element-wise multiply vectors X_j , e_j
06. Accumulate result into vector T_j
07. Shuffle vector X_j
08. **end for**
09. Set $T_j = T_{j\oplus 1}$, $j=0,1, ..., n-1$
10. **end for**
11. Store T_i, $i = 0,1, ..., n-1$ into memory

Initially, the rows of the input matrix, X_i, i = 0, 1,..., n-1, are loaded into vector registers; each row is loaded into a separate register. The unit vectors, e_i, i = 0, 1,..., n-1, are pre-set into other vector registers. Figure 4 is an n = 4 example showing the alignment of the matrix rows X_i and the unit vectors e_i inside the vector registers. The output rows T_i are zeroed inside other four vector registers, which are not shown in the figure. Note that the order of the vector register elements are from right to left, as the actual index of the elements of the vector registers.

In the inner-loop at Line 04, for each pair of vector registers X_i and e_i: (1) perform element-wise multiplication followed by (2) element-wise accumulation of the results into output vector register T_i, then, (3) the vectors X_i are shuffled or circularly shifted one element to the right. Before starting the next iteration of the outer-loop at Line 03, vectors T_i are renamed as shown in Line 09 where the operator $\oplus$ is for addition modulo n. The outer-loop is repeated n iterations to get the final transpose of X. At the end, the output vector registers are stored back into memory.

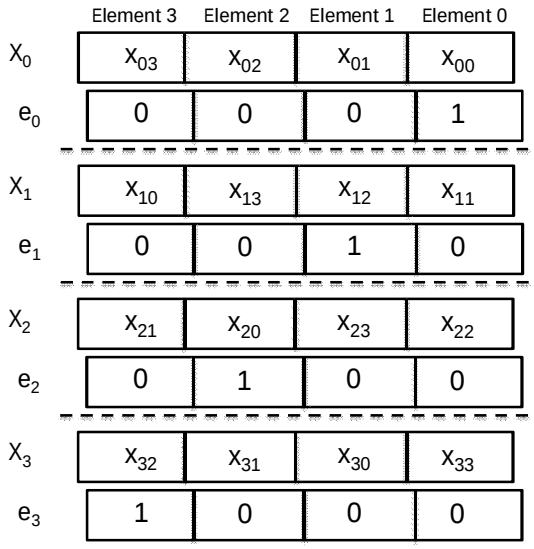


Figure 4. The four rows X_i, i = 0,1,2,3 of the 4x4 single precision floating-point matrix X, and the corresponding unit vectors e_i, i = 0, 1, 2, 3 preloaded into four vector registers.

4. IMPLEMENTATION USING INTEL AVX INSTRUCTIONS

In this section, we present our implementation of the proposed parallel algorithm given in Section 3 using AVX instructions. We consider the kernel 4x4 matrix transpose of single precision floating-point numbers that is pervasive in many graphics and multimedia applications. Larger matrix sizes are treated by partitioning the matrix into 4x4 blocks and using our optimized implementation on each individual block.

The code listing in Figure 5 shows our implementation of the proposed parallel matrix transpose algorithm using C language compiler intrinsics. The transpose of the 4x4 matrix is done in four iterations. The listing shows the first and second iterations only where the third and fourth iterations are processed in the same way. Since the matrix row size is 128-bit, i.e., four float numbers, and the AVX registers are 256-bit wide, then two 4x1 rows of the input matrix X are accommodated in one 256-bit AVX register. Therefore, one 256-bit AVX instruction can be applied to get the result of the element-wise multiply of two matrix rows X_i and X_j with the corresponding two unit vectors e_i and e_j. The result matrix is then accumulated in two output vectors T_i and T_j where each output vector in one AVX register. In addition, the shuffling of two rows of matrix X two elements to the right is processed using one AVX instruction.

```
// Iteration 1
 t12= _mm256_mul_ps(t4,t0);
 t4=_mm256_shuffle_ps(t4,t4,0x4E);
 t8=_mm256_add_ps(t8,t12);
 t13=_mm256_mul_ps(t5,t1);
 t5=_mm256_shuffle_ps(t5,t5,0x4E);
 t9=_mm256_add_ps(t9,t13);
// Iteration 2
 t12=_mm256_mul_ps(t4,t0);
 t4=_mm256_shuffle_ps(t4,t4,0x93);
 t9=_mm256_add_ps(t9,t12);
 t13=_mm256_mul_ps(t5,t1);
 t5=_mm256_shuffle_ps(t5,t5,0x93);
 t8=_mm256_add_ps(t8,t13);
// Intermediate Permute
 t10=_mm256_permute2f128_ps(t8,t9,0x21);
 t11=_mm256_permute2f128_ps(t9,t8,0x21);
```

Figure 5. The first two iterations of our implementation of the proposed matrix transpose algorithm using C language intrinsics of the AVX instructions, n =4.

The order of the elements after the shuffle operation is determined by the hexadecimal immediate pattern 4E that is applied to both 128-bit lanes of the AVX register. The final result of the matrix transpose is obtained after four iterations (n = 4) o f the algorithm. The last two lines in the code listing represent an intermediate step required to permute result vectors before the third and fourth iterations, this is the same permutation explained earlier in Figure 2. Note that the intrinsics are applied using variables of type 256-bit introduced in AVX header files. The vector t4 contains the first and second rows of matrix X. The first row is loaded in the first lane of a 256-bit register and the second row in the second lane. The vector t5 contains the third and fourth rows. Vectors t8 and t9 contains the four output rows of matrix $T = X^T$, which are permuted and moved to t10 and t11 at the end of iteration two.

Figure 6 shows the layout of input matrix X and output matrix T inside the AVX registers at iterations 1,2,3, and 4 of our parallel algorithm. The same sequence of instructions is applied to the four iterations. Note that, the two output AVX registers, shaded in the figure, holding the rows of the transposed matrix T are permuted at the end of iteration 2 so that data is properly aligned for the rest of iterations. Note that, after the last iteration of the algorithm, a similar permutation between the output registers needs to be applied if we demand the rows of the transposed matrix T in the usual rows order. In other words, at the end of iteration four, the contents of the first output register as shown in the figure is $[T_0, T_3]$ and the second register is $[T_2, T_1]$. After the permutation they should become $[T_1, T_0]$ and $[T_3, T_2]$, respectively.

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed parallel transposed algorithm and its implementation using the AVX instruction set extension. Also, we compare the performance of the AVX implementation with the standard sequential algorithm and other SSE implementations.

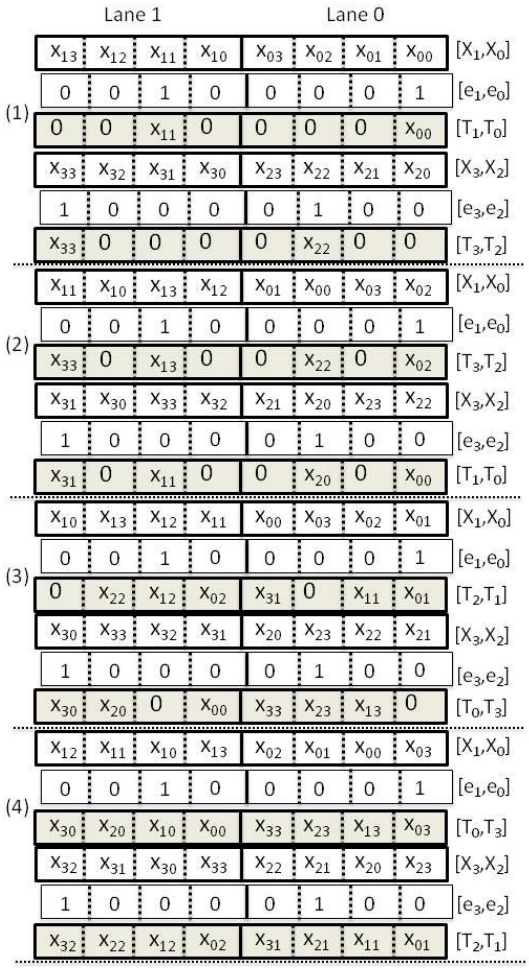


Figure 6. The layout of rows *Xi* and the unit vectors e_i, i = 0,1,2,3 inside 256-bit AVX registers before starting each iteration. The shaded registers are the output vector T_i, i = 0,1,2,3 at the end of each iteration. The numbers to the left of the figure indicate the iteration number. For each AVX register, bits 0-127 are in Lane 0, and bits 128-255 are in Lane 1.

5.1 Experimental Setup

In our implementation of the parallel matrix transpose algorithm, we used the AVX compiler intrinsics for the GNU GCC 4.6.1 compiler. We tested our programs on a Dell notebook with Intel Core i7 2670QM CPU at 2.20 GHz, and running the operating system Ubuntu Linux 12.04 64-bit. The code is tested using one core of the four available physical cores of the Core i7 processor since we are optimizing the 4x4 matrix transpose kernel which is a basic block in a whole application. To hide the loop overhead, we unrolled the outer and inner loops in the proposed parallel algorithm. In order to achieve the maximum parallelism during execution on the Core i7, the AVX instructions from different iterations are interleaved in execution. We also applied the -O2 optimization flag in all the tested codes in order to invoke other code optimizations by the gcc compiler. We implemented the sequential matrix transpose algorithm and used the same optimization options to get a fair comparison. We also implemented an SSE version of our algorithm and a GCC SSE implementation used in the built-in intrinsic _mm_transpose4_ps() in order to compare them with our AVX implementation.

The run times reported measures the execution times of all implemented algorithms without counting the time of loading the input matrix into the vector registers and storing the result back to memory. The execution time for each algorithm is an average of hundred runs.

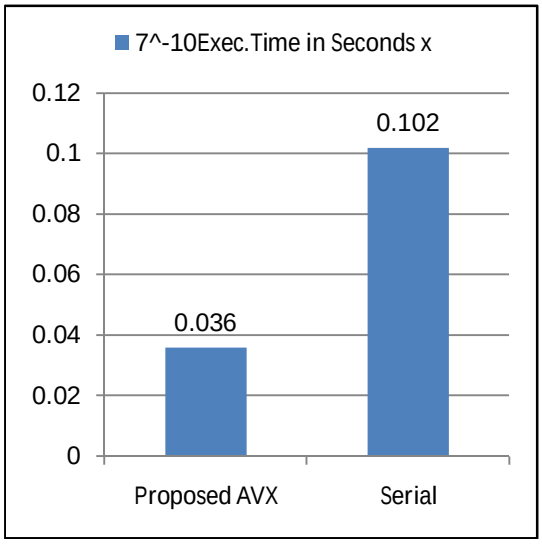


Figure 7. The performance of our AVX parallel implementation of the 4x4 single-precision floating point matrix transpose, A^T, compared to the sequential implementation.

5.2 Results

For comparing the results, we calculated the speedup which is the ratio of the execution time of the sequential or traditional algorithm to the execution time of the parallel or enhanced algorithm. Figure 7 the performance comparison between our AVX implementation and the standard sequential matrix transpose algorithm. Figure 8 compares the performance of our AVX implementation with the SSE implementations of our algorithm and of the GCC built-in code. The execution time is measured in seconds x 10^{-7}.

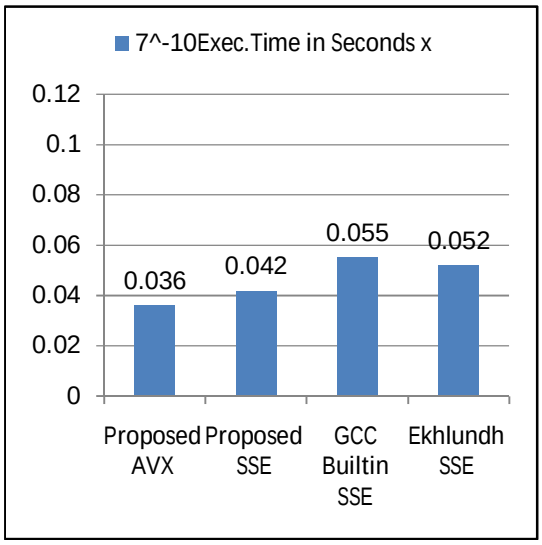


Figure 8. Comparing the performance of our AVX implementation of the 4x4 single-precision floating point matrix transpose, A^T, to the SSE implementations of our algorithm, GCC built-in library, and Eklundh's algorithm.

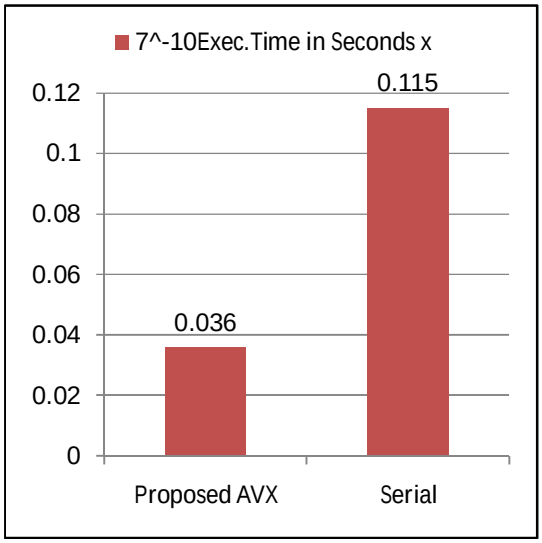


Figure 9. The performance of our AVX parallel implementation of the 4x4 single-precision floating point matrix transpose, $B + A^T$, compared to the sequential implementation.

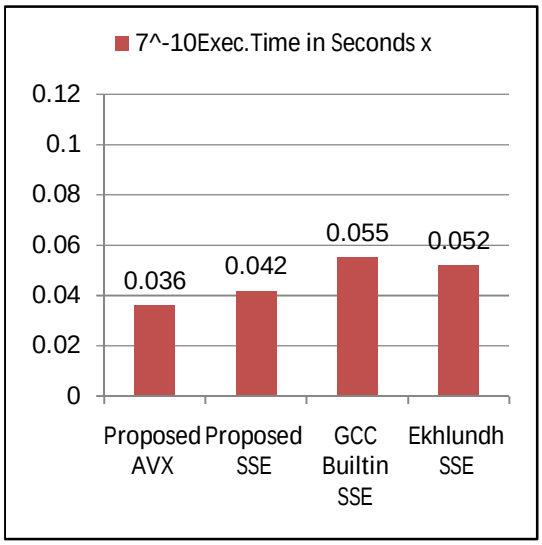


Figure 10. Comparing the performance of our AVX implementation of the 4x4 single-precision floating point matrix transpose, $B + A^T$, to the SSE implementations of our algorithm, GCC built-in library, and Eklundh's algorithm.

Table 1. The speedup calculated for all four implementations in relation to the serial implementation of the 4x4 floating-point matrix transpose A^T and the update $B + A^T$.

Operation	**Proposed AVX**	**Proposed SSE**	**GCC Built-in SSE**	**Eklundh SSE**
A^T	2.83	2.40	1.84	1.96
$B + A^T$	3.19	2.74	2.09	2.21

The displayed results on Table 1 shows a 2.83 speedup over the standard sequential implementation, and 1.54 speedup over the GCC built-in implementation. Our proposed algorithm can perform a matrix addition to the transposed matrix without any additional processing. That is, the matrix update $B + A^T$ does not require any modification to our proposed algorithm. Only matrix B is loaded into the output registers which store the transpose of matrix A. Table 1 also shows our timing results after updating the AVX and SSE implementations. We can see that an enhanced speedup of 3.19 of the AVX implementation over the sequential algorithm is obtained, see Figure 9. Meantime, all other SSE implementations behave the same as our AVX implementation. This is because the AVX and SSE implementations exploited the underlying architecture of the processor so that the additional floating-point add instructions are overlapped in execution with the shuffle and permute instructions. This can be seen from comparing the timing results of Figure 10 and the previous figure, Figure 9 where no reductions in the execution time of the SSE implementations. However, the add instructions will increase the size of the SSE implementations of Eklundh's variants.

7. CONCLUSIONS

The AVX short-vector instructions are 256-bit vector extensions to general-purpose instruction sets to accelerate the execution of the data-parallel computations in scientific, engineering, and signal processing applications. In this paper, we have presented a new vector-based matrix transpose algorithm amenable to implementations on modern GP-CPUs that constitute the core execution engine of high-performance distributed and parallel systems. We have concentrated on

enhancing the performance of kernel operations that have direct impact on enhancing the overall performance of whole applications that require the transpose of large size matrices.

We have presented a detailed implementation of the kernel 4x4 single precision floating-point matrix transpose which is a pervasive kernel in graphics, multimedia, and image/signal processing. To the best of our knowledge, we didn't see any implementation of the 4x4 matrix transpose using AVX short vector instructions. Therefore, our implementation may fill a gap when transforming old 128-bit SSE implementations of matrix and vector computations into newer 256-bit AVX implementations and even future extensions to 512- and 1024-bit.

To transpose a larger size matrix, the blocking approach can be applied so that the input matrix is partitioned into small kernel blocked that fit into the vector registers. Hence, optimized kernel such as the one we presented in this paper can be employed.

Our algorithm is scalable and doesn't require any modification if implemented in future vector extensions with vector register sizes more than 256-bit. This scalability is based on the architectural support of a data shuffle instruction that cyclically shift the register elements.

We evaluated the performance of our AVX implementation on a Core i7 processor, and our results showed a speedup of 2.83 over the standard sequential implementation. Also, our implementation showed a maximum speedup of 1.53 over a GCC library implementation that use SSE instructions. When the transpose operation is combined with the matrix addition, $B + A^T$, the speedup over the sequential algorithm increased to 3.19 without any modifications to the original transpose algorithm. These results demonstrate the merit of using the 256-bit AVX instruction extension over the 128-bit SSE extension for kernel matrix transpose, and other matrix kernels as well.

Our next ongoing research is focused on linear algebra and image processing applications where we can apply our implementations of the kernel matrix transpose for enhancing these applications on multi-core processors.

REFERENCES

[1] Choi J., Dongarra J. and Walker D.,"Parallel matrix transpose algorithms on distributed memory concurrent computers." Parallel Computing, 21(9):1387-1405, 1995.

[2] Eklundh J., "A fast computer method for matrix transposing." IEEE Trans. Computing, 21(7):801-803, July 1972.

[3] Free Software Foundation, Inc., "GCC implementation of the SSE header file."http://opensource.apple.com/source/gcc/gcc-1762/gcc/config/i386/xmmintrin.h

[4] Intel Corporation. "Intel ® Advanced Vector Extensions Programming Reference." Number 319433-010. Apr 2011.

[5] Intel Corporation."Intel ® 64 and IA-32 Architectures Optimization Reference Manual, Volume A." Number 327268-026. April 2012.

[6] Kaufmann M., Meyer U. and Sibeyn J., "Matrix transpose on meshes: Theory and practice." In IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing, pages 315-319, 1997.

[7] Kaushik S. et. al.,"Efficient transposition algorithms for large matrices." In Supercomputing '93:Proceedings of the 1993 ACM/IEEE conference on Supercomputing, pages 656-665, New York, NY, USA, 1993.

[8] Krishnamoorthy S. et al., "Efficient parallel out-of-core matrix transposition." International Journal of High Performance Computing and Networking, 2(2/3/4), 2004.

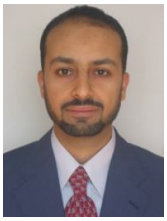
[9] Nenakhov S., "AVX register transpose."http://www.kevinstock.org/2011/03/avx-register-transpose.html, March 2011.

[10] Petkov N., Systolic Parallel Processing. Elsevier Science Inc., New York, NY, USA, 1992.

[11] Slingerland N. and Smith A., "Multimedia extensions for general purpose microprocessors: a survey." Microprocessors and Microsystems, 29(5):225-246, 2005.
[12] Suh J. and Prasanna V., "An efficient algorithm for out-of-core matrix transposition." IEEE Trans. Computing, 51(4), 2002.
[13] Tsay J., Ding K. and WangW., "Optimal algorithm for matrix transpose on wormhole-switched meshes." J. Inf. Sci. Eng., 19(1):167-177, 2003.
[14] Zekri A. and Sedukhin S., "Matrix transpose on 2D torus array processor." In The 6th IEEE International Conference on Computer and Information Technology, page 45, Seoul, Korea, September 2006.

Authors

Ahmed S. Zekri



Received both B.Sc. and M.Sc. in computer science from Department of Mathematics & Computer Science, Faculty of Science, Alexandria University. He has a Ph.D. degree in computer science from The University of Aizu, Japan 2008. Presently he is working as Assistant Professor at Alexandria University and Beirut Arab University. His research interests include parallel and distributed computing, image processing, and scientific computing.