Linear-time Matrix Transpose Algorithms Using Vector Register File With Diagonal Registers*

Bedros Hanounik Xiaobo (Sharon) Hu †

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
bhanouni@nd.edu, shu@cse.nd.edu

Abstract

Matrix transpose operation (MT) is used frequently in many multimedia and high performance applications. Therefore, using a faster MT operation results in a shorter execution time of these applications. In this paper, we propose two new MT algorithms. The algorithms exploit diagonal register properties to achieve a linear-time execution of MT operation using vector processor that supports diagonal registers. We demonstrate the algorithms as well as proofs, examples, and various enhancements to the proposed algorithms. A performance evaluation shows that the proposed algorithms are at least twice as fast as one of the leading MT algorithms such as an algorithm that is implemented using Motorola's AltiVec architecture ($n \geq 16$). We believe that our work opens new doors to improve the execution time of many two-dimensional operations such as DCT, DFT, and Shearsort.

1. Introduction

The execution time of matrix transpose operation (MT) is crucial to many applications such as multimedia and scientific computing. Such applications are becoming popular, and their performance in a given architecture is one of the factors that determines the success of that architecture. Therefore, modern processors are incorporating vector units and special instruction set in order to speedup critical operations such as MT. Examples are, Intel's MMX extension [8], Motorola's AltiVec architecture [10], and MIPS' shuffle instructions [11]. All these extensions are very similar when it comes to execute MT. They all execute MT of a matrix of size $n \times n$ in $O(nlogn)$ vector step using shuffle and merge instructions.

The MT operation, like any two-dimensional operation, accesses data elements in rows as well as in columns. Hence, to accelerate such an operation, the architecture of a vector processor (or a vector processing unit) should allow instant access to data in both dimensions of the vector register file (VRF). Since an n-way SIMD (Single Instruction, Multiple Data) machine (or a vector processor) manipulates n data elements concurrently , in a single instruction, an operation (such as MT) which takes n^2 scalar step would take linear number of vector steps; given that the SIMD machine allows accessing data in both dimensions of the VRF. The conventional organization of VRF allows accessing data elements, residing in one row, concurrently. This same organization restricts all other accessing patterns of data in conventional vector processors, which results in an inefficient execution of MT operation.

The efficiency of an algorithm is greatly affected by the resources provided by an architecture which is used to execute the algorithm. One example is the vector processing concept, which accelerate applications that manipulate multiple data concurrently. This concept has a limitation of accessing only data residing in one row. Therefore, to accelerate two-dimensional operations, the conventional organization of VRF should be improved to allow two-dimensional access to its contents. Our contribution is to demonstrate how a two-dimensional access to VRF is possible, and to present new sophisticated algorithms that benefit from such an access to accelerate frequently used operations such as matrix transpose operation.

In this paper, we present two new algorithms which exploit the properties of diagonal registers to execute matrix transpose in a linear number of vector steps (instructions).

*This research is supported in part by an External Research Program Grant from Hewlett-Packard Laboratories, Bristol, England and by NSF under grant number MIP-9701416.

†The authors would like to thank Peter M. Kogge and Jay B. Brockman for suggesting the matrix transpose problem.

In Section 2, we discuss related work. Section 3 discusses the concept of diagonal registers and their properties. Section 4 discusses the new proposed algorithms. Evaluation of diagonal registers cost and the performance of the algorithms are discussed in Section 5. The conclusion is in Section 6.

2. Related work

The fastest method to transpose a matrix is to use direct paths between every pair of elements being swapped. In such an implementation, MT operation is done in one step; assuming all elements are swapped concurrently. Despite the theoretical speed of this implementation, it is not practical to build a device that supports it. The data routing overhead is the main hurdle that prevents such an implementation from being a reality.

The next fastest method to transpose a matrix is to transpose one column and one row every step, transposing n elements every step, hence, resulting in a time complexity of $O(n)$ step (for a matrix of size $n \times n$). Many designers exploit this concept to build a special device that achieves linear-time matrix transpose. Such as *Systolic Arrays for Matrix Transpose* [6] and *Universal Architecture for Matrix Transposition* [7]. These architectures are especially built to execute MT only. Adding such a special hardware device that only executes MT to a general purpose computer system is a waste of resources, because the silicon area occupied by such a device is not used for any purpose other than MT. More details about specialized hardware to achieve fast MT are discussed in [5].

The above mentioned methods are theoretically fast. Nevertheless, they are neither practical, nor efficient for a general purpose computer system. An alternative method to achieve fast MT is to use existing general purpose computer architecture with an extension that allows fast execution of MT. Such an approach, inspired by Eklundh's algorithm [3] (see Figure 1), is used by major computer designers to accelerate multimedia applications. The time complexity of Eklundh's algorithm and its derivatives is $O(nlogn)$ vector (or SIMD) step to execute an MT of a matrix of size $n \times n$. The derivatives of Eklundh's algorithm share the same basic concept and vary in the order and method of execution of the transpose. They are explained in more details in documentations from Intel [2], Motorola [10], MIPS [11], and many others (refer to [5] for more details).

In addition to the above mentioned methods, there are two other architectures that are designed especially to accelerate multimedia and similar two-dimensional applications. The first architecture is *Imagine Architecture* [9], which is capable of executing basic multimedia kernels in a pipelined fashion which results in a shorter execution time. The second architecture is *Memory Skewing* concept, which is described in detail in [1]. The main goal of using memory skewing is to allow accessing both columns and rows of a two-dimensional array instantly. Hence, accelerating two-dimensional operations such as MT. However, the memory skewing concept has a disadvantage of requiring data shuffling for every access to the memory; thus, slowing down every memory access, which in turn slows down the whole system significantly. The two mentioned architectures are non-traditional architectures, in a sense that they require fair amount of modifications to the hardware, operating system, and applications of a vector computer system that supports them. Where as the proposed architecture requires minimal changes to the current vector architecture.

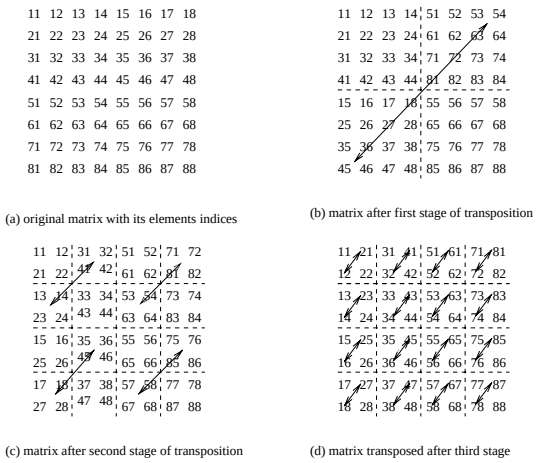

Figure 1. Eklundh's matrix transpose algorithm

To summarize, existing designs for accomplishing fast MT either are too costly to be incorporated in a general purpose computer system, are not as fast as desired, or add unnecessary complexity. We present in this paper an alternative to improve current architectures in order to build an architecture which is more suitable for two-dimensional applications than the existing approaches.

3. New vector register file organization

Vector architectures are essential for obtaining high performance. As a result, many processors incorporate vector extensions in their design as in Pentium [8] and PowerPC [10] architectures. The heart of vector architectures is the vector register file (VRF) whose structure uses bitlines to access the data. This approach prohibits columns from being accessed instantly. On the other hand, many multimedia and high performance applications require handling two-dimensional data frequently. However, existing VRF designs make it hard to keep the original structure and at

the same time access columns and rows in VRF. Therefore, it is desirable to modify VRF design in order to accelerate two-dimensional applications.

A novel VRF design is proposed. The goal is to extend the VRF access pattern while maintaining the basic structure of VRF. In addition to accessing the data in one row in parallel as it is done in a traditional VRF. The data along the diagonal direction of the VRF can also be accessed in parallel. This approach is very similar to that of memory skewing [1], but it reduces the complexity of maintaining the skewed data by designing more sophisticated algorithms that access non-skewed data.

Our design transforms SIMD machines from being one dimensional, as it is traditionally designed, to being new two-dimensional SIMD machines. In the new VRF design, two diagonal patterns, diagonal-down and diagonal-up are introduced. Thus, a VRF now contains three types of registers: *Row registers, diagonal-down registers (DL)*, and *diagonal-up registers (DH)*. Row registers access data in VRF in rows and are referred to as RR. Diagonal registers access data in diagonals (up or down) and are referred to as DR.

3.1. Diagonal-down registers DL

Diagonal-down (DL) registers are illustrated in Figure 2. Every DL register has a different fill pattern. For the sake of simplicity, not all the registers are shown. The first DL register extends from the top left corner of VRF to the bottom right corner of VRF and represents the main diagonal of VRF, as shown in Figure 2. All other DL registers wrap around the VRF, such that when a DL register reaches the lower edge of VRF, it wraps around VRF and continues from the other edge. Thus, the number and the width of DL registers are the same as those of row registers.

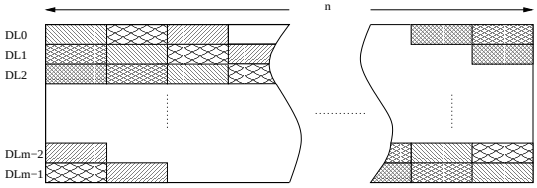

Figure 2. Vector register file accessed using diagonal registers DL

3.2. Diagonal-up registers DH

Diagonal-up (DH) registers are another pattern of accessing VRF in addition to row and DL registers. This pattern is illustrated in Figure 3. As in a DL register, every DH register has the same width as row registers RR. The DH registers extend from the bottom left corner of VRF to the top right corner of VRF, and wrap around VRF when needed.

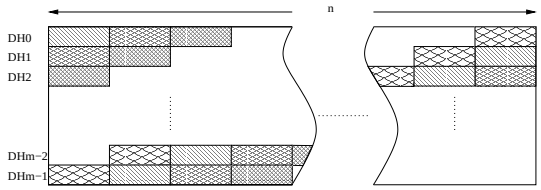

Figure 3. Vector register file accessed using diagonal registers DH

3.3. Properties of diagonal registers

DH and DL registers share similar properties. In fact they are duals. When the base index of a two-dimensional array (VRF) is at the top-left corner of the array, DL registers represent the forward (main) diagonals and DH registers represent the backward diagonals. On the other hand, if the base index is at the top-right corner of the array, DH registers represent the forward diagonals, and DL registers represent the backward diagonals.

The dual property of the diagonal registers becomes more apparent by examining the relations between DL, DH, and RR registers (shown in Table 1). The table illustrates how the indices of a data element are transformed when the same element is accessed using different types of registers. Its derivation based on the following notation definitions:

- The indices of the row and the column in VRF are i and j, respectively.
- The base of the two-dimensional array is at the top left corner of VRF (i.e. $i = 0$ and $j = 0$ at that point).
- The indices i and j in a formula are the row (or diagonal) and column indices before applying that formula.
- R_0, DL_0, and DH_0 are the registers before transformation.
- All the fields in the table are $MOD\ m$, where m is the width of the VRF.
- For negative values (as in $-n$) the resultant value is $(2m - n)\ MOD\ m$.

To show how to use the table, the following example is presented. Given the indices of an element in DL register (say $DL(2, 5)$), from Table 1, we know that $DL_0(i, j)$ maps to $DH(i + 2j, j)$. Hence, the indices of $DL(2, 5)$ are $DH(12, 5)$ in DH register space.

$R_0(i,j)$	$DL_0(i,j)$	$DH_0(i,j)$
$R(i,j)$	$R(i+j,j)$	$R(i-j,j)$
$DL(i-j,j)$	$DL(i,j)$	$DL(i-2j,j)$
$DH(i+j,j)$	$DH(i+2j,j)$	$DH(i,j)$

Table 1. The formulas describing the relations between diagonal and row registers

Table 1 is very useful, as it will be used to prove the algorithms presented in the next section. Moreover, new algorithms can be derived based on the information in the table by knowing the initial positions and the final positions of the data elements in an array.

Diagonal registers add a simple, yet powerful, extension to SIMD machines. Any instruction operates on the row registers can operate on the diagonal registers, which means no additional instructions are added to the instruction set architecture (ISA) in order to use diagonal registers extension. Herewith, the diagonal registers extend SIMD machines parallelism to the other dimension of two-dimensional data. A SIMD instruction can now operate on the second dimension of the data in VRF. The introduction of diagonal registers can greatly improve the execution time of those applications requiring data access along both row and column directions.

4. The new proposed matrix transpose algorithms

To demonstrate the advantage of the new vector register file design, we have derived efficient algorithms for MT. In this section, we present these algorithms as well as proves, examples, and various enhancements. For simplicity, only matrix size of $n \times n$ is discussed. Other sizes can be considered with slight modifications to the algorithms presented here.

Transposing a matrix of size $n \times n$ by moving only one data element in each step results in a time complexity of $O(n^2)$ scalar operation. SIMD machines are capable of moving n elements in one step (n scalar operation per one vector operation). Hence, an MT algorithm implemented on a SIMD machine that efficiently utilizes the width of SIMD machine would have a time complexity of $O(n)$ SIMD operation. To achieve overall time complexity of $O(n)$ SIMD operation, an MT algorithm should meet the following:

1. Has a fixed number of stages independent of n.
2. Execute $O(n)$ SIMD operation in every stage.

It seems impossible to find an algorithm with these characteristics, especially knowing that current SIMD machines are one-dimensional machines. However, diagonal registers in our new VRF transform SIMD machines into two-dimensional machines, and an MT algorithm with a linear-time complexity is possible using this extension.

We have derived two *linear-time MT algorithms*. The first algorithm uses load and store operations to align the data in VRF before performing an MT operation and it is called *Load-Store Matrix Transpose (LSMT)*. The second algorithm transposes the data already loaded in VRF and it is called *In-Place Matrix Transpose (IPMT)*.

4.1. Load-store matrix transpose (LSMT)

This algorithm is used to transpose a matrix that fits in VRF and resides in the memory. Note that the MT operation consists of moving data in two dimensions, the vertical and the horizontal dimensions. To explain $LSMT$ algorithm, an MT is illustrated in Figure 4. The numbers are the indices of the data elements in the matrix. Part (a) shows the original matrix before MT is performed. Part(b) shows the same matrix transposed; the data elements (21, 31, 41) are moved horizontally and vertically, as the arrows indicate, after MT operation. Part (c) illustrates the fact that every n data elements in a matrix of size $n \times n$ (4×4 in this example) have the same moving (rotation) value. Moreover, these data elements reside on the diagonal of the matrix, hence, DR can be used to group these elements and meet the conditions stated earlier to achieve linear-time matrix transpose.

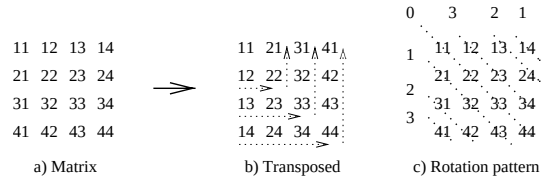

Figure 4. Matrix transpose operation properties

The key to achieve a linear-time matrix transpose, without using any extra temporary space, is to load the elements that have the same rotation value into the same row in VRF, then rotate the rows in linear time, finally, store them back in the memory. The actual algorithm can be summarized by the following three stages.

1. Load row_i from memory into DH_i register, $i = 0, .., n-1$
2. Rotate R_i register to the right by i positions, $i = 1, .., n-1$

3. Store DL_i register back in memory rows 0 to $n-1$, $i = 0, n-1, n-2, ..., 1$

Figure 5, demonstrates the result after each stage of the matrix transpose using $LSMT$. This figure shows how the store operation is performed using DL registers in part (b) of the figure to obtain the resultant MT in part (c) as the arrows indicate.

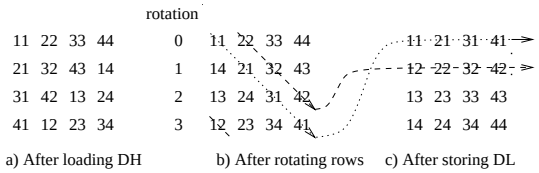

Figure 5. Load-Store matrix transpose algorithm ($LSMT$) using DR

Since the load and store operations are required by all algorithms and are not part of the transpose itself, the number of operations required by this algorithm is only those in the second stage which is equal to $n-1$ steps. When counting the load and store operations, it is $3n-1$ steps. This algorithm is simple to implement, it does not require any temporary storage, and has simple control flow. This is very desirable in modern pipelined processors, because the pipeline gets stalled frequently due to the complex control flow of a program being executed.

A dual version of the $LSMT$ algorithm will be presented next. When $LSMT$ algorithm and its dual run after each other, back to back in a pipelined fashion, the throughput is maximized, and the load and store stages are overlapped between every two matrices. Hence, *VRF can transpose a matrix every* $2n-1$ *steps including steps to load and store the matrix*. Therefore, the transpose requires *zero steps* on average.

The dual of $LSMT$ can be obtained by loading the matrix using DL registers and storing the matrix using DH registers. The dual of $LSMT$ is summarized in Figure 6 and by the following three stages:

1. Load rows 0 to $n-1$ from memory into DL_i register, $i = n-1, n-2, ..., 0$
2. Rotate R_i register to the right $(i+1)$ MOD n positions, $i = 0, .., n-1$
3. Store DH_i register back in memory rows 0 to $n-1$, $i = n-1, 0, 1, .., n-2$

Figure 6 demonstrates the result after each stage of the dual $LSMT$ algorithm. The advantage of having a dual $LSMT$ becomes critical when transposing a matrix that does not fit into VRF and a stream of MT operations of sub matrices is required.

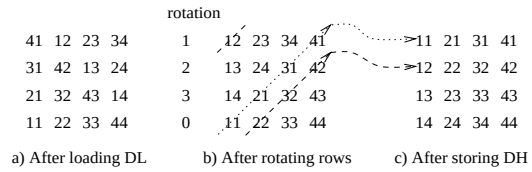

Figure 6. The dual of Load-Store matrix transpose algorithm ($LSMT$) using DR

The correctness of $LSMT$ and its dual algorithm can be verified by manipulating formulas provided earlier in Table 1 in section 3.

Let us verify the correctness of $LSMT$ by applying the formulas in Table 1. We assume the matrix is stored in the memory. Every element of such a matrix can be accessed by $M(i,j)$, where i and j are the row and column indices respectively. Now, the execution of $LSMT$ is:

- In the first stage, every row of the matrix in the memory is loaded into its corresponding DH register. Therefore, $M(i,j)$ is loaded into $DH(i,j)$. From Table 1, $DH(i,j) \implies R(i-j,j)$, where R is the row register set.
- In the second stage, the $(i-j)th$ row register is rotated to the right by $(i-j)$ positions. Therefore, the word that is originally indexed with $R(i-j,j)$ will be indexed with $R(i-j, j+(i-j)) \implies R(i-j,i)$ after the rotation.
- In the third stage, the DL registers are used to index the elements in the VRF. Therefore we transform the indices from R to DL using Table 1. $R(i-j,i) \implies DL(i-j-i,i) \implies DL(-j,i) \implies DL(-j \bmod m, i)$.
- By using the order specified in the third stage, when $j = 0$, DL_0 is stored into 0th row of the matrix in the memory. When $j > 0$, DL_j is stored into $(m-j)$th row of the matrix in the memory. Therefore, Matrix $M(i,j)$ will be transposed into $M^T(j,i)$.

In a similar manner, the dual algorithm of $LSMT$ can be proved. However, due to the page limit, this proof is not presented here.

4.2. In-place matrix transpose (IPMT)

Sometimes MT needs to be performed on data already loaded in VRF. In this case, $LSMT$ algorithm, discussed above, will not work without using memory or temporary storage. Since it is desirable to have a linear-time algorithm without the use of a temporary storage, another linear-time algorithm that performs MT on data already loaded

in VRF is developed and called *In-Place Matrix Transpose* ($IPMT$).

The algorithm requires two stages of $n-1$ SIMD operations each, without the need for load/store operations for alignment of the data. The stages are as follows:

1. Rotate DH_i register to the right i positions, $i = 1, .., n-1$
2. Rotate R_i register to the right $2i$ positions, $i = 1, .., n-1$

In Figure 7, a step-by-step execution of the above algorithm for 4×4 matrix is illustrated. The algorithm takes only two stages of $n-1$ steps each, which makes it a very attractive method to transpose a matrix in linear-time without using *any* temporary storage.

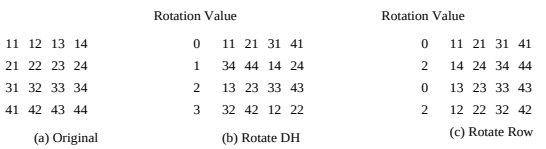

Figure 7. $IPMT$ matrix transpose algorithm using DR

The order of the rows of the transposed matrix after executing $IPMT$ algorithm is not in the right order. This side effect can be ignored since any row in VRF can be accessed directly and the matrix can be read in the right order.

The correctness of the $IPMT$ can be verified in the same manner as that for the $LSMT$ algorithm. The mapping formulas provided in Table 1 are applied to the stages of $IPMT$ and the matrix $M(i,j)$ is transposed into $M^T(j,i)$. The proof is as follows:

- The data elements are stored in VRF and accessed by $R(i,j)$. In the first stage, the access is done using DH registers. Therefore, we transform the indices into DH indices. $R(i,j) = DH(i+j,j)$.
- The $(i+j)$th DH register is rotated to the right by $(i+j)$. Hence, $DH(i+j,j) \Longrightarrow DH(i+j, j+(i+j))$
- The result of first stage is: $R(i,j) \Longrightarrow DH(i+j,j) \Longrightarrow DH(i+j, i+2j)$.
- The data element originally indexed with $R(i,j)$ should be indexed with $R(i+j-(i+2j), i+2j) = R(-j, i+2j)$. This is done by transforming the indices of first stage results into R indices. Rotating the $(-j \bmod m)$th row, in stage two, to the right by $2(-j \bmod m)$ results in $R(-j \bmod m, i+2j+2(-j \bmod m)) \Longrightarrow R(-j \bmod m, i)$.
- After executing $IPMT$, the 0th column of the original matrix will be in 0th row of the transposed matrix. The jth column $(0 < j)$ of the original matrix is located in $(m-j)$th row of the transposed matrix.

5. Experiments and performance evaluation

In this section, we compare our new algorithms with some existing ones. In order to evaluate the actual speed-up resulted from using the proposed VRF and MT algorithms, the overhead of the new extension is discussed first. Then, the proposed MT algorithms are compared with a modern MT implementation in a vector processing unit such as Motorola's AltiVec architecture.

An efficient implementation of VRF with diagonal access plays an important role in garnering the benefits introduced by diagonal registers. The challenge is to keep the added access time as short as possible. A VRF consists of multiple columns, each comprises of a depth of one word and a width of the number of registers in VRF. To allow diagonal access in VRF, the words that reside on the diagonal are selected concurrently. There are two possible implementations. One simple implementation is to add a new dedicated select line for every diagonal register, and the output of the selected cells is multiplexed with the row registers' output. This implementation is call *Added-ports implementation*. Another possible implementation is to have a decoder for every column of VRF. In this implementation, the decoders in different columns select different rows in VRF to access a diagonal register. The latter approach is called *local decoding implementation*.

5.1. The implementation

We only evaluate the added-port implementation, since it requires minimum changes to the traditional VRF and it has the worst delay overhead. Which gives us the worst case scenario to analyze. We only evaluate the time delay of the implementation and leave the power and area analysis for future work. This is because our immediate interest is to find the overall performance improvement of the algorithms and its required architecture (i.e. diagonal registers).

The added-port implementation uses a dedicated select line for every diagonal register (see Figure 8). Two new select lines, DH_i and DL_i, are added to every register. Each cell is also extended by two additional access ports, which are controlled by the corresponding select lines. Since the diagonal access adds only new ports to the register file, the same structure of traditional VRF is used with the exception of having longer select lines for the diagonal registers. To compensate for the larger load added to the select lines, a larger driver circuit should be used.

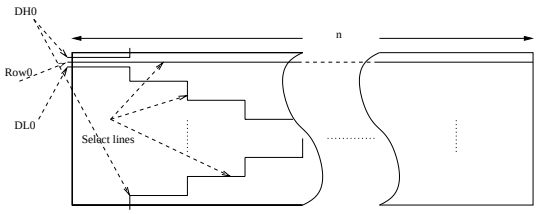

Figure 8. Added-ports implementation of proposed Vector Register File

5.2. Analyzing the simulation data

To evaluate our implementation, we have built an RC network model that represents VRF. The RC values fed into the model are calculated from MOSIS $.25\mu m$ fabrication process. We omit the implementation details due to the limited space. The details of the implementation can be found in [5]. For the register file configurations, we have used the one from a modern processor such as $AMD\ K-7$ [4]. We varied only the number of the bits in VRF; the word (column) size is assumed to be 8-bit wide. To find the number of registers in VRF, we assumed that VRF has a square shape. Which means the number of registers is always equal to the number of words.

Figure 9 shows the access times of traditional VRF (TVRF); the row register in the proposed VRF, and the diagonal register in the proposed VRF. The driver circuit of the word line is kept the same as the one in TVRF. The data is normalized to TVRF access time; the delay overhead of access time of diagonal registers ranges from 12% when VRF is 64-bit wide, to 36% when VRF is 512-bit wide.

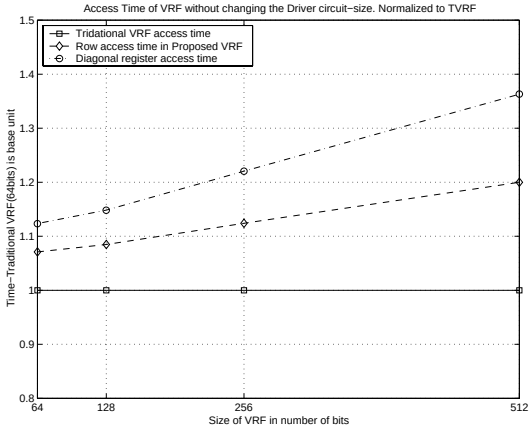

Figure 9. The total access times of traditional VRF and the proposed VRF normalized. The driver of the select line size is not adjusted.

In Figure 10, the driver circuit size is adjusted such that a row register access time of the proposed VRF is equal to TVRF access time. The data is normalized to TVRF access time. The delay overhead of access time of diagonal registers ranges from 5%, when VRF is 64-bit wide, to 16% when VRF is 512-bit wide.

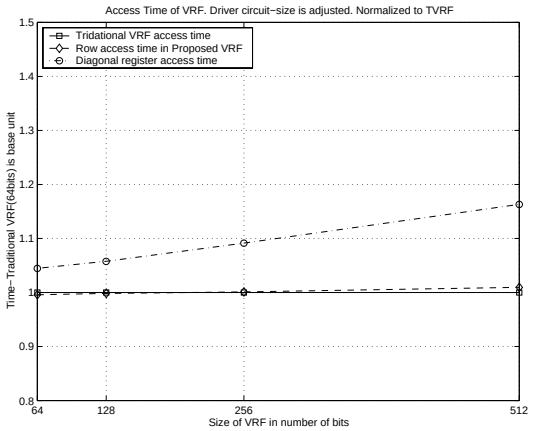

Figure 10. The total access times of traditional VRF and the proposed VRF normalized. The driver of the select line size is adjusted.

When a very large VRF is built (256 bits or more), the overhead is noticeable (about 10%). This is because of the very long select lines with only one driver and without any buffering. Solutions like driving the select line from both ends, or dividing the VRF into two banks with driving circuit in the middle; are common design techniques. They can be applied to our proposed architecture when building a very large VRF. Thus overcoming such a large overhead.

5.3. $IPMT$ algorithm versus AltiVec's algorithm

To assess the performance of $IPMT$ algorithm, it is compared with one of the leading vector architectures such as AltiVec architecture. The steps of both algorithms are dominated by the access time of VRF. Therefore, the cycle time of the processor that uses either architecture is equivalent to VRF access time. As it is shown earlier in this section, the overhead of the proposed extension to VRF is very small, and can be reduced to zero by adding buffers to the select lines in VRF. Hence, the processor cycle time of both architectures are identical in the latter case. The comparison is done using the number of processor cycle each architecture uses to perform MT assuming that the processor is fully pipelined and can issue one instruction every cycle. In the case when there is an overhead delay that lengthens the cycle time, the total time (cycle time $\times$ No. of cycles) is adjusted accordingly. An overhead of 6% is used in the comparison, because that is the overhead obtained from the

simulation data when the size of VRF is 128-bits; same as AltiVec's size.

Table 2 shows the performance of AltiVec's and $IPMT$ algorithms. When replacing n with the actual size of AltiVec's ($n = 16$), *$IPMT$ algorithm executes MT in less than half the time required by AltiVec's algorithm.*

	T. complexity	Time in cycles
AltiVec	$nlogn$	64 AltiVec's cycle
$IPMT$	$2.12(n-1)$	31.8 AltiVec's cycle
$IPMT$ (buffered)	$2n-2$	30 AltiVec's cycle

Table 2. Comparison of AltiVec's and $IPMT$

5.4. $LSMT$ algorithm versus AltiVec's algorithm

In the previous comparison, it is assumed that the matrix fits entirely and already loaded in VRF. Here, the assumption is that the matrix resides in the memory and extra cycles are required to load and store the matrix. All other assumptions in the previous comparison are valid in this comparison. It is also assumed that the memory access time is equivalent to the processor cycle time. This is a valid assumption, because most modern processors utilize memory cache running at the processor clock rate.

Unlike other MT algorithms, $LSMT$ algorithm includes load and store operations to execute MT, hence, we get a more efficient MT operation. The improvement of other MT algorithms fades when load/store operation is considered in the comparison, but because $LSMT$ incorporates load/store into its execution, the speed-up of using the diagonal registers, by means of $LSMT$, to execute MT does not fade.

Table 3 shows the performance of AltiVec's and $LSMT$ algorithms. The speed-up of $LSMT$ compared with AltiVec's is similar to that of $IPMT$. *$LSMT$ is also twice as fast as AltiVec's* when $n = 16$.

	T. complexity	Time in cycles
AltiVec + L/S	$2n + nlogn$	96 AltiVec's cycle
$LSMT$	$3.18(n-1)$	47.7 AltiVec's cycle
$LSMT$ (buffered)	$3n-1$	47 AltiVec's cycle

Table 3. Comparison of AltiVec's and $LSMT$

6. Conclusions

We have presented two algorithms that use the diagonal registers concept to achieve linear-time execution of matrix transpose operation. The properties of the diagonal registers are discussed in order to facilitate the introduction of our new algorithms. Then, an overall performance evaluation of the proposed algorithms along with the diagonal registers architecture is presented. Our evaluation shows that the new algorithms are at least twice as fast as one of the leading matrix transpose algorithms executed using Motorola's AltiVec architecture when the matrix size is $n \geq 16$. We have shown that the diagonal registers and the new proposed algorithms demonstrate the fact that it is possible to execute two-dimensional operations efficiently. We believe that our work opens a new door to many two-dimensional applications to achieve an efficient execution time. Such as Shearsort, 2D-DCT, 2D-IDCT, 2D-DFT, and image filtering.

References

[1] Budnik, P. and Kuck, D. J. The organization and use of parallel memories. *IEEE transactions on computers*, Dec. 1971.

[2] Dulong, Carole et al. Method for transposing a two-dimensional array. *US. patent 5,815,421*, (Intel Corp.), Sep. 1998.

[3] Eklundh, J. O. A fast computer method for matrix transposing. *IEEE transactions on computers*, 21:801–803, July 1972.

[4] Golden, Michael et al. A seventh-generation x86 microprocessor. *IEEE journal of solid-state circuits*, 34(11):1466–1477, Nov. 1999.

[5] Hanounik, Bedros. Diagonal registers: novel vector register file design for high performance and multimedia computing. *University of Notre Dame, CSE Dept. technical report TR11-00*, Aug. 2000.

[6] O'Leary, Dianne P. Systolic arrays for matrix transpose and other reorderings. *IEEE transactions on computers*, C-36(1), Jan 1987.

[7] Panchanathan, S. Universal architecture for matrix transposition. *IEE proceedings E computers and digital techniques*, Sep. 1992.

[8] Peleg, Alex and Weiser, URI. MMX technology extension to the intel architecture. *IEEE Micro*, pages 42–50, Aug. 1996.

[9] Rixner, Scott et al. A bandwidth-efficient architecture for media processing. *IEEE international symposium on microarchitecture*, pages 3–13, 1998.

[10] Tyler, Jon et al. Altivec: bringing vector technology to the powerpc processor family. *IEEE international performance, computing, and communication conference*, 1999.

[11] van Hook et al. Alignment and ordering of vector elements for single instruction multiple data processing. *US. patent 5,933,650*, (MIPS Technologies Inc.), Aug. 1999.