Multimed Tools Appl (2018) 77:15971–15994
DOI 10.1007/s11042-017-5171-2

CPU-based real-time maximim intensity projection via fast matrix transposition using parallelization operations with AVX instruction set

Heewon Kye[1] · **Se Hee Lee**[2] · **Jeongjin Lee**[3]

Received: 28 March 2017 / Revised: 24 August 2017 / Accepted: 29 August 2017 /
Published online: 11 September 2017

Abstract Rapid visualization is essential for maximum intensity projection (MIP) rendering, since the acquisition of a perceptual depth can require frequent changes of a viewing direction. In this paper, we propose a CPU-based real-time MIP method that uses parallelization operations with the AVX instruction set. We improve shear-warp based MIP rendering by resolving the bottle-neck problems of the previous method of a matrix transposition. We propose a novel matrix transposition method using the AVX instruction set to minimize bottle-neck problems. Experimental results show that the speed of MIP rendering on general CPU is faster than 20 frame-per-second (fps) for a 512 × 512 × 552 volume dataset. Our matrix transposition method can be applied to other image processing algorithms for faster processing.

Keywords Maximum intensity projection · Matrix transposition · Parallelization · AVX instruction set · Volume rendering

✉ Jeongjin Lee
leejeongjin@ssu.ac.kr

Heewon Kye
kuei@hansung.ac.kr

Se Hee Lee
twoh25@naver.com

[1] Division of Computer Engineering, Hansung University, 116 Samseongyoro-16Gil Seongbuk-gu, Seoul 136-792, South Korea

[2] Department of Information Systems Engineering, Hansung University, 116 Samseongyoro-16Gil Seongbuk-gu, Seoul 136-792, South Korea

[3] School of Computer Science & Engineering, Soongsil University, 369 Sangdo-Ro, Dongjak-Gu, Seoul 156-743, South Korea

1 Introduction

Maximum intensity projection (MIP) is a volume visualization technique for displaying the maximum density of 3D volume data for a given viewpoint and direction. The clinical usability of MIP has been extensively evaluated, and MIP has proved to be useful in the generation of angiographic images from computed tomography (CT) and magnetic resonance (MR) imaging data [1]. For MIP rendering, a virtual projection ray is generated and passed along the viewing direction through volume data, and the maximum intensity is acquired for the intensities of voxels along this projection ray. This maximum density value is stored for the corresponding pixel, and displayed on the screen. Figure 1 shows that the maximum density value, 230 is stored for the corresponding pixel, since this value is the maximum intensity along the projection ray, which is parallel to the viewing direction.

3D volume data can be acquired from slice images of a human body taken by CT or MR. Using MIP rendering technique, clinically meaningful images can be acquired for human tissue, contrast-enhanced vessels, and skeletons. However, since there is no depth information in the MIP image, a doctor might frequently change the viewing direction to acquire a depth cue for diagnosis. Therefore, numerous methods have been researched for the fast generation of MIP imagery. First, a region of a volume data is removed during the preprocessing stage, when this region has low probability of being rendered in the MIP image. Schreiner et al. proposed a fast algorithm by projecting the high density first. All of the data was sorted according to its density value in the preprocessing step for a few minutes [24]. Mroz et al. proposed a 1%-lossy compression method, which removed about the half of the unnecessary volume data in the preprocessing stage [19]. Mroz et al. proposed a cell projection method. A cell is a data block that, after the removal of unnecessary data, is sorted according to the density along the viewing direction [20].

Object-order MIP rendering methods have been researched to improve the sampling process and efficiency of cache memory. To minimize the computational cost of trilinear sampling, shear-warp [5] and bilinear sampling [21] were applied. Mora et al. proposed an object-order MIP ray casting method with a theoretical analysis [18]. Kiefer et al. proposed an object-order MIP ray casting method, which improved the cache efficiency by propagating the ray along the inclined direction [6].

These researches can be summarized in a pipelined flow. First, volume data is divided into blocks. Unnecessary blocks are removed in the preprocessing step [19].

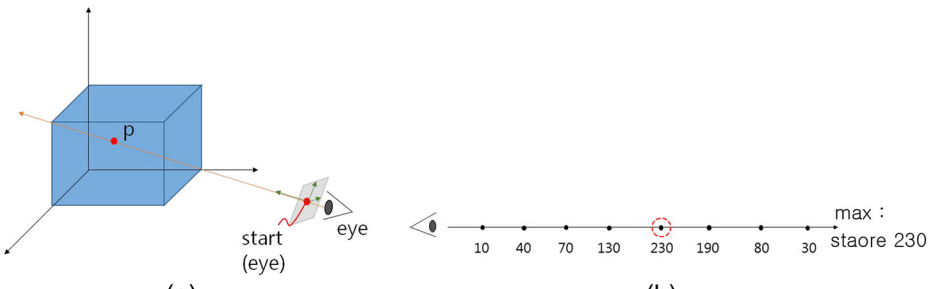


Fig. 1 Process of MIP rendering. **a** Generation of a projection ray, and **b** determination of MIP value

Since the result of MIP rendering is independent of the rendering order, data is sorted, and rendering is performed in this sorted order [20, 21, 24]. The processing is performed in each block by using block-based ray casting [6, 18], so that the efficiency of a cache memory is improved. In spite of these efforts, most previous methods required a long preprocessing step, and the rendering speed was only around 2 frame per second (fps).

With the recent advances of graphics processing unit (GPU) performance, many researches into volume data visualization have been conducted. In the past, volume visualization using 3D texture on GPU was only possible on a workstation [4], and it then became possible on a general GPU [22]. However, there are a few drawbacks to using a GPU for volume data visualization. First, the installation and development methodology of specialized libraries, such as CUDA (compute unified device architecture) [17] and DirectX [25], are required for the development and application of a visualization algorithm. A proper hardware unit with GPU must be installed for the use of the developed application in a clinical environment. Finally, since the memory size of a GPU is generally smaller than that of a CPU, a large dataset such as 4D-CT cannot be installed on a GPU, and direct volume rendering is impossible [27].

To overcome the problems of previous researches, Kye proposed a parallel algorithm using single instruction multiple data (SIMD) functions of CPU [7]. In this method, the parallelism of CPU was utilized with no preprocessing step. Since intensity comparison operations are frequently used in MIP to compute the maximum intensity, these operations are interpreted into branch operations in CPU, resulting in ineffient computations. On the other hand, an intensity comparison is processed in SIMD by fast arithmetic operations, resulting in a large improvement in a computation speed. In addition, many comparisons are concurrently processed, resulting in further gain in the performance improvement. As a result, interactive MIP rendering was possible using a CPU without the help of a GPU. However, the variations of a rendering speed were very large, according to the viewing direction.

In this paper, we propose a fast MIP algorithm using advanced vector extensions (AVX), by improving the method proposed by Kye [7]. In this method, a coordinate transformation is performed based on a shear-warp decomposition, so that the accesses of input medical volume data and output MIP image data are sequentially performed. This technique improves the efficiency of memory access. Intensity comparison operations, which are most frequently used in MIP, are replaced by fast parallelized arithmetic operations of AVX, resulting in the improvement of computational efficiency. Compared with the previous method [7], the visualization speed is much improved, since one operation handles a dataset four times. In this paper, we propose an efficient method of a matrix transposition. In the previous method [7], the efficiency of a memory access was improved, but matrix transformation was required in some viewing directions, such as a sagittal view. Since a matrix transformation is generally inefficient, MIP rendering time in a sagittal view was about four times longer than that in other viewing directions. In our method, we developed a matrix transposition method with 64 operations of a 16×16 matrix. In this paper, we prove that a matrix transposition of an n x n matrix using SIMD requires at least $n \log_2 n$ operations. Experimental results showed that the speed variations were minimized by more than two times according to the viewing direction, compared with the previous method.

Consequently, our method performs real-time MIP visualization of $512 \times 512 \times 552$ medical image data with 20 fps speed using a general and single-core CPU without any

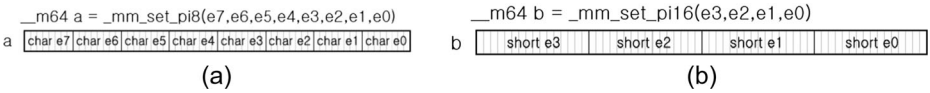


Fig. 2 SIMD instruction to input multiple data to one 64-bit variable. **a** Eight 8-bit data input, and **b** four 16-bit data input

preprocessing step. This speed cannot be acquired from any other previous CPU-based MIP rendering. Our method can be used in clinical practice, without the help of a GPU. Our method does not require any specialized hardware, and can be used in any kind of general purpose hardware, such as a desktop personal computer (PC), or notebook PC.

The remainder of this paper is organized as follows. The next section describes SIMD operations and MIP implementations using SIMD operations. Section 3 presents the proposed method. Section 4 presents the results of the application of the proposed method to clinical datasets, along with a comparison with other previous methods. Finally, Section 5 summarizes the results of our findings, and discusses directions for future work.

2 Related works

In this chapter, Section 2.1 explains the SIMD instruction that is the basis of the proposed research, while Section 2.2 describes the MIP algorithm that utilizes it.

2.1 SIMD instruction

SIMD is a technology or hardware structure that is capable of simultaneously executing a single command in parallel on a plurality of data. Intel developed MultiMedia eXtension (MMX), which is the SIMD extension instruction set for Pentium in 1997. With MMX technology, 64-bit data can be calculated in parallel by division into the size desired by the user. For example, four 16-bit integer (short) or eight 8-bit integer (char) values can be concurrently computed Figs. 2, 3 and 4.

In addition, applying the max instruction in SIMD makes it possible to compare two sets of values and extract only larger values at once. Therefore, the inefficient conditional branch instruction is replaced by a simple arithmetic instruction. Applying the SIMD instructions to the MIP in this way makes it possible to obtain the maximum values efficiently.

However, there are preconditions for efficiently performing the SIMD instruction. First, the data must be continuously present in memory. In the above example, the data values 6, 8, 2 and 5 must be sequentially present in the memory. If the data is scattered in the memory, large inefficiency occurs when it is transferred to a SIMD register. Second, parallel computation should be performed in element units. In the above example, calculating the larger value of 6

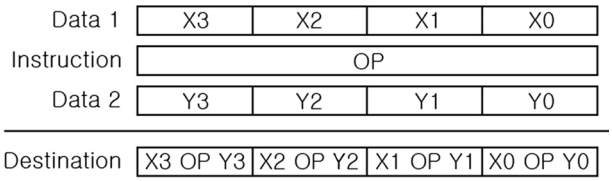


Fig. 3 Multi data operation using SIMD

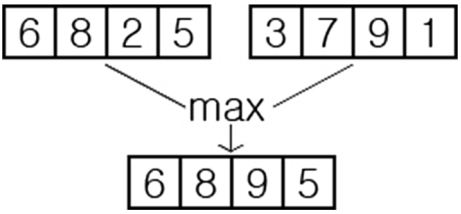


Fig. 4 Max operation using SIMD

and 3 (or 8 and 7), which is called a vertical operation, is efficient. However, getting the largest value 8 among 8, 6, 2 and 5 belonging to one register, which is called a horizontal operation, is very inefficient. In Section 2.2, we show a fast MIP implementation that satisfies these two preconditions.

2.2 Fast MIP using MMX instruction

In this section, we explain a MIP implementation of Kye's method [7] that can use MMX instructions because of the transformation of the coordinate system. In advance, it is assumed that the coordinate axes of the volume data are denoted by X, Y, and Z axes, and the coordinate axes of the output image are denoted by i, j, and k axes. Therefore, the volume data is stored sequentially along the X-axis, and the image data is stored along the i-axis in the physical memory. The viewing direction of the observer is the k-axis. Each voxel (pixel in medical images) has a 2 byte integer value, which is the standard voxel size of CT or MR images.

If the viewing direction (k-axis) is exactly parallel to the Z-axis, it is simple to implement MIP using MMX. First, coordinate transformation is organized. Since the Z-axis corresponds to the k-axis, the X-axis of the volume can correspond to the i-axis of the image, and the Y-axis of the volume corresponds to the j-axis of the image.

Then, the four voxels sequentially existing in the X-axis correspond to four pixels in the i-axis on a one-to-one basis. Now, if we read four consecutive voxel data at once in a 8-byte MMX register and use the max instruction with four consecutive pixels in the image and update it, the process of the four voxels is completed at once without the branch instruction. When this process is repeated in each Y-axis and Z-axis direction, one output image is generated.

Algorithm 1. MIP algorithm for viewing along the Z-axis using MMX.

Algorithm 1. MIP algorithm for viewing along the Z-axis using MMX.

MIP on Z-axis

```
1:  FOR EACH slice (Z)
2:    FOR EACH scanline (Y)
3:      get IMAGE_POS from (Y, Z)
4:      get VOL_POS from (Y, Z)
5:      FOR X=0 to volume_width, X = X + 4
6:          store MAX2Bx4( VOL_POS + X,   IMAGE_POS + X) to IMAGE_POS+X
```

Similarly, when we observe the volume data in parallel with the Y-axis, the coordinate system is transformed as follows. Since the Y-axis corresponds to the k-axis, which is the

viewing direction, the X-axis can correspond to the i-axis. MIP can be efficiently performed, as described above. For reference, the Z-axis automatically corresponds to the j-axis.

However, observation in the X-axis usually causes inefficiency. Four consecutive values in the X-axis of the volume data correspond to the k-axis, and they cannot be stored on the efficient i-axis. On the other hand, if we apply the ray casting method [23], it is also inefficient to parallelize the calculation of maximum value along the X-axis, because this is the horizontal operation described in Section 2.1.

To reduce the inefficiency, a related research [7] performed a matrix transposition operation on a 4 × 4 sized voxel unit (a tile or matrix). A tile in the X and Y-axis directions is stored in four MMX registers, and the X-axis and Y-axis are exchanged by the transposition. As a result, the X-axis corresponds to the i-axis, and it is possible to efficiently apply the SIMD instructions.

Since at this time the transposition takes a considerable portion of the total computation time, more improvements are needed. In this research, we propose a new method to transpose on a 16 × 16 sized tile using AVX2. The transposition on a 8 × 8 sized tile was studied in existing research [16], but the transposition on a 16 × 16 sized tile is first studied in this research.

3 The proposed method

Figure 5 shows the overall process of this research. Acceleration is performed using the CPU-based SIMD instructions (Section 3.1), and our coordinate system applies the shear-warp factorization (Section 3.2). In this process, different coordinate transformations are performed according to the observation direction (Section 3.3). When the observation direction is the X-axis, the fast transposition method is applied (Section 3.4).

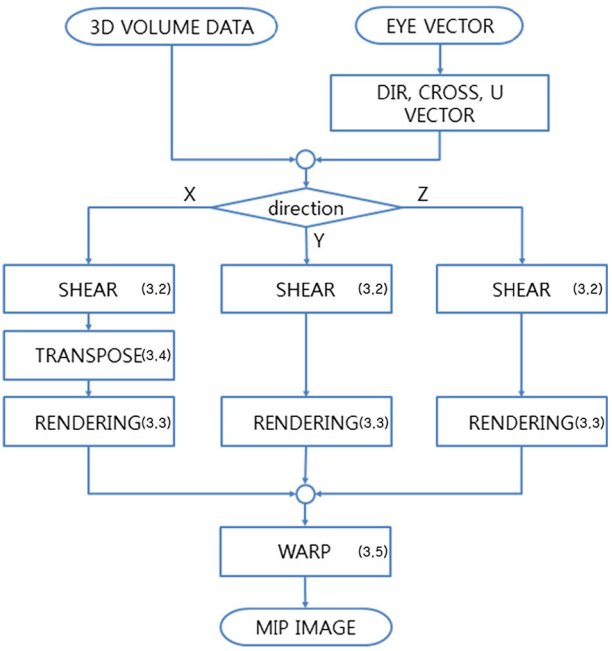


Fig. 5 A flowchart of the proposed MIP method

3.1 SIMD technologies of CPU and AVX2

In 1996, the first SIMD for PC was announced as the MMX technology of 8 bytes capacity. Since Pentium 3 was released in 1999, Streaming SIMD Extensions (SSE) has become available. SSE enables parallel execution of floating point instructions of 16 bytes. Integer data processing became possible in 2001 when the SSE2 was developed. Advanced Vector Extensions (AVX), introduced in 2008, can handle 32 bytes at once, and can handle floating point instructions. AVX2, which is available from Intel's Sandy Bridge CPU released in 2011, can process integer data in parallel in units of 32 bytes.

In this research, the AVX2 instruction set is used to take care of 2-byte integer voxels. Since it can be used on CPUs after 2011, this research can be applied on most existing computers, without requiring additional equipment.

3.2 Shear-warp factorization

The user usually observes patient data in an oblique direction. In this research, shear-warp factorization [8, 9] is used for accelerated visualization. Shear-warp factorization divides the transformation of coordinate system into two stages: shearing and warping. We denote M_{view} as the matrix that converts a point (Vol_x, Vol_y, Vol_z) in the volume coordinate to a point (Img_x, Img_y, Img_z) in the image coordinate.

$$\begin{bmatrix} Img_x \\ Img_y \\ Img_z \\ 1 \end{bmatrix} = \begin{bmatrix} M_{view} \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} \tag{1}$$

Then, shearing transformation is applied, so that the viewing direction becomes parallel to the Z-axis. As a result, the volume data is skewed in the sheared coordinate, as shown in Fig. 6. The matrix M_{shear} transforms each point in volume coordinate to the sheared coordinate (see Eq. (2)). Each patient image constituting the volume data (each colored bar) can be projected onto the base plane. We calculate the position on the base plane for each image, and project the image in the same way as in Section 2.2.

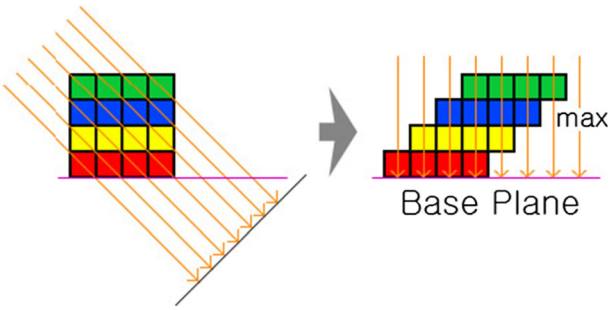


Fig. 6 Shearing transformation along the viewing direction

$$\begin{bmatrix} Shear_x \\ Shear_y \\ Shear_z \\ 1 \end{bmatrix} = \begin{bmatrix} M_{shear} \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} \tag{2}$$

The projected image in the sheared coordinate should be transformed to the output coordinate. We apply two-dimensional image processing as shown in Eq. (3) to generate the final output image. Since we have computed M_{view} and M_{shear}, we can calculate M_{warp}, the product of M_{view} and M_{shear}^{-1}. We can obtain the final output image by applying the warping transformation to the base plane.

$$\begin{aligned} \begin{bmatrix} Img_x \\ Img_y \\ Img_z \\ 1 \end{bmatrix} &= \begin{bmatrix} M_{view} \end{bmatrix} \begin{bmatrix} M_{shear} \end{bmatrix}^{-1} \begin{bmatrix} M_{shear} \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} M_{view} \end{bmatrix} \begin{bmatrix} M_{shear} \end{bmatrix}^{-1} \begin{bmatrix} Shear_x \\ Shear_y \\ Shear_z \\ 1 \end{bmatrix} = \begin{bmatrix} M_{warp} \end{bmatrix} \begin{bmatrix} Shear_x \\ Shear_y \\ Shear_z \\ 1 \end{bmatrix} \end{aligned} \tag{3}$$

3.3 Rendering method for viewing directions

In this section, similar to Section 2.2, it is assumed that the coordinate axes of the volume data are denoted by X, Y, and Z axes, and the coordinate axes of the output image are denoted by i, j, and k axes. Even though the volume can be observed in the axial (Z-axis), coronal (Y-axis), or sagittal (X-axis) directions. it is efficient to make the X-axis correspond to the i-axis, because the SIMD can handle sequential data in the memory.

In observing along the Y-axis, the Y-axis corresponds to the k-axis. Therefore, the X, Y, and Z axes correspond to the i, k, and j axes, respectively. Images parallel to the X-Z plane are projected to the base plane, while increasing by 1 in the Y-axis (see Fig. 7b).

In observing along the X-axis, the X-axis corresponds to the k-axis, and cannot correspond to the i-axis (see Fig. 7c). In order to solve this inefficiency problem, we transpose all images parallel to the X-Y plane. As the result, the X, Y, and Z axes can correspond to the i, k, and j axes, respectively (see Fig. 7d).

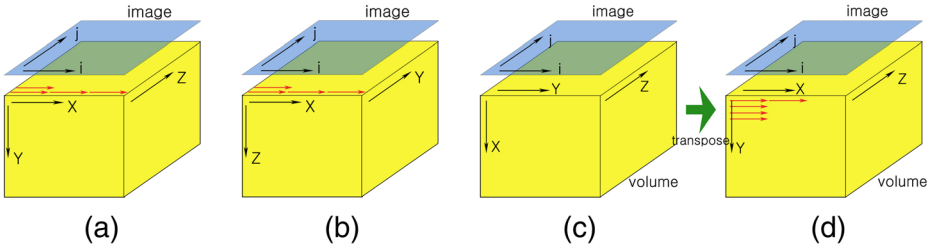


Fig. 7 Transfomation of coordinate system and accessing order of volume data (red arrow). Observation in the **a** Z-axis, **b** Y-axis, **c** X-axis, and **d** X-axis after transposition

Since voxels along the X-axis and pixels along the i-axis correspond one-to-one, the calculation of the coordinates becomes simple, as shown in Eq. (4). The position of the sheared coordinate when $vol_x = 0$ is calculated once, and the X vol is added to obtain the exact coordinates on the base plane. Coordinate calculation is completed with one addition operation, instead of a matrix multiplication operation.

$$\begin{bmatrix} Shear_x \\ Shear_y \\ Shear_z \\ 1 \end{bmatrix} = \begin{bmatrix} M_{shear} \end{bmatrix} \begin{bmatrix} Vol_x \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix}$$
$$= \begin{bmatrix} M_{shear} \end{bmatrix} \begin{bmatrix} 0 \\ Vol_y \\ Vol_z \\ 1 \end{bmatrix} + \begin{bmatrix} M_{shear} \end{bmatrix} \begin{bmatrix} Vol_x \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} Shear_x | Vol_x = 0 \\ Shear_y \\ Shear_z \\ 1 \end{bmatrix} + \begin{bmatrix} Vol_x \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (4)$$

We summarize it in Algorithms 2 and 3. When observing along the Y-axis, the calculation is performed in units of 32 bytes along the X-axis. If one scanline along the X-axis is projected to the end, it proceeds along the Z-axis. If a plane along the X-Z plane is projected to the end, it proceeds along the Y-axis. When observing along the X-axis, a X-Y plane consists of several 16 × 16 tiles. Each tile is projected onto the base plane after transposition. If all tiles in a X-Y plane are projected to the end, it proceeds along the Z-axis.

Algorithm 2. MIP algorithm for viewing along the Y-axis using AVX2.

Algorithm 2. MIP algorithm for viewing along the Y-axis using AVX2.

```
Y-axis based MIP
1:  FOR each Y
2:    FOR each Z
3:      get IMAGE_POS from (Y, Z)
4:      get VOL_POS from (Y, Z)
5:      FOR X=0 to volume_width, X = X + 16
6:          store MAX2Bx16( VOL_POS + X,   IMAGE_POS + X) to IMAGE_POS+X
```

Algorithm 3. MIP algorithm for viewing along the X-axis using AVX2.

Algorithm 3. MIP algorithm for viewing along the X-axis using AVX2.

```
X-axis based MIP
1:  FOR each Z
2:    FOR Y = 0 to volume_height, Y = Y + 16
3:      FOR X = 0 to volume_width, X = X + 16
4:          tile = MAKE 16x16 TILE from (X, Y)
5:          tile = TRANSPOSE(tile)
6:          get IMAGE_POS from (Y, Z)
7:          FOR each ROW in the tile
8:              store MAX2Bx16( ROW,   IMAGE_POS) to IMAGE_POS
9:              IMAGE_POS = IMAGE_POS + image_width
```

3.4 Fast matrix transposition

The transposition is needed only when observing on the X-axis, so that the observer feels that it unnaturally slows down. This problem can be solved by improving the efficiency of the transposition. In this section, we show a method of performing a 16 × 16 sized matrix transposition with 16 × $\log_2$ 16 = 64 instructions. It is proven below that this method uses the minimum number of instructions.

For example, one 8-byte MMX register can store four voxels, so a 4 × 4 matrix is stored in four registers, as shown in Fig. 8. If the input values for the four registers are {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, and {13, 14, 15, 16}, respectively, {1, 5, 9, 13}, {2, 6, 10, 14}, {3, 7, 11, 15}, and {4, 8, 12, 16} must be obtained through transposition.

The data exchanging operation of SIMD fills one output register by various combinations with two input registers. Because 1, 5, 9, and 13 are scattered in each of four different registers, it is impossible to obtain the result by one instruction. Therefore, at least two instructions are required. As shown in Fig. 8, {1, 5, 3, 7} and {9, 13, 11, 15}, which are intermediate values, are generated, then {1, 5, 9, 13} is obtained. The total number of instructions is three, but the intermediate results can be reused. The total number of instructions is 8, and two instructions are needed to generate each output register.

We need to gather the scattered values in the eight registers to transpose an 8 × 8 matrix. Since one more instruction to bind two registers gathered from the four input registers is required, at least three instructions for each output register are needed. Likewise, four instructions are necessary for a 16 × 16 matix, and at least $\log_2$ n operation is required to gather the scattered values in n registers for an n x n matrix. Therefore, at least n $\log_2$ n instructions are necessary to transpose the n x n matrix.

Since n $\log_2$ n instructions are required for n^2 elements, the overhead for each element is $ov(n) = \frac{n\log_2 n}{n^2} = \frac{n\log_2 n}{n^2}$. This function is monotonically decreasing for n, and becomes $\lim_{n\to\infty}$

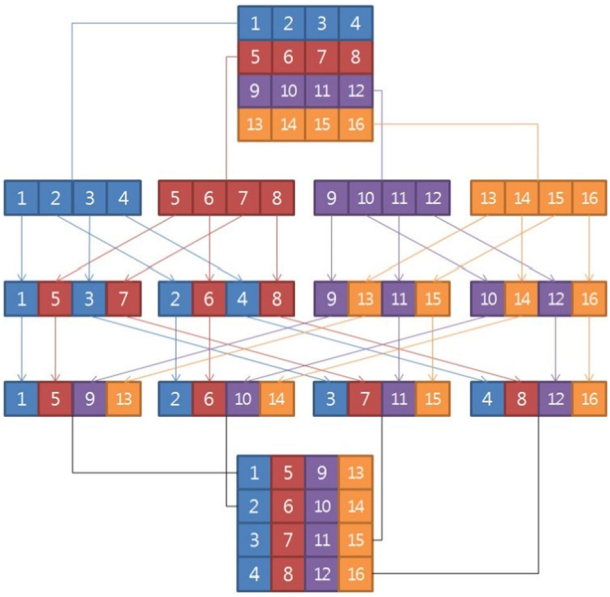


Fig. 8 An example of 4 × 4 sized matrix transposition using SIMD [7]

$ov(n) = \lim_{n\to\infty} \frac{\log_2 n}{n} = 0$. Therefore, as larger registers are used, n increases and overhead decreases, and the relative overhead to transpose a matrix converges to zero.

In related work [7], 4 × 4 voxels were considered as a tile. The transposition was implemented using the macro function provided by Intel. In this research, we developed a new method to transpose a 16 × 16 tile with 16 × $\log_2$ 16 = 64 instructions using AVX2, which supports 64-byte register.

First of all, this paper describes the transpose for an 8 × 8 tile using SSE2. One 16-byte SSE2 register can store 8 voxels, and 8 registers are used to store a 8 × 8 tile. For each 8 × 8 tile, eight instructions are used in one phase, and the transposition is completed in three phases, and 24 instructions [2]. Algorithm 4 shows an example of the transposition implemented in SSE2, while Fig. 9 shows the change of data when each phase is executed. The symbol UNPACK in Algorithm 4 is a kind of data swizzle instruction [8] which combines the parts of two registers into one register.

Algorithm 4. 8 × 8 transposition.

Algorithm 4. 8x8 transposition

8x8 transpose

```
1:  TRANSPOSE_8x8(VA[0:127], VB[0:127], VC[0:127], VD[0:127], VE[0:127], VF[0:127],
     VG[0:127], VH[0:127])
2:  {
3:    SET TA[0:127], TB[0:127], TC[0:127], TD[0:127], TE[0:127], TF[0:127], TG[0:127],
      TH[0:127];
4:    // phase1
5:    TA[0:127] <- UNPACK(VA[0:15], VB[0:15], VA[16:31], VB[16:31], VA[32:47],
            VB[32:47], VA[48:63], VB[48:63])
6:    TB[0:127] <- UNPACK(VA[64:79], VB[64:79], VA[80:95], VB[80:95], VA[96:111],
            VB[96:111], VA[112:127], VB[112:127])
7:    TC[0:127] <- UNPACK(VC[0:15], VD[0:15], VC[16:31], VD[16:31], VC[32:47],
            VD[32:47], VC[48:63], VD[48:63])
8:    TD[0:127] <- UNPACK(VA[64:79], VB[64:79], VA[80:95], VB[80:95], VA[96:111],
            VB[96:111], VA[112:127], VB[112:127])
9:    TE[0:127] <- UNPACK(VE[0:15], VF[0:15], VE[16:31], VF[16:31], VE[32:47],
            VF[32:47], VE[48:63], VF[48:63])
10:   TF[0:127] <- UNPACK(VE[64:79], VF[64:79], VE[80:95], VF[80:95], VE[96:111],
            VF[96:111], VE[112:127], VF[112:127])
11:   TG[0:127] <- UNPACK(VG[0:15], VH[0:15], VG[16:31], VH[16:31], VG[32:47],
            VH[32:47], VG[48:63], VH[48:63])
12:   TH[0:127] <- UNPACK(VG[64:79], VH[64:79], VG[80:95], VH[80:95], VG[96:111],
            VH[96:111], VG[112:127], VH[112:127])

13:   // phase2
14:   VA[0:127] <- UNPACK(TA[0:31], TC[0:31], TA[32:63], TC[32:63])
15:   VB[0:127] <- UNPACK(TA[64:95], TC[64:95], TA[96:127], TC[96:127])
16:   VC[0:127] <- UNPACK(TB[0:31], TD[0:31], TB[32:63], TD[32:63])
17:   VD[0:127] <- UNPACK(TB[64:95], TD[64:95], TB[96:127], TD[96:127])
18:   VE[0:127] <- UNPACK(TE[0:31], TG[0:31], TE[32:63], TG[32:63])
19:   VF[0:127] <- UNPACK(TE[64:95], TG[64:95], TE[96:127], TG[96:127])
20:   VG[0:127] <- UNPACK(TF[0:31], TH[0:31], TF[32:63], TH[32:63])
21:   VH[0:127] <- UNPACK(TF[64:95], TH[64:95], TF[96:127], TH[96:127])

22:   // phase3
23:   TA[0:127] <- UNPACK(VA[0:63], VE[0:63])
24:   TB[0:127] <- UNPACK(VA[64:127], VE[63:127])
25:   TC[0:127] <- UNPACK(VB[0:63], VF[0:63])
26:   TD[0:127] <- UNPACK(VB[64:127], VF[63:127])
27:   TE[0:127] <- UNPACK(VC[0:63], VG[0:63])
28:   TF[0:127] <- UNPACK(VC[64:127], VG[63:127])
29:   TG[0:127] <- UNPACK(VD[0:63], VH[0:63])
30:   TH[0:127] <- UNPACK(VD[64:127], VH[63:127])
31: }
```

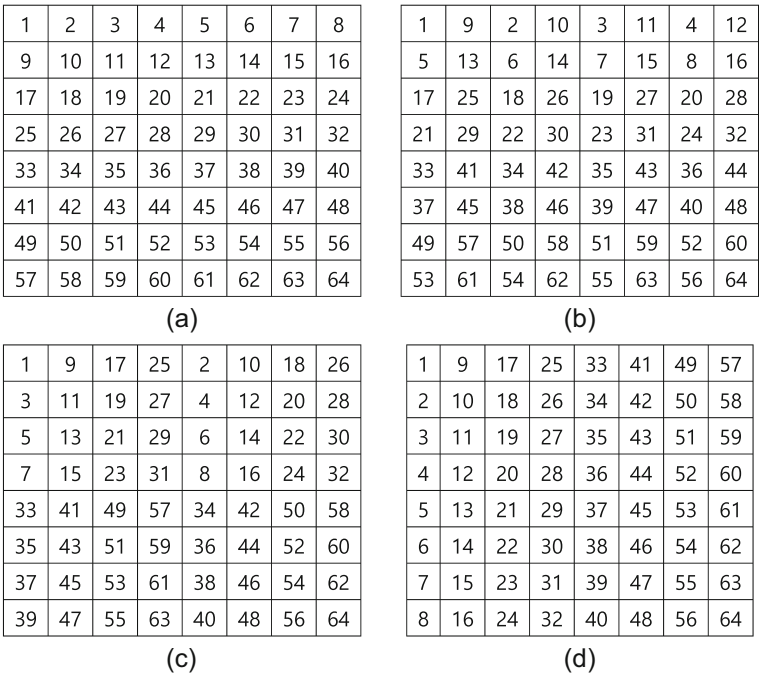


Fig. 9 Input data **a** after phase1, **b** after phase2, **c** after phase3; and **d** the 8 × 8 transposition process using SSE2

In this research, we extended this concept, and developed a new transposition using AVX2. (For specific details, refer to the Appendix) A 32 byte AVX2 register can store 16 voxels, and 16 registers are used to transpose a 16 × 16 matrix. By executing four phases of the command, it is possible to perform the transposition of the 16 × 16 matrix with 16 × 4 = 64 instructions.

If the existing MMX technology is used to transpose a 16 × 16 matrix, it is necessary to transpose 4 × 4 matrices 16 times, so that 8 instructions × 16 times = 128 instructions are required. The proposed method uses only half ($\frac{ov(16)}{ov(4)} = 0.5$) of the instructions.

3.5 Warping

Figure 5 shows the final stage of this research, which is warping to generate the output image. Warping is an image processing technique that corresponds to two-dimensional affine

Table 1 Volume data set

Name	Dimension	Pixel Size [mm]	Slice thickness [mm]	Size (MB)
Chest #1	512 × 512 × 277	0.57 × 0.57	1.00	138.5
Chest #2	512 × 512 × 110	0.63 × 0.63	3.00	55.0
Lung	512 × 512 × 528	0.66 × 0.66	0.75	264.0
Head	512 × 512 × 552	0.42 × 0.42	1.00	276.0
Lower extremity	512 × 512 × 1165	0.78 × 0.78	1.50	582.5

Table 2 Randomly generated viewing direction vectors

	1	2	3	4	5	6	7
x-dir	(0.927, 0.261, 0.271)	(0.722, −0.188, −0.666)	(−0.799, −0.228, 0.556)	(0.605, 0.587, −0.538)	(0.879, 0.460, −0.125)	(0.634, 0.542, −0.551)	(0.981, 0.189, −0.044)
y-dir	(−0.132, 0.951, −0.278)	(0.0489, 0.811, −0.583)	(0.242, 0.887, 0.393)	(0.394, 0.777, 0.491)	(0.530, 0.840, 0.117)	(−0.628, −0.723, −0.289)	(0.385, −0.913, −0.134)
z-dir	(−0.103, −0.667, −0.738)	(0.653, −0.375, −0.658)	(−0.470, 0.260, −0.843)	(0.696, −0.0412, −0.717)	(0.695, −0.020, −0.718)	(0.102, −0.0161, −0.995)	(−0.118, 0.436, −0.892)

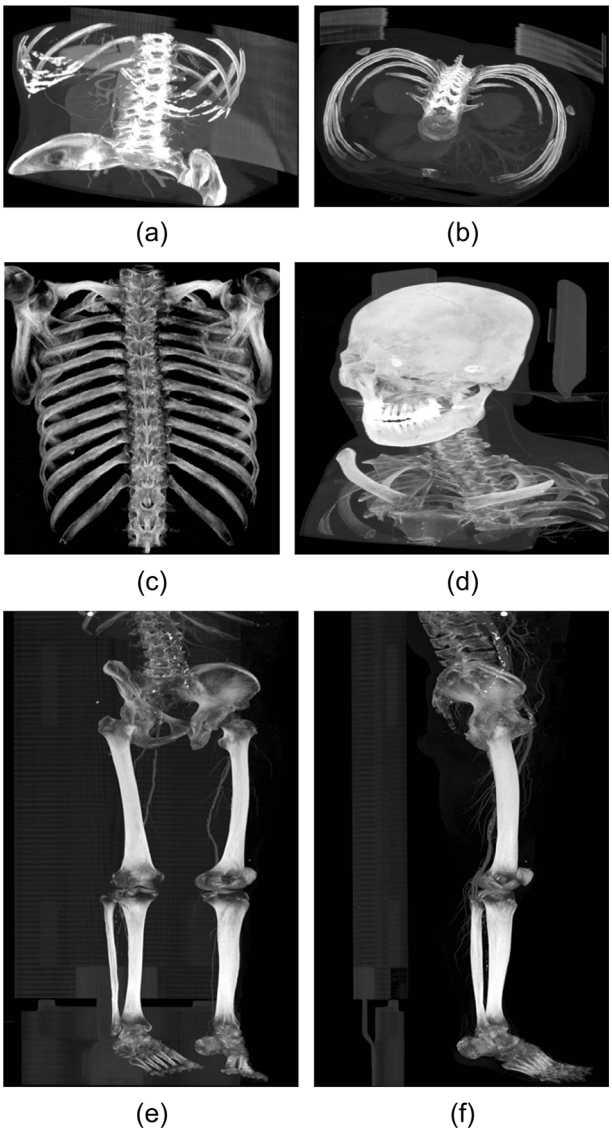


Fig. 10 MIP rendering images of test data sets. **a** Chest #1, **b** Chest #2, **c** Lung, **d** Head, **e** Lower extremity, and **f** Lower extremity with the other viewing direction

translation, which is the process of correcting the tilted image created in the previous step. The transformation can be calculated as Eq. (3).

Table 3 MIP rendering time comparison with conventional methods (ms = 10^{-3} s)

MMX [7]	SSE2 [8]	Proposed method
37.038	31.273	23.972

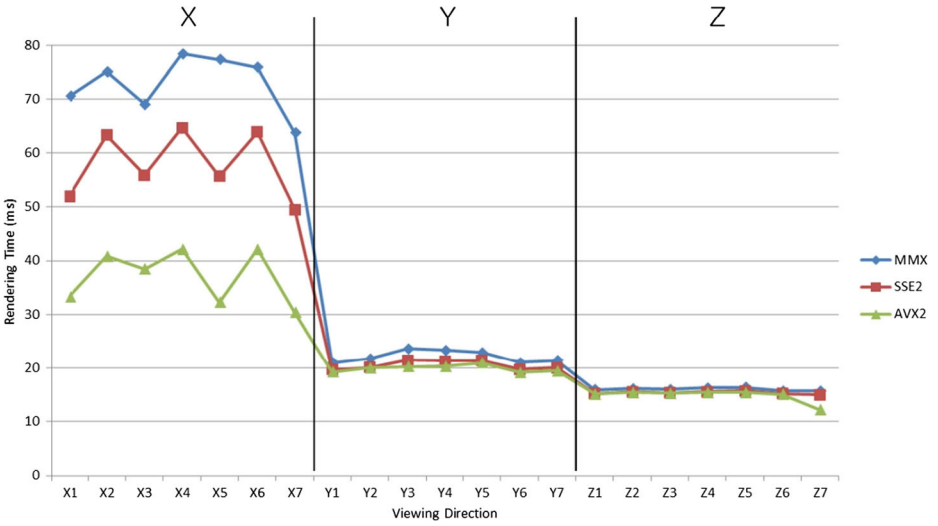


Fig. 11 Comparison of MIP rendering time for each axis viewing direction, using MMX [7], SSE2 [8], and the proposed method

Since the forward mapping generates holes in image processing, we used the backward mapping using M_{warp}^{-1} with bilinear interpolation. In modern hardware environments, two-dimensional warping takes less than 2 ms, and the influence on overall performance is very small.

4 Experimental results

4.1 Experimental environment and visual inspection

The experiments were performed using an Intel i5-4460 desktop system with a 3.2 GHz processor and 8 GB of main memory. We implemented our method using Visual Studio 2015 C++ on Window7 operating system. Table 1 shows our tests of the proposed method on four data sets. The number of images per scan ranged from 110 to 1165. Each image had a matrix size of 512 × 512. Pixel sizes ranged from 0.42 to 0.78 mm. CT (computed tomography) was

Table 4 Matrix transposition time comparison with the conventional methods (ns = 10^{-9} s)

	MMX [7] 4 × 4 matrix	Zekri et al. [26] 4 × 4 matrix	Proposed method 16 × 16 matrix
Processing time of one operation	4.721	3.6	27.62
Processing time of 16 × 16 matrix transpose	75.536	57.6	27.62
Clock number of 16 × 16 matrix transpose	271.7	126.7	88.4

performed with a 16-channel multi-detector row CT scanner (Siemens Somatom 16; Siemens, Forchheim, Germany) in all the subjects. The scanning parameters were 120 kVp, 200 mA, 0.5-s rotation time, and 512 × 512-pixel matrix with a detector-row configuration of 0.75 × 16.0 mm.

We evaluated MIP rendering speed at various viewing directions, since the rendering performance varied according to the viewing direction. First, we randomly generated 21 viewing direction vectors, as shown in Table 2. Accordingly, each principal viewing direction (x-, y-, and z-direction) has 7 randomly generated viewing direction vectors around each principal viewing direction.

Figure 10 shows the MIP rendering results of test data sets. In addition, we evaluate the image quality of our MIP rendering methods by subtracting all of the MIP rendering images of the proposed method from those of MMX [7] and SSE2 [8], while each data set was rotated from 0 degree to 360 degree with an angle increase of 1 degree around the diagonal axis (1, 1, 1). None of the subtraction images from test data sets with 360 viewing directions had any non-zero intensity pixel. We concluded that the proposed method showed no loss of image quality.

4.2 Comparison with the conventional methods

We compared the computational efficiency of the proposed method with those of conventional methods: MMX [7] and SSE2 [8]. Table 3 shows the average MIP rendering time for these three algorithms. We measured the average MIP rendering time at 21 viewing directions (Table 2) using Chest #1 data set. At each viewpoint, we performed 100 times rendering for accurate measurement. SSE2 [8] was faster than MMX [7], and the proposed method was faster than SSE2 [8]. Consequently, our method achieved about 1.55 times speedup of MIP rendering on average, without any loss of image quality, compared with the conventional method [7].

To analyze the performance according to the viewing direction, Fig. 11 shows the rendering time of each method according to the viewing direction. Since our method and conventional methods are based on shear-warp decomposition, the rendering time varies greatly according to the viewing direction. Our method achieved about 1.1 times speedup for the y-axis or z-axis viewing direction, since the memory was sequentially accessed at these axes. For the x-axis viewing direction, our method achieved about 2 times speedup, compared with MMX [7]. MIP rendering of the x-axis viewing direction was slower than that of the other viewing directions, due to the matrix transposition. However, the fast transposition of our method enabled the speedup of MIP rendering. In the previous method [7], the rendering speed for the x-axis viewing direction was about 3.8 times slower than for the y-axis or z-axis viewing direction. On the other hand, the rendering speed for the x-axis viewing direction in our method was about 2.1 times slower than for the y-axis or z-axis viewing direction. Our method much improved the speed difference along the viewing direction in Fig. 11. Furthermore, the proposed method doesn't require any time-consuming pre-processing or additional memory usage.

Next, we compared the processsing time of the matrix transposition of our method with those of MMX [7] and Zekri et al. [26]. Table 4 shows the average processing time after one hundred million times of the same computation for each method. This time does not include the memory copy time. The experimental results of Zekri et al.

Table 5 MIP rendering time comparison of test data sets with the conventional methods (μs = 10^{-6} s)

	X-axis direction			Y-axis direction			Z-axis direction		
	MMX	SSE2	Our method	MMX	SSE2	Our method	MMX	SSE2	Our method
Chest #1	70.54734	51.96790	33.33280	20.99611	19.69106	19.27156	15.96361	15.21291	15.12345
Chest #2	28.01430	20.48483	13.18679	7.66047	7.23729	7.03868	6.29126	5.99895	6.04863
Lung	138.28922	100.87867	68.45288	45.88528	41.38809	39.94884	31.35695	29.92903	29.89445
Head	144.15236	103.70526	70.60609	48.73287	43.88085	41.97602	32.52242	31.22059	31.14376
Lower extremity	284.33343	203.81860	138.73640	121.41248	108.11024	104.23639	64.89549	62.87185	62.54959

[26] in Table 4 were performed using an Intel i7-2670 desktop system with a 2.2 GHz processor, and the float operation was performed in 4 × 4 unit. The processing speed of our method was about two times faster than that of Zekri et al. [26]. With respect to the clock number, the performance of our method was about 1.5 times faster than that of Zekri et al. [26].

In addition, we measured the MIP rendering time for various kinds of data sets to prove the roubustness of our method. Table 5 shows the average rendering time of each method after 100 times repetition for the x-axis direction (0.926509, 0.260581, 0.271438), y-axis direction (−0.131742, 0.951469, −0.278122), and z-axis direction (−0.102672, −0.667368, −0.737617), respectively. We found that the MIP rendering time linearly increased in proportion to the volume data size. For example, the size of the head data set was two times larger than that of the chest #1 data set, so that MIP rendering time of the head data set was about two times longer than that of the chest #1 data set. This characteristic is one feature of the shear-warp based method, which is impemented on volume data. We can approximately estimate the MIP rendering time if we use another volume data set. Consequently, our method achieved about 20 fps speed for MIP rendering of the 512 × 512 × 552 head data set using a general CPU.

5 Conclusion and future Works

In this paper, we proposed a fast MIP algorithm using AVX, by improving the method proposed by Kye [7]. Our method performed a real-time MIP visualization of 512 × 512 × 552 medical image data with 20 fps speed, using a general and single-core CPU without any preprocessing step. In our method, we developed a matrix transposition method with 64 operations of a 16 × 16 matrix. Experimental results showed that compared with the previous method, the speed variations were minimized more than two times according to the viewing direction. Our matrix transposition method can be applied to other image processing algorithms for faster processing.

In our method, we used a single-core CPU, and interpolation was not applied. If we use multi-core CPU techniques, such as openMP technology, real-time MIP rendering might be possible with an interpolation. In addition, since AVX-512 is planned to be launched to handle 64 bytes in late 2017, 160 (32 × 5) operations can perform a matrix transposition of 32 × 32 matrix. In future work, our algorithm can be extened into the various kinds of applications in mobile platform [13, 15]. And our method can be applied into a real-world dataset [12, 14]. And our method can be utilized in tracking and visual analysis applications [3, 10, 11].

(The executable file with a sample dataset is provided at the author's web-site: http://www.gilab.co.kr).

Acknowledgements This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (No. 2017R1A2B3011475).

Appendix

16 × 16 matrix transposition. AVX2 instructions and the processing examples

Table 6 16 × 16 tile transposition using AVX2

AVX2 transpose (16x16)

```
1:  Transpose_16x16(vA, vB, vC, vD, vE, vF, vG, vH, vI, vJ, vK, vL, vM, vN, vO, vP)
2:  {
3:    __m256i tA, tB, tC, tD, tE, tF, tG, tH, tI, tJ, tK, tL, tM, tN, tO, tP;

4:    // phase 1
5:    tA = _mm256_unpacklo_epi16(vA, vB);
6:    tB = _mm256_unpackhi_epi16(vA, vB);
7:    tC = _mm256_unpacklo_epi16(vC, vD);
8:    tD = _mm256_unpackhi_epi16(vC, vD);
9:    tE = _mm256_unpacklo_epi16(vE, vF);
10:   tF = _mm256_unpackhi_epi16(vE, vF);
11:   tG = _mm256_unpacklo_epi16(vG, vH);
12:   tH = _mm256_unpackhi_epi16(vG, vH);
13:   tI = _mm256_unpacklo_epi16(vI, vJ);
14:   tJ = _mm256_unpackhi_epi16(vI, vJ);
15:   tK = _mm256_unpacklo_epi16(vK, vL);
16:   tL = _mm256_unpackhi_epi16(vK, vL);
17:   tM = _mm256_unpacklo_epi16(vM, vN);
18:   tN = _mm256_unpackhi_epi16(vM, vN);
19:   tO = _mm256_unpacklo_epi16(vO, vP);
20:   tP = _mm256_unpackhi_epi16(vO, vP);

21:   // phase 2
22:   vA = _mm256_unpacklo_epi32(tA, tC);
23:   vB = _mm256_unpackhi_epi32(tA, tC);
24:   vC = _mm256_unpacklo_epi32(tB, tD);
25:   vD = _mm256_unpackhi_epi32(tB, tD);
26:   vE = _mm256_unpacklo_epi32(tE, tG);
27:   vF = _mm256_unpackhi_epi32(tE, tG);
28:   vG = _mm256_unpacklo_epi32(tF, tH);
29:   vH = _mm256_unpackhi_epi32(tF, tH);
30:   vI = _mm256_unpacklo_epi32(tI, tK);
31:   vJ = _mm256_unpackhi_epi32(tI, tK);
32:   vK = _mm256_unpacklo_epi32(tJ, tL);
33:   vL = _mm256_unpackhi_epi32(tJ, tL);
34:   vM = _mm256_unpacklo_epi32(tM, tO);
35:   vN = _mm256_unpackhi_epi32(tM, tO);
36:   vO = _mm256_unpacklo_epi32(tN, tP);
37:   vP = _mm256_unpackhi_epi32(tN, tP);

38:   // phase 3
39:   tA = _mm256_unpacklo_epi64(vA, vE);
40:   tB = _mm256_unpackhi_epi64(vA, vE);
41:   tC = _mm256_unpacklo_epi64(vB, vF);
42:   tD = _mm256_unpackhi_epi64(vB, vF);
43:   tE = _mm256_unpacklo_epi64(vC, vG);
44:   tF = _mm256_unpackhi_epi64(vC, vG);
45:   tG = _mm256_unpacklo_epi64(vD, vH);
46:   tH = _mm256_unpackhi_epi64(vD, vH);
47:   tI = _mm256_unpacklo_epi64(vI, vM);
48:   tJ = _mm256_unpackhi_epi64(vI, vM);
49:   tK = _mm256_unpacklo_epi64(vJ, vN);
50:   tL = _mm256_unpackhi_epi64(vJ, vN);
51:   tM = _mm256_unpacklo_epi64(vK, vO);
52:   tN = _mm256_unpackhi_epi64(vK, vO);
53:   tO = _mm256_unpacklo_epi64(vL, vP);
54:   tP = _mm256_unpackhi_epi64(vL, vP);

55:   // phase 4
56:   vA = _mm256_permute2x128_si256(tA, tI, 0x20);
57:   vB = _mm256_permute2x128_si256(tB, tJ, 0x20);
58:   vC = _mm256_permute2x128_si256(tC, tK, 0x20);
59:   vD = _mm256_permute2x128_si256(tD, tL, 0x20);
60:   vE = _mm256_permute2x128_si256(tE, tM, 0x20);
61:   vF = _mm256_permute2x128_si256(tF, tN, 0x20);
62:   vG = _mm256_permute2x128_si256(tG, tO, 0x20);
63:   vH = _mm256_permute2x128_si256(tH, tP, 0x20);
64:   vI = _mm256_permute2x128_si256(tA, tI, 0x31);
65:   vJ = _mm256_permute2x128_si256(tB, tJ, 0x31);
66:   vK = _mm256_permute2x128_si256(tC, tK, 0x31);
67:   vL = _mm256_permute2x128_si256(tD, tL, 0x31);
68:   vM = _mm256_permute2x128_si256(tE, tM, 0x31);
69:   vN = _mm256_permute2x128_si256(tF, tN, 0x31);
70:   vO = _mm256_permute2x128_si256(tG, tO, 0x31);
71:   vP = _mm256_permute2x128_si256(tH, tP, 0x31);
72: }
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128
129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144
145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176
177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192
193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208
209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224
225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256

(a)

1	17	2	18	3	19	4	20	9	25	10	26	11	27	12	28
5	21	6	22	7	23	8	24	13	29	14	30	15	31	16	32
33	49	34	50	35	51	36	52	41	57	42	58	43	59	44	60
37	53	38	54	39	55	40	56	45	61	46	62	47	63	48	64
65	81	66	82	67	83	68	84	73	89	74	90	75	91	76	92
9	85	70	86	71	87	72	88	77	93	78	94	79	95	80	96
97	113	98	114	99	115	100	116	105	121	106	122	107	123	108	124
101	117	102	118	103	119	104	120	109	125	110	126	111	127	112	128
129	145	130	146	131	147	132	148	137	153	138	154	139	155	140	156
133	149	134	150	135	151	136	152	141	157	142	158	143	159	144	160
161	177	162	178	163	179	164	180	169	185	170	186	171	187	172	188
165	181	166	182	167	183	168	184	173	189	174	190	175	191	176	192
193	209	194	210	195	211	196	212	201	217	202	218	203	219	204	220
197	213	198	214	199	215	200	216	205	221	206	222	207	223	208	224
225	241	226	242	227	243	228	244	233	249	234	250	235	251	236	252
229	24	230	246	231	247	232	248	237	253	238	254	239	255	240	256

(b)

Fig. 12 The process of 16 × 16 matrix transposition. **a** Original, **b** after phase 1, **c** after phase 2, **d** after phase 3, and **e** after phase 4

1	17	33	49	2	18	34	50	9	25	41	57	10	26	42	58
3	19	35	51	4	20	36	52	11	27	43	59	12	28	44	60
5	21	37	53	6	22	38	54	13	29	45	61	14	30	46	62
7	23	39	55	8	24	40	56	15	31	47	63	16	32	48	64
65	81	97	113	66	82	98	114	73	89	105	121	74	90	106	122
67	83	99	115	68	84	100	116	75	91	107	123	76	92	108	124
69	85	101	117	70	86	102	118	77	93	109	125	78	94	110	126
71	87	103	119	72	88	104	120	79	95	111	127	80	96	112	128
129	145	161	177	130	146	162	178	137	153	169	185	138	154	170	186
131	147	163	179	132	148	164	180	139	155	171	187	140	156	172	188
133	149	165	181	134	150	166	182	141	157	173	189	142	158	174	190
135	151	167	183	136	152	168	184	143	159	175	191	144	160	176	192
193	209	225	241	194	210	226	242	201	217	233	249	202	218	234	250
195	211	227	243	196	212	228	244	203	219	235	251	204	220	236	252
197	213	229	245	198	214	230	246	205	221	237	253	206	222	238	254
199	215	231	247	200	216	232	248	207	223	239	255	208	224	240	256

(c)

1	17	33	49	65	81	97	113	9	25	41	57	73	89	105	121
2	18	34	50	66	82	98	114	10	26	42	58	74	90	106	122
3	19	35	51	67	83	99	115	11	27	43	59	75	91	107	123
4	20	36	52	68	84	100	116	12	28	44	60	76	92	108	124
5	21	37	53	69	85	101	117	13	29	45	61	77	93	109	125
6	22	38	54	70	86	102	118	14	30	46	62	78	94	110	126
7	23	39	55	71	87	103	119	15	31	47	63	79	95	111	127
8	24	40	56	72	88	104	120	16	32	48	64	80	96	112	128
129	145	161	177	193	209	225	241	137	153	169	185	201	217	233	249
130	146	162	178	194	210	226	242	138	154	170	186	202	218	234	250
131	147	163	179	195	211	227	243	139	155	171	187	203	219	235	251
132	148	164	180	196	212	228	244	140	156	172	188	204	220	236	252
133	149	165	181	197	213	229	245	141	157	173	189	205	221	237	253
134	150	166	182	198	214	230	246	142	158	174	190	206	222	238	254
135	151	167	183	199	215	231	247	143	159	175	191	207	223	239	255
136	152	168	184	200	216	232	248	144	160	176	192	208	224	240	256

(d)

Fig. 12 (continued)

1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255
16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256

(e)

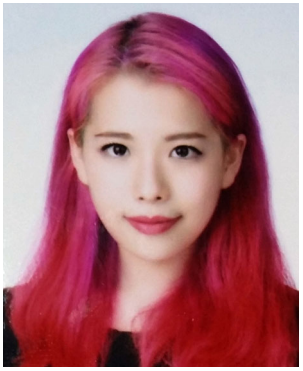
Fig. 12 (continued)

References

1. Belina S, Cuk V, Klapan I (2009) Virtual endoscopy and 3D volume rendering in the management of frontal sinus fractures. Coll Antropol 33:43–51
2. Chen K, Duan Y, Yan L, Sun J, Guo Z (2012) Efficient SIMD optimization of HEVC encoder over X86 processors. Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), pp 1–4
3. Cui J, Liu Y, Xu Y, Zhao H, Zha H (2013) Tracking generic human motion via fusion of low- and high-dimensional approaches. IEEE Trans Syst Man Cybern Syst 43(4):996–1002
4. Dachille F, Kreeger K, Chen B, Bitter I, Kaufman A (1998) High-quality volume rendering using texture mapping hardware. SIGGRAPH/Eurographics Workshop on Graphics Hardware'98, pp 69–76
5. Fang L, Wang Y, Qiu B, Qian Y (2002) Fast maximum intensity projection algorithm using shear warp factorization and reduced resampling. Magn Reson Med 47:696–700
6. Kiefer G, Lehmann H, Weese J (2006) Fast maximum intensity projections of large medical data sets by exploiting hierarchical memory architectures. IEEE Trans Inf Technol Biomed 10(2):385–394
7. Kye H (2009) Efficient maximum intensity projection using SIMD instruction and streaming memory transfer. J Korea Multimedia Soc 12(4):512–520
8. Lacroute P (1995) Fast volume rendering using a shear-warp factorization of the viewing transformation. Technical Report: CSL-TR-95-678
9. Lacroute P, Levoy M (1994) Fast volume rendering using a shear-warp factorization of the viewing transformation. Proceeding SIGGRAPH '94 Proceedings of the 21st annual conference on Computer graphics and interactive techniques, pp 451–458
10. Liu Y, Zhang X, Cui J (2010) Visual analysis of child-adult interactive behaviors in video sequences. 16th International Conference on Virtual Systems and Multimedia, pp 26–33
11. Liu Y, Cui J, Zhao H, Zha H (2012) Fusion of low-and high-dimensional approaches by trackers sampling for generic human motion tracking. 21st International Conference on Pattern Recognition, pp 898–901
12. Liu Y, Nie L, Han L, Zhang L, Rosenblum DS (2015) Action2Activity: recognizing complex activities from sensor data. Proceedings of the 24th International Conference on Artificial Intelligence, pp 1617-1623

13. Liu Y, Nie L, Liu L, Rosenblum DS (2016) From action to activity: sensor-based activity recognition. Neurocomputing 181(12):108–115
14. Liu L, Cheng L, Liu Y, Jia Y, Rosenblum DS (2016) Recognizing complex activities by a probabilistic interval-based model. Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, pp 1266–1272
15. Lu Y, Wei Y, Liu L, Zhong J, Sun L, Liu Y (2017) Towards unsupervised physical activity recognition using smartphone accelerometers. Multimedia Tools Appl 76(8):10701–10719
16. McFarlin DS, Arbatov V, Franchetti F, Püschel M (2011) Automatic SIMD vectorization of fast fourier transforms for the larrabee and AVX instruction sets. Proceedings of the international conference on Supercomputing, pp 265–274, Tucson, Arizona, USA
17. Mensmann J, Ropinski T, Hinrichs KH (2010) An advanced volume raycasting technique using GPU stream processing. International Conference on Computer Graphics Theory and Applications, pp 190–198
18. Mora B, Ebert DS (2005) Low-complexity maximum intensity projection. ACM Trans Graph 24(4):1392–1416
19. Mroz L, König A, Gröller E (1999) Real-time maximum intensity projection. Data Visualization '99, pp 135–144
20. Mroz L, Hauser H, Gröller E (2000) Interactive high-quality maximum intensity projection. Comput Graphics Forum 19(3):341–350
21. Pekar V, Hempel D, Kiefer G, Busch M, Weese J (2003) Efficient visualization of large medical image datasets on standard PC hardware. Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization, pp 135–140
22. Rezk-Salama C, Engel K, Bauer M, Greiner G, Ertl T (2000) Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage-rasterization. Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware'00, pp 109–118
23. Sabella P (1988) A rendering algorithm for visualizing 3D scalar fields. ACM SIGGRAPH 1988 Proceedings of the 15th annual conference on Computer graphics and interactive techniques, pp 51–58
24. Schreiner S, Galloway RL Jr (1993) A fast maximum-intensity projection algorithm for generating magnetic resonance angiograms. IEEE Trans Med Imaging 12(1):50–57
25. Vollrath JE, Weiskopf D, Ertl T (2005) A generic software framework for the gpu volume rendering pipeline. Proceedings of Vision, Modeling, and Visualization, pp 391–398
26. Zekri AS (2014) Enhancing the matrix transpose operation using intel avx instruction set extension. Int J Comput Sci Inf Technol (IJCSIT) 6(3):67–78
27. Zhao K, Sakamoto N, Koyamada K (2014) Fused visualization for large-scale time-varying volume data with adaptive particle-based rendering. AsiaSim 2014, 14th International Conference on Systems Simulation, pp 228–242

Heewon Kye is an associate professor in the Division of Computer Engineering at Hansung University, Korea. He received the BS (1999), MS (2001) and PhD degree (2005) in Computer Science and Engineering from Seoul National University, Korea, respectively. His current interests are real-time rendering, volume visualization, and medical image processing.



Se Hee Lee is a MS candidate in the Department of Information Systems Engineering at Hansung University, Korea. She received the BS degree (2016) in Department of Information Systems Engineering from Hansung University, Korea, respectively. Her current interests are real-time rendering, volume visualization, and medical image processing.

Jeongjin Lee is an associate professor in the School of Computer Science and Engineering at Soongsil University, Korea. He received the B.S. degree in mechanical engineering, and the M.S. and Ph.D. degrees in computer science and engineering from Seoul National University, Korea in 2002 and 2008, respectively. He worked as a research professor in Department of Radiology, University of Ulsan, Korea (Seoul Asan Medical Center) from October 2007 to February 2009. He also worked as a C.T.O. in Clinical Imaging Solution from January 2008 to May 2010. He worked as an assistant professor in Department of Digital Media, The Catholic University of Korea from March 2009 to February 2013. His research interests include image registration, image segmentation, computer-aided diagnosis, computer-aided surgery, volume rendering, and virtual endoscopy.