A Comparison of Two SIMD Implementations of the 2D Discrete Wavelet Transform

Asadollah Shahbahrami[1, 2] Ben Juurlink[1]
[1]Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, The Netherlands
Phone: +31 15 2787362. Fax: +31 15 2784898.
E-mail: {shahbahrami,benj,stamatis}@ce.et.tudelft.nl
[2]Department of Electrical Engineering, Faculty of Engineering, The University of Guilan, Rasht, Iran.

Abstract—

There are generally two algorithms to traverse an image to implement the 2D Discrete Wavelet Transform (DWT), namely *Row-Column Wavelet Transform* (RCWT) and *Line-Based Wavelet Transform* (LBWT). In the RCWT algorithm, the 2D DWT is divided into two 1D DWT: horizontal and vertical filtering. The horizontal filtering processes the rows of the original image and stores the wavelet coefficients in an auxiliary matrix. Thereafter, the vertical filtering phase processes the columns of the auxiliary matrix and stores the results back in the original matrix. In the LBWT algorithm, the vertical filtering is started as soon as a sufficient number of rows, as determined by the filter length, has been horizontally processed. In this paper, we provide answers to the following questions: first, which implementation is easier to vectorize using SIMD instructions? Second, which SIMD implementation provides more performance? Our initial results for Daubechies' transform with four coefficients show that the SIMD implementation of the LBWT algorithm is more complicated than the SIMD implementation of the RCWT algorithm, while the former algorithm is 1.60 times faster than the latter algorithm for an image of size 4096×4096.

Keywords: Discrete Wavelet Transform, Multimedia Extensions, SIMD.

I. Introduction

JPEG2000 is a wavelet-based image compression standard. This standard has some important features in compared to Discrete Cosine Transform (DCT) block-based JPEG standard. For example, the JPEG2000 standard provides performance superior at low bit rates, decomposes the image into a multiple resolution representation, and support region of interest coding [11]. The main reason why the JPEG2000 standard provides these features is due to using the Discrete Wavelet Transform (DWT). However, the DWT is the main time consuming function in the JPEG2000 standard and has higher computational requirements than the DCT. Our results that have been obtained by profiling the JasPer software tool kit [2] shows that the 2D DWT consumes on average 46% of the encoding time for lossless compression. For lossy compression, the DWT even requires 68% of the total encoding time on average. Results presented by other researchers [1, 8] also show that the 2D DWT is very time-consuming and consumes a significant part of the total JPEG2000 encoding time. Consequently, improving the performance of the 2D DWT is an important issue to increase the performance of the multimedia compression standard.

One way to improve the performance of the 2D DWT is exploiting the Data Level Parallelism (DLP) by vectorization. This is because there is DLP in this application. Vectorization determines and extracts DLP, which employs the ability to execute the Single Instruction on Multiple Data (SIMD) elements concurrently. Recently, general-purpose processors have been enhanced by the SIMD instructions such as Pentium 4, which includes the SSE instruction set [19].

There are generally two algorithms to traverse an image to implement the 2D DWT, namely *Row-Column Wavelet Transform* (RCWT) and *Line-Based Wavelet Transform* (LBWT) [5, 12]. In the RCWT approach, the 2D DWT is divided into two 1D DWT, namely *horizontal filtering* and *vertical filtering*. The horizontal filtering filters whole rows of an image followed by vertical filtering processes the columns. The LBWT algorithm uses a single loop to process both rows and columns together.

In this paper, we provide answers to the following questions: first, which implementation is easier to vectorize using SIMD instructions? Second, which SIMD implementation provides more performance? Our initial results for Daubechies' transform with four coeffi-

This research was supported in part by the Netherlands Organisation for Scientific Research (NWO).

cients [24] show that the SIMD implementation of the LBWT is 1.60 times faster than RCWT for an image of size 4096×4096, while the former implementation is more complicated than the latter implementation.

This paper is organized as follows. Section II describes the DWT and discusses the different techniques to traverse an image to implement it. Section III and Section IV describe the SIMD vectorization of the RCWT and LBWT algorithms, respectively. The experimental evaluation is illustrated in Section V. Related work is discussed in Section VI. Finally, the paper ends with some conclusions in Section VII.

II. Background

This section describes the DWT as well as RCWT and LBWT techniques.

A. *Discrete Wavelet Transform*

The DWT was introduced by Crochiere et al. in 1976 [14]. The basic idea is the partitioning of the image signal spectrum into several frequency bands that are coded and transmitted separately. The DWT provides a time-frequency representation of a signal. The wavelet representation of a discrete signal X consisting of N samples can be computed by convolving X with the low-pass and high-pass filters and down-sampling the output signal by 2, so that the two frequency bands each contain $N/2$ samples. With the correct choice of filters, this operation is reversible. This process should be applied in both horizontal and vertical directions for 2D signals. It decomposes the original image into four subbands denoted by LL, LH, HL, and HH, containing both low and high frequency components [23]. In other words, this transform is computed by performing lowpass and highpass filtering of the image pixels as shows in Figure 1. The low pass and high pass filters are denoted by h and g, respectively. Figure 1 depicts the three levels DWT decomposition. At each level, the high pass filter generates detail image pixels information, while the low pass filter produces the coarse approximations of the input image. For an $N \times M$ image, there are exactly NM wavelet coefficients the same as the number of image pixels.

There are different algorithms to implement 2D DWT such as traditional convolution-based and lifting scheme methods. The convolutional methods apply filtering by multiplying the filter coefficients with the input samples and accumulating the results. Their implementation is almost similar to Finite Impulse Response (FIR) implementation. The Daubechies' transform with four coefficients [24] (Daub-4) and the Cohen, Daubechies and Feauveau 9/7 filter [13] (CDF-9/7) are examples of this category. The implementations of the convolutional methods such as Daub-4 and CDF-9/7 are similar. In this work, we have focused on the implementation of the Daub-4 transform. Nevertheless, the proposed methodology is general and equally applicable to other transforms.

The basic idea of the lifting scheme is to use the correlation in the image pixels values to remove the redundancy [15, 16]. In this paper, we focus on the convolution-based transforms. In addition, we suppose that the image data is stored as row-major order in the memory.

In addition, there are different algorithms to traverse an image to implement 2D DWT, namely *Row-Column Wavelet Transform* (RCWT) and *Line-Based Wavelet Transform* (LBWT) [3–6, 12]. These algorithms are discussed in the following sections.

B. *Row-Column Wavelet Transform Algorithm*

In the RCWT algorithm, the 2D DWT is divided into 2 1D DWT, namely *horizontal filtering* and *vertical filtering*. The horizontal filtering usually processes the rows of the original image and stores the wavelet coefficients in an auxiliary matrix. Thereafter, the vertical filtering phase processes the columns of the auxiliary matrix and stores the results back to the original matrix. In other words, this algorithm requires that all lines be horizontally filtered before the vertical filtering starts. In addition, the computational complexity of both horizontal filtering and vertical filtering is the same. Each of these filtering is applied separately. Each $N \times M$ matrix requires NMc_{dwt} bytes of memory, where c_{dwt} denotes the number of bytes that represent one wavelet coefficient in memory. Figure 2 depicts both horizontal and vertical filtering of this algorithm.

Figure 3 and Figure 4 depict the C implementation of horizontal and vertical filtering, respectively, using the Daub-4 transform for an $N \times M$ image. Both low- and high pass filter coefficients have been rounded to four decimal points. Array *low* and *high* in figures store these values. It is important to note that the loop in the implementation of the vertical filtering in Figure 4 has been interchanged. This is because the straightforward implementation, which processes each column entirely before advancing to the next column is not able to exploit spatial locality. In order to improve spatial locality *loop interchange* has been ap-

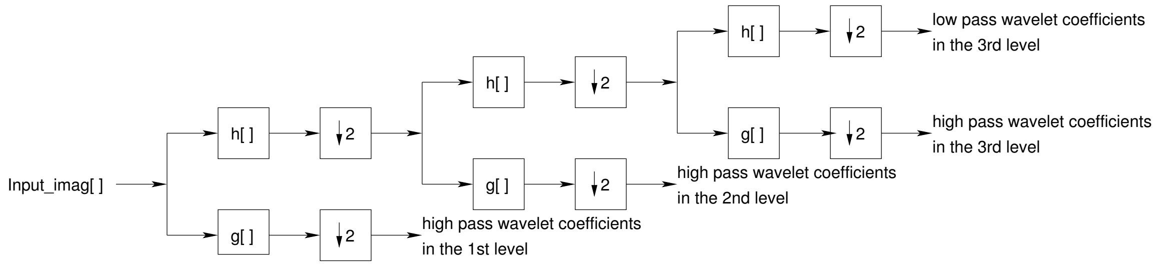
Input_imag[ ]
h[ ]
g[ ]
↓2
high pass wavelet coefficients in the 1st level
high pass wavelet coefficients in the 2nd level
low pass wavelet coefficients in the 3rd level
high pass wavelet coefficients in the 3rd level

Fig. 1. Three level DWT decomposition of an input image using filtering approach.

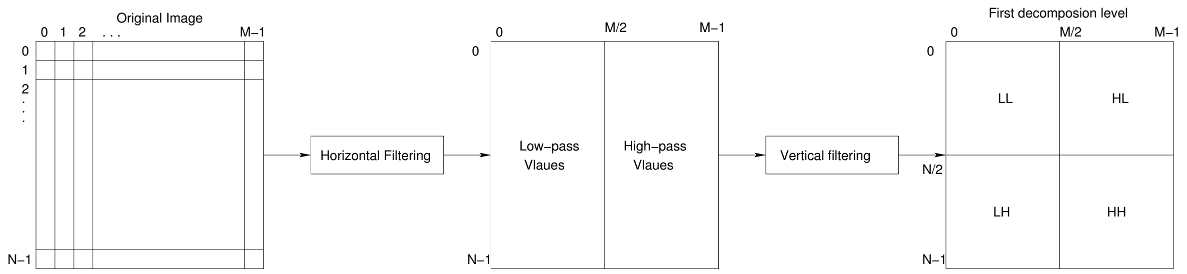

Fig. 2. Horizontal and vertical filtering is separately applied in the row-column wavelet transform technique.

plied, which is a well-known compiler technique. The loop interchange technique places the loop with index j after the loop with index i allowing to process the same rows successively, thereby helping to reduce cache misses. The experimental results that have been presented in [20,21] clearly show that the implementations with interchanged loops are much more efficient than the straightforward implementations. For this reason, we have considered this algorithm for vertical filtering of the RCWT method.

C. Line-Based Wavelet Transform Algorithm

The LBWT algorithm uses a single loop to process both rows and columns together. The horizontal filtering filters sufficient number of rows and stores the processed coefficients of low-pass and high-pass values in place in a small buffer. Thereafter, the vertical filtering is started as soon as sufficient number of rows, as determined by the filter length, have been horizontally processed. Figure 5 depicts this algorithm.

The algorithm processes L rows, where L is number of filter length, and store the low- and high-pass values interleave in a buffer of size $L \times M$. Thereafter, the columns of this small buffer are processed and the calculated wavelet coefficients are stored in different subbands in an auxiliary matrix in the order expected by the quantization step. This means that the separation of the interleaved subbands into separate low- and high-pass subbands is implicitly performed without an extra rearrangement step. In general, this algorithm has three parts, namely prolog, main, and epilog parts. The prolog and epilog are the beginning and the end parts of the algorithm. These parts are implemented separately. For example, for Daub-4 transform, two input image rows are horizontally processed in the prolog part. The main part participates the most codes of the program. Figure 6 represents a part of the C implementation of the main part, for the Daub-4 transform. As this figure depicts, there is an outer loop with index i. Inside this loop, there are two inner loops. The first inner loop with index j is related to the horizontal filtering on the input matrix, and the other inner loop with also index j is related to the vertical filtering on the calculated results from previous loop. In each iteration, the horizontal filtering processes two consecutive image rows, and it passes four rows of calculated wavelet coefficients to the vertical filtering.

III. Vectorization of the RCWT algorithm

This section describes the vectorization of the RCWT algorithm in order to utilize the SIMD instructions.

The SIMD implementation of the horizontal filtering is more difficult than the vertical filtering. In other words, vectorization of the horizontal filtering involves a substantial reordering of operations. To explain the reason for this, Figure 7 depicts the data flow graph of the horizontal filtering, where $x_{0i}, 0 \leq i < 8$ are the input samples and $c_0, \ldots, c_3$ denote the filter coefficients. As this figure shows, four different input samples are multiplied with four different coefficients. The

```
void Daub_4_horizontal() {
int i, j, jj;
float low[] ={-0.1294, 0.2241, 0.8365 , 0.4830};
float high[]={-0.4830, 0.8365, -0.2241, -0.1294};
for (i=0; i<N; i++)
  for(j=0, jj=0; jj<M; j++, jj +=2) {
    ou_image[i][j]             = in_image[i][jj]     * low[0]  + in_image[i][jj + 1]  * low[1]
                               + in_image[i][jj + 2] * low[2]  +  in_image[i][jj + 3] * low[3];

    ou_image[i][j  + M/2]      = in_image[i][jj]     * high[0] + in_image[i][jj + 1]  * high[1]
                               + in_image[i][jj + 2] * high[2] + in_image[i][jj + 3]  * high[3];
  }
}
```

Fig. 3. C implementation of horizontal filtering using the Daub-4 transform.

```
void Daub_4_vertical() {
int i, j, jj;
float low[] ={-0.1294, 0.2241, 0.8365 , 0.4830};
float high[]={-0.4830, 0.8365, -0.2241, -0.1294};
for (i=0, ii=0; ii<N; i++, ii +=2)
  for(j=0; j<M; j++) {
    in_image[i][j]  = ou_image[ii][j]   * low[0]  + ou_image[ii+1][j] * low[1]  + ou_image[ii+2][j] * low[2]
                    + ou_image[ii+3][j] * low[3];

    in_image[i+N/2][j] = ou_image[ii][j]  * high[0] + ou_image[ii+1][j]* high[1]+ou_image[ii+2][j] * high[2]
                       + ou_image[ii+3][j]* high[3];
 }
}
```

Fig. 4. C implementation of vertical filtering using the Daub-4 transform. Note that the loops have been interchanged w.r.t. the straightforward implementation.

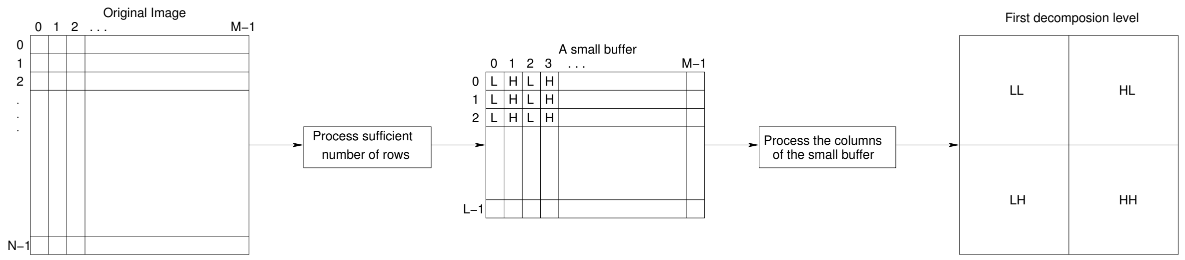
Original Image
0 1 2 ... M−1
0
1
2
N−1
Process sufficient number of rows
A small buffer
0 1 2 3 ... M−1
L H L H
L−1
Process the columns of the small buffer
First decomposition level
LL
HL
LH
HH

Fig. 5. Processing both rows and columns in the line-based wavelet transform technique.

intermediate results are accumulated into one destination operand, while there are no such SIMD instructions in the SSE extension.

To vectorize the horizontal filtering, the data flow graph in Figure 7 must be converted so that four even-numbered values or four odd-numbered values should be stored in an SIMD register. For example, four even-numbered values of $x00, x02, x04, x06$ and four odd-numbered values of $x01, x03, x05, x07$ are separately stored in two SIMD registers. Many overhead instructions are needed to reorder these input sequences. Figure 8 depicts some of these overhead instructions that used in the SIMD implementation of the horizontal filtering using the SSE extension.

Another way of vectorizing the horizontal filtering is using transposition. However, the overhead cost that is involved for the transposition is larger than the benefits of vectorization.

On the other hand, vectorization of the vertical filtering is much easier than the horizontal filtering. This is because the image pixels that can be processed simultaneously are stored consecutively in memory. In vertical filtering, instead of a single sample of a single column, four horizontally neighboring samples are read from memory into a packed register. Figure 9 illustrates the data flow graph of the vertical filtering. It can be seen that four different input samples of each row are multiplied with one filter coefficient simulta-

```
  void Both_Hori_Ver() {
  int i, j, jj;
  float low[] ={-0.1294, 0.2241, 0.8365 , 0.4830};
  float high[]={-0.4830, 0.8365, -0.2241, -0.1294};
  for (i=0, ii=0; i<N; ii++, i +=2) {
    k = ( ii % 2 ) * 2;
    for(j=0, jj=0; jj<M; j++, jj +=2) {
      BufLow[k][jj]        = in_image[i][jj]       * low[0]   + in_image[i][jj + 1] * low[1]  +
                             in_image[i][jj + 2]   * low[2]   + in_image[i][jj + 3] * low[3];
      BufLow[k][jj + 1]    = in_image[i][jj]       * high[0]  + in_image[i][jj + 1] * high[1] +
                             in_image[i][jj + 2]   * high[2]  + in_image[i][jj + 3] * high[3];

      BufLow[k+1][jj]      = in_image[i+1][jj]      * low[0]  +  in_image[i+1][jj + 1] * low[1]  +
                             in_image[i+1][jj + 2]  * low[2]  +  in_image[i+1][jj + 3] * low[3];
      BufLow[k+1][jj + 1]  = in_image[i+1][jj]      * high[0] + in_image[i+1][jj  + 1] * high[1] +
                             in_image[i+1][jj + 2]  * high[2] + in_image[i+1][jj  + 3] * high[3];
    }
    for(j=0, jj=0; jj<M; j++, jj +=2) {
      ou_image[ii][j]       = BufLow[0][jj]   * low[0]  +  BufLow[1][jj] * low[1]  +
                              BufLow[2][jj]   * low[2]  +  BufLow[3][jj] * low[3];
      ou_image[ii + N/2][j] = BufLow[0][jj]   * high[0] +  BufLow[1][jj] * high[1] +
                              BufLow[2][jj]   * high[2] +  BufLow[3][jj] * high[3];

      ou_image[ii][j + M/2]  = BufLow[0][jj + 1]  * low[0] +  BufLow[1][jj + 1] * low[1]      +
                               BufLow[2][jj + 1]  * low[2] +  BufLow[3][jj + 1] * low[3];

      ou_image[ii + N/2][j + M/2] = BufLow[0][jj + 1] * high[0] + BufLow[1][jj + 1] * high[1] +
                                    BufLow[2][jj + 1] * high[2] + BufLow[2][jj + 1] * high[3];
  }
 }
}
```

Fig. 6. A part of the C implementation of the line-based wavelet transform algorithm for the Daub-4 transform.

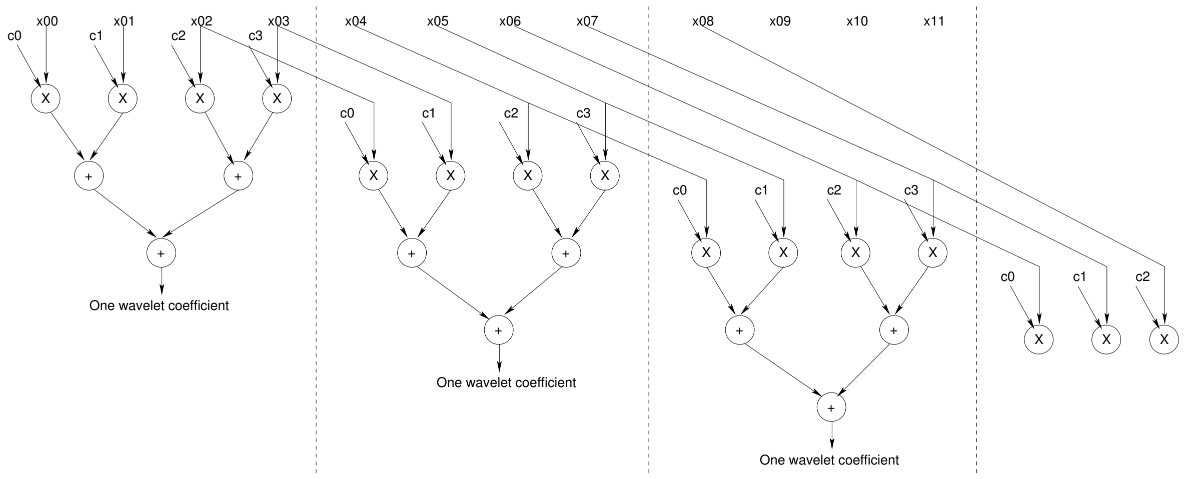

Fig. 7. Data flow graph of the horizontal filtering of the Daub-4 transform.

```
movaps   xmm0, (esi)
movaps   xmm1,16(esi)
movaps   xmm2, xmm0
unpcklps xmm0, xmm1
unpckhps xmm2, xmm1
movaps   xmm1, xmm0
unpcklps xmm0, xmm2
unpckhps xmm1, xmm2
movups   xmm2,8(esi)
movups   xmm3,24(esi)
movaps   xmm4, xmm2
unpcklps xmm2, xmm3
unpckhps xmm4, xmm3
movaps   xmm3, xmm2
unpcklps xmm2, xmm4
unpckhps xmm3, xmm4
movaps   xmm4, xmm0
movaps   xmm5, xmm1
movaps   xmm6, xmm2
movaps   xmm7, xmm3
```

Fig. 8. SSE instructions needed to rearrange the input sequences for the horizontal filtering of the Daub-4 transform.

neously. Each filter coefficient should be spread across four different subwords of a media register. After four multiplications of four consecutive rows with different coefficients, the results of each column are added to each other. Finally, four wavelet coefficients are calculated simultaneously. There are SIMD instructions in the SIMD architectures for these operations.

IV. Vectorization of the LBWT Algorithm

For SIMD implementation of the LBWT, we have defined a circular queue buffer of size $L \times M$, where L is the filter length and M is the image height. In the LBWT algorithm, $L-2$ rows of the input image are horizontally processed in the prolog part and stored in the $L-2$ rows of the buffer. In the main loop of the program, first, two extra rows of the input image are horizontally processed and stored in the last two rows of the buffer. Second, the filled buffer is vertically filtered and the calculated wavelet coefficient are stored in an auxiliary matrix in the order expected by the quantization step. In the next iteration of the main loop, only two rows of the input matrix are horizontally processed and stored in the first two rows of the buffer. This is because the remaining $L-2$ rows from 2 to $L-1$ are reused for the next iteration. In addition, for another iteration, rows from 4 to $L-1$ and rows 0 and 1 are reused. We have used this buffer as a circular queue and in each iteration two rows are sequentially replaced with wavelet coefficient, which have been horizontally obtained. Figure 10 depicts three iterations of this algorithm. As this figure shows, each four low-pass and four high-pass values are interleaved. This is because of the 4-way SIMD implementation.

Processor	Intel Pentium 4
CPU Clock Speed	3.0GHz
L1 Data Cache	8 KBytes, 4-way set associative, 64 Bytes line size
L2 Cache	512 KBytes, 8-way set associative, 64 Bytes line size, On Chip

TABLE I
Parameters of the experimental platform.

V. Performance Evaluation

In this section, we evaluate the performance of the RCWT and LBWT algorithms.

A. Experimental Setup

Four programs have been implemented. Two programs have been completely written in C. One performs the 2D DWT using RCWT algorithm, the other performs the 2D DWT using LBWT algorithm. These programs will be referred to as C-RCWT and C-LBWT, respectively. They were compiled using the gcc compiler with optimization level *-O2*. Other two programs have been written using SSE instruction set. These programs are the vectorized versions of the C-RCWT and C-LBWT programs, which are referred to as SSE-RCWT and SSE-LBWT, respectively. In all programs, the first level decomposition has been implemented and the input images are considered as a single tile.

As experimental platform, we have employed a 3.0GHz Pentium 4 processor. The main architectural parameters of our system are summarized in Table I.

All programs were executed on a lightly loaded system. Performance was measured using the IA-32 cycle counter [17]. Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program [7, 22]. In order to eliminate the effects of context switching and compulsory cache misses, the *K-best* measurement scheme and a *warmed up* cache have been used [7]. That means that the function is repeatedly (K times) executed and the fastest time is recorded.

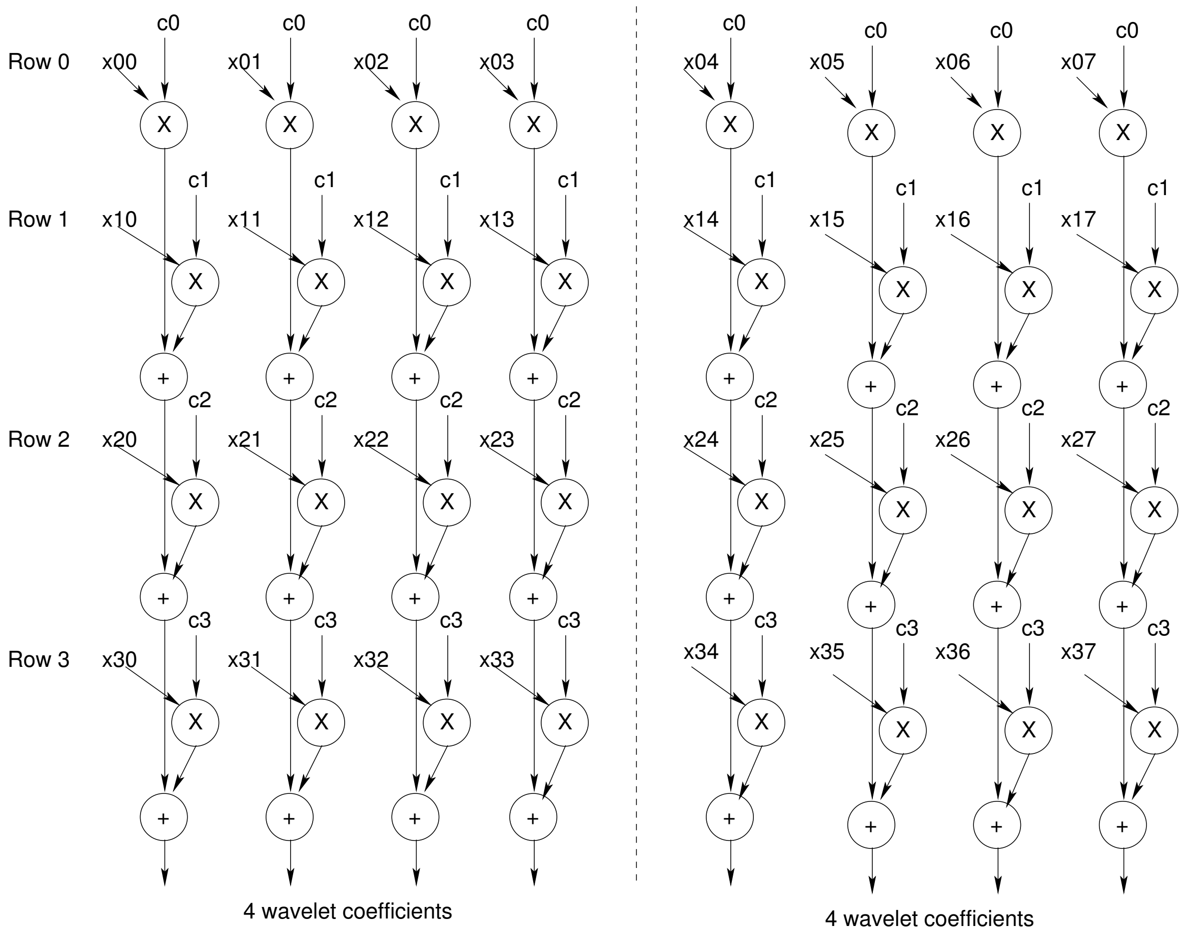
Row 0
Row 1
Row 2
Row 3
c0
c1
c2
c3
x00 x01 x02 x03 x04 x05 x06 x07
x10 x11 x12 x13 x14 x15 x16 x17
x20 x21 x22 x23 x24 x25 x26 x27
x30 x31 x32 x33 x34 x35 x36 x37
X
+
4 wavelet coefficients
4 wavelet coefficients

Fig. 9. Data flow graph of the vertical filtering of the Daub-4 transform.

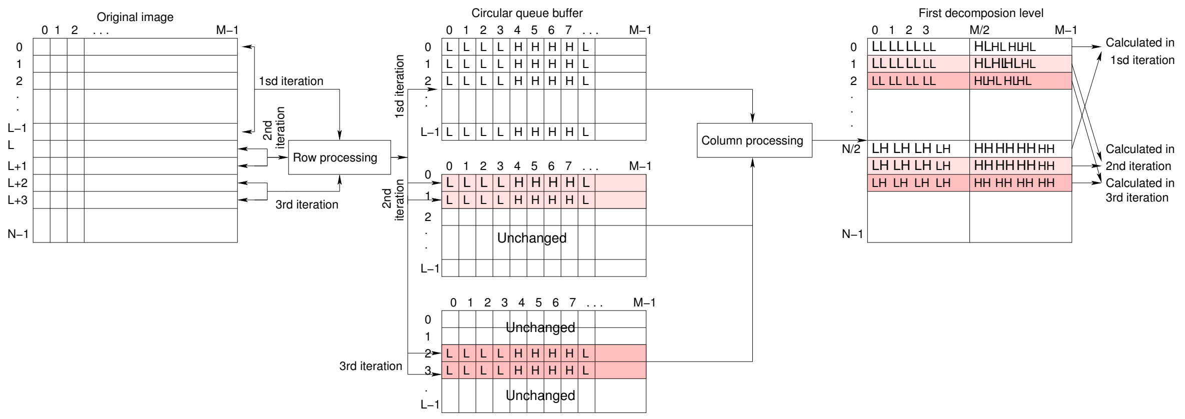

Fig. 10. Three iterations of the SIMD implementation of the line-based wavelet transform using 4-way parallelism.

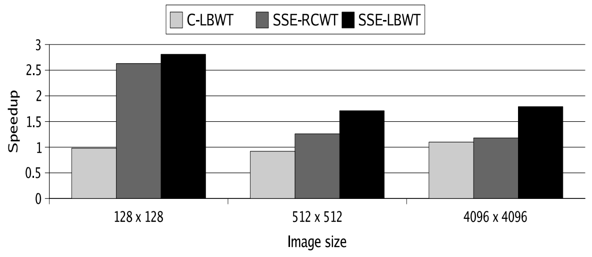

Fig. 11. Speedup of the C-LBWT and SSE-RCWT over C-RCWT as well as the speedup of the SSE-LBWT over C-LBWT for the first level decomposition of the 2D DWT using different image size.

Executing the function at least once before starting the measurement minimizes the effects of both instruction and data cache misses.

B. Performance Evaluation Result

Figure 11 depicts the speedup of the C-LBWT and SSE-RCWT over C-RCWT as well as the speedup of the SSE-LBWT over C-LBWT for the first level decomposition of the 2D DWT. The performance of the C-LBWT program is almost the same as C-RCWT. The speedup of the SSE-RCWT implementation for 128×128 image size is 2.63, while the speedup for image sizes larger than 128×128 is 1.22 on average. The main reason for this is that for small image sizes, almost all reads hit the L1 data cache except for compulsory misses. Therefore, the speedup obtained is the speedup resulting from SIMD vectorization. For larger image sizes, the speedup decreases because this SIMD implementation has become memory-bound. In other words, in the vertical filtering, four input rows that are needed to compute one output row can be kept in cache for small image sizes, while for larger image sizes they cannot. This means that for larger image sizes, in addition to compulsory misses, there are capacity misses.

As can be seen in Figure 11, the speedup of the SSE-LBWT over the corresponding C implementation for 128×128 image size is 2.81, while for large image sizes it is 1.75 on average. In general, the SSE-LBWT implementation yields more performance than SSE-RCWT. The performance improvement of the SSE-LBWT over SSE-RCWT ranges from 1.10 to 1.60.

C. Discussion

The SSE-LBWT improves performance more than the SSE-RCWT, while its implementation is more complicated. As mentioned in Section II-C, this algorithm has three phases: prolog, main, and epilog. The horizontal filtering is implemented in both prolog and main parts. In the prolog part, the horizontal filtering is used to process two input image rows. The vertical filtering is also implemented in both main and epilog phases. The vertical filtering is repeated once in the epilog part to calculate the last output row. In other words, the prolog and epilog parts are employed to correctly handle the first and last output rows. The SIMD implementations of these parts have to be implemented individually. This makes the code size of the program increase. In addition, there are many repetitions of instructions. For example, each horizontal and vertical processing has to be implemented twice.

Although, the speedup of the SSE-RCWT is smaller than the speedup of the SSE-LBWT, its code size is smaller than SSE-LBWT. This is because each horizontal and vertical filtering is implemented only once. In addition, there are no any prolog and epilog parts in this algorithm.

VI. Related Work

SIMD vectorization of the 2D DWT has been considered in [9,10,18]. Chaver et al. [9] used SSE and the CDF-9/7 filter. They focused on automatic vectorization and did not consider assembly-level programming. The Intel compiler, however, can only vectorize simple loops, and therefore some manual code modifications had to be performed. Furthermore, only horizontal filtering could be automatically vectorized (they assumed column-major order). They also combined aggregation with a line-based approach for their SIMD implementation. In [10] they have vectorized vertical filtering of CDF-9/7 by hand using built-in SSE functions. In order to do so, however, an additional data transposition stage was required, which reduces the benefits of SIMD vectorization.

Kutil [18] has implemented the $(9, 7)$ lifting scheme using built-in SSE functions. He proposed a single loop approach to SIMD vectorization. In this approach horizontal and vertical filtering are combined into a single loop. This is called line-based computation in [12] and pipeline computation in [9], where it has been used to vectorize the CDF-9/7 transform. The single-loop approach requires a buffer whose size is equal to 16 rows of data. If this buffer does not fit in the cache, the temporal locality will be reduced.

VII. Conclusions

In this paper, we have focused on comparing different SIMD implementations of the 2D DWT. There are generally two algorithms to traverse an image to implement the 2D DWT. The first algorithm is row-column wavelet transform, which divides the 2D DWT into two 1D DWT, namely horizontal and vertical filtering. The horizontal filtering filters the whole rows of the original image and stores the intermediate results in an auxiliary matrix. Thereafter, the vertical filtering filters the whole columns of the auxiliary matrix and stores the results back in the original matrix. On the other hand, in the LBWT algorithm, the vertical filtering is started as soon as a sufficient number of rows have been horizontally filtered. In this research paper, we have found that the vectorization of the RCWT algorithm is easier than LBWT, while the performance improvement of the SIMD implementation of LBWT is larger than RCWT. For instance, our results for Daub-4 transform showed that the SIMD implementation of the LBWT is 1.60 times faster than the SIMD implementation of the RCWT for an image of size 4096×4096.

References

[1] M. D. Adams and F. Kossentini. JasPer: A Software-Based JPEG-2000 Codec Implementation. In *Proc. IEEE Int. Conf. on Image Processing*, pages 53–56, October 2000.

[2] M. D. Adams and R. K. Ward. JasPer: A Portable Flexible Open-Source Software Tool Kit for Image Coding/Processing. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 5, pages 241–244, May 2004.

[3] Y. Andreopoulos, K. Masselos, P. Schelkens, G. Lafruit, and J. Cornelis. Cache Misses and Energy-Dissipation Results for JPEG-2000 Filtering. In *Proc. 14th IEEE Int. Conf. on Digital Signal Processing*, pages 201–209, 2002.

[4] Y. Andreopoulos, P. Schelkens, and J. Cornelis. Analysis of Wavelet Transform Implementations for Image and Texture Coding Applications in Programmable Platforms. In *Proc. IEEE Signal Processing Systems*, pages 273–284, 2001.

[5] Y. Andreopoulos, P. Schelkens, G. Lafruit, K. Masselos, and J. Cornelis. High-Level Cache Modeling for 2-D Discrete Wavelet Transform Implementations. *Journal of VLSI Signal Processing*, 34:209–226, 2003.

[6] Y. Andreopoulos, N. D. Zervas, G. Lafruit, P. Schelkens, T. Stouraitis, C. E. Goutis, and J. Cornelis. A Local Wavelet Transform Implementation Versus an Optimal Row-Column Algorithm for the 2D Multilevel Decomposition. In *Proc. IEEE Int. Conf. on Image Processing*, volume 3, pages 330–333, 2001.

[7] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2003.

[8] S. Chatterjee and C. D. Brooks. Cache-Efficient Wavelet Lifting in JPEG 2000. In *Proc. IEEE Int. Conf. on Multimedia*, pages 797–800, August 2002.

[9] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. 2-D Wavelet Transform Enhancement on General-Purpose Microprocessors: Memory Hierarchy and SIMD Parallelism Exploitation. In *Proc. Int. Conf. on the High Performance Computing*, December 2002.

[10] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. Vectorization of the 2D Wavelet Lifting Transform Using SIMD Extensions. In *Proc. 17th IEEE Int. Symp. on Parallel and Distributed Image Processing and Multimedia*, 2003.

[11] C. Christopoulos, A. Skodras, and T. Ebrahimi. The JPEG2000 Still Image Coding System: An Overview. *IEEE Trans. on Consumer Electronics Euromicro Conference*, 46(4):1103–1127, November 2000.

[12] C. Chrysafis and A. Ortega. Line-Based, Reduced Memory, Wavelet Image Compression. *IEEE Trans. on Image Processing*, 9(3):378–389, March 2000.

[13] A. Cohen, I. Daubechies, and J. C. F. Eauveau. Biorthogonal Bases of Compactly Supported Wavelets. *Communications on Pure and Appl. Math.*, 45(5):485–560, June 1992.

[14] R. Crochiere, S. Webber, and J. Flanagan. Digital Coding of Speech in Subbands. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, pages 233–236, April 1976.

[15] I. Daubechies and W. Sweldens. Factoring Wavelet Transforms into Lifting Steps. *Journal of Fourier Analysis and Applications*, 4(3):247–269, 1998.

[16] M. Ferretti and D. Rizzo. A Parallel Architecture for the 2-D Discrete Wavelet Transform with Integer Lifting Scheme. *Journal of VLSI Signal Processing*, 28:165–185, 2001.

[17] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual Volume 3 System Programming Guide*, 2004. Order Number: 253668.

[18] R. Kutil. A Single-Loop Approach to SIMD Parallelization of 2D Wavelet Lifting. In *Proc. 14th Euromicro Int. Conf. on Parallel, Distributed, and Netwrok-Based Processing (PDP06)*, pages 413–420, February 2006.

[19] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium 3 Processor. *IEEE Micro*, pages 47–57, July-August 2000.

[20] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. Performance Comparison of SIMD Implementations of the Discrete Wavelet Transform. In *Proc. 16th IEEE Int. Conf. on Application-Specific Systems Architectures and Processors (ASAP)*, July 2005.

[21] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. Improving the Memory Behavior of Vertical Filtering in the Discrete Wavelet Transform. In *Proc. 3rd ACM Int. Conf. on Computing Frontiers*, pages 253–260, May 2006.

[22] D. B. Stewart. Measuring Execution time and Real-Time Performance. In *Embedded Systems Conf.*, pages 1–15, April 2001.

[23] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for Computer Graphics: Theory and Applications.* Morgan Kaufmann, 1996.

[24] M. A. Trenas, J. Lopez, E. L. Zapata, and F. Arguello. A Memory System Supporting the Efficient SIMD Computation of the Two Dimensional DWT. In *Proc. IEEE Int. Conf. on Acoustics Speech and Signal Processing*, volume 3, pages 1521–1524, May 1998.