Parallel Numerics '05, 141-149 M. Vajteršic, R. Trobec, P. Zinterhof, A. Uhl (Eds.)
Chapter 6: Multimedia ISBN 961-6303-67-8

SIMD Parallelization of Common Wavelet Filters

Rade Kutil*, Peter Eder, Markus Watzl

Department of Scientific Computing
University of Salzburg
Jakob Haringerstr. 2, 5020 Salzburg, Austria

Much work has been done to optimize wavelet transforms for SIMD extensions of modern CPUs. However, these approaches are always restricted to the vertical part of 2-D transforms with line-wise organized memory layouts because this leads to a rather straight forward SIMD-implementation. This work shows for some common wavelet filters that SIMD operations can also be used on 1-D transforms and produce reasonable speedups. As a result, the performance of algorithms that use wavelet transforms, such as JPEG2000, can be increased significantly.

1 Introduction

The wavelet transform is a well-established method used in many applications in signal processing and multimedia processing and compression [1]. It provides a redundancy-free time-frequency representation in real coefficients and has several advantages over related transforms. First, in contrast to blocked DCT (discrete cosine transform) coefficient manipulation does not imply unpleasant blocking artifacts. Second, the discrete orthogonal variants with finitely supported basis functions have linear computational complexity $O(n)$ while that of DCT or other Fourier-related transforms is $O(n \log n)$. The reason for the optimal complexity of the latter variants is the applicability of multiresolution methods. In this case, the wavelet transform can be implemented in terms of a hierarchical application of FIR filter banks together with down-sampling of the low-frequency parts.

*Corresponding author. E-mail: rkutil@cosy.sbg.ac.at

Despite the speed of the algorithm it is still demandable to investigate speedup techniques, since many applications have to satisfy realtime constraints and processed data sets are becoming larger. While a significant amount of work has been done for MIMD parallelization [2, 3, 4] and old SIMD arrays [5, 6, 7], the use of SIMD extensions of modern general purpose processors for wavelet transforms [8, 9] is still improvable as is shown in this work.

The wavelet transform is divided into several levels, each of which consists of the application of a quadrature mirror filter pair. Common filters have 6 to 12 taps. For 2-D data a horizontal and a vertical filtering of each row and column has to be performed at every level. If the memory layout is such that horizontally neighbored data is placed next to each other then the vertical transform can be SIMD-enabled easily by performing the sequential algorithm on vectors of horizontally neighbouring values instead of scalar values [8, 9]. The horizontal filtering is not so straight forward to parallelize. The same problem arises in 1-D transforms and applications with memory constraints [10]. The reason for this is that consecutive data that is read into a single packed word requires changing treatment because of badly aligned filters and down-sampling. Therefore, data has to be rearranged within packed registers and packed filter vectors have to be set properly. This work presents approaches for some common filters and shows that reasonable speedups can be achieved.

2 The Haar Filter

The Haar filter is the most simple orthogonal wavelet filter. It is a 2-tap filter. We consider it in this section to explain the basic approach to SIMD-parallelization of wavelet filters. The coefficients are $(a, a) = (\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2})$ in the low-pass form and $(a, -a) = (\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2})$ in the high-pass form. Together with down-sampling by a factor of 2 the following assignments, which have to be executed for all i, define the filtering process of the Haar wavelet transform.

$$L_i \leftarrow ax_{2i} + ax_{2i+1},\ H_i \leftarrow ax_{2i} - ax_{2i+1}$$

L and H are the low-pass and the high-pass subbands respectively. As a first sequential improvement we can reuse already computed products, which leads to

$$p \leftarrow ax_{2i},\ q \leftarrow ax_{2i+1},\ L_i \leftarrow p + q,\ H_i \leftarrow p - q\ .$$

We see that for each pair L_i, H_i of output values we have to read two input values x_{2i}, x_{2i+1}. Since it is reasonable to read and write only full packed

words when using SIMD, we consequently have to read two packed words in each iteration. We denote packed multiplication and addition by $\odot$ and $\oplus$ respectively. To access packed words and to rearrange data in packed registers (shuffle) we use the notation $y_{(i_0,\ldots,i_m)} := (y_{i_0}, \ldots, y_{i_m})$. Thus, we can write the SIMD parallelization of the Haar filter for word size 4 as

$$\begin{aligned} &p \leftarrow x_{(8i,\ldots,8i+3)} \odot (a,a,a,a), && q \leftarrow x_{(8i+4,\ldots,8i+7)} \odot (a,a,a,a), \\ &r \leftarrow (p,q)_{(0,2,4,6)}, && s \leftarrow (p,q)_{(1,3,5,7)}, \\ &L_{(4i,\ldots,4i+3)} \leftarrow r \oplus s, && H_{(4i,\ldots,4i+3)} \leftarrow r \ominus s\,. \end{aligned}$$

In the first line two perfectly aligned packed words are read and each element is immediately multiplied by the coefficient a with a single packed multiply operation for each word. In the second line the elements are rearranged into one word containing all even elements and one containing all uneven elements using shuffle operations. To calculate the sum and difference of every two neighbouring elements, we just have to add and subtract the two words, which is done in the third line.

While the sequential algorithm requires two multiplies and two additions (or subtractions) for every two input values, the SIMD version requires two packed multiplies and two packed additions for every eight input values. This makes a theoretical speedup of 4. However, since the shuffle operations also require some execution time and memory access can be a bottle-neck, the speedup is reduced. On an Intel Pentium 4 CPU with 2.8GHz using the SSE extension with packed words of 4 single precision numbers we get a speedup of 2.7 for the forward transform and 1.3 for the backward transform.

In the following sections we will discuss more complicated examples with more practical relevance. Note that all examples will show the same phases: memory read, coefficient multiplication, data rearrangement, summation and memory write. Some will have a different order of execution, though. Especially coefficient multiplication and data rearrangement will be interchanged.

3 Biorthogonal 4/12

The biorthogonal 4/12 filter is an example for an even, symmetrical filter. We will use the following symbols for abbreviation of the coefficients:

$$\begin{aligned} &\alpha \leftarrow 0.0138107,\ \beta \leftarrow 0.041432,\ \gamma \leftarrow 0.0524806,\ \delta \leftarrow 0.267927, \\ &\epsilon \leftarrow 0.0718155,\ \zeta \leftarrow 0.966748,\ \iota \leftarrow 0.176777,\ \kappa \leftarrow 0.53033\,. \end{aligned}$$

The downsampling factor of 2 and the offsets of -5 for low-pass and -1 for high-pass yields the following formulas for the low-pass subband L and the

high-pass subband H:

$$\begin{aligned} L_i \leftarrow & -\alpha x_{2i-6} + \beta x_{2i-5} + \gamma x_{2i-4} - \delta x_{2i-3} - \epsilon x_{2i-2} + \zeta x_{2i-1} \\ & + \zeta x_{2i} - \epsilon x_{2i+1} - \delta x_{2i+2} + \gamma x_{2i+3} + \beta x_{2i+4} - \alpha x_{2i+5}, \\ H_i \leftarrow & \iota x_{2i-2} - \kappa x_{2i-1} + \kappa x_{2i} - \iota x_{2i+1}\,. \end{aligned}$$

Our SIMD parallelization idea is similar to the one we used with the Haar filter: we compute 8 results within one loop iteration: 4 for the low-pass subband and 4 for the high-pass subband. In order to achieve this, we have to read 5 packed words of our source (20 values) and shuffle them in a way so we have a packed word for each x in the L_i-, H_i-formulas:

$$\begin{aligned} & s_1 \leftarrow x_{(8i-6,\ldots,8i-3)}, && s_2 \leftarrow x_{(8i-2,\ldots,8i+1)}, \\ & s_3 \leftarrow x_{(8i+2,\ldots,8i+5)}, && s_4 \leftarrow x_{(8i+6,\ldots,8i+9)}, \\ & s_5 \leftarrow x_{(8i+10,\ldots,8i+13)}, \end{aligned}$$

$$\begin{aligned} a_1 &\leftarrow (s_1, s_2)_{(0,2,4,6)}, & b_1 &\leftarrow (s_1, s_2)_{(1,3,5,7)}, \\ g_1 &\leftarrow (s_1, s_2, s_3)_{(2,4,6,8)}, & d_1 &\leftarrow (s_1, s_2, s_3)_{(3,5,7,9)}, \\ e_1 &\leftarrow (s_2, s_3)_{(0,2,4,6)}, & z_1 &\leftarrow (s_2, s_3)_{(1,3,5,7)}, \\ z_2 &\leftarrow (s_2, s_3, s_4)_{(2,4,6,8)}, & e_2 &\leftarrow (s_2, s_3, s_4)_{(3,5,7,9)}, \\ d_2 &\leftarrow (s_3, s_4)_{(0,2,4,6)}, & g_2 &\leftarrow (s_3, s_4)_{(1,3,5,7)}, \\ b_2 &\leftarrow (s_3, s_4, s_5)_{(2,4,6,8)}, & a_2 &\leftarrow (s_3, s_4, s_5)_{(3,5,7,9)}\,. \end{aligned}$$

Now we can replace each coefficient symbol σ with the packed word $(\sigma, \sigma, \sigma, \sigma)$ and the multiplications, additions and subtractions with their packed counterparts $\odot$, $\oplus$ and $\ominus$. The resulting formulas for the low-pass subband are:

$$\begin{aligned} L_i \leftarrow & \ominus(\alpha, \alpha, \alpha, \alpha) \odot (a_1 \oplus a_2) \oplus (\beta, \beta, \beta, \beta) \odot (b_1 \oplus b_2) \\ & \oplus(\gamma, \gamma, \gamma, \gamma) \odot (g_1 \oplus g_2) \ominus (\delta, \delta, \delta, \delta) \odot (d_1 \oplus d_2) \\ & \ominus(\epsilon, \epsilon, \epsilon, \epsilon) \odot (e_1 \oplus e_2) \oplus (\zeta, \zeta, \zeta, \zeta) \odot (z_1 \oplus z_2)\,. \end{aligned}$$

If we use the same idea for the high-pass subband, it can easily be seen that we need not compute new shuffles, but can reuse e_1, e_2, z_1 and z_2 from the low-pass subband:

$$H_i \leftarrow (\iota, \iota, \iota, \iota) \odot (e_2 \ominus e_1) \oplus (\kappa, \kappa, \kappa, \kappa) \odot (z_1 \ominus z_2)\,.$$

As further improvement, we can reuse the packed source words s_3, s_4 and s_5 as well as the shuffles d_2,g_2,b_2 and a_2 in the next iteration. So we can replace some of the assignments of auxiliary words with the following operations:

$$\begin{aligned} & s_1 \leftarrow s_3,\ s_2 \leftarrow s_4,\ s_3 \leftarrow s_5, \\ & a_1 \leftarrow d_2,\ b_1 \leftarrow g_2,\ g_1 \leftarrow b_2,\ d_1 \leftarrow a_2\,. \end{aligned}$$

In this way we pass reusable packed words from one iteration to the next. This saves 3 memory reads and 4 shuffle operations. Initially, $s_3, s_4, s_5, d_2, g_2, b_2$ and a_2 have to be set correctly of course. Note also that all five phases (see Section 2) are present and coefficient multiplication comes after data rearrangement.

With these optimizations, we need 14 packed additions and 8 packed multiplies for 8 destination values, while the simple sequential version takes 16 additions and 16 multiplies for 2 destination values. In the parallelized version, we further need 4 packed loads and stores and 12 shuffle operations for 8 destination values. On an AMD Mobile Duron Processor we get a speedup of 2.6 for the forward and 2.0 for the backward transform using the SSE extension.

4 Biorthogonal 7/9

The biorthogonal 7/9 filter is an example for an uneven, symmetrical filter. It has 9 lowpass coefficients $(a, b, c, d, e, d, c, b, a)$ and 7 high-pass coefficients (p, q, r, s, r, q, p). The sequential algorithm is:

$$\begin{aligned} L_i \leftarrow\ & ax_{2i-4} + bx_{2i-3} + cx_{2i-2} + dx_{2i-1} + ex_{2i} \\ & + dx_{2i+1} + cx_{2i+2} + bx_{2i+3} + ax_{2i+4}, \\ H_i \leftarrow\ & px_{2i-2} + qx_{2i-1} + rx_{2i} + sx_{2i+1} + rx_{2i+2} + qx_{2i+3} + px_{2i+4}\,. \end{aligned}$$

Our experiments show that the reuse of already computed values (e.g. $dx_{2i+1} = dx_{2j-1}$ for $j = i + 1$) does not produce any speedup in this case. Accordingly, we use a rather straight forward approach for the SIMD parallelization.

$$u \leftarrow x_{(8i+4,\ldots,8i+7)},\ v \leftarrow x_{(8i+8,\ldots,8i+11)},$$
$$u_0 \leftarrow u_{(0,0,0,0)},\ u_1 \leftarrow u_{(1,1,1,1)},\ \ldots,\ v_3 \leftarrow v_{(3,3,3,3)},$$

$$\begin{aligned} S \leftarrow\ & S \oplus u_0 \odot (a, c, e, c) \oplus u_1 \odot (0, b, d, d) \oplus u_2 \odot (0, a, c, e) \oplus u_3 \odot (0, 0, b, d) \\ & \oplus v_0 \odot (0, 0, a, c) \oplus v_1 \odot (0, 0, 0, b) \oplus v_2 \odot (0, 0, 0, a), \\ T \leftarrow\ & T \oplus u_0 \odot (p, r, r, p) \oplus u_1 \odot (0, q, s, s) \oplus u_2 \odot (0, p, r, r) \oplus u_3 \odot (0, 0, q, s) \\ & \oplus v_0 \odot (0, 0, p, r) \oplus v_1 \odot (0, 0, 0, q) \oplus v_2 \odot (0, 0, 0, p), \end{aligned}$$

$$L_{(4i,\ldots,4i+3)} \leftarrow S,\ H_{(4i,\ldots,4i+3)} \leftarrow T,$$

$$\begin{aligned} S \leftarrow\ & u_0 \odot (a, 0, 0, 0) \oplus u_1 \odot (b, 0, 0, 0) \oplus u_2 \odot (c, a, 0, 0) \oplus u_3 \odot (d, b, 0, 0) \oplus \\ & v_0 \odot (e, c, a, 0) \oplus v_1 \odot (d, d, b, 0) \oplus v_2 \odot (c, e, c, a) \oplus v_3 \odot (b, d, d, b), \\ T \leftarrow\ & u_2 \odot (p, 0, 0, 0) \oplus u_3 \odot (q, 0, 0, 0) \oplus \\ & v_0 \odot (r, p, 0, 0) \oplus v_1 \odot (s, q, 0, 0) \oplus v_2 \odot (r, r, p, 0) \oplus v_3 \odot (s, s, q, 0)\,. \end{aligned}$$

In the first line two packed words are read. They are rearranged immediately after that so that there is one word for each of the 8 elements, consisting entirely of copies of that element. These are then used in the coefficient multiplication phase, where the consecutive coefficients are spread across consecutive

multiplicator words. Note that, again, two packed words S, T are passed from one iteration to the other. Therefore, each memory location has to be accessed only once.

The sequential algorithm requires $7 + 9 = 16$ multiplies and 14 additions for 2 input values. The parallelized one requires 28 packed multiplies and 26 packed additions for 8 input values. This gives a theoretical speedup of only 2.3. One reason for this is that the parallelization is not optimal. Another reason is that because of alignment problems due to the uneven filter length, some multiplier words have to contain zeros, thus introducing useless computations. On an Intel Pentium 4 CPU with 2.8GHz we get a speedup of 1.75.

5 Biorthogonal 7/9 with Lifting

The biorthogonal 7/9 filter can also be implemented by applying the lifting scheme [11]. This method can reduce the number of multiplies by a factor of up to 2. For this algorithm we are also able to obtain some speedup using SIMD operations. The general algorithm is shown below.

$$\begin{aligned} &x_{2i+1} \leftarrow x_{2i+1} + a(x_{2i} + x_{2i+2}), && x_{2i} \leftarrow x_{2i} + b(x_{2i-1} + x_{2i+1}),\\ &x_{2i+1} \leftarrow x_{2i+1} + c(x_{2i} + x_{2i+2}), && x_{2i} \leftarrow x_{2i} + d(x_{2i-1} + x_{2i+1}),\\ &x_{2i+1} \leftarrow -e x_{2i+1}, && x_{2i} \leftarrow \tfrac{1}{e} x_{2i} \end{aligned}$$

Note that in contrast to all other algorithms in this paper, each of these assignments has to be executed for all i before proceeding with the next assignment. To transform this algorithm into a single outer loop, we have to rewrite it applying passing of values between iterations again. This leads to the following algorithm:

$$\begin{aligned} &o \leftarrow q, && p \leftarrow x_{2i+3},\ q \leftarrow x_{2i+4},\\ &r \leftarrow s, && s \leftarrow p + a(o + q),\\ &t \leftarrow u, && u \leftarrow o + b(r + s),\\ &v \leftarrow w, && w \leftarrow r + c(t + u),\\ &L_i \leftarrow t + d(v + w) \cdot \tfrac{1}{e}, && H_i \leftarrow w \cdot (-e)\,. \end{aligned}$$

To be able to use SIMD operations, we need full packed words again. Using the same notation as in the above examples we can write the SIMD parallelization

as

$$
\begin{array}{lll}
h \leftarrow x_2, & x_1 \leftarrow x_{(8i+4,\ldots,8i+7)}, & x_2 \leftarrow x_{(8i+8,\ldots,8i+11)}, \\
q \leftarrow (h, x_1)_{(0,2,4,6)}, & p \leftarrow (h, x_1, x_2)_{(3,5,7,9)}, & o \leftarrow (h, x_1)_{(2,4,6,8)}, \\
r \leftarrow s, & s \leftarrow (a,a,a,a) \odot (o \oplus q) \oplus p, & r \leftarrow (r, s)_{(3,5,6,7)}, \\
t \leftarrow u, & u \leftarrow (b,b,b,b) \odot (r \oplus s) \oplus o, & t \leftarrow (t, u)_{(3,5,6,7)}, \\
v \leftarrow w, & w \leftarrow (c,c,c,c) \odot (t \oplus u) \oplus r, & v \leftarrow (v, w)_{(3,5,6,7)}, \\
\multicolumn{3}{c}{L_{(4i,\ldots,4i+3)} \leftarrow ((d,d,d,d) \odot (v \oplus w) \oplus t) \odot (\tfrac{1}{e}, \tfrac{1}{e}, \tfrac{1}{e}, \tfrac{1}{e}),} \\
\multicolumn{3}{c}{H_{(4i,\ldots,4i+3)} \leftarrow (-e,-e,-e,-e) \odot w\,.}
\end{array}
$$

It is not possible to implement this algorithm straight forward, because SIMD extensions (e.g. Intel SSE2 instruction set) do not support shuffling in the way used here. Therefore the sample implementation uses some auxiliary variables to work around and this of course decreases performance.

As with the Haar filter the theoretical possible speedup of SIMD implementation is 4. However, due to complex shuffling operations the measured value is much lower. Using SIMD operations in combination with lifting algorithms should theoretically speed up the system by a factor of 6.5 compared to a sequential implementation of standard Biorthogonal (7/9) filter. This speedup is composed of 64% from the algorithm [11] and 400% from the SIMD operations.

Of course, speedup depends on the implementation of SIMD features on the processor. Measurements gave a speedup of 2.65 for forward and 1.7 for backward transform compared to plain implementation of the lifting algorithm. Measurements have been taken on an Intel Pentium 4 with 2.8GHz using the SSE extension with packed words of 4 single precision numbers.

6 Conclusion

We have shown that the 1-D wavelet filter operation is indeed parallelizable with SIMD extensions of common general purpose processors. Speedups in the range 1.5 up to 3 can be achieved for packed words of 4 single precision floating point numbers.

The efficiency of the parallelization depends largely on the filter lengths, the alignments and even on the coefficients of the filters. Uneven filter lengths force some words in the SIMD computations to be zero, introducing useless computations and thus limiting the speedup. If some of the coefficients are equal, as there are for symmetrical filters, the sequential algorithm can be optimized by reusing already computed values. To do the same in the SIMD parallelized algorithm often implies complicated shuffle operations.

Generally, the need for many shuffle operations probably reduces the speedup most. Another source of speedup limitation is memory access as bottleneck.

Together with cache issues and data organization problems for 2-D transforms it can influence the speedup either in a negative or positive way. Investigating these relations is an issue of future work.

Strikingly, backward transforms always produce lower speedups than forward transforms. The actual reasons for this are not quite clear. However, the following differences between forward and backward transforms are probably responsible. Since backward filters are basically equal to forward filters mirrored at position 1, backward filters are usually aligned worse, i.e. at positions that are not divisible by 4. Another difference is that in the forward case two consecutive words are read from memory and two non-consecutive are written, while in the backward case the situation is reverse. This may lead to different cache effects.

Apart from speedup issues, algorithms have to be found to derive optimal solutions. This is important because each parallelization presented in this work is one of many possible solutions and it is not at all clear that the shown solutions could not be improved. Since it would be an almost unaccomplishable amount of work to hand-code a variety of solutions to find the best, automatic optimization techniques are required.

References

[1] ISO/IEC JPEG committee. JPEG 2000 image coding system — ISO/IEC 15444-1:2000, December 2000.

[2] R. Kutil and A. Uhl. Optimization of 3-d wavelet decomposition on multiprocessors. *Journal of Computing and Information Technology (Special Issue on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia)*, 8(1):31–40, 2000.

[3] M-L. Woo. Parallel discrete wavelet transform on the Paragon MIMD machine. In R.S. Schreiber et al., editors, *Proceedings of the seventh SIAM conference on parallel processing for scientific computing*, pages 3–8, 1995.

[4] J. Fridman and E.S. Manolakos. On the scalability of 2D discrete wavelet transform algorithms. *Multidimensional Systems and Signal Processing*, 8(1–2):185–217, 1997.

[5] M.M. Pic, H. Essafi, and D. Juvin. Wavelet transform on parallel SIMD architectures. In F.O. Huck and R.D. Juday, editors, *Visual Information Processing II*, volume 1961 of *SPIE Proceedings*, pages 316–323. SPIE, August 1993.

[6] C. Chakrabarti and M. Vishvanath. Efficient realizations of the discrete and continous wavelet transforms: From single chip implementations to mappings on SIMD array computers. *IEEE Transactions on Signal Processing*, 3(43):759–771, 1995.

[7] M. Feil and A. Uhl. Wavelet packet decomposition and best basis selection on massively parallel SIMD arrays. In *Proceedings of the International Conference "Wavelets and Multiscale Methods" (IWC'98), Tangier, 1998.* INRIA, Rocquencourt, April 1998. 4 pages.

[8] C. Tenllado, D. Chaver, L. Piñuel, M. Prieto, and F. Tirado. Vectorization of the 2D wavelet lifting transform using SIMD extensions. In *Workshop on Parallel and Distributed Image Processing, Video Processing, and Multimedia, PDIVM '03*, Nice, France, April 2003.

[9] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2-D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and SIMD parallelism exploitation. In *Proceedings of the 2000 International Conference on High Performance Computing*, Bangalore, India, December 2002.

[10] C. Chrysafis and A. Ortega. Line based, reduced memory, wavelet image compression. *IEEE Transactions on Image Processing*, 9(3):378–389, March 2000.

[11] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis Applications*, 4(3):245–267, 1998.