Parallel Processing Letters
© World Scientific Publishing Company

PARALLELIZATION OF WAVELET FILTERS USING SIMD EXTENSIONS

RADE KUTIL and PETER EDER
Department of Computer Sciences, University of Salzburg
Jakob Haringerstr. 2, 5020 Salzburg, Austria

Received September 2005
Revised May 2006
Communicated by Roman Trobec

ABSTRACT

Much work has been done to optimize wavelet transforms for SIMD extensions of modern CPUs. However, these approaches are mostly 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 an example of a common wavelet filter new approaches to use SIMD operations on 1-D transforms that are able to produce reasonable speedups. As a result, the performance of algorithms that use wavelet transforms, such as JPEG2000, can be increased significantly. Various variants of parallelization are presented and compared. Their advantages and disadvantages for general filters are discussed.

Keywords: SIMD, short vector, signal processing, wavelets, multimedia

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 fast wavelet transform 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.

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. 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 is investigated for the 2-D case in [8,9].

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 neighbouring 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 that 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 treatment that changes from word to word 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.

Apart from the direct FIR filter implementation, there is also the lifting scheme [11] which is based on the factorization of the filter pair. It can reduce the number of multiplies and adds and yields theoretical speedups of up to 2. The lifting scheme is also considered in this work and compared to the standard implementation. Often, implementations of the lifting scheme produce interleaved low-pass and high-pass subbands. In this work, separated subbands are used for all filter algorithms for two reasons. First, the presented filter algorithms can be applied recursively on the low-pass subbands without modification to give a full wavelet transform. Second, most applications, such as JPEG2000, prefer separated subbands.

This work presents new approaches for an example of a common filter, i.e. the biorthogonal 7/9 filter, and shows that reasonable speedups can be achieved. All results in this work have been conducted on an Intel Pentium 4 CPU with 3.2GHz and 2MB cache size using the SSE extension with packed words of 4 single precision numbers. All implementations use the same amount of code optimization, i.e. memory access through incremented pointers instead of indexed arrays, and compilation with gcc 3.3.5 with the -O3 option. SIMD operations are implemented using gcc's built-in functions for vector extensions and the -msse option. Note that in order to have full control over generated code, no automatic vectorization is applied.

The approaches proposed in this work are not compared to automatically vectorized code since it has already been shown in [8,9] that automatic vectorization is still not able to produce a performance increase for this problem. However, they are compared to hand-optimized code by human experts, i.e. the Intel® Integrated Performance Primitives (IPP) v5.0, and are able to compete with and even outperform it depending on the algorithm class and data size. Note that the IPP library also uses SIMD operations, but the applied methods are not known to the authors.

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, while following sections will concentrate on the biorthogonal 7/9 filter. 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 define the filtering process of the Haar wavelet transform.

$$\text{for all } i : L_i \leftarrow ax_{2i} + ax_{2i+1},\ H_i \leftarrow ax_{2i} - ax_{2i+1}$$

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

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

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 $y_{(i_0,\dots,i_m)} := (y_{i_0}, \dots, y_{i_m})$. Thus, we can write the SIMD parallelization of the Haar filter for word size 4 as

$$\begin{array}{l}\text{for all } i : \\ \quad \begin{array}{ll} p \leftarrow x_{(8i,\dots,8i+3)} \odot (a,a,a,a), & q \leftarrow x_{(8i+4,\dots,8i+7)} \odot (a,a,a,a), \\ r \leftarrow (p,q)_{(0,2,4,6)}, & s \leftarrow (p,q)_{(1,3,5,7)}, \\ L_{(4i,\dots,4i+3)} \leftarrow r \oplus s, & H_{(4i,\dots,4i+3)} \leftarrow r \ominus s\ . \end{array}\end{array}$$

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 gives a theoretical speedup of 4. However, since the shuffle operations also require some execution time and memory access can be a bottleneck, the speedup is reduced and we get an actual speedup of 2.7.

In the following sections we will discuss the more complicated example of the biorthogonal 7/9-tap filter which is used in many multimedia applications such as the JPEG2000 standard [1]. Note that all algorithms 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 7/9 without Lifting

3.1. Sequential Algorithm

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

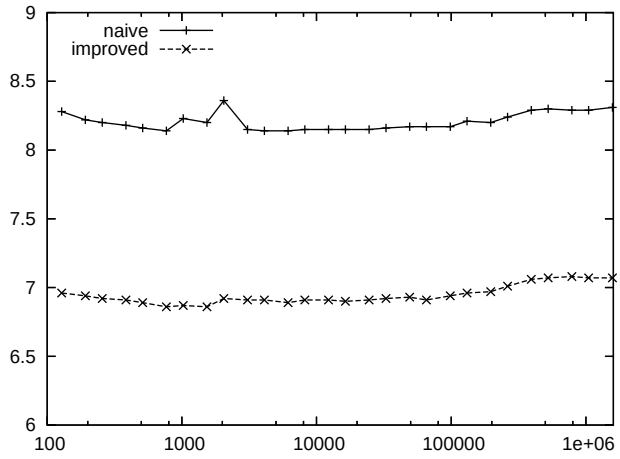

Fig. 1. Execution time of naive and improved sequential algorithm in ns/sample. The horizontal axis shows the size of the repeatedly transformed data set in number of single precision values.

The sequential algorithm is

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

However, this algorithm can be optimized in terms of number of required multiplication operations due to the symmetry of the filters. Samples that have to be multiplied by the same coefficient and added afterwards can be added before multiplication instead, saving one multiply.

$$\begin{aligned}&\text{for all } i:\\ &\quad L_i \leftarrow a(x_{2i-4} + x_{2i+4}) + b(x_{2i-3} + x_{2i+3}) + c(x_{2i-2} + cx_{2i+2})\\ &\qquad\quad + d(x_{2i-1} + x_{2i+1}) + ex_{2i}\,,\\ &\quad H_i \leftarrow p(x_{2i-2} + x_{2i+4}) + q(x_{2i-1} + x_{2i+3}) + r(x_{2i} + x_{2i+2}) + sx_{2i+1}\,.\end{aligned}$$

Thus, 14 adds and only 9 multiplies (instead of 16) are required in each iteration. To see the gain in performance of the optimized sequential algorithm, look at Fig. 1. This plot shows the execution times in ns/sample over the size of transformed data. The algorithm has been performed several times on the same data in order to unveil the influence of cache on the execution time. However, the fact that execution times per sample do not vary significantly with the data size shows that accessing cached data has little impact on the performance. This shows that memory access is not a bottleneck and the speedups shown in this and the following sections represent algorithmic improvements. The improved algorithm gains a sequential speedup of 1.18. All parallel speedups in this section will be measured against the improved algorithm.

3.2. SIMD Parallelization – Variant 1

There are many possibilities to parallelize the above algorithm. The main difference between these variants is when to apply the phase of shuffle operations – before or after multiplying with filter coefficients. The first variant performs this multiplication directly after source data is read from memory.

As with the Haar filter, two packed words have to be read to calculate one new low-pass word and one new high-pass word. However, since the filter is now longer than two taps, the contents of more than two packed words are actually needed. This can be overcome by reusing intermediate results from previous iterations, which amounts to passing values from iteration to iteration.

In this first variant, the values of each of the two recently read words are immediately multiplied by all necessary filter coefficients. Then appropriate shuffles of the products have to be added, leading to the following algorithm:

for all i :

$$
\begin{aligned}
&Y \leftarrow x_{(8i+4,\ldots,8i+7)}, Z \leftarrow x_{(8i+8,\ldots,8i+11)}\\
&A \leftarrow C, \quad B \leftarrow D, \quad C \leftarrow Y \odot (a,b,a,b), \quad D \leftarrow Z \odot (a,b,a,b),\\
&E \leftarrow G, \quad F \leftarrow I, \quad G \leftarrow Y \odot (c,d,c,d), \quad I \leftarrow Z \odot (c,d,c,d),\\
&J \leftarrow M, \quad K \leftarrow N, \quad M \leftarrow Y \odot (e,0,e,0), \quad N \leftarrow Z \odot (e,0,e,0),\\
&L_{(4i,\ldots,4i+3)} \leftarrow (A,B)_{(0,2,4,6)} \oplus (A,B)_{(1,3,5,7)} \oplus (E,F,G)_{(2,4,6,8)} \oplus\\
&\quad (E,F,G)_{(3,5,7,9)} \oplus (K,M)_{(0,2,4,6)} \oplus (F,G)_{(1,3,5,7)} \oplus (F,G,I)_{(2,4,6,8)} \oplus\\
&\quad (B,C,D)_{(3,5,7,9)} \oplus (C,D)_{(0,2,4,6)},\\
&P \leftarrow R, \quad Q \leftarrow S, \quad R \leftarrow Y \odot (p,q,p,q), \quad S \leftarrow Z \odot (p,q,p,q),\\
&T \leftarrow V, \quad U \leftarrow W, \quad V \leftarrow Y \odot (r,s,r,s), \quad W \leftarrow Z \odot (r,s,r,s),\\
&H_{(4i,\ldots,4i+3)} \leftarrow (P,Q,R)_{(2,4,6,8)} \oplus (P,Q,R)_{(3,5,7,9)} \oplus (U,V)_{(0,2,4,6)} \oplus\\
&\quad (U,V)_{(1,3,5,7)} \oplus (U,V,W)_{(2,4,6,8)} \oplus (Q,R,S)_{(3,5,7,9)} \oplus (R,S)_{(0,2,4,6)}
\end{aligned}
$$

Fig. 2 depicts the algorithm as a data-flow diagram. After multiplying the two new source words by words of appropriate filter coefficients, they are rearranged by shuffle operations (thin arrows) so that the sum of the resulting words is the desired destination word containing four low-pass filtered samples. Note that the intermediate words (after multiplication) are passed from the previous iteration (dashed arrows). In this way one can avoid half of the multiplication operations.

Only the low-pass calculations are shown. The operations for high-pass filtering are similar. A big disadvantage of this variant is that no intermediate results can be shared between the low- and high-pass part. Moreover, many shuffle operations have to be composed by two or more shuffles. One reason for this is that some such operations require three source words. Another reason is that the processor's instruction set does not allow arbitrary shuffles, i.e. not all possible maps from two source words to one destination word can be done in one instruction. Altogether this algorithm can be implemented by 10 multiplies, 14 adds, and 26 shuffles.

3.3. SIMD Parallelization – Variant 2

A major disadvantage of the first variant is that values that have to be collected in a single word are spread over several intermediate words, requiring more shuffle operations. The reason for this is that downsampling causes every second value

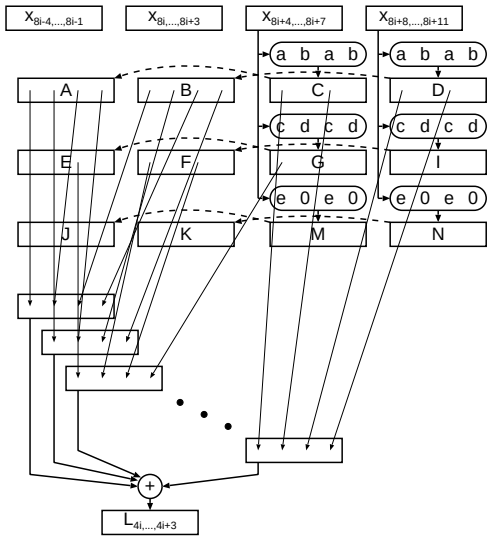

Fig. 2. Variant 1 of SIMD-parallel algorithm. Packed words are indicated by boxes, multiplication by boxes with rounded edges, addition by a circle with a +, shuffle operations by thin arrows, and the passing of values between iterations by dashed arrows. Only the low-pass calculations are shown, high-pass operations are similar.

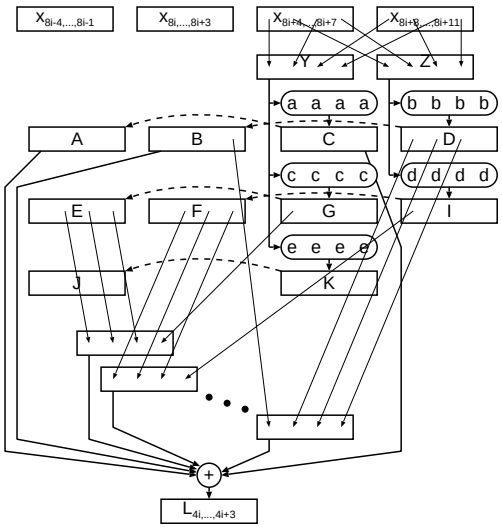

Fig. 3. Variant 2 of SIMD-parallel algorithm.

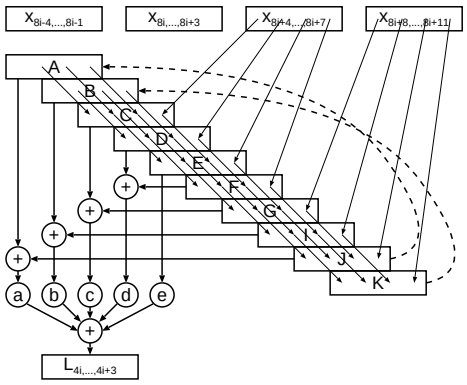

Fig. 4. Variant 3 of SIMD-parallel algorithm. Multiplication by a packed word of equal coefficients is depicted by a single circle.

to belong together. Therefore, the second variant inserts a single step of shuffling before the multiplication, putting even and odd samples into separate words. This leads to the following algorithm, which is also shown in Fig. 3.

for all i :

$$
\begin{aligned}
&Y \leftarrow x_{(8i+4,8i+6,8i+8,8i+10)}, Z \leftarrow x_{(8i+5,8i+7,8i+9,8i+11)}\\
&A \leftarrow C, \quad B \leftarrow D, \quad C \leftarrow Y \odot (a,a,a,a), \quad D \leftarrow Z \odot (b,b,b,b),\\
&E \leftarrow G, \quad F \leftarrow I, \quad G \leftarrow Y \odot (c,c,c,c), \quad I \leftarrow Z \odot (d,d,d,d),\\
&J \leftarrow K, \qquad\qquad K \leftarrow Y \odot (e,e,e,e),\\
&L_{(4i,\ldots,4i+3)} \leftarrow A \oplus B \oplus (E,G)_{(1,2,3,4)} \oplus (F,I)_{(1,2,3,4)} \oplus (J,K)_{(2,3,4,5)} \oplus\\
&\quad (F,I)_{(2,3,4,5)} \oplus (E,G)_{(3,4,5,6)} \oplus (B,D)_{(3,4,5,6)} \oplus C\\
&P \leftarrow R, \quad Q \leftarrow S, \quad R \leftarrow Y \odot (p,p,p,p), \quad S \leftarrow Z \odot (q,q,q,q),\\
&T \leftarrow V, \quad U \leftarrow W, \quad V \leftarrow Y \odot (r,r,r,r), \quad W \leftarrow Z \odot (s,s,s,s),\\
&H_{(4i,\ldots,4i+3)} \leftarrow (P,R)_{(1,2,3,4)} \oplus (Q,S)_{(1,2,3,4)} \oplus (T,V)_{(2,3,4,5)} \oplus\\
&\quad (U,W)_{(2,3,4,5)} \oplus (T,V)_{(3,4,5,6)} \oplus (Q,S)_{(3,4,5,6)} \oplus R
\end{aligned}
$$

This has two advantages. First, there is one less multiplication for the e-coefficient. Second, no shuffle requires more than two source words. Moreover, the two results of the first shuffling step can be reused in the high-pass part. Thus, this algorithm is implemented by only 9 multiplies, 14 adds, and 20 shuffles.

3.4. SIMD Parallelization – Variant 3

The third variant adopts the scheme of the improved sequential algorithm. First, the input words are shuffled so that the remaining operations can be performed as in the sequential case. This reverses the order of phases completely. Then, words that have to be multiplied by the same filter coefficients are added, followed by

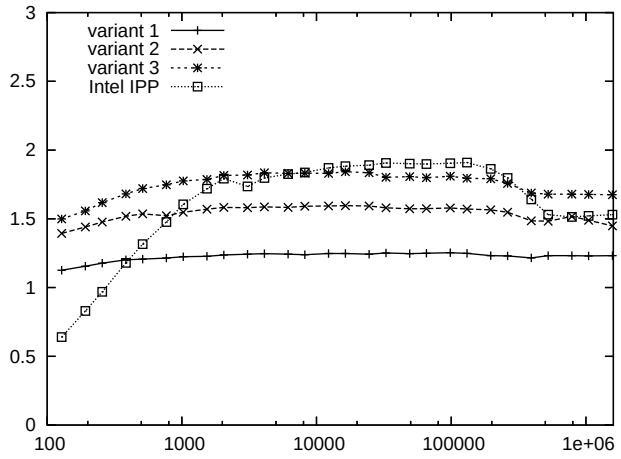

Fig. 5. Speedups of the SIMD parallelization variants against the improved sequential algorithm. The horizontal axis again shows the size of the repeatedly transformed data set.

multiplication and the final sum. The following algorithm is also shown in Fig. 4.

for all i :

$$
\begin{aligned}
&Y \leftarrow x_{(8i+4,\ldots,8i+7)}, \quad Z \leftarrow x_{(8i+8,\ldots,8i+11)},\\
&A \leftarrow J,\ B \leftarrow K, \qquad C \leftarrow (A,Y)_{(1,2,3,4)}, \qquad D \leftarrow (B,Y)_{(1,2,3,5)},\\
&E \leftarrow (C,Y)_{(1,2,3,6)}, \quad F \leftarrow (D,Y)_{(1,2,3,7)}, \quad G \leftarrow (E,Z)_{(1,2,3,4)},\\
&I \leftarrow (F,Z)_{(1,2,3,5)}, \quad J \leftarrow (G,Z)_{(1,2,3,6)}, \quad K \leftarrow (I,Z)_{(1,2,3,7)},\\
&L_{(4i,\ldots,4i+3)} \leftarrow (A \oplus J) \odot (a,a,a,a) \oplus (B \oplus I) \odot (b,b,b,b) \oplus\\
&\quad (C \oplus G) \odot (c,c,c,c) \oplus (D \oplus F) \odot (d,d,d,d) \oplus E \odot (e,e,e,e)\\
&H_{(4i,\ldots,4i+3)} \leftarrow (C \oplus J) \odot (p,p,p,p) \oplus (D \oplus I) \odot (q,q,q,q) \oplus\\
&\quad (E \oplus G) \odot (r,r,r,r) \oplus F \odot (s,s,s,s)
\end{aligned}
$$

Note that only two words have to be passed to the next iteration. This reduces the demand for register allocation significantly. The biggest advantage of this algorithm is that all results of the shuffle phase can be reused in the high-pass part. Unfortunately, none of the shuffles, as depicted in Fig. 4, can be implemented as a single instruction. However, through appropriate rearrangements some of the additional instructions can be avoided. Altogether, this variant requires 9 multiplies, 14 adds, and 12 shuffles.

3.5. Experimental Results

As variants 2 and 3 of the SIMD algorithms have the same number of multiplies and adds as the improved sequential algorithm, only with packed words instead of single numbers, there is a potential speedup of 4. However, due to massive shuffle operations this speedup cannot be reached, as one can see in Fig. 5. According to expectations, variant 3 is the best, giving speedups of 1.8.

Again, accessing cached data has only a minor influence on performance. The decay of speedup for small data sizes is due to complex startup and close-off operations, e.g. for initializing registers, which become more dominant for small data sizes. The slight decay for large data sizes is probably due to cache effects.

The hand-optimized Intel® IPP library has slightly better speedups for medium data sizes. However, it seems to be more dependent on cache since its performance decreases noticeably for large data sizes. Also, it seems to have even more problems with startup operations for small data sizes, although filter allocation is performed only once for all repeated calls in the experiment. Note that `ippsWTFwd_32f` is used here which does not apply lifting and where filters are not fixed, i.e. defined at runtime.

3.6. Arbitrary Filters

The approaches presented here can all be applied to other filters as well. It is not apparent, however, which one would be the best for a given filter, or if some modification of a variant can do even better. Let us, therefore, look at how the features of the presented variants behave on other kinds of filters.

Variants 1 and 2 rely on the fact that a single filter coefficient has to be applied to either even or odd samples, but not both. However, this is only true for uneven symmetrical filters, or filter without any symmetry. This means that variant 3 has even more advantages for even symmetrical filters. On the other hand, variant 3 might imply redundant multiplications for non-symmetrical filters if some low- and high-pass coefficients are equal. This happens mostly for orthogonal wavelets. In this case, however, filters have even length and, as a consequence, a low-pass coefficient for even samples always corresponds to an equal high-pass coefficient for uneven samples, or vice versa. Therefore, variant 3 does not produce redundant multiplications for orthogonal wavelets, since multiplied even samples can never be reused for the high-pass filtering.

Important questions arise for particularly long filters. Variants 2 and 3 need to store at least one word for each filter tap to pass it to the next iteration. This requires the allocation of many CPU registers and leads to additional memory access when the compiler runs out of available registers. On the other hand, variant 3 has to keep all shuffled words in registers, whereas variants 1 and 2 can drop shuffled words (and even some other intermediate words) after having added them to the final sum. However, variant 3 can also drop these if the filter is non-symmetrical.

All these remarks are only hints, of course. Filters reveal surprisingly diverse features with respect to SIMD parallelization. Each particular filter should be examined thoroughly, based on the approaches presented in this work.

4. Biorthogonal 7/9 with Lifting

4.1. Sequential Algorithm

As most wavelet filters, the biorthogonal 7/9 filter can also be implemented by applying the lifting scheme [11]. It factors the filter pair into several predict and update steps, where odd values (values at odd position) are predicted from

even values and replaced by the difference between prediction and actual value, and even values are updated to represent a local average. This method significantly reduces the number of multiplies in the sequential algorithm. In this specific case the sequential biorthogonal 7/9 without lifting uses 9 multiplies for every two samples (improved version), whereas biorthogonal 7/9 with lifting as shown here only requires 6 multiplies.

$$\begin{array}{ll} \text{for all } i: x_{2i+1} \leftarrow x_{2i+1} + a(x_{2i} + x_{2i+2}), & \text{for all } i: x_{2i} \leftarrow x_{2i} + b(x_{2i-1} + x_{2i+1}), \\ \text{for all } i: x_{2i+1} \leftarrow x_{2i+1} + c(x_{2i} + x_{2i+2}), & \text{for all } i: x_{2i} \leftarrow x_{2i} + d(x_{2i-1} + x_{2i+1}), \\ \text{for all } i: x_{2i+1} \leftarrow -e x_{2i+1}, & \text{for all } i: x_{2i} \leftarrow \frac{1}{e} x_{2i} \end{array}$$

The low-pass and high-pass subbands are then found interleaved in even and odd positions, respectively. Note that the coefficients $a, \ldots, e$ are not the same as in the sequential algorithm, but are the result of the factorization process on which the lifting scheme is based. Note also that each of these assignments has to be executed for all i before proceeding with the next assignment.

However, the lifting scheme can also be implemented in a single-loop manner in the sense that each input value is read from memory only once and each output value is written to memory once without subsequent updates. While this is an improvement in itself, since it minimizes memory access, it turns out to be the only reasonable way to go for the SIMD parallelization. To see why, let us examine the number of operations in a single lifting pass $x_{2n} \leftarrow x_{2n} + \alpha(x_{2n-1} + x_{2n+1})$. There are 2 adds and 1 multiply for every second sample, which makes 1 add and $\frac{1}{2}$ multiply per sample. Now, assume that SIMD operates on packed words of 4 samples. We can implement this operation by

$$x_{(2n,\ldots,2n+3)} \leftarrow x_{(2n,\ldots,2n+3)} + (\alpha, 0, \alpha, 0) \odot (x_{(2n-1,\ldots,2n+2)} + x_{(2n+1,\ldots,2n+4)}) .$$

Since $x_{(2n-1,\ldots,2n+2)}$ and $x_{(2n+1,\ldots,2n+4)}$ require shuffle operations, we need 2 shuffles, 2 adds and 1 multiply for every 4 samples, giving $\frac{1}{2}$ shuffle, $\frac{1}{2}$ add and $\frac{1}{4}$ multiply per sample or - taken together - 1.25 operations instead of 1.5 in the non-SIMD case. This is, obviously, not a satisfying speedup, given the theoretical maximum speedup of 4.

Therefore, we develop a new algorithm with a single outer loop. To do so, we have to rewrite it by applying the well known loop fusion technique. Immediately after iteration (i, j) of loop i, iteration $(i + 1, k)$ of the subsequent loop $i + 1$ is executed that depends on iteration (i, j) and does not depend on an iteration (i, l) in loop i occurring later in that loop ($l > j$). The process begins with the first loop. After one iteration of each loop has been executed, one iteration of the fused loop is completed and the process starts over with a subsequent iteration. As iteration (i, j) also depends on iteration $(i, j-1)$, values have to be passed between iterations. For every two input values, two output values can be calculated, one low-pass and

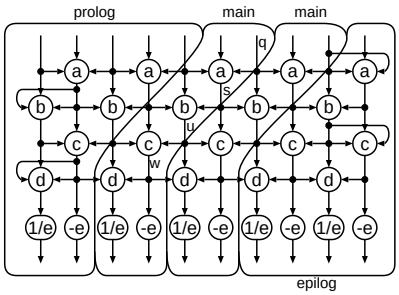

Fig. 6. Sequential single-loop algorithm for the biorthogonal 7/9 filter with lifting. Circles with three inputs (l left, r right, u upper) denote basic lifting operations $y = u + \alpha(l + r)$. Rounded frames indicate single iterations.

one high-pass coefficient. This leads to the following algorithm:

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

This algorithm is also shown in Fig. 6 for a very short data length of 10. Iteration, as described above, are denoted "main". Longer data would, of course, require more "main" iterations. Note that intermediate values q, s, u, w are passed from iteration to iteration, indicated by arrows that cross iteration borders in Fig. 6. These four values have to be set properly at the beginning of the loop. Also, the end of the loop needs special treatment. Fig. 6 shows how this must be done in the case of mirroring border handling in the phases denoted by "prolog" and "epilog".

4.2. SIMD Parallel Algorithm

To be able to obtain speedup using SIMD operations, again full packed words have to be loaded. Like in variant 2 of the biorthogonal filter without lifting data is shuffled after being read from memory. Then SIMD operations are applied. This leads to intermediate results which have to be shuffled again before proceeding. These results can be reused in the next iteration step, much like in the sequential

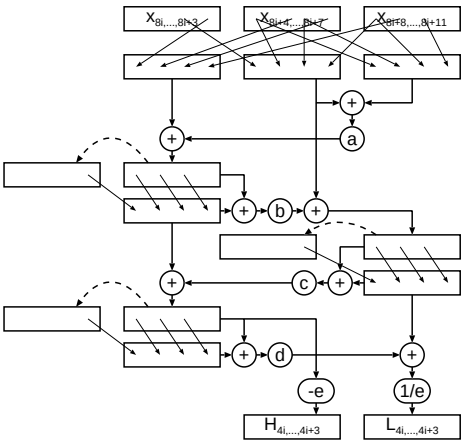

Fig. 7. SIMD-algorithm of biorthogonal 7/9 filter with lifting. Heavy use of shuffle-operations may cause non-optimal speedups. Like in the sequential case, intermediate values are passed between iterations (dashed lines).

algorithm, which leads to the following algorithm:

for all i :

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

See also Fig. 7 for a data-flow diagram of the algorithm. The algorithm can also be interpreted as being equivalent to variant 3 of the non-lifting algorithm, applied to each of the four stages for coefficients a, b, c, d. To see this, consider each stage as the application of the short filters $(a, 1, a), \ldots, (d, 1, d)$. Then each stage consists of the steps shuffle, add, multiply, and sum, just like variant 3 in Section 3.4. Variants 1 and 2 could also be used here. However, considerations show that these would immediately imply unreasonable slow-downs. For other filters given in lifting scheme, a similar approach can be applied, interpreting the lifting steps as short filters.

Again, it is not possible to implement the algorithm straight forward because SIMD extensions (e.g. Intel SSE2 instruction set) do not support shuffling from three sources into a single destination in a single instruction. However, the algorithm can be implemented with 6 multiplies, 8 adds, and 11 shuffles.

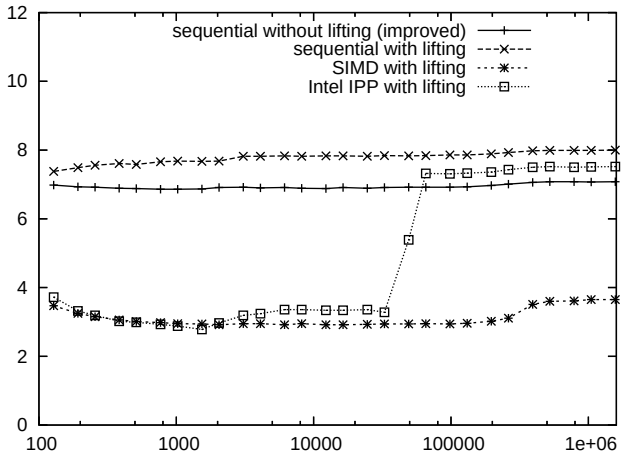

Fig. 8. Execution times in ns/sample of sequential and SIMD implementations with and without lifting over the size of the repeatedly transformed data set (number of floats).

4.3. Experimental Results

Fig. 8 shows execution times of the sequential and SIMD implementations of the lifting algorithm in comparison to the non-lifting algorithm. Interestingly, the sequential implementation is slower with lifting than without, despite the reduced number of multiplies and adds. Theoretical considerations [11] would imply a speedup of 1.64. An investigation of the assembler code showed no obvious reason, the faster code being significantly longer. A guess is that there is a peculiar problem in scheduling the instructions optimally which can be resolved more easily in the longer code.

However, the SIMD implementation is able to reduce the execution times significantly. Again, cached values do not seem to play an important role. Fig. 9 shows the speedup of the SIMD implementation compared to versions without lifting or SIMD. While compared to the sequential lifting algorithm we get a speedup of up to 2.66 (of a theoretical maximum of 4), the speedup is only 2.36 (of theoretical $1.64 \cdot 4 = 6.56$) compared to the sequential algorithm without lifting since the latter is faster, as mentioned above. However, the SIMD algorithm with lifting is faster than that without lifting. There is a speedup of about 1.3 (of theoretical 1.64). The speedup decay for large data sizes is again probably due to cache problems.

Again, the Intel® IPP library is not able to outperform our SIMD implementation of wavelet lifting, as can be seen in Fig. 8. It shows equal performance for small and slightly worse for medium data sizes. For large data sizes there seems to be a major cache problem, since its performance even drops below that of the sequential non-lifting algorithm. Note that `ippiWTFwdRow_D97_JPEG2K_32f_C1R` is used where lifting is applied and the filter is fixed, as in our implementation.

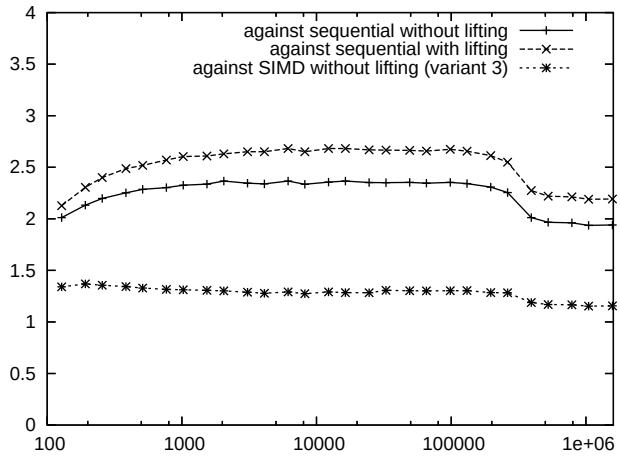

Fig. 9. Speedup of the SIMD implementation with lifting against implementations without lifting or SIMD.

5. Conclusion

We have shown that the 1-D wavelet filter operation is indeed parallelizable with SIMD extensions of common general purpose processors without hand-optimizing assembly code. For the example of the very common biorthogonal 7/9 filter, speedups in the range from 2 to 2.66 can be achieved for packed words of 4 single precision floating point numbers.

The efficiency of the parallelization depends largely on the filter lengths, their alignments and even on the coefficients of the filters. If some of the coefficients are equal, as there are for symmetrical filters, the sequential algorithm can be optimized by reusing computed values. To do the same in the SIMD parallelized algorithm often implies complicated shuffle operations.

Generally, the need for many shuffle operations reduces the speedup most. Memory access as a bottleneck could also limit speedups. However, investigations show that the execution times are almost invariant to whether source data is in cache or not. This means that the speedups shown in this paper represent purely algorithmic improvements.

Compared to the hand-optimized Intel® IPP library, our approaches show at least comparable but mostly better performance, especially for small and large data sizes due to lower initialization complexity and less cache dependence, respectively.

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 in practice it would be an almost unaccomplishable amount of work to hand-code a variety of solutions to find the best, automatic optimization techniques as in [12] 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 continuous 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.

[12] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.