Chapter 13
Short-Vector SIMD Parallelization in Signal Processing

Rade Kutil

Abstract

Short-vector Single-instruction-multiple-data (SIMD) units have become common in signal processors. Moreover, almost all modern general-purpose processors include SIMD extensions, which makes SIMD also important in high performance computing. This chapter gives an overview of approaches to the vectorization of signal processing algorithms. Despite their complexity, these algorithms have a relatively regular data flow. This regularity makes them good candidates for SIMD vectorization. They fall in two categories: filter banks that operate on streaming signal data, and Fourier-like transforms that operate on blocks of data. For the first category, simple FIR filters, IIR filters and more complicated filter banks from the field of wavelet transforms are investigated to develop and present general vectorization strategies. Well-known loop transformations as well as novel vectorization approaches are combined and evaluated. For the second category, basic approaches for the fast Fourier transform (FFT) are shown and the workings of automatic vectorizing performance tuning systems are explained. The presented solutions are tested on Intel processors with SIMD extensions and the results are compared. Wherever possible, the reasons for performance gains or losses are uncovered so that good vectorization strategies can be derived for arbitrary signal processing algorithms.

13.1 Introduction

The trend in parallelization goes toward multi-level parallelism. In addition to the combination of clusters, shared-memory architectures, and multi-core processors, CPU cores exploit more and more internal parallelity. Among methods such as ex-

Rade Kutil
Department of Computer Sciences, University of Salzburg, J.-Haringer-Strasse 2,
5020 Salzburg, Austria, e-mail: rkutil@cosy.sbg.ac.at

R. Trobec et al. (eds.), *Parallel Computing*, DOI 10.1007/978-1-84882-409-6_13,

cessive pipelining, specialized units, as used in signal processors, and VLIW (very large instruction word), SIMD (single instruction multiple data) plays an important role. One reason for its popularity is the availability of short-vector SIMD extensions in all modern general-purpose processors.

These processors are very cost-effective and, thus, heavily used in high performance computing (HPC). As a consequence, their SIMD extensions are exploited in most HPC software. SIMD always benefits from regularity in algorithms. Fortunately, this is exactly what makes the difference between signal processing and other applications. In signal processing, large amounts of data are processed in a continuous way, which makes the use of SIMD techniques promising.

13.1.1 Signal Processing Algorithms

Most signal processing algorithms fall into two categories: filter banks and Fourier-like transforms. Other algorithms are usually quite similar to one of the two, or include at least one of the two as an essential ingredient.

There are differences between the two categories. The most important one is that Fourier-type transforms operate on blocks of signal data, while filters operate on streams of data. Another difference is that filters have the simple algorithmic form of a convolution, whereas fast Fourier-type transforms employ more complicated butterfly-like schemes. Note also that it is possible to implement convolutions and, thus, filters via Fourier transforms by applying the convolution theorem. This method is feasible whenever the filters are long. Yet another difference is that Fourier-type algorithms usually operate on complex numbers, whereas filter banks are almost always real-valued.

Let us look at the basic algorithms in more detail. The simplest form of a finite impulse response (FIR) filter is

$$y(n) = \sum_k x(n-k)h(k)\,, \tag{13.1}$$

where x is the discrete input signal, y the output signal, and h the (finite) filter. For causal filters, k is non-negative. In any case, k has finite limits. The general case can have more than one input and output signals. This leads to the form

$$y_i(n) = \sum_j \sum_k x_j(n-k)h_{i,j}(k)\,. \tag{13.2}$$

Additionally, input and output signals can be down-sampled, i.e., only every m-th value has to be calculated in the output signal, or is non-zero in the input signal. While this reduces the computational demand by omitting zero products, as well as memory demands by omitting zero values from arrays, it complicates the algorithms. Moreover, some values of $h_{i,j}(k)$ may be equal, or just have opposite signs. This happens for symmetric filters and quadrature mirror filter pairs, for instance.

Depending on the position of the filter coefficients and down-sampling factors, this may lead to redundant products, which means further potential for computational reduction at the price of higher algorithmic irregularity. Finally, the filters may have "holes," i.e., inner zero coefficients. All this renders a general-purpose implementation highly inefficient. Each filter bank has to be handled individually, or automatic compilation techniques must be used.

Infinite impulse response (IIR) filters are an extension of FIR filters, where the output signal is reused as input signal.

$$y(n) = \sum_l y(n-l)a(l) + \sum_k x(n-k)b(k)\,, \tag{13.3}$$

where, of course, $l > 0$. The main difficulty in implementing this scheme is the recursive data flow that introduces loop dependencies and, thus, complicates parallelization and makes algebraic reformulations of the filter algorithm necessary.

On the other hand, Fourier-type algorithms are relatively irregular to start with. Despite the easy definition of the discrete Fourier transform

$$y(n) = \underset{N}{\mathscr{F}} x(n) = \sum_{k=0}^{N-1} x(k) e^{-i\frac{2\pi}{N}kn}\,, \tag{13.4}$$

where N is the size of the input signal block $(x(0),\ldots,x(N-1))$, and $0 \le n < N$, fast versions of the Fourier transform employ more complicated recursive reformulations such as

$$\underset{N}{\mathscr{F}} x = (\hat{x}_0 + \hat{x}_1, \hat{x}_0 - \hat{x}_1)\,, \quad \hat{x}_0 = \underset{N/2}{\mathscr{F}} x_0\,, \quad \hat{x}_1(n) = \underset{N/2}{\mathscr{F}} x_1(n) e^{-i\frac{2\pi}{N}n}\,, \tag{13.5}$$

where x is split into even samples $x_0 = (x(0), x(2), \ldots, x(N-2))$, and odd samples $x_1 = (x(1), x(3), \ldots, x(N-1))$. This scheme is due to Cooley and Tukey [1]. In this version, N has to be even for one recursion level and a power of two for full recursion (radix 2). Similar schemes can be found for other radices. Further schemes include the split-radix algorithm [2] and the Rader algorithm [3] for prime sizes N. All these schemes may be mixed and lead to different memory access patterns with different computational performances which depend also on machine properties. Automatic tuning systems have been developed [4, 5] which recursively search the space of possible implementations, starting from abstract formulations of the algorithms to rewriting schemes in dedicated signal processing languages such as SPL [6].

13.1.2 Short-Vector SIMD

In SIMD architectures, data is organized in registers containing vectors of several values. These registers can be used in operations such as multiplication and addition just as normal registers. The difference is that the values in the vectors are operated

on independently in parallel. Since it is common that a vector consists of $p = 4$ values, we will use this for demonstration throughout this chapter. A vector is written as $a = (a_0, a_1, a_2, a_3)$. Vector operators are displayed with circles:

$$a \odot b = (a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3), \quad a \oplus b = (a_0 + b_0, \ldots, a_3 + b_3)\,. \tag{13.6}$$

SIMD computers have been popular in the 1980s and early 1990s, mainly due to MasPar and the Connection Machines. Modern SIMD extensions of general purpose CPUs are different from those in that the vectors are much shorter, i.e., $p = 2$, 4, or 8, hence the name "short-vector SIMD." All these architectures have different constraints in accessing and arranging data in vector registers. While traditional vector computers only offered certain shift or rotation operations, new SIMD extensions include almost general variations of values in vector registers, written as

$$a_{(p,q,r,s)} = (a_p, a_q, a_r, a_s)\,, \tag{13.7}$$

or, in the more common form with two operands,

$$(a,b)_{(p,q,r,s)} = (c_p, c_q, c_r, c_s)\,, \tag{13.8}$$

where $c = (a,b) = (a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$, and $0 \leq p,q,r,s < 8$. Not all of these so-called *shuffle operations* are available as single instruction on all architectures. As an important example, in Intel MMX and SSE, the shuffle operation has the restriction that the first two values of the destination vector have to be from the first operand and the last two from the second operand, i.e., $0 \leq p,q < 4 \leq r,s < 8$ in Eq. (13.8). Additionally, there are two operations called "unpack operations" which interleave the values of the first or second halves of the source operands, i.e., $(a,b)_{(0,4,1,5)}$ and $(a,b)_{(2,6,3,7)}$. The maximum number of necessary instructions for an arbitrary shuffle operation is two. On the other hand, the Motorola AltiVec architecture provides instructions for arbitrary shuffle operations.

Architectures can also differ in the allowed numerical precisions, and in the vector size depending on the precision. The common configuration, though, is that vector registers have 128 bit, so they support 4-fold SIMD for single precision (i.e., 32 bit) and 2-fold SIMD for double precision floating point numbers (i.e., 64 bit). Integer numbers are also possible, but we will concentrate on floating point numbers in this chapter.

Another restriction of most SIMD architectures is that they require aligned data access to memory. This means that p consecutive values that are read from memory into a vector register must have a starting address that is a multiple of the vector size. As a consequence, the programmer has to take care that arrays are properly aligned when they are allocated, and that they are read and written in non-overlapping blocks of p values. Although some processors allow unaligned reads and writes, these are usually much slower than aligned accesses.

13.2 General Vectorization Approaches

Most compilers today include options to automatically vectorize the code in order to utilize SIMD extensions. Although these vectorizations rarely lead to optimal code, it is advisable to look at vectorization strategies that might also help in manual vectorization of our signal processing algorithms.

13.2.1 Loop Unrolling

If the inner loop of the algorithm contains only a small number of operations, as is the case for the filter algorithm, then a simple approach is to unroll p iterations of the inner loop, where p is the vector length. The corresponding p operations, one from each iteration, are scheduled to be executed in parallel in a vector instruction.

This approach has only one advantage and many disadvantages. The advantage is that the data to be processed probably lies consecutively in memory and can simply be read into a vector register. However, this is mostly not true for both, input and output data simultaneously. Moreover, the data is unlikely to be aligned. For instance, in a simple filter algorithm the data to be read is shifted by one for every outer loop iteration. Therefore, it is aligned only every p-th time.

If iterations depend on previous iterations, the method is hardly usable at all. This is partly so for the filter algorithm. The multiplication of source data with filter coefficients can be done in parallel, but the summation of the products is inherently serial. Some SIMD architectures provide instructions for horizontal sums which could be used in this situation. However, this reintroduces scalars in the algorithm and, therefore, is suboptimal.

Nevertheless, unrolling a larger number of iterations, or even the whole inner loop, may allow good vectorization through clever shuffling of data in registers. This is, however, a complex problem to solve, and is treated next.

13.2.2 Straight Line Code Vectorization

Algorithms may contain blocks of code with no loops at all. If not, such blocks can be produced by loop unrolling. Reference [7] presents a basic approach to automatic vectorization of such a block. It starts with the speculative aggregation of destination variables into vector variables, followed by a depth-first search for appropriately aggregated operations and source variables. If no feasible solution can be found, backtracking is used to explore other combinations of variables into vectors. Because a full search may be too expensive, heuristics are used for choosing good candidates for aggregation.

This optimizing compiler technique is used and is especially important in automatically tuned FFT packages [8, 9], where small FFTs are recursively expanded into straight line codelets which are then included in larger FFTs.

13.2.3 Loop Fusion

If an algorithm consists of several passes that process the same arrays of data, where each pass reads the data that a previous pass has written, these data accesses degrade the performance and make the algorithm dependent on large cache sizes. Often, it is possible to fuse these passes into a single one. This is done by interleaving the loop iterations of different passes. Of course, one has to make sure that data is not read by an iteration of a later pass before it is written by an iteration of an earlier pass. In other words, a proper rescheduling of all passes' loop iterations has to be applied through a reformulation of the algorithm that respects data dependencies.

As a consequence, intermediate data is likely to be read immediately after it is written. Therefore, it is better to remove these writes and reads in the first place and keep the data in registers, local variables, or local buffers instead. The resulting algorithm consists of a single fused loop containing a larger loop body. In addition to the improved performance due to decreased cache dependency, the larger loop body may be vectorized more easily using techniques for straight line code vectorization.

13.2.4 Loop Transposition

Most algorithms contain nested loops. The inner loop is likely to have dependencies between iterations, which makes vectorization difficult. On the other hand, the outer loop very often has independent iterations. This is the case, for instance, if the outer loop iterates the output index, and the output values are calculated independently from each other, or if the outer loop iterates rows of a row-wise transform.

It should then be possible to transpose the outer and inner loop in order to eliminate dependencies in the new inner loop. This corresponds to the commutation of sum operators if the algorithm is formulated as double sum. Temporary variables that pass data between iterations, such as running sums, have to be avoided or taken care of by storing one value for each outer iteration.

Of course, this introduces new memory accesses and reduces the parallel efficiency. Therefore, it may be better to transpose only blocks of the outer loop, ideally blocks of exactly p iterations. This leads to an algorithm that is basically a copy of the original algorithm, but operates on vectors instead of scalars. Temporary variables are kept in vectors as well and do not have to be saved.

This approach is a simple example of iteration rescheduling. It may have benefits even if the outer loop has dependencies. However, a disadvantage is that data access may not be contiguous any more. This can make shuffle operations or even redun-

dant data accesses necessary. In many cases, a simple $p \times p$ block transposition can solve the problem. Such a transposition can be implemented by

$$\begin{aligned} b^{(0)} &= (a^{(0)}, a^{(1)})_{(0,2,4,6)}, & b^{(1)} &= (a^{(2)}, a^{(3)})_{(0,2,4,6)}, \\ b^{(2)} &= (a^{(0)}, a^{(1)})_{(1,3,5,7)}, & b^{(3)} &= (a^{(2)}, a^{(3)})_{(1,3,5,7)}, \\ c^{(0)} &= (b^{(0)}, b^{(1)})_{(0,2,4,6)}, & c^{(1)} &= (b^{(2)}, b^{(3)})_{(0,2,4,6)}, \\ c^{(2)} &= (b^{(0)}, b^{(1)})_{(1,3,5,7)}, & c^{(3)} &= (b^{(2)}, b^{(3)})_{(1,3,5,7)}. \end{aligned} \tag{13.9}$$

This scheme uses a minimum of eight shuffle instructions and can also be used on Intel SSE architectures. It arranges non-consecutive data $(a_i^{(0)}, a_i^{(1)}, a_i^{(2)}, a_i^{(3)})$ into the vectors $c^{(i)}$. On the other hand, it distributes the consecutive data in vectors $a^{(j)}$ to corresponding slots of different vectors $(c_j^{(0)}, c_j^{(1)}, c_j^{(2)}, c_j^{(3)})$. Very often, algorithms can operate more easily on transposed vectors $c^{(i)}$.

13.2.5 Algebraic Transforms

If it is possible to reformulate an algorithm algebraically, it is worth checking whether the reformulation is more suitable for vectorization. Reformulations can be as simple as applying associative and distributive laws to addition and multiplication. The associative law can, for instance, reverse the dependencies of summing loops.

Moreover, it is important to distinguish between dynamic and static data. In our algorithms dynamic data is mainly signal data that keeps changing. Static data consists of filter or transform coefficients that are constant over loops and, in most cases, available at compile-time. By applying the distributive law, it can be possible to shift operations on dynamic data to operations on static data.

An example would be $a(x+y)+by$, where x and y represent dynamic signal data and a and b are static coefficients. This expression can be transformed into $ax+(a+b)y$, where $a+b$ can be calculated outside of the signal data loop, thus saving one addition per iteration.

This approach can also reduce shuffle operations if applied cleverly. Combined with loop unrolling and vector aggregation, the space of possible reformulations is usually large. Therefore, algorithm specific approaches have to be found, or automatic optimizers with heuristics have to be applied.

Exploring the space of reformulations is even more important for Fourier-type transforms. This is already done in optimized sequential algorithms [4, 5], as stated in Sect. 13.1.1. Vectorization of automatically generated straight-line code blocks (codelets) increases the necessity for testing different possible code blocks since some may be vectorized more efficiently than others. Inside the code blocks, the above method of algebraic reformulation could be applied if simple rescheduling, i.e., the aggregation strategy [8], is not sufficient. However, sequential optimization is usually the only algebraic reformulation step within code blocks.

13.3 Convolution Type Algorithms

The most common type of algorithm in signal processing is filtering. Filtering is basically a convolution of signal data $x(t)$ with the filter impulse response $h(t)$. If the impulse response is finite, the convolution can be implemented directly. If it is infinite or too large, a recursive formulation has to be found that is equal to, or approximates the filter. The latter will be treated in the next section.

In this section we will examine simple filters as well as more complex filter banks in order to develop and evaluate the most important vectorization approaches. As examples of filter banks, filter pairs which are common in wavelet transforms (see Sect. 13.6.1) are used. Automatic vectorization so far has not produced any performance increase for wavelet transforms [10, 11]. Also, approaches on old SIMD arrays [12–14] cannot be adapted directly. Therefore, good manual vectorization strategies [15, 16] are important.

Experimental results will also be presented, which were conducted on an Intel Pentium 4 CPU with 3.2 GHz and 2 MB cache size using the SSE extension with vectors 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.

13.3.1 Simple FIR Filter

The simplest case of an FIR filter has one input signal x and one output signal y, and does not apply any down- or upsampling. It is defined by

$$y(n) = \sum_k x(n-k)h(k)\,. \tag{13.10}$$

There are two loops, the inner one for k and the outer for n. The loop iteration dependencies are shown in Fig. 13.1. We will now vectorize this expression by various methods and evaluate their advantages and disadvantages. The first method to try is simple loop vectorization. It is depicted as method A in Fig. 13.1. Four consecutive iterations shall be combined into one vectorized iteration. However, as the sum operation imposes dependencies between iterations, we have to break the parallelity. We get

$$y(n) = \sum_k S(x(n-4k-m,\ldots,n-4k-m+3) \odot h(4k+m,\ldots,4k+m-3))\,. \tag{13.11}$$

The operator $S()$ calculates the scalar sum of a vector's elements. On some architectures there is an instruction that implements the S-operator. If there is no such

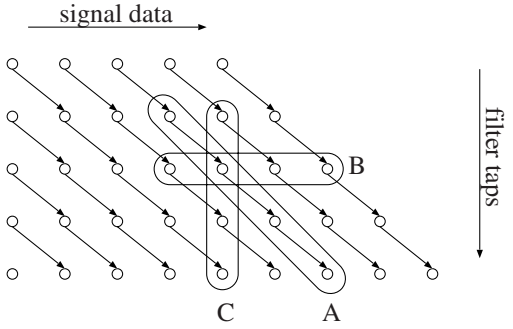


Fig. 13.1 Loop iteration dependencies and vectorization strategies for simple FIR filtering.

instruction, a sequence of shuffle and add operations followed by an element extraction must be used, which is costly and may degrade the performance.

The dislocation parameter m does not have an influence on the result. It has, however, an influence on the range of k. If indices of $h(\cdot)$ lie outside of its finite support, h has to be padded with zeros, which introduces redundant calculations and degrades the parallel efficiency, especially for short filters. For causal filters, where indices have a minimum of 0, $m = 3$ avoids zero padding at least at the lower end of indices. m also determines the alignment of vectorized data access. To make the read operations on x aligned, m should depend on n such that $n - m$ is a multiple of the vector size p, i.e., four in our examples. The alignment of read operations on h cannot be set independently, but this could be solved by preparing p copies of h with different alignments.

The application of the S operator already makes mild use of the associative law. It can be further exploited to vectorize most of the summing operation by commutating the sum and S operator:

$$y(n) = S\left(\sum_k x(n-4k-m,\ldots,n-4k-m+3) \odot h(4k+m,\ldots,4k+m-3)\right). \tag{13.12}$$

There are still scalar operations in this algorithm such as the S operator and also the store operation on y. To make the entire process parallel, we have to look for a different approach. Therefore, we make use of the loop transposition method described in Sect. 13.2.4 by introducing another index l that shall be used to vectorize blocks of n-indices. It turns out that we have two options to reformulate Eq. (13.10), namely

$$B:\ y(n+l) = \sum_k x(n+l-k)h(k), \quad \text{and} \tag{13.13}$$

$$C:\ y(n+l) = \sum_k x(n-k)h(k+l). \tag{13.14}$$

Let us look at method C first. The resulting vectorization strategy is depicted in Fig. 13.1 as C, and can be formulated as

$$y(n,\ldots,n+3) = \sum_k x(n-k)_{(0,0,0,0)} \odot h(k,k+1,k+2,k+3)\,, \tag{13.15}$$

where the so-called *splat* operator $a_{(0,0,0,0)} = (a,a,a,a)$ on a scalar a creates a vector filled with the value a. We see that this method is still not completely vectorized because it reads the x array sequentially before applying the splat operator. However, this may be circumvented by vectorized reads followed by four simple shuffle operations for each read, i.e., $x(n-k,\ldots,n-k+3)_{(i,i,i,i)}$ for $0 \le i < 4$.

Note that the range of the index k has to be extended to generate all products. For causal filters, k has to start at $k = -3$. This introduces the need of additional zero-padding of h and, as a consequence, redundant operations. Moreover, the access of the h array is entirely non-aligned.

Therefore, our hope lies in method B. Its vectorization strategy is depicted in Fig. 13.1 as B, and can be formulated as

$$y(n,\ldots,n+3) = \sum_k x(n-k,\ldots,n-k+3) \odot h(k)_{(0,0,0,0)}\,. \tag{13.16}$$

This method has the big advantage that no zero-padding of h is necessary. Therefore, there are no redundant calculations. Two disadvantages are the non-aligned access of x and the sequential access of h. The latter problem can be reduced by preparing vectors $h(k)_{(0,0,0,0)}$ in advance, which is favorable especially for short filters.

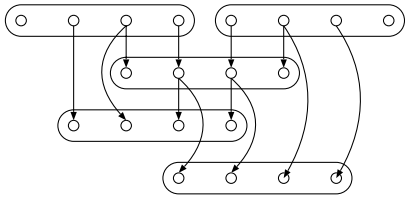


Fig. 13.2 Shuffle operations for all vector realignments on Intel architecture.

The non-aligned access of x implies one shuffle operation per non-aligned read, i.e., $p-1=3$ shuffles for $p=4$ reads. However, these shuffle operations may not be available as single instructions on certain architectures. Unfortunately, this is the case for Intel SSE. However, as all possible realignments are necessary, shuffled vectors can be reused in other shuffle operations to also achieve a rate of one shuffle per non-aligned read. The method is depicted in Fig. 13.2 and can be written as

$$\begin{aligned}
a &= (x(n,\ldots,n+3), x(n+4,\ldots,n+7))_{(2,3,4,5)}\,,\\
x(n+1,\ldots,n+4) &= (x(n,\ldots,n+3), a)_{(1,2,5,6)}\,,\\
x(n+2,\ldots,n+5) &= a\,,\\
x(n+3,\ldots,n+6) &= (a, x(n+4,\ldots,n+7))_{(1,2,5,6)}\,.
\end{aligned} \tag{13.17}$$

To summarize, we have applied the associative law and the loop transposition method to reschedule and reformulate loop iterations in order to vectorize the simple FIR filter algorithm. Method B turns out to be the most efficient due to the lack of

redundant calculations. This is confirmed by experiments. We will now apply these insights in the vectorization of some exemplary and more complicated filter banks.

13.3.2 The Haar Filter

The Haar filter is the simplest orthogonal wavelet filter. It is a 2-tap 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, where the low- and high-pass filters form a filter bank. Together with down-sampling by a factor of 2, the following assignments define the filtering algorithm of the Haar wavelet transform.

$$\text{for all } i : L(i) \leftarrow ax(2i) + ax(2i+1),\ H(i) \leftarrow ax(2i) - ax(2i+1) \tag{13.18}$$

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

$$\text{for all } i : p \leftarrow ax(2i),\ q \leftarrow ax(2i+1),\ L(i) \leftarrow p+q,\ H(i) \leftarrow p-q\,. \tag{13.19}$$

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 we want to read and write only full vectors when using SIMD, we consequently have to read two vectors in each iteration. We find the vectorization of the Haar filter as

$$\begin{array}{l}\text{for all } i : \\ \quad p \leftarrow x(8i,\ldots,8i+3) \odot a_{(0,0,0,0)},\ q \leftarrow x(8i+4,\ldots,8i+7) \odot a_{(0,0,0,0)}, \\ \quad r \leftarrow (p,q)_{(0,2,4,6)}, \qquad\qquad s \leftarrow (p,q)_{(1,3,5,7)}, \\ \quad L(4i,\ldots,4i+3) \leftarrow r \oplus s, \qquad H(4i,\ldots,4i+3) \leftarrow r \ominus s\,.\end{array} \tag{13.20}$$

In the first line two perfectly aligned vectors are read and each element is immediately multiplied by the coefficient a. In the second line the elements are rearranged into one vector containing all even elements and one containing all uneven elements using shuffle operations. To calculate the sum and difference of every two neighboring elements, we just have to add and subtract the two vectors, 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.

13.3.3 Biorthogonal 7/9 Without Lifting

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 [17]. 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.

13.3.3.1 Sequential Algorithm

The biorthogonal 7/9 filter is an example of an uneven, symmetrical filter. It has 9 low-pass (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

$$\begin{aligned}
&\text{for all } i: \\
&\quad L(i) \leftarrow ax(2i-4)+bx(2i-3)+cx(2i-2)+dx(2i-1)+ex(2i) \\
&\qquad\quad +dx(2i+1)+cx(2i+2)+bx(2i+3)+ax(2i+4), \\
&\quad H(i) \leftarrow px(2i-2)+qx(2i-1)+rx(2i)+sx(2i+1) \\
&\qquad\quad +rx(2i+2)+qx(2i+3)+px(2i+4)\,.
\end{aligned} \tag{13.21}$$

However, this algorithm can be optimized in terms of the number of required multiplications 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.

$$\begin{aligned}
&\text{for all } i: \\
&\quad L(i) \leftarrow a(x(2i-4)+x(2i+4))+b(x(2i-3)+x(2i+3)) \\
&\qquad\quad +c(x(2i-2)+x(2i+2))+d(x(2i-1)+x(2i+1))+ex(2i)\,, \\
&\quad H(i) \leftarrow p(x(2i-2)+x(2i+4))+q(x(2i-1)+x(2i+3)) \\
&\qquad\quad +r(x(2i)+x(2i+2))+sx(2i+1)\,.
\end{aligned} \tag{13.22}$$

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. 13.3. 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.

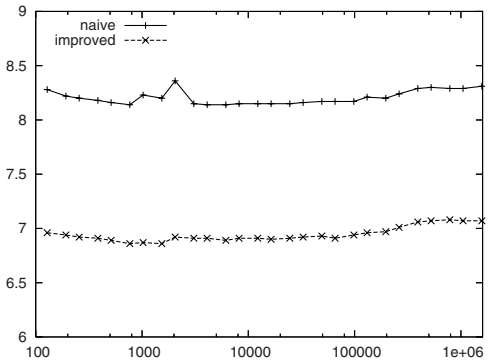


Fig. 13.3 Execution time of naive and improved sequential algorithm in ns/sample. The *horizontal axis* shows the size of the repeatedly transformed data set in number of single precision values.

13.3.3.2 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 vectors have to be read to calculate one new low-pass vector and one new high-pass vector. However, since the filter is now longer than the two taps, the contents of more than two vectors 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 vectors are immediately multiplied by all necessary filter coefficients. Then appropriate shuffles of the products have to be added, leading to the following algorithm:

$$
\begin{aligned}
&\text{for all } i: \\
&\quad Y \leftarrow x(8i+4,\ldots,8i+7), Z \leftarrow x(8i+8,\ldots,8i+11) \\
&\quad A \leftarrow C,\ B \leftarrow D,\ C \leftarrow Y \odot (a,b,a,b),\ D \leftarrow Z \odot (a,b,a,b), \\
&\quad E \leftarrow G,\ F \leftarrow I,\ \ G \leftarrow Y \odot (c,d,c,d),\ I \leftarrow Z \odot (c,d,c,d), \\
&\quad J \leftarrow M,\ K \leftarrow N,\ M \leftarrow Y \odot (e,0,e,0),\ N \leftarrow Z \odot (e,0,e,0), \\
&\quad L(4i,\ldots,4i+3) \leftarrow (A,B)_{(0,2,4,6)} \oplus (A,B)_{(1,3,5,7)} \oplus (E,F,G)_{(2,4,6,8)} \oplus \\
&\qquad (E,F,G)_{(3,5,7,9)} \oplus (K,M)_{(0,2,4,6)} \oplus (F,G)_{(1,3,5,7)} \oplus (F,G,I)_{(2,4,6,8)} \oplus \\
&\qquad (B,C,D)_{(3,5,7,9)} \oplus (C,D)_{(0,2,4,6)}, \\
&\quad P \leftarrow R,\ Q \leftarrow S,\ \ R \leftarrow Y \odot (p,q,p,q),\ S \leftarrow Z \odot (p,q,p,q), \\
&\quad T \leftarrow V,\ U \leftarrow W,\ V \leftarrow Y \odot (r,s,r,s),\quad W \leftarrow Z \odot (r,s,r,s), \\
&\quad H(4i,\ldots,4i+3) \leftarrow (P,Q,R)_{(2,4,6,8)} \oplus (P,Q,R)_{(3,5,7,9)} \oplus (U,V)_{(0,2,4,6)} \oplus \\
&\qquad (U,V)_{(1,3,5,7)} \oplus (U,V,W)_{(2,4,6,8)} \oplus (Q,R,S)_{(3,5,7,9)} \oplus (R,S)_{(0,2,4,6)}
\end{aligned}
\tag{13.23}
$$

Figure 13.4 depicts the algorithm as a data-flow diagram. After multiplying the two new source vectors by vectors of appropriate filter coefficients, they are rearranged by shuffle operations (thin arrows) so that the sum of the resulting vectors is

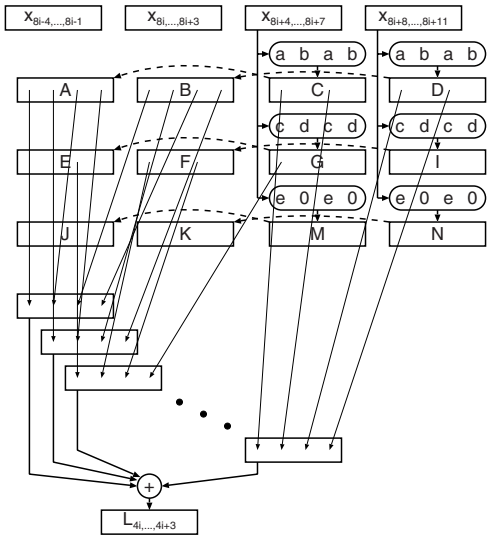


Fig. 13.4 Variant 1 of SIMD-parallel algorithm. Vectors are indicated by boxes, multiplication by boxes with rounded edges, addition by a circle with a $+$, shuffle operations by *thin arrows*, and the passing of values between iterations by *dashed arrows*. Only the low-pass calculations are shown, high-pass operations are similar.

the desired destination vector containing four low-pass filtered samples. Note that the intermediate vectors (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 instructions. One reason for this is that some such operations require three source vectors. Another reason is that the Intel processor's instruction set does not allow arbitrary shuffles. Altogether this algorithm can be implemented by 10 multiplies, 14 adds, and 26 shuffles.

13.3.3.3 SIMD Parallelization – Variant 2

A major disadvantage of the first variant is that values that have to be collected in a single vector are spread over several intermediate vectors, requiring more shuffle operations. The reason for this is that downsampling causes every second value to belong together. Therefore, the second variant inserts a single step of shuffling before the multiplication, putting even and odd samples into separate vectors. This leads to the following algorithm, which is also shown in Fig. 13.5.

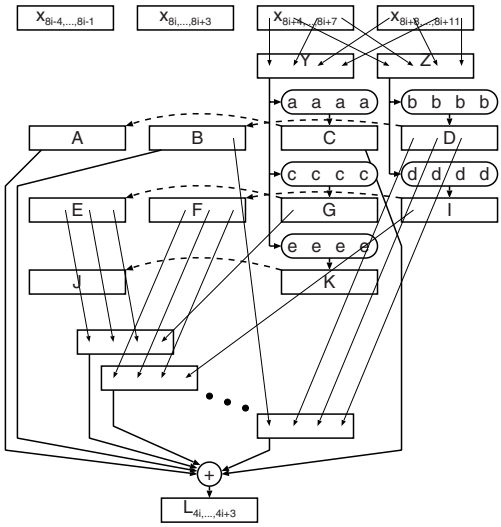


Fig. 13.5 Variant 2 of SIMD-parallel algorithm.

for all i :

$$
\begin{aligned}
&Y \leftarrow x(8i+4,8i+6,8i+8,8i+10), Z \leftarrow x(8i+5,\ldots,8i+11),\\
&A \leftarrow C,\ B \leftarrow D,\ \ C \leftarrow Y \odot (a,a,a,a),\ \ D \leftarrow Z \odot (b,b,b,b),\\
&E \leftarrow G,\ F \leftarrow I,\ \ \ G \leftarrow Y \odot (c,c,c,c),\ \ I \leftarrow Z \odot (d,d,d,d),\\
&J \leftarrow K,\qquad\qquad K \leftarrow Y \odot (e,e,e,e),\\
&L(4i,\ldots,4i+3) \leftarrow A \oplus B \oplus (E,G)_{(1,2,3,4)} \oplus (F,I)_{(1,2,3,4)} \oplus \qquad (13.24)\\
&\quad (J,K)_{(2,3,4,5)} \oplus (F,I)_{(2,3,4,5)} \oplus (E,G)_{(3,4,5,6)} \oplus (B,D)_{(3,4,5,6)} \oplus C\\
&P \leftarrow R,\ \ Q \leftarrow S,\ \ R \leftarrow Y \odot (p,p,p,p),\ S \leftarrow Z \odot (q,q,q,q),\\
&T \leftarrow V,\ \ U \leftarrow W,\ V \leftarrow Y \odot (r,r,r,r),\quad W \leftarrow Z \odot (s,s,s,s),\\
&H(4i,\ldots,4i+3) \leftarrow (P,R)_{(1,2,3,4)} \oplus (Q,S)_{(1,2,3,4)} \oplus (T,V)_{(2,3,4,5)} \oplus\\
&\quad (U,W)_{(2,3,4,5)} \oplus (T,V)_{(3,4,5,6)} \oplus (Q,S)_{(3,4,5,6)} \oplus R
\end{aligned}
$$

This has two advantages. First, there is one multiplication less for the e-coefficient. Second, no shuffle requires more than two source vectors. 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.

13.3.3.4 SIMD Parallelization – Variant 3

The third variant adopts the scheme of the improved sequential algorithm. First, the input vectors are shuffled so that the remaining operations can be performed as in the sequential case. This reverses the order of phases completely. Then, vectors that have to be multiplied by the same filter coefficients are added, followed by multiplication and the final sum. The following algorithm is also shown in Fig. 13.6.

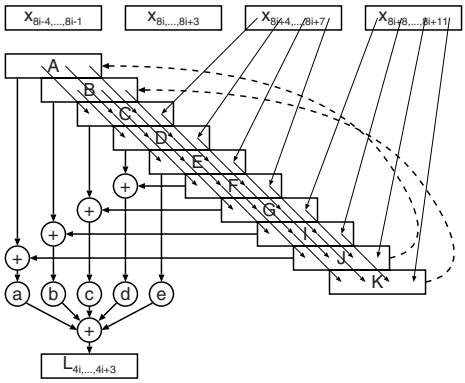


Fig. 13.6 Variant 3 of SIMD-parallel algorithm. Multiplication by a vector of equal coefficients is depicted by a single circle.

$$
\begin{aligned}
&\text{for all } i: \\
&\quad Y \leftarrow x(8i+4,\ldots,8i+7), Z \leftarrow x(8i+8,\ldots,8i+11), \\
&\quad A \leftarrow J, B \leftarrow K, \quad C \leftarrow (A,Y)_{(1,2,3,4)},\ D \leftarrow (B,Y)_{(1,2,3,5)}, \\
&\quad E \leftarrow (C,Y)_{(1,2,3,6)}, F \leftarrow (D,Y)_{(1,2,3,7)}, G \leftarrow (E,Z)_{(1,2,3,4)}, \\
&\quad I \leftarrow (F,Z)_{(1,2,3,5)},\ J \leftarrow (G,Z)_{(1,2,3,6)},\ K \leftarrow (I,Z)_{(1,2,3,7)}, \\
&\quad L(4i,\ldots,4i+3) \leftarrow (A \oplus J) \odot (a,a,a,a) \oplus (B \oplus I) \odot (b,b,b,b) \oplus \\
&\qquad (C \oplus G) \odot (c,c,c,c) \oplus (D \oplus F) \odot (d,d,d,d) \oplus E \odot (e,e,e,e) \\
&\quad H(4i,\ldots,4i+3) \leftarrow (C \oplus J) \odot (p,p,p,p) \oplus (D \oplus I) \odot (q,q,q,q) \oplus \\
&\qquad (E \oplus G) \odot (r,r,r,r) \oplus F \odot (s,s,s,s)
\end{aligned}
\tag{13.25}
$$

Note that only two vectors have to be passed to the next iteration. This reduces the stress on 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. 13.6, 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.

13.3.3.5 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 vectors 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. 13.7. 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.

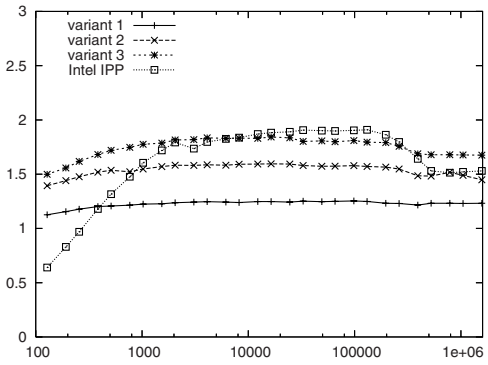


Fig. 13.7 Speedups of the SIMD parallelization variants against the improved sequential algorithm. The *horizontal axis* again shows the size of the repeatedly transformed data set.

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.

13.3.3.6 Applicability to Arbitrary Filter Banks

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 filters 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 vector for each filter tap to pass it to the next iteration. This requires the allocation of many CPU registers and leads to additional memory accesses when the compiler runs out of available registers. On the other hand, variant 3 has to keep all shuffled vectors in registers, whereas variants 1 and 2 can drop shuffled vectors

(and even some other intermediate vectors) 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 above.

13.3.4 Biorthogonal 7/9 With Lifting

As most wavelet filters, the biorthogonal 7/9 filter can also be implemented by applying the lifting scheme [18]. It is a method to implement wavelet filter pairs in a joint pass. In this way it is possible to reduce the total number of operations.

13.3.4.1 Sequential Algorithm

The lifting approach 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 requires only 6 multiplies.

$$\begin{aligned}
&\text{for all } i : x(2i+1) \leftarrow x(2i+1) + a(x(2i) + x(2i+2)),\\
&\text{for all } i : x(2i) \leftarrow x(2i) + b(x(2i-1) + x(2i+1)),\\
&\text{for all } i : x(2i+1) \leftarrow x(2i+1) + c(x(2i) + x(2i+2)),\\
&\text{for all } i : x(2i) \leftarrow x(2i) + d(x(2i-1) + x(2i+1)),\\
&\text{for all } i : x(2i+1) \leftarrow -ex(2i+1),\\
&\text{for all } i : x(2i) \leftarrow \tfrac{1}{e}x(2i)
\end{aligned} \tag{13.26}$$

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.

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. We can vectorize these operations by

$$x(2n,\ldots,2n+3) \leftarrow x(2n,\ldots,2n+3)+ \\ (\alpha,0,\alpha,0)\odot(x(2n-1,\ldots,2n+2)+x(2n+1,\ldots,2n+4)). \quad (13.27)$$

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 four 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 (see Sect. 13.2.3). 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 one high-pass coefficient. This leads to the following algorithm:

$$\begin{array}{l} \text{for all } i: \\ \quad o \leftarrow q,\ p \leftarrow x(2i+3),\ q \leftarrow x(2i+4), \\ \quad r \leftarrow s,\ s \leftarrow p+a(o+q), \\ \quad t \leftarrow u,\ u \leftarrow o+b(r+s), \\ \quad v \leftarrow w,\ w \leftarrow r+c(t+u), \\ \quad L(i) \leftarrow t+d(v+w)\cdot\frac{1}{e},\ H(i) \leftarrow w\cdot(-e). \end{array} \quad (13.28)$$

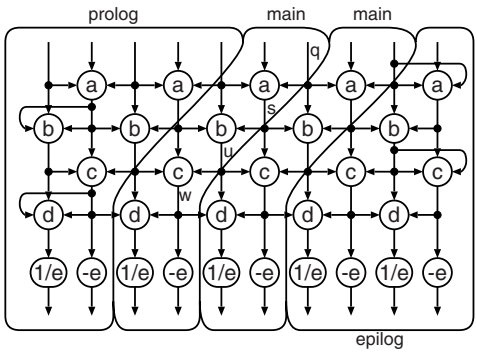


Fig. 13.8 Sequential single-loop algorithm for the biorthogonal 7/9 filter with lifting. Circles with three inputs (l left, r right, u upper) denote basic lifting operations $y = u+\alpha(l+r)$. Rounded frames indicate single iterations.

This algorithm is also shown in Fig. 13.8 for a very short data length of 10. Iterations, 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. 13.8. These four values have to be set properly at the beginning of the loop. Also, the end of the

loop needs special treatment. Figure 13.8 shows how this must be done in the case of mirroring border handling in the phases denoted by "prolog" and "epilog."

13.3.4.2 SIMD Parallel Algorithm

To be able to obtain speedup using SIMD operations, again full vectors have to be read. 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 algorithm, which leads to the following algorithm:

$$
\begin{array}{llll}
\text{for all } i: & & & \\
\quad h \leftarrow x_2, & x_1 \leftarrow x(8i+4,\ldots,8i+7), & x_2 \leftarrow x(8i+8,\ldots), & \\
\quad q \leftarrow (h,x_1)_{(0,2,4,6)}, & p \leftarrow (h,x_1,x_2)_{(3,5,7,9)}, & o \leftarrow (h,x_1)_{(2,4,6,8)}, & \\
\quad r \leftarrow s, & s \leftarrow (a,a,a,a) \odot (o \oplus q) \oplus p, & r \leftarrow (r,s)_{(3,5,6,7)}, & \\
\quad t \leftarrow u, & u \leftarrow (b,b,b,b) \odot (r \oplus s) \oplus o, & t \leftarrow (t,u)_{(3,5,6,7)}, & \\
\quad v \leftarrow w, & w \leftarrow (c,c,c,c) \odot (t \oplus u) \oplus r, & v \leftarrow (v,w)_{(3,5,6,7)}, & \\
\quad L(4i,\ldots,4i+3) \leftarrow ((d,d,d,d) \odot (v \oplus w) \oplus t) \odot (\frac{1}{e},\frac{1}{e},\frac{1}{e},\frac{1}{e}), & & & \\
\quad H(4i,\ldots,4i+3) \leftarrow (-e,-e,-e,-e) \odot w. & & &
\end{array}
\tag{13.29}
$$

See also Fig. 13.9 for a data-flow diagram of the algorithm.

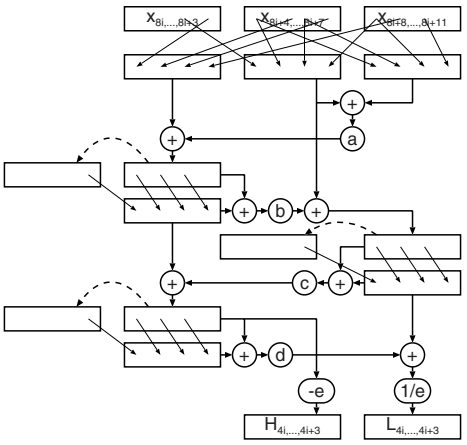


Fig. 13.9 SIMD-algorithm of biorthogonal 7/9 filter with lifting. Heavy use of shuffle-operations may cause non-optimal speedups. Like in the sequential case, intermediate values are passed between iterations (*dashed lines*).

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 Sect. 13.3.3.2. Variants 1 and 2 could also be used here. However, consid-

erations 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 in a straight forward way because SIMD extensions (e.g., Intel SSE 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.

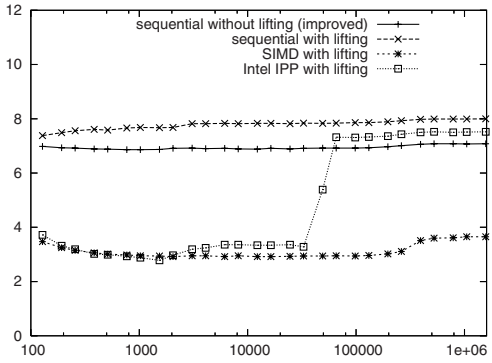


Fig. 13.10 Execution times in ns/sample of sequential and SIMD implementations with and without lifting over the size of the repeatedly transformed data set (number of floats).

13.3.4.3 Experimental Results

Figure 13.10 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 [18] 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. Figure 13.11 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 $1.64 \times 4 = 6.56$) 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. 13.10. It shows equal performance for small

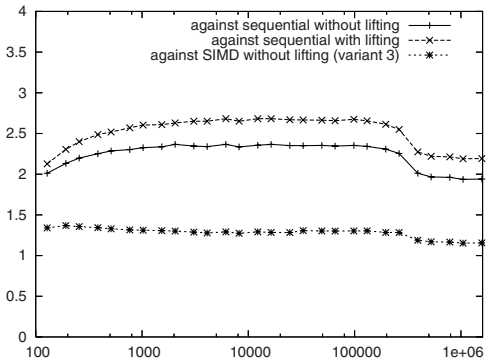


Fig. 13.11 Speedup of the SIMD implementation with lifting against implementations without lifting or SIMD.

and slightly worse performance 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.

13.3.5 Conclusion

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 above represent purely algorithmic improvements.

Apart from speedup issues, algorithms have to be found to derive optimal solutions. This is important because each parallelization presented here is one of many possible solutions and it is still possible that the shown solutions can 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 [19] are required.

13.4 Recursive Algorithms

Algorithms of the convolution type are non-recursive, which means that output values are independent of each other. Whenever previous output values are reused in the computation of new values, the algorithm is called recursive. The IIR filter technique is the most important example of such an algorithm. Therefore, we shall investigate it and examine vectorization strategies.

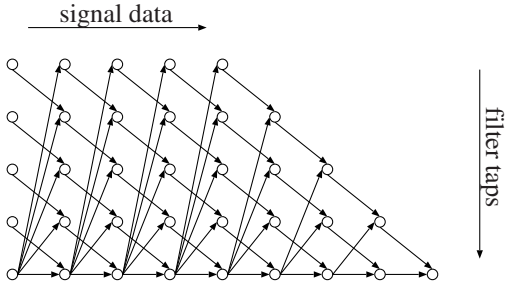


Fig. 13.12 Loop dependencies in IIR filtering.

From a computational point of view, the difference between FIR and IIR filters lies in the dependencies between loop iterations. Again, there are two loops, one over signal data and the other over filter taps. In the FIR case, iterations of the outer loop, i.e., entire inner loops, are independent of each other, leading to a rather straight-forward SIMD parallelization where the two loops (inner and outer) are transposed for a number of outer iterations equal to the SIMD vector size p, as shown in Sect. 13.3.1. In the IIR case, the dependencies are more complicated since all previous output values are required to calculate a new one. See Fig. 13.12 and compare to Fig. 13.1. Therefore, SIMD parallelization is more difficult.

In this section we will first apply usual rescheduling techniques and then show how algebraic transforms of the algorithm can improve the vectorization significantly, which is verified by experimental results. These are conducted on an Intel Pentium 4 CPU with 3.2 GHz and 2 MB cache size using the SSE extension with vectors 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 4.1.2 with the -O3 option. SIMD operations are implemented using gcc's built-in intrinsics for vector extensions and the -msse option. Note that in order to have full control over generated code, no automatic vectorization is applied. The results are compared to the hand-optimized Intel Integrated Performance Primitives (IPP) v5.3. Note that the IPP library also uses SIMD operations, but the applied methods are not known to the author.

13.4.1 Sequential IIR Algorithm

The goal of IIR filtering is to calculate the signal y from the signal x by

$$y(n) = \sum_{i=1}^{N-1} a(i)y(n-i) + \sum_{i=0}^{M-1} b(i)x(n-i)\,, \tag{13.30}$$

where the second term is an FIR part with coefficients $b(i)$ and the first term is the IIR part with coefficients $a(i)$. M is the number of FIR filter taps and N is the number of IIR filter taps. The formula reveals the outer loop over n and two inner loops over i.

The sequential implementation is optimized for performance to have a reasonable comparison for the SIMD parallelized version. It turns out that maintaining a pointer for $y(n)$ and $x(n)$ and addressing $x(n-i)$ and $y(n-i)$ via relative addressing is fastest. Using extra buffers or local register variables for reused values does *not* improve the performance. Therefore, a similar implementation style is adopted for the SIMD parallelization.

13.4.2 Scheduling Approach

Rescheduling approaches only change the order in which iterations and operations are executed. They have therefore limited power if there are too many data dependencies, as there are in IIR filtering. Examples can be found in [20,21]. We will use a rather straight forward approach that will be improved by algebraic transforms in the next section.

The FIR part is vectorized simply as in Sect. 13.3.1 (method B), with the result given in u. The IIR part can be parallelized in just the same way for those iterations where $i \geq p$, i.e., where the source vector $y(n-i,\ldots,n-i+p-1)$ does not overlap with the destination vector $y(n,\ldots,n+p-1)$ that is being calculated. The iterations $i = 0,\ldots,p-1$ might be implemented sequentially after computing the others in a vectorized way first by

$$v = u \oplus \sum_{i=p}^{N-1} y(n-i,\ldots,n-i+p-1) \odot (a_i,\ldots,a_i)\,, \tag{13.31}$$

followed by

$$y(n+k) = v_k + \sum_{i=1}^{p-1} a(i)y(n+k-i) \quad \text{for} \quad k = 0,\ldots,p-1. \tag{13.32}$$

A first attempt to parallelize the latter part is to split it into two phases. The first phase treats those terms that reference $y(n+k-i)$ where $n+k-i < n$, i.e., already available values.

$$\begin{aligned}&\text{for } i = 1,\ldots,p-1\text{:}\\ &\quad v \leftarrow v \oplus (y(n-p+i),\ldots,y(n-1),0,\ldots) \odot \\ &\qquad (a(p-i),\ldots,a(p-i),0,\ldots)\end{aligned} \tag{13.33}$$

The second phase uses those elements of v that already represent $y(n+k)$ values. At the beginning, only $v_0 = y(n)$. Using this value, v_1 can be calculated to hold $y(n+1)$, and so on. This leads to the following algorithm:

$$\begin{aligned}&\text{for } k = 0,\ldots,p-2:\\ &\quad v \leftarrow v \oplus (\ldots,0,v_k,\ldots,v_k) \odot (\ldots,0,a_1,\ldots,a_{p-1-k})\\ &y(n,\ldots,n+p-1) \leftarrow v\end{aligned} \tag{13.34}$$

This first approach yields an overhead of $p-1$ multiply-accumulate vector operations, since each phase has $p-1$ iterations, resulting in $2(p-1)$ operations, where only $p-1$ would be necessary if there were no problems with data dependencies.

13.4.3 Algebraic Transforms

Algebraic transforms of the algorithm can be used to eliminate troubling data dependencies [22]. Here, we will follow an approach that fuses filter taps together to resolve data dependencies [23]. Let us look at the second iteration ($k = 1$) of the last algorithm. Here, $v_1 = y(n+1) = v'_1 + v_0 a(1)$, where v' comes from the preceding iteration. Now, we calculate the new v_2 as $v_2 + v_1 a(1)$, which can consequently be expressed as $v_2 + v'_1 a(1) + v_0 a(1)^2$. Moreover, $v_2 = v'_2 + v_0 a(2)$, as calculated in the first iteration. Together, we get $v'_1 a(1) + v_0(a(1)^2 + a(2))$. The term $v'_1 a(1)$ could be calculated in the last iteration of the first phase, and the term $v_0(a(1)^2 + a(2))$ can be calculated in the first iteration of the second phase because we have eliminated v_1 from the term.

Following this approach even further recursively, we get the following algorithm that substitutes both phases:

$$\begin{aligned}&\text{for } i = 1,\ldots,p:\\ &\quad v \leftarrow v \oplus (y(n-p+i),\ldots,y(n-1),0,v_i,\ldots v_i) \odot s(i)\\ &y(n,\ldots,n+p-1) \leftarrow v\end{aligned} \tag{13.35}$$

$s(i)$ holds the fused filter tap coefficients and has the following form:

$$\begin{aligned} s(1) &= (a(p-1),\ldots,a(p-1),0)\\ s(2) &= (a(p-2),\ldots,a(p-2),0,c(1))\\ &\ldots\\ s(p-1) &= (a(1),0,c(1),c(2),\ldots,c(p-2))\\ s(p) &= (0,c(1),c(2),\ldots,c(p-1)), \end{aligned} \tag{13.36}$$

where

$$c(k) = \sum_{i=1}^{k} a(k)c(k-i), \qquad c(0) = 1\,. \tag{13.37}$$

This approach finally has only an overhead of one multiply-accumulate vector operation, since it has p iterations. For better comprehensibility, let us write the algorithm for the case $p = 4$ as in the Intel SSE architecture:

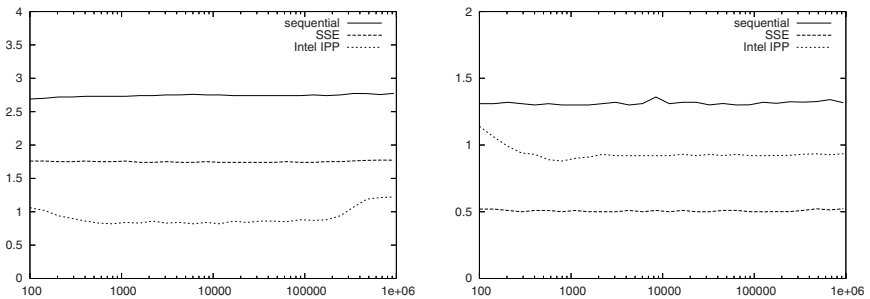
$$\begin{aligned}
&v \leftarrow v \oplus (y(n-3), y(n-2), y(n-1), 0) \odot (a(3), a(3), a(3), 0)\\
&v \leftarrow v \oplus (y(n-2), y(n-1), 0, v_2) \odot (a(2), a(2), 0, a(1))\\
&v \leftarrow v \oplus (y_{n-1}, 0, v_1, v_1) \odot (a(1), 0, a(1), a(1)^2 + a(2))\\
&v \leftarrow v \oplus (0, v_0, v_0, v_0) \odot (0, a(1), a(1)^2 + a(2), a(1)^3 + 2a(1)a(2) + a(3))\\
&y(n, \ldots, n+3) \leftarrow v
\end{aligned}$$

Of course, each operation requires at least one shuffle operation, maybe two on the Intel SSE architecture.

If the number of IIR-taps N is smaller than the vector size p, the above approach unfortunately only reduces to $p-1$ operations. In this case, some divide-and-conquer algorithm might further reduce the overhead. However, $\lceil \log_2(p+1) \rceil$ seems to be the lower bound, since $y(n+p-1)$ depends on the $p+1$ values $u_0, \ldots, u_{p-1}, y(n-1)$ if N takes the minimal value 2.

13.4.4 Experimental Results

In Sect. 13.3 we have seen that the performance of an implementation of a filtering algorithm possibly depends on whether the signal data is in the cache or not. Therefore, we will adopt the method of varying data size to examine the cache behavior.



(a) 2 taps (b) 10 taps

Fig. 13.13 Execution time in ns per sample point and filter tap depending on the data length for repeated filtering, showing the cache dependency of the algorithms.

The calculation time is expected to depend linearly on the data size and on the number of filter taps $N+M$. Therefore, we calculate the execution time per sample point and filter tap from the total execution time of the algorithm by $t_{\text{total}}/S/(N+M)$, where S is the data size.

Figure 13.13 shows the results for $N=M=2$ and $N=M=10$. It also includes performance measures of the Intel IPP library. While the IPP library code seems to depend a little on the data size, the major reason for this seems to be startup-overhead when filling the delay-lines, which is significant only for small data sizes. The sequential algorithm and the SIMD algorithm are completely independent of the cache state.

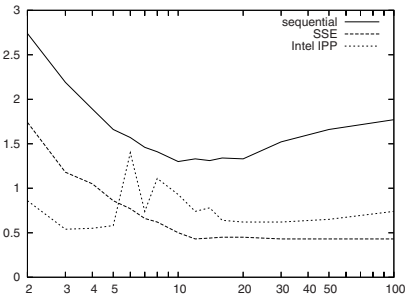


Fig. 13.14 Execution time in ns per sample point and filter tap depending on the number of filter taps.

For small numbers of taps, the IPP library code seems to be faster. This is also shown in Fig. 13.14. For $N=M\leq 5$, the SIMD algorithm cannot compete with the IPP code. The reason is probably that hand-optimized assembler code, as in the IPP library, is more important for short loops. For $N>5$, however, our SIMD approach outperforms the IPP library by a speedup of about 1.7 and also shows more regular behavior. Compared to the sequential algorithm, speedups from 1.5 for small N to 4.5 for large N are obtained.

13.5 Block Algorithms

Algorithms that operate on blocks of signal data usually have a more irregular structure than streaming algorithms such as filtering. The most prominent example is, of course, the FFT as defined in Sect. 13.1.1. Almost all other blocked transforms are variants of the FFT and have very similar structure. As a consequence, vectorization strategies are basically the same. Therefore, we will concentrate on the FFT.

13.5.1 Data Layout

The FFT operates on complex data, which raises the question where real and imaginary parts of complex numbers are stored. The most common is an alternating scheme to keep real and imaginary parts closely together. The other possibility is to store them in separate arrays. What does that mean for vectorization efficiency? In the alternating scheme, $\frac{p}{2} = 2$ complex numbers are kept in a vector. Simultaneous addition of $2+2$ complex numbers simply takes the form of a vector addition. However, vectorized multiplication is more complicated. The point-wise complex product of arrays $z(n) = x(n)y(n)$ can be implemented by

$$\begin{array}{l}\text{for all } n: \\ \quad a \leftarrow (\Re x(n), \Im x(n), \Re x(n+1), \Im x(n+1)) \\ \quad b \leftarrow (\Re y(n), \Im y(n), \Re y(n+1), \Im y(n+1)) \\ \quad c \leftarrow a \odot b, \quad d \leftarrow a \odot b_{(1,0,3,2)} \\ \quad e \leftarrow (c,d)_{(0,4,2,6)}, \quad f \leftarrow (c,d)_{(1,5,3,7)} \odot (-1,1,-1,1) \\ \quad (\Re z(n), \Im z(n), \Re z(n+1), \Im z(n+1)) \leftarrow e \oplus f\end{array} \tag{13.38}$$

This scheme in principle needs two vector multiplications and one vector addition for $2+2$ complex numbers, whereas the sequential version needs four multiplications and two additions, or, more precisely, one addition and one subtraction for $1+1$ complex numbers, which seems perfect. However, there is an additional multiplication with $(-1,1,-1,1)$ that is necessary for the sign change in the vectorized addition, and there are 3 shuffle operations. Moreover, the two shuffles in line 4 need two instructions on Intel SSE, which makes a total of five shuffles. As a consequence, the speedup we get if we implement a sequence of complex multiplications in this way is actually a slowdown of about 0.7. This is a bad thing to start with when trying to vectorize an algorithm that is based on complex numbers.

On the other hand, the data layout with separate arrays for real and imaginary parts implies a vectorized algorithm that is equivalent to the sequential algorithm:

$$\begin{array}{l}\Re z(n,\ldots,n+3) = \Re x(n,\ldots) \odot \Re y(n,\ldots) \ominus \Im x(n,\ldots) \odot \Im y(n,\ldots) \\ \Im z(n,\ldots,n+3) = \Re x(n,\ldots) \odot \Im y(n,\ldots) \oplus \Im x(n,\ldots) \odot \Re y(n,\ldots)\end{array} \tag{13.39}$$

It uses 4 vector multiplications and 2 vector additions for $4+4$ complex numbers, which is perfect, and there are no shuffle operations at all. As a consequence, we get a speedup of about 3.7 for a sequence of multiplications.

However, the data layout might be predetermined by existing software or interface definitions. In this case, data could be rearranged after reading from memory and before writing to memory. This can be done by one shuffle operation per input and output vector. Intermediate stages of the algorithm can keep the separated data organization, though.

This rearrangement can be incorporated into the bit-reverse sorting pass that is part of the beginning or end of the FFT algorithm. Bit-reverse sorting moves $x(m)$ to $y(n)$, where the binary representations of m and n satisfy

$$m = m_0 2^0 + \cdots + m_{B-1} 2^{B-1} = BR(n) := n_{B-1} 2^0 + \cdots + n_0 2^{B-1}, \tag{13.40}$$

hence the name. If we combine these movements with the separation of real and imaginary parts, the sorting algorithm almost does not change. Suppose the array $\tilde{x}$ holds the alternated parts of the complex x, i.e., $\tilde{x}(2n, 2n+1) = (\Re x(n), \Im x(n))$. If the data block size is at least 8, i.e., $0 \le n < N \le 8$, or, equivalently, $B \ge 3$, then the sorting plus separation can be vectorized by

$$\begin{array}{l}
\text{for all } n: \\
\quad a \leftarrow \tilde{x}(BR(n), \ldots, BR(n)+3), \\
\quad b \leftarrow \tilde{x}(BR(n+1), \ldots, BR(n+1)+3), \\
\quad c \leftarrow \tilde{x}(BR(n+2), \ldots, BR(n+2)+3), \\
\quad d \leftarrow \tilde{x}(BR(n+3), \ldots, BR(n+3)+3), \\
\quad e \leftarrow (a,b)_{(0,2,4,6)}, \quad f \leftarrow (a,b)_{(1,3,5,7)}, \\
\quad g \leftarrow (c,d)_{(0,2,4,6)}, \quad h \leftarrow (c,d)_{(1,3,5,7)}, \\
\quad \Re y(n, \ldots, n+3) \leftarrow (e,g)_{(0,4,1,5)}, \\
\quad \Im y(n, \ldots, n+3) \leftarrow (f,h)_{(0,4,1,5)}, \\
\quad \Re y(n+4, \ldots, n+7) \leftarrow (e,g)_{(2,6,3,7)}, \\
\quad \Im y(n+4, \ldots, n+7) \leftarrow (f,h)_{(2,6,3,7)},
\end{array} \tag{13.41}$$

where n is a multiple of $2p = 8$. This requires eight shuffles for four input vectors.

13.5.2 Basic FFT-Blocks

After bit-reverse sorting, the actual algorithm ensues with recursions such as that in Eq. (13.5). If the data size N in a recursion iteration is greater than 4, then the iteration consists of point-wise multiplication of half of the complex data by complex factors of the form $e^{-i\frac{2\pi}{N}n}$, followed by addition and subtraction with the other half of the data. Due to our data layout, this can be done easily by vectorized multiplications as in Eq. (13.39).

If the data consists of four complex values, then vector-local computations are necessary. The FFT of size $N = 4$, i.e., $y = \underset{N}{\mathscr{F}} x$ is written out sequentially as

$$\begin{array}{ll}
b(0) \leftarrow x(0) + x(1), & b(1) \leftarrow x(0) - x(1), \\
b(2) \leftarrow x(2) + x(3), & b(3) \leftarrow x(2) - x(3), \\
y(0) \leftarrow b(0) + b(2), & y(1) \leftarrow b(1) - ib(3), \\
y(2) \leftarrow b(0) - b(2), & y(3) \leftarrow b(1) + ib(3),
\end{array} \tag{13.42}$$

where x is assumed to be already bit-reverse sorted, i.e., $x(1)$ and $x(2)$ are swapped. This algorithm looks quite regular, but the imaginary factor $-i$ that accompanies $b(3)$ disturbs the regularity significantly. Nevertheless, a straight forward vectorization can be given by

$$\begin{aligned}
&\Re b \leftarrow \Re x \odot (1,-1,1,-1) \oplus \Re x_{(1,0,3,2)},\\
&\Im b \leftarrow \Im x \odot (1,-1,1,-1) \oplus \Im x_{(1,0,3,2)},\\
&\Re y \leftarrow \Re b_{(0,1,0,1)} \oplus (\Re b, \Im b)_{(2,7,2,7)} \odot (1,1,-1,-1),\\
&\Im y \leftarrow \Im b_{(0,1,0,1)} \oplus (\Im b, \Re b)_{(2,7,2,7)} \odot (1,-1,-1,1)\,.
\end{aligned} \tag{13.43}$$

We see that there are again vector multiplications for sign change. Note that the algorithm itself does not include any multiplications at all. There are six shuffle operations, whereof two require two instructions on Intel SSE. To get rid of the multiplications, we reschedule the operations so that additions and subtractions are separated, which is possible because there is always an equal number of positive and negative signs. This leads to the following algorithm:

$$\begin{aligned}
&a \leftarrow (\Re x, \Im x)_{(0,2,4,6)}, \quad b \leftarrow (\Re x, \Im x)_{(1,3,5,7)}, \quad c \leftarrow a \oplus b, \quad d \leftarrow a \ominus b,\\
&e \leftarrow (c,d)_{(0,2,4,6)}, \quad f \leftarrow (c,d)_{(1,3,7,5)}, \quad g \leftarrow e \oplus f, \quad h \leftarrow e \ominus f,\\
&\Re y \leftarrow (g,h)_{(0,2,4,6)}, \quad \Im y \leftarrow (g,h)_{(1,7,5,3)}\,.
\end{aligned} \tag{13.44}$$

There are still six shuffle operations, only one of which needs two instructions on Intel SSE. Surprisingly, this algorithm is about 20% slower than that in Eq. (13.43). The reason is probably increased dependency of vector instructions and, thus, worse schedulability. All this shows that code optimization is difficult due to architecture dependencies, but necessary nevertheless. This problem is addressed in the next section.

13.5.3 Automatic Tuning and Signal Processing Languages (SPL)

Because implementations of algorithms show different performance characteristics on different architectures, optimal implementations have to be found on each architecture separately. This not only requires implementation efforts on each architecture, but many implementations have to be tested on each architecture. As this is rarely done manually, implementations are likely to be suboptimal.

To solve this problem, automatic tuning systems have been developed [4, 5], an approach that is well known in matrix algebra [24–26]. The idea behind these systems is that the transform is represented by a matrix M, i.e., $y = Mx$, and this matrix can be factored into sparse matrices M_k as

$$M = M_1 M_2 \cdots M_m\,. \tag{13.45}$$

These matrices can be built from the following primitive matrices:

- the identity matrix $I_n = \mathrm{diag}(1,\ldots,1)$,
- the stride permutation matrix $L^{rs}_r = \delta(js+k, j+kr)$ of size $rs \times rs$, where $0 \le j < r$ and $0 \le k < s$, and
- the "twiddle"-matrix $T^{rs}_r = \mathrm{diag}(w^{0\cdot 0},\ldots,w^{0\cdot(r-1)},w^{1\cdot 0},\ldots,w^{(s-1)(r-1)})$, where $w = e^{-i\frac{2\pi}{rs}}$.

The primitive matrices can be combined by the following operations:

- matrix multiplication,
- direct sum $A \oplus B = \begin{pmatrix} A & \\ & B \end{pmatrix}$,
- Kronecker product $A \otimes B = \begin{pmatrix} A_{0,0}B & \cdots & A_{0,s-1}B \\ \vdots & \ddots & \vdots \\ A_{r-1,0}B & \cdots & A_{r-1,s-1}B \end{pmatrix}$, and
- recursion, i.e., the use of smaller matrices with the same definition.

Together, these matrices and operations form a framework of a SPL [6]. As an example, it is possible to define the Fourier transform of size 4 (DFT_4) in this language through the formula

$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathrm{DFT}_2) L_2^4 . \tag{13.46}$$

Such a formula does not only represent a way to construct the matrix of the transform, it also defines an algorithm by which the transform can be implemented. A recursively expanded formula can automatically be converted into an actual algorithm in some programming language by substituting the primitive matrices or simple combinations M_j of them by appropriate loops of arithmetic operations. Because the matrices M_j are supposed to be sparse, the resulting algorithm usually reduces the computational complexity. For the Fourier transform, the complexity reduction is from $O(N^2)$ to $O(N \log N)$.

If a formula such as Eq. (13.46) is defined with symbolic indices (e.g., $\mathrm{DFT}_{rs} = \ldots$), then the formula constitutes a rule that can be applied in the recursive expansion of formulas. Usually, the parameters of a rule allow for several possible instantiations (e.g., $rs = 2 \cdot 4$ or $4 \cdot 2$). Moreover, there can be several applicable rules. Thus, a vast space of algorithmic implementations of a certain transform can be generated automatically.

The goal of the automatic tuning system is to traverse this space, to measure the implementations' performances, and to choose the one implementation with the best performance. However, some heuristics are necessary since it is usually too expensive to include the entire space of implementations.

There are two vectorization approaches that can be derived from this automatic tuning technique. The first one is simply to generate blocks of straight line code (i.e., code without loops) out of formulas and rules, to vectorize these "codelets" as described in Sect. 13.2.2. This is the approach taken in [7–9].

Another approach is to use the rules to generate vectorized code. If the expanded formulas contain right-sided Kronecker products with I_p, where p is the vector size, then the algorithm is directly vectorizable. This is the approach taken in [27, 28]. Special care has to be taken about shuffle operations. The formulas should be chosen so that the permutation matrices produce only permutations that are implementable as single shuffle instructions at a given architecture [29].

The question arises whether the SPL approach can also be used for convolution type streaming algorithms. A problem here is that the data size is unbounded, which

would imply matrices of infinite size in the SPL formulation. To work around this problem, one could select a small number of consecutive iterations of the outer loop and apply the SPL approach to this block. To choose the vector size as the block size might be a good choice. The block algorithm is then iterated for consecutive blocks. This approach is taken in [30] for the LMS algorithm. A disadvantage is that the technique cannot automatically choose how the block iterations interact, i.e., what data is passed between iterations. An extension of SPL to infinite cyclic matrices would certainly be a general solution, but this is future work.

13.6 Mixed Algorithms

There are algorithms in signal processing that cannot be classified as either convolution or Fourier oriented. Frequently, Fourier transforms are used on blocks of streaming data. This is mostly combined with overlapped windowed blocks, i.e., window functions applied to blocks before the transform to reduce artifacts due to the lack of periodicity. The well-known short-time Fourier transform (STFT), including the Gabor transform, is the most prominent kind of such a transform in time-frequency analysis. Vectorization strategies here are basically the same as for Fourier-type transforms, as those are the main part of a STFT.

On the other hand, filter operations can be applied on blocks of data, where the handling of block borders is either zero-padded, periodic, or mirrored. Moreover, filters can be applied in several phases, which includes recursive splitting of frequency bands, as in the wavelet transform, or multi-dimensional filtering. In these cases, the passing of vector data between phases might be optimized for overall performance. Therefore, we will examine a representative example more closely.

13.6.1 Recursive Convolution – Wavelet Transforms

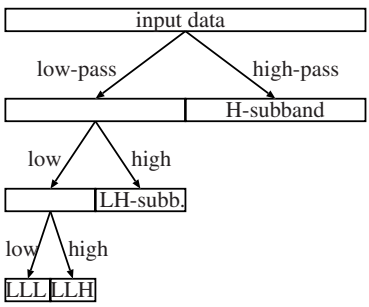


Fig. 13.15 Wavelet transform.

The wavelet transform is implemented by filter pairs such as those in Sects. 13.3.2, 13.3.3, and 13.3.4. We get a low-pass and a high-pass subband with half the size of the original data each. The low-pass subband is then filtered further to be substituted by two subbands of a quarter of the size of the original data, and so on. See Fig. 13.15.

Note that the original definition of the lifting scheme in Eq. (13.26) yields an interleaved data layout of the output data. This means that the input data of further passes is non-contiguous, which is very bad for vectorization. Fortunately, the approach with fused loops in Eq. (13.28) can separate the subbands easily, which is also true for the vectorized algorithm in Eq. (13.29).

Thus, the whole algorithm consists of several passes, where each one reads the output of the preceding pass. This is subject to cache issues, even more so with SIMD acceleration because the cache is more likely to be a bottleneck in faster algorithms. Therefore, the loop fusion technique can also be applied to all passes of the wavelet transform.

Note that special care has to be taken of block borders. See Fig. 13.8 for the case of mirrored border handling. The prolog and epilog phases in this algorithm appear in every pass of the wavelet transform. Therefore, the loop fusion has to incorporate these phases plus a certain number of main-phase iterations into big prolog and epilog phases, which can be arduous to hand-code.

13.6.2 Multi-dimensional Algorithms

The multi-dimensional Fourier transform is implemented in separate passes for each dimension. If the dimension of a certain pass accesses non-contiguous data, i.e., all passes but the first, then there is an easy method for vectorization. One simply has to perform the sequential algorithm while operating on vectors of several neighboring data values, thus transforming several columns at once. This approach can also be applied in the first dimension by transposing $p \times p$ blocks of input and output data after reading and before writing to memory, respectively, thus transforming p rows of data at once. See Eq. (13.9) for the vectorized transposition of such blocks.

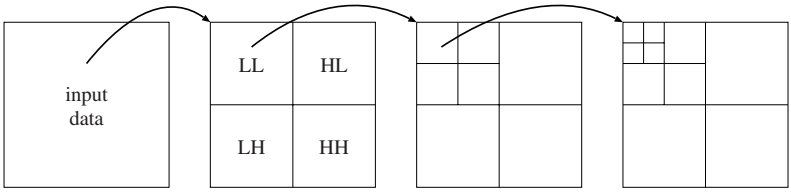


Fig. 13.16 2-D wavelet transform.

The same is true for the wavelet transform [10, 11]. Let us examine the 2-D wavelet transform. Here, each line is filtered by this scheme followed by columns

being processed in the same way, giving four subbands denoted by LL, LH, HL, HH. See Fig. 13.16. As explained before, we choose a data layout with separated subbands. This has the advantage that further passes can access the subbands in the same way and the same algorithm can be used. Otherwise, methods for the transform as a whole would have to be developed [31].

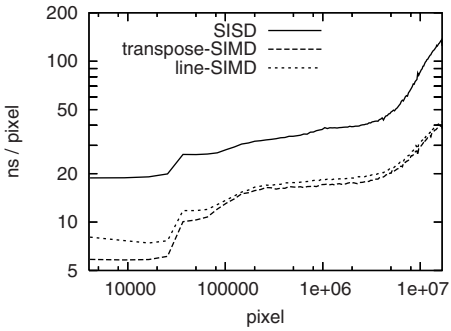


Fig. 13.17 Execution times per sample point (pixel) for one separate horizontal and vertical wavelet filtering pass, with and without SIMD.

See Fig. 13.17 for the execution times of a 2-D filtering pass. There is one horizontal and one vertical filtering step. The two vectorization approaches "line-SIMD," i.e., using the algorithm of Sect. 13.3.4.2 for horizontal filtering, and "transpose-SIMD," i.e., using the above transposition approach, are compared to the sequential "SISD" algorithm. We see that there is a performance gain by a factor of about 2.8 over the whole range of data sizes. The transposition-based parallelization is slightly better than the pure horizontal approach, mainly due to the lesser total number of shuffle operations.

We also see that there is a dependency on cached data and the algorithm does not scale linearly with the data size. To reduce cache dependencies, we will now fuse the horizontal and vertical pass [32]. In the 1-D case, we pass four values from one iteration to the other. To do a similar thing in the second dimension, we apply an approach that is known as pipeline or line-based computation [33]. If we imagine a whole row as a single value (as in the easy vertical SIMD algorithm, only with vectors of the size of a whole row), we must pass four such rows from one iteration to the other. This amounts to a buffer of four rows. In the 1-D case, we read two values from memory in a single iteration. In our row-wise approach this means that we need two new rows to start an iteration.

Since the source data for this row-wise vertical filtering is the output of the horizontal filtering, we try to use the output of the horizontal filtering in the vertical transform immediately after it is available. Thus, we have to perform two horizontal filterings (on two consecutive rows) at once. For each row we get a low-pass and a high-pass coefficient, which makes four values in total. The two low-pass values are fed into an iteration of the vertical type which produces an LL- and an LH-type coefficient, followed by the same operation on the two high-pass coefficients which produces an HL- and an HH-type coefficient. In each iteration the vertical part up-

dates four values in the four-row buffer, which are reused when the next two rows are processed.

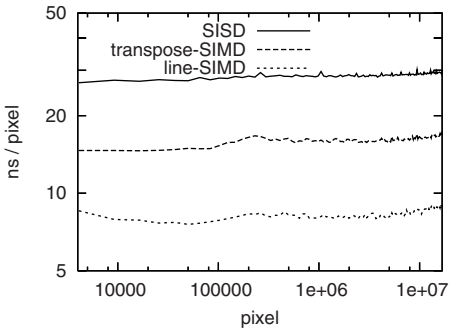


Fig. 13.18 Execution times per sample point (pixel) for the single-loop implementation with and without SIMD.

This algorithm can be vectorized without major problems, so we get a SIMD implementation of a 2-D wavelet filtering step in a single loop. The execution times are shown in Fig. 13.18. There is no cache dependency any more. This time the transposition based algorithm is significantly worse than the pure line-SIMD approach. The reason for this is increased buffer size destroying data locality, and an increased number of concurrently processed intermediate vectors per iteration making register allocation more difficult. The line-SIMD algorithm, however, performs about 3.7 times faster than the non-parallelized, which is very close to the theoretical maximum of 4.

13.7 Conclusion

Short-vector single-instruction-multiple-data (SIMD) processing is an interesting choice for parallel signal processing. The regularity of the data flow of algorithms used in signal processing enables manual and automatic vectorization techniques to efficiently exploit fine-grained parallelity for code acceleration.

The task of vectorization, however, is difficult. The reason is that there is no serve-all approach, but each algorithm has to be treated separately. This is even true if only characteristics like filter length or symmetry are changed for an otherwise simple filtering algorithm. However, most successful vectorization attempts are based on well-known strategies such as loop unrolling, loop fusion, loop transposition, and algebraic transforms. Even hard cases such as recursive filters can be parallelized efficiently in this way.

Whereas there are no general automatic vectorization systems for convolution type filtering algorithms, and manual strategies seem to be the only way to go, the space of possible implementations for Fourier-type algorithms is so large that automatic performance tuning systems that traverse this space to find the fastest im-

plementation cannot be beat by manual implementations, at least not in the general case.

However, the approaches presented in this chapter together with automatic performance tuning techniques may spawn efficient automatic vectorization systems for a broader range of signal processing algorithms in the future. A promising way to go might be the extension of SPL, as used in block transforms, to streaming data, as processed in filter banks.

References

1. J. W. Cooley, J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, Mathematics of Computation 19 (1965) 297–301.
2. P. Duhamel, M. Vetterli, Fast Fourier transforms: A tutorial review and a state of the art, Signal Processing 19 (4) (1990) 259–299.
3. C. M. Rader, Discrete Fourier transforms when the number of data samples is prime, in: Proc. of the IEEE, Vol. 56 (1968), pp. 1107–1108.
4. M. Frigo, S. G. Johnson, FFTW: An adaptive software architecture for the FFT, in: Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP), Vol. 3 (1998), pp. 1381–1384.
5. M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, R. W. Johnson, SPIRAL: A generator for platform-adapted libraries of signal processing algorithms, High Performance Computing and Applications (2004) 21–45.
6. J. Xiong, J. Johnson, R. Johnson, D. Padua, SPL: A language and compiler for DSP algorithms, in: Proc. Programming Language Design and Implementation (PLDI), ACM (2001), pp. 298–308.
7. S. Kral, F. Franchetti, J. Lorenz, C. W. Überhuber, SIMD vectorization techniques for straight line code, Tech. Rep. TR2003-02, Institute of Applied Mathematics and Numerical Analysis, Vienna University of Technology (2003).
8. S. Kral, F. Franchetti, J. Lorenz, C. W. Überhuber, SIMD vectorization of straight line FFT code, in: Proc. Euro-Par (2003), pp. 251–260.
9. M. Frigo, S. G. Johnson, The design and implementation of FFTW3, in: Proc. IEEE, Vol. 93 (2005), pp. 216–231.
10. C. Tenllado, D. Chaver, L. Piñuel, M. Prieto, 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 (2003).
11. D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, 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 (2002).
12. M. Pic, H. Essafi, D. Juvin, Wavelet transform on parallel SIMD architectures, in: F. Huck, R. Juday (Eds.), Visual Information Processing II, Vol. 1961 of SPIE Proceedings, SPIE (1993) pp. 316–323.
13. C. Chakrabarti, 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) (1995) 759–771.
14. M. Feil, 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 (1998), 4 pages.
15. R. Kutil, P. Eder, M. Watzl, SIMD parallelization of common wavelet filters, in: Parallel Numerics '05, Portorož, Slovenia (2005), pp. 141–149.

16. R. Kutil, P. Eder, Parallelization of wavelet filters using SIMD extensions, Parallel Processing Letters 16 (3) (2006) 335–349.
17. ISO/IEC 15444-1, Information technology – JPEG2000 image coding system, Part 1: Core coding system (Dec. 2000).
18. I. Daubechies, W. Sweldens, Factoring wavelet transforms into lifting steps, Journal of Fourier Analysis Applications 4 (3) (1998) 245–267.
19. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo, SPIRAL: Code generation for DSP transforms, Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93 (2) (2005) 232–275.
20. R. Schaffer, M. Hosemann, R. Merker, G. Fettweis, Recursive filtering on SIMD architectures, in: Proc. IEEE Workshop on Signal Processing Systems (SIPS), 2003, pp. 263–268.
21. M. Hosemann, G. Fettweis, On enhancing SIMD-controlled dsps for performing recursive filtering, Journal of VLSI signal processing 43 (2–3) (2006) 125–142.
22. J. Robelly, G. Cichon, H. Seidel, G. Fettweis, Implementation of recursive digital filters into vector SIMD DSParchitectures, in: Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 5 (2004), pp. 165–168.
23. R. Kutil, Parallelization of IIR filters using SIMD extensions, in: Proceedings of the 15th International Conference on Systems, Signals and Image Processing (IWSSIP), Bratislava, Slovak Republic (2008), pp. 65–68.
24. R. C. Whaley, J. Dongarra, Automatically tuned linear algebra software (ATLAS), in: Proc. Supercomputing (1998).
25. J. Bilmes, K. Asanović, C. W. Chin, J. Demmel, Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodoly, in: Proc. Int. Conf. Supercomputing (ICS) (1997), pp. 340–347.
26. E.-J. Im, K. Yelick, Optimizing sparse matrix computations for register reuse in SPARSITY, in: Proc. Int. Conf. Computational Sciences (ICCS) (2001), pp. 127–136.
27. F. Franchetti, M. Püschel, Short vector code generation for the discrete Fourier transform, in: Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2003), pp. 58–67.
28. F. Franchetti, M. Püschel, Short vector code generation and adaption for DSP algorithms, in: Proc. International Conference on Acoutstics, Speech and Signal Processing (ICASSP), Vol. 2 (2003), pp. 537–540.
29. F. Franchetti, M. Püschel, Generating SIMD vectorized permutations, in: Proc. Compiler Construction (CC) (2008), pp. 116–131.
30. J. Robelly, G. Cichon, H. Seidel, G. Fettweis, Design and automatic code generation of the LMS algorithm for SIMD signal processors, in: Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 5 (2005), pp. 81–84.
31. G. Lafruit, B. Vanhoof, L. Nachtergaele, F. Catthoor, J. Bormans, The local wavelet transform: a memory-efficient, high-speed architecture optimized to a region-oriented zero-tree coder, Integrated Computer-Aided Engineering 7 (2) (2000) 89–103.
32. R. Kutil, A single-loop approach to SIMD parallelization of 2-D wavelet lifting, in: Proceedings of the 14th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP), Montbeliard-Sochaux, France (2006), pp. 413–420.
33. C. Chrysafis, A. Ortega, Line based, reduced memory, wavelet image compression, IEEE Transactions on Image Processing 9 (3) (2000) 378–389.