IMPLEMENTATION OF THE WAVELET TRANSFORM WITH SSE EXTENSIONS

Tadeusz Łyszkowski[1], Tomasz Wiechno[2], Mykhaylo Yatsymirskyy[2]

[1]Higher Vocational State School in Wloclawek
tadeusz.lyszkowski@pwsz.wloclawek.pl

[2]Institute of Information Technology, Lodz University of Technology
tomasz.wiechn@p.lodz.pl, mykhaylo.yatsymirskyy@p.lodz.pl

Abstract

It has been shown that application of assembly implementation of Streaming SIMD Extensions (SSE) shortens the time needed to apply filtration in two-channel filter bank by tenfold, comparing to non-optimized version, written in Microsoft Visual C++ 2010 Express, without assembler extensions.

The implementation described in this paper can be applied to computation of Discrete Wavelet Transform on general-purpose processors..

Key words: Orthogonal Filters, Discrete Wavelet Transform, SSE extensions

1 Introduction

Discrete Wavelet Transform (DWT) is applied to data compression, system identification, signal approximation and interpolation, image processing and recognition as well as synthesis of digital watermarking [1-4].

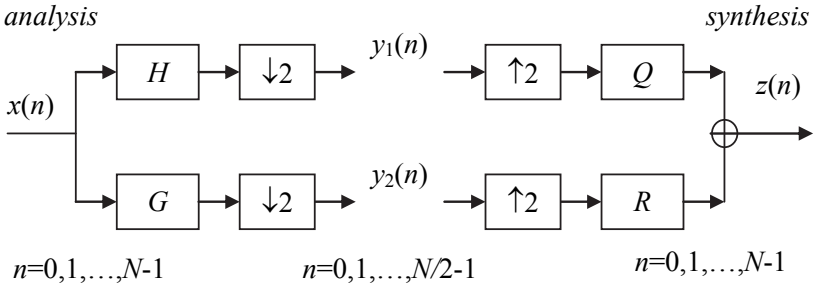

Figure 1. Diagram of one stage of analysis and synthesis of Discrete Wavelet Transform.

Because of such wide and profound applications, there is a lot of research on the improvements of Fast Computational Algorithms for the Discrete Wavelet Transform [5-10]. The construction of the algorithm is based on parallel or pyramidal repetition of basic analysis stage for forward transform and a basic synthesis stage for inverse transform. The two channel biorthogonal filter banks shown on Figure 1. [11] are a classic model of such a transform.

Blocks H, G, Q and R, are linear filters with finite impulse response $H = h_0,h_1,...,h_{K-1}$, $G = g_0,g_1,...,g_{K-1}$, $Q = q_0,q_1,...,q_{K-1}$ i $R = r_0,r_1,...,r_{K-1}$, where the length of the filter K is an even natural number. Blocks ↓2 and ↑2 denote, respectively, the operations of decimation in time of input sequence (downsampling) and upsampling by a factor of 2, i.e. inserting zeroes between each sample of a input sequence.The results of analysis stage of (forward) DWT can be expressed as two convolutions with decimation [12]

$$\begin{aligned} y_{2i} &= \sum_{k=0}^{K-1} h_{K,K-1-k}\, x_{2i+k}, \\ y_{2i+1} &= \sum_{k=0}^{K-1} g_{K,K-1-k}\, x_{2i+k} \qquad\qquad i = 0,1,...,N/2\text{-}1, \end{aligned} \tag{1}$$

where $h_{K,k}$,$g_{K,k}$ for $k = 0,1,...,K\text{-}1$ are impulse responses of filters H_K, G_K, and N is the length of input sequence.

If coefficients of impulse responses of filters H_K and G_K are written in reversed order:

$$h1_{K,k} = h_{K,K\text{-}1\text{-}k}, \quad g1_{K,k} = g_{K,K\text{-}1\text{-}k} \quad k = 0,1,...,K\text{-}1$$

formulas (1) can be rewritten in the form (2) that is more convenient for implementation

$$\begin{aligned} y_{2i} &= \sum_{k=0}^{K-1} h1_{K,k}\, x_{2i+k}, \\ y_{2i+1} &= \sum_{k=0}^{K-1} g1_{K,k}\, x_{2i+k} \qquad \text{for } i = 0,1,...,N/2\text{-}1. \end{aligned} \tag{2}$$

From (2) it is clear that the time needed for computation of DWT expressed as a convolution, depends on the effectiveness of floating point multiplications and additions. Exploiting Data Level Parallelism this can be enhanced by the usage of Streaming SIMD Extensions (SSE) available on contemporary general-purpose processors.

The paper describes construction of assembler implementations of DWT algorithms (2) that make use of SSE. The algorithms are given for a number of filter lengths K = 6,8,10 and 12 and the results are compared with the reference algorithms written in pure C++.

The problem solved in the paper is important as the majority of personal computers in use, is equipped with processors that are compliant with SSE rather than newer AVX extensions, introduced in 2011 [13].

2 SSE in IA-32 architecture

Beginning from the Pentium III processor the Streaming SIMD Extensions (SSE) were introduced to the IA-32 architecture. The SSE expands the SIMD execution model introduced with the Intel (Multimedia Extension) MMX technology by providing a new set of eight 128-bit registers *xmm*0, *xmm*1 ... *xmm*7 and the ability to perform (single-instruction, multiple-data) SIMD operations on four 32-bit packed single-precision floating-point values [13].The same operation can be performed at the same instruction cycle on four *float* elements stored in *xmm* register or in four array elements kept in memory.

Because of this parallelism in data processing, application of SSE Extensions can yield even fourfold performance gain comparing to a code that is non SSE aware. It is worth noting that data level parallelism reduces up to four times the number of instructions needed to write the algorithm.

3 Implementation of one stage of forward DWT computed as a convolution

Figure 2 shows the reference C++ implementation of DWT written according to the formula (2).

```
// DWT in C++
for (int i=0;i<N;i+=2)
{
     float t1=h1[0]*x[i], t2=g1[0]*x[i];
     for (int k=1;k<K;k++)
     {
       t1+=h1[k]*x[i+k];  t2+=g1[k]*x[i+k];
     }
     y[i]=t1;  y[i+1]=t2;
}
```

Figure 2. Algorithm of the one stage of forward DWT computed as a convolution in C++.

The algorithm needs *K* floating-point multiplications and *K-1* floating-point additions to compute one output element. However because of the data level parallelism it is possible to significantly shorten the time of this computation by the application of SSE extensions.To maximize performance gain, the whole algorithm has been programmed in assembly language.Furthermore, the inner loop that computes the sum of the product of input values times coefficients of impulse response (in reversed order), has been unfolded and optimized for the selected filter lengths, to shorten the most computation intensive part. The outer loop that contains mainly instructions for reading samples and writing output coefficients has been left intact.

Hence, further discussion in this section will concern major parts of the two assembler implementations of forward DWT for *N* being divisible by 4, namely: version A, for filers of length *K*=6 and 8, version B, for *K*=10 and 12 as well as some elements of version C, for *N* being even and *K*=6.

3.1 Version A. Implementation of DWT using assembler with SSE extensions

The implementation of this version, for filter length *K*=8 is shown on Figure 3. For the sake of clarity and speed of computation it has been assumed that the number of input samples *N* is divisible by 4. It is not really a constraint as, in majority of DWT applications, the length of input sequence is power of 2 with the exponent greater than 1. However, this makes it possible to compute and keep four output coefficients in *xmm* register as well as store them into the memory on every iteration of the loop.

In the discussed implementation there are eight steps. The first step shown on part a) of Figure 3. loads four input samples x_3, x_2, x_1, x_0 into the register *xmm0* and next four samples x_7, x_6, x_5, x_4 into the register *xmm1*. It is illustrated by the comments to the code, where four parts of the relevant register are shown for every instruction. In part b) registers *xmm*4, *xmm*5, *xmm*6 i *xmm*7 are loaded with coefficients of impulse responses *h1* and *g1* in reversed order.

```
mov ecx,0           ;(i=0)ecx=0
movaps xmm0,x[ecx]  ; xmm0=x3|x2|x1|x0
movaps xmm1,x[ecx+16]    ; xmm1=x7|x6|x5|x4
movaps buf1,xmm1    ; buf1=x7|x6|x5|x4
```

– Loading input data to the *xmm* registers

```
movaps xmm4,h1      ; xmm4=h13|h12|h11|h10
movaps xmm5,h1[16]  ; xmm5=h17|h16|h15|h14
movaps xmm6,g1      ; xmm6=g13|g12|g11|g10
```

```
  movaps xmm7,g1[16]  ; xmm7=g1_7|g1_6|g1_5|g1_4
```

– Loading parameters h1 and g1 to the xmm registers

```
iloop:
movaps xmm2,xmm0    ; xmm2=x_i+3|x_i+2|x_i+1|x_i+0
mulps xmm2,xmm4     ; xmm2=x_i+3*h1_3|x_i+2*h1_2|
                    ; x_i+1*h1_1|x_i+0*h1_0
movaps xmm3,xmm1    ; xmm3=x_i+7|x_i+6|x_i+5|x_i+4
mulps xmm3,xmm5     ; xmm3=x_i+7*h1_7|x_i+6*h1_6|
                    ; x_i+5*h1_5|x_i+4*h1_4
addps xmm2,xmm3     ; xmm2=x_i+7*h1_7+x_i+3*h1_3|
                    ; x_i+6*h1_6+x_i+2*h1_2|
                    ;x_i+5*h1_5+x_i+1*h1_1|x_i+4*h1_4+x_i+0*h1_0
movaps xmm3,zeros   ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3    ; xmm2=0.0|0.0|
                    ;x_i+7*h1_7+x_i+3*h1_3+x_i+6*h1_6+x_i+2*h1_2|
                    ; x_i+5*h1_5+x_i+1*h1_1+x_i+4*h1_4+x_i+0*h1_0
haddps xmm2,xmm3    ; xmm2=0.0|0.0|0.0|
                    ; x_i+7*h1_7+x_i+6*h1_6+x_i+5*h1_5+x_i+4*h1_4+
                    ; x_i+3*h1_3+x_i+2*h1_2+x_i+1*h1_1+x_i+0*h1_0
movaps buf5,xmm2    ; buf5=0.0|0.0|0.0|t1
```

– Computation of coefficient t1 of DWT (at that moment it is yi+0)

```
movaps xmm2,xmm0      ; xmm2=x_i+3|x_i+2|x_i+1|x_i+0
mulps xmm2,xmm6       ; xmm2=x_i+3*g1_3|x_i+2*g1_2|
                      ; x_i+1*g1_1|x_i+0*g1_0
movaps xmm3,xmm1      ; xmm3=x_i+7|x_i+6|x_i+5|x_i+4
mulps xmm3,xmm7       ; xmm3=x_i+7*g1_7|x_i+6*g1_6|
                      ; x_i+5*g1_5|x_i+4*g1_4
addps xmm2,xmm3       ; xmm2=x_i+7*g1_7+xi_+3*g1_3|
                      ; x_i+6*g1_6+x_i+2*g1_2|
                      ; x_i+5*g1_5+x_i+1*g1_1|
                      ; x_i+4*g1_4+x_i+0*g1_0
movaps xmm3,zeros     ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3      ; xmm2=0.0|0.0|
                      ; x_i+7*g1_7+x_i+3*g1_3+
                      ; x_i+6*g1_6+x_i+2*g1_2|
                      ; x_i+5*g1_5+x_i+1*g1_1+
                      ; x_i+4*g1_4+x_i+0*g1_0
haddps xmm2,xmm3      ; xmm2=0.0|0.0|0.0|
                      ; x_i+7*g1_7+x_i+6*g1_6+
```

```
                        ; x_{i+5}*g1_5+x_{i+4}*g1_4+x_{i+3}*g1_3+
                        ; x_{i+2}*g1_2+x_{i+1}*g1_1+x_{i+0}*g1_0
shufps xmm2,xmm2,11110011b ; xmm2=0.0|0.0|t2|0.0
addps xmm2,buf5         ; xmm2=0.0|0.0|t2|t1
movaps buf5,xmm2        ; buf5=0.0|0.0|y_{i+1}|y_{i+0}
```

– Computation of coefficient t2 of DWT (at that moment it is yi+1)

```
movaps xmm2,x[ecx+32]       ; xmm2=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
movaps buf0,xmm2            ; buf0=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
shufps xmm0,xmm1,01001110b ; xmm0=x_{i+5}|x_{i+4}|x_{i+3}|x_{i+2}
shufps xmm1,xmm2,01001110b ; xmm1=x_{i+9}|x_{i+8}|x_{i+7}|x_{i+6}
```

– Loading new input data to the xmm registers

```
movaps xmm2,xmm0        ; xmm2=x_{i+5}|x_{i+4}|x_{i+3}|x_{i+2}
mulps xmm2,xmm4         ; xmm2=x_{i+5}*h1_3|x_{i+4}*h1_2|
                        ; x_{i+3}*h1_1|x_{i+2}*h1_0
movaps xmm3,xmm1        ; xmm3=x_{i+9}|x_{i+8}|x_{i+7}|x_{i+6}
mulps xmm3,xmm5         ; xmm3=x_{i+9}*h1_7|x_{i+8}*h1_6|
                        ; x_{i+7}*h1_5|x_{i+6}*h1_4
addps xmm2,xmm3         ; xmm2=x_{i+9}*h1_7+x_{i+5}*h1_3|
                        ; x_{i+8}*h1_6+x_{i+4}*h1_2|
                        ; x_{i+7}*h1_5+x_{i+3}*h1_1|
                        ; x_{i+6}*h1_4+x_{i+2}*h1_0
movaps xmm3,zeros       ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3        ; xmm2=0.0|0.0|
                        ; x_{i+9}*h1_7+x_{i+5}*h1_3+
                        ; x_{i+8}*h1_6+x_{i+4}*h1_2|
                        ; x_{i+7}*h1_5+x_{i+3}*h1_1+
                        ; x_{i+6}*h1_4+x_{i+2}*h1_0
haddps xmm3,xmm2        ; xmm3=0.0|
                        ; x_{i+9}*h1_7+x_{i+8}*h1_6+
                        ; x_{i+7}*h1_5+x_{i+6}*h1_4+
                        ; x_{i+5}*h1_3+x_{i+4}*h1_2+
                        ; x_{i+3}*h1_1+x_{i+2}*h1_0|
                        ; 0.0|0.0
movaps buf6,xmm3        ; buf6=0.0|t1|0.0|0.0
```

– Computation of coefficient t1 of DWT (at that moment it is yi+2)

```
movaps xmm2,xmm0        ; xmm2=x_{i+5}|x_{i+4}|x_{i+3}|x_{i+2}
```

```
mulps xmm2,xmm6        ; xmm2=x_{i+5}*g1_3|x_{i+4}*g1_2|x_{i+3}*g1_1|
                       ; x_{i+2}*g1_0
movaps xmm3,xmm1       ; xmm3=x_{i+9}|x_{i+8}|x_{i+7}|x_{i+6}
mulps xmm3,xmm7        ; xmm3=x_{i+9}*g1_7|x_{i+8}*g1_6|x_{i+7}*g1_5|
                       ; x_{i+6}*g1_4
addps xmm2,xmm3        ; xmm2=x_{i+9}*g1_7+x_{i+5}*g1_3|
                       ; x_{i+8}*g1_6+x_{i+4}*g1_2|
                       ; x_{i+7}*g1_5+x_{i+3}*g1_1|
                       ; x_{i+6}*g1_4+x_{i+2}*g1_0
movaps xmm3,zeros      ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3       ; xmm2=0.0|0.0|
                       ; x_{i+9}*g1_7+x_{i+5}*g1_3+
                       ; x_{i+8}*g1_6+x_{i+4}*g1_2|
                       ; x_{i+7}*g1_5+x_{i+3}*g1_1+
                       ; x_{i+6}*g1_4+x_{i+2}*g1_0
haddps xmm3,xmm2       ; xmm3=0.0|x_{i+9}*g1_7+x_{i+8}*g1_6+
                       ; x_{i+7}*g1_5+x_{i+6}*g1_4+
                       ; x_{i+5}*g1_3+x_{i+4}*g1_2+
                       ; x_{i+3}*g1_1+x_{i+2}*g1_0|0.0|0.0
shufps xmm3,xmm3,10000000b ; xmm3=t2|0.0|0.0|0.0
addps xmm3,buf6        ; xmm3=t2|t1|0.0|0.0
addps xmm3,buf5        ; xmm3=y_{i+3}|y_{i+2}|y_{i+1}|y_{i+0}
movapsy[ecx],xmm3      ; y[ecx]=y_{i+3}|y_{i+2}|y_{i+1}|y_{i+0}
```

– Computation of coefficient t2 of DWT (at that moment it is yi+3), assembling yi+3, yi+2, yi+1, yi+0, in xmm register and storing its content into the memory

```
add ecx,16             ;(i=i+4) i.e. ecx=ecx+16
movaps xmm0,buf1       ; xmm0=x_{i+7}|x_{i+6}|x_{i+5}|x_{i+4}
movaps xmm1,buf0       ; xmm1=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
movaps buf1,xmm1       ; buf1=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
cmpecx,NN              ; Test the end of loop
                       ; condition(ecx = NN),
                       ;where NN=(N/4)*16
jneiloop               ; Jump to the label iloop
                       ; mentioned in step c)
                       ; if  ecx ≠ NN
```

– updating xmm0 i xmm1 before the next iteration of the loop and exit from the loop.

Figure 3. Steps of computation of the coefficients y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} for i = 0,4,8,...,N-4 in Version A of the implementation of DWT using assembler with SSE extensions

After the initialization steps a) - b) the algorithm enters the loop shown in steps c) - h). The step c) shows how the output coefficient y_{i+0} is being computed and put into the least significant part of the *xmm* register. Two SSE multiplications and three SSE additions are performed in this phase that is equivalent to eight floating point multiplications and seven additions. The results are saved to appropriate parts of the *xmm* register.

The next phase of the loop, namely step d) is devoted to computation of output coefficient y_{i+1} and saving it in the subsequent, more significant part of the *xmm* register. After this step, the register contains coefficients y_{i+1}, y_{i+0}in its lower part and floating zeros in the upper part.

This step is almost identical to the preceding one, with the exception of replacing impulse response $h1$with $g1$.

In step e) registers *xmm*0 and *xmm*1 are loaded with input samples shifted by two positions, in relation to their previous content. Namely,*xmm0* contains samples x_5, x_4 ,x_3 ,x_2, and *xmm1*samples x_9, x_8, x_7, x_6. These data will be used to compute y_{i+3}, y_{i+2}.

Step f) show the details of computation of coefficient y_{i+2} and points out that its value is stored in the *xmm* register, next to already computed y_{i+1}, y_{i+0}. Again, this phase is very similar to step c), the only difference is the position in *xmm* register where the value of y_{i+2} is being saved.

Similarly, the step g) concerns computation of the forth coefficient y_{i+3}. It is stored in the most significant part of the register *xmm*. The final quadruplet y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} is saved from the *xmm* register into the memory.

The last phase of loop shown as step h) increments loop counter in *ecx* register by 16 (i.e. the size of *xmm* register in bytes) This value will be used to address input samples x and output coefficients y.

In the following lines of code, registers *xmm*0 and *xmm*1 are being prepared for computation of output coefficients y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} in the next iteration of the loop (with i incremented by 4).

The loop concludes with a test condition $ecx \neq NN$, where $NN=(N/4)*16$.

If this condition is met the code jumps to the label `iloop` discussed in step c), i.e. the beginning of the loop.

The code for case K=8 can be also used for K=6, provided the two most significant coefficients of reversed order impulse responses are set to zero, i.e. $h1_7=h1_6=g1_7=g1_6=0$. However, coefficients $h1_5 \ldots h1_0$ and $g1_5 \ldots g1_0$ need to be initialized with corrected values, appropriate for K=6.

3.2 Version B. Implementation of DWT using assembler with SSE extensions

The implementation for filter length K=12 is similar to Version A. Again it has been assumed that the number of input samples N is divisible by 4, which makes it possible to process, in one iteration of the loop, four output coefficient in the register *xmm*.

The implementation can be logically divided into 8 steps. At first, the registers *xmm0* are loaded with values x_3, x_2, x_1, x_0, *xmm1* with x_7, x_6, x_5, x_4 and *xmm2* with x_{11}, x_{10}, x_9, x_8. The reversed coefficients of impulse responses $h1_7$, $h1_6$, $h1_5$, $h1_4$and $h1_3$, $h1_2$, $h1_1$, $h1_0$ are sent to *xmm*5 and *xmm*4, while $g1_7$, $g1_6$, $g1_5$, $g1_4$and $g1_3$, $g1_2$, $g1_1$, $g1_0$ are sent to *xmm*7 and *xmm*6. However, because of the limited number of *xmm* registers the most significant parts of $h1_{11}$, $h1_{10}$, $h1_9$, $h1_8$ and $g1_{11}$, $g1_{10}$, $g1_9$, $g1_8$ will be fetched from memory.

Following the above initialization code there are six steps in the loop, as they were in version A.The first step shows how the output coefficient y_{i+0} is being computed and put into the least significant part of the *xmm* register. Three SSE multiplications and four SSE additions are performed in this phase which is equivalent to twelve floating point multiplications and eleven additions. The results are saved to appropriate parts of the *xmm* register.

The next phase of the loop is devoted to computation of output coefficient y_{i+1} and saving it into the subsequent, more significant part of the *xmm* register. After this step, the register contains coefficients y_{i+1}, y_{i+0}in its lower part and floating zeros in the upper part. Thatphase of the algorithm is almost identical to the preceding one, with the exception of replacing impulse response $h1$with $g1$.

In the next step registers *xmm*0, *xmm*1 and *xmm2* are loaded with input samples shifted by two positions, in relation to their previous content. Namely,*xmm0* contains samples x_5, x_4 ,x_3 ,x_2,, *xmm1*samples x_9, x_8, x_7, x_6, and*xmm*2 samples x_{13}, x_{12}, x_{11}, x_{10}. These data will be used to compute y_{i+3}, y_{i+2}.

In the subsequent step, coefficient y_{i+2} is being computed and stored in the *xmm* register, next to the already computed y_{i+1}, y_{i+0}. Again, this phase is very similar to the computation of y_{i+0}, the only difference is the position in *xmm* register where the value of y_{i+2} is being saved.

Similarly, the following step, concerns computation of the last coefficient y_{i+3}. It is stored in the most significant part of the register *xmm*. The final quadruplet y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} is transferred from the *xmm* register into the memory.

The last phase of loop increments loop counter in *ecx* register by 16 (i.e. the size of *xmm* register in bytes). This value will be used to address input samples x and output coefficients y.

In the following lines of code, registers *xmm* are being prepared for computation of output coefficients y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} in the next iteration of the

loop (with *i* incremented by 4). The loop concludes with a test condition $ecx \neq NN$, where $NN=(N/4)*16$.If this condition is met the code jumps to the label `iloop`, i.e. computation of y_{i+0}.

The code that computes DWT for K=12 can be also used for K=10, provided the most significant coefficients of reversed impulse responses are set to zero, i.e. $h1_{11}=h1_{10}=g1_{11}=g1_{10}=0$. However, coefficients $h1_9 \ldots h1_0$ and $g1_9 \ldots g1_0$ need to be initialized with corrected values, appropriate for K=10.

3.3 Version C. Implementation of DWT using assembler with SSE extensions

This version of DWT implementation assumes that K=6, but the number of input samples $N$is even. In that case, on every odd iteration of the loop, the computations are performed in the same way as in version A (see Figure 3. Step c), and the pair of output coefficients y_{i+1}, y_{i+0} is saved on the least significant positions od 128-bit long buffer *buf*. On every even iteration, a following pair of output coefficients is being computed and saved in memory, together with a preceding pair, as a quadruplet of properly ordered coefficients.

If N is not divisible by 4, the last save operation concerns only the last pair of y_{i+1}, y_{i+0}.The code for computation of output coefficients is identical in versions A and C of the algorithm, so is the time of computation for K=6.

4 Test environment

All DWT implementations presented in the paper were written as C++ inline assembly (with SSE extensions) and compiled with Microsoft Visual C++ 2010 Express Version 10.0.40219.1 SP1Rel. The compiled code was executed on MS Windows 7 Home Premium PC with Intel® Core™ i5 CPU 650 3.20GHz and 4GB of RAM on board. Further, to neglect impact of concurrent operations of the processor on the computation time, all tests were run pN times, and the minimum time of execution, obtained with 64-bit clock cycle counter (measuring the number of clock cycles of the very code responsible for the computation of DWT), has been taken as a actual result of the measurement.

5 Experimental results

In order to compare effectiveness of the proposed implementation of DWT using assembler with SSE extensions, it was compared against the reference program written in pure C++, for the selected lengths of filter K, and a few

lengths of a input sequence N being divisible by 4. The resulting measurements expressed in cycles are gathered in Table 1.

Table 1. Results of measurement for pN = 100 000 000

K	Implementation	*N*=64	*N*=256	*N*=1024	*N*=4096
6	C++	3 465	13 983	55 902	223 611
	Assembler with SSE (version A)	459	1 845	7 395	29 490
8	C++	4 479	18 246	72 927	291 693
	Assembler with SSE (version A)	459	1 845	7 389	29 862
10	C++	5 577	22 374	89 193	357 006
	Assembler with SSE (version B)	594	2 403	9 621	38 880
12	C++	6 579	26 628	106 965	425 748
	Assembler with SSE (version B)	597	2 424	9 657	38 616

As can be seen from the table above, for filters of length K=8 and all tested values of N, implementation of DWT in assembler, with SSE extensions is performed almost 10 times faster than pure C++ version. For K=12 the optimized code is almost 11 times faster. This speedup may be attributed to manually optimized assembler implementation with parallel processing of data using SSE extensions. As it was mentioned in the information about SSE in IA-32 architecture, this may shorten the time of computation up to four times.

Further reduction of execution time, results from unfolding the inner loop which is the most computationally intensive. The outer loop contains mainly instructions for reading samples x and writing output coefficients y.

Because the implementation for K=6 and 8 uses the same version (A) of the algorithm, execution time is almost identical in both cases. The same holds true for version B and K=10 and 12.

Regardless of the version of implementation and the value of K, the amount of time needed to compute DWT is proportional to the length of the input sequence (and number of iterations). It is a direct conclusion from the formula (2).

Eventually the version C, for even N, has been examined. The results are shown in Table 2.

Table 2. Results of measurement for pN = 100 000 000

Implementation	***N=64***	***N=256***	***N=1024***	***N=4096***
Assembler with SSE (version C)	465	1839	7389	29436

The execution times for version C are virtually identical to version A. As a matter of fact, it is an expected result as both implementations share the same code to compute output pairs of coefficients. Moreover, although the construction and analysis of version C is more complex than version A, the speed of version C remains the same. Therefore, it is sufficient to use version A for K=6 and 8, and version B for K=10 and 12 and exclude special implementations for N, that are even but not divisible by four.

6 Conclusions

The paper discusses a number of implementations of Discrete Wavelet Transform written as a formula (2). The experimental results show that manually optimized C++, with unfolded inner loop and inline assembly code with SSE extensions, is about 10 times more robust than reference program written in pure C++. What is more, the obtained speedup looks favorably, comparing to the results shown in [14] where the SSE enabled code was performed 6x faster than naïve, C++ implementation of the convolution algorithm.

Although it is possible to achieve even further speedup with the application of the thread level parallelism of contemporary multi-core processors, the necessary algorithms are considerably more complicated. Hence, the proposed solution that use only Data Level Parallelism with SSE extensions is an attractive alternative, available even on a simple one core processors.

Due to the lower complexity of versions A and B, they are recommended as effective templates for computation of DWT with application of SSE.

References

1. Zieliński T. P.,2009,*Cyfrowe przetwarzanie sygnałów. Od teorii do zastosowań*, WKŁ, Warszawa
2. Fleet P. J.: 2008, *Discrete wavelet transformations: An elementary Approach with applications*, John Wiley&Sons, New Jersey.
3. Strang G., Nguyen T., 1999,*Wavelets and filter banks*, Wellesley-Cambridge Press.
4. Lipiński P.,2011,*Watermarking software in practical applications*, Bulletin of Polish Academy of Sciences: Technical Sciences, Vol. 59, nr 1, pp. 21-25.

5. Cooklev T., 2006,*An efficient architecture for orthogonal wavelet transforms*, IEEE Signal processing letters, Vol. 13, nr 2, pp. 77-79.
6. Olkkonen J. T., Olkkonen H., 2007,*Discrete lattice wavelet transform*, IEEE Transactions on circuits and systems – II: Express briefs, Vol. 54, nr 1, pp. 71-75.
7. Daubechies I., Sweldens W., 1998,*Factoring Wavelet Transform into Lifting Steps*, The Journal of Fourier Analysis and Applications, Vol. 4, nr 3, pp. 245-267.
8. Denk T. C., Parhi K. K., 1997,*VLSI architectures for lattice structure based orthonormal discrete wavelet transforms*, IEEE Transactions on circuits and systems – II: Analog and digital signal processing, Vol. 44, nr 2, pp. 129-132.
9. Bernabe G., Garcia J. M., Gonzalez J., 2003,*Reducing 3D Wavelet Transform Execution Time through the Streaming SIMD Extensions*, IEEE Computer Society, Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 209-223
10. ShahbahramiA., Juurlink B., Vassiliadis S., 2008,*Implementing the 2-D Wavelet Transform on SIMD-Enhanced General-Purpose processors*, IEEE Transactions on multimedia, Vol. 10, nr 1, pp. 43-51.
11. Yatsymirskyy M., 2011,*Nowy model macierzowy dwukanałowego banku biortogonalnych filtrów*, Metody Informatyki Stosowanej, nr 1/2011 (26), Polska Akademia Nauk Oddział w Gdańsku, Komisja Informatyki, pp. 205-212.
12. Yatsymirskyy M.,Stokfiszewski K., 2012, Effectiveness of Lattice Factorization of Two-Channel Orthogonal Filter Banks, New Trends in Audio and Video/ Signal Processing Algorithms, Architectures, Arrangements and Applications, 27-29 September, Łódź, pp. 275-279.
13. *Intel® 64 and IA-32 Architectures. Software Developer's Manual, Volume 1: Basic Architecture*.
14. Gomersall H., 2012, *Speedy fast 1D convolution with SSE,* http://hgomersall.wordpress.com/2012/11/02/speedy-fast-1d-convolution-with-sse/