Vectorization of the 2D Wavelet Lifting Transform Using SIMD Extensions

D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, F. Tirado
Departamento de Arquitectura de Computadores y Automática
Facultad de C.C. Físicas. Universidad Complutense.
Ciudad Universitaria s/n 28040 Madrid. Spain.
{dani02,ctenllado,lpinuel,mpmatias,ptirado}@dacya.ucm.es

Abstract
This paper addresses the vectorization of the lifting-based wavelet transform on general-purpose microprocessors in the context of JPEG2000. Since SIMD exploitation strongly depends on an efficient memory hierarchy usage, this research is based on previous work about cache-conscious DWT implementations [1,2,3]. The experimental platform on which we have chosen to study the benefits of the SIMD extensions is an Intel Pentium-4 (P-4) based PC. However, unlike other authors [4], the vectorization has been performed avoiding assembler language programming in order to improve both code portability and development cost.

Index Terms— JPEG2000, lifting, SIMD optimization.

1. Introduction
A significant amount of work on the optimization of the lifting-based 2D discrete wavelet transform (DWT) has been performed in recent years in the context of the JPEG2000 [1,2,5]. This interest is caused by the considerable percentage of execution time involved in this component of the standard. According to some authors, it accounts for 40-60% [1,2,5] of the JPEG2000 encoding time.

From a performance point of view, one of the main bottlenecks of the DWT is caused by the discrepancies between the memory access patterns of the two principal components of the 2D DWT: the vertical and the horizontal filtering. These differences cause one of these components to exhibit poor data locality in the straightforward implementations of the algorithm. As a consequence, most of the previous work about DWT performance optimization has been focused on memory hierarchy exploitation.

A different strategy has been followed in [4]. In this case, the performance of the JPEG2000 DWT has been improved by means of fixed-point arithmetic and Intel’s MMX ISA (Instruction Set Architecture) extensions.

The aim of our research is to structure the lifting computations in order to take advantage of both the memory hierarchy and the SIMD parallelism. In fact, as we have shown in previous studies, an efficient exploitation of the SIMD ISA extensions available in modern microprocessors strongly depends on an efficient memory hierarchy usage [3,6].

The experimental platform on which we have chosen to study the benefits of the SIMD extensions is an Intel Pentium-4 (P-4) based PC. Despite using a specific platform we should remark that, unlike other studies [4,7], we have avoided coding at the assembly language level in order to improve portability (it also prevents long development times).

The rest of this paper is organized as follows. Some related work and the experimental environment are covered in Sections 2 and 3 respectively. Section 4 describes some details of our DWT implementations and discusses the performance results obtained without vectorization, which is analyzed in detail in Section 5. Finally, the paper ends with some conclusions.

2. Related Work
As mentioned above, the performance optimization of the DWT is not a new research issue. The optimization of both the convolution-based and the lifting-based DWTs has already been done for all sorts of computer systems. Focusing on the target of this paper, i.e. general-purpose microprocessors, several optimizations aimed at improving the cache performance have been proposed in [1,2,8].

Basically, [1] investigates the benefits of traditional loop-tiling techniques, while [2,8] investigate the use of specific array layouts as an additional mean of improving data locality.

The thesis of [8] is that row-major or column-major layouts (canonical layouts) are not advisable in many
applications, since they favor the processing of data in one direction over the other. For convolution-based DWT, they studied the benefits of two non-linear layouts, known in the literature as 4D and Morton [8]. In these layouts the original \( m \times n \) image is conceptually viewed as an \([m/\text{tr}] \times [n/\text{tc}]\) array of \( \text{tr} \times \text{tc} \) tiles. Within each tile, a canonical (row-major or column-major) layout is employed. In [2], this study has been extended with the analysis of the lifting-based DWT, although for this approach, a lightly different non-linear layout is employed (more details about it are explained in section 4.1).

The approach investigated in [1] is less aggressive. Nevertheless, they addressed the memory exploitation problem in the context of the whole JPEG2000 image coding application, which is more tedious to optimize than a wavelet kernel. The solution investigated by these authors, which they dubbed aggregation, is similar to the classical loop-tiling strategy that we have applied to the vertical filtering in [12] (the one that lacks spatial locality if a row-major layout is employed).

3. Experimental platform

Our experimental platform consists of a P-4 (2.4 GHz Model 2) machine running under Linux, the main features of which are described in the table below (see also [9]).

<table>
<thead>
<tr>
<th>Table 1. Pentium-4 system main features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>L1 Data Cache</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
</tr>
<tr>
<td>Motherboard</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Operating system</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ICC Compiler switches</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>GCC switches</td>
</tr>
</tbody>
</table>

All optimizations have been carried out at the source code level to avoid assembly language programming. In addition, in order to isolate compiler effects, we have employed two different compilers: the GNU GCC 3.2 [10] and the Intel C/C++ 6.0 (ICC) [11]. In both cases, we have used generic optimization switches.

Both compilers provide access to the P-4’s SIMD ISA extensions (known as SSE: Streaming SIMD Extensions) by means of the same set of intrinsic functions, which allows C/C++ style coding instead of assembly language [10,11]. Consequently, the same hand-tuned code is employed in both cases.

In addition to these intrinsics, which most of them map one-to-one to SSE instructions, the ICC provides an automatic vectorizer, which in our case is activated by means of the “–vec –restrict” switches. Nevertheless, for the programs under study, fully automatic vectorization is not possible, and both code modifications and guided-compilation are required.

4. Memory hierarchy exploitation

4.1 Implementation details

Two types of DWTs are considered in the JPEG2000: the lossless algorithm is based on an integer 5-tap/3-tap filter, whereas the lossy compression uses the popular Daubechies 9-tap/7-tap floating-point filter [4]. In this paper we have only covered the latter, although the proposed optimizations can also be applied to the reversible filter. In particular, our implementation uses single precision data to represent both the image elements and the wavelet coefficients. Nevertheless, we should also mention that, at the time of writing this paper, we are analyzing the potential benefits of using fixed-point arithmetic.

Due to the memory hierarchy bottleneck, an important design decision must be made regarding memory management. As is well known, the lifting scheme allows an inplace computation of the DWT [4], i.e. the transform can be calculated without allocating auxiliary memory (see figure 1). However, this memory saving is at the cost of scattering the wavelet coefficients throughout the original matrix, which in the context of the JPEG2000 involves a post-processing step where the coefficients are rearranged. This way, each sub-band is contiguously stored in memory, which simplifies the subsequent quantization stage [2].

In order to avoid this rearrangement overhead, we have also considered two additional strategies. The first one, which we have denoted by mallat, was proposed in [2]. It uses an auxiliary matrix to store the results of the horizontal filtering. In this way, as figure 2 shows, the horizontal high and low frequency components are not interleaved in memory. The vertical filtering reads these components and writes the results into the original matrix following the order expected by the quantization step. In order to improve data locality we have employed a recursive data layout [2] where each sub-band is laid out
contiguously in memory. As we will explain below, this approach also allows a better exploitation of the SIMD parallelism.

**Figure 1: Inplace strategy (logical view on the top; recursive data layout on the bottom).**

Furthermore, we have introduced an additional strategy, which we have denoted by inplace-mallat. It can be considered as a trade-off between the inplace and mallat alternatives. It performs the horizontal filtering inplace but uses an auxiliary matrix to store the final wavelet coefficients as soon as they are computed. In this way, at the end of the calculations, the transformed image is stored in the expected order, thus avoiding the post-processing stage. As above, a recursive data layout is employed in order to improve data locality. Figure 3 graphically describes this alternative. Only the low frequency components in each direction (denoted by LL) are stored in the original matrix (apart from the deepest decomposition level) whereas the other components (denoted by LH, HL and HH) are moved into the auxiliary matrix in their correct final positions.

The recursive data layout benefits the spatial locality of the memory access pattern. Nevertheless, further data locality improvements are possible by means of loop-tiling (aggregation in [1]). Supposing a column-major layout on every wavelet sub-band (the whole image for the inplace strategy), memory access becomes a bottleneck in the horizontal filtering. In order to reduce this overhead, instead of processing the image rows one after the other, which produces very low data locality, the horizontal filtering is applied column by column so that the spatial locality can be more effectively exploited.

**Figure 2: Mallat strategy (logical view on the top; recursive data layout on the bottom).**

4.2 Performance results

Figure 4 and 5 show the experimental results for the different strategies under study using different image sizes. The reported execution times correspond to the processing of a single color component, treating the entire image as a single tile.
Figure 4: Execution time breakdown for 256², 512² and 1024² images using both compilers. I, IM and M denote inplace, inplace-mallat, and mallat strategies respectively. Each bar shows the execution time of each level and the post-processing step (when required), denoted by Post.

As expected, the mallat and the inplace-mallat approaches outperform the inplace version for levels 2 and above. The reason for such behavior lies in the improvement of the data locality introduced by the recursive layout. On the other hand, we observe that these approaches also entail a noticeable slowdown for the first decomposition level due to both a larger working set (remember that they require an auxiliary matrix) and a more complex access pattern. However, this overhead is by far compensated in the inplace-mallat version, which achieves the best global execution time if we take into account the post-processing stage required in the inplace case. In contrast, this does not happen in the mallat approach, which exhibits the poorest performance in most cases.

Figure 5: Execution time breakdown for 2048², 4096² and 8192² images using both compilers.

Finally, focusing on compiler performance, we should note that the native ICC compiler outperforms GCC in the mallat and the inplace-mallat approaches. However, and contrary to our expectations, the opposite behavior is observed for the inplace code.

5. Vectorization

5.1 Semi-automatic vectorization

From a programmer’s point of view, the most suitable way to exploit SIMD extensions is automatic
vectorization, since it avoids low level coding techniques, which are platform dependent. Nevertheless, loops must fulfill some requirements in order to be automatically vectorized, and in most practical cases both code modifications and guided compilation are necessary.

In our experimental platform, this kind of vectorization is only possible using ICC [11]. In particular, this compiler can only vectorize simple loop structures. Primarily, only inner loops with simple array index manipulation (i.e. unit increment) and which iterate over contiguous memory locations are candidates. In addition, global variables must be avoided since they inhibit vectorization. Finally, if pointers are employed inside the loop, pointer disambiguation is mandatory (this must be done by hand using compiler directives).

Considering these restrictions and supposing column-major layouts, only the horizontal filtering can be automatically vectorized (see Algorithm 1). Furthermore, in the case of the *inplace* version, the vectorization is limited to the first decomposition level since data are interleaved above this level.

```
/* Column loop */
for(j=2,k=1;j<=(#columns-4);j+=2,k++)
{
    /* Vectorizable row loop. Every 4 rows from each column are operated in parallel */
    #pragma vector aligned
    for(i=0;i<#rows;i++)
    {
        /* 1st operation */
        colj+3[i]=colj+3[i]+ alfa*( colj+4[i]+ colj+2[i]);
        /* 2nd operation */
        colj+2[i]=colj+2[i]+ beta*( colj+3[i]+ colj+1[i]);
        /* 3rd operation */
        colj+1[i]=colj+1[i]+ gama*( colj+2[i]+ colj[i]);
        /* 4th operation */
        col[i]=col[i]+ delt*( colj+1[i]+ colj-1[i]);
        /* Last step */
        detail k[i]=(colj+1[i]= colj+1[i]*phi_inv);
        aprox k[i]=(col[i] = col[i]*phi);
    }
}
```

**Algorithm 1.** Automatically vectorizable horizontal filtering (*mallat*), assuming a column-major layout for the image. The variables colj, colj+1, colj+2, colj+3, colj-1 are local pointers to the Matrix1 columns j, j+1, j+2, j+3 and j-1 respectively, whereas detail k and aprox k are local pointers to the destination Matrix2 columns (see figure 1).

### 5.2 Hand-coded vectorization

Obviously, this approach involves more coding effort than the automatic case, since the SIMD parallelism has to be explicitly expressed.

Although intrinsics allow more flexibility, it is also convenient in this case to store the wavelet coefficients contiguously in memory. This way, they can be directly packed into vectorial registers. Under column-major layouts, this means that only the horizontal filtering can be effectively vectorized (as above, just on the first decomposition level for the *inplace* version). The vertical filtering could be vectorized now but at the expense of an additional data transposition stage [3], which reduces the benefits of the SIMD parallelism. Although we have considered this strategy in the optimization of the convolution-based DWT, this approach is not covered in this paper due to space limitations. The interested reader can find more information in [3].

![Figure 6. Vectorial computation of a single horizontal lifting stage. The white arrows indicate how the image is scanned.](image)

Figure 6 describes how vectorial computations are performed in one of the lifting stages. The image is scanned following the same order as in the scalar version but all the calculations are carried out in groups of four. The specific hand-tuned DWT algorithm is described in the Algorithm 2.

### 5.3 Performance results

Before analyzing the benefits on the whole DWT, it is convenient to isolate the improvements achieved by the vectorization on the horizontal filtering. Figure 7 compares the scalar and vectorial versions of this processing for the different strategies under study. For the sake of simplicity, only the results for a 1024² pixels image are considered. Nevertheless, similar behavior can be observed for the other images sizes.
Algorithm 2. Hand-Coded vectorized horizontal filtering (mallat), assuming a column-major layout for the image and using the Intel C/C++ Compiler intrinsic functions.

As can be noticed, the vectorization achieves a significant performance gain. The speedup ranges between 4 and 6 depending on the strategy. The reason for such a high improvement is due not only to the vectorial computations, but also to a considerable reduction in the memory accesses caused by the exploitation of the packed loads/stores provided by the SSE extensions, the reuse of the vectorial registers, etc.). This enhancement of the memory behavior also explains why the speedup is higher in the first decomposition level due to its larger working set.

Figure 7. Execution time breakdown of the horizontal filtering for all the codes under study using a 1024² pixels image. I, IM and M denote inplace, inplace-mallat and mallat approaches respectively. S, A and H denote scalar, automatic-vectorized and hand-coded-vectorized versions respectively.

We should also remark that the speedups achieved by the strategies with recursive layouts (i.e. inplace-mallat and mallat) are higher than the inplace version counterparts, since the computation on the latter can only be vectorized in the first level. Nevertheless, in the ICC versions, a small reduction in the execution time is observed in this case for the other levels due to the use of vectorial memory transfers (which are automatically introduced by the compiler when vectorization is enabled).

Focusing on Intel’s ICC, it is interesting to note that both vectorization approaches (i.e. automatic and hand-tuned) produce similar speedups. In fact, both versions generate almost the same assembly code, which highlights the quality of the ICC vectorizer.

\[\text{...}\]

\[\text{...}\]

\[\text{...}\]

\[\text{...}\]

\[\text{...}\]
Figure 8. Execution time breakdown for all the codes under study, using a 1024² pixels image. I, IM and M denote inplace, inplace-mallat and mallat approaches respectively. S, A and H denote scalar, automatic-vectorized and hand-coded-vectorized versions respectively.

Figure 8 shows the global DWT execution time for the same image size. The improvement achieved by the vectorization of the horizontal processing translates into an overall speedup of between 1.5 and 2. The shortest execution time is reached in this case by the ICC mallat version (when using GCC both recursive-layout strategies obtain similar results). This surprising behavior of the mallat approach (remember that it provides the worst results in the scalar version) is a consequence of its better performance on the vertical filtering. Unlike the scalar version, this is now the most costly DWT component and hence the disadvantages of the mallat horizontal filtering (see figure 7) are compensated (more details can be found in [9]).

Figure 9 compares the speedup of the different vectorial codes over the inplace-mallat approach (the best scalar version) and over the inplace approach. The speedup grows with the image size since, as mentioned above, the vectorization improvements are not only due to the vectorial computations but also to the memory usage. On average, the speedup is about 1.8 over the inplace-mallat scheme, growing to about 2 when considering it over the inplace strategy. Focusing now on the compilers, ICC clearly outperforms GCC by a significant 20-25% for all the image sizes.

Figure 9. Speedup achieved by the different vectorial codes over the inplace-mallat (top chart) and inplace (bottom chart) versions respectively.

Figure 10. Normalized execution time. Execution time to image size (secs/pixels) ratio for the hand-coded vectorial versions (ICC compiler).

Figure 10 compares the performance of the different approaches using as performance metric a normalized execution time (execution_time/image_size). This figure
emphasizes the superior behavior of the vectorial mallat scheme for the ICC since, as can be noticed, it exhibits the best performance scalability (i.e. the execution time per pixel remains almost constant with the image size).

6. Conclusions

In this paper we have studied the optimization of a JPEG2000-aware lifting-based DWT on modern general-purpose microprocessors. The main conclusions can be summarized as follows:

1. Focusing on the scalar version, a novel scheme based on recursive layouts has been introduced. This scheme, which we have denoted by inplace-mallat, outperforms both a cache-conscious inplace implementation and a recent approach proposed by Chatterjee et al. [2] (denoted by mallat).

2. Based on our previous studies on SIMD exploitation, we have proposed some code modifications that allow the vectorial processing of the lifting algorithm. Two different methodologies have been explored in the case of the ICC compiler: semi-automatic and intrinsic-based vectorizations. However, both of them provide similar results.

3. Exceeding our expectations, the speedup achieved in the horizontal filtering is about 4-6 (depending on the code) since vectorization also reduces the pressure on the memory system. This enhancement translates into a significant performance gain in the whole transform (around 2 on average). In addition, this gain increases with image size.

4. In contrast with the scalar version, the vectorial mallat approach outperforms the other schemes. Moreover, it exhibits a better scalability and its benefits grow with image size.

Finally, we should note that most of our insights are compiler independent, but additional analysis on other computing platforms will improve the generality of this study. In future research we also plan to integrate the proposed optimizations in a reference implementation of the JPEG2000 in order to improve the diffusion and understanding of our results and to facilitate further comparisons.

7. Acknowledgements

This work has been supported by the Spanish research grants TIC 99-0474 and TIC 2002-750. We would also like to thank the anonymous referees for their valuable comments.

8. References


