Vectorization of the 2D Wavelet Lifting Transform using SIMD extensions

C. Tenllado, D. Chaver, L. Piñuel, M. Prieto and F. Tirado

***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]. Furthermore, the vectorization has been performed avoiding assembler language programming in order to improve code portability.**

***Index Terms*—Wavelet Transform, Lifting, SIMD Extensions, JPEG2000.**

I. 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 accesses patterns of the two principal components of the 2D DWT: the vertical and the horizontal filtering. This difference causes 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 this 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 the 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 previous studies [4,7], we have avoided coding at the assembly language level in order to improve portability (it also prevent long development times).

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

Authors are with the Departamento de Arquitectura de Computadores y Automatica, Universidad Complutense, Madrid, Spain. E-mails: {ctenllado,dani02,lpinuel,mpmatias, ptirado}@dacya.ucm.es

II. Experimental Platform

Our experimental platform consists on a P-4 (2.4 GHz Model 2) machine running under Linux, the main features of which are described in [8].

In order to isolate compiler effects, we have employed two different compilers: the GNU GCC 3.2 [9] and the Intel C/C++ 6.0 (ICC) [10]. In both cases, we have used generic optimization switches (namely, "-O3 -msse" on the GCC and "-O3 -tpp7 -xW" on the ICC).

Both compilers provide access to the P-4's SIMD ISA extensions (known as SSE: Streaming SIMD Extensions) by means of a the same set of intrinsic functions, which allows C/C++ style coding instead of assembly language [9,10]. 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.

III. Memory Hierarchy Exploitation

A. 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. 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 of using fixed-point arithmetic.

Due to the memory hierarchy bottleneck, an important design decision is the memory management. As is well known, the lifting scheme allows an *inplace* computation of the DWT, i.e. the transform can be calculated without allocating auxiliary memory. 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 stored contiguous 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 as *mallat*, was proposed in [2]. It uses an auxiliary matrix to store the results of the horizontal filtering. In this way, as figure 1 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.

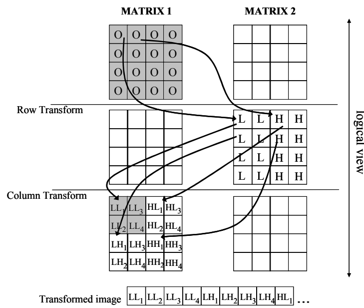

Fig. 1. *Mallat* strategy (logical view on the top; recursive data layout on the bottom).

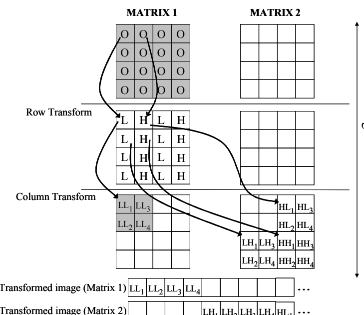

Fig. 2. *Inplace-mallat* (logical view on the top; recursive data layout on the bottom).

In this research we have introduced an additional strategy, which we have denoted as *inplace-mallat*, can be considered as a trade-off between the *inplace* and *mallat* strategies. 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 2 graphically describes this alternative. Only the low frequency components in each direction (denoted as LL) are stored into the original matrix (apart from the deepest decomposition level) whereas the other components (denoted as LH, HL and HH) are moved into the auxiliary matrix in their correct final positions.

As explained above, the recursive data layout benefits the spatial locality of the memory access pattern. However, further optimizations are possible. Supposing a column mayor 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, we have optimized this filtering by means of a loop-tiling technique [11] (denoted as aggregation in [1] and strip-mine in [2]). Thus, 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.

B. Performance Results

Figure 3 shows 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.

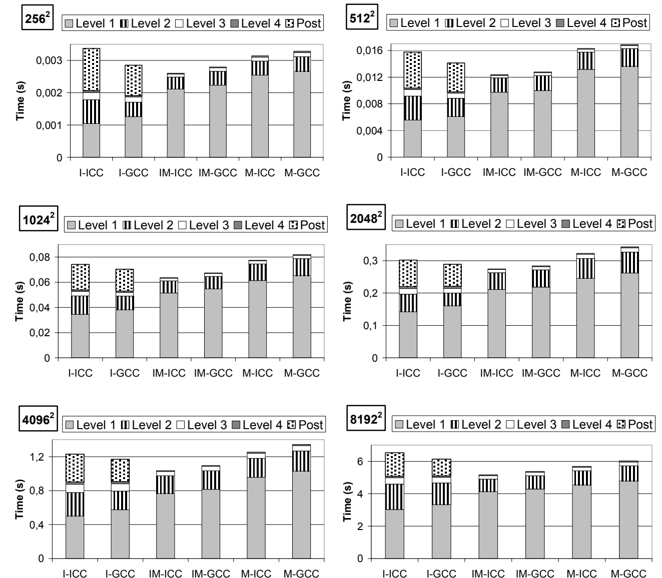

Fig. 3. Execution time breakdown for different image sizes using both compilers. I , IM and M denote *inplace*, *inplace-mallat*, and *mallat* strategies respectively. Each bar shows the execution time of each level and the post-processing step (when required), denoted as *Post*.

As expected, the *mallat* and the *inplace-mallat* approaches outperform the *inplace* version for levels 2 and above. The reason for such behavior lies on 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.

We should remark that these insights are compiler independent. Nevertheless, different conclusions could be obtained using other general-purpose microprocessors and as we will mention below, we plan to extend our performance comparison with other computing platforms in order to improve the generality of our analysis.

Finally, focusing on the compilers 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.

IV. Vectorization

A. 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 just possible using ICC [10]. 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. Furthermore, in the case of the *inplace* version, the vectorization is limited to the first decomposition level since data are interleaved above this level.

B. 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 allows 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 4 graphically 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 can be downloaded at http://www.dacya.ucm.es/atc/jpeg2000.

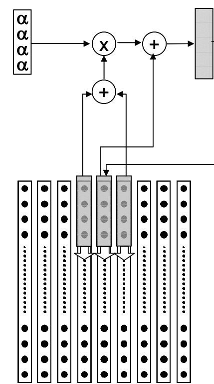

Fig. 4. Vectorial computation of a single horizontal lifting stage. The white arrows indicate how the image is scanned.

C. 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 5 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 1024^2 pixels image are considered. Nevertheless, similar behavior can be observed for the other images sizes.

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[1] (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.

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

[1] Memory accesses have been measured through the P4 performance monitoring counters. More details can be found in [8].

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).

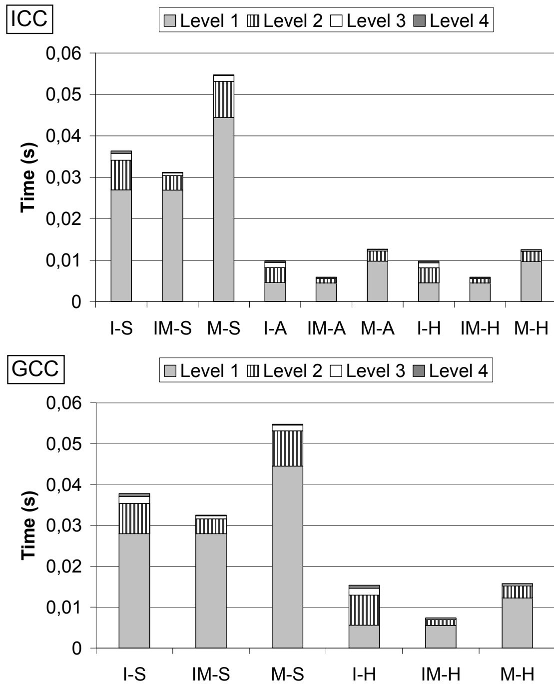

Fig. 5. Execution time breakdown of the horizontal filtering for all the codes under study using a 1024^2 pixels image. I, IM and M denote *inplace*, *inplace-mallat* and *mallat* approaches respectively. S, A and H denote *scalar*, *automatic-vectorized* and *hand-coded-vectorized* versions respectively.

Focusing on the 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.

Figure 6 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 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 worse 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 5) are compensated. (more details can be found in [8]).

Figure 7 compares the speedup of the different vectorial codes over the best scalar version (*inplace-mallat*) and the *inplace* approach respectively. 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 best scalar scheme, growing to about 2 for the *inplace* strategy. Focusing now on both compilers, ICC clearly outperforms GCC by a significant 20-25% for all the image sizes.

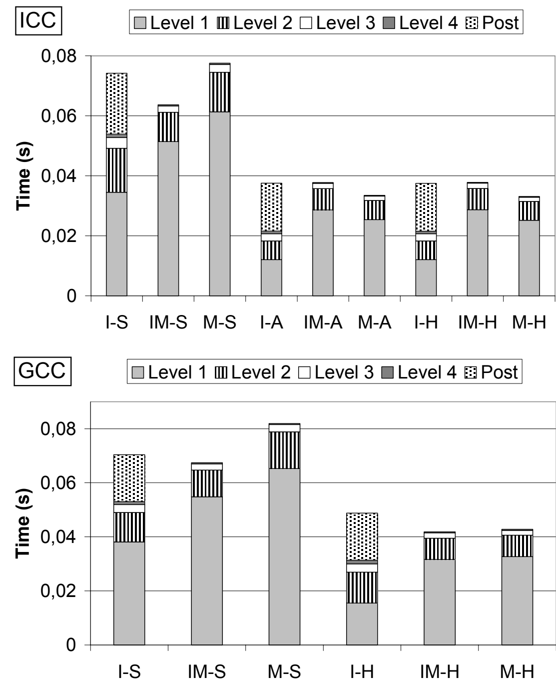

Fig. 6. Execution time breakdown for all the codes under study, using a 1024^2 pixels image. I, IM and M denote *inplace*, *inplace-mallat* and *mallat* approaches respectively. S, A and H denote *scalar*, *automatic-vectorized* and *hand-coded-vectorized* versions respectively.

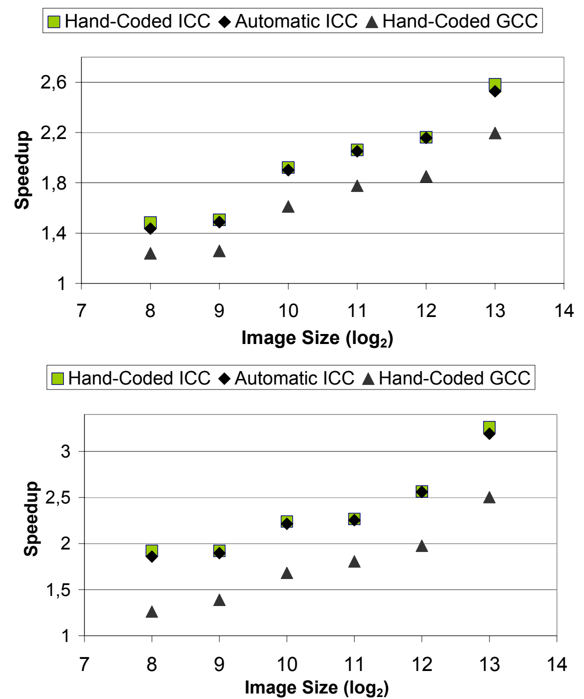

Fig. 7. Speedup achieved by the different vectorial codes over the *inplace-mallat* (top chart) and *inplace* (bottom chart) versions respectively. I, IM and M denote *inplace*, *inplace-mallat* and *mallat* approaches respectively.

Figure 8 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 keeps constant with the image size)

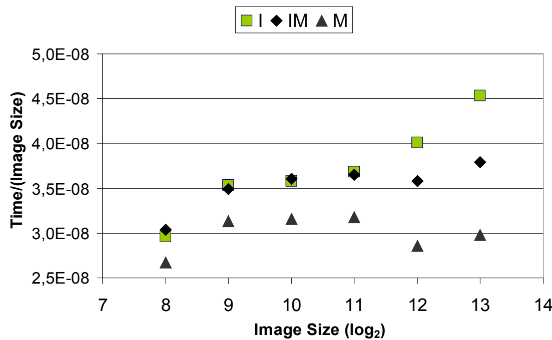

Fig. 8. Execution time (secs) to image size (pixels) ratio for the hand-coded vectorial versions (ICC compiler). I, IM and M denote *inplace*, *inplace-mallat* and *mallat* approaches respectively.

V. Conclusion

We have studied in this paper 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 as *inplace-mallat*, outperforms both a cache-conscious *inplace* implementation and a recent approach proposed by Chatterjee et al. [2] (denoted as *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 the image size.
4. In contrast with the scalar version, the vectorial *mallat* approach outperforms the other schemes. Moreover, its exhibits a better scalability and its benefits grow with the 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. As a 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.

References

[1] P. Meerwald, R. Norcen, and A. Uhl. *"Cache issues with JPEG2000 wavelet lifting"*. In proceedings of 2002 Visual Communications and Image Processing (VCIP'02), volume 4671 of SPIE Proceedings, San Jose, CA, USA, January 2002.

[2] S. Chatterjee and C. D. Brooks. *"Cache-efficient wavelet lifting in JPEG 2000"*. IEEE Int'l Conference on Multimedia and Expo, Lousanne, Switzerland, Aug 2002S.

[3] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto and F. Tirado. *"2-D Wavelet Transform Enhancement on General-Purpose Microprocessors: Memory Hierarchy and SIMD Parallelism Exploitation*". To be published in the Proceeding of the 2000 International Conference on High Performance Computing. Bangalore, India, December, 2002.

[4] D. S. Taubman and M. W. Marcellin. "*Jpeg2000: Image Compression Fundamentals, Standards, and Practice"* Kluwer International Series in Recursive data layout by itself produces little improvement in

[5] D. Santa-Cruz and T. Ebrahimi, "A study of JPEG 2000 still image coding versus other standards," in Proceedings of the X European Signal Processing Conference, Sept. 2000, vol. 2, pp. 673–676.

[6] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto and F. Tirado. Wavelet Transform for Large Scale Image Processing on Modern Microprocessors. To be published by Springer-Verlag in the Post-Conference book of Vecpar 2002, Porto, Portugal, June, 2002.

[7] Intel Corp. Real and Complex FIR Filter Using Streaming SIMD Extensions. Intel Application Note AP-809. Available at http://developer.intel.com.

[8] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto and F. Tirado. Vectorizing the Lifting Wavelet Transform on the Intel Pentium-4. Technical Report 02-077. Dept. of Computer Architecture. Complutense University, 2002.

[9] GNU GCC home page. http://gcc.gnu.org

[10] Intel Corp. Intel C/C++ Compiler for Linux. Information available at http://www.intel.com/software/products/compilers

[11] D. Chaver, M. Prieto, L. Piñuel, F. Tirado. Parallel Wavelet Transform for Large Scale Image Processing. Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'2002). Florida, USA, April 2002.