Wavelet Transform for Large Scale Image Processing on Modern Microprocessors [1]

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

Departamento de Arquitectura de Computadores y Automatica
Facultad de Ciencias Fisicas, Universidad Complutense
28040 Madrid, Spain
{dani02, tenllado, lpinuel, mpmatias, ptirado}@dacya.ucm.es

Abstract. In this paper we discuss several issues relevant to the vectorization of a 2-D Discrete Wavelet Transform on current microprocessors. Our research is based on previous studies about the efficient exploitation of the memory hierarchy, due to its tremendous impact on performance. We have extended this work with a more detailed analysis based on hardware performance counters and a study of vectorization, in particular, we have used the Intel Pentium SSE instruction set. Most of our optimizations are performed at source code level to allow automatic vectorization, though some compiler intrinsic functions have been introduced to enhance performance. Taking into account the abstraction at which the optimizations are performed, the results obtained on an Intel Pentium III microprocessor are quite satisfactory, even though further improvement can be obtained by a more extensive use of compiler intrinsics.

1. Introduction

Over the last few years, we have witnessed an important development in applications based on the discrete wavelet transform. The most outstanding success of this technology has been achieved in image and video coding. In fact, standards such as MPEG-4 or JPEG-2000 are based on the discrete wavelet transform (DWT). Nevertheless, it is without doubt a valuable tool for a wide variety of applications in many different fields [1][2]. This growing importance makes a performance analysis of this kind of transformation of great interest.

Our study focuses on general-purpose microprocessors. In these particular systems, the main aspects to be addressed are the efficient exploitation of the memory hierarchy, especially when handling large images, and how to structure the computations to take advantage of the SIMD extensions available on modern microprocessors.

With regard to the memory hierarchy, the main problem of this transform is caused by the discrepancies between the memory access patterns of two principal components of the 2-D wavelet transform: the vertical and the horizontal filtering [2]. This difference causes one of these components to exhibit poor data locality in the

[1] This work has been supported by the Spanish research grant TIC 99-0474

straightforward implementations of the algorithm. As a consequence, the performance of this application is highly limited by the memory accesses.

The platform on which we have chosen to study the benefits of the SIMD extensions is an Intel Pentium-III based PC. However, we should remark that most of the optimizations that we have performed to take advantage of this kind of parallelism do not depend on the particular characteristics of the Intel Pentium's SSE instruction set [3]. In fact, due to portability reasons, we have avoided the assembly language programming level. All the optimizations have been performed at the source code level. Basically, we have introduced some directives which inform the compiler about pointer disambiguation and data alignment, and some code modifications, such as changing the scope of the variables in order to allow automatic vectorization. Furthermore, we have also compared this approach with a hand-tuned vectorization based on language intrinsics, in order to measure the quality of the compiler.

This paper is organized as follows. The investigated wavelet transform and some related work are described in sections 2 and 3 respectively. The experimental environment is covered in section 4. In Section 5 we discuss the memory hierarchy optimizations, then in section 6 our automatic vectorization technique is explained and some results are presented. Finally, the paper ends with some conclusions.

2. 2-D Discrete Wavelet Transform

The discrete wavelet transform (DWT) can be efficiently performed using a pyramidal algorithm based on convolutions with Quadrature Mirror Filters (QMF). The wavelet representation of a discrete signal *S* can be computed by convolving *S* with the lowpass filter *H(z)* and highpass filter *G(z)* and downsampling the output by 2. This process decomposes the original image into two sub-bands, usually denoted as the coarse scale approximation (lower band) and the detail signal (higher band) [2].

This transform can be easily extended to multiple dimensions by using separable filters, i.e. by applying separate 1-D transforms along each dimension. In particular, we have studied the most common approach, commonly known as the *square* decomposition. This scheme alternates between operations on rows and columns, i.e. one stage of the 1-D DWT is applied first to the rows of the image and then to the columns. This process is applied recursively to the quadrant containing the coarse scale approximation in both directions. In this way, the data on which computations are performed is reduced to a quarter in each step (see figure 1) [2].

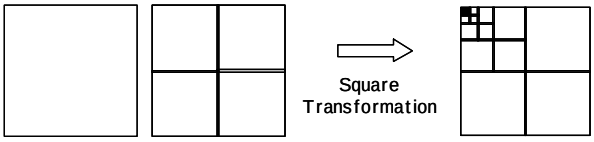

Fig. 1. The *Square* variant of the 2-D DWT.

From a performance point of view, the main bottleneck of this transformation is caused by the vertical filtering (the processing of image columns) or the horizontal one (the processing of image rows), depending on whether we assume a row-major or

a column-major layout for the images. In particular, all the measurements taken in our research have been obtained performing the whole wavelet decomposition using a (9,7) tap biorthogonal filter [2]. Nevertheless, qualitatively our results are filter-independent.

3. Related Work

A significant amount of work on the efficient implementation of the 2-D DWT has already been done for all sorts of computer systems. However, most previous research has concentrated on special purpose hardware for mass-market consumer products [4][5][6]. Focusing on general purpose microprocessors, S. Chatterjee *et al.* [7] and P. Meerwald *et al.* [8] proposed several optimizations aimed at improving cache performance. Basically, [8] investigates the benefits of traditional loop-tiling techniques, while [7] investigates the use of specific array layouts as an additional means of improving data locality.

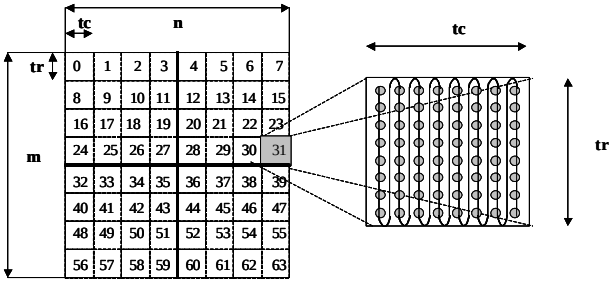

Fig. 2. 4-D layout.

The thesis of [7] 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. As an alternative, they studied the benefits of two non-linear layouts, known in the literature as 4-D (see fig. 2) and Morton [7]. In these layouts the original $m \acute{} n$ image is conceptually viewed as an $\lceil m/tr \rceil \times \lceil n/tc \rceil$ array of $tr \times tc$ tiles. Within each block, a canonical (row-major or column-major) layout is employed. For a benchmark suite composed of different dense matrix kernels and two different wavelet transforms, both layouts have low implementation costs (2-5% of the total running time) and high performance benefits. In particular, focusing on the wavelet transform, the running time improvements achieved on a DEC workstation (equipped with a 500 MHz Alpha 21164 microprocessor and 2 MB of L3 cache) reached up to 60% compared to a more traditional version of the code [7] (the Morton layout performance was slightly better).

The approach investigated in [8] is less aggressive. Nevertheless, they addressed the memory exploitation problem in the context of a whole application, the JPEG2000 image coding, which is more tedious to optimize than a wavelet kernel. In particular, they considered the reference implementations of the standard (known as jasper and

jj2000). By default, both implementations use a five-level wavelet decomposition with (7,9) biorthogonal filters as the intra-component transform of the coding [8]. The solution investigated by these authors consists in applying a loop-tiling strategy to the vertical filtering (the reference implementations used a row-major layout), which they dubbed "aggregation". In this scheme, instead of processing every image column all the way down in one step, which produces very low data locality (on a row-major layout, rows are aligned along cache lines), the algorithm is improved by processing several columns concurrently so that the spatial locality can be more effectively exploited.

We have extended these previous studies by assessing the influence of the SIMD extensions and including a more elaborate analysis based on the Intel PIII's performance counters. In this first version of our study we have followed the kernel approach chosen by S. Chatterjee *et al.* although, as a future research area, we intend to introduce the proposed optimizations on a entire well-known application such as the JPEG-2000.

4. Experimental Environment

The performance analysis presented in this paper has been carried out on a Pentium-III 866 MHz (0,18microns) based PC running under Linux, the main features of which are summarized in [20]. The programs have been compiled using the Intel C/C++ Compiler for Linux (v5.0.1) [9] and the compiler switches (-O3 –tpp6 -xK) described also in [20].

Our measurements have been made using the performance-monitoring counters available on the P6 processor family [3]. This micro-architecture provides two 40-bit performance counters, allowing two types of events to be monitored simultaneously. In order to avoid assembly language programming and due to portability reasons, we have employed a high-level application-programming interface, PAPI (v2.0.1 beta) [10]. This API includes platform-independent procedures for initialising, starting, stopping, and reading the counters, although it needs some operating system support for user level access to the counters. In Linux, this tool relies on the *perfctr* kernel driver (v2.3.2) [11], which also supplies 64-bit resolution virtual counter support (i.e. per process counter). We should note that to improve counter accuracy we have employed native events instead of PAPI predefined ones, and we have avoided monitoring strategies such as multiplexing and sampling (i.e. only two events are considered per execution).

5. Cache analysis

As mentioned before, the wavelet transform poses a major hurdle for the memory hierarchy, due to the discrepancies between the memory access patterns of the two main components of the 2-D wavelet transform: the vertical and horizontal filterings [2]. Consequently, the improvement in the memory hierarchy use represents the most

important challenge of this algorithm from a performance perspective. In this section, we have exhaustively analyzed the cache behavior of the three different approaches introduced previously, namely the 4-D and Morton layouts (non-linear layouts) and the row-major layout combined with the aggregation technique (see section 3). We have divided this section into 3 different parts. First, we describe the different ways to apply the vertical and horizontal filters and their relation to the image layout. Section 5.2 discusses some implementation issues and the experimental results are presented in section 5.3.

5.1 Tile layout and block processing type

The following algorithms show four reasonable ways of applying a 1-D filter to a tile of $tr \times tc$ elements, which we have denoted as vertical, horizontal, N and Z element processing:

* *Vertical:* `foreach column{ foreach row{ foreach coef{ filter }}}`
* *Horizontal:* `foreach row{ foreach column{ foreach coef{ filter }}}`
* *N:* `foreach column{ foreach coef{ foreach row{ filter }}}`
* *Z:* `foreach row{ foreach coef{ foreach column{ filter }}}`

Depending on the tile memory layout, some processing types are preferable over others due to data locality. It is relatively obvious that for the row-major layout the best access pattern is produced when elements are processed horizontally, for either vertical or horizontal filtering. On the other hand, for the column-major layout, it is better to process the elements vertically. The Z and N approaches represent a hybrid approach that was considered in the previous versions of our codes since they are easily vectorizable, as will be explained later in section 6.

In order to make a fair comparison of the different alternatives analyzed in this section, we employed the best processing type for each kind of block layout. Given that for the 4-D and Morton layouts we have opted to use the same approach as that followed in [7] (within each block a column-major layout is employed), we have chosen vertical processing in these cases. For the row-major layout combined with the aggregation technique we have employed horizontal processing. Nevertheless, we should remark that due to the symmetry of the problem, these particular choices have no effect on the overall performance.

Figure 3 graphically describes the processing of the image tiles in the 4-D and Morton approaches (for simplicity, boundary data have been ignored). The vertical filtering does not need any special treatment since the image columns are stored contiguously in memory. In this case, the main problem is due to the horizontal filtering, since processing the tile row by row does not take advantage of the spatial locality. In order to remedy this situation, the tile is swept column by column in a similar way to the aggregation technique proposed in [8].

Figure 4 illustrates both filtering types when a row-major layout is employed for the whole image. The horizontal filtering does not cause any problem whereas the vertical one has to be improved by means of aggregation [8].

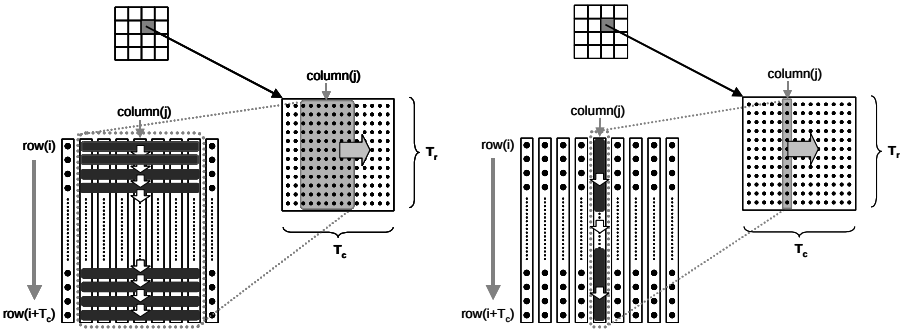

Fig. 3. Horizontal (left-hand chart) and vertical (right-hand) filtering employed in the 4-D and Morton approaches.

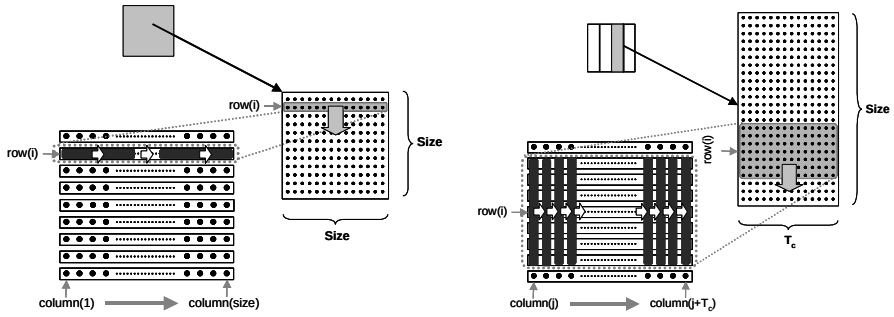

Fig. 4. Horizontal filtering (left-hand chart) and vertical filtering (right-hand chart) for the row-major layout.

5.2 Implementation issues

* *Filter loop unroll:* We have observed that the compiler does not automatically unroll the filter loop in any components of the transform. Due to its reduced number of iterations (filter length), this modification can be easily performed by hand. In addition, this modification allows the compiler to perform further optimizations and, as section 6.1 explains, also permits automatic vectorization to be employed in the vertical and horizontal processing.
* *Data alignment:* Strictly speaking, data alignment [12][13] is not required in our codes since the SSE instruction set includes instructions that allow unaligned data to be copied into and out of the vector registers. However, such operations are much slower than aligned accesses, which may cause a significant overhead. To avoid this drawback we have employed 16-byte aligned data in all our codes, although for the scalar versions this optimization has no significant effect.

5.3 Experimental results

The results reported in this section have been obtained using the experimental framework explained in section 4. Before analyzing them, we should briefly explain the metrics involved. The execution time measurements have been obtained using the PAPI virtual time routines [10], which are context-switch independent. The memory hierarchy behavior has been monitored through the "DCU LINES IN" and "L2 LINES IN" events, which represent the number of lines that have been allocated in the L1 data cache, and the number of L2 allocated lines respectively. These events are strongly related with the number of L1 and L2 cache misses.

Figures 5 and 6 represent memory hierarchy behavior and the execution time for an image size of 8192^2 pixels. When employing horizontal filtering all the approaches obtain, for the optimal block size, comparable results in execution time and in both L1 and L2 allocated lines. However, with vertical filtering, Morton and 4-D produce a significantly lower number of misses than the row-major layout in both levels of the memory hierarchy, as well as a slower running time.

This relatively bad row-major layout behavior is due to the poor spatial locality of its memory access pattern. Furthermore, for the L2 cache, elements belonging to the same wavelet coefficient computation are using the same block set, resulting in a high number of conflicts (we use a 9-coefficient filter, and the Pentium-III has only 8 blocks per set). In [8], this problem is overcome using array padding (dubbed "row extensions technique" by the authors) to force the image width not to be a power of two. However, as the authors themselves suggest, this simple technique has the disadvantage that the original input image has to be modified. In an application such as the JPEG-2000, inserting dummy data changes the final coded bitstream [8]. Using 4-D or Morton, array padding is not necessary, since in these cases conflict misses are not a function of the image size but of the tile size.

Regarding the L1 data cache behavior we should mention that, although less influential on performance, the improvements of both the 4-D and Morton layouts on the row-major layout are also significant. We should also remark that the similarity of the curves for the execution time and L2 allocated lines suggests a strong relationship between performance and L2 behavior. As a result, the 4-D and Morton approaches produce a speedup gain of about 2.25 over the row-major layout. Comparing the two non-linear approaches, we observe that the results are almost the same (1.3% of difference), especially for the optimum tile sizes.

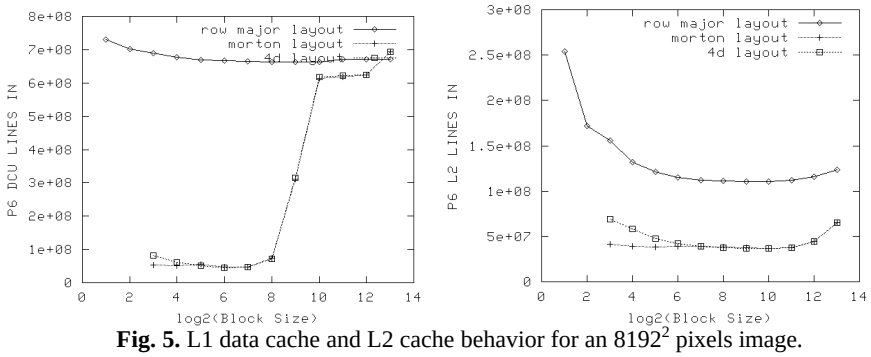

Fig. 5. L1 data cache and L2 cache behavior for an 8192^2 pixels image.

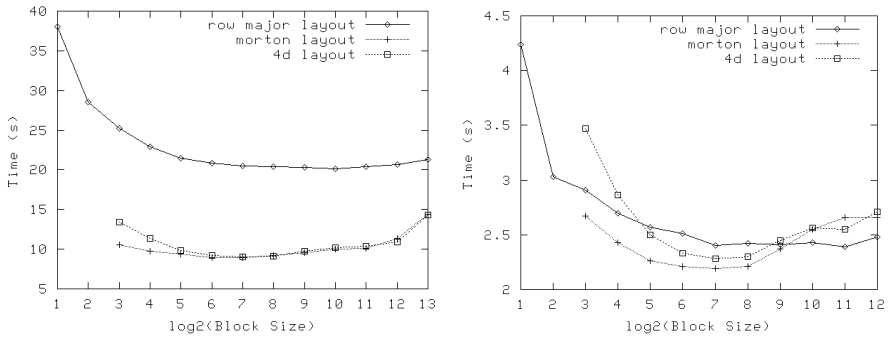

Fig. 6. Execution time for an 8192^2 pixels image (left chart) and for a 4096^2 pixels image (right chart).

Figure 6 (right chart) shows the execution time for an 4096^2 pixel image. Performance is strongly related to the L2 behavior, although differences in the L1 behavior now translate as small variations in execution time (with a 8192^2 image they were almost inappreciable in terms of time). The benefits of non-linear layouts are less significant in this case, since the stride of the row-major layout in the vertical filtering is lower, resulting in a smaller number of L2 cache conflicts (only half of the elements of wavelet coefficient computation are competing for the same block set). Comparing the behavior of the Morton and 4-D layouts, we remark once again that they are very similar (4% of difference).

From the previous results we can conclude that, in general, Morton and 4-D are preferable to the row-major layout, since their memory access patterns exhibit more locality. The memory hierarchy is therefore more efficiently exploited and thus the execution time is significantly reduced. The running time benefits of these approaches are higher for image sizes of 8192^2 pixels and above, mainly due to the L2 behavior explained earlier. Taking into account that the 4-D layout is simpler to implement (it does not need a lookup table to handle blocks [7]) and achieves a similar performance to the Morton, we have chosen this method to study the vectorization.

6. SIMD Optimization

Most previous research on parallel wavelet transforms has concentrated on special purpose hardware (as mentioned in section 3) and out-of-date SIMD architectures, such as the Connection Machine [14]. Work on general purpose multiprocessor systems includes [15] and [16], where different parallel strategies for the 2-D wavelet transform were compared on the SGI Origin 2000, the IBM SP2 and the Fujitsu VPP3000 systems respectively. In [17] a highly-parallel wavelet transform is presented but at the cost of changing the wavelet transform semantic. Other work includes [18], where several strategies for the wavelet-packet decomposition are studied.

We have focused our research on the potential benefits of Single Instruction Multiple Data (SIMD) extensions. Among related work, we can mention [19], where

an assembly language vectorization of real and complex FIR filters is introduced based on Intel SSE. Our main interest is to assess whether it is possible to take advantage of such extensions to exploit the data parallelism available in the wavelet transform, though in a filter-independent way and avoiding low level programming.

Most of the results reported in this work have been obtained by using automatic vectorization. As expected, the compiler was not able to vectorize any loop by itself, so both code modifications and guided compilation were necessary. However, it should be noted that the analysis of the vectorization inhibitors provided by the Intel compiler has been a considerable aid. We have also optimized the code using the intrinsic functions that the Intel compiler offers. This technique involves additional improvements at the expense of a greater coding effort, although it is more portable than coding at the assembly level since most compilers provide similar functions.

This section is divided into two parts. In the first, we have studied how to vectorize the horizontal filtering. In the second, we have extended the vectorization method to the whole transform. For the sake of simplicity, we have only considered the 4-D column-major layout, although analogous results can be obtained for the Morton approach (see [20]).

6.1 Horizontal filtering vectorization

Depending on the memory layout, either column-major or row-major, we can vectorize either the horizontal or vertical filtering using the methodology presented below. In particular, we have applied this technique to the horizontal filtering since we are focusing on the 4-D column-major layout.

6.1.1 Methodology

Loops must fulfill some requirements in order to be automatically vectorized. Primarily, only loops with simple array index manipulation (i.e. unit increment) and which iterate over contiguous memory locations are considered (thus avoiding non-contiguous accesses to vector elements). Obviously, only inner loops can be vectorized. 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, it is obvious that not all the processing types and filtering components can be vectorized automatically. In particular, for a Row-major Layout only the Vertical Filtering using *Horizontal* or *Z Processing* can be automatically vectorized, while for a Column-major Layout only the Horizontal Filtering using *Vertical* or *N Processing* do. Nevertheless, we should remark that when using either assembly language or function intrinsics these limitations can be overcome at the expense of more coding effort.

6.1.2 Vectorization

This technique consists in calculating the wavelet coefficients following the element layout in the memory. We shall suppose that to evaluate a certain wavelet coefficient we must center the filter on element j of row i in one of the 4-D layout tiles. The

optimum processing consists in moving the filter downwards, from row to row, to calculate all the wavelet coefficients of column *j*. From figure 7 (left chart) it seen that, in this particular case (a 7-tap filter), each coefficient requires 7 floating point multiplications and 6 floating point additions. Consequently, to calculate 4 coefficients, 28 floating point multiplications and 24 floating point additions are necessary. However, if vectorization is enabled (figure 7, right chart) the calculations of every 4 coefficients can be performed concurrently. Since the elements of each column are stored contiguously, the compiler is able to load the matrix elements into the SSE registers in groups of four (thus using less instructions).

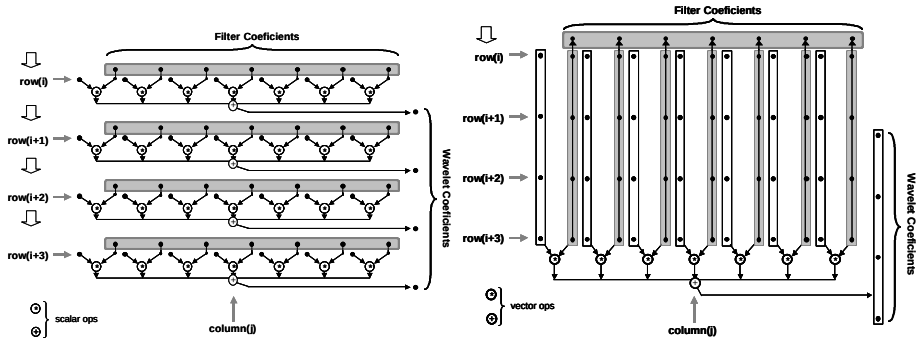

Fig. 7. Horizontal filtering using vertical sweep for the scalar version (left chart) and Horizontal filtering using vertical sweep for the vector version (right chart).

6.1.3 Experimental Results

A) Vectorized vs. Non-vectorized horizontal Filtering

Figure 8 shows the execution time for 8192^2 and 4096^2 pixel images using both vectorized and non-vectorized versions of the code. We observed that for every configuration and image size under study, the vectorized horizontal filtering beats the scalar running time. In particular, for the optimum block size it achieves a speedup of about 2 (for both image sizes), which translates to a speedup of about 1.4 for the whole transform. Obviously, the vertical filtering behavior has not changed since this part of the program has not been modified. Considering the entire transform, we should note that the optimum block size for each version of the code is different, because the contribution of the vectorized horizontal component is lower than that of the scalar component.

We have not included memory event counts, since the vectorization does not affect the number of L1 and L2 allocated lines but only reduces the number of memory accesses. In other words, the hierarchy memory exploitation remains the same.

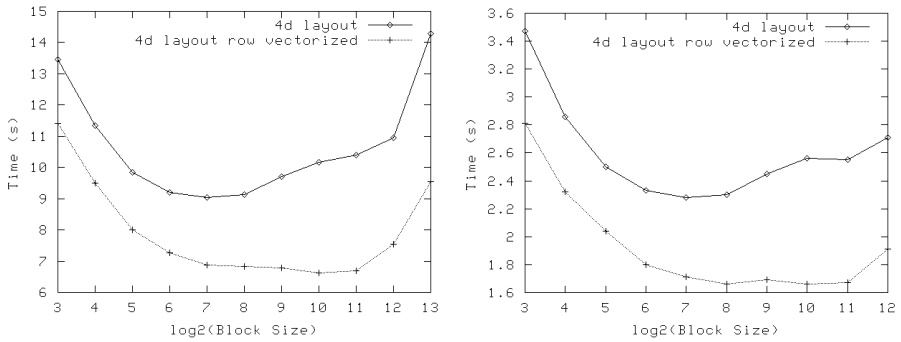

Fig. 8. Execution time for 8192x8192 (left chart) and for 4096x4096 (right chart).

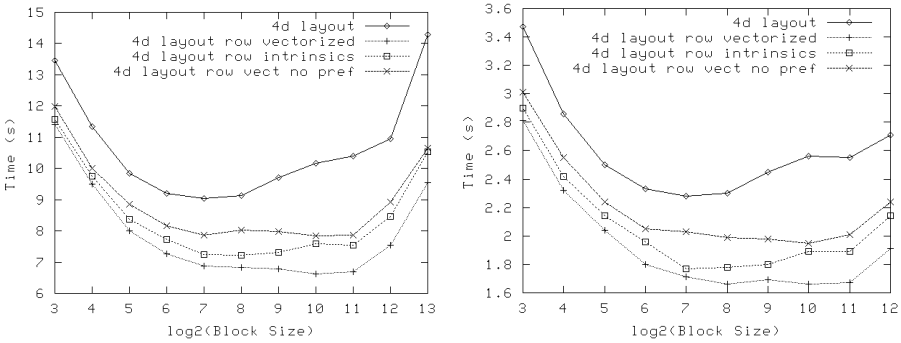

Fig. 9. Execution time for 8192x8192 (left chart) and for 4096x4096 (right chart).

B) Automatically vs. hand-coded (intrinsic) vectorization

We have also attempted to evaluate the efficiency of the compiler-generated vectorial code. To do this, we have written an optimal hand-tuned code using the compiler intrinsics (the interested reader can find more information in [20]). Figure 9 shows the results for these two versions of the code. The initial comparison of these codes was a little surprising since the automatic version turned out to be faster than our best manual code. After a detailed analysis at the assembly level, we realized that this difference is caused by the prefetching introduced by the compiler when automatic vectorization is enabled. We have verified this conclusion by removing the prefetch instructions from the assembler code, the results of which are also shown in figure 9. As can be seen, with regard to vectorization the automatic code is worse than the manual (about 21% worse).

We should remark that the compiler does not perform automatic prefetching in the hand-tuned code. In addition, introducing manual prefetching is a tough task and the resulting code is highly platform-dependent, which makes the automatic vectorization preferable since it produces a higher speedup with a minor programming effort.

Thus, returning to our original comparison (automatic vectorizable vs. scalar), the speedup of the automatic version (about 2) over the scalar code is due not only to vectorial operations but also to pre-fetching, each with the same contribution to the overall gain in speedup.

6.2 Full vectorization

We have obtained excellent results from horizontal filtering vectorization due to the matrix elements being stored contiguously in columns. In order to apply the same technique to the vertical filtering, we needed the elements to be stored contiguously in rows. One possible solution was to apply the horizontal filtering followed by a transposition of the resulting wavelet coefficients. This then allowed us to use the same vectorization technique vertically. To carry out the transposition efficiently we could not work with the whole matrix at the same time. Therefore, this transform was performed tile by tile taking advantage of the 4-D layout, which required the use of an auxiliary buffer of tile size. Note that this kind of transposition is not feasible when using the row-major layout since the image is not divided into tiles.

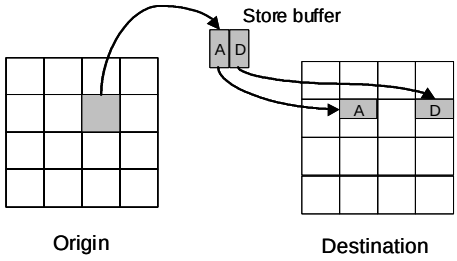

Fig. 10. Block transposition.

As can be seen in figure 10, first we performed the horizontal filtering and stored the resulting coefficients in the auxiliary buffer. Then we transposed the buffer, storing the coefficients in the destination matrix in rows to be consequently subjected to vectorized vertical filtering.

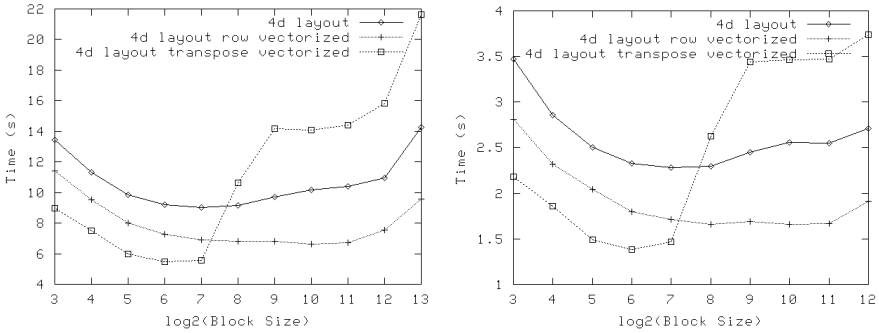

Fig. 11. Execution time for the whole wavelet transform for 8192x8192 (left chart) and for 4096x4096 (right chart).

The block transposition is divided into a 4x4 matrix transposition, which is implemented by using the Intel `_MM_TRANSPOSE4_PS` intrinsic function [9]. Obviously the transpose computation implies a cost in time since it is necessary to use extra load and store instructions. However, the smaller the tile size the more efficiently all these extra memory accesses exploit the time and spatial locality. For images of 4096^2 the speedup achieved with this full vectorization compared to the scalar is 1.7 and for

images of 8192^2 it is 1.6. Therefore, as we can see in figure 11 the cost of the tile transposition is by far compensated by the improvement obtained through the vectorization of the vertical filtering.

7. Conclusions

In this paper we have introduced a novel approach to optimize the computation of the 2D DWT for large scale image processing based on non-linear data layouts, and automatic prefetching and vectorization. The main conclusions can be summarized as follows:

1. As shown by previous research [7], an increase in speedup is obtained through non-linear accesses, such as the 4-D or Morton layouts, which exploit spatial locality more effectively. The difference in speedup between the 4-D and Morton layouts is insignificant. Due to the simplicity of the 4-D compared to Morton, we recommend the former.
2. We have introduced a novel approach to structure the computation of the wavelet coefficients that allows automatic vectorization and pre-fetching, which is independent of the filter size and the computing platforms (assuming that similar SIMD extensions are available).
3. Our hand-tuned code achieves a better exploitation of SIMD parallelism at the expense of more coding effort. However, the compiler cannot perform prefetching in this code. In addition, introducing prefetching by hand is a tough task and it is highly platform-dependent. As a consequence, the automatic version is strongly preferable since the lower SIMD exploitation is by far compensated by prefetching.
4. In order to apply the vectorization to both filterings (horizontal and vertical) a block transposition is required. However, the performance gain achieved through vectorization by far compensated the transposition overhead.

References

[1] Z. Zhang and R. S. Blum. A Categorization of Multiscale-Decomposition-Based Image Fusion Schemes with a Performance Study for a Digital Camera Application. Proceeding of the IEEE, Vol. 87(8):1315-1325, August 1999

[2] E. J. Stollnitz, T. D. DeRose and D. H. Salesin. Wavelets for Computer Graphics: Theory and Applications. Computer Graphics and Geometric Modeling, Morgan Kaufmann Publishers, Inc. San Francisco, 1996

[3] Intel Corp. Pentium-III processor. http://developer.intel.com/design/PentiumIII

[4] C. Chakrabarti and C. Mumford. Efficient realizations of encoders and decoders based on the 2-D discrete wavelet transforms. IEEE Trans. VLSI Syst., pp. 289-298, September 1999

[5] T. Denk and K. Parhi. LSI Architectures for Lattice Structure Based Orthonormal Discrete Wavelet Transforms. IEEE Trans. Circuits and Systems, vol. 44, pp. 129-132, February 1997

[6] C. Chrysafis and A. Ortega. Line Based Reduced Memory Wavelet Image Compression. IEEE Trans. on Image Processing, Vol 9, No 3, pp. 378-389, March 2000
[7] S. Chatterjee, V. V. Jain, et al. Nonlinear Array Layouts for Hierarchical Memory Systems. Proceedings of 1999 ACM International Conference on Supercomputing, pp. 444-453, Rhodes, Greece, June 1999
[8] P. Meerwald, R. Norcen, et al. Cache issues with JPEG2000 wavelet lifting. In C.-C. Jay Kuo, editor, Visual Communications and Image Processing 2002 (VCIP'02), volume 4671 of SPIE Proceedings, San Jose, CA, USA, January 2002
[9] Intel Corp. C/C++ Compiler. http://www.intel.com/software/products/compilers
[10] K. London, J. Dongarra, et al. End-user Tools for Application Performance Analysis, Using Hardware Counters. Presented at International Conference on Parallel and Distributed Computing Systems. August 2001
[11] Perfctr Linux driver. Info. available at http://www.csd.uu.se/~mikpe/linux/perfctr
[12] Intel Corp. Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler. Intel Application Note AP-833. Available at http://developer.intel.com
[13] Intel Corp. Intel Architecture Optimization. Reference Manual. Available at http://developer.intel.com
[14] M. Holmström. Parallelizing the fast wavelet transform. Parallel Computing, 11(21):1837-1848, April 1995
[15] 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
[16] O.M. Nielsen and M. Hegland. Parallel Performance of Fast Wavelet Transform. International Journal of High Speed Computing, 11 (1): 55-73, June 2000
[17] L. Yang and M. Misra. Coarse-Grained Parallel Algorithms for Multi-Dimensional Wavelet Transforms. The journal of Supercomputing 11:1-22 , 1997
[18] M. Feil and A. Uhl. Multicomputer algorithms for wavelet packet image decomposition. Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'2000), pages 793-798, Cancun, Mexico, 2000
[19] Intel Corp. Real and Complex FIR Filter Using Streaming SIMD Extensions. Intel Application Note AP-809. Available at http://developer.intel.com
[20] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto and F. Tirado. Vectorizing the Wavelet Transform on the Intel Pentium III Microprocessor. Technical Report 02-001. Dept. of Computer Architecture. Complutense University, 2002