Reducing 3D Wavelet Transform Execution Time through the Streaming SIMD Extensions

Gregorio Bernabé and José M. García
Universidad de Murcia
Dpto. Ingeniería y Tecnología de Computadores
30071 MURCIA (SPAIN)
gbernabe, jmgarcia@ditec.um.es

José González
Intel Barcelona Research Center
Intel Labs, Barcelona
08034 BARCELONA (SPAIN)
josex.gonzalez.gonzalez@intel.com

Abstract

This paper focuses on reducing the execution time of the video compression algorithms based on the 3D wavelet transform. We present several optimizations that could not be applied by the compiler due to the characteristics of the algorithm. First, we use the Streaming SIMD Extensions (SSE) for some of the dimensions of the sequence (y and $time$), in order to reduce the number of floating point instructions, exploiting Data Level Parallelism. Then, we apply loop unrolling and data prefetching to critical parts of the code, and finally the algorithm is vectorized by columns, allowing the use of SIMD instructions for the y dimension. Results show improvements of up to 1.54 over a version compiled with the maximum optimizations of the Intel C/C++ compiler. Our experiments also show that, allowing the compiler to perform some of these optimizations (i.e. automatic code vectorization) causes performance slowdown which demonstrates the effectiveness of our optimizations.

1 Introduction

The increase in the volume of medical video generated in hospitals, as well as its strict regulations and quality constraints makes the research in compression techniques especially oriented to this video an interesting area.

In the last few years, the application of the wavelet transform [14][30] has reached an important development. The wavelet transform has been mainly applied to image compression. Several coders have been developed using 2D wavelet transform [3][21][29]. Moreover, the last image compression standard, JPEG2000 [23][28] is also based on the 2D discrete wavelet transform with a dyadic mother wavelet transform.

The 2D wavelet transform has been used for compressing video [17] as well. However, three dimensional (3D) compression techniques seem to offer better results than two dimensional (2D) compression techniques which operate in each frame independently. Muraki introduced the idea of using 3D wavelet transform to efficiently approximate 3D volumetric data [24][25]. Since one of the three spatial dimensions can be considered similar to time, a 3D subband coding using the zerotree method (EZW) was presented to code video sequences [10] and posteriorly improved with an embedded wavelet video coder using 3D set partitioning in hierarchical trees (SPIHT) [18]. Nowadays, the standard MPEG-4 [4][5] supports an ad-hoc tool for encoding textures and still images based on a wavelet algorithm.

However, one of the main drawbacks of using the 3D wavelet transform is its excessive execution time. Since three dimensions are exploited in order to obtain high compression rates, the working set becomes huge and the algorithm becomes memory bound. Blocking is a well-known optimization technique for improving the effectiveness of memory hierarchies [1][19][22]. Instead of operating on entire rows, columns or frames of the working set, blocked algorithms operate on working subsets or blocks, so that data loaded into the faster levels of the memory hierarchy are reused. Blocking has been shown to be useful for many algorithms in linear algebra like BLAS [15], LAPACK [2] and more recently ATLAS [32].

In previous works [6][7], we presented an implementation of a lossy encoder for medical video based on the 3D Fast Wavelet Transform (FWT). This encoder achieves high compression ratios with excellent quality, so that medical doctors cannot find differences between the original and the reconstructed video. Later, we proposed a memory conscious 3D wavelet transform that exploits the memory hierarchy by means of blocking algorithms [8], thus reducing the final execution time. In particular, we proposed and evaluated several blocking approaches that differ in the way that the original working set is divided. We also proposed

the reuse of some computations to save floating point (FP) operations as well as memory accesses. Results showed that the rectangular overlapped approach provided the best execution times among all tested algorithms, maintaining both the compression ratio and the video quality.

The introduction of Multimedia Extensions (MMX) [20] and the Streaming SIMD Extensions (SSE) [12] available on modern processors, provide a technology designed to accelerate multimedia and communications software, being able to reduce the execution time of the applications. Ranganathan et al. [27] show that the Sun VIS media ISA extensions provide an additional 1.1 to 4.2 performance improvement over several image and video processing applications. Nachtergaele et al. [26], proposed a software implementation of the MPEG-4 based on the integer wavelet transform using Multimedia Extensions. Conte et al. [11] evaluated several applications obtaining substantial speedups with MMX/SSE code.

In this paper, we present several techniques in order to reduce the execution times of the rectangular overlapped approach based on the 3D-FWT, maintaining the same compression ratio and video quality. Particularly, we attempt to take advantage of the Streaming SIMD Extensions efficiently [9], using the new Intel C/C++ Compiler [13]. We also employ others classic methods like data prefetching and loop unrolling. Finally, we examine the source code in order to exploit the temporal and spatial locality in the memory cache. A method to enhance the locality of the memory hierarchy, based on the computation of the wavelet transform in the x and y dimensions, will be presented taking into account that the mother wavelet function is the Daubechie's of four coefficients (Daub-4).

The rest of this paper is organized as follows. The background is presented in Section 2. Section 3 describes several approaches in order to reduce the execution times in the 3D-FWT algorithm. We present the main details of each method. Experimental Results on some test medical video are analyzed in Section 4. Finally, Section 5 summarizes the work and concludes the paper.

2 Background

In this Section, we review the framework on top of it our enhancements have been built. We first review the initial blocking version of the Wavelet Transform, then we introduce the Intel C/C++ compiler along with the advanced multimedia extensions.

2.1 Blocking the Wavelet Transform

Previous Wavelet-based encoder obtained excellent results both in compression rate and quality (PSNR), as it can be observed in [6][7]. These results were obtained with

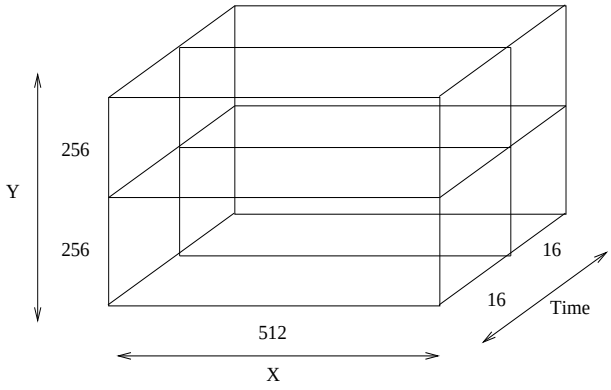

Figure 1. Rectangular approach

the 3D-FWT working on video sequences of 64 frames of $512x512$ pixels (16 MBytes of working set). This huge working set limits the performance of such algorithm, making it unfeasible for real-time video transmission. Initial results showed that this algorithm is completely memory bound, therefore, blocking techniques become an interesting approach to reduce its memory requirements and thus the execution time.

The goal of blocking algorithms is to exploit the locality exhibited by memory references by means of partitioning the initial working set in limited chunks that fit in the different levels of the memory hierarchy. In this way two positive effects appear: in the one hand, memory accesses are accelerated since data are actually at the higher levels of the memory hierarchy (closer to the processor core). On the other hand, traffic between main memory and the processor chip is drastically reduced, obtaining a better use of the bandwidth provided by the baseline computer system.

However, applying blocking algorithms to video coders is a challenge: not only the memory hierarchy must be exploited by means of an optimum data partitioning, but also quality must be preserved. Note that partitioning the working set into independent blocks may lead to unexpected degradations on the quality of the resulting video due to artifacts in the block bounds. It is detected an increasing degree of visibility of the discontinuity in the reconstruction at adjacent subcubes boundaries because artifacts effects appear. This is due to the way that computation is performed in the FWT, where, for a particular pixel, the value of its coefficient after the transform is correlated with the original values of its neighboring pixels.

The rectangular overlapped partitioning [8], where the original cube is divided into several rectangles, as we can observe in figure 1, was presented as the best approach for two main reasons: first, it exploits better the spatial locality of memory references because the frames are stored in memory following a row order, and second, there is a reuse of floating point operations. The optimal block size ($512x64x16$) obtains a speedup of 2.75 compared to the

non-blocking overlapped wavelet transform maintaining the same compression rate and quality as the non-blocking approaches.

However, analyzing the cache behavior, we observe that the number of L1 and L2 cache misses are still too high even for the optimal block size. On the other hand, the number of floating point instructions executed is also very large. Note that, since after blocking the algorithm is not completely memory bound, speeding-up floating points instructions become crucial to obtain better performance results.

In the next section, we will present different methods to decrease the L1 and L2 cache misses and the floating point instructions executed in order to decrease the execution time.

2.2 Intel C/C++ Compiler

Up to now, we have used the gcc compiler of gnu. In [8], results were obtained with the gcc compiler. However, all the algorithms evaluated in this work, including the rectangular overlapped approach, have been compiled using the Intel C/C++ Compiler for Linux (v5.0.1) [13], in order to allow our proposals to perform the best on Intel architecture-based computers. This compiler provides high performance using new compiler optimizations. It supports for Streaming SIMD Extensions, data prefetching, automatic vectorization and generates and optimizes code specialized for the Pentium III processor, in order to obtain the best performance.

2.3 Streaming SIMD Extensions (SSE)

Pentium III processor family features a rich set of SIMD instructions on packed integers and floating-point numbers that can be used to boost the performance of loops that perform a single operation on different elements in data set.

The Pentium III processor introduced the 128-bit streaming SIMD extensions [31], supporting floating-point operations on 4 single-precision floating-point numbers, implemented through of eight 128-bit data registers, called xmm0, xmm1, ..., xmm7.

On the other hand, the Intel C/C++ Compiler follows the standard approach to the vectorization of inner loops [33]. First, statements in a loop are reordered according to a topological sort of the acyclic condensation of the data dependence graph for this loop. Then, statements involved in a data dependence cycle are either recognized as certain idioms that can be vectorized, or distributed out into a loop that will remain serial. Finally, vectorizable loops are translated into SIMD instructions.

However, automatic vectorization is still difficult to achieve due to high restrictions imposed by compilers and the nature of the algorithm of the wavelet transform. Instead, and as will be shown in next Section, we have manually vectorized the code, since it was simple and more effective than giving hints to help the compiler.

```
/* c0, c1, c2, c3: Daub-4 coefficients */
/* pixels 1..n = p[0..n] */
/* temporal vector: low-pass */
float low[n/2], high[n/2];
for(i = 0, j = 0; i < n; i += 2, j++) {
low[j] = c0*p[i] + c1*p[i+1] + c2*p[i+2] + c3*p[i+3];
high[j] = c3*p[i] - c2*p[i+1] + c1*p[i+2] - c0*p[i+3];
}
```

Figure 2. Algorithm of 1D-FWT overlapped with Daub-4

3 Optimizing the Rectangular Overlapped Approach

In this Section, several enhancements over the original blocking algorithms are presented, with the aim of reducing both the number of FP instructions and the pressure over the memory subsystem.

3.1 SSE Extensions for the Wavelet Transform

The SSE are used to exploit fine-grained parallelism by vectorizing loops that perform a single operation on multiple elements in a data set. Therefore, SSE are applied to our wavelet overlapped transform algorithm for an unidimensional signal (1D-FWT) of n pixels with the Daub-4 as mother wavelet function.

As we can observe in Figure 2, the value of each resulting wavelet coefficient depends on four pixels, and 8 floating point multiplications and 6 floating points additions are needed to obtain the low and high pass for each pixel. Then, for 4 coefficients, 32 floating point multiplications and 24 floating points additions are necessary.

Figure 3 shows the computation of the first four low-pass resulting wavelet coefficients. We will refer to this optimization as SSE vectorization by hand. First, four SSE registers (xmm0, xmm1, xmm2 and xmm3) are initialized with the Daub-4 coefficients. Second, the pixels are loaded in group of four into the SSE registers (xmm4, xmm5, xmm6 and xmm7). Finally, 4 floating point multiplications and 3 floating points additions are performed among the registers SSE in order to obtain the same wavelets coefficients as in the algorithm of 1D-FWT overlapped with Daub-4. We can obtain the high-pass wavelet coefficients in the same way with 4 floating point multiplications and 3 floating points additions more. Therefore, the total number of floating point instructions has been reduced from 56 to 15 instructions.

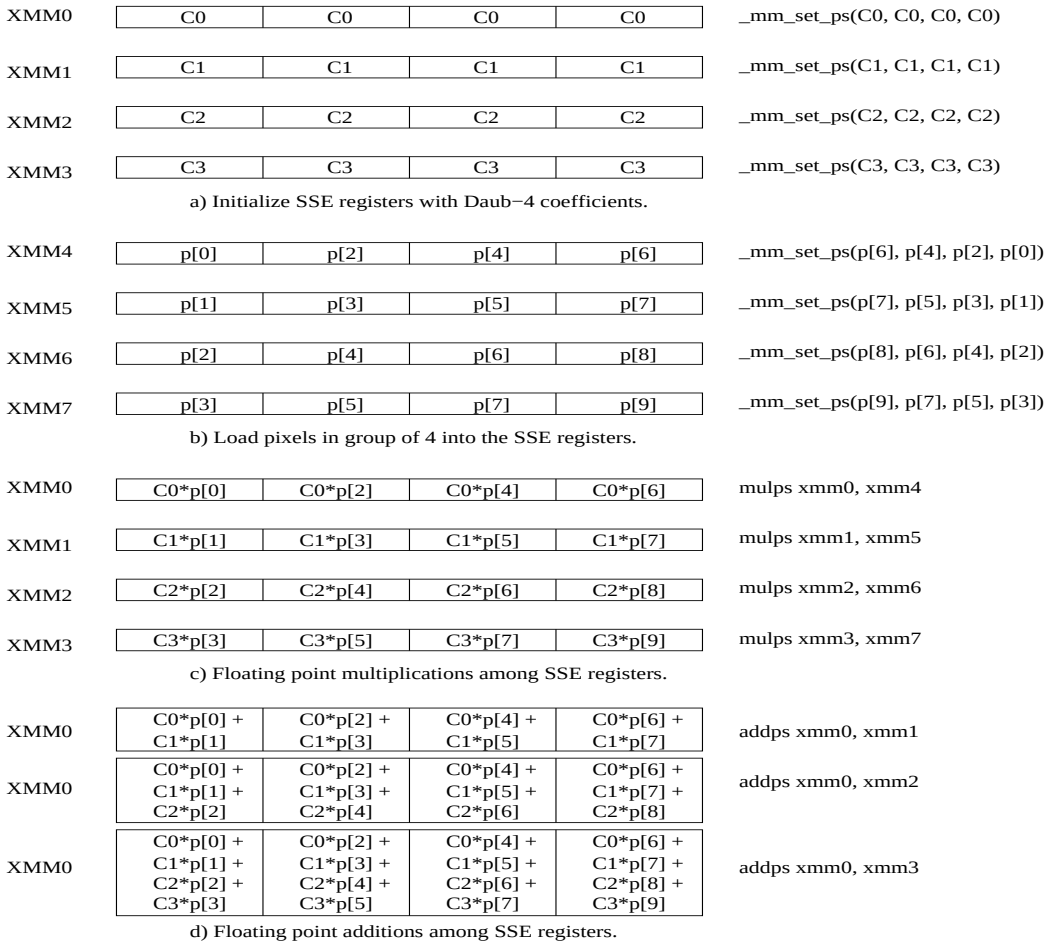

Figure 3. Phases for the compute of the first four low-pass wavelet coefficients with the SSE registers

3.2 Loop Unrolling and Prefetching Data

Loop unrolling is usually applied by the compiler if it sees a room for improvement. However, due to the nature of the Wavelet algorithm (3 nested loops for the time dimension) and the compiler constraints, we have had to manually unroll the time dimension. In this dimension, if the wavelet transform is applied twice, the first iteration will be applied over 20 frames (for blocks of 16 frames plus 4 frames needed for the overlapped wavelet transform), and the second iteration over 8 frames (the half of first iteration).

Note that, with the utilization of SSE, the time loop is only executed three times for the first application of the FWT, since each iteration computes four low-pass and four high-pass coefficients. For the second application of the FWT, just one iteration will suffice to compute all coefficients.

Another feasible optimization is data prefetching, which improves the performance due to the accelerated data delivery. In this way, data prefetching can hide the memory latency in part, if we predict which memory page our program will request next, we can fetch that page into cache (if it is not already in cache) before the program asks for it. In our wavelet transform algorithms, it is necessary to reference many pixels for computing the wavelets coefficients, and we can predict what are the next pixels that it can be needed, in order to drop down the latency, because the computation of the wavelet transform follows a specific pattern which depends on the mother wavelet function chosen it, as we can observe in figure 2.

3.3 Columns Vectorization

In the 3D-FWT, the wavelet is applied in the x, y and $time$ dimensions. In previous sections, we have analyzed the time dimension and we have applied the SSE vectorization by hand, loop unrolling and data prefetching. In the x dimension, the wavelet transform is applied successively for all rows of each frame. As the video sequence is stored in

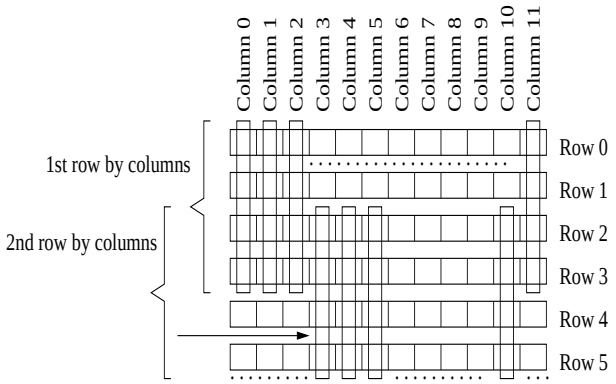

Figure 4. Columns vectorization

memory following a row order, spatial locality is exploited when the transform is applied in this dimension. The main problem regarding memory appears when the transform is applied in the y dimension. Pixels from successive rows are needed to compute coefficients of each column of y dimension, causing many cache misses even for the blocking version of the algorithm (for this version, L1 data cache still presents a high number of misses).

In this Section, we present "columns vectorization" as an effective way of applying the transform for the y dimension, exploiting the locality of references and the fact that the transform was already applied for the x dimension.

As the wavelet transform is applied by rows in the x dimension, in order to compute the first coefficient for the y dimension, only the resulting wavelet coefficients of the first four rows are needed, since each coefficient of the Daub-4 mother function depends on four pixels, as we can observe in Figure 2. Then, in order to compute a new row for the y dimension two more rows of wavelet coefficients of the x dimension are needed.

Figure 4 shows an example for a piece of frame of 6 rows by 12 columns. Once the wavelet transform is applied for the four first rows in the x dimension, it can be applied for the first row in the y dimension (i.e. in order to compute a coefficient, values obtained for rows 0,1,2,3 are needed). Furthermore, this computation is carried out using SSE extensions (4 coefficients fit in a XMM register). The second row in the y dimension depends on the rows 2, 3, 4 and 5. Therefore, only two new rows in the x dimension are necessary.

4 Experimental Results

The evaluation has been carried out on a 1GHz-Intel Pentium-III processor with 512 Mbytes of RAM. The main properties of the memory hierarchy are summarized in table 1. The operating system was Linux 2.2.14. The programs have been written in the C programming language.

TLBs	L1 instr TLB, 4K page, 4-way, 32 entries L1 data TLB, 4K page, 4-way, 64 entries
Level 1	L1 instr cache, 16 KB, 4-way, 32 byte line L1 data cache, 16 KB, 4-way, 32 byte line
Level 2	L2 unified cache, 256 KB, 8-way, 32 byte line
Level 3	512 Mbytes DRAM

Table 1. Description of the memory hierarchy

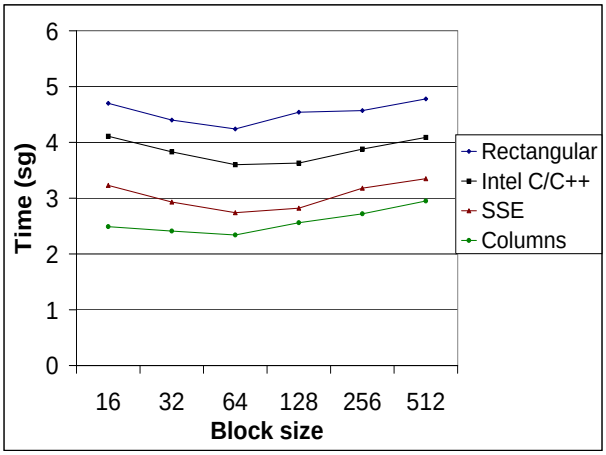

Figure 5. Execution Time of the 3D-FWT with different optimizations for Heart video sequence

Performance has been measured through the monitoring counters available in the P6 processor family. The Intel Pentium-series processors include a 64-bit cycle counter, and two 40-bit event counters, with a list of events and additional semantics that depend on the particular processor. We have used a library, Rabbit (v.2.0.1) [16], to read and manipulate Intel processor hardware event counters in C under the Linux operating system.

We have compared execution time consumed by the 3D-wavelet transform combining the different optimizations presented in section 2.1 and with the original 3D-FWT rectangular overlapped approach [8], on a $heart$ video medical sequence of 64 frames of $512x512$ pixels coded in gray scale (8 bits per pixel).

Figure 5 shows the execution time obtained with the fast wavelet transform to compute 64 frames of $512x512$ pixels for the rectangular blocking overlapped approach ($Rectangular$) compiled with the gcc/gnu compiler. $Intel\ C/C++$ represents the same blocking rectangular approach compiled with the Intel Compiler [13]. SSE includes SSE vectorization by hand as well as loop unrolling and data prefetching, all of them for the time dimension. Finally, $Columns$ includes Columns Vectorization and the SSE vectorization by hand in the computation of wavelet

coefficients for the y dimension . Results are presented for different block sizes, from $512x16x16$ to $512x512x16$.

First of all, we can observe that each new optimization clearly reduces the execution time of previous approaches for all configurations. The optimal block size ($512x64x16$) is maintained in all approaches. For this block size, the version just compiled with the Intel C/C++ obtains a speedup of 1.18 with respect the one compiled with gnu/gcc. From now on, we will refer to the Intel C/C++ version as the baseline, since it just represents our previous proposal recompiled with a better compiler. Adding SSE extensions along with prefetching and loop unrolling obtain a speedup of 1.31 regarding to the baseline and Columns vectorization provides a speedup of 1.54. It is important to note that all of these optimizations in the algorithm maintain the same compression rate and quality as the rectangular overlapped approach, which confirms the potential of these methods.

The results in Intel C/C++ are obtained with the option $-tpp6$ which generates code optimized for Pentium III processors and the advantages of the new compiler which improves the original execution time. In addition, we have enabled the automatic vectorization with the options $-xK$ and $-axK$, which generates code specialized for Streaming SIMD extensions, and, although the execution times are better than in the original rectangular overlapped approach (i.e. that compiled with gnu/gcc), they are worse than without automatic vectorization for the Intel C/C++ compiler. The reason of the decrease on performance experienced with automatic vectorization is that the vectorization of the Wavelet Transform is tricky, i.e. it has to be carefully applied to the computations that could obtain benefit from it. Remember that there are three nested loops and, for instance, vectorizing the innermost loop does not provide any benefit. Thus, manually vectorizing the code, as we propose in this work is, so far, the best option for achieving benefits when SIMD extensions are applied in the Wavelet Transform.

In SSE optimizations, performance benefits come from three different ways: first, the utilizations of SSE extensions for the time dimension, second, the effect of loop unrolling to increase Instruction Level Parallelism, and third, the effect of data prefetching. In particular, at the same time as 4 wavelet coefficients are being calculated, pixels needed for the next coefficients are being prefetched.

Finally, with $Columns$ optimization, execution times are reduced significantly for all configurations. This is due to the better exploitation of the temporal and spatial locality of the cache memory. We also manually vectorized the computations of the y dimension without the $Column$ optimization (i.e, coefficients in the y dimension are computed after the wavelet has been completed for the whole x dimension). This optimization does not provide any performance benefit, thus, as mentioned before, SSE extensions provide

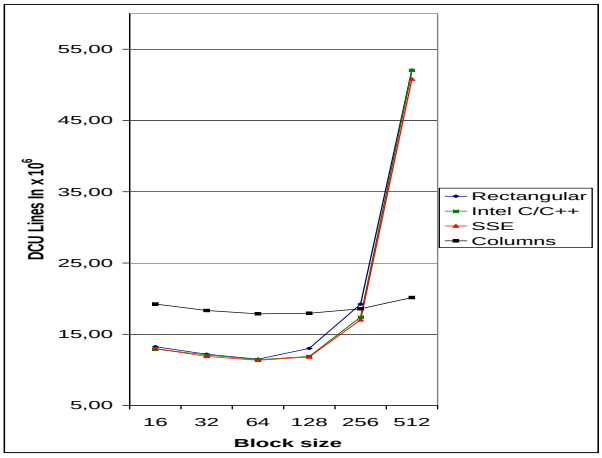

Figure 6. DCU Lines In of different approaches for Heart video sequence

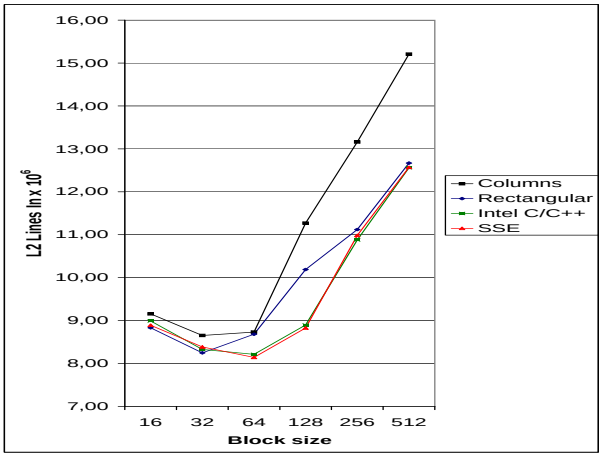

Figure 7. L2 Lines In of different approaches for Heart video sequence

benefit for the FWT only if they are applied carefully, which involves sometimes smart code reordering.

In order to gain some insight about the speedups obtained by the previous approaches, Figures 6 and 7 present the memory cache behavior for the heart video sequence. We measure this behavior using Data Cache Unit (DCU) Lines In and L2 Lines In events of the performance counters, which represent the number of lines allocated in the L1 Data Cache and the L2 cache respectively (i.e. the number of accesses that miss in each cache respectively).

It can be observed that the $IntelC/C++$ approach allocates less number of L1 and L2 lines than the rectangular overlapped approach for all configurations, justifying the decrease in the execution time. With respect to SSE vectorization by hand, we can observe that in most of configurations, this approach produces less number of L1 and

L2 misses than the Intel C/C++ but the difference is not very significant. Note that the main benefit provided by SSE optimizations come from the reduction in the number of Floating Point Instructions, as we can observe in Figure 8, due to the manual vectorization. Applying SSE extensions do not reduce the overall number of FP operations, which only depends on the algorithm, but it does reduce the number of FP instructions[1], since operations are performed in parallel in single SIMD instructions. Thus, what we are exploiting is $Data\ Level\ Parallelism$. Furthermore, the benefit provided by data prefetching cannot be measured using the number of L1 or L2 misses, since prefetching instructions does cause cache misses, but data is prefetched ahead enough so that misses do not make dependent instructions to wait on the processor.

Finally, in the Columns approach, there is a significant increment in the number of lines allocated in L1 and L2 caches, compared with previous approaches, although the columns vectorization exploit better the spatial and temporal locality for the calculation of x and y dimension. This increment in L1 and L2 misses is due to an implementation issue. When columns vectorization is applied, two rows are generated for the y dimension, a low-pass and a high-pass. High pass coefficients must be saved in other memory location different from the frame itself in order not to delete the original pixels, that are already needed for the rest of x computations. This increases the number of memory lines used for the transform (the original ones plus those needed for the temporal location of high-pass coefficients). Also, due to data movements back and forth to this temporal locations, locality is not so exploited, affecting the final performance of memory operations.

However, even with this problem with the memory instructions, the execution times have been drastically reduced for all configurations. This reduction is due by two reasons. First, since this optimization is built in top of the previous ones (the original blocking, prefetching and so on), the original 3D-FWT is not so memory bound, even with the latter memory "inefficiency". Second, since the algorithm is not memory bound, and following the Amdahl Law, any optimization in the computation side will provide a great impact in performance. Note that with the $Columns$ approach, the number of FP instructions executed have dropped spectacularly, 53%, 48% and 34% compared to Rectangular compiled with gnu/gcc, rectangular compiled with Intel C/C++ and SSE vectorization by hand respectively as we can see in Figure 8. This reduction in the number of instructions comes, again, through the exploitation of Data Level Parallelism, achieved by manually vectorizing computation in the y dimension.

[1]Recall that, in general, SSE optimizations maintain the number of FP operations but reduces the number of instructions by collapsing four operations in one multimedia instruction

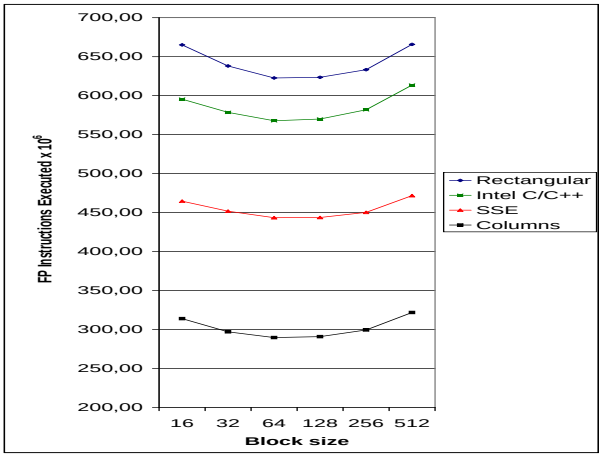

Figure 8. Floating point operations executed with different approaches for Heart video sequence

5 Conclusions

In this work, we have focused on reducing the execution time of the 3D-Fast Wavelet Transform when it is applied to code medical video. We have presented four proposals. First, we proposed and evaluated both the automatic and manually vectorization SSE, that exploits Data Level Parallelism by collapsing FP operations on single SIMD instructions. We have shown that the native compiler, Intel C/C++, is not able to obtain performance benefits through automatic optimizations and we have proposed several modifications on the algorithm that provide significant benefits by vectorizing computations in y and $time$ dimensions. Third, we have manually unrolled the time dimension loop and inserted prefetching instructions, in order to both reduce the impact of cache misses and exploit Instruction Level Parallelism. Fourth, we proposed and evaluated the columns vectorization in the y dimension, in order to reduce the floating point instructions and the memory accesses exploiting the spatial and temporal locality of the memory hierarchy.

Results show a speedup of 1.81 over the rectangular overlapped wavelet transform (compiled with gnu/gcc) and 1.54 when compared to the rectangular overlapped wavelet transform compiled with the Intel C/C++ compiler. Furthermore, all presented approaches maintain the video quality and the compression ratio of the original encoder.

6. Acknowledgments

This work has been partially supported by the Spanish CICYT under grant TIC2000-1151-C07-03. The video sequences have been donated by the Hospital Recoletas (Al-

bacete, Spain). We are grateful to the reviewers for their valuable comments.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. *In Proceedings of Supercomputing*, November 2000.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. M. Kenney, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. *Tech. Report CS-90-105, (LAPACK Working Note #20), Univ. Of Tennessee, Knoxville*, 1990.

[3] M. Antonini and M. Barlaud. Image coding using wavelet transform. *IEEE Transactions on Image Processing*, 1(2):205–220, April 1992.

[4] S. Battista, F. Casalino, and C. Lande. MPEG-4: A multimedia standard for the third millenium, part 1. *IEEE Multimedia*, 6(4):74–83, October 1999.

[5] S. Battista, F. Casalino, and C. Lande. MPEG-4: A multimedia standard for the third millenium, part 2. *IEEE Multimedia*, 7(1):76–84, January 2000.

[6] G. Bernab´e, J. Gonz´alez, J. M. Garc´ıa, and J. Duato. A new lossy 3-d wavelet transform for high-quality compression of medical video. *Proc. of IEEE EMBS International Conference on Information Technology Applications in Biomedicine*, pages 226–231, November 2000.

[7] G. Bernab´e, J. Gonz´alez, J. M. Garc´ıa, and J. Duato. Enhancing the entropy encoder of a 3d-fwt for high-quality compression of medical video. *Proc. of IEEE International Symposium for Intelligent Signal Processing and Communication Systems*, November 2001.

[8] G. Bernab´e, J. Gonz´alez, J. M. Garc´ıa, and J. Duato. Memory conscious 3d wavelet transform. *Proceedings of the 28th Euromicro Conference. Multimedia and Telecommunications. Dortmund, Germany*, September 2002.

[9] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium IV processor-based systems. *Available at http://developer.intel.com/*.

[10] Y. Chen and W. A. Pearlman. Three-dimensional subband coding of video using the zero-tree method. *Proc. of SPIE-Visual Communications and Image Processing*, pages 1302–1310, March 1996.

[11] G. Conte, S. Tommesani, and F. Zanichelli. The long and winding road to high-perfomance image processing with MMX/SSE. *Proceedings of the Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, 2000.

[12] Intel Corporation. Ia-32 Intel architecture software developer's manual. *Available at http://developer.intel.com/*.

[13] Intel Corporation. Intel C/C++ compiler for Linux. *Available at http://www.intel.com/software/products/compiler/c50/linux*.

[14] I. Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992.

[15] J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprogram. *ACM Trans. Math. Soft*, 14:1–17, 1988.

[16] D. Heller. Rabbit: A perfomance counters library for Intel/Amd processors and Linux. *Available at http://www.scl.ameslab.gov/Projects/Rabbit/*.

[17] M. L. Hilton, B. D. Jawerth, and A. Sengupta. Compressing still and moving images with wavelets. *Multimedia Systems*, 2(3), 1994.

[18] B.-J. Kim and W. A. Pearlman. An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (SPIHT). *Proceedings of Data Compression Conference*, 1997.

[19] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache perfomance and optimizations of blocked algorithms. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, April 1991.

[20] O. Lempel, A. Peleg, and U. Weiser. Intel's MMX technology - a new instruction set. *Proccedings of 42nd IEEE Computer Society International Conference*, 1997.

[21] A. S. Lewis and G. Knowles. Image compression using the 2-d wavelet transform. *IEEE Transactions on Image Processing*, 1(2):244–256, April 1992.

[22] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine patitioning. *In Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[23] M. W. Marcellin, M. J. Gormish, A. Bilgin, and M. P. Boliek. An overview of JPEG-2000. *Proceedings of Data Compression Conference*, March 2000.

[24] S. Muraki. Approximation and rendering of volume data using wavelet transforms. *Proceedings of Visualization*, pages 21–28, October 1992.

[25] S. Muraki. Multiscale volume representation by a dog wavelet. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):109–116, June 1995.

[26] L. Nachtergaele, G. Lafruit, J. Bormans, and I. Bolsens. Fast software implementation of the MPEG-4 reversible integer wavelet transform on pentium mmx, sharc adsp and trimedia tm1000. *Proccedings of Packet Video*, 2000.

[27] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media isa extensions. *International Symposium on Computer Architecture*, May 1999.

[28] D. Santa-Cruz and T. Ebrahimi. A study of JPEG 2000 still image coding versus others standards. *Proc. of the X European Signal Processing Conference*, September 2000.

[29] J. M. Shapiro. Embedded image coding using zerotrees of wavelets coefficients. *IEEE Transcations on Signal Processing*, 41(12):3445–3462, December 1993.

[30] G. Strang and T. Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1997.

[31] S. Thakkar and T. Huff. Internet streaming SIMD extensions. *IEEE Computer*, 32:26–34, 1999.

[32] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

[33] M. J. Wolfe. *High Perfomance Compilers for Parallel Computer*. Addison-Wesley Publishing Company, 1996.