Real-Time 3-D Wavelet Lifting

David Barina Pavel Zemcik

Faculty of Information Technology
Brno University of Technology
Bozetechova 1/2, Brno
Czech Republic
{ibarina,zemcik}@fit.vutbr.cz

ABSTRACT

This work presents a fast streaming unit for computing a 3-D discrete wavelet transform. The unit can continuously consume source data and instantly produce resulting coefficients. Considering this approach, every input as well as output sample is visited only once. The streaming unit can be further improved by exploiting suitable SIMD instruction set. Depending on the platform, the proposed method reaches speedup 11× and 8× compared to the naive implementation. The measurements presented in the paper confirm the linear theoretical complexity of the transform. Our method requires a constant amount of time to transform a sample independently of the data size.

Keywords
wavelet transform, volumetric data, real-time processing

1 INTRODUCTION

Discrete wavelet transform (DWT) is mathematical tool that uses discretely sampled wavelets to decompose input data into several scales. Regarding multi-dimensional signals, every such scale is also split into several directionally selective subbands. The transform is used as the basis of many sophisticated algorithms and applications. A forward direction of the transform is composed of several consecutive levels of signal decompositions. These can be briefly understood as convolutions with two complementary FIR filters – low-pass and high-pass one. An inverse direction of the transform has a symmetric nature to the forward one.

Considering the number of arithmetic operations, a lifting scheme [12] is often the most efficient way for computing the discrete wavelet transform. The entire calculation consists of several lifting steps. These steps alternately update odd and even intermediate results.

This paper focuses on the Cohen-Daubechies-Feauveau (CDF) 9/7 wavelet [11], which is often used for image compression (e.g., JPEG 2000 standard). The resulting coefficients using this wavelet can be computed by the convolution with two FIR filters, one with 7 and the other with 9 real-valued coefficients. Daubechies *et al.* demonstrated that the transform employing this wavelet can be computed in four successive lifting steps.

The 1-D discrete wavelet transform can be straightforwardly extended to transform 3-D signals. A separated decomposition along each axis is commonly used. The applications of 3-D DWT include video coding, medical data compression, video watermarking, medical image enhancement, or segmentation of medical volumes.

This paper presents SIMD-accelerated single-loop algorithm for 3-D discrete wavelet transform computation. Considering this proposed algorithm, every source memory element is visited only once while the resulting output coefficients are instantly produced.

This paper discusses implementation of 3-D DWT implemented over single-precision floating-point format (referred to as binary32 in IEEE 754 standard). As indicated above, the CDF 9/7 wavelet was employed. Only a single level of the transform is considered wherein subband coefficients (LLL, LLH, LHL, LHH, HLL, HLH, HHL, HHH) are kept interlaced in an output memory area. All the methods presented in this paper are evaluated using mainstream PCs with Intel x86 CPUs. The SIMD-accelerated methods was coded using the Streaming SIMD Extensions (SSE) instruction set. Intel Core2 Quad Q9000 running at 2.0 GHz and AMD Opteron 2380 running at 2.5 GHz were used in the tests below. Intel CPU has 32 KiB of level 1 data cache and 3 MiB of level 2 shared cache (two cores share one cache unit). In contrast, AMD CPU has 64 KiB of level 1 data, 512 KiB of level 2 cache per core and 6 MiB of level 3 shared cache (all four cores share one unit). All the algorithms below were implemented in the C language, using the SSE compiler intrinsics.[1] In all cases, a 64-bit code compiled using GCC 4.8.1 with `-O3` flag was used.

The rest of the paper is organized as follows. Section 2 discusses the state of the art. Especially, the lifting scheme and the single-loop approach is described here. Section 3 proposes a novel SIMD-accelerated single-loop algorithm for computing the discrete wavelet transform in 3-D. Finally, Section 4 summarizes the paper and outlines the future work.

2 RELATED WORK

On conventional architectures, the lifting scheme [20, 12] is the most efficient scheme for computing the discrete wavelet transform. Any discrete wavelet transform can be factorized into a sequence of N pairs of lifting steps. These steps are denoted as the predict P_n and update U_n convolution operators where each P_n corresponds to an FIR filter $p_i^{(n)}$ and each U_n to a filter $u_i^{(n)}$. The lifting factorization is not generally unique. This non-uniqueness can be exploited to maintain the symmetry of lifting steps in case of symmetric DWT filters.

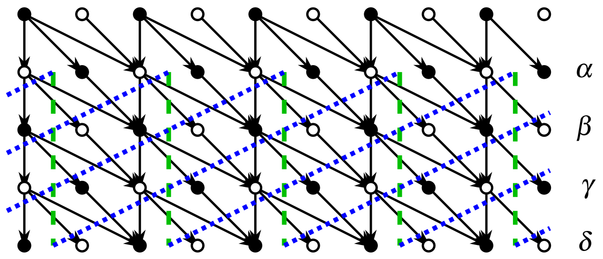


Figure 1: Lifting scheme of CDF 9/7 wavelet. Areas between dashed green lines represent the vertical and dotted blue lines the diagonal vectorization.

In [12], Daubechies *et al.* demonstrated an example of CDF 9/7 transform factorization. This lifting scheme consists of four lifting steps (two pairs). After these steps, an additional scaling of the coefficients is performed. The individual lifting steps use 2-tap symmetric filters. When evaluating this lifting scheme, some intermediate results can be appropriately shared between neighbouring coefficients. The original calculation is presented as an in-place algorithm, which means the transform might be calculated in a place of the input signal. However, this formulation is not advantageous especially due to a) a complicated border treatment (e.g., symmetric extension), b) a tendency to evicting the intermediate results from the CPU cache.

The entire calculation of CDF 9/7 DWT (without scaling) is depicted in Figure 1. In this figure, the $\alpha, \beta, \gamma, \delta$ are real constants specific to CDF 9/7 transform. Formally, the forward transform in the referred figure can be expressed by the dual polyphase matrix

$$\tilde{P}(z) = \begin{bmatrix} 1 & \alpha\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta\left(1+z\right) & 1 \end{bmatrix} \begin{bmatrix} 1 & \gamma\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \delta\left(1+z\right) & 1 \end{bmatrix} \begin{bmatrix} \zeta & 0 \\ 0 & 1/\zeta \end{bmatrix}. \quad (1)$$

The ζ is called the scaling constant.

A naive approach of the lifting scheme evaluation can directly follow the lifting steps. Using this strategy, an entire input signal is updated several times using the predict P_n and update U_n convolution operators. This method will hereinafter be referred to as a horizontal vectorization. Chrysafis *et al.* [10] addressed the problem of online (pipelined, line-based) implementations of the lifting scheme. However, their approach was very general and it is not focused on a real implementation. In contrast, Kutil [16] presented an useful implementation of CDF 9/7 lifting employing SSE instruction set. He splits the lifting scheme into vertical areas (see Figure 1). Thus, his method is below referred to as a vertical vectorization. Due to data dependencies between adjacent vertical areas, the vertical vectorization cannot be directly parallelized. For this reason, we have presented a diagonal vectorization (Figure 1) of the scheme in [1]. This vectorization allows the use of SIMD instructions directly without buffering of the coefficients into blocks.

In the image processing, a fast implementation of 2-D DWT is a widely examined area. At this place, we will briefly revise important works. Chatterjee *et al.* [7] proposed two techniques intended for an optimization of the 2-D transform. The first one interleaves the operations of several 1-D transforms on multiple columns. The second one modifies the memory layout so that each sub-band is situated there contiguously. Chaver *et al.* [8] pipelined computation in a vertical direction on subsequent rows. This is similar to pipelined strategies mentioned above in case of 1-D transform. Other similar approaches can be found in [8] and [9], where the authors vectorized a transform using an approach similar to the one described in [16]. Considering the CPU cache, the authors of [18] as well as [19] proposed several techniques for reducing cache misses.

However, the most important work was done by Kutil in [16]. The author fused vertical and horizontal 1-D filterings into a single loop (from here the single-loop approach). This was partly done in the pipelined 2-D approaches above, although not performed on the whole image at once. Let us revise the Kutil's work in more detail. The original single-loop approach is based on the vertical vectorization. One step of this vectorization requires two values to perform a single iteration. Consequently, the approach needs to perform two filterings on two consecutive rows at once. For each row, two

[1] `http://www.fit.vutbr.cz/research/prod/?id=211`

coefficients are produced, which makes four values in total. The vertical vectorization algorithm passes four values from one iteration into the other for each row. Finally, the Kutil's approach needs to pass four rows between merged iterations. Moreover, using the prime stride between subsequent rows of the image was suggested in his paper.

In [2], we have extracted the minimal core of the original single-loop approach leading to a novel single-loop core approach. This core can be built over the vertical as well as the diagonal vectorization. Several such cores can be further fused together in order to exploit a suitable SIMD instruction set. The core approach completely eliminated the problems of a complicated border treatment used by the original approach. This fact is caused by the separation of data accesses from the intrinsic transform. The influence of CPU caches on the 2-D transform was also studies in this work. In the end, we have proposed a parallelization of the SIMD-vectorized core approach that led to additional speedups.

Furthermore, many papers exist in the field of an efficient 3-D DWT implementation. Let us to mention the most significant works. In [4], Bernabe *et al.* presented two methods reducing the 3-D transform execution time. However, in both of these methods, they employed the convolution scheme that does not take advantage from benefits of the lifting scheme. In their first method, they split the original 3-D volume into several independent sub-volumes. Thus, they have performed several independent transforms[2] which is different and much easier task comparing to what we are dealing with in this paper. On the other hand, such method benefits from a small working set of the transformed data. The independent transforms can be further applied in an overlapped and non-overlapped manner. The second method is just a modification of the first one where the independent transforms are applied on cuboid subvolumes instead of cubes. The method should better exploit the memory locality occurred due to their particular memory layout. The authors also exploited a fine-grained parallelism by vectorizing loops using the SSE instructions. Unfortunately, their methods are far away from the single-loop and also the single-loop core approaches. Finally, the authors reported 5× speedup compared to the non-tuned implementation. In [5], Bernabe *et al.* published a subsequent work. They exploited advantages of a parallel processing using multiple threads. The work is closely focused on hyperthreading (HT) technology. However, the principles of the methods employed remained the same as in previous paper.

[2] introducing a block effect

In [17], Lopez *et al.* introduced a fast frame-based 3-D DWT video encoder with low memory usage. The authors used the convolution scheme. In their approach, the video frames are continuously consumed by the 2-D DWT algorithm. Then, this transformed frame is stored in a buffer. Unfortunately, this buffer must be able to hold as many frames as the number of taps for the FIR filter in the temporal direction. Although their encoder reduced memory as well as computational requirements compared to the original 3D-SPIHT algorithm, it is still far away from the true 3-D pipelined transform.

In another two papers [15, 3], the authors applied separatelly 2-D spatial and 1-D temporal transform. Both of the works deal with a video compression. As in the previous case, their approaches still need several input frames to be accumulated in a buffer in order to filter the frames along the third dimension.

The implementation of 3-D DWT was also studied on modern programmable graphics cards (GPUs). For instance, the authors of [14] and [6] used the convolution scheme keeping transforms along three dimensions separated.

As it can be seen, the problem of the efficient 3-D discrete wavelet transform implementation on conventional PCs was studied to some extent. However, we see several gaps which can allow for additional speedups.

3 PROPOSED METHOD

In this section, several 3-D transform techniques are proposed. At the beginning, we discuss a problem of appropriate choice of row and slice strides (steps between consecutive rows or slices). Then, we gradually extend the single-loop core approach to three dimensions.

3.1 Naive Approaches

Several works, e.g. [16], studied a performance degradation of 2-D transform in the vertical filtering caused by poor choice of the row stride. This degradation is especially significant when the stride is a power of two. The same problem occurs in 3-D. Fortunately, the problem is eliminated by using the single-loop approach in which there is no separated horizontal and vertical filtering.

Based on our previous works [1, 2], we have implemented the 3-D transforms in two naive ways. Firstly, a volume was transformed using the horizontal vectorization separately along each dimension. This is the most naive method employing the lifting scheme. Secondly, we have implemented this transform using the vertical vectorization. This method suffers from fewer CPU cache related problems than the one employing the horizontal vectorization. However, both of them exhibit huge amount of cache misses as soon as a size of

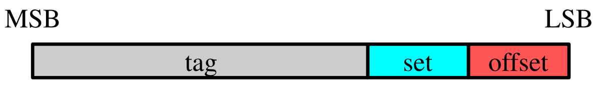


Figure 2: Address structure in relation to the CPU cache. The sizes of the individual parts depend on the particular architecture.

the data exceeds a size of the CPU cache. In both implementations, the transform in the first dimension is computed coupled with copying the data into a destination area. This is followed by a real in-place computation for the remaining two dimensions.

The cache of the considered CPUs consists of many 64-bytes buckets called cache lines. They are divided into several sets according to a associativity of the cache (e.g., four sets for 4-way associativity). Considering such CPU cache, a virtual address is split into three parts. Low six bits specify offset in a cache line. Few upper bits specify the associativity set of the cache. The rest of bits represent a tag stored for each individual cache line. Such address structure is outlined in Figure 2. The details can be found in [13].

We have measured the performance for both of the described naive methods in combination with unchanged as well as modified strides. Note that we are talking about the row stride as well as the slice stride. The unchanged stride means that rows and slices are placed without gaps densely one by one. In relation to the second choice, we have modified the strides in the following way. Low six bits of the addresses correspond to offsets in cache lines and thus do not directly influence the selection of the cache set. These are set to zero. The other bits are changed to the next highest prime number (the "set" and "tag" in Figure 2). Although a choice of the next highest odd number is sufficient since any odd number is coprime with any power of two. The performance comparison is shown in Figure 5.

As it can be seen, the performance of unchanged stride is unstable. It exhibits poor properties especially on the power-of-two strides. However, the data has smaller memory footprint. So, if the stride length does not hit the inappropriate value, the entire transform runs faster than in the case with modified stride. In the rest of this paper, we will continue using the modified stride (i.e. prime stride).

As an intermediate step towards the full 3-D single-loop transform, we have experimented with highly optimized 2-D transforms followed by a separated filtering in the third dimension. The implementation of the optimized 2-D transform is described in our previous paper [2]. Essentially, this transform is computed "out of place" in the single loop by a single-threaded SIMD-optimized 4×4 core. Afterwards, the transform in the third dimension is computed "in place" using the vertical vectorization. For 4×4 core approach, the auxiliary buffer of a slice edge size times four has to be allocated. The buffer is used to transfer information between steps of the vertical vectorization along y axis direction. The comparison with the previous two methods is shown in Figure 6. We have plot only the implementations using the modified (stable) strides. The transform performed in the slices outperforms the two naive implementations practically for all sizes on the x axis.

3.2 True 3-D Approaches

Based on our work in [2], we have created two baseline implementations transforming the entire volume in the single loop. These implementations employing 2^3 cores – first built over the vertical and the second over the diagonal vectorization. Both of them process the data out of a place. The implementation with the diagonal core is inherently accelerated by SIMD instructions. The vertical implementation does not allow SIMD-optimizations at this stage.

In more detail, both 2^3 cores are composed from three parts. The first part loads the input data from a source memory area. The inner part performs a fragment of the transform computation. Finally, the last part stores the resulting coefficients into a destination area. The first and last parts treat data borders using the symmetric extensions.

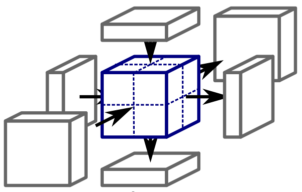


Figure 3: Illustration of 2^3 cubic core. Portions of auxiliary buffers to update are shown on each side.

Access to three auxiliary buffers is required during the inner part of the computation. See Figure 3. This most important part consists of three blocks performing calculations corresponding to the three dimensions. Each block updates the necessary intermediate results in the corresponding auxiliary buffer. This is followed by scaling of the output coefficients.

In the most generic variant, each 2-D auxiliary buffer has the same size like the corresponding volume side. The depth of each auxiliary buffer is 4 coefficients for the vertical vectorization or 12 coefficients for the diagonal one. See Figure 4. This memory consumption can be reduced using a appropriate processing order. When using the horizontal[3] order, it is not necessary to allocate the full side size buffers. It is sufficient to allocate only the following sizes. One full side size buffer for the first dimension. One edge size buffer for the second dimension. Finally, one point size buffer for the third

[3] raster scan

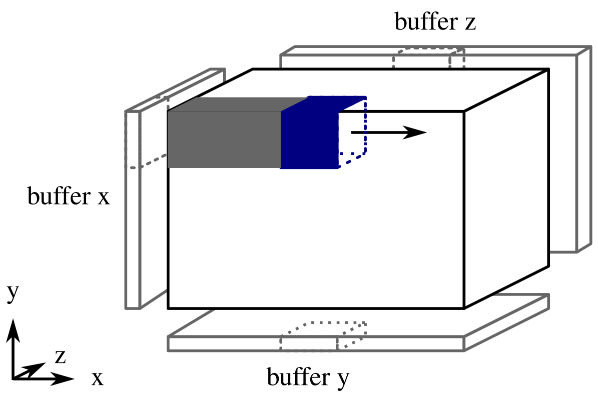


Figure 4: Complete processing of the volumetric data by the 3-D single-loop core. Auxiliary buffer for each dimension is shown on the sides of the volume.

dimension. For instance, considering the vertical vectorization and 2^3 core, it is sufficient to allocate buffers of total size $4 \cdot (s_x \cdot s_y + s_x \cdot 2 + 2 \cdot 2)$ elements, where s_x, s_y are sizes of the volume in first two dimensions.

Now, we take a closer look on the conjoined scaling. With the scaling constant ζ, the scaling of one pair of coefficients can be written as element-wise multiplication by a scaling vector

$$Z_1(x) = \begin{bmatrix} \zeta^{-1} & \zeta \end{bmatrix}, \tag{2}$$

where the coordinate $x \in \{0, 1\}$.

In 2-D case, the scaling of one 2×2 block of coefficients can be written as element-wise multiplication by a scaling matrix

$$Z_2(x,y) = Z_1(x) \cdot Z_1(y), \tag{3}$$

where the coordinates $x, y \in \{0, 1\}$, giving

$$Z_2(x,y) = \begin{bmatrix} \zeta^{-2} & 1 \\ 1 & \zeta^2 \end{bmatrix}. \tag{4}$$

Finally, in 3-D case, the scaling of one 2^3 box of coefficients can be written as element-wise multiplication by a scaling cube

$$Z_3(x,y,z) = Z_1(x) \cdot Z_1(y) \cdot Z_1(z), \tag{5}$$

where $x, y, z \in \{0, 1\}$. For better understanding, the slices through the z axis look like

$$\begin{aligned} Z_3(x,y,0) &= \begin{bmatrix} \zeta^{-2} & 1 \\ 1 & \zeta^2 \end{bmatrix} \cdot \zeta^{-1} &= \begin{bmatrix} \zeta^{-3} & \zeta^{-1} \\ \zeta^{-1} & \zeta \end{bmatrix} \\ Z_3(x,y,1) &= \begin{bmatrix} \zeta^{-2} & 1 \\ 1 & \zeta^2 \end{bmatrix} \cdot \zeta &= \begin{bmatrix} \zeta^{-1} & \zeta \\ \zeta & \zeta^3 \end{bmatrix} \end{aligned}. \tag{6}$$

We have compared the performance of the two 3-D core approaches to the previous methods. The result can be seen in Figure 7. Apparently, the 3-D approaches approximately above 1 megavoxel outperform all the previous methods.

	Intel Core2		AMD Opteron	
method	time	speedup	time	speedup
naive horiz.	159.8	1.0	105.7	1.0
naive vert.	100.1	1.6	73.5	1.4
vert. slice 4^2	53.8	2.9	41.0	2.5
vertical 2^3	23.3	6.8	21.7	4.7
diagonal 2^3	22.8	6.9	21.1	4.9
vertical 4^3	**13.5**	**11.7**	**12.9**	**8.0**

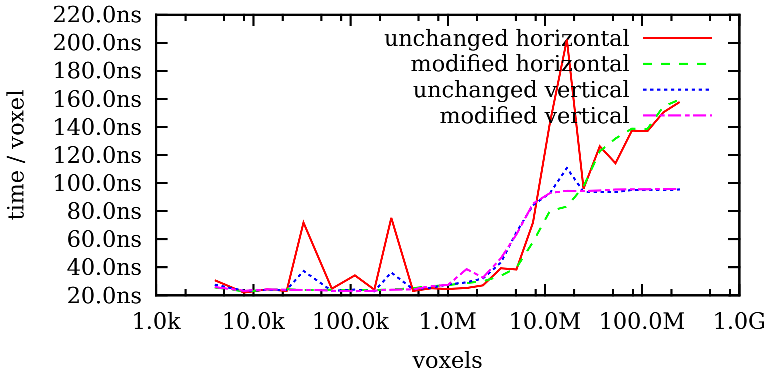
Table 1: Performance evaluation for large data. Best results are in bold.

Within the context of the vertical vectorization, utilization of SIMD instruction set is straightforward. In first two dimensions, 2×2 small 2^3 cores are merged together in analogous manner as in [2]. In the third dimension, there is no reason to increase the size of the core as SIMD can be used directly. In all three dimensions, the basic building block of the transform is 2×4 core. In this 2×4 core, four steps of the vertical vectorization are computed in parallel using SIMD instructions. The scaling of coefficients is performed together as the last step of the calculation. As a result, $4 \times 4 \times 2$ SIMD-vectorized core was formed.

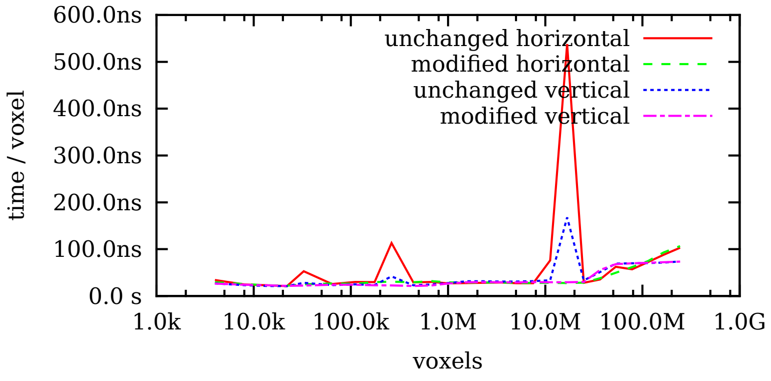
Although there is no reasonable justification, it can be tempting to build a compact 4^3 core as an analogy to the well-performing 4^2 counterpart. Such core is internally composed of two dependent $4 \times 4 \times 2$ sub-cores. However, such a connection may be slightly advantageous due to prefetching of intermediate results. Moreover, the implementation is more regular.

The final comparison is shown in Figure 8. The SIMD-optimized 3-D cores exhibit the best results. The 4^3 core slightly outperforms the baseline $4 \times 4 \times 2$ one. Above initial transients, all the single-loop approaches confirm the linear asymptotic complexity of the discrete wavelet transform.

Based on [2], the single-loop core method can be further accelerated utilizing the multi-threading. Table 1 summarizes performances and speedups for a volume of 238 megavoxels. The testing platforms are described in Section 1. The speedups are given against the separable method using the horizontal vectorization and the prime stride. The achieved processing time 13 nanoseconds per sample is roughly equivalent to process 37 frames per second with Full HD resolution (1920×1080 per frame). For $4 \times 4 \times 2$ core and any infinite video sequence, only two frames (the currently coded and the immediately preceding) have to be held in memory. The summarizing comparison of all significant approaches is shown in Figure 9.

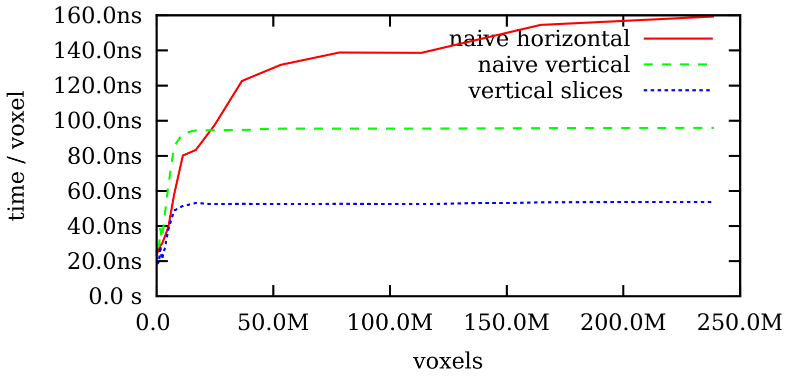


(a) Intel Core2

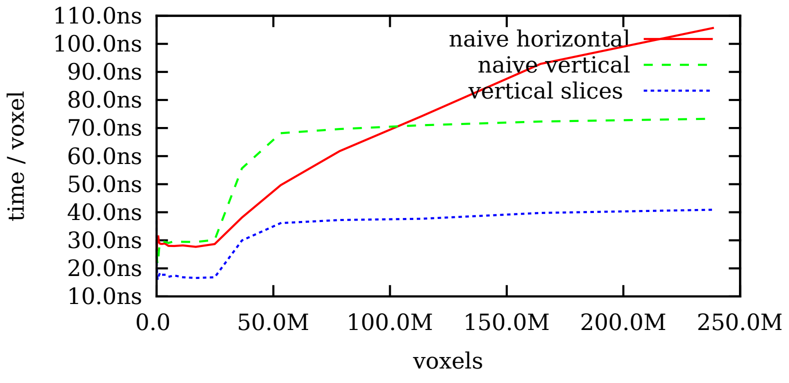


(b) AMD Opteron

Figure 5: Performance comparison of naive approaches with unchanged and prime strides.

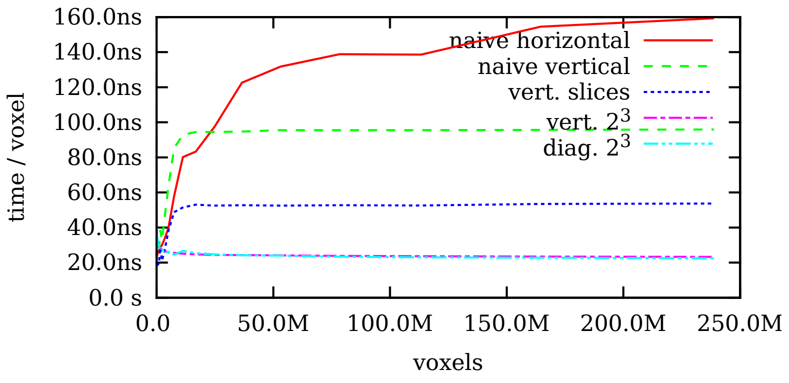


(a) Intel Core2

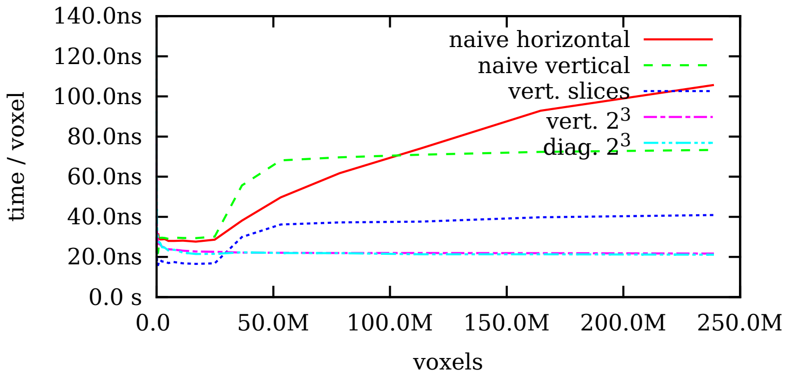


(b) AMD Opteron

Figure 6: Performance comparison of slicing 4^2 and naive approaches under the prime stride.

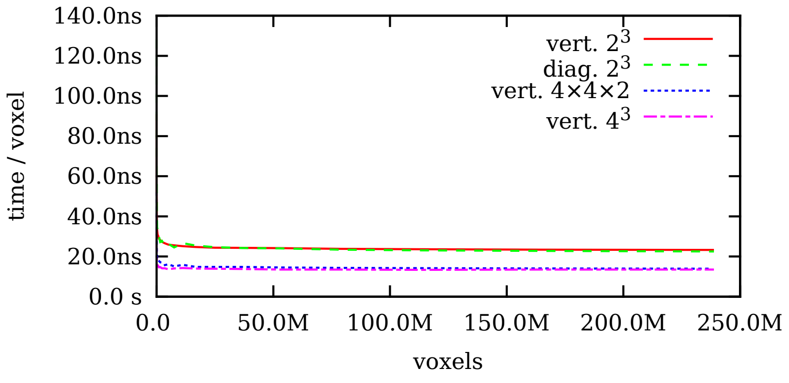


(a) Intel Core2

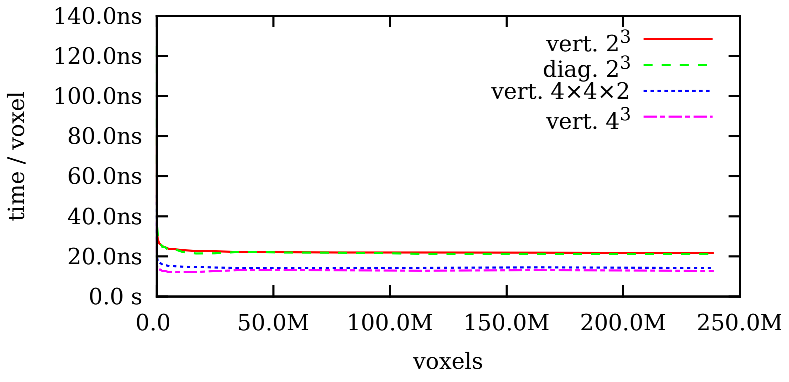


(b) AMD Opteron

Figure 7: Performance comparison of two baseline 3-D approaches with previous ones.

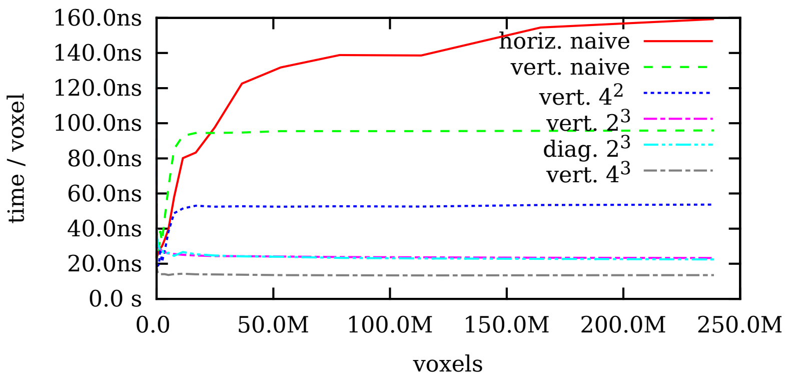


(a) Intel Core2

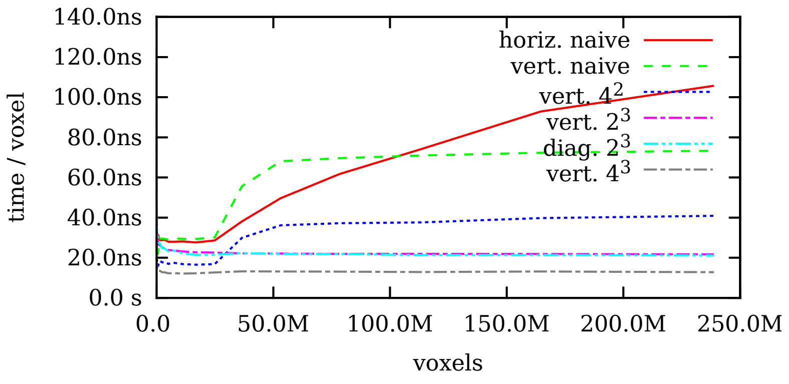


(b) AMD Opteron

Figure 8: Performance comparison of all 3-D approaches.



(a) Intel Core2



(b) AMD Opteron

Figure 9: Summarizing performance comparison of all approaches under the prime stride.

4 CONCLUSIONS

We have presented a true single-loop approach designed to transform 3-D data. The proposed method can be understood as a streaming unit which visits every input as well as output data element only once. The core can further be improved by exploiting suitable SIMD instruction set. The best of proposed methods reaches speedup $11\times$ on Intel and $8\times$ on AMD compared to the naive implementation. The measurements confirm the linear theoretical complexity of the transform in 3-D. In absolute numbers, the best method can transform a data sample in 13 nanoseconds what is roughly equivalent to transform 37 frames per second in Full HD.

As a future work, our methods can be parallelized by multithreading. For real applications, it can be necessary to perform several levels of such decomposition.

Acknowledgements

This work has been supported by the IT4Innovations Centre of Excellence (no. CZ.1.05/1.1.00/02.0070).

REFERENCES

[1] D. Barina and P. Zemcik. Minimum memory vectorisation of wavelet lifting. In *Advanced Concepts for Intelligent Vision Systems*, volume 8192 of *Lecture Notes in Computer Science*, pages 91–101. Springer, 2013. ISBN 978-3-319-02894-1. doi: 10.1007/978-3-319-02895-8_9.

[2] D. Barina and P. Zemcik. Vectorization and parallelization of 2-D wavelet lifting. *Journal of Real-Time Image Processing*, 2015. ISSN 1861-8200. doi: 10.1007/s11554-015-0486-6.

[3] E. Belyaev, K. Egiazarian, and M. Gabbouj. Low complexity bit-plane entropy coding for 3-D DWT-based video compression. In *Proceedings of SPIE*, volume 8304, 2012. doi: 10.1117/12.912017.

[4] G. Bernabe, J. Garcia, and J. Gonzalez. Reducing 3D fast wavelet transform execution time using blocking and the streaming SIMD extensions. *Journal of VLSI signal processing systems for signal, image and video technology*, 41(2): 209–223, 2005. ISSN 0922-5773. doi: 10.1007/s11265-005-6651-6.

[5] G. Bernabe, R. Fernandez, J. M. Garcia, M. E. Acacio, and J. Gonzalez. An efficient implementation of a 3D wavelet transform based encoder

on hyper-threading technology. *Parallel Computing*, 33(1):54–72, 2007. ISSN 0167-8191. doi: 10.1016/j.parco.2006.11.011.

[6] G. Bernabe, G. Guerrero, and J. Fernandez. CUDA and OpenCL implementations of 3D fast wavelet transform. In *IEEE Third Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4, Feb. 2012. doi: 10.1109/LASCAS.2012.6180318.

[7] S. Chatterjee and C. D. Brooks. Cache-efficient wavelet lifting in JPEG 2000. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*, volume 1, pages 797–800, 2002. doi: 10.1109/ICME.2002.1035902.

[8] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. Wavelet transform for large scale image processing on modern microprocessors. In *High Performance Computing for Computational Science – VECPAR 2002*, volume 2565 of *Lecture Notes in Computer Science*, pages 549–562. Springer, 2003. ISBN 978-3-540-00852-1. doi: 10.1007/3-540-36569-9_37.

[9] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. Vectorization of the 2D wavelet lifting transform using SIMD extensions. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 8, 2003. doi: 10.1109/IPDPS.2003.1213416.

[10] C. Chrysafis and A. Ortega. Minimum memory implementations of the lifting scheme. In *Proceedings of SPIE, Wavelet Applications in Signal and Image Processing VIII*, volume 4119 of *SPIE*, pages 313–324, 2000. doi: 10.1117/12.408615.

[11] A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45(5):485–560, 1992. ISSN 1097-0312. doi: 10.1002/cpa.3160450502.

[12] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, 4(3):247–269, 1998. ISSN 1069-5869. doi: 10.1007/BF02476026.

[13] U. Drepper. What every programmer should know about memory, 2007. URL `http://www.akkadia.org/drepper/cpumemory.pdf`.

[14] J. Franco, G. Bernabe, J. Fernandez, and M. Ujaldon. Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs. *Procedia Computer Science*, 1(1):1101–1110, 2010. ISSN 1877-0509. doi: 10.1016/j.procs.2010.04.122. ICCS 2010.

[15] V. Galiano, O. Lopez-Granado, M. Malumbres, and H. Migallon. Multicore-based 3D-DWT video encoder. *EURASIP Journal on Advances in Signal Processing*, 2013(1):84, 2013. doi: 10.1186/1687-6180-2013-84.

[16] R. Kutil. A single-loop approach to SIMD parallelization of 2-D wavelet lifting. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 413–420, 2006. ISBN 0-7695-2513-X. doi: 10.1109/PDP.2006.14.

[17] O. Lopez, M. Martinez-Rach, P. Pinol, M. Malumbres, and J. Oliver. A fast 3D-DWT video encoder with reduced memory usage suitable for IPTV. In *2010 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1337–1341, July 2010. doi: 10.1109/ICME.2010.5583570.

[18] P. Meerwald, R. Norcen, and A. Uhl. Cache issues with JPEG2000 wavelet lifting. In *Visual Communications and Image Processing (VCIP)*, volume 4671 of *SPIE*, pages 626–634, 2002.

[19] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. Improving the memory behavior of vertical filtering in the discrete wavelet transform. In *Proceedings of the 3rd conference on Computing frontiers (CF)*, pages 253–260. ACM, 2006. ISBN 1-59593-302-6. doi: 10.1145/1128022.1128056.

[20] W. Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Applied and Computational Harmonic Analysis*, 3 (2):186–200, 1996. ISSN 1063-5203.