Parallel Implementation of 2D Forward Discrete Wavelet Transform on multicore CPUs

Zdenek Prusa, Pavel Rajmic

Faculty of Electrical Engineering and Communication, Brno University of Technology
Email: zdenek.prusa@phd.feec.vutbr.cz, rajmic@feec.vutbr.cz

Abstract *– In this paper we present parallel implementation of 2D forward DTWT using C++ and Intel Threading Building Blocks library. Developed algorithm is described in detail and reached speedup is shown.*

The image is processed blockwise with certain overlap between blocks so that the result it equal to the result obtained by processing the image as a whole. Consequently, the blocking artifact is completely eliminated. The exact dimensions of the overlap is given by the developed algorithm. The algorithm works with any block size also the blocks do not have to have equal dimensions. The overlap given by the algorithm ensures that blocks can be processed independently. Necessary redundancy is discussed also.

1 Introduction

The discrete-time wavelet transform (DTWT) is a standard tool in the field of signal and image processing and it has many applications in signal analysis, compression [1], denoising [2], watermarking [3] and computer vision [4]. Aside from effective representation of signals by wavelet coefficients, wavelet transform also gained its popularity due to Mallat's pyramidal algorithm [5], which computes dyadic DTWT using recursive two channel (bi)orthogonal filter bank providing only linear computational complexity.

Mallat's pyramidal algorithm was modified so that it can be performed segmentwise on one-dimensional signals – SegDTWT [6]. This means that the input signal is divided into overlapping segments, each of them producing certain number of wavelet coefficients which can be combined to make DTWT of the whole signal. This algorithm is primarily suitable for real-time processing.

In this paper, we propose a generalization of this algorithm for two dimensional signals (images). In comparison to the one-dimensional version, the new algorithm can be simplified considering assumptions summarized in Section 3. As a result, DTWT of any image can be performed on independent segments and hence in parallel.

The requirements on the algorithm can be stated as follows:

1. **Segment size.** The algorithm must be able to process segments of arbitrary sizes (up to some minimum values).
2. **Segment position.** The algorithm must be applicable to any rectangular segment chosen from image.
3. **Order of computation.** The segments can be processed in arbitrary order.
4. **Type of wavelet filters.** Both orthogonal and biorthogonal filter banks can be used.

To achieve the requirements several modifications of the original algorithm need to be done.

2 Background and related work

At this moment several research papers has been published that deal with parallelization of DTWT on images. At some point they have to take into account some kind of segmentation of an image. Several approaches have been proposed but all of them have drawbacks. First of all, we are only interested in algorithms, which do not produce errors at the edges of segments. This fact leaves us with algorithms in which parallelism is present only in the form of separate processing of rows and columns in each decomposition step [7], or with algorithms that limit themselves to segments of dimensions equal to powers of two [8]. The dimension restriction makes dynamic task scheduling impossible and it can introduce substantial redundancy of the computation.

Most of the papers also aims at algorithm for a concrete device and/or concrete wavelet filters [9].

2.1 Classical DTWT algorithm

When performing DTWT of finite length signal, several samples beyond signal boundaries need to be known so that signal reconstruction is exact. Several types of boundary treatment can be used [10]. Usually the DTWT is required to be *non-expansive* (each step produces exactly half the number of coefficient than it was in the previous step). This is possible only on one of these cases:

- Periodic extension is used.
- Symmetric filters and corresponding symmetric extensions are used.
- Specialized matrix method is used [11].

All of these cases are limited in some way, so in the following text only *expansive* DTWT is considered. In such a case, the length of the vector of coefficients is given by the length of linear convolution of two finite-length signals.

The following algorithm is depicted in Fig. 1.

Algorithm 1:[Decomposing pyramid algorithm DTWT] Let $\boldsymbol{x}$ be a discrete input signal of length s, the two wavelet decomposition filters are defined highpass $\boldsymbol{g}$ (length $m_{\boldsymbol{g}}$) and lowpass $\boldsymbol{h}$ (length $m_{\boldsymbol{h}}$), J is a positive integer denoting the decomposition depth. Also, the type of boundary treatment has to be known:

1. Denote the input signal $\boldsymbol{x}$ by $\boldsymbol{a}_0$ and set $j = 0$ and $m = m_{\boldsymbol{g}}$.
2. One decomposition step:
   (a) *Extending the input vector.* Extend $\boldsymbol{a}_j$ from both the left and the right sides by $(m-1)$ samples, according to the type of boundary treatment.
   (b) *Filtering.* Convolve the extended signal with filter $\boldsymbol{g}$.
   (c) *Cropping.* Take from the result just its central part, so that the remaining "tails" on both the left and the right sides have the same length $(m-1)$ samples.
   (d) *Decimation.* Downsample the resultant vector. Denote the result by $\boldsymbol{d}_{j+1}$ and store it. Then set $m = m_{\boldsymbol{h}}$ and repeat items b) d), now with filter $\boldsymbol{h}$, denoting and storing the result as $\boldsymbol{a}_{j+1}$.
3. Increase j by one. If it now holds $j < J$, return to item 2, in the other cases the algorithm ends.

After Algorithm 1 has been finished, we have the wavelet coefficients stored in $J+1$ vectors (of different length) $\boldsymbol{a}_J, \boldsymbol{d}_J, \boldsymbol{d}_{J-1}, \ldots, \boldsymbol{d}_1$.

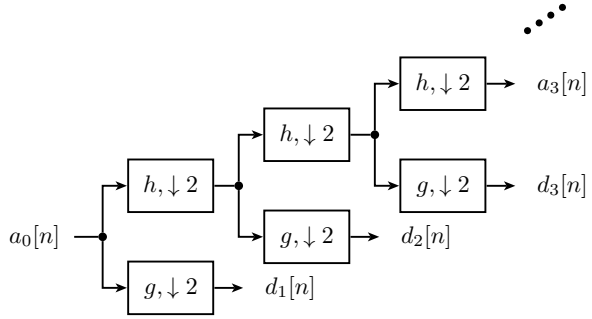

Figure 1: Iterated filter bank for pyramidal algorithm DTWT with $J = 3$.

2.2 SegDTWT algorithm

In this part, the SegDTWT algorithm version 1.0 from [6] is presented. This algorithm was modified later in [12], however these modifications were made mainly for processing acoustic signals in real time and they are not important when processing images. The SegDTWT algorithm was developed for orthogonal filter banks, but biorthogonal filters can also be used, zero padded to the same length.

The one-dimensional input signal $\boldsymbol{x}$ is divided into $S \geq 1$ segments of equal length s. The last one can be less long than s. To achieve correct follow-up of two sets of wavelet coefficients in decomposition level j it is necessary for two consecutive segments to be properly extended. It has been shown that two consecutive segments must have:

$$r(j) = (2^j - 1)(m-1) \tag{1}$$

input samples in common after they have been extended. This extension has to be divided into right extension of the first segment (of length R) and the left extension of the following segment (of length L) so that $r(j) = R + L$, however $R, L \geq 0$ cannot be chosen arbitrarily. The minimum suitable right extension of the n-th segment for $n = 1, 2, \ldots, S-2$ is

$$R_{\min}(n) = 2^J \text{ceil}\left(\frac{ns}{2^J}\right) - ns, \tag{2}$$

and the maximum left extension of $(n+1)$-th segment is

$$L_{\max}(n+1) = r(J) - R_{\min}(n). \tag{3}$$

Algorithm 2:[SegDTWT v. 1.0] Let the wavelet filters $\boldsymbol{g}, \boldsymbol{h}$ of length m, decomposition level J and boundary treatment be given. The input signal $\boldsymbol{x}$ is divided into segments of equal length $s \geq 2^J$ and the segments are denoted by ${}^1\boldsymbol{x}, {}^2\boldsymbol{x}, {}^3\boldsymbol{x}, \ldots$.

1. Set $n = 1$.
2. Read first segment, ${}^1\boldsymbol{x}$, label it as "current" and extend it from left by $r(J)$ zero samples.
3. **If** the current segment is the last one at the same time, compute DTWT of this segment using Algorithm 1 and finish.
4. Load $(n+1)$-th segment and label it as "next".
5. **If** the next segment is the last one:
   (a) Combine the current and the next segment, set this new segment as current (the current becomes last one).
   (b) Extend the current segment by $r(J)$ zero samples from the right.
   (c) Calculate DTWT of depth J from the extended current segment using the Algorithm 1 with modification in step 2(a), which will be: "*Modify the vector $\boldsymbol{a}_j$ at indexes $r(J-j)-m+2, \ldots, r(J-j)$, counted from right side of $\boldsymbol{a}_j$ according to chosen boundary treatment.*"

   Else
   (d) Determine $L_{\max}(n+1)$ for the next segment and $R_{\min}(n)$ for current segment using formulas (2) and (3).
   (e) Extend current segment from the right by $R_{\min}(n)$ samples from the next segment. Extend next segment from the left by $L_{\max(n+1)}$ samples from the current segment.
   (f) **If** the current segment is the first one, calculate the DTWT of depth J from the extended current segment using the algorithm 1 with modification in step 2(a), which will be: "*Modify the the vector $\boldsymbol{a}_j$ at indexes $r(J-j)-m+2, \ldots, r(J-j)$ according to chosen boundary treatment.*" **Else** calculate the DTWT of depth J from the extended current segment using the algorithm 1 with omitting step 2(a).

6. Modify the vectors containing the wavelet coefficients by trimming off a certain number of redundant coefficients from left side, specifically: on the j-th level, $j = 1, 2, \ldots, J-1$ trim off $r(J-k)$ coefficients.
7. **If** the current segment is last segment, trim off the vectors in the same manner as in previous step $r(J-k)$ but this time from the right.
8. Store the result as ${}^n\boldsymbol{a}_J, {}^n\boldsymbol{d}_J, {}^n\boldsymbol{d}_{J-1}, \ldots, {}^n\boldsymbol{d}_1$.
9. **If** the current segment is not the last one, set the next segment as current, increase n by 1 and go to item 4.

2.3 Separability in two dimensions

The simplest way how to perform the DTWT on images is taking rows and columns as 1D signals. This approach is called separable and it means that algorithms described in previous section can be exploited. Computing one step of the DTWT of the image is equivalent to two consecutive 1D DTWTs. As shown in Fig. 2, first the rows, then the columns are processed (or vice versa). In this paper only *non-standard* division of spectra is considered [13], so that each row and column are not decomposed all the way to the level J, but one step at a time and in j-th step four subbands are produced $\boldsymbol{a}^j_{\mathrm{LL}}, \boldsymbol{d}^j_{\mathrm{LH}}, \boldsymbol{d}^j_{\mathrm{HL}}, \boldsymbol{d}^j_{\mathrm{HH}}$ (the subscripts denote filter which was used for rows and columns respectively, superscript denotes level of decomposition). The next step is then performed on subband $\boldsymbol{a}^j_{\mathrm{LL}}$.

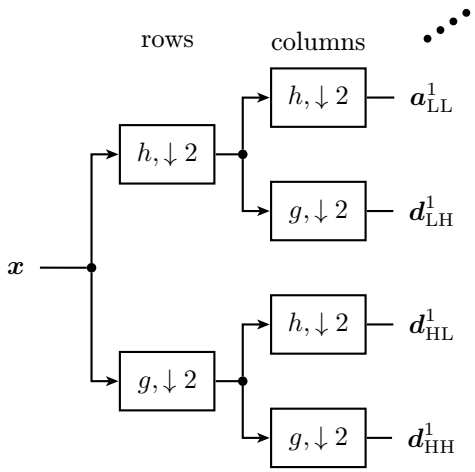

Figure 2: One level of filter bank for non-standard division of the spectra.

3 The Algorithm

In this section we present basic premises and modification of the SegDTWT algorithm so it can be generalized to images. The image will be divided to rectangular segments of dimensions $s_x(n_x, n_y)$, $s_y(n_x, n_y)$, m denotes length of the (possibly longer) filter. Working with images, their dimensions are usually known beforehand. Due to this fact, some simplifications can be introduced:

- Initially whole image has to be extended at the borders by $r(J)$ pixels, the values of the closest $(m-1)$ ones are set according to the selected boundary treatment method, and the rest is set to zero.
- The extensions of any single segment can be calculated beforehand so the segments do not have to be processed in the natural order.
- There is no longer need for special treatment of the next to the last and the last segments as well.

3.1 Generalization of SegDTWT

More detailed study of the SegDTWT algorithm (e.g. from [6, 12]) reveals two restricting features:

- The algorithm was primarily developed for real time processing of acoustic signals therefore the left extension $L_{\max}$ is chosen to be as high as possible to maximize re-usage of received samples. It is not necessary for images, so more general approach is taken in the following text.
- The algorithm considers only segments of equal size. In the following text it is shown that lengths of both the right and the left extensions of n-th and $(n+1)$-th segment, respectively, depend only on the position of their dividing line in the image.

Next results are based on Theorem 8.11 from [6] which can be written as:

Theorem 3: *Let the n-th segment is given, whose length including its left extension is $l(n)$. Then the left extension of the next segment $L(n+1)$ can be computed by the formula:*

$$L(n+1) = l(n) - 2^J i \quad \text{for} \quad i \in \left[\frac{l(n) - r(J)}{2^J}, \frac{l(n)}{2^J}\right], i \in \mathbb{N}^0. \tag{4}$$

And for the right extension of n-th segment the following holds:

$$R(n) = r(J) - L(n+1). \tag{5}$$

Maximum left extension $L_{\max}$ is naturally reached for $i = \mathrm{ceil}\left(\frac{l(n)-r(J)}{2^J}\right)$. Minimum left extension $L_{\min}$ for $i = \mathrm{floor}\left(\frac{l(n)}{2^J}\right)$. $R_{\min}$ (for $L_{\max}$) is already known (2) and $R_{\max}$ (for $L_{\min}$) can be written as

$$R_{\max}(n) = 2^J \mathrm{floor}\left(\frac{ns + r(J)}{2^J}\right) - ns, \tag{6}$$

(Proof is the same as in Theorem 8.14 in [6]). Using Theorem 3 we can write:

$$L(n+1) = l(n) - 2^J \left[\mathrm{ceil}\left(\frac{l(n) - r(J)}{2^J}\right) + k\right] \text{ where } k \in \mathbb{N}^0, \tag{7}$$

satisfying $L(n+1) \geq L_{\min}(n+1)$ at the same time. Comparing formulas (2), (6) and (7), right extension of n-th segment can be written as:

$$R(n) = 2^J \left[\mathrm{ceil}\left(\frac{ns}{2^J}\right) + k\right] - ns, \quad \text{where } k \in \mathbb{N}^0 \tag{8}$$

satisfying $R(n) \leq R_{\max}(n)$ at the same time.

The next theorem describes situation with segments not sharing the same size. The proof is omitted here due to its extent.

Theorem 4: *This result is graphically shown in Fig 3.*

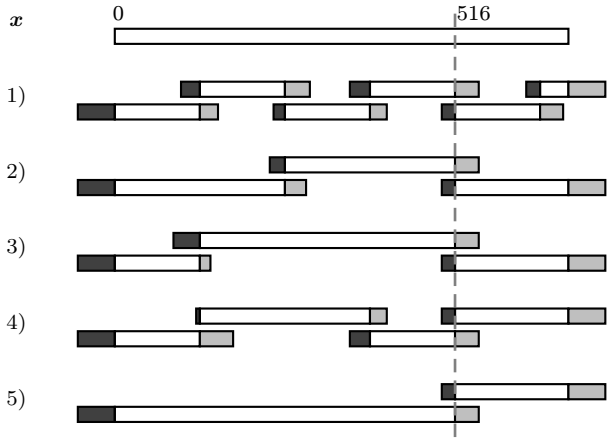

Figure 3: Figure presenting Theorem 4. Five cases of division of the input signal $\boldsymbol{x}$ are shown. There is always a division of a pair of segments between samples 515 and 516, but the divisions of the preceding part differs from case to case. Nevertheless, the lengths of related extensions of the neighboring segments are equal in all cases.

3.2 SegDTWT2D algorithm

In this section we present the actual generalization of the SegDTWT algorithm considering separability and non-standard spectral division as it was mentioned in Section 2.3. The new algorithm is called SegDTWT2D. For sake of simplicity only zero padding at the image boundaries is considered.

Let us denote current segment by (n_x, n_y) and its dimensions by $s_x(n_x, n_y)$ and $s_y(n_x, n_y)$. The image is divided into rectangular segments which are not required to have equal sizes (see, for example, Fig. 4). A special care is taken when working with segments of left boundary $(1, \cdot)$, right boundary $(\mathrm{end}_x, \cdot)$, top boundary $(\cdot, 1)$ and bottom boundary $(\cdot, \mathrm{end}_y)$.

The algorithm exploits the modified formulas (5), (8) so that for $n_x = 1, \ldots, \mathrm{end}_x - 1$ and $n_y = 1, \ldots, \mathrm{end}_y - 1$ and some k (satisfying condition in (8)) holds:

$$R(n_x) = 2^J \left[\mathrm{ceil}\left(\frac{S_{n_x}}{2^J} \right) + k \right] - S_{n_x}, \tag{9}$$

$$B(n_y) = 2^J \left[\mathrm{ceil}\left(\frac{S_{n_y}}{2^J} \right) + k \right] - S_{n_y}, \tag{10}$$

$$L(n_x + 1) = r(J) - R(n_x), \tag{11}$$

$$T(n_y + 1) = r(J) - B(n_y), \tag{12}$$

where n_x, n_y denotes current segment, S_{n_x}, S_{n_y} denotes the size of the smallest rectangle drawn from from the top left corner of the original image which still contains whole current segment. $R(n_x), B(n_y)$ are right and bottom extension of current segment, $L(n_x+1), T(n_y+1)$ are left and top extension of segments neighboring in the corresponding directions. For segments with $n_x = 1, n_y = 1, n_x = \mathrm{end}_x, n_y = \mathrm{end}_y$ we can show that

$$L(1) = T(1) = R(\mathrm{end}_x) = B(\mathrm{end}_y) = r(J). \tag{13}$$

Algorithm 5:[SegDTWT2D via separability] Let the image $\boldsymbol{x}$ of width S_x and height S_y, highpass filter $\boldsymbol{g}$, lowpass filter $\boldsymbol{h}$, both of length m, the depth of wavelet decomposition J be given:

1. The image is extended at boundaries by $r(J)$ zero pixels. However, for further reference, the original dimensions S_x, S_y and the origin are used.
2. **If** all pixels of the image have not been processed: Choose a segment (n_x, n_y) of dimensions $s_x(n_x, n_y), s_y(n_x, n_y)$ and set it as "current". **Else** finish the algorithm.
3. Extend the current segment according to formulas (9), (10), (11), (12) and (13).
4. Calculate 2D DTWT of depth J via separability property using one step of algorithm 1 at time on all rows and then on all columns but omitting step 2(a).
5. Modify the calculated wavelet coefficients by trimming off certain number of columns from the left and certain number of rows from the top, specifically: on the j-th level, $j = 1, 2, \ldots, J-1$ trim off $r(J-k)$ columns and $r(J-k)$ rows of coefficients.
6. **If** the segment index is equal to $n_x = \mathrm{end}_x$ or $n_y = \mathrm{end}_y$ trim off the coefficients in the same manner as was described in previous step, but this time from the right and bottom, respectively.
7. Store the resulting sets of coefficients ($3J+1$ for each segment) under ${}^{n_x,n_y}\boldsymbol{a}^J_{\mathrm{LL}}$, ${}^{n_x,n_y}\boldsymbol{d}^J_{\mathrm{LH}}, {}^{n_x,n_y}\boldsymbol{d}^J_{\mathrm{HL}}, {}^{n_x,n_y}\boldsymbol{d}^J_{\mathrm{HH}}, \ldots, {}^{n_x,n_y}\boldsymbol{d}^1_{\mathrm{HH}}$ and continue to item 2.

The algorithm can be extended to any type of boundary treatment like it was made in Algorithm 2.

4 Implementation

For performance purposes the new algorithm was implemented in C++. We used the Intel Threading Building Blocks (TBB) library to introduce parallelism into our program. TBB provides algorithms and concepts to fully exploit possibilities of multi-core processors. It also provides automatic task scheduler and automatic thread management. For more detailed information on using TBB in image processing see [14] or [15].

A basic concept of TBB is that tasks are recursively divided to smaller parts and they are processed in parallel. When processing an image, the initial task range spans the whole image. This range is than recursively split into halves and when the new range is small enough, it is

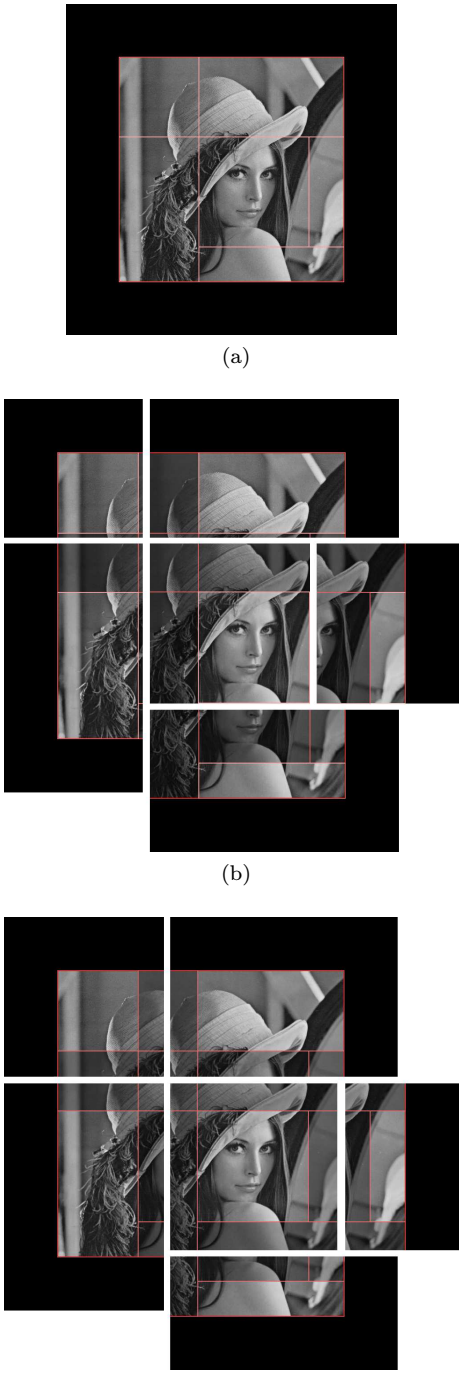

Figure 4: Example of division of image of size $S_x = 512, S_y = 512$. (a) The image is extended by $r(J)$ zero pixels for $J = 4, m = 9$, from which $m - 1$ closest pixels is changed by symmetric extension. It corresponds with step 1 in Algorithm 5. (b) Six segments with their extensions calculated using formulas (9), (10), (11), (12) and (13) with $k = 0$. It corresponds with step 3 in Algorithm 5. (c) Alternative extensions of the segments calculated using $k = 4$ (producing the same transform coefficients in the end).

processed. TBB manages this splitting automatically and even allows task stealing to achieve load balancing between working threads. A programmer can control size of the smallest range by parameter `grainsize`, but if TBB decides not to split the range any more, the `grainsize` does not have to be reached so `grainsize` is only a coarse value defining size up to which range will not be split. Hence, the size of the segment to be processed is not known beforehand. At this point the new algorithm enters and after every division the extensions of two affected segments are computed. Regarding to the Theorem 4 these extensions are not affected by any other division of ranges.

At this point a very important fact needs to be highlighted. Every division of the range brings redundancy of the computation. Obviously the number of redundant rows or columns of input pixels is equal to $r(J)$, which is dependent on the length of the filters m and the depth of wavelet decomposition J, see (1). So it is advisable to use as few divisions as possible, but at the same time it is important to effectively exploit all available threads to achieve speedup.

4.1 Testing

Via testing, we would like to establish the optimum `grainsize` for a given $r(J)$ to reach the highest speedup possible. The serial version, to which parallel versions in different setups are compared to, is computed as if whole image was one segment, so there are extensions only at the borders by $r(J)$. For testing purposes we used system running Intel C2Q Q9550 (4 cores). All data types were single precision 32-bit floating points. The compiler associated with Microsoft Visual Studio 2008 was used with `\O2` optimization parameter. Every test was performed ten times and as the result the median was taken.

Firstly, we performed several tests with 4096×4096 px image for fixed $r(5)$ and varying `grainsize`. It can be seen in Fig. 5 that the speedup is relatively independent on choice of `grainsize`, but with increasing $r(J)$ bigger `grainsize` is needed. Using that, speedup for different $r(J)$ with almost optimum choice of `grainsize` $\sim 2r(J)$ is shown in Fig. 6.

4.2 Scalability

We got an other interesting result through Intel parallel Universe Portal. It is a web service where Intel offers computing time on their 8-Core Intel Xeon@2.80GHz with hyperthreading (effectively 16 cores). The tests were performed in the following setup: image 4096×4096 px, $m = 10$, $J = 4$ leading to $r(J) = 135$, `grainsize` was set to 512 in both directions. In Fig. 7 obvious scalability of speedup can be seen – that means the performance is increasing with increasing number of cores.

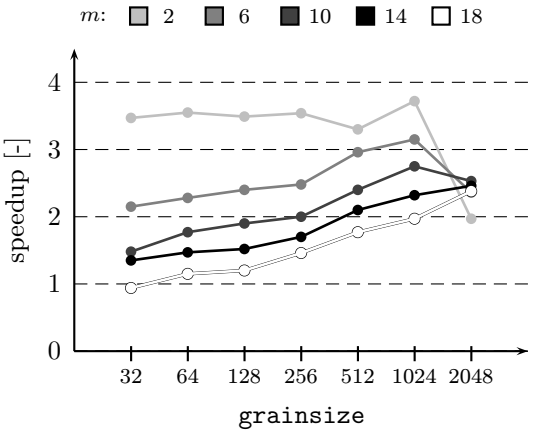

Figure 5: Speedup for increasing `grainsize` for different values of $r(5)$ (31, 155, 279, 403, 527 corresponding to $m = 2, 6, 10, 14, 18$)

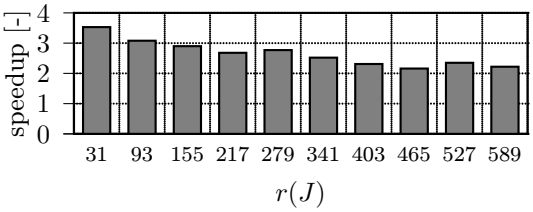

Figure 6: Speedup for increasing $r(J)$ for $J = 5$ and $m = 2..20$

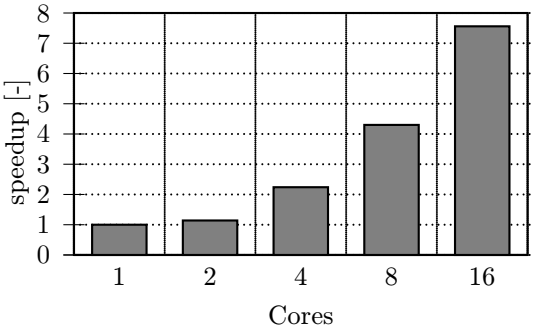

Figure 7: Speedup on Intel Parallel Universe

5 Software

The described implementation is distributed as a static library for 32bit OS Windows freely under GPLv2 license. The library is accessible from `http://www.utko.feec.vutbr.cz/~rajmic/papers/segdtwt2Dv10_win.zip`. To use the library, it is necessary to include header file `segDTWT.h` from `include` directory and set linker to include `segDTWT2D.lib` from the `lib` directory. The function, which performs the parallel forward wavelet transform is `segDTWTfwd_32f_C1` and accepts the following parameters:

```
segDTWTfwd_32f_C1(float* i_data,
                  int i_widthStep,
                  float* subbands[],
                  int widthSteps[],
                  int levels,
                  Size size,
                  separableWavelet* w);
```

`float* i_data` – pointer to the beginning of the image
`int i_widthStep` – distance between two consecutive rows of the image in memory in bytes
`float* subbands[]` – array of pointers to the output subbands. For details see below.
`int widthSteps[]` – array of the distances between two consecutive rows in bytes in EACH individual subband
`int levels` – depth of decomposition, the J variable
`Size size` – dimensions of the input image

```
typedef struct
  int width;
  int height;
Size;
```

`separableWavelet* w` – object defining wavelet filters

Prior to the calling of this function one must have `separableWavelet` object prepared and also the memory for the output subbands need to be allocated. One of the constructors for `separableWavelet` class accepts wavelet name and file name, where the wavelet filters definitions are located.

```
separableWavelet(string name="default",
          const char* file = "wavelets.dat")
```

The `wavelets.dat` file is distributed with the library and it contains wavelet filters defined in MatLab wavelet toolbox, but more filters can be added keeping the prescribed format.

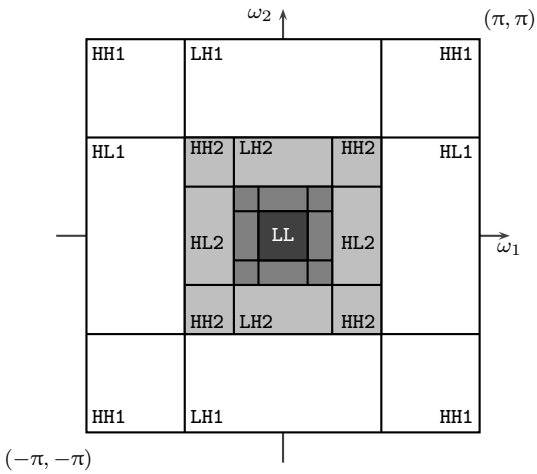

Figure 8: Subbands labeling

The `subbands[]` is array of pointers to the output subbands. The pointers are stored in the following order: `subbands[0]=HL1,[1]=HH1,[2]=LH1,[3]=HL2,[4]=HH2,[5]=LH2 ...[last]=LL` assuming the subband labeling as depicted in fig. 8. The memory allocation can by done by means of the `allocateSubbands` function:

```
allocateSubbands(float* subbands[],
                 int widthSteps[],
                 int level,
                 int filter_length_L,
                 int filter_length_H,
                 Size size);
```

`float* subbands[]` – array of pointers to be filled with pointers to the subbands

`int widthSteps[]` – array of the distances between two consecutive rows in bytes in EACH individual subband

`int filter_length_L/H]` – length of the low- high-pass filter

The subband dimensions $N_{\mathrm{x}}(m,j), N_{\mathrm{y}}(m,j)$ in the decomposition step $j = 1 \dots J$ are the following:

$$N_{\mathrm{x}}(m,j) = \mathrm{floor}(2^{-j} S_{\mathrm{x}} + (1 - 2^{-j}(m-1)) \qquad (14)$$

$$N_{\mathrm{y}}(m,j) = \mathrm{floor}(2^{-j} S_{\mathrm{y}} + (1 - 2^{-j}(m-1)), \qquad (15)$$

where original dimensions of the input image are denoted as $S_{\mathrm{x}}, S_{\mathrm{y}}$.

In addition, two global variables can be set. The `wrapType extensionType` defining the boundary treatment method:

```
typedef enum
  ZEROS_PADDING = 0,
  CONST_PADDING = 1,
  SYMETRIC_HP = 2,
  SYMETRIC_WP = 3
 wrapType;
```

and `Size segSize` defining the `grainsize` for the `parallel_for` algorithm.

6 Conclusion

In this paper, the new algorithm SegDTWT2D was presented. It allows segmentwise computation of 2D DTWT. Several assumptions were established and two theorems extending the original algorithm SegDTWT were proposed. It was shown that the new algorithm is suitable for parallel execution on multi-core processors and it brings significant speedup of an execution, especially for small values of $r(J)$. Also the parallel implementation seems to scale well on processors with higher number of cores.

Further effort will be devoted to algorithm for segmented inverse wavelet transform and to more general filter banks (eg. wavelet packets, nonseparable filter banks).

Acknowledgments

This paper was supported by FEKTJ-10-8/225 grant of the Brno University of Technology and COST grant OC08057.

References

[1] D. S. TAUBMAN and M. W. MARCELLIN, *JPEG2000, Image compression fundamentals, standards and practice*, ser. 3. USA: Kluvert Academic Publishers, 2002.

[2] D. L. DONOHO, "De-noising by soft-thresholding," 1994.

[3] J. Z. WANG and G. WIEDERHOLD, "Wavemark: Digital image watermarking using daubechies' wavelets and error correcting coding," in *Proceedings of the SPIE Int. Symp. on Voice, Video, Data Communications*, 1998.

[4] R. N. STRICKLAND and H. I. HAHN, "Wavelet transform methods for object detection and recovery," *IEEE Trans Image Process*, vol. 6, no. 5, pp. 724–35, 1997.

[5] S. MALLAT, *A Wavelet tour of signal processing, The Sparse way*, 3rd ed. Boston: Academic Press, 2008.

[6] P. RAJMIC, "Exploitation of the wavelet transform and mathematical statistics for separation signals and noise, (in czech)," Ph.D. dissertation, Brno University of Technology, Brno, 2004.

[7] D. CHAVER, M. PPRIETO, L. PINUEL, and F. TIRADO, "Parallel wavelet transform for large scale image processing," *Parallel and Distributed Processing Symposium, International*, vol. 1, p. 0004, 2002.

[8] D. ONCHIS and C. MARTA, "Multiple 1d data parallel wavelet transform," *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on*, vol. 0, pp. 178–181, 2005.

[9] J. FRANCO, G. BERNABE, J. FERNANDEZ, and M. E. ACACIO, "A parallel implementation of the 2d wavelet transform using cuda," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, pp. 111–118, 2009.

[10] T. NGUYEN and G. STRANG, *Wavelets and Filter Banks*. Wellesley College, 1996.

[11] V. SILVA, L. SA, and L. SÁ, "General method for perfect reconstruction subband processing of finite length signals using linear extensions," *IEEE Trans. on Signal Processing*, vol. 47, 1999.

[12] P. RAJMIC, "Algorithms for segmentwise computation of forward and inverse discrete-time wavelet transform," *Journal of Concrete and Applicable Mathematics,Special Issues on Applied Mathematics and Approximation Theory*, 2010.

[13] R. VARGIC, *Wavelets and filter banks*. Bratislava: STU in Bratislava, 2004, in Slovak.

[14] Z. PRUSA and J. MALY, "Parallel image processing using intel libraries," in *32nd International Conference on TELECOMMUNICATIONS AND SIGNAL PROCESSING*, Dunakiliti: ASSZISZTENCIA Congress Bureau, 2009, pp. 1–4.

[15] J. REINDERS, *Intel Threading Building Blocks*. New York: McGraw-Hill, 2007.