The Journal of Supercomputing (2022) 78:11539–11563
https://doi.org/10.1007/s11227-022-04331-8

An efficient implementation of one-dimensional discrete wavelet transform algorithms for GPU architectures

Kamil Stokfiszewski[1] · Kamil Wieloch[1] · Mykhaylo Yatsymirskyy[1]

Accepted: 18 January 2022 / Published online: 14 February 2022

Abstract

In this paper, the authors present several self-developed implementation variants of the Discrete Wavelet Transform (DWT) computation algorithms and compare their execution times against the commonly approved ones for representative modern Graphics Processing Units (GPUs) architectures. The proposed solutions avoid the time-consuming modulo divisions and conditional instructions used for DWT filters wrapping by proper expansion of the DWTs input data vectors. The main goal of the research is to improve the computation times for popular DWT algorithms for representative modern GPU architectures while retaining the code's clarity and simplicity. The relations between algorithms execution time improvements for GPUs are also compared with their counterparts for traditional sequential processors. The experimental study shows that the proposed implementations, in the case of parallel realization on GPUs, are characterized by very simple kernel code and high execution time performance.

Keywords Time effectiveness optimization · Graphics processing unit (GPU) · Discrete wavelet transform (DWT) · Lattice structure · Matrix-based approach

1 Introduction

The digital signal processing (DSP) has become an integral part of everyday life. The increasing number of electronic devices has led to a situation where almost everyone has to deal with digitally processed data. A digital signal can be a sequence

✉ Kamil Stokfiszewski
kamil.stokfiszewski@p.lodz.pl

Kamil Wieloch
kamil.wieloch@edu.p.lodz.pl

Mykhaylo Yatsymirskyy
mykhaylo.yatsymirskyy@p.lodz.pl

[1] Institute of Information Technology, Lodz University of Technology, ul. Wolczanska 215, 90-924 Lodz, Poland

of samples extracted from a continuous signal or any discrete set of data like, e.g., image in bitmap form. However, still more often, the digital signal is directly derived from the continuous signal as its representation [1].

Nowadays, numerous computational problems are so complex that they not only require computer aided processing, but also a very efficient processing of huge data sets, often in real time [2]. Therefore, there are efforts to optimize the known algorithms both from the perspective of the processing time and the precision of the results as well. Current processors are often optimized to an extreme physical operational conditions, e.g., the widths of the electric paths are regularly close to the atomic size. This causes the need to look for new techniques to increase the computational efficiency. Increasing the frequency also meets the physical barriers. Those are only few of the reasons why parallel computing is becoming more and more popular [3, 4]. The conversion of traditional, sequential computation algorithms to their parallel counterparts requires suitable implementations and poses a real challenge for software engineers involved in algorithm optimization [5].

The quality of the developed algorithms is, of course, subject to evaluation. There are different evaluation methods. Often we refer to computational complexity or time complexity. In the case of parallel devices and, in particular, graphics cards, the problem is much more complicated. The number of basic operations such as arithmetic, logic, and memory accesses does not directly affect the actual computation time. Extra parameters like versatility, simplicity, ease of implementation, ease of modification, utilization of hardware resources, achieved acceleration, complexity of communication, synchronizations, portability and others are also highly important. Among this and other reasons, time efficiency models are created for specific architectures. In the end, it often eventually turns out that only experimental studies give a real insight into performance characteristics of a chosen computational method. Software designer always tries to minimize the computational complexity, memory usage or other criteria of the overall cost [6]. However, such approaches are not always successful. It is quite often the case that a simpler algorithm with worse traditional computational complexity performs better because of the advantages resulting from its computational structure design.

In this paper, we present several optimization variants of commonly used DWT computation algorithms, namely the matrix and the lattice structure-based approaches, and compare their execution time effectiveness for both CPU and GPU implementations. The results indicate that, despite of twofold reduction in computational complexity of the lattice structure-based approach in comparison with the matrix-based method, the former algorithm performs significantly worse for large transform sizes due to its more complex computational structure when implemented on GPU.

2 Discrete wavelet transform

An important reason for transforming data from one form into another is the desire to analyze the features of the signal that are more visible after the transformation than in the original form. Such transformations are also sometimes called mappings.

The fundamental group of transformations is linear transformations which are one of the axioms of linear algebra. These are homomorphic mappings preserving the spatial structure. A special case of linear transformations is orthogonal ones possessing an important property of their inverse matrix being the transposition of the forward one. Computing the inverse matrix for multidimensional data can be a much more complex problem than a simple transposition for transformations. It is common to use harmonic functions as basis functions due to the fact that harmonic signals do not change their shape after passing through a linear system (only their phase and amplitude changes) [7].

The wavelet transform provides a time-frequency representation of a signal. It is similar to the Short Time Fourier Transform (STFT), but the wavelet transform uses a multi-resolution technique where different frequencies are analyzed at different resolutions. Its name is derived from the term "wavelet" meaning localized wave (Fig. 1). The energy of such signal is concentrated in time and space. Another difference between wavelet transform and Fourier transform is that Fourier transform uses waves to analyze signal, while wavelet transform uses short waves of finite energy. It can be stated that wavelet transform gives good time resolution at high frequencies and poor frequency resolution at low frequencies.

The discrete wavelet transform (DWT) is a discretized version of the continuous wavelet transform (CWT). In CWT, the signals are analyzed using a set of basis functions which relate to each other by simple scaling and translation. In the case of DWT, a timescale representation of the digital signal is obtained using digital filtering techniques. The signal is passed through filters with different cutoff frequencies at different scales [8].

The basic method of calculating the linear transformation is an algebraic approach based on multiplying a matrix by a vector of the input signal. This operation can be simply stated as

$$\mathbf{y} = \mathbf{A}\mathbf{x},$$

where $\mathbf{x}$ is a N-element input vector, $\mathbf{y}$ is a N-element output vector, and $\mathbf{A}$ is a $N \times N$ -element matrix performing the filtering. There are infinitely many wavelets, so there is also an infinite number of possible wavelet transforms.

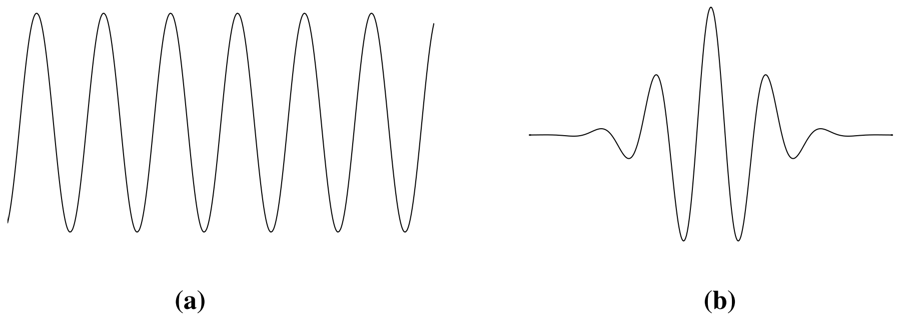


Fig. 1 Example of wave **a** and wavelet **b**

The wavelet transform for discrete signals is specified by the following formula:

$$W_{\Psi}[j,k] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n]\psi_{j,k}[n],$$

where x represents the signal, and $\Psi_{j,k}$ is the wavelet transform kernel for which the parameters j and k represent shifts in the frequency and time domain. Wavelet transformation can be implemented using low-pass and high-pass filters. The organization of the calculations means that in each step, the high-pass filter produces detailed results, while the result of the low-pass filter depends on the scaling function. Thus, the time resolution remains at a good level for high frequencies and the frequency resolution remains at a good level for low frequencies. The number of decomposition stages depends on the size of the transform. The reconstruction process is based on the inverted version of the scheme presented in Fig. 2.

The discrete wavelet transform in direct matrix form can be implemented as a stage in the analysis of a two-channel filter bank with a finite impulse response. The two filters h and g with the number of coefficients K each compose the transformation matrix [9–11]:

$$\mathbf{A} = \begin{bmatrix} h_{K-1} & \cdots & h_1 & h_0 & 0 & 0 & \cdots & 0 & 0 \\ g_{K-1} & \cdots & g_1 & g_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & h_{K-1} & \cdots & h_1 & h_0 & \cdots & 0 & 0 \\ 0 & 0 & g_{K-1} & \cdots & g_1 & g_0 & \cdots & 0 & 0 \\ \vdots & & \vdots & & \vdots & \vdots & \ddots & \vdots & \vdots \\ h_{K-3} & \cdots & h_1 & h_0 & 0 & 0 & \cdots & h_{K-1} & h_{K-2} \\ g_{K-3} & \cdots & g_1 & g_0 & 0 & 0 & \cdots & g_{K-1} & g_{K-2} \end{bmatrix}.$$

Two channel filter banks can be implemented in various ways. These are direct form, polyphase structure, lattice structure and lifting structure [12]. From the efficiency point of view, each of these has its advantages and drawbacks. For example, the lattice structure having good computational complexity cannot be used for all types of filters, and in the case of parallel implementation, its structure forces the need for synchronizations. In the case of polyphase implementations, it is easy to achieve a high degree of parallelism of computations, but the compute complexity is higher than in the case of the lattice form [1]. In the case of a lifting structure, the problem of quantization error arises, which is increased with each step. Despite this,

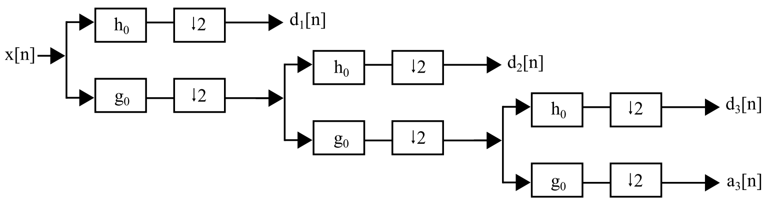


Fig. 2 Example decomposition tree of DWT with filters h and g

the lattice structure is acclaimed as an efficient tool in the implementation of two channel filter banks with finite impulse response.

The computations for the first $K/2$ stages of lattice structure-based DWT are described by the $\Gamma_{i,j}$ operations (denoted by '•' in Fig. 3):

$$\Gamma_{i,j} = \begin{bmatrix} 1 & s_{i,j} \\ t_{i,j} & 1 \end{bmatrix}, \tag{1}$$

where $s_{i,j}$, $t_{i,j}$ are parameters whose values are determined during the factorization process, $i = 0, 1, \dots, K/2 - 1$ and $j = 0, 1, \dots, N/2 - 1$. A single $\Gamma_{i,j}$ operation consists of two multiplications and two additions.

The wavelets used in DWT can be classified into two categories: orthogonal and biorthogonal. The coefficients of orthogonal filters are real numbers. Both filters have the same length and are not symmetric. If we mark the low-pass filter as g and the high-pass filter as h, then the following relation should be true: $h_0(k) = z^{-N} g_0(-z^{-1})$. For biorthogonal filters, the low-pass filter and the high-pass filter have different lengths. The low-pass filter is symmetric and the coefficients are real numbers or integers. There are many functions that can be used to prepare wavelets for the DWT. Some of the most common ones are shown in Fig. 4.

One of the earlier wavelets examples is the Haar wavelet, in today's practice though Daubechies wavelets are very common. Many wavelets are created for specific features. The field of designing wavelets is very wide and still growing. It involves both theoretical aspects of wavelets and their implementation. Wavelets are the result of work in many fields, including mathematics, physics, computer science and others but the target of research is to develop tools for describing functions in time and frequency domain at the same time. Each practical application of DWT requires wavelet bases with different properties. In practical applications, the larger

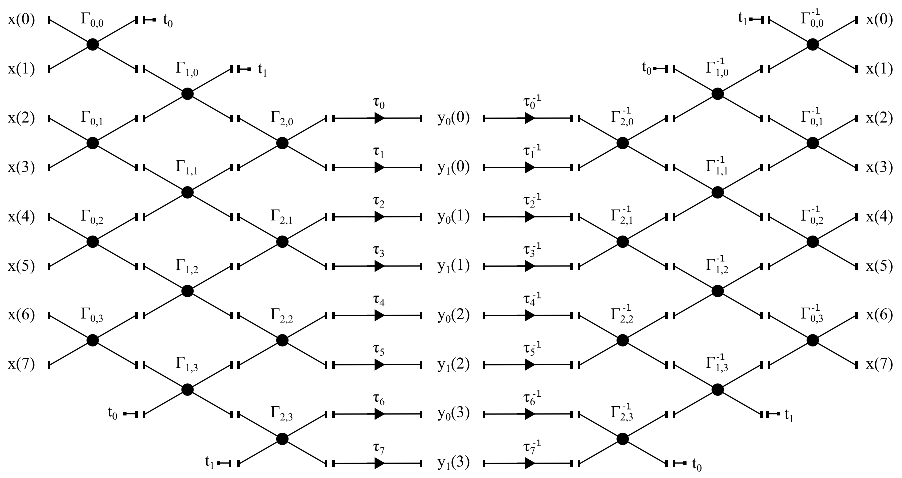

x(0)
x(1)
x(2)
x(3)
x(4)
x(5)
x(6)
x(7)
$\Gamma_{0,0}$
$\Gamma_{0,1}$
$\Gamma_{0,2}$
$\Gamma_{0,3}$
$\Gamma_{1,0}$
$\Gamma_{1,1}$
$\Gamma_{1,2}$
$\Gamma_{1,3}$
$\Gamma_{2,0}$
$\Gamma_{2,1}$
$\Gamma_{2,2}$
$\Gamma_{2,3}$
t_0
t_1
τ_0 … τ_7
$y_0(0)$, $y_1(0)$, $y_0(1)$, $y_1(1)$, $y_0(2)$, $y_1(2)$, $y_0(3)$, $y_1(3)$
τ_0^{-1} … τ_7^{-1}
$\Gamma_{2,0}^{-1}$, $\Gamma_{2,1}^{-1}$, $\Gamma_{2,2}^{-1}$, $\Gamma_{2,3}^{-1}$
$\Gamma_{1,0}^{-1}$, $\Gamma_{1,1}^{-1}$, $\Gamma_{1,2}^{-1}$, $\Gamma_{1,3}^{-1}$
$\Gamma_{0,0}^{-1}$, $\Gamma_{0,1}^{-1}$, $\Gamma_{0,2}^{-1}$, $\Gamma_{0,3}^{-1}$

Fig. 3 Lattice structure of $N = 8$ point DWT (both decomposition and reconstruction) and filters of length $K = 6$

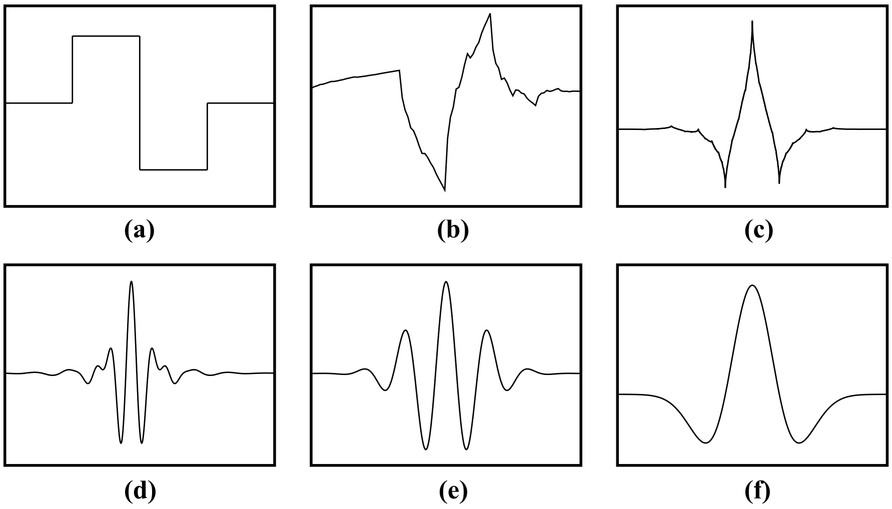


Fig. 4 Example of few popular wavelets: **a** Haar, **b** Daubechies, **c** Coiflet, **d** Meyer, **e** Morlet, **f** Mexican Hat

the set of wavelets the better. In order to prepare the correct and useful wavelets, a set of restrictions is imposed. Among many conditions, most important are regularity, symmetry, compact support, orthogonality with polynomials of degree *n*. It is not possible to achieve all carious properties of wavelets at the same time because different properties may be incompatible with each other. Biorthogonal wavelets have most of the properties of orthogonal wavelets, with the advantage that they are more flexible. There are many more biorthogonal wavelets than orthogonal wavelets. Detailed rules for designing wavelets are discussed in the paper [7].

The DWT finds countless applications today, such as digital signal processing, high-speed data transmission protocols, biometric data and image compression, e.g., in JPEG2000 standard, in case of which DWT enables attaining high compression ratios with good quality of image reconstruction.

2.1 Theoretical effectiveness of selected DWT algorithms

We have chosen to base our considerations on two selected implementations of the DWT, namely the direct matrix-based approach, also called the convolutional form, and the lattice structure-based approach. The direct matrix approach and the lattice structure-based one both have features that are particularly important in the context of parallel implementations on a GPU. The convolutional form is very simple, and its matrix structure is beneficial for the GPU architecture. In contrast, the lattice form reduces computational complexity, but its staged structure generates some implementation difficulties and sequentially parallel execution. Moreover, the stepwise structure creates another obstacle for the implementation, because in the traditional approach, the kernel code for the last step should be different than all previous steps, due to the fact that the last step consists only of multiplication, while each previous step consists of two multiplications and two additions. This generates the need to call a special kernel code for the last stage or to use a conditional instruction in the kernel code and both solutions are clearly disadvantageous in terms of GPU computing performance. However, let's take a closer look at the theoretical time complexity of these algorithms.

Computational complexity of the direct matrix-based approach to DWT calculation requires $\mathcal{C}_{\mathrm{MUL}}^{\mathrm{MAT}} = NK$ multiplications and $\mathcal{C}_{\mathrm{ADD}}^{\mathrm{MAT}} = N(K-1)$ additions, where K is the filter length, and N is the size of wavelet transformation. Of course, these values can be lowered for a lattice structure due to the use of a factorization process. Such factorization exploits the dependencies between the filters of the two banks that are inherent in the perfect reconstruction condition imposed on the filter banks. As a result, the computational complexity of the wavelet transform using a lattice structure is characterized by a theoretical computational complexity of $\mathcal{C}_{\mathrm{MUL}}^{\mathrm{LAT}} = \frac{1}{2}N(K+2)$ multiplications and $\mathcal{C}_{\mathrm{ADD}}^{\mathrm{LAT}} = \frac{1}{2}NK$ additions. Therefore, it can be said that from the point of view of both multiplication and addition operations, the lattice structure has twice better computational complexity. The relationship between theoretical computational complexities is particularly important in the case of sequential implementations, but in the case of parallel processors, a very precise model is necessary to translate the theoretical complexities into real computational time. Among the group of purely theoretical evaluation tools for parallel algorithms, there is a property called *step-complexity* [13, 14]. Stepwise complexity is defined as the minimum required number of sequential steps necessary for a parallel algorithm to complete its calculations, considering the infinite amount of arithmetic processing units available during its activity. Using this definition and combining the operations of addition and multiplication, we can say that the step-complexity of the matrix algorithm is $\mathcal{S}_{\mathrm{ALL}}^{\mathrm{MAT}} = 2K - 1$, while the step-complexity of the lattice algorithm is $\mathcal{S}_{\mathrm{ALL}}^{\mathrm{LAT}} = 2K + 2$. Evaluation of parallel algorithms is a tricky issue because the number of elementary operations influencing the computational complexity and memory utilization lose their significance. Instead, important are aspects such as universality, simplicity of the algorithm, ease of implementation and possible modification, degree of utilization of available resources, acceleration achieved, cost, communication complexity, portability. Efforts to develop accurate execution time prediction models for parallel platforms have been carried out for a period many years now, but still no model has been accepted as a general parallel computing model or even GPU limited model. There are simulators, but they are expensive to use in terms of simulation time and difficult to maintain for developers as they require constant updates for new layouts. For this reason, conventional experimental research is still very popular in the case of GPGPU.

3 Related work

There are some different architectures for implementing a two-channel DWT filter bank. The most popular implementations are direct matrix-based structure, polyphase structure, lifting structure and lattice structure. The transformation itself is highly acclaimed and finds countless practical applications [32]. The classical DWT algorithm is a direct implementation of a 2-channel filter bank based on building a transformation matrix [7]. Lattice factorization of two-channel orthogonal filter bank for wavelet transform has been proposed in the following papers [9, 15, 16]. The effectiveness of the mentioned lattice factorization against existing solutions has been tested for CPU in paper [3] and for GPU in paper [17]. The research on efficient realizations of DWT, e.g., using lattice structures is also very common for both CPU and GPU processors [18].

The proposed implementation of the DWT without wrapping, to the best of our knowledge, is a novelty in the implementation of both the lattice and matrix-based DWT algorithms. By extending the input data sets, we are able to avoid time-expensive modulo divisions and conditional instructions. All implementation details are presented in the next chapter. In addition, we present the performance of the developed implementations for a few different granularities of computations division. As shown in the research [19, 20], the partitioning of a computational task directly affects general performance.

Although computational complexities of the proposed algorithms and their step-efficiencies, which we have presented in the previous chapter, can be quite easily derived, we have decided to evaluate the proposed solutions experimentally, along with validation of the achieved numerical accuracy. This comes from the fact that based on our previous experiences we know that in the case of GPU implementations, often only experimental tests can provide a real insight into the algorithms performance. However, it is still worth noting that there are many models for parallel computations, such as, e.g., the general PRAM model (Parallel Random Access Machine) which are useful in, at least coarse, estimation of parallel algorithms execution performance. Unfortunately, many of them are significantly inaccurate because of the complexity of GPUs architectures, which combine together the features of both sequential and parallel computations [5]. We here should note the existing analytical, statistical, simulation-based and hybrid models [21–29]. The precision of those models, depending on the type of algorithm, oscillates from about 90% upward, with a maximum prediction mismatch of up to 50%. The interested Reader can find a comprehensive review of the existing models in our paper [30] in which we have also proposed the new execution time prediction model for parallel GPU implementations of the discrete transforms computation algorithms, i.e., for the algorithms analyzed in the present work.

4 DWT implementations

In this section, we present our self-developed and tested implementations of the DWT computation algorithms along with their reference implementations. During our study, we have tested a significant amount of speed up techniques of the algorithms analyzed in this paper which cannot all be presented in this work for obvious reasons. Among all the variations we have prepared and tested, we have chosen those which gave the best performance increase during the tests. The improvements were made at the level of implementation, compilation and the algorithm structure. We noticed that wrapping the filters within the transformation matrix is computationally expensive so we proposed solution without wrapping for both (matrix-based and lattice) algorithms. We also tested the impact of granularity of the computations on the total time in the case of graphics cards.

For the sake of clarity, we have labeled the selected sequential implementations analyzed in this article as *cpu_dwt_m2_ww_ref*, *cpu_dwt_m2_nw*, *cpu_dwt_l2_ww*, *cpu_dwt_l2_nw*, and the parallel GPU implementations as *gpu_dwt_m1_ww_ref*, *gpu_dwt_m2_ww*, *gpu_dwt_m4_ww*, *gpu_dwt_m2_nw*, *gpu_dwt_m4_nw*, *gpu_dwt_l2_ww* and *gpu_dwt_l2_nw*. The naming convention used for the implementations is designed in such a way to immediately give an intuitive hint on how a given algorithm is implemented. A name starting with *"cpu"* means that it is an implementation designed for CPUs, while the beginning *"gpu"* indicates that the implementation is designed for GPUs. The next segment, *"dwt"*, is exactly the same for all implementations, as they all implement the DWT transformation (using different algorithms). The following segment indicates whether the implementation uses a lattice algorithm or a direct matrix approach along with the algorithm's granularity, e.g., *"m4"* stands for a matrix algorithm with granularity *"4"*, *"l2"* stands for a lattice algorithm with granularity *"2"*. The next segment specifies if the implementation contains wrapping, *"ww"* is used for implementations that wrap the DWT filters and *"nw"* is used for implementations that bypass the filter wrapping. At the very end of the tested implementation name, the *"ref"* postfix may appear to indicate that we are treating this particular implementation as a reference for the others.

Let us now present the details of the proposed implementations. For the wrapped matrix model, the input signal has a length of N samples and the transformation matrix has a size of $N \times N$-elements. In the case of the matrix model without wrapping, for an input signal of length N samples the transformation matrix has size $N \times (N + K - 2)$ where K represents the filters length. The input signal is expanded by $K - 2$ initial samples. Likewise, for the lattice model without wrapping, the input signal must be extended by $K - 2$ initial samples (K defines the length of the filters). Then, because of the introduction of additional operations in each step, wrapping can be bypassed.

Parallel implementations were developed with different granularity. The granularity choice when implementing an algorithm is especially important for parallel architectures, but it is also affecting sequential program performance. For matrix-based algorithms, granularity "1" means the maximum level of parallelism, unfortunately it means at the same time the need to use the low-pass and high-pass filter selection logic. Granularity "2" eliminates the need to select the appropriate filter for even and odd indexes of the input signal, in other words, means processing of 2 rows of the matrix by a single thread, analogically, granularization "4" means processing of "4" rows of the matrix by a single thread. In the case of the lattice-based DWT algorithms, the most intuitive granularity is a multiple of "2," because every single butterfly operation always works on "2" elements of the input vector. The detailed impact of granularity selection for GPU efficiency has been tested using the Fast Fourier Transform in the following paper [19]. All the code listings, presented in the following section, use a consistent variable naming: K - the filter length (integer, odd, natural number), N - input signal length (integer, odd, natural number, $N > K$), `hp[]` - vector with first filter samples (array of floats), `gp[]` - vector with second filter samples (array of floats), `wtg[]` - vector with tangent factors of base operations (array of floats, $K/2+1$ elements), `x[]` - input/output signal (N - samples), `y[]` - helper table for computations (N - elements), finally `hpgp[]` - the merged `hp[]` and `gp[]` arrays ($2K$ elements).

4.1 Sequential implementations on CPUs

4.1.1 *cpu_dwt_m2_ww_ref* - sequential, matrix-based, 2-point, with wrapping

We will start our analysis by presenting a chosen reference implementation. There are many examples of DWT implementations for CPUs in the literature. However, we have chosen an implementation presented in the book [7] whereby the authors proposed a pseudocode for the forward fast wavelet transform, as well as its inversed version. What's more, code was presented in a clear manner and supported by mathematical derivation of the transform formulas with finite sequence of coefficients. The authors of the book are acknowledged professionals in the field of discrete linear transformations and although their aim was not to optimize the implementation, we believe that it is an excellent starting point for optimization and further research. This implementation is representative of direct matrix approach Fig. 5a using modulo division to perform filter wrapping. We have optimized this proposal by reducing number of modulo divisions, integration of *for* loops and reducing the organisation cost. In our implementation, instead of two internal *for* loops, there is only one, where single iteration computes two elements of the output vector. The filter arrays h and g remain separated. The implementation is considered a reference implementation for the other sequential implementations.

$$\begin{bmatrix} y_0\\ y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7\\ y_8\\ y_9 \end{bmatrix} = \begin{bmatrix} h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & & & & \\ g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & & & & \\ & & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & & \\ & & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & & \\ & & & & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 \\ & & & & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 \\ h_1 & h_0 & & & & & h_5 & h_4 & h_3 & h_2 \\ g_1 & g_0 & & & & & g_5 & g_4 & g_3 & g_2 \\ h_3 & h_2 & h_1 & h_0 & & & & & h_5 & h_4 \\ g_3 & g_2 & g_1 & g_0 & & & & & g_5 & g_4 \end{bmatrix} \begin{bmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8\\ x_9 \end{bmatrix}$$

(a)

$$\begin{bmatrix} y_0\\ y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7\\ y_8\\ y_9 \end{bmatrix} = \begin{bmatrix} h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & & & & & & & & \\ g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & & & & & & & & \\ & & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & & & & & & \\ & & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & & & & & & \\ & & & & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & & & & \\ & & & & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & & & & \\ & & & & & & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 & & \\ & & & & & & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 & & \\ & & & & & & & & h_5 & h_4 & h_3 & h_2 & h_1 & h_0 \\ & & & & & & & & g_5 & g_4 & g_3 & g_2 & g_1 & g_0 \end{bmatrix} \begin{bmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8\\ x_9\\ x_0\\ x_1\\ x_2\\ x_3 \end{bmatrix}$$

(b)

Fig. 5 Matrix model with wrapping **a** and without wrapping **b** for a transformation of size N=10 and filters of length K = 6

Listing 1 Code fragment of the ***cpu_dwt_m2_ww_ref*** implementation.

```
float t1, t2;                                                       // Output:
for (int k = 0; k < N; k += 2){y[k] = x[k]; y[k + 1] = x[k + 1];}  // x[] - signal after transform
for (int Nd2 = N / 2, i1 = 0, i = 0; i < N; i += 2, i1++)          //
{                                                                   //
  t1 = y[i] * hp[0] + y[i + 1] * hp[1];                             // Input:
  t2 = y[i] * gp[0] + y[i + 1] * gp[1];                             // x[] - input signal (float)
  for (int j, k = 2; k < K; k += 2)                                 // N - signal length (int)
  {                                                                 // t1, t2 - temp variables (float)
    j = (i + k) % N;                                                // y[] - temporary table (float)
    t1 += y[j] * hp[k] + y[j + 1] * hp[k + 1];                      // hp[] - first filter (float)
    t2 += y[j] * gp[k] + y[j + 1] * gp[k + 1];                      // gp[] - second filter (float)
  }                                                                 // K - filter length (int)
  x[i1] = t1; x[i1 + Nd2] = t2;                                     //
}                                                                   //
```

4.1.2 ***cpu_dwt_m2_nw*** - sequential, matrix-based, 2-point, without wrapping

Based on the reference implementation *cpu_dwt_m2_ww_ref*, we present an implementation of *cpu_dwt_m2_nw*. The main difference is that this implementation completely bypasses the need to wrap the filters by using the concept presented in the previous section (Fig. 5b). In fact, it is very similar to *cpu_dwt_m2_ww_ref* but does not have any modulo operations or conditional statements. To achieve this, the input signal had to be extended by $K-2$ initial samples. Therefore, there is an additional loop in the code performing $K-2$ iterations. This implementation is the first evaluator of the proposed *no-wrap* optimization 5(a). The filters remain separated into two arrays. Every single loop iteration computes two samples of the output vector with the use of two separated low-pass and high-pass filters.

Listing 2 Code fragment of the ***cpu_dwt_m2_nw*** implementation.

```
float t1, t2;                                                       // Output:
for (int k = 0; k < N; k += 2) {y[k] = x[k]; y[k + 1] = x[k + 1];}// x[] - signal after transform
for (int k = 0; k < K - 2; k++){ y[k + N] = y[k]; }                 //
for (int Nd2 = N / 2, i1 = 0, i = 0; i < N; i += 2, i1++){         // Input:
  t1 = y[i] * hp[0] + y[i + 1] * hp[1];                             // x[] - input signal (float)
  t2 = y[i] * gp[0] + y[i + 1] * gp[1];                             // N - signal length (int)
  for (int k = 2; k < K; k += 2) {                                  // t1, t2 - temp variables (float)
    t1 += y[i + k] * hp[k] + y[i + k + 1] * hp[k + 1];              // y[] - temporary table (float)
    t2 += y[i + k] * gp[k] + y[i + k + 1] * gp[k + 1];              // hp[] - first filter (float)
  }                                                                 // gp[] - second filter (float)
  x[i1] = t1;  x[i1 + Nd2] = t2;                                    // K - filter length (int)
}                                                                   //
```

4.1.3 ***cpu_dwt_l2_ww*** - sequential, lattice, 2-point, with wrapping

The *cpu_dwt_l2_ww* is the first presented lattice implementation with wrapping (Fig. 6a). It has been developed based on the work of published in papers [3, 9, 15, 16]. This implementation uses, instead of two filters, a single vector of $K/2+1$ samples of factorized two-point base operations with tangent multipliers. Wrapping for butterfly operations out of the array size is implemented with the use of time-efficient helper variables. We will call such an implementation the wrapped one. A single butterfly operation works on two input samples. The

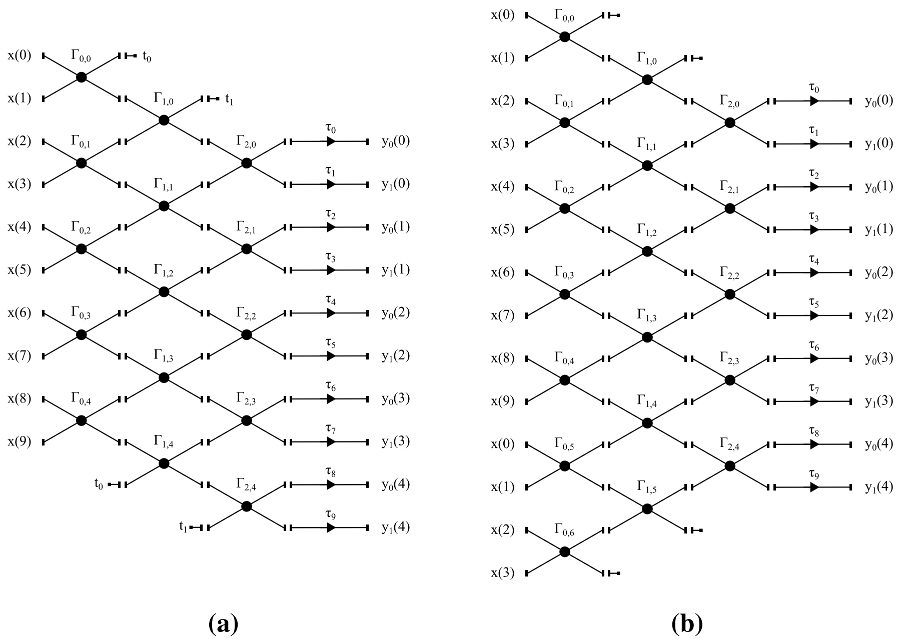


Fig. 6 Lattice model with wrapping **a** and without wrapping **b** for a transformation of size N=10 and filters of length K=6

number of steps in the structure depends on the size of the input vectors and is *K*/2, noting that the last step consists of multiplications only.

Listing 3 Code fragment of the ***cpu_dwt_l2_ww*** implementation.

```
float a, psi; int ind = 0;                                        // Output:
                                                                  // x[] - signal after transform
for (int i = 0; i < N; i += 2)                                    //
{                                                                 // Input:
  y[i] = x[i]; y[i + 1] = x[i + 1];                               // x[] - input signal (float)
}                                                                 // N - signal length (int)
                                                                  // t1, t2 - temp variables (float)
for (int k = 0; k < K / 2 - 1; k++)                               // y[] - temporary table (float)
{                                                                 // hp[] - first filter (float)
  a = wtg[ind++];                                                 // gp[] - second filter (float)
  float t = y[0] + a * y[1];                                      // K - filter length (int)
  y[0] = a * y[0] - y[1];                                         // a, psi - temp variables (float)
  for (int i = 2; i < N - 1; i += 2)                              // wtg[] - tangens factors (float)
  {                                                               //
    y[i - 1] = y[i] + a * y[i + 1];                               //
    y[i] = a * y[i] - y[i + 1];                                   //
  }                                                               //
  y[N - 1] = t;                                                   //
}                                                                 //
                                                                  //
a = wtg[ind++];                                                   //
psi = wtg[ind];                                                   //
                                                                  //
for (int Nd2 = N / 2, i1 = 0, i = 0; i < N; i += 2, i1++)         //
{                                                                 //
  x[i1] = (a * y[i] - y[i + 1]) *  psi;                           //
  x[i1 + Nd2] = (y[i] + a * y[i + 1]) *  psi;                     //
}                                                                 //
```

4.1.4 *cpu_dwt_l2_nw* - sequential, lattice, 2-point, without wrapping

The *cpu_dwt_l2_nw* is modified *cpu_dwt_l2_ww* lattice implementation in which wrapping has been bypassed according to the scheme Fig. 6b involving the extension of the input vector by $K-2$ initial samples. Like the base implementation, the filters were factored to a single vector of $K/2+1$ coefficients of the tangent basis operations. The final step of the structure is combined with the organization of the data in the resulting vector. This is a sequential implementation of the lattice algorithm without wrapping.

Listing 4 Code fragment of the ***cpu_dwt_l2_nw*** implementation.

```
float a, psi, t;                                                   // Output:
int ind = 0, K21 = K / 2 - 1, NK2 = N + K - 2;                     // x[] - signal after transform
                                                                   //
for (int i = 0; i < N; i += 2)                                     // Input:
{                                                                  // x[] - input signal (float)
  y[i] = x[i]; y[i + 1] = x[i + 1];                                // N - signal length (int)
}                                                                  // t1, t2 - temp variables (float)
                                                                   // y[] - temporary table (float)
for (int k = 0; k < K - 2; k++)                                    // hp[] - first filter (float)
{                                                                  // gp[] - second filter (float)
  y[k + N] = y[k];                                                 // K - filter length (int)
}                                                                  // a, psi - temp variables (float)
                                                                   // wtg[] - tangens factors (float)
for (int k = 0; k < K21; k++)                                      //
{                                                                  //
  a = wtg[ind++];                                                  //
  for (int i = k; i < NK2 - k; i += 2)                             //
  {                                                                //
    t = y[i] + a * y[i + 1];                                       //
    y[i + 1] = a * y[i] - y[i + 1];                                //
    y[i] = t;                                                      //
  }                                                                //
}                                                                  //
                                                                   //
a = wtg[ind++];                                                    //
psi = wtg[ind];                                                    //
                                                                   //
for (int Nd2 = N / 2, i1 = 0, i = K21; i < N + K21; i += 2, i1++) //
{                                                                  //
  x[i1] = (a * y[i] - y[i + 1]) *  psi;                            //
  x[i1 + Nd2] = (y[i] + a * y[i + 1]) *  psi;                      //
}                                                                  //
```

4.2 Parallel implementations on GPUs

4.2.1 *gpu_dwt_m1_ww_ref* - parallel, matrix-based, 1-point, with wrapping

The general approach of parallel computing optimization aims to maximize the level of parallelism. For this reason, the first implementation we present and, at the same time, take as a reference for the following ones is a parallel direct matrix with 1 point granularity. This means that a single thread computes one element of the output vector, and the number of parallel threads is N. For such implementation, it is necessary to switch filters h and g for even and odd elements of the vector, respectively. Because the filters are stored in memory one by one, their selection is done with shifting the index by a constant value. The determination of the parity is done by modulo two divisions. The implementation implements a direct matrix approach with wrapping. We consider it as a reference implementation for the others analyzed later.

Listing 5 Kernel code of the ***gpu_dwt_m1_ww_ref*** implementation.

```
float v = 0;                                                    // Output:
int i, k1, k2, id = blockIdx.x*blockDim.x + threadIdx.x;        // y[] - signal after transform
k1 = 2 * (id / 2);                                              //
k2 = (id % 2)*K;                                                // Input:
for (i = 0; i < K; i++){                                        // id - ID (N elements)
  v = v + x[k1] * gh[k2];                                       // K - filter length (int)
  k1 = (k1 + 1) % N;                                            // x[] - input signal (float)
  k2 = k2 + 1;                                                  // gh[] - both filters (float)
}                                                               // N - signal length (int)
y[id] = v;                                                      // v, k1, k2, id - temp variables
```

4.2.2 *gpu_dwt_m2_ww* - parallel, matrix-based, 2-point, with wrapping

The *gpu_dwt_m2_ww* is an optimization of the *gpu_dwt_m1_ww_ref* implementation. Here, a reduced granularity of the computational task partitioning is used. The maximum level of parallelism is lower, it is possible to run max $N/2$ threads in parallel, but each thread is responsible for slightly more computations because it computes two elements of the output vector. There is no need to choose the filter. Both filters are stored in separate arrays h and g. Wrapping is implemented via *"if"* instruction. The implementation is direct matrix approach with wrapping.

Listing 6 Kernel code of the GPU-B implementation.

```
int i1 = (blockIdx.x*blockDim.x + threadIdx.x);                 // Output:
int i = i1 * 2, k, ik;                                          // x[] - signal after transform
float t1 = y[i] * hp[0] + y[i + 1] * hp[1];                     // Input:
float t2 = y[i] * gp[0] + y[i + 1] * gp[1];                     // i1 - ID (N/2 elements)
for (k = 2; k < K; k += 2){                                     // N - signal length
  if ((ik = i + k) >= N) ik -= N;                               // K - filter length (int)
  t1 += y[ik] * hp[k] + y[ik + 1] * hp[k + 1];                  // x[] - input signal (float)
  t2 += y[ik] * gp[k] + y[ik + 1] * gp[k + 1];                  // hp[], gp[] - filters (float)
}                                                               // y[] - temp vector (float)
x[i1] = t1;                                                     // Nd2 - half signal length (int)
x[i1 + Nd2] = t2;                                               // i, k, ik, t1, t2 - temp vars
```

4.2.3 *gpu_dwt_m4_ww* - parallel, matrix-based, 4-point, with wrapping

The *gpu_dwt_m4_ww* implementation is very similar to the implementation *gpu_dwt_m2_ww*. The only difference is once again the reduced level of parallelism. Here, it is possible to run $N/4$ threads, and each thread is responsible for yet more calculations as it determines 4 elements of the output vector. The filters are stored in separate arrays h and g. The implementation uses a direct matrix approach with wrapping via a conditional *"if"* instruction.

Listing 7 Kernel code of the ***gpu_dwt_m4_ww*** implementation.

```
int i1 = 2 * (blockIdx.x*blockDim.x + threadIdx.x);              // Output:
int i = i1 * 2, k, ik;                                           // x[] - signal after transform
float t1 = y[i] * hp[0] + y[i + 1] * hp[1];                      //
float t2 = y[i] * gp[0] + y[i + 1] * gp[1];                      // Input:
float t3 = y[i + 2] * hp[0] + y[i + 3] * hp[1];                  // i1 - ID (N/4 elements)
float t4 = y[i + 2] * gp[0] + y[i + 3] * gp[1];                  // K - filter length (int)
for (k = 2; k < K; k += 2){                                      // x[] - input signal (float)
  if ((ik = i + k) >= N-2) ik -= N;                              // hp[] - first filter (float)
  t1 += y[ik] * hp[k] + y[ik + 1] * hp[k + 1];                   // gp[] - second filter (float)
  t2 += y[ik] * gp[k] + y[ik + 1] * gp[k + 1];                   // N - signal length (int)
  t3 += y[ik + 2] * hp[k] + y[ik + 3] * hp[k + 1];               // Nd2 - half signal length (int)
  t4 += y[ik + 2] * gp[k] + y[ik + 3] * gp[k + 1];               // i, k, ik - temp variables
}                                                                // t1, t2, t3, t4 - temp variables
x[i1] = t1;                                                      // y[] - temp vector (float)
x[i1 + 1] = t3;                                                  //
x[i1 + Nd2] = t2;                                                //
x[i1 + Nd2 + 1] = t4;                                            //
```

4.2.4 *gpu_dwt_m2_nw* - parallel, matrix-based, 2-point, without wrapping

The *gpu_dwt_m2_nw* implementation is similar to the implementation *gpu_dwt_m2_ww*. The major difference is that filter wrapping is bypassed, as in 5(b). The maximum level of parallelism is the same as for *gpu_dwt_m2_ww* because a max limit of $N/2$ parallel threads. The filters are stored in separate arrays. In contrast, there are no conditional instructions or modulo division. To make this possible, the input signal had to be extended by $K-2$ of initial samples, which was done in an efficient native way using the *cudaMemcpy* function. This implementation is a direct matrix approach without wrapping, with granularity labeled by us as 2.

Listing 8 Kernel code of the ***gpu_dwt_m2_nw*** implementation.

```
int i1 = (blockIdx.x*blockDim.x + threadIdx.x);                  // Output:
int i = i1 * 2, k;                                               // x[] - signal after transform
float t1 = y[i] * hp[0] + y[i + 1] * hp[1];                      // Input:
float t2 = y[i] * gp[0] + y[i + 1] * gp[1];                      // i1 - ID (N/2 elements)
for (k = 2; k < K; k += 2){                                      // Nd2 - half signal length (int)
  t1 += y[i + k] * hp[k] + y[i + k + 1] * hp[k + 1];             // K - filter length (int)
  t2 += y[i + k] * gp[k] + y[i + k + 1] * gp[k + 1];             // x[] - input signal (float)
}                                                                // hp[], gp[] - filters (float)
x[i1] = t1;                                                      // y[] - temp vector (float)
x[i1 + Nd2] = t2;                                                // i, k, t1, t2 - temp variables
```

4.2.5 *gpu_dwt_m4_nw* - parallel, matrix-based, 4-point, without wrapping

The *gpu_dwt_m4_nw* implementation is a direct modification of the implementation of *gpu_dwt_m2_nw*. The only difference is the twofold reduction in the maximum level of parallelism. The implementation works on up to $N/4$ threads. Filters without changes are stored in two arrays. Each single thread determines four samples of the output vector. There are no conditional instructions or modulo division. The input signal is extended by $K-2$ initial samples. This implementation is a direct matrix approach without wrapping with granularity we label as 4.

Listing 9 Kernel code of the ***gpu_dwt_m4_nw*** implementation.

```
int i1 = 2 * (blockIdx.x*blockDim.x + threadIdx.x);                    // Output:
int i = i1 * 2, k;                                                      // x[] - signal after transform
float t1 = y[i] * hp[0] + y[i + 1] * hp[1];                             //
float t2 = y[i] * gp[0] + y[i + 1] * gp[1];                             // Input:
float t3 = y[i + 2] * hp[0] + y[i + 3] * hp[1];                         // i1 - ID (N/4 elements)
float t4 = y[i + 2] * gp[0] + y[i + 3] * gp[1];                         // K - filter length (int)
for (k = 2; k < K; k += 2){                                             // x[] - input signal (float)
  t1 += y[i + k] * hp[k] + y[i + k + 1] * hp[k + 1];                    // hp[] - first filter (float)
  t2 += y[i + k] * gp[k] + y[i + k + 1] * gp[k + 1];                    // gp[] - second filter (float)
  t3 += y[i + 2 + k] * hp[k] + y[i + k + 3] * hp[k + 1];                // N - signal length (int)
  t4 += y[i + 2 + k] * gp[k] + y[i + k + 3] * gp[k + 1];                // Nd2 - half signal length (int)
}                                                                       // i, k - temp variables
x[i1] = t1;                                                             // t1, t2, t3, t4 - temp variables
x[i1 + 1] = t3;                                                         // y[] - temp vector (float)
x[i1 + Nd2] = t2;                                                       //
x[i1 + Nd2 + 1] = t4;                                                   //
```

4.2.6 *gpu_dwt_l2_ww* - parallel, lattice-based, 2-point, with wrapping

The *gpu_dwt_l2_ww* is the second last implementation evaluated in this work. However, it is also the first parallel implementation that uses a lattice structure. In general terms, it is a parallel version of the *cpu_dwt_l2_ww* implementation, additionally optimized to run on GPUs. The implementation has maximum available granularity by running on $N/2$ parallel threads. Each thread performs a single butterfly operation. The filters are stored in the form of a single auxiliary array, in which both the low-pass and high-pass filters are properly factorized into two-point base operations with tangent multipliers. It is worth noting that the lattice structure is staged, so $K/2$ kernel function calls are needed. The last stage is different than others, so we added an *"if"* conditional instruction in the kernel code. The kernel is launched sequentially using *for* loop. This is a 2 point parallel implementation of a lattice structure with wrapping.

Listing 10 Kernel code of the ***gpu_dwt_l2_ww*** implementation.

```
int i = 2 * (blockIdx.x * blockDim.x + threadIdx.x);                   // Output:
int i1, i2;                                                             // x[] - signal after transform
if ((i1 = i + k) >= N) i1 -= N;                                         //
if ((i2 = i1 + 1) >= N) i2 -= N;                                        // Input:
float t1 = x[i1];                                                       // x[] - input signal (float)
float t2 = x[i2];                                                       // N - signal length (int)
if (k == K / 2 - 1){                                                    // i - ID (N/2 elements)
  t1 *= psi;                                                            // K - filter length
  t2 *= psi;                                                            // k - current stage from 0 to K/2
  int ii = i1;                                                          // psi - wtg[K / 2]
  i1 = i2;                                                              // a - wtg[/stage/]
  i2 = ii;                                                              // t1, t2 - temp variables
}                                                                       // wtg[] - tangens factors (float)
x[i1] = t1 + a * t2;                                                    //
x[i2] = a * t1 - t2;                                                    //
```

4.2.7 *gpu_dwt_l2_nw* - parallel, lattice-based, 2-point, without wrapping

The last implementation studied, *gpu_dwt_l2_nw*, is a modification of the *gpu_dwt_l2_ww* that is similar to *cpu_dwt_l2_nw*, bypass wrapping as proposed in the previous section. Lattice-based kernel for GPU is utilizing $N/2$ parallel threads. Both *h* and g filters are transformed and stored in an array, in which both low-pass and high-pass filters are properly factorized into two-point basis operations with tangent multiplications. The input signal is extended by $K - 2$ samples to bypass wrapping with the use of native on-gpu *cudaMemcpy* function. The kernel code is short and simple. This implementation is a parallel lattice algorithm without wrapping.

Listing 11 Kernel code of the ***gpu_dwt_l2_nw*** implementation.

```
int i = k + 2 * (blockIdx.x*blockDim.x + threadIdx.x);          // y[] - input and output signal
                                                                // i - ID ((N+K-2)/2 elements)
float t = y[i] + a * y[i + 1];                                  // N - signal size, K -filter size
y[i + 1] = a * y[i] - y[i + 1];                                 // k - current stage from 0 to K/2
y[i] = t;                                                       // a - wtg[/stage/]
```

5 Results of experimental research

In this section, we present the experimental results of research on the effectiveness of the proposed DWT implementations on GPUs. The experiments were performed on a server equipped with a *GeForce RTX 2080 Ti* graphics cards with 4352 CUDA cores each, and an *Intel Xeon Silver 4112* CPU processors with 4 cores (8 threads) each and 256GB of DDR4 RAM memory. Programs were written in C language using *CUDA Toolkit 11.2*. Presented time results are averaged from at least 100 trials. GPU was warmed up before the computations to eliminate the problem of its low performance at first run. To measure the time, we used techniques based on processor cycles. The kernel code time was measured directly on the device using the techniques proposed by the chip manufacturer, namely the *"cudaEventRecord"* function. Total time and CPU time were measured using *"QueryPerformanceCounter"* function. All implementations were designed to work on 32-bit floating point variables and provide numerically exactly the same results what was confirmed for each experiment using few error measures. The knowledge about the execution time of kernel functions is critically important for implementation optimization. There are several ways to measure kernel performance. An excellent solution is to use a dedicated profiler. This toolkit is called `nvprof` and collects detailed information about the timeline of CPU and GPU activity during program execution. It captures details of kernel execution, data transfers and all CUDA API calls. The tool is powerful in helping to understand many of the dependencies, such as the time required for actual computation and the time required for data transfers. For an efficient algorithm implementation, it is essential to balance between communication and computation. If the application spends more time on calculation than on data transfer, it may be possible to overlap these computations and completely hide the delay associated with data transfer. If the application spends less time on calculation than on data transfer, it is important to minimize the transfer between the host and the device. During implementation optimization, it is also possible to determine how the application stands up to the device's theoretical limits. The measured values can be compared to theoretical peak values, and it can be determined whether the application is limited by arithmetic or by memory bandwidth. The device's peak single precision floating operations per second (FLOPS) performance can be determined using the following formula:

$$\mathrm{P} = N_{\mathrm{cl}} * N_{\mathrm{gppu}} * N_{\mathrm{smpgp}} * N_{\mathrm{cpsm}} * N_{\mathrm{ipc}}, \tag{2}$$

where P is FP32 peak performance, N_{cl} is GPU clock, N_{gppd} is number of graphics processors per device, N_{smpgp} is number of streaming multiprocessors per graphics processor, N_{cpsm} is number of cores per streaming multiprocessor and N_{ipc} is number of instructions per cycle. Substituting the data of the *GeForce RTX 2080 Ti* graphics card read directly from device, i.e., $N_{\mathrm{cl}} = 1545\mathrm{MHz}$, $N_{\mathrm{gppd}} = 1$, $N_{\mathrm{smpgp}} = 68$, $N_{\mathrm{cpsm}} = 128$, $N_{\mathrm{ipc}} = 1$, we get the peak performance for the tested GPU:

$$\mathrm{P} = 154\mathrm{MHz} * 1 * 68 * 128 * 1 = 13,447 * 10^6\,\mathrm{MFLOPS} = 13,45\,\mathrm{TFLOPS}.$$

The Peak Memory Bandwidth can be determined in the similar way using the following expression:

$$\mathrm{B} = N_{\mathrm{gppu}} * \left(\frac{N_{\mathrm{mcl}} * N_{\mathrm{mbw}}}{8}\right) * \mathcal{M}, \tag{3}$$

where B is the peak memory bandwidth, N_{gppd} is the number of graphics processors per device, N_{mcl} is the memory clock, N_{mbw} is the memory bus width and $\mathcal{M}$ is memory type multiplayer, for GDDR3 $\mathcal{M} = 2$, for GDDR5 $\mathcal{M} = 4$ and for GDDR6 $\mathcal{M} = 8$. By applying device parameters to the above formula, we get:

$$\mathrm{B} = 1 * \left(\frac{1750 * 352}{8}\right) * 8 = 616000\mathrm{MB/s} = 616\mathrm{GB/s}.$$

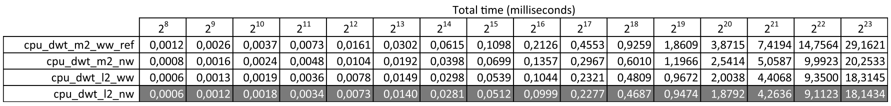
To measure the achieved percentage of utilization with respect to the theoretical maximum, we used the newest *NVIDIA Nsight Compute* toolset. The study clearly revealed that the highest level out of all, in case of computing *SoL*, achieves the *gpu_dwt_m2_nw* implementation with the value of 54% in peak. In the case of the *memory SoL* criterion, the best performing implementation is *gpu_dwt_l2_nw* with a value of 83% in peak. Hence, both leading implementations are non-wrapping ones.

CUDA toolkit provides a very precise timing tool on the GPU side based on events. Despite the fact that the kernel call is asynchronous for the host, the *cudaDeviceSynchronize* command can wait for all parallel threads to finish computations. An event in CUDA is a marker in a CUDA stream associated with a certain point in the flow of operations in that stream. It is possible to measure the elapsed time of CUDA operations marked by two events using the *cudaEventElapsedTime* function. This function returns the elapsed time in milliseconds between two events. The events start and stop do not need to be associated with the same CUDA stream. Such a timing technique is also widely accepted for performance studies in the case of GPUs [5].

In Table 1, we list the most important characteristics of the GPU we've used in our tests. The results in Figs. 7, 8, 9, 10, 11 and 12 show the actual computation times in milliseconds. The columns successively represent the size of the transformation formerly marked as *N*. The size of the input vectors has always been powers

Table 1 Specification of the RTX2080Ti GPU

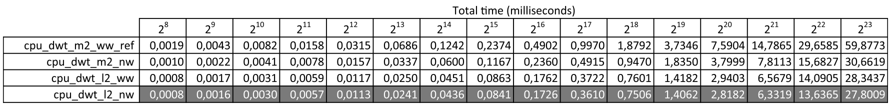
Parameter	Value
Grpahics processor	TU102
CUDA Capability Major/Minor version number:	7.5
Multiprocessors	68
CUDA Cores:	8704
GPU Max Clock rate:	1.54 GHz
Memory Clock rate:	7000 Mhz
Total number of registers available per block:	65536
Maximum number of threads per multiprocessor:	1024
Maximum number of threads per block:	1024
Warp register allocation granularity	64



Total time (milliseconds)

	2^{8}	2^{9}	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
cpu_dwt_m2_ww_ref	0,0012	0,0026	0,0037	0,0073	0,0161	0,0302	0,0615	0,1098	0,2126	0,4553	0,9259	1,8609	3,8715	7,4194	14,7564	29,1621
cpu_dwt_m2_nw	0,0008	0,0016	0,0024	0,0048	0,0104	0,0192	0,0398	0,0699	0,1357	0,2967	0,6010	1,1966	2,5414	5,0587	9,9923	20,2533
cpu_dwt_l2_ww	0,0006	0,0013	0,0019	0,0036	0,0078	0,0149	0,0298	0,0539	0,1044	0,2321	0,4809	0,9672	2,0038	4,4068	9,3500	18,3145
cpu_dwt_l2_nw	0,0006	0,0012	0,0018	0,0034	0,0073	0,0140	0,0281	0,0512	0,0999	0,2277	0,4687	0,9474	1,8792	4,2636	9,1123	18,1434

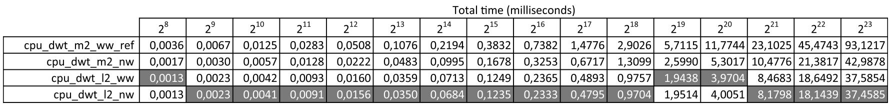
Fig. 7 Time results of sequential implementations on CPU for filter length 4



Total time (milliseconds)

	2^{8}	2^{9}	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
cpu_dwt_m2_ww_ref	0,0019	0,0043	0,0082	0,0158	0,0315	0,0686	0,1242	0,2374	0,4902	0,9970	1,8792	3,7346	7,5904	14,7865	29,6585	59,8773
cpu_dwt_m2_nw	0,0010	0,0022	0,0041	0,0078	0,0157	0,0337	0,0600	0,1167	0,2360	0,4915	0,9470	1,8350	3,7999	7,8113	15,6827	30,6619
cpu_dwt_l2_ww	0,0008	0,0017	0,0031	0,0059	0,0117	0,0250	0,0451	0,0863	0,1762	0,3722	0,7601	1,4182	2,9403	6,5679	14,0905	28,3437
cpu_dwt_l2_nw	0,0008	0,0016	0,0030	0,0057	0,0113	0,0241	0,0436	0,0841	0,1726	0,3610	0,7506	1,4062	2,8182	6,3319	13,6365	27,8009

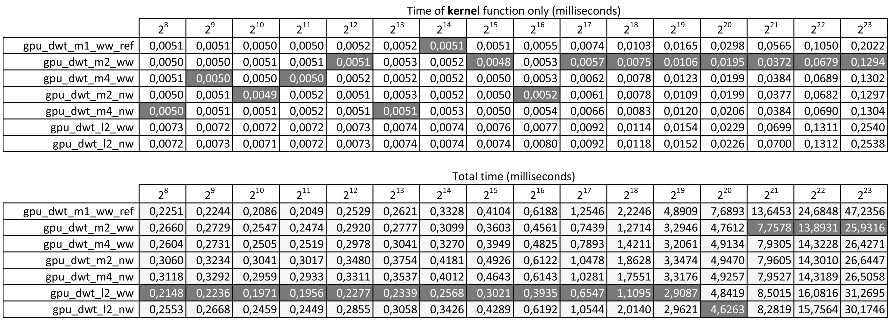
Fig. 8 Time results of sequential implementations on CPU for filter length 8



Total time (milliseconds)

	2^{8}	2^{9}	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
cpu_dwt_m2_ww_ref	0,0036	0,0067	0,0125	0,0283	0,0508	0,1076	0,2194	0,3832	0,7382	1,4776	2,9026	5,7115	11,7744	23,1025	45,4743	93,1217
cpu_dwt_m2_nw	0,0017	0,0030	0,0057	0,0128	0,0222	0,0483	0,0995	0,1678	0,3253	0,6717	1,3099	2,5990	5,3017	10,4776	21,3817	42,9878
cpu_dwt_l2_ww	0,0013	0,0023	0,0042	0,0093	0,0160	0,0359	0,0713	0,1249	0,2365	0,4893	0,9757	1,9438	3,9704	8,4683	18,6492	37,5854
cpu_dwt_l2_nw	0,0013	0,0023	0,0041	0,0091	0,0156	0,0350	0,0684	0,1235	0,2333	0,4795	0,9704	1,9514	4,0051	8,1798	18,1439	37,4585

Fig. 9 Time results of sequential implementations on CPU for filter length 12



Time of **kernel** function only (milliseconds)

	2^{8}	2^{9}	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
gpu_dwt_m1_ww_ref	0,0051	0,0051	0,0050	0,0050	0,0052	0,0052	0,0051	0,0051	0,0055	0,0074	0,0103	0,0165	0,0298	0,0565	0,1050	0,2022
gpu_dwt_m2_ww	0,0050	0,0050	0,0051	0,0051	0,0051	0,0053	0,0052	0,0048	0,0053	0,0057	0,0075	0,0106	0,0195	0,0372	0,0679	0,1294
gpu_dwt_m4_ww	0,0051	0,0050	0,0050	0,0050	0,0052	0,0052	0,0052	0,0050	0,0053	0,0062	0,0078	0,0123	0,0199	0,0384	0,0689	0,1302
gpu_dwt_m2_nw	0,0050	0,0051	0,0049	0,0052	0,0051	0,0053	0,0052	0,0050	0,0052	0,0061	0,0078	0,0109	0,0199	0,0377	0,0682	0,1297
gpu_dwt_m4_nw	0,0050	0,0051	0,0051	0,0052	0,0051	0,0051	0,0053	0,0050	0,0054	0,0066	0,0083	0,0120	0,0206	0,0384	0,0690	0,1304
gpu_dwt_l2_ww	0,0073	0,0072	0,0072	0,0072	0,0073	0,0074	0,0074	0,0076	0,0077	0,0092	0,0114	0,0154	0,0229	0,0699	0,1311	0,2540
gpu_dwt_l2_nw	0,0072	0,0073	0,0071	0,0072	0,0073	0,0074	0,0074	0,0074	0,0080	0,0092	0,0118	0,0152	0,0226	0,0700	0,1312	0,2538

Total time (milliseconds)

	2^{8}	2^{9}	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
gpu_dwt_m1_ww_ref	0,2251	0,2244	0,2086	0,2049	0,2529	0,2621	0,3328	0,4104	0,6188	1,2546	2,2246	4,8909	7,6893	13,6453	24,6848	47,2356
gpu_dwt_m2_ww	0,2660	0,2729	0,2547	0,2474	0,2920	0,2777	0,3099	0,3603	0,4561	0,7439	1,2714	3,2946	4,7612	7,7578	13,8931	25,9316
gpu_dwt_m4_ww	0,2604	0,2731	0,2505	0,2519	0,2978	0,3041	0,3270	0,3949	0,4825	0,7893	1,4211	3,2061	4,9134	7,9305	14,3228	26,4271
gpu_dwt_m2_nw	0,3060	0,3234	0,3041	0,3017	0,3480	0,3754	0,4181	0,4926	0,6122	1,0478	1,8628	3,3474	4,9470	7,9605	14,3010	26,6447
gpu_dwt_m4_nw	0,3118	0,3292	0,2959	0,2933	0,3311	0,3537	0,4012	0,4643	0,6143	1,0281	1,7551	3,3176	4,9257	7,9527	14,3189	26,5058
gpu_dwt_l2_ww	0,2148	0,2236	0,1971	0,1956	0,2277	0,2339	0,2568	0,3021	0,3935	0,6547	1,1095	2,9087	4,8419	8,5015	16,0816	31,2695
gpu_dwt_l2_nw	0,2553	0,2668	0,2459	0,2449	0,2855	0,3058	0,3426	0,4289	0,6192	1,0544	2,0140	2,9621	4,6263	8,2819	15,7564	30,1746

Fig. 10 Time results of parallel implementations on GPU for filter length 4

Time of **kernel** function only (milliseconds)

	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
gpu_dwt_m1_ww_ref	0,0051	0,0049	0,0049	0,0050	0,0050	0,0053	0,0054	0,0056	0,0066	0,0088	0,0136	0,0227	0,0429	0,0813	0,1547	0,3015
gpu_dwt_m2_ww	0,0051	0,0048	0,0050	0,0050	0,0050	0,0051	0,0051	0,0049	0,0056	0,0068	0,0090	0,0139	0,0251	0,0452	0,0832	0,1595
gpu_dwt_m4_ww	0,0050	0,0050	0,0051	0,0050	0,0049	0,0052	0,0051	0,0054	0,0061	0,0075	0,0101	0,0166	0,0289	0,0519	0,0929	0,1755
gpu_dwt_m2_nw	0,0051	0,0048	0,0050	0,0050	0,0049	0,0052	0,0051	0,0050	0,0058	0,0069	0,0096	0,0143	0,0251	0,0450	0,0825	0,1576
gpu_dwt_m4_nw	0,0050	0,0049	0,0049	0,0050	0,0050	0,0052	0,0051	0,0051	0,0058	0,0073	0,0103	0,0156	0,0277	0,0486	0,0870	0,1628
gpu_dwt_l2_ww	0,0129	0,0126	0,0128	0,0128	0,0127	0,0130	0,0130	0,0130	0,0140	0,0161	0,0209	0,0289	0,0440	0,1372	0,2598	0,5053
gpu_dwt_l2_nw	0,0127	0,0126	0,0128	0,0129	0,0127	0,0131	0,0131	0,0130	0,0138	0,0159	0,0215	0,0286	0,0438	0,1371	0,2595	0,5048

Total time (milliseconds)

	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
gpu_dwt_m1_ww_ref	0,2031	0,1993	0,1986	0,2176	0,2201	0,2926	0,3461	0,4660	0,6753	1,1920	2,2040	4,8911	7,7431	13,4139	24,7830	50,0137
gpu_dwt_m2_ww	0,2507	0,2372	0,2412	0,2683	0,2501	0,2975	0,3179	0,3739	0,4791	0,7533	1,2805	3,2815	4,7995	7,8027	13,7549	25,7863
gpu_dwt_m4_ww	0,2375	0,2343	0,2429	0,2613	0,2523	0,3046	0,3261	0,4059	0,5035	0,8018	1,4150	3,1427	4,9393	7,8880	14,1287	26,2831
gpu_dwt_m2_nw	0,2887	0,2832	0,2940	0,3062	0,3087	0,3691	0,3959	0,5318	0,6691	1,1014	1,8732	3,3199	4,9778	8,0590	14,1448	26,4283
gpu_dwt_m4_nw	0,2919	0,2832	0,2879	0,3060	0,3177	0,3709	0,3891	0,4765	0,6251	1,0259	1,7743	3,3061	4,9315	7,9532	14,2255	26,3280
gpu_dwt_l2_ww	0,2081	0,1932	0,1967	0,2054	0,2062	0,2362	0,2870	0,3166	0,4340	0,6728	1,1798	2,9299	4,7654	8,5458	15,9117	31,2307
gpu_dwt_l2_nw	0,2433	0,2339	0,2372	0,2624	0,2635	0,3198	0,3461	0,4461	0,6704	1,1116	2,1572	2,9899	4,6297	8,3923	15,6067	30,2610

Fig. 11 Time results of parallel implementations on GPU for filter length 8

Time of **kernel** function only (milliseconds)

	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
gpu_dwt_m1_ww_ref	0,0055	0,0056	0,0056	0,0055	0,0053	0,0055	0,0057	0,0061	0,0072	0,0104	0,0165	0,0284	0,0547	0,1041	0,2001	0,3913
gpu_dwt_m2_ww	0,0051	0,0051	0,0053	0,0052	0,0051	0,0051	0,0052	0,0055	0,0059	0,0076	0,0112	0,0178	0,0324	0,0586	0,1105	0,2130
gpu_dwt_m4_ww	0,0052	0,0052	0,0053	0,0053	0,0054	0,0053	0,0054	0,0058	0,0065	0,0084	0,0125	0,0216	0,0381	0,0703	0,1283	0,2469
gpu_dwt_m2_nw	0,0052	0,0053	0,0051	0,0052	0,0052	0,0053	0,0054	0,0055	0,0062	0,0079	0,0115	0,0178	0,0323	0,0584	0,1100	0,2121
gpu_dwt_m4_nw	0,0053	0,0053	0,0052	0,0052	0,0053	0,0054	0,0053	0,0054	0,0062	0,0080	0,0115	0,0182	0,0329	0,0590	0,1075	0,2054
gpu_dwt_l2_ww	0,0192	0,0192	0,0189	0,0189	0,0189	0,0189	0,0193	0,0190	0,0192	0,0234	0,0312	0,0435	0,0649	0,2042	0,3885	0,7564
gpu_dwt_l2_nw	0,0189	0,0189	0,0190	0,0190	0,0189	0,0189	0,0192	0,0191	0,0194	0,0232	0,0313	0,0429	0,0643	0,2039	0,3879	0,7558

Total time (milliseconds)

	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
gpu_dwt_m1_ww_ref	0,2292	0,2172	0,2138	0,2205	0,2081	0,2770	0,3610	0,4482	0,7000	1,1428	2,0030	4,4785	7,4594	13,0577	24,5444	47,6990
gpu_dwt_m2_ww	0,2819	0,2685	0,2628	0,2661	0,2485	0,2815	0,3614	0,3618	0,5056	0,7329	1,1511	2,7876	4,3887	7,2603	13,4676	25,7239
gpu_dwt_m4_ww	0,2826	0,2672	0,2603	0,2838	0,2545	0,2875	0,3722	0,3812	0,5410	0,7661	1,3319	2,6737	4,2227	7,2249	13,5871	26,3005
gpu_dwt_m2_nw	0,3300	0,3350	0,3145	0,3245	0,2960	0,3599	0,4493	0,4719	0,7019	1,0230	1,6272	2,8195	4,3451	7,2816	13,7897	26,2522
gpu_dwt_m4_nw	0,3291	0,3285	0,3118	0,3225	0,3143	0,3424	0,4356	0,4617	0,6637	0,9562	1,4858	2,7633	4,1979	7,3116	13,7992	26,3880
gpu_dwt_l2_ww	0,2366	0,2311	0,2235	0,2221	0,2270	0,2436	0,3190	0,3080	0,4455	0,6746	1,1303	2,5806	4,5820	8,4127	16,4364	32,1237
gpu_dwt_l2_nw	0,2872	0,2816	0,2663	0,2794	0,2671	0,3056	0,3974	0,4728	0,6642	1,0073	1,8264	2,6386	4,3837	8,2288	16,0116	31,0270

Fig. 12 Time results of parallel implementations on GPU for filter length 12

of 2, so they are labeled from 2^8 to 2^{23}, for example, 2^{10} is a signal consisting of 1024 samples. All results were divided into groups for filter lengths of 4, 8, 12, respectively. The dark gray background highlights the lowest values in each column. The times were obtained on the basis of 100 repetitions for long input signals and 1000 repetitions for short ones. The time *KERNEL* indicates the time of the kernel function only or, if multiple calls are needed, the time of the sequence of kernel function calls. Total time includes preprocessing, postprocessing and copying data to and from the device. We consistently use the proposed names of the implementations that we discussed in detail in the previous section.

6 Conclusions

6.1 Summary of experimental research on CPU

We have tested four sequential implementations running on the CPU: *cpu_dwt_m2_ww_ref*, *cpu_dwt_m2_nw*, *cpu_dwt_l2_ww*, and *cpu_dwt_l2_nw*. The algorithms used in *cpu_dwt_m2_ww_ref* and *cpu_dwt_m2_nw* are based on the convolutional approach, while in *cpu_dwt_l2_ww* and *cpu_dwt_l2_nw* are based on the lattice structure. We have evaluated the standard implementations with wrapping and the proposed implementations without wrapping. The implementation used in *cpu_dwt_m2_ww_ref* and *cpu_dwt_l2_ww* performs wrapping with the use of conditional instructions or modulo division. In the case of the algorithms *cpu_dwt_m2_nw* and *cpu_dwt_l2_nw*, no wrapping was needed because the modification described earlier. The implementation *cpu_dwt_m2_ww_ref* is considered to be the reference implementation. All mentioned implementations were self-developed and tested.

The first insight confirms the theoretical computational complexity, and the lattice structure is superior to the matrix implementation. When comparing the implementation of *cpu_dwt_m2_ww_ref* to *cpu_dwt_l2_ww*, namely, the classic matrix-based implementation to the lattice structure without wrapping, it turns out that the lattice structure is clearly faster. In fact, average advantage after all the experiments is even larger than expected since it is 2.21×, while the theoretical advantage is at a level of about 2×. The next conclusion is that the reference method *cpu_dwt_m2_ww_ref* can be clearly accelerated with wrapping elimination. The implementation without wrapping is clearly faster than the implementation with wrapping. The speedup after averaging all results is 1.85×, and still the *cpu_dwt_m2_nw* compared to *cpu_dwt_m2_ww_ref* has slightly worse computational and memory complexity.

After evaluation of the lattice implementation without wrapping, *cpu_dwt_l2_nw*, it turns out that also in the case of the lattice implementation, bypassing the wrapping allows to speed up the computations, but speedup this time is minimal and is close to 3%. However, it is enough to make the *cpu_dwt_l2_nw* implementation the most efficient among all tested ones.

This part can be summarized in two points. First, for the implementation of the DWT for CPUs, the non-wrapping algorithms indicate better performance than the algorithms with wrapping. Second, the lattice algorithm without wrapping is the most efficient one tested, although its advantage over the lattice algorithm with wrapping and the optimized matrix algorithm is quite low.

6.2 Summary of experimental research on GPU

We have prepared seven parallel implementations profiled for GPUs. These implementations are *gpu_dwt_m1_ww_ref*, *gpu_dwt_m2_ww*, *gpu_dwt_m4_ww*, *gpu_dwt_m2_nw*, *gpu_dwt_m4_nw*, *gpu_dwt_l2_ww*, and *gpu_dwt_l2_nw*. The algorithms used in *gpu_dwt_m1_ww_ref*, *gpu_dwt_m2_ww*, *gpu_dwt_m4_ww*, *gpu_dwt_m2_nw*, *gpu_dwt_m4_nw* are implementations of the traditional convolutional algorithm, while for the implementations of *gpu_dwt_l2_ww* and *gpu_dwt_l2_nw*, a lattice structure was used.

We have studied the traditional implementations with wrapping and the proposed implementations without wrapping. Implementations with wrapping are *gpu_dwt_m1_ww_ref*, *gpu_dwt_m2_ww*, *gpu_dwt_m4_ww*, *gpu_dwt_l2_ww* and without wrapping are *gpu_dwt_m2_nw*, *gpu_dwt_m4_nw* and *gpu_dwt_l2_nw*. In addition, we have also studied the granularity aspect of partitioning the computational task by reducing the overall level of parallelism by charging individual threads with more computations. Thus, we came up with implementations using 1-point granularity: *gpu_dwt_m1_ww_ref*, 4-point granularity: *gpu_dwt_m4_ww* , *gpu_dwt_m4_nw* and 2-point granularity: *gpu_dwt_m2_ww*, *gpu_dwt_m2_nw*, *gpu_dwt_l2_ww*, *gpu_dwt_l2_nw*. As reference approach, we have chosen the *gpu_dwt_m1_ww_ref* implementation because it is closest to the traditional parallel implementation of the DWT algorithm in matrix-based form.

This time, we measured the total time as well as the time of the kernel function itself. The approach of presenting only kernel time is well known there, because in future, all computations can hopefully be performed internally within the GPU. However, currently, the GPU and CPU cooperation is still the main solution. Therefore, we believe that for our implementations, it is also appropriate to present the total processing time, because, e.g., for an implementation without wrapping, the input signal has to be extended, which, although it is already performed on the GPU by simple memory copying operations, does not belong to the kernel code but is a fundamental part of the algorithm.

Looking only at the kernel code time, one can easily see a very interesting relationship. Implementations *gpu_dwt_l2_ww* and *gpu_dwt_l2_nw* which are both tested variants of the lattice algorithm have significantly lower performance than any implementation of the matrix-based algorithm. The loss of the lattice algorithm is very noticeable and what is more, it grows as the amount of processed data increases. This is due to the design of the lattice algorithm for which synchronizations after each stage are essential. Thus, the cost of organizing the computations is so large that it has a greater impact on the total time than the computations themselves. A comparison of kernel codes for the *gpu_dwt_l2_ww* and *gpu_dwt_l2_nw*, i.e., the lattice structure with and without wrapping, indicates almost identical performance which only confirms the earlier point.

In terms of matrix-based implementations, the versions with wrapping, without wrapping, with 2 point granularity, and with 4 point granularity namely *gpu_dwt_m2_ww*, *gpu_dwt_m2_nw*, *gpu_dwt_m4_ww*, *gpu_dwt_m4_nw*, respectively, have nearly identical kernel code performance. One can see a minimal advantage of the optimized 2-point implementation without wrapping, i.e., *gpu_dwt_m2_ww*. In addition, changing the granularity to 4-point, which means reducing the total number of threads working in parallel, does not positively affect the computation time. Finally, it is worth mentioning that the reference implementation, i.e., *gpu_dwt_m1_ww_ref* is noticeably slower than the proposed implementations with 2-point granulation. The device limits expressed in TFLOPS are most closely approached by the *gpu_dwt_m4_nw* implementation, this is due to the high ratio of instructions in the kernel code to execution time.

In the case of the total processing time where preprocessing, postprocessing and all copy times to and from the device are included, the situation is slightly different. First, it should be noted that the total time is shorter than the CPU computation time only for large transformations and long filters slightly exceeding 30% speedup. Of course, the advantage would be clear when using filters with a much more samples, but we focused on the practical cases, so the filters of lengths of 4, 8 and 12 points were chosen. The results are very encouraging because it is clear that after the shift to 2D signals, the use of GPU will surely be beneficial.

The first conclusion is that total times for all implementations are very close to each other. Only the 1-point (reference) implementation, i.e., *gpu_dwt_m1_ww_ref* is, on average, a slightly slower, however, this did not stop it from being very competitive in the case of short input signals. The 4-point granulation applied to the *gpu_dwt_m4_ww* and *gpu_dwt_m4_nw* implementations similarly to the kernel time itself did not provide a clear advantage, however again, the times are very close. When considering total time, both lattice implementations only perform better than the reference implementation. The simple matrix form, although characterised by higher computational complexity, is faster. The times of all the proposed matrix implementations are very similar. Bypassing wrapping, in each case, gives at least a minimal advantage. The dependencies gained on the CPU do not carry over to the GPU, so formulating such clear conclusions as for the CPU is not possible. There is no single best solution, but the results presented should help one to choose the most beneficial implementation for a specific computational problem.

6.3 Perspective on further research

Based on the presented conclusions, we infer that it undoubtedly seems to be an interesting direction to study the developed implementations for 2D signals. This will surely be the part of our future research on the performance of DWT implementations on GPUs. Once again, it is interesting to focus on lattice structure and matrix-based implementations, because from the computational point of view, they represent inherently different approaches in terms of their computational structures when implemented on GPUs.

References

1. Porwik P (2015) Wybrane metody cyfrowego przetwarzania sygnalow z przykladami programow w Matlabie. Wydawnictwo Uniwersytetu Slaskiego, Katowice
2. Sorensen H (2012) "High-Performance Matrix-Vector Multiplication on the GPU", M. Alexander et al. (Eds) Euro-Par 2011 Parallel Processing Workshops, Lecture Notes in Computer Science, vol 7155. Springer, Berlin, Heidelberg
3. Yatsymirskyy M, Stokfiszewski K (2012) "Effectiveness of lattice factorization of two-channel orthogonal filter banks", 2012 Joint Conference New Trends In Audio, Video And Signal Processing: Algorithms. Architectures, Arrangements And Applications (NTAV/SPA), pp 275–279
4. Sanders J, Kandrot E (2010) CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, USA (**ISBN: 978-0-13-138768-3**)

5. Cheng J, Grossman M, McKercher T (2014) Professional CUDA C programming. John Wiley & Sons, Inc., Indianapolis (**ISBN: 978-1-118-73932-7**)
6. Barlas G (2015) Multicore and GPU programming: an integrated approach. Morgan Kaufmann Publishers Inc., USA (**ISBN: 9780124171404**)
7. Gomes J, Velho L (2015) From fourier analysis to wavelets. Springer, Switzerland (**ISBN: 978-3-319-22074-1**)
8. Ahmed UN, Rao KR (1974) Orthogonal transforms for digital signal processing. Springer-Verlag, New York (**ISBN 0-387-06556-3**)
9. Yatsymirskyy M (2011) A novel matrix model of a two channel bank of orthogonal filters. Methods Appl Comput Sci 1(26):205–212
10. Galiano V, López O, Malumbres MP, Migallón H (2013) Parallel strategies for 2D Discrete Wavelet Transform in shared memory systems and GPUs. J Supercomput 64:4–16
11. Zhao D (2015) Fast filter bank convolution for three-dimensional wavelet transform by shared memory on mobile GPU computing. J Supercomput 71:3440–3455
12. van der Laan W, Jalba A, Roerdink J (2011) Accelerating wavelet lifting on graphics hardware using CUDA. IEEE Trans Parallel Distrib Syst 22(1):132–146
13. Harris M, Sengupta S, Owens JD (2007) “Parallel prefix Sum (Scan) with CUDA”, in H. Nguyen (Ed.), GPU Gems 3, Part VI: GPU Computing, Addison Wesley, pp. 851-876
14. Sengupta S, Lefohn AE, Owens JD (2006) “A Work-Efficient Step-Efficient Prefix Sum Algorithm”, Proc. Workshop on Edge Comp. Using New Commodity Architectures, pp. D-26-27
15. Yatsymirskyy M (2009) Lattice structures for synthesis and implementation of wavelet transforms. J Appl Comput Sci 17(1):133–141
16. Yatsymirskyy M, Shaleva O (2012) Simplified lattice factorization of two-channel orthogonal filter bank. Model Inform Technol 64:149–156
17. Puchala D, Stokfiszewski K, Wieloch K, Yatsymirskyy M (2018) “Comparative study of massively parallel GPU realizations of wavelet transform computation with lattice structure and matrix-based approach”, 2018 IEEE Second International Conference on Data Stream Mining and Processing (DSMP), pp. 88-93
18. van der Laan WJ, Jalba AC, Roerdink JBTM (2011) Accelerating wavelet lifting on graphics hardware using CUDA. IEEE Trans Parallel Distrib Syst 22(1):132–146
19. Wieloch K, Stokfiszewski K, Yatsymirskyy M (2017) “Effectiveness of partitioning strategies of Fast Fourier Transform in GPU implementations,” 2017 12th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT), pp. 322-325
20. Stratton JA et al. (2012) “Optimization and architecture effects on GPU computing workload performance,” 2012 Innovative Parallel Computing (InPar), pp. 1-10
21. Akl SG (1997) Parallel computation. Models and methods. Prentice Hall, Upple Saddle River
22. Czech Z (2010) “Wprowadzenie do obliczeń równoległych”, Wydawnictwo Naukowe PWN, Warszawa, ISBN: 978-83-01-16384-6
23. Bakhoda A, Fung WL, Wong H, Yuan GL (2009) “Analyzing CUDA workloads using a detailed GPU simulator”, ISPASS 2009 - International Symposium on Performance Analysis of Systems and Software, pp.163-174
24. Arafa Y, Badawy AH, Chennupati G, Santhi N, Eidenbenz S (2019) “PPT-GPU: Scalable GPU performance modeling”, IEEE Computer Architecture Letters
25. Sunpyo H, Hyesoon K (2009) An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. ACM SIGARCH Comput Architect News 37:152–163
26. Luo C, Suda R (2011) “An execution time prediction analytical model for GPU with instruction-level and thread-level parallelism awareness”, IPSJ SIG Tech. Report, vol. 2011-HPC-130, no. 19, pp. 1-9
27. Baldini I, Fink SJ, Altman E (2014) “Predicting GPU Performance from CPU Runs Using Machine Learning,” 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, Jussieu, pp. 254-261
28. Malhotra G, Goel S, Sarangi S (2015)“GpuTejas: A parallel simulator for GPU architectures” 2014 21st International Conference on High Performance Compting, HiPC
29. Punniyamurthy K, Boroujerdian B, Gerstlauer A (2017) “GATSim: Abstract timing simulation of GPUs,” Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, pp. 43-48

30. Puchala D, Stokfiszewski K, Wieloch K (2021) "Execution time prediction model for parallel GPU realizations of discrete transforms computation algorithms", Bulletin of the Polish Academy of Sciences: Technical Sciences, no. 70
31. Stolarek J (2011) Adaptive synthesis of a wavelet transform using fast neural network. Bullet Polish Acad Sci: Tech Sci 59:9–13
32. Strang G, Nguyen T (1999) Wavelets and filter banks. Wellesley-Cambridge Press, Cambridge
33. Shahbahrami A (2012) Algorithms and architectures for 2D discrete wavelet transform. J Supercomput 62:1045–1064
34. Wang Z, Liu Y, Chiu S (2016) An efficient parallel collaborative filtering algorithm on multi-GPU platform. J Supercomput 72:2080–2094
35. Zhao D, Jinhang Y (2015) Efficiently solving tri-diagonal system by chunked cyclic reduction and single-GPU shared memory. J Supercomput 71(2):369–390
36. Lyons RG (2010) "Wprowadzenie do cyfrowego przetwarzania sygnalow", Wydawnictwa Komunikacji i Lacznosci, Warszawa, ISBN: 978-83-206-1764-1
37. Al-Ayyoub M, Abu-Dalo Y, Jararweh M, Jarrah M (2015) A GPU-based implementations of the fuzzy C-means algorithms for medical image segmentation. J Supercomput 71(8):3149–3162
38. Franco J, Bernabe G, Fernandez J, Acacio ME (2009) "A parallel implementation of the 2D Wavelet Transform Using CUDA", 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, Weimar, pp. 111-118
39. Cheng L, Reiji S (2011) An execution time prediction analytical model for GPU with instruction-level and thread-level parallelism awareness. Summer United Workshops on Parallel, Distributed and Cooperative Processing, Hakoshima, Japan
40. Wojciechowski A, Galaj T (2016) GPU supported dual quaternions based skinning. In: Wojciechowski A, Napieralski P (eds) Computer game innovations. Lodz University of Technology Press, Lodz, pp 5–23