ISSN 2395-8618

Parallel Implementation of db6 Wavelet Transform

Eduardo Rodriguez-Martinez, Cesar Benavides-Alvarez, Fidel Lopez-Saca, Carlos Aviles-Cruz, and Andrés Ferreyra-Ramirez

***Abstract*—This work describes a data-level parallelization strategy to accelerate the discrete Wavelet transform, which was implemented and compared in two multi-threaded architectures, both with shared memory. The first considered architecture was a multi-core server and the second one was a graphic processing unit. Comparisons were based on performance metrics (i.e. execution time, speeedup, efficiency, and cost) for five decomposition levels of the DWT Daubechies db6 over random arrays of length 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, and 10^9. Execution times in our proposed GPU strategy were around 1.2×10^{-5} seconds, compare to 3501×10^{-5} seconds of the sequential implementation. On the other hand, the maximum achievable speedup and efficiency was reached by our proposed multi-core strategy for a number of assigned threads equal to 32.**

***Index Terms*—Wavelet transform, brain-computer interface, OpenMP, db6, CUDA, GPU.**

I. Introduction

Development of brain-computer interfaces (BCI) for detection of movement intention (MI) has focused on the design of descriptors that allow a better characterization of the electroencephalographic signals (EEG)[1]. The most successful descriptors are based on the power spectrum, which is calculated using the short-time Fourier transform (STFT) [2]. However, the temporal and spectral resolution of the STFT are highly dependent on the sample length, the number of coefficients, and other parameters. The Discrete Wavelet Transform (DWT) overcomes the limited resolution of the STDT by applying a cascading set of orthonormal filters. The coefficients of these filters describe a characteristic finite-length pulse called mother wavelet. At each level in the cascading structure, the mother wavelet is stretched or compressed at different scales, leading to a particular time-frequency representation at each level.

In the analysis of EEG signals, the DWT has been used to remove noise cased by involuntary facial gestures [3], [4], for pinpointing the source of chronic stress [5], and for epileptic seizures identification [6]. The use of DWT in portable computing applications is attractive, since it can be implemented in devices with low power consumption, without losing computing capacity. Such combination would eliminate the compromise between power consumption and performance, currently affecting most of the BCI systems. Widespread strategies to eliminate the aforementioned compromise tend to detect the start of MI by means of a separate unit, that is independent of the BCI and acts in real time, thus avoiding turning on the BCI every time there is a false positive. For instance, the work in [7] presents a Wavelet-based strategy that reduces MI onset detection time (OdT) to three seconds, however, in lag-sensitive applications [8], [9], [10] one expects the OdT to be below one second. On the other hand, the work in [7] not only uses the DWT to characterize the EEG signal, but it also forms a feature vector composed of auto-regression coefficients and FFT coefficients. Said vector is projected onto a space with better discrimination and lower dimensionality using Fisher's discriminant analysis method (FDA). Such descriptor is used to train a classifier which produces a reliable MI-start label.

The DWT has also been applied to image compression [11], where the most popular family has been the Daubechies for its orthogonal nature, and for the ability of its members to locate specific frequency bands, grouping them into well-defined segments [12]. The conventional method to implement a DWT is through a set of quadrature filters, where the low-pass and high-pass filters are referred as h and g, respectively. The Daubechies family members are named according to the number of coefficients in their filters, e.g. D8, alternatively db4, has 8 coefficients in each filter. Commonly used members in medical image compression and texture analysis are the db4 and the db6, as these achieve better compression levels with less complex implementation [13]. A low-cost implementation of the Daubechies family is presented in [14], which modifies the popular lifting algorithm [12] with the use of an integer polyphase matrix, reducing arithmetic operations by almost a factor of two by avoiding the use of multipliers.

This work proposes a data-level parallelization strategy to accelerate computation of the DWT for Dabubechies family. Said strategy was implemented and compared in two multi-threaded architectures, both with shared memory. The first considered architecture was a multi-core server, where one or more processes are assigned to each core. The second architecture was a graphic processing unit (GPU), programmed using CUDA. Comparison metrics were based on execution times for five decomposition levels of the DWT Daubechies

Manuscript received on 18/06/2019, accepted for publication on 20/08/2019.

The authors are with the Universidad Autónoma Metropolitana, Unidad Azcapotzalco, Mexico ({erm,cesarbenavides,caviles,fra}@azc.uam.mx).

ISSN 2395-8618

db6 over random arrays of length 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, and 10^9.

The organization of this paper is as follow. Section II details the proposed methodology. Section III shows the experimental results. Finally, section IV argues on the limitations of this work and presents courses for future actions.

II. Methodology

The DWT heart consists of the iterative application of a pair of orthonormal filters, a low-pass filter $g = [g_0, g_1, g_2, \ldots, g_{m-1}]$ and a high-pass filter $h = [h_0, h_1, h_2, \ldots, h_{m-1}]$, defined by the coefficients of their respective impulse response, $\{g_j\}$ and $\{h_j\}$, $j = 0, 1, 2, \ldots, m-1$. Figure 1 shows the cascade structure of the DWT for three decomposition levels.

Given the input signal $x = [x_0, x_1, x_2, \ldots, x_{n-1}]$, with $n >> m$, its discrete Wavelet transform at the k-th decomposition level is given by the approximation and detail coefficients, A_k and D_k, respectively defined as

$$\begin{aligned} A_k &= (\downarrow 2)\left(g * A_{k-1}\right), \\ D_k &= (\downarrow 2)\left(h * A_{k-1}\right), \end{aligned}$$

where $k = 1, 2, \ldots, L$, $A_0 = x$, $(\downarrow 2)(y)$ indicates subsampling of signal y, and $(w * z)$ denotes convolution between the discrete signals w and z.

At each decomposition level, the response of both filters to the input A_{k-1} has to be calculated as the convolution with their respective impulse responses. Let $y = g * A_{k-1}$ be the response of filter g to the input A_{k-1}, with length $p = n + m - 1$. Each element of y can be expressed as

$$y[i] = \sum_{j=0}^{m-1} g[j] A_{k-1}[i-j], \tag{1}$$

where $y[j] = y_j$ is the j-th element of array y. We can observe in Eq. (1) that each $y[i]$ only depends on m elements of A_{k-1}, consequently, it is possible to parallel compute all elements of y. Such parallelization can be used to compute the response of both filters to the input A_{k-1}, thus getting at once a single decomposition level in the DWT. To implement db6 DWT using the cascade structure, simply set the coefficients of each filter as shown in Table I.

The following subsections describe two implementations of the db6 DWT, which differ in the parallel strategy adopted to compute the elements of y. The first implementation was carried out in a multi-core architecture with shared memory, where the number of simultaneously executed threads is equal to the number of available cores. The second implementation was carried out in a GPU, where it is possible to simultaneously execute as many processes as operations are needed. Each implementation takes advantage of the unique characteristics of the architecture to maximize throughput.

A. Multi-core Server Strategy

The strategy designed for the multi-core architecture is shown in Figure 2 as a block diagram. It consists of four operations inside a main loop. Each iteration in the main loop performs one-level decomposition of signal A_{k-1}. The first operation performs padding on A_{k-1} by concatenating the last m elements of A_{k-1} at its head and the first m elements of A_{k-1} at its tail. Such padding scheme was used to implement circular convolution as described in [15].

The second operation in the main loop of Figure 2 performs parallel convolution on the padded signal. It creates t threads, each of which computes q elements of the convolution sequence y, so that the relationship $p = qt + r$ is met, where $r \in \mathbb{Z}_+$ is the amount of extra operations assigned to the first r threads. The k-th thread computes $y[i], \forall i \in E_k = \{k-1, t+k-1, 2t+k-1, \ldots, p-t+k-1\}$, such that $|E_k| = q, \forall k \in \{1, 2, 3, \ldots, t\}$, when $r = 0$. On the other hand, when $0 < r < t$, the first r threads additionally compute the elements $y[j]$, $j = qt+1$, $qt+2$, $\ldots$, $qt+r$. Each thread computes the convolution sequence for both filters, the low-pass filter response is stored in array R_g and the high-pass filter response is stored in array R_h.

The third block carry out subsampling on the filter responses to the padded signal. Padding is removed after subsampling, discarding the first and last m elements of R_g and R_h, leading into the approximation coefficients A_k and the detail coefficients D_k. Finally, the fourth operation, depicted as block number four in Figure 2, saves A_k and D_k.

B. Graphic Processing Unit Strategy

Figure 3 shows our proposed GPU strategy. It mainly differs from our multi-core strategy in how convolution is computed. Eq. (2) presents the symmetrical version of Eq. (1). It is called symmetrical because $m/2$ elements before and after $A_{k-1}[i]$ are required to compute $y[i]$:

$$y[i] = \sum_{j=0}^{m-1} g[m-1-j] A_{k-1}\left[j - \frac{m-1}{2} + i\right]. \tag{2}$$

There is an extra requirement in order for Eq. (1) and Eq. 2 to produce the same convolution sequence, both convolutions must be circular. Therefore, padding A_{k-1} is needed again but only m elements are added. As can be seen in Figure 3, padding concatenates the last $m/2$ elements of A_{k-1} at its head, and the first $m/2$ elements of A_{k-1} at its tail.

Once padding is done, we compute y using as many threads as elements in the convolution sequence, since each thread computes one element of $y[i]$. Threads belong to computational units called CUDA blocks. The number of CUDA blocks b is automatically computed using the relationship $p = ab + c$, where a is the number of threads per block, and $c \in \mathbb{Z}_+$ is the amount of extra operations assigned to the last CUDA block. The computational model in CUDA requires to build a grid of blocks.

ISSN 2395-8618

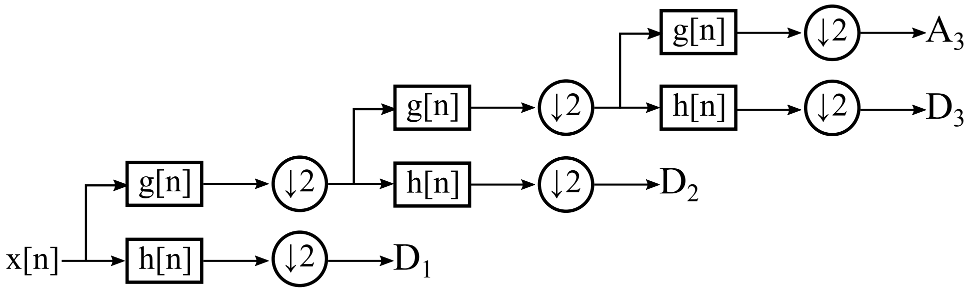


Fig. 1. DWT cascade structure for three decomposition levels

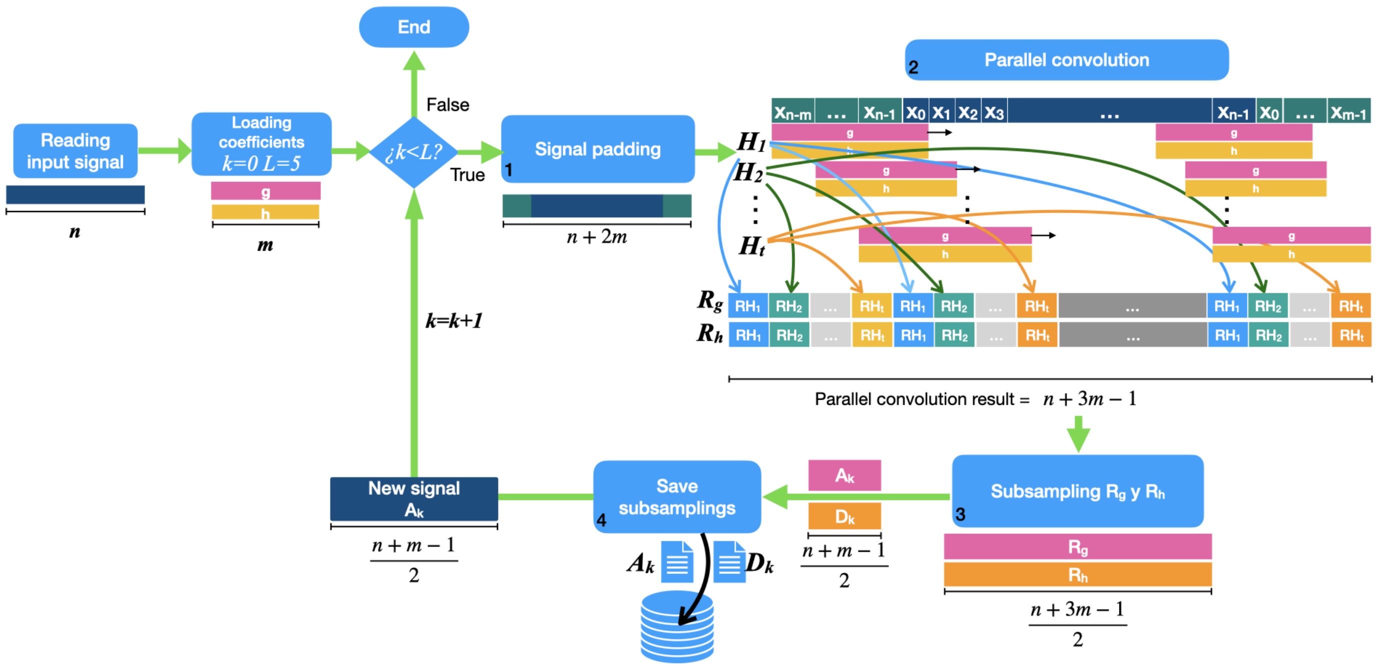


Fig. 2. Multi-core strategy for parallel DWT

TABLE I
FILTER COEFFICIENTS NEEDED TO IMPLEMENT DB6 DWT

$j =$	0	1	2	3	4	5
h_j	−0·1115407434	0·4946238904	−0·7511339080	0·3152503517	0·2262646940	−0·1297668676
g_j	−0·0010773011	0·0047772575	0·0005538422	−0·0315820393	0·0275228655	0·0975016056
$j =$	6	7	8	9	10	11
h_j	−0·0975016056	0·0275228655	0·0315820393	0·0005538422	−0·0047772575	−0·0010773011
g_j	−0·1297668676	−0·2262646940	0·3152503517	0·7511339080	0·4946238904	0·1115407434

TABLE II
DEVICE GEFORCE GTX 1080

CUDA Driver Version / Runtime Version	11.0 / 10.2
CUDA Capability	6.1
Total amount of global memory	8 GB
(20) Multiprocessors, (128) CUDA Cores/MP	2560 CUDA Cores
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)

ISSN 2395-8618

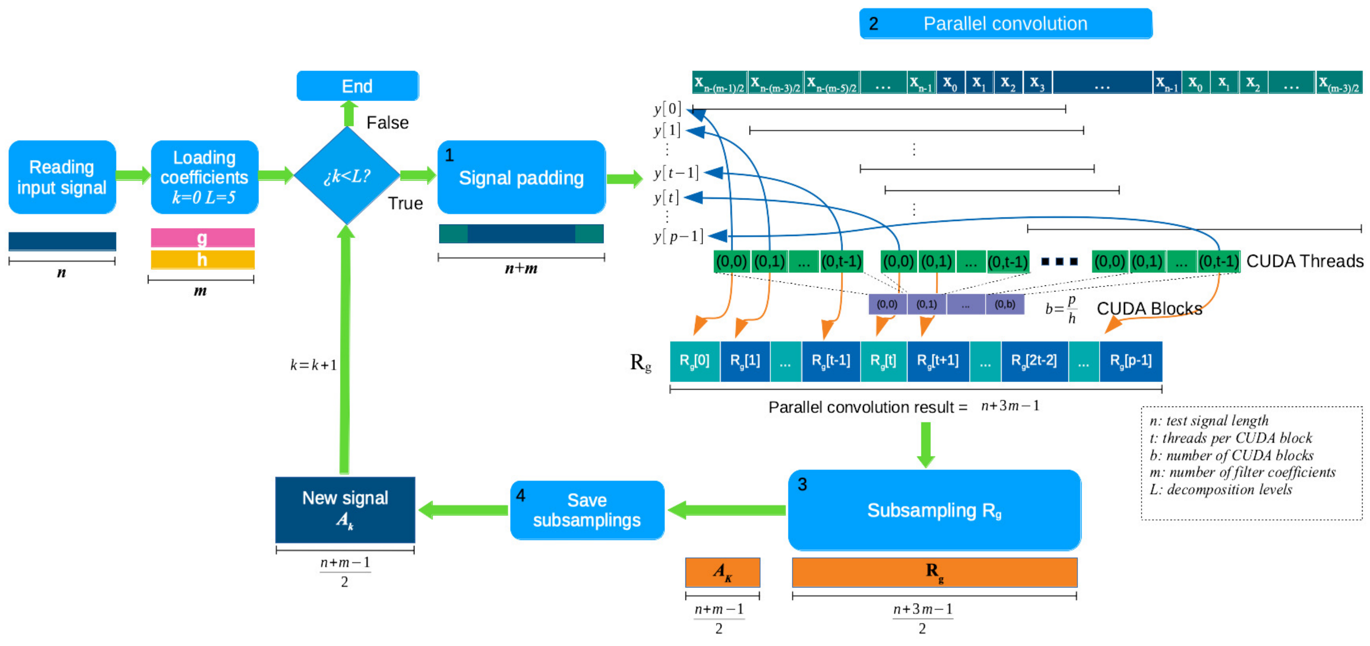


Fig. 3. GPU strategy for parallel DWT

Each block is modeled as a grid of threads. For this work purposes, we consider one-dimensional grids for both blocks and threads. Therefore each block is a one-dimensional array of threads. In Figure 3 we can see an example of such grid.

Eq. (2) also can be used to compute the high-pass filter response, substituting the filter coefficients g with those of h. Thus, each thread computes one element of both responses, which are stored in R_g and R_h, respectively. The following stages in our proposed GPU strategy are similar to those of the multi-core strategy with the exception that subsampling only removes $m/2$ samples from the start and end of both R_g and R_h to produce A_k and D_k, respectively.

III. Experimental Setup and Results

The parallel architecture selected to implement the proposed multi-core strategy consisted of a PowerEdge T630 server with two Intel Xeon E5-2670 processors and 70 GB in RAM. Each processor hosts 19 cores and can simultaneously manage up to 48 active threads. The GPU strategy was implemented on an NVIDIA GeForce GTX 1080 with the characteristics listed in Table II. All programs were implemented using C language, the OpenMP API for the multi-core strategy, and the CUDA API for the GPU strategy.

A. Parallel Convolution Performance and Scalability

Since parallel convolution is the core of both proposed strategies, we decided to analyze its performance using three common metrics, namely speedup, efficiency and cost. Additionally, we measured the scalability of parallel convolution by computing each of the aforementioned metrics over a set of test signals with lengths varying from 10^3 to 10^9 with a unit increment in the exponent. The test signals were built using a Gaussian random number generator with zero mean and unit covariance.

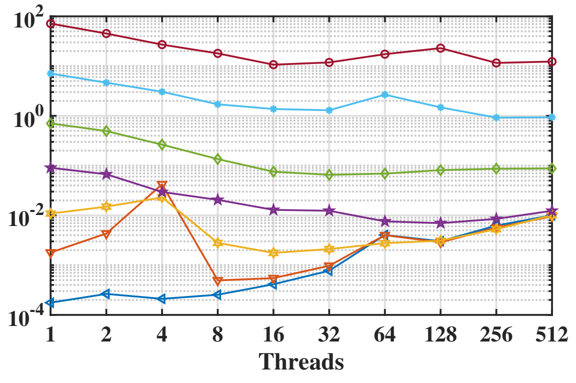
The parallel convolution algorithm in each proposed strategy was used to compute the convolution of each test signal with the impulse response of the low-pass filter g detailed in Table I. Since performance of such algorithm depends on the number of assigned threads t, we decided to compute the aforementioned metrics as t changes from 2 to 512.

1) Multi-core Strategy: The parallel convolution performance and scalability for the proposed multi-core strategy are described by the plots shown in Figure 4. As expected, execution times are fairly linear, the more threads are assigned to the algorithm, the faster it finishes. Regarding speedup, the algorithm displays a sublinear behavior for most of the test signals, with exception of those with length 10^3 and 10^4.

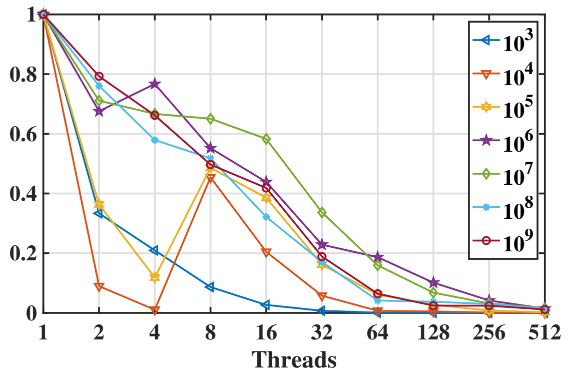
Considering efficiency measures the fraction of time for which a thread is active, the parallel convolution algorithm assigns less work to a given thread as the number of threads increases, thus decreasing the time each thread remains active. The efficiency becomes almost linear as the test signal length increases.

Based on the previous results, we can say the proposed parallel convolution algorithm scales relatively well to the input size, as the efficiency increases with the input size for a given number of assigned threads. Additionally, the algorithm can be made cost-optimal by adjusting the number of assigned threads and the input size. Cost-optimality occurs when the

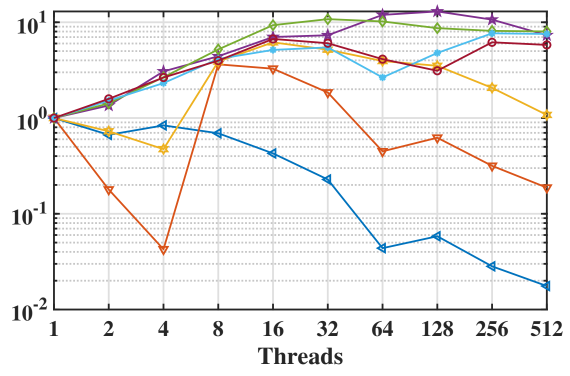
ISSN 2395-8618



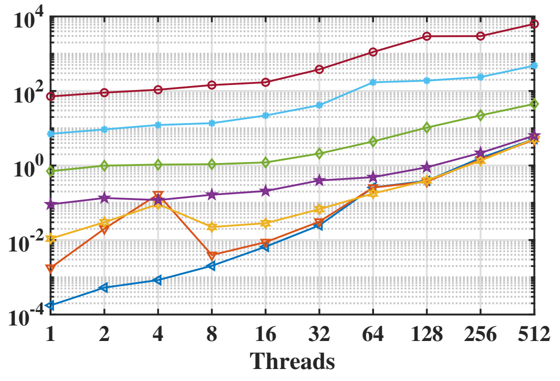
(a) Execution time (sec.)



(b) Efficiency



(c) Speedup



(d) Cost

Fig. 4. Scalability and performance analysis of parallel convolution for the multi-core strategy. The number of threads t assigned to compute the convolution sequence is shown in the x-axis. The legend in (b) shows the color and marker used to plot the results for each test signal

algorithm is assigned between 32 to 64 threads, and the test signal length is greater than 10^6.

2) Graphic Processing Unit Strategy: The fundamental difference between the multi-core strategy and the GPU strategy lies in how the total number of convolution elements p are divided among threads. In the multi-core strategy each thread must compute $q = p/t$ elements of y, while in the GPU strategy each thread computes one element of y, thus we need to create p threads at once. Threads are grouped and executed in CUDA blocks, which in turn are assigned and executed in multiprocessors [16]. We can change the amount of work performed by each multiprocessor varying the number of assigned threads t.

The performance and scalability analysis of the parallel convolution algorithm in our proposed GPU strategy followed the same design as in the multi-core strategy, but due to constraints in the memory size of the selected GPU, the maximum test signal length was 10^8.

The analysis results are detailed by the plots in Figure 5, where instead of changing the number of threads assigned to the parallel convolution algorithm, we changed the number of threads t per CUDA block. Clearly, execution times significantly decrease when compared to the parallel convolution in our multi-core strategy. Nonetheless, all the curves in Figure 5(a) show similar behavior and range within the same interval, pointing out not significant difference between the amount of work done by the parallel convolution as the test signal length increases.

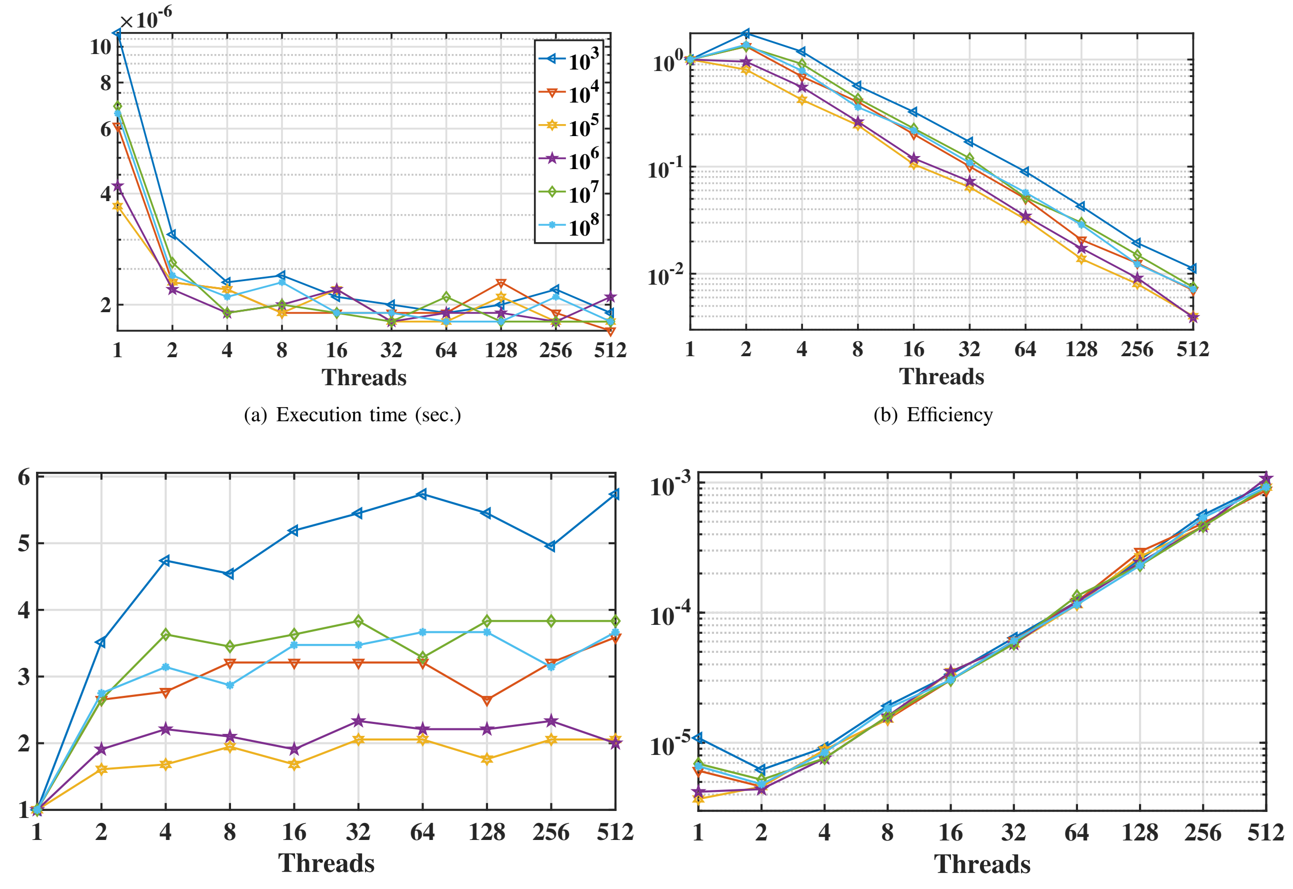
Despite speedup is not as impressive as in the multi-core strategy, where almost a 10 fold gain can be achieved, the parallel convolution algorithm behavior is sublinear for all the test signals. Efficiency and cost plots are certainly linear, in log scale, after four threads per CUDA block have been assigned.

The curves for all test signals are almost identical in Figure 5(b) and Figure 5(d), hence the scalability of parallel convolution in our proposed GPU strategy scales better than that of the multi-core strategy as the efficiency and cost can be kept constant as the problem size increases. This algorithm can also reach cost-optimality by assigning 32 threads per CUDA block, as in this point we get a fair trade-off between speedup, efficiency and cost.

B. DWT Performance and Scalability

To test performance and scalability of the cascade implementation of the DWT, we computed the db6 DWT of each test signal using a five-level decomposition, thus the parallel convolution algorithm was executed five times, one

ISSN 2395-8618



(a) Execution time (sec.)

(b) Efficiency

(c) Speedup

(d) Cost

Fig. 5. Scalability and performance analysis of parallel convolution for the GPU strategy. The number of threads t per CUDA block assigned to compute the convolution sequence is shown in the x-axis. The legend in (a) shows the color and marker used to plot the results for each test signal

per decomposition level. The number of assigned threads were kept constant among levels. We recorded the DWT execution time for each test signal and computed speedup, efficiency and cost.

This analysis was performed for both strategies. The performance and scalability results for the multi-core strategy are shown in Figure 6 and those for the GPU strategy are displayed in Figure 7.

1) Multi-core Strategy: Execution times for a five-level DWT decomposition using the multi-core strategy are displayed in Figure 6(a). Each curve in the referred subfigure shows how execution times change with the number of assigned threads t for a given test signal. It should be noted that one can clearly tell each curve apart before $t = 32$, after that all curves cluster together in a single linear trend.

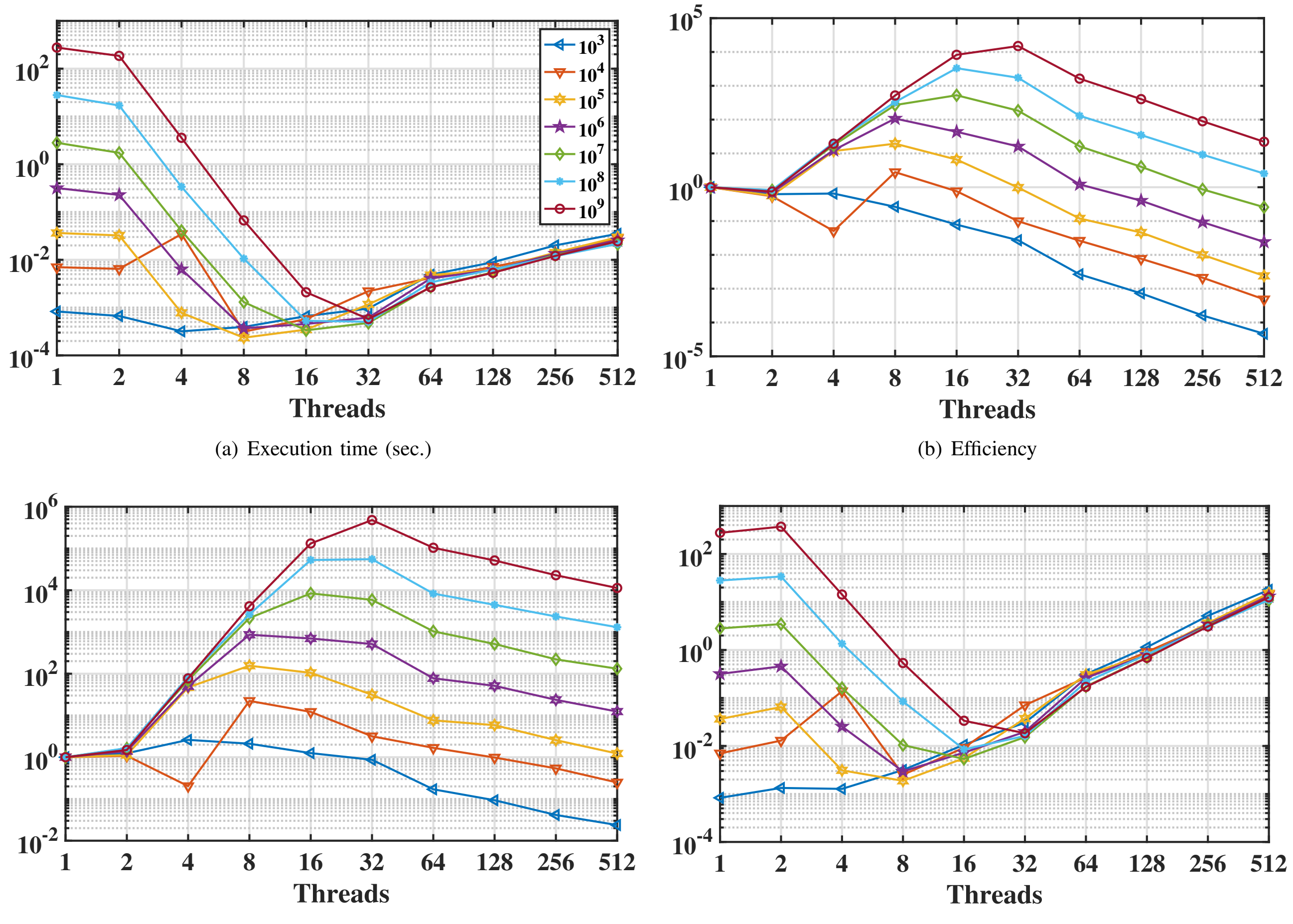
A similar behavior is observed for cost curves in Figure 6(d), as cost is defined as the product of execution time and the number of assigned threads. Thus, the DWT execution time is almost the same for all test signals after $t = 32$, no matter the input size.

Regarding speedup, the multi-core DWT implementation inherits the parallel convolution properties, showing a sublinear behavior for all test signals, as can be observed in Figure 6(c). Efficiency curves for all test signals show an inflection point, located at different t values, after such point all efficiency curves are almost parallel and decrease linearly.

Based on our previous analysis, an appropriate DWT operation point in the multi-core strategy is $t = 32$, as it offers a good trade off between execution time, efficiency, speedup and cost. Additionally, we can say that such algorithm boast excellent scalability as efficiency can be kept constant by simultaneously increasing the number of assigned threads and the test signal length.

2) Graphic Processing Unit Strategy: The GPU DWT implementation looses the desired characteristics of its multi-core counterpart, namely excellent scalability and inflection points in efficiency and speedup curves. It also inherits the properties of its parallel convolution algorithm, that is to say similar execution times, efficiencies and costs for all test signal lengths as the number of threads per CUDA blocks changes, as can be observed in Figures 7(a), 7(b), and 7(d), respectively.

ISSN 2395-8618



(a) Execution time (sec.) (b) Efficiency (c) Speedup (d) Cost

Fig. 6. Scalability and performance analysis of the DWT for the multi-core strategy. The number of threads t assigned to the parallel convolution is shown in the x-axis. The legend in (a) shows the color and marker used to plot the results for each test signal

Execution times are also significantly lower that in the DWT multi-core implementation. The fact that we launch as many threads as elements in the convolution sequence leads to flat execution-time and speedup curves after $t = 8$, as it will always launch p threads in total no matter how many threads are assigned per CUDA block.

Based on the previous analysis and on the results shown in Figure 7, we can say that a suitable DWT operation point in the GPU strategy is $t = 8$, as in such point we get the highest speedup and similar execution times for most test signals.

IV. Conclusion

We presented a data-level parallelization strategy to accelerate computation of the DWT. Said strategy was implemented and compared in two multi-threaded architectures, both with shared memory. The first considered architecture was a multi-core server and the second one was a graphic processing unit (GPU). All programs were implemented using C language, the OpenMP API for the multi-core server, and the CUDA API for the GPU. The DWT was implemented by means of the cascade structure as shown in Figure 1, which consists in iteratively applying a pair of orthonormal filters to the approximation coefficients (i.e. the low-pass filter response). As the main operation to compute the approximation and detail coefficients at each decomposition level is convolution, the proposed data-level parallelization strategy focused on distributing computation of the convolution sequence elements among as many threads as possible, leading to the design of a parallel convolution algorithm.

A significant difference in the parallel convolution algorithm designed for each architecture was the convolution sum structure, which led to two different padding methods. The multi-core strategy used the classical convolution sum as described in Eq. 1, where causal filters are assumed. Conversely, the GPU strategy used the symmetric convolution as described in Eq 2, where time-centered filters are assumed. The resulting convolution sequences are equivalent up to a unit delay.

ISSN 2395-8618

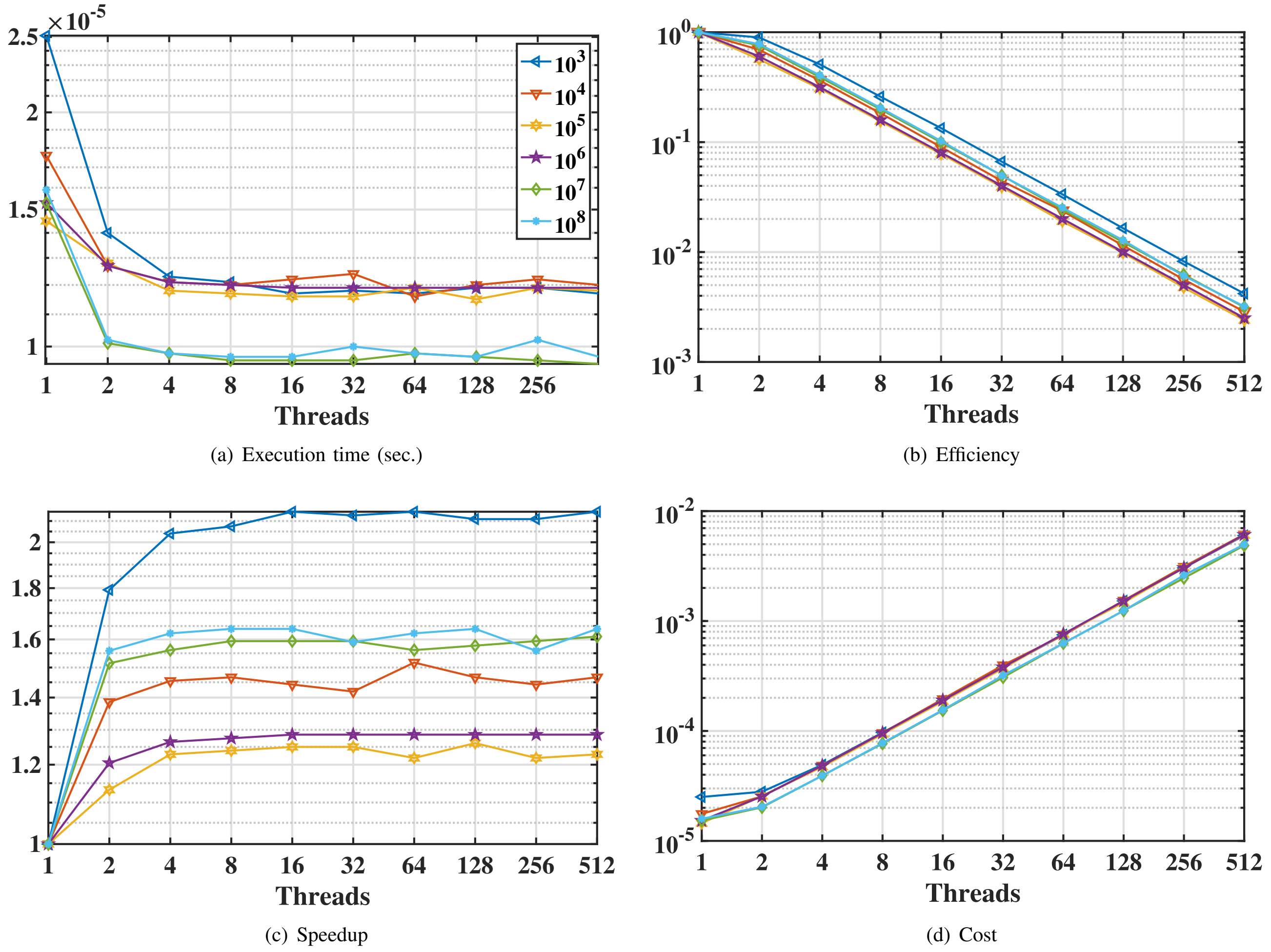


Fig. 7. Scalability and performance analysis of the DWT in the GPU strategy. The number of threads t per CUDA block assigned to the parallel convolution is shown in the x-axis. The legend in (a) shows the color and marker used to plot the results for each test signal

We analyzed the parallel convolution algorithm using execution times, speedup, efficiency and cost as performance metrics in each multi-threaded architecture. Results showed the multi-core strategy scales fairly well to the input size, however execution times remain high when compared to those of the GPU strategy. Both parallel implementations displayed sublinear behavior in their speedup curves as the number of assigned threads increased, but in the GPU strategy there was no significance gain after $t = 8$ because workload is already evenly distributed among threads.

Based on the performance analysis results, we were able to identify an optimal number of threads assigned to the parallel convolution algorithm for both architectures. Such number nearly matches the available cores in the multi-core strategy, while in the GPU strategy it is convenient to select the smallest t in the flat zone for the speedup curves.

To sum up, although the multi-core strategy boast excellent scalability, the GPU strategy is preferred as being faster. It is recommended to use the multi-core strategy for signals greater than 10^8 elements in length, as the GPU architecture can't allocate enough RAM memory in the selected device. Future work consist in implementing better convolution schemes that reduces RAM requirements, such as overlap-and-add or overlap-and-save, in frequency space. Additionally, we could design truly parallel DWT algorithms taking advantage of hardware pipeline designs that improve throughput such as the lifting scheme.

REFERENCES

[1] C. Guger, B. Z. Allison, and K. Miller, Eds., *Brain-Computer Interface Research: A State-of-the-Art Summary 8*, ser. SpringerBriefs in Electrical and Computer Engineering. Springer, Cham, 2020.

[2] J. Wolpaw and E. W. Wolpaw, Eds., *Brain-Computer Interfaces: Principles and Practice*. Oxfor University Press, 2012.

[3] H. Peng, B. Hu, Q. Shi, M. Ratcliffe, Q. Zhao, Y. Qi, and G. Gao, "Removal of ocular artifacts in EEG – an improved approach combining DWT and ANC for ubiquitous applications," *IEEE Journal of Biomedical and Health Informatics*, vol. 17, no. 3, pp. 600–607, 2013.

[4] B. Hu, H. Peng, Q. Zhao, B. Hu, D. Majoe, F. Zheng, and P. Moore, "Signal quality assessment model for wearable EEG sensor on prediction of mental stress," *IEEE Transactions on Nanobioscience*, vol. 14, no. 5, pp. 553–561, 2015.

ISSN 2395-8618

[5] H. Peng, B. Hu, F. Zheng, D. Fan, W. Zhao, X. Chen, Y. Yang, and Q. Cai, "A method of identifying chronic stress by EEG," *Personal and Ubiquitous Computing*, vol. 17, no. 7, pp. 1341–1347, 2013.

[6] A. Sharmila and P. Geethanjali, "DWT based detection of epileptic seizure from EEG signals using naive Bayes and k-nn classifiers," *IEEE Access*, vol. 4, pp. 7716–7727, 2016.

[7] A. Chamanzar, M. Shabany, A. Malekmohammadi, and S. Mahammadinejad, "Efficient hardware implementation of real-time low-power movement intention detector system using FFT and adaptive Wavelet transform," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 3, pp. 585–596, 2017.

[8] J. A. Healey and P. W. Picard, "Detecting stress during real-world driving task using physiological sensors," *IEEE Transactions on Intelligent Transportation Systems*, vol. 6, no. 2, pp. 156–166, 2005.

[9] E. Strickland, "Mind games," *IEEE Spectrum*, vol. 55, no. 1, pp. 40–41, 2018.

[10] S. Qiu, Z. Li, W. He, L. Zhang, C. Yang, and C.-Y. Su, "Brain-machine interface and visual compressive sensing-based teleoperation control of an exoskeleton robot," *IEEE Transactions on Fuzzy Systems*, vol. 26, no. 1, pp. 58–59, 2017.

[11] M. Rabbani and R. Josh, "An overview of the JPEG 2000 still image compression standard," *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 3–48, 2002.

[12] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, pp. 245–267, 1998.

[13] B. K. Mohanty, A. Mahajan, and P. K. Meher, "Area- and power-efficient architecture for high-throughput implementationof lifting 2-D DWT," *IEEE Trans. Circuits Syst. II-Express Briefs*, vol. 59, no. 7, pp. 434–438, 2012.

[14] M. M. Hasan and K. A. Wahid, "Low-cost architecture of modified Daubechies lifting Wavelets using integer polynomial mapping," *IEEE Trans. Circuits Syst. II-Express Briefs*, vol. 64, no. 5, pp. 585–589, 2017.

[15] J. Proakis and D. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*. Pearson-Prentice Hall, 2007.

[16] J. Hennessy and D. Patterson, *Computer architecture: A quantitative approach*, 6th ed. Morgan Kaufmann, 2019.