BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 70(1), 2022, Article number: e139393
DOI: 10.24425/bpasts.2021.139393

Execution time predicition model for parallel GPU realization of dicscrete transforms computation algorithms

Dariusz PUCHALA, Kamil STOKFISZEWSKI*, and Kamil WIELOCH

Institute of Information Technology, Łódź University of Technology, ul. Wólczańska 215, 90-924 Łódź, Poland

Abstract. Parallel realizations of discrete transforms (DTs) computation algorithms (DTCAs) performed on graphics processing units (GPUs) play a significant role in many modern data processing methods utilized in numerous areas of human activity. In this paper the authors propose a novel execution time prediction model, which allows for accurate and rapid estimation of execution times of various kinds of structurally different DTCAs performed on GPUs of distinct architectures, without the necessity of conducting the actual experiments on physical hardware. The model can serve as a guide for the system analyst in making the optimal choice of the GPU hardware solution for a given computational task involving particular DT calculation, or can help in choosing the best appropriate parallel implementation of the selected DT, given the limitations imposed by available hardware. Restricting the model to exhaustively adhere only to the key common features of DTCAs enables the authors to significantly simplify its structure, leading consequently to its design as a hybrid, analytically–simulational method, exploiting jointly the main advantages of both of the mentioned techniques, namely: time-effectiveness and high prediction accuracy, while, at the same time, causing mutual elimination of the major weaknesses of both of the specified approaches within the proposed solution. The model is validated experimentally on two structurally different parallel methods of discrete wavelet transform (DWT) computation, i.e. the direct convolution-based and lattice structure-based schemes, by comparing its prediction results with the actual measurements taken for 6 different graphics cards, representing a fairly broad spectrum of GPUs compute architectures. Experimental results reveal the overall average execution time and prediction accuracy of the model to be at a level of 97.2%, with global maximum prediction error of 14.5%, recorded throughout all the conducted experiments, maintaining at the same time high average evaluation speed of 3.5 ms for single simulation duration. The results facilitate inferring the model generality and possibility of extrapolation to other DTCAs and different GPU architectures, which along with the proposed model straightforwardness, time-effectiveness and ease of practical application, makes it, in the authors' opinion, a very interesting alternative to the related existing solutions.

Key words: graphics processing unit (GPU); execution time prediction model; discrete wavelet transform (DWT); lattice structure; convolution-based approach; orthogonal transform; orthogonal filter banks; time effectiveness; prediction accuracy.

1. INTRODUCTION

Despite continuous increase in computational effectiveness of electronic devices, the development of time-efficient algorithms seems to be invariably an up-to-date issue. This is also still the case for a very important class of computational methods, namely, discrete transforms (DTs) computation algorithms (DTCAs), widely used in a variety of applications, such as general signal and image processing [1–3], data and image compression [4–6], signal analysis and pattern recognition [7–9], and in many other common digital signal processing tasks. This, and the ever-growing amount of digitally processed data, makes the research on improvement of those algorithms very intense. In recent years the domain of high-performance computing (HPC) has changed significantly due to the emergence of the entirely new approach to the architecture design of graphics processing units (GPUs), which allows developers to perform general-purpose, parallel computations very efficiently without excessive effort [10, 11]. Thus, GPUs have attracted attention of scientific community across a broad spectrum of computational research areas and successfully have become a basic, powerful tool for industrial and research computations, which combines cost-effective hardware solutions with relatively high computational performance [12]. The main challenge of the design of GPU algorithms is to adapt classic sequential solutions, well-suited for CPU-based systems, to meet the requirements of the parallel GPUs architectures. It turned out quickly that many computational problems, e.g. [13–17], can be significantly accelerated with the use of GPUs. However, the parallel algorithms construction process remains still a great challenge, since the diversity of GPUs architectures details makes it very difficult to predict the actual effectiveness of the given solution, designed for a chosen family of hardware platforms, and the improvement results are often far from assumed, when verified on physical hardware. To ease this challenge for the important case of DTCAs, the authors propose a novel execution time prediction model, which allows for accurate and rapid estimation of execution times

*e-mail: kamil.stokfiszewski@p.lodz.pl

Manuscript submitted 2021-02-15, revised 2021-08-26, initially accepted for publication 2021-09-27, published in February 2022.

of various kinds of structurally different DTCAs performed on GPUs of distinct architectures, without the necessity of conducting the actual experiments on real hardware.

Many reputed models are present in the literature, see e.g. [18–46], and may be assigned to 5 general categories, namely: analytic, statistical, machine learning-based, simulation-based and hybrid. All of the mentioned approaches exhibit different effectiveness characteristics. Analytic approaches, see e.g. [21–26], are usually rapid and allow fast estimation of execution times of the selected, GPU dedicated, implementations. They often, however, exhibit unsatisfactory accuracy performance and are difficult to apply in practice, since they require preliminary, intense analytical effort to evaluate numerous model parameters, based on the static analysis of the algorithm code and the immense knowledge of internal hardware mechanisms, required from the designer. The next two classes of models are statistical, e.g. [26–29], and machine learning-based ones, see [26, 30–32], which are similar in their general construction principles. Statistical models often utilize such techniques as *principal component analysis* (PCA) and/or *linear regression* (LR) methods to predict the performance of GPU implementations of the certain pool of computational problems. Machine learning-based models employ *neural networks* (NNs), *support vector machines* (SVMs), *random forests* (RFs) and/or *decision trees* (DTs), in the process of execution time prediction of GPU applications. In comparison to analytic methods, statistical and machine learning-based approaches relax the need for preliminary complex analysis of software and hardware architecture interactions and internal hardware mechanisms, maintaining, at the same time, rapid characteristics of the ultimate estimation process. Although potentially very promising results can be achieved with the use of such models, they suffer from a few significant limitations, i.e. the need for aggregation of large amounts of the training data, collected for different use-case scenarios and hardware architectures, what might be in many cases unrealizable practically, and also the considerable time inefficiency of the training process, required for establishing the model parameters. On the other side of the spectrum simulation-based models are present, see e.g. [33–37]. Their main advantages are high prediction accuracy and ease of practical application. Here, the designer is required only to supply the model with a properly prepared solution code, with almost no necessity of its prior analysis. The model then emulates the execution process and responds with the estimated execution time. However, this comes at a price of immense time required for the simulation process to generate the estimation results. This often prohibits practical application of such models when large amounts of tests or extensive data sizes are involved in sufficient validation of the designed algorithmic solution. The last class of the GPU performance estimation solutions are the hybrid models (see [38–41]), which combine the two previously described prediction methods – analytic and simulation-based, in an attempt to eliminate the earlier mentioned drawbacks of both of the discussed approaches, namely, the limited prediction accuracy and the immense simulation time. Our model adheres to this last category however, its restriction to the considered class of DTs computation algorithms, enables us to greatly simplify the model structure in comparison to the existing hybrid methods. Both, the restriction of the model to the class of DCTAs and, as a consequence, its simplification within the simulation part, cause the proposed model not only to significantly increase the time performance and reduce preliminary code analysis effort, but also considerably enhance the prediction accuracy characteristics with respect to the existing, more general approaches. The specified enhancements are possible, due to the key common structural features of DTCAs, whose exploitation enable the mentioned simplifications to be administered. These are: (i) the lack of conditional statements in most of the known DTCAs, (ii) sufficiently steady memory access patterns, which can be described by a minimum number of constant scaling factors contained within a model, and (iii) the presence of multiple, sequential and globally synchronized, computational stages in many of the practical DTCAs implementations. Our model depends on those features, which obviously narrows its applicability on the one hand, but, on the other, immensely increases its general performance with respect to the existing solutions in the DTCAs domain. At last, it is also worth noting that the proposed model can be applied to any kinds of GPU implementations of parallel algorithms, as long as they adhere to the mentioned assumptions.

In order to evaluate the performance of our model, we have chosen two exemplary, structurally different parallel methods of discrete wavelet transform (DWT) computation to be tested, namely – the direct, convolution-based and the lattice structure-based DWT computation algorithms. Our choice was motivated by the facts that: (i) both of the mentioned methods are, in the structural sense, good representatives of other well-known DTCAs commonly used in practice, and (ii) their classic computational as well as parallel step complexities are of the same order, which is not often the case for many other, mutually structurally different, DTs computation methods, e.g. such as the DFT and the FFT algorithms, i.e. the direct, matrix-based and fast, in-place, butterfly structure-based discrete Fourier transform computation procedures, see e.g. [1]. The last feature of the selected exemplary DWT computation algorithms makes the conducted comparisons much more subtle, which in turn, puts significantly higher demands on the model accuracy characteristics than in the case of many other DTCAs, such as the ones mentioned above. We have validated our model by performing numerous execution time prediction tests of the chosen DWT computation methods for a range of different DWT data sizes and filter lengths, and comparing the resulting estimations with the actual measurements of the considered DTCAs execution times taken for 6 different graphics cards, representing a fairly broad spectrum of NVIDIA GPUs hardware microarchitectures with compute capability (CC) indices ranging from 2.0 (Fermi microarchitecture) to 7.5 (Turing microarchitecture). The choice of NVIDIA GPUs for our model validation, apart from their undoubtedly high popularity and accessibility, was motivated by the fact that they may be considered to be a good example of typical, modern, general-purpose,the architectures of mass-parallel compute devices. The results enable us to infer model generality and possibility of extrapolation to other

DTCAs performed on GPU devices with distinct, diverse architectures.

To complete our introductory discussion, we will now provide a brief outline of the rest of the paper. In Section 2 we will make a review of the existing, representative execution time prediction models, which might be helpful in making the objective comparison of our model general performance with the efficiencies of the related solutions. In Section 3 we will present the chosen DWT computation methods and provide an insight into their algorithmic structure along with their theoretical computational and parallel step complexities, which eases the interpretation of the obtained experimental results. In Section 4 we will briefly discuss CUDA (NVIDIA *Compute Unified Device Architecture*) program execution model on which our solution is solely based on. Section 5, constituting the main part of the paper, will be dedicated fully to the detailed presentation of the proposed model and the most significant aspects of its practical application. In Section 6 experimental results, along with the methodology of the conducted tests, will be presented and discussed in detail. At last, Section 7 contains the most important conclusions regarding the proposed model general efficiency and the prospects for the future work.

2. GPU PERFORMANCE PREDICTION MODELS

In this section we will present a survey on the most widely-known, existing GPU performance prediction models, according to their types, described in the introduction, along with their general operational and accuracy characteristics, helpful in making the objective comparison of our model general performance with the efficiencies of the related solutions.

2.1. Analytical models

Let us start with the existing analytical models, appreciated for their generality, relatively good prediction accuracy and high evaluation time effectiveness. The primary approaches here are the BSP (*Bulk Synchronous Parallel*) and PRAM (*Parallel Random Access Machine*) models and their diverse variants, see, e.g. [18].

The first reputed model, which adheres to the mentioned category is presented in paper [21]. It is an analytical model whose foundations rely on the notions of *Memory Warp Parallelism* and *Computation Warp Parallelism*. Based on those metrics, derived from GPU code static as well as dynamic analysis, the model predicts the selected GPU solution execution time with significant accuracy and in a highly rapid fashion. It is average prediction error is around 13.3% for general computational tasks (with maximum prediction mismatch reaching 45%). However, the related tasks of vector and matrix calculations (to which DTCA might be assigned), its average prediction error drops down to 7% with the maximum prediction mismatch of 30%.

Model proposed in paper [22] is a fast analytical model that estimates the execution time of massively parallel applications using the instruction-level and thread-level parallelism. It is based on the analysis of CUDA PTX (*parallel thread execution instruction set*) files for the execution process reconstruction on the instruction and memory consumption levels. Assembly codes and specific instruction time costs are used for final execution time prediction. The key idea for the mentioned model is to find the maximum number warps that can execute in parallel and estimate CPI (*cycles per instruction*) factor, which determines the actual level of computations parallelism. The model average prediction error is around 10% for general computational tasks (with maximum prediction mismatch of about 30%). Once again, for vector and matrix related calculations its average prediction error drops down to only about 5% with the corresponding maximum prediction mismatch of around 15%.

In paper [23] is a simple and intuitive BSP-based model is proposed. It relies on the number of arithmetic and memory access operations preformed by the GPU, with additional information on cache, shared and global memory usage, obtained from the profiling data. It was tested on the matrix related computational tasks, where it is average prediction error was close to 5% with the maximum mismatch of about 40%.

The next model, worth noticing, is the one presented in paper [24]. It is an extension of existing analytical solutions. The developed model is able to capture parallelism, latency-hiding and occupancy together in one framework. The model is aimed at identification of performance bottlenecks, reduction of the configuration space for kernel execution parameters and is able to predict achievable execution times and capture their trends for various kernel launch configurations. In case of matrix related computations, its average prediction error oscillates within 15% with the respective maximum prediction mismatch of about 60%.

Last but not least, there is the model presented in [25]. It is a performance prediction model for the CUDA GPGPU platform based on PRAM, BSP and QRQW (*Queue-Read Queue-Write*) models. The proposed model is used to analyze pseudo-code of CUDA kernels to obtain a performance estimate with three experimental case studies: matrix multiplication, list ranking, and histogram generation. It is average prediction error is approximately 21% for general computational tasks with maximum prediction inaccuracy reaching about 100%.

2.2. Statistical and machine learning-based models

The next, important class of models is constituted by the statistical and machine learning-based (ML-based) predictors. They are constructed around the common principle of the automatic adjustment of the assumed model parameters, conducted on the basis of the statistical data, collected during extensive empirical measurements, performed for a wide variety of computation task classes with the use of a multiplicity of distinct GPU hardware platforms.

Very interesting and comprehensive research on comparison of the statistical, machine learning and analytical approaches to GPU performance modeling, is presented in the work [26]. There, the machine learning techniques, namely support vector machines (SVMs) and random forests (RFs), are compared with the statistical methods, based on linear regression (LR), and finally confronted with BSP-based analytical model. Studies show relative superiority of the analytical model, which in case of the matrix-related computations achieves the smallest average prediction error of around 12%, with maximum prediction mismatch of 25%. For the statistical LR model, the respective errors are about 15% and 60%, while for machine learning SVM and RF approaches, they are, at their best, equal to 21% and 158%, respectively. On the other hand, the authors point out that machine learning and statistical models are more generalizable to different applications and GPU architectures, compared to the baseline BSP analytical model, which requires constant refinement.

Let us proceed to other, worth noticing, solutions. And so, the solution presented in paper [27] is a statistical performance model, designed for the OpenCL programming standard applications dedicated for NVIDIA GPUs. It is based on the use of the principal component analysis (PCA) methods and is reported to predict application execution times with about 9% error on average, with the respective maximum prediction error value not exceeding 40%.

The *Eiger* framework [28], is based on linear regression techniques and provides an automated model construction method in which designers may profile and characterize GPU workloads, automatically construct performance models, and evaluate their sensitivity to different hardware configurations. Because of its extensive area of applicability, its overall prediction error is relatively high, reaching the level of about 35% for general computational tasks, with the maximum prediction mismatch of about 110% for DTCA-like kernels.

Solution proposed in paper [29] is the statistical model for performance and power consumption analysis for ATI GPUs. It is build with the use of statistical random forest (RF) approach. Its general prediction error is reported to be at a level of 13.1%.

Finally, let us consider models using inherently machine learning-type adaptation techniques, such as the neural networks and/or non-linear regression with feature extraction methods. As such, solution proposed in paper [30] describes a GPU performance and power consumption estimation model that uses a dedicated multilayer, nonlinear perceptron neural network, trained on the data sets gathered from measurements performed for numerous GPU devices and an extensive collection of GPU applications. The obtained scaling curve of kernel execution characteristics is used to estimate the performance and power consumption of the newly presented test applications ran under different GPU configuration settings. Solution achieves the overall average prediction error of around 15%.

The next model, presented in paper [31], is also multilayer neural network-based approach to GPU applications performance prediction. It is reported that the average network prediction error lays within 5%, across a large, multidimensional parameter space, considered in the undertaken experiments.

The last of the well-known machine learning-based models we want to take notice of is the one proposed in [32]. It focuses on exploiting the correlations between program properties, hardware characteristics and GPU execution time, using non-linear regression methods combined with classification and feature extraction algorithms. For a wide variety of mixed types GPU applications, the model achieves the overall average prediction error of approximately 18% with the maximum prediction inaccuracy of about 65%.

2.3. Simulation-based models

Let us now move forward to the next, very important, class of execution time prediction models, namely, the simulation-based models. Among other considered solutions, they are definitely the ones with the smallest prediction error and many of them are even aimed to achieve cycle-level accuracy. Unfortunately, the price they pay for reaching the maximum prediction accuracy levels is quite high, and manifests itself in the immense amount of time needed for a given GPU application execution process to be fully simulated and thus, the execution time prediction to be evaluated, see e.g. [36]. On the other hand, they are the most easily applicable ones from the user's perspective, since all they often require from the user is the delivery of a properly prepared solution code, with almost no necessity of its prior analysis.

The most reputed model in this category is undoubtedly that developed under the *GPGPU-Sim* project [33], see also [34], widely cited, referred to, and often utilized by other solutions as their constituent part. It is a cycle-accurate, modular and extendible GPU device emulator, capable of precise evaluation of execution times for nearly all types of diverse computational workloads. For DTCA-like problems, its average prediction error lowers to about 1%, with a maximum inaccuracy level of 6%, which has to be considered an excellent result. At the same time, GPGPU-Sim emulation process is on average, approximately 9 orders of magnitude, i.e. 10^9 times, slower, compared to real hardware, assuming the usage of consumer segment CPU-based system for the simulation purposes and GPU of the same class for the application execution, see [36].

Next model in this category is the *GPUSimPow* simulator [35] – a detailed, architecture-level power consumption simulation framework for the compute parts of contemporary GPUs running CUDA or OpenCL workloads, which is as an extension of GPGPU-Sim, and, as a sort of a side-effect, enables also precise execution times evaluation. Similarly to the previous model, for DTCA-like computations, its average prediction error is about 1%, with maximum prediction inaccuracy on the level of approximately 5% and a simulation slowdown of a factor of about 10^9 in comparison to real device computations.

GPGPU-MiniBench simulation framework [36] focuses on automatic generation and execution of miniature, yet representative GPGPU workloads, exhibiting similar execution characteristics as the ones whose performance is about to be evaluated, thereby dramatically accelerating architectural simulation, without loosing the ability to conduct accurate predictions. Its overall average prediction error is reported to be at a level of 4.7% across a broad set of GPU benchmarks, and the achieved average speedup factor is approximately equal to $49\times$ with respect to GPU architectural simulation, which amounts to an approximate slowdown of a factor of about $2 \cdot 10^7$ in comparison to real device execution times.

Finally, the last well-known solution is the *Multi2Sim* simulator [37], an open-source, modular, and fully configurable toolset that enables ISA-level simulation of x86 CPUs and and AMD GPUs. The authors focused on AMD graphics cards, for which the simulator replaces the OpenCL library and automatically emulates the GPU computations. Its average execution time prediction error for DTCAs-like workloads oscillates within a level of 12% with respective maximum prediction error of about 20% and the average simulation slowdown of a factor of about $8 \cdot 10^7$ in relation to real device computations.

2.4. Hybrid models

The last important class of the GPU execution time prediction models are the hybrid ones. They usually combine an analytic approach with a restricted simulation-based execution time prediction techniques, in order to significantly speed up the model prediction evaluation process with respect to the simulation-based counterparts, maintaining at the same time, high prediction accuracy, characteristic to the models belonging the last of the mentioned categories. They are also important from our perspective, since model proposed in this work can be considered to adhere to this particular class.

Focusing on the most reputed and widely cited hybrid models, we will start by introducing the *GpuTejas* framework [38], which is a hybrid, Java-based parallel GPU execution time prediction model that introduces a novel scheduling and partitioning scheme for parallelizing GPU simulations. The main goal of the GpuTejas authors was to create an accurate functional simulator, much faster, than the popular GPGPU-Sim. They have achieved an average acceleration of about 430× in relation to GPGPU-Sim, due to block level calculations parallelization, which amounts to a slowdown of a factor of about $2.3 \cdot 10^6$ in comparison to real device computations. The reported overall average inaccuracy of the GpuTejas model is about 15% with the maximum prediction error of nearly 30%.

The next well-established hybrid model is the *PPT-GPU* framework, presented in [39], being, as the authors themselves explain, a scalable and accurate simulation framework that enables GPU code developers and architects to predict the performance of applications in a fast and accurate manner on different GPU architectures. It is fully parametrized performance prediction toolset that relies on (PTX) ISA and GPU configurations to predict applications runtime without having to execute them on real hardware. Solution accuracy is validated through the execution times comparison of the set of benchmarks evaluated by the model against those measured on real devices, as well as reported by the GPGPU-Sim emulator. The results show that PPT-GPU overall prediction inaccuracy lies within 10% compared to the real device measurements and, at the same time, the simulation is about 160× faster, on average, than that of GPGPU-Sim, which amounts to a simulation slowdown of a factor of about $6.25 \cdot 10^6$ in relation to real device computations. Model presented in the paper [40] is a hybrid framework for fast and accurate GPU performance estimation. Kernel execution flow is statically analyzed, producing the functional execution trace, which is then dynamically simulated to obtain the performance prediction. Authors report the model overall average prediction inaccuracy to be at the level of 17.04% and 150× average speedup in relation to the respective GPGPU-Sim emulation time, which gives the simulation slowdown of a factor of about $6.7 \cdot 10^6$ compared to real GPU average performance. The last model, which we want to take notice of is the *GATSim* abstract timing simulation model, proposed in [41]. It is based on a hybrid approach of separation of functional and timing models, combining a fast functional kernel execution on the existing simulators or native GPU hardware with light and accurate abstract timing model. Its average overall prediction error is reported to be at a level of 4% with maximum prediction mismatch of about 16%. At the same time, the results show that the model prediction evaluation is about 400× faster, on average, compared to a cycle-accurate GPU simulators for standard GPU benchmarks, which gives the average simulation slowdown of a factor of about $2.5 \cdot 10^6$ in relation to the physical GPU devices computation time.

2.5. Models characteristics summary

In this section we have reviewed the most representative GPU performance prediction models, according to their types, along with their general operational and accuracy characteristics. There of course exist other well-established GPU execution modeling solutions, enabling GPUs performance prediction, especially functional and cycle-accurate simulators (like *Accel-Sim* [42], *Barra* [43] or the *Ocelot* [44] framework) or heterogeneous CPU-GPU simulation frameworks (e.g. *gem5-gpu* [45] or *FusionSim* [46]), but either they are directly based on the solutions described in the previous paragraphs or their prediction results are very similar to those reported above. This makes the presented review fairly comprehensive in terms of the general characteristics of the results obtained within each of the main categories of the existing GPU performance prediction models. For the sake of later comparisons, let us summarize the best characteristics obtained by the models in each of the considered categories, in a brief form, whose respective summary is given in Table 1.

Table 1
Summary of the best prediction efficiencies characteristics within each of the considered models categories

Model type	Average prediction error	Maximum prediction mismatch	Average evaluation slowdown	Algorithm type	Reference materials
Analytical	5%	15%	negligible	DTCA-like	[22]
Statistical	9%	40%	negligible	general	[27]
ML-based	6%	45%	negligible	general	[31]
Simulation-based	1%	5%	10^8	DTCA-like	[35]
Hybrid	4%	16%	10^6	general	[41]

The average prediction errors along with the respective maximum prediction divergences are given in columns 2nd and 3rd of Table 1 for each of the considered models category. It should be kept in mind, however, that some of them refer to prediction characteristics of the presented models for general algorithm classes (c.f. attribute “general” in column 5), covering a wide range of structurally and functionally different GPU applications. This may, in principle, lower their reported prediction efficiency in comparison to models, in which only a certain class of the GPU computation algorithms (i.e. DTCA-like computations) are taken into account, which, in turn, are of main interest of ours. The fact that more detailed information could not be effectively extracted from the source publications and other additional materials we have studied, forced us to restrict our insight into more general estimations in cases of some of the selected prediction methods presented in the current work, nevertheless, we have decided to include them in our comparisons, since we hope that they eventually might still have a considerable informative value despite the mentioned inconveniences. At last, it is worth explaining that in column 4th of Table 1 the models average prediction evaluation slowdown is reported, which relates to the real GPU device execution time needed for the respective application to finish its computations. Here, the attribute “negligible” indicates the respective model prediction output response time being of order of milli or even microseconds and, mostly, independent of the physical execution time of the measured GPU application.

3. DISCRETE WAVELET TRANSFORM AND ITS COMPUTATION ALGORITHMS

The discrete wavelet transform (DWT) is a mathematical tool that decomposes a signal into representation that gradually exposes signal details and trends as a function of time (or space, in case of 2D DWT). DWT has become popular in many applications regarding general digital signal and image processing tasks, e.g. [8, 47–49], due to such its advantages as the use of localized basis functions and the existence of the variety of effective computation algorithms, see e.g. [50–58]. In our work we will consider two such algorithms, namely, the matrix-based (or equivalently direct, convolution-based) and the lattice structure-based approaches to DWT computation. These approaches are, in the structural sense, good representatives of other well-known DTCAs commonly used in practice, what was discussed more specifically in Section 1. Let us now present the considered DWT computation methods and review their characteristics, important from this work perspective.

As in the case of other discrete linear transforms, DWT can be calculated directly by multiplying the transform matrix by the input signal vector, which can be shortly written as

$$\mathbf{y} = \mathbf{A}\mathbf{x}, \tag{1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are N – element input and transformed vectors, respectively and $\mathbf{A}$ is $N \times N$ – element transform matrix. Matrix representation of the discrete wavelet transform can be derived by interpreting the DWT as the analysis stage of a finite impulse response two-channel orthogonal filter bank, see [54, 55]. In such view, both of the K – element h and g analysis filters form a transformation matrix of the form:

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

For orthogonal DWTs, the inverse transform matrix is the transpose of its forward counterpart, i.e. $\mathbf{A}^{-1} = \mathbf{A}^T$, so it is easy to verify that the computational structure of the inverse DWT is the same as the forward one, which enables us to state that the considerations presented in this paper apply to both cases, i.e. the forward and the inverse DWT computational tasks.

To reduce computational complexity, instead of using direct matrix approach, DWT can also be calculated using more efficient algorithms. Here, definitely an interesting option is the use of lattice structure-based approach, which is a powerful tool of implementing finite response two-channel filter banks. It can be shown, see e.g. [48], that the DWT can be calculated with the use of the considered filter banks, which consequently means that it also can be evaluated with the lattice structure, presented in Fig. 1. The calculations for the first $K/2$ stages of lattice structure-based method of DWT computation are described by base operations $\Gamma_{i,j}$, defined as

$$\Gamma_{i,j} = \begin{bmatrix} 1 & s_{i,j} \\ t_{i,j} & 1 \end{bmatrix}, \tag{3}$$

and depicted in Fig. 1 with the ‘•’ symbol, where $s_{i,j}$, $t_{i,j}$, $i = 0,\dots,K/2-1$ and $j = 0,\dots,N/2-1$ are parameters whose values are determined during transform factorization and τ_n represents single multiplication. The result is a representation of the input signal $\mathbf{x}$ in low and high frequency bands.

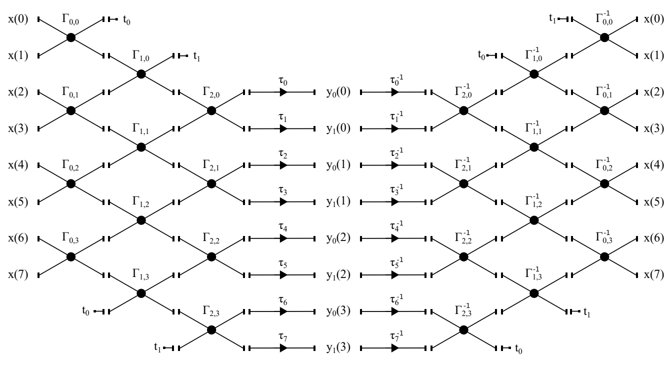

Fig. 1. Lattice structures of forward and inverse DWTs for transform size $N = 8$ and filter length $K = 6$

3.1. Theoretical time effectiveness of the considered DWT computation approaches

Let us now consider theoretical time effectiveness of the two of the analyzed DWT computation methods in case of their classic, sequential realizations as well as their parallel implementations. For sequential realizations we can consider traditional computational complexities of the examined approaches to be good theoretical time effectiveness measures. In all of the subsequent analysis we will consider computational complexities to take into account both the number of addition and multiplication operations together, carried out during the course of the considered DWT computation method. It can be shown that for DWT matrix-based approach its computational complexity equals to

$$\mathbf{C}_M^{1D}(N,K) = N(2K-1), \tag{4}$$

where, as earlier, N denotes the transform size and K is the filter length. For the lattice structure-based approach, since a single operation $\Gamma_{i,j}$ consists of 2 multiplications and 2 additions, it can be concluded that its computational complexity is

$$\mathbf{C}_L^{1D}(N,K) = N(K+1). \tag{5}$$

For specifying theoretical time effectiveness of parallel realizations of the considered DWT computation methods we will take advantage of the notion of *step complexity*, widely used in literature as the execution time characteristics for parallel algorithms, see e.g. [59, 60]. Step complexity is defined as the minimum necessary number of sequential steps needed for the parallel algorithm to complete its computation, given an infinite number of arithmetic processing units it may utilize during its course. In light of such definition it can be seen that matrix-based DWT computation step complexity is

$$\mathbf{S}_M^{1D}(K) = 2K-1, \tag{6}$$

while for lattice structure-based approach it would be equal to

$$\mathbf{S}_L^{1D}(K) = 2K+2. \tag{7}$$

Looking at the above step complexities we can conclude that both matrix and lattice structure-based approaches are *theoretically equivalent* in terms of time effectiveness for the case of their parallel realizations however, in such a case, matrix-based approach consumes about 2 times more computational resources than its lattice structure-based counterpart. Additionally, in the case of classic, sequential realizations (e.g. CPU implementations), matrix-based DWT computation is theoretically about 2 times slower than the lattice structure-based approach, what can be concluded on the basis of equations (4) and (5).

Since discrete wavelet transforms also find their essential applications in 2D signal analysis, such as data and image processing tasks, see e.g. [48] or [2,9,61,62], it is beneficial to consider also the complexities of the analyzed DWT algorithms for the two-dimensional case. Assuming separability of the considered discrete wavelet transforms variants, calculation of the 2D DWT may be performed by application of 1D DWTs to each row of the input signal matrix and then, successively, by applying 1D DWTs to each column of the resulting intermediate matrix. Such technique is commonly referred to as the *row-column* method of 2D DWT computation, see e.g. [2, 9, 61, 62]. It can be verified that the rows and the columns 1D DWT scans in 2D DWT row-column calculation method can be realized with the use of a single application of the computation, structurally identical to the 1D DWT matrix or lattice structure-based calculations, performed on flattened input or intermediate signal matrices with each of their rows or columns padded with their $K-2$ cyclic element repetition. Such a procedure is schematically depicted in Fig. 2 for lattice structure-based DWT with filter length $K = 6$ and 2×6 – element input matrix. Exactly the same procedure can be also applied in the case of the matrix-based 2D DWT computation. Because of the above facts it is now easy to express 2D DWT computational and step complexities, utilizing two of the analyzed approaches, in terms of their one-dimensional counterparts. Since both of the approaches use structurally identical data processing schemes as for 1D case, with the only difference that they need to be applied to a flattened and properly padded input and intermediate signal matrices, all one has to do in order to express the analyzed 2D DWT

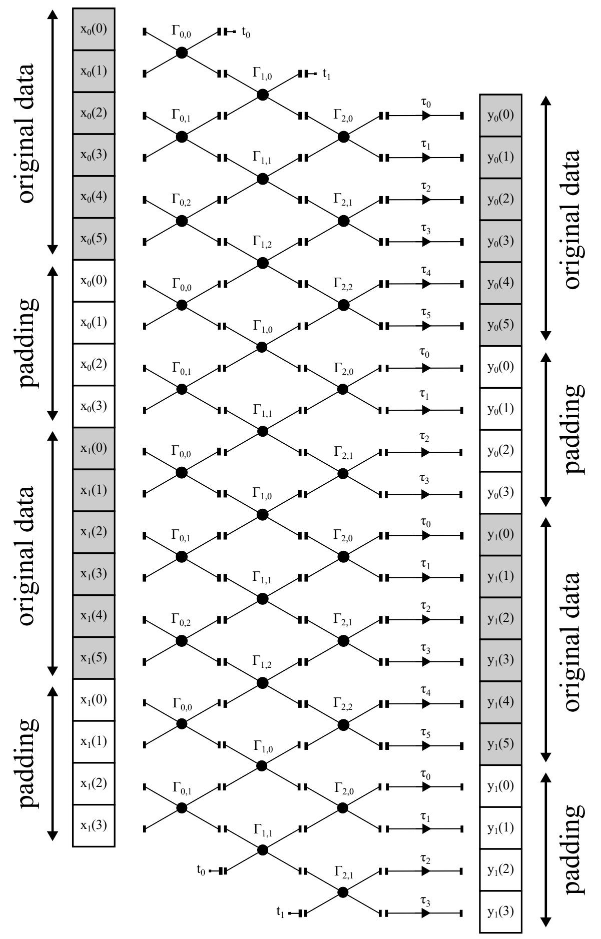

Fig. 2. Rows stage of 2D DWT row-column calculation using 1D lattice-structured DWT data flow graph with filter length $K = 6$ for 2×6 – element input matrix padded with $K-2$ cyclic element repetition

complexities, is to apply the respective formulas (4)–(7) for 1D DWT cases with flattening and padding operations being accounted for in the transform input data size. So, for 2D DWT matrix-based approach the computational complexity would be equal to

$$\mathbf{C}_M^{2D}(N,K) = 2\cdot\mathbf{C}_M^{1D}(N(N+K-2),K), \tag{8}$$

where N is the number of rows and columns of the $N\times N$ element transform input signal matrix, K is the filter length and factor 2 is present above to account for the need for performing the considered calculation twice, referring to the scans of rows and columns in the row-column 2D DWT computation method. It is worth adding that here, as well as in later considerations, we will not include the input matrix transpositions steps complexities, present between the rows and the columns 2D DWT calculations stages, as the ones which are not significant from the standpoint the undertaken analysis. Continuing, for the 2D DWT lattice structure-based approach the computational complexity can similarly to (8) be written in terms of its 1D counterpart in the following form

$$\mathbf{C}_L^{2D}(N,K) = 2\cdot\mathbf{C}_L^{1D}(N(N+K-2),K), \tag{9}$$

where all the terms have the same meaning as before. At last let us also evaluate step complexities for the considered 2D DWT computation methods. Similarly as for computational complexities we can immediately use dependencies (6) and (7), appropriate for 1D DWTs step complexities case, including additionally the necessity of performing two computation passes, referring to the scans of rows and columns in the row-column 2D DWT computation method. So, for 2D DWT matrix-based approach we obtain

$$\mathbf{S}_M^{2D}(K) = 2\cdot\mathbf{S}_M^{1D}(K), \tag{10}$$

and for lattice structure-based method we have

$$\mathbf{S}_L^{2D}(K) = 2\cdot\mathbf{S}_L^{1D}(K). \tag{11}$$

Because of identical forms of dependencies (8), (9) and (10), (11) of 2D DWT computational complexities in relation to their respective one-dimensional counterparts, all conclusions formulated in the previous paragraph, regarding time effectiveness and resources consumption of the analyzed 1D computation approaches, also hold, without any modifications, for the case of 2D DWT computations.

To sum up, for both 1D and 2D considered DWT computation methods the lattice structure-based approach is twice as fast as the matrix-based algorithm in terms of the theoretical computational complexity, while theoretical parallel step complexities of both of the methods are equivalent; however matrix-based approach demands allocation of twice as many computational resources as its lattice structure-based counterpart. We will see later how much those theoretical considerations diverge form true results obtained after implementations of both of the methods on physical GPUs, which can be considered yet one more strong argument for the need for DTCAs execution time modeling on GPUs.

4. GPU PROGRAM EXECUTION MODEL OVERVIEW

In order to develop efficient parallel solutions for GPU it is necessary to understand basic concepts underlying physical GPU architecture. At present, the use of graphics processing units for general purpose numerical computations includes several options regarding different device manufacturers. In this paper we focus on CUDA (*Compute Unified Device Architecture*) by NVIDIA Corporation as a well-established representative of modern GPU architectures. Since the release of the very first devices supporting this technology, CUDA architecture is incessantly being developed and consecutive versions, called *Compute Capability* (CC), bring innovations and continuous increase in overall computational performance while still maintaining relatively low-cost characteristics. This makes the architecture one of the most popular solutions for implementing parallel computations and motivates our choice to focus on that particular alternative.

CUDA devices are representatives of the so-called *Single Instruction Multiple Thread* (SIMT) architecture that strongly resembles the SIMD (Single Instruction Multiple Data) execution model. Figure 3 gives the perspective on the placement of modern GPU architectures in light of Flynn's taxonomy, see e.g. [63]. Both are based on the principle of processing the same instructions by multiple computational units, but additionally, SIMT allows multiple threads to execute independently within single thread set called the *warp*. Within all CUDA devices a warp consists of 32 threads. All threads in a warp execute the same instruction at the same time instant, but on different data which is assigned to each of them and each individual thread has its own instruction address counter and register state.

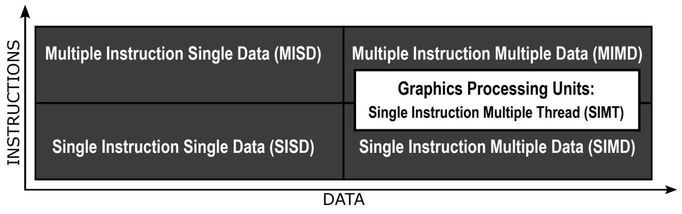
INSTRUCTIONS
Multiple Instruction Single Data (MISD)
Multiple Instruction Multiple Data (MIMD)
Graphics Processing Units:
Single Instruction Multiple Thread (SIMT)
Single Instruction Single Data (SISD)
Single Instruction Multiple Data (SIMD)
DATA

Fig. 3. Architecture of modern GPUs in relation to Flynn's taxonomy

The heart of CUDA architecture is a scalable array of *Streaming Multiprocessors* (SMs). Massive parallelism of a GPU device is built around architectural and logical replications of blocks and operating schemes. Every modern GPU consists of many SMs, and every SM consists of many CUDA cores. From the point of view of logical division, there are threads, blocks and grids. Grids can contain many blocks, and each block can contain many threads. A single block is always assigned to one SM, but a single SM can hold more than one block of threads. For this reason, thousands of threads can be executed at the same time within a single graphics card. During execution of the *kernel function*, i.e. the GPU program, the thread blocks are divided equally among all available SMs. Once a block of threads is assigned to an SM, its threads are grouped into warps and the block stays there until all its threads finish their execution.

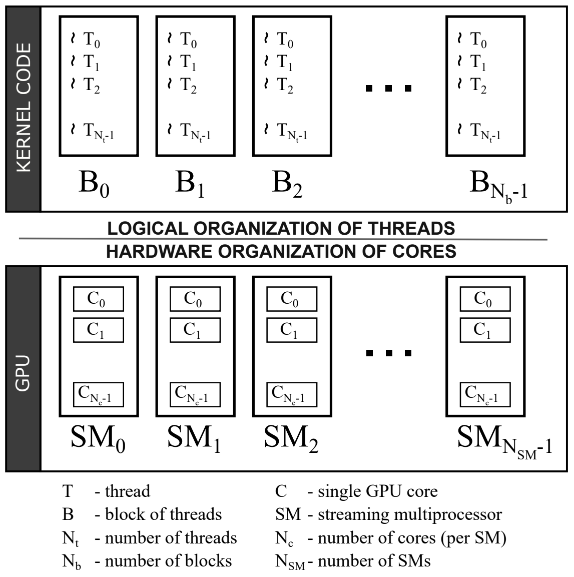

Fig. 4. Comparison of the logical organization of threads and GPU hardware construction

From the logical perspective, threads and blocks can be arranged into, even up to three dimensional, grid structure, but on the hardware side they are always arranged one-dimensionally. Each thread has its unique identifier within the block. Threads are assigned to blocks on a warp-size granularity, so if there is a need to run, e.g. 140 threads, it will be necessary to use 5 warps within a block. The last, incomplete warp, will still use SM resources anyway. Besides CUDA cores, every SM includes also other components, such as: shared memory, load/store operation units, register file, warp schedulers and instruction dispatch units. Simplified scheme presenting the comparison between logical organization of threads within a kernel function and GPUs hardware construction is depicted in Fig. 4, while Fig. 5 shows a sketch view of a single streaming multiprocessor built in CUDA Turing architecture. Proper organization of blocks and threads using the right sizes of grids and blocks can have a huge impact on kernel execution time, so selection of appropriate kernel execution configuration is the key issue in obtaining top overall performance. The limits at each level of the thread hierarchy are device dependent. Execution process sooner or later meets those limits. Of course, experimental research for the most advantageous configuration is possible but is time-consuming and still leaves a doubt if a selected one is surely the best. Furthermore, such trial-and-error approach raises a question why selected execution configuration outperforms the others. In order to understand those dependencies, it is necessary to analyze the performance from the hardware perspective as well.

Yet another important aspect for parallel code efficiency optimization is so-called *latency hiding*. Instruction latency is the time (measured, e.g. in clock cycles) between instruction being issued and completed. Instructions can be classified into two categories, i.e. arithmetic-logic instructions and memory access instructions. Latency of time-consuming instruction can be hidden by issuing other instruction from another resident warp. Memory access instructions require significantly more GPU cycles than arithmetic instructions and can be overlapped between different warps. On the other hand, relatively fast arithmetic-

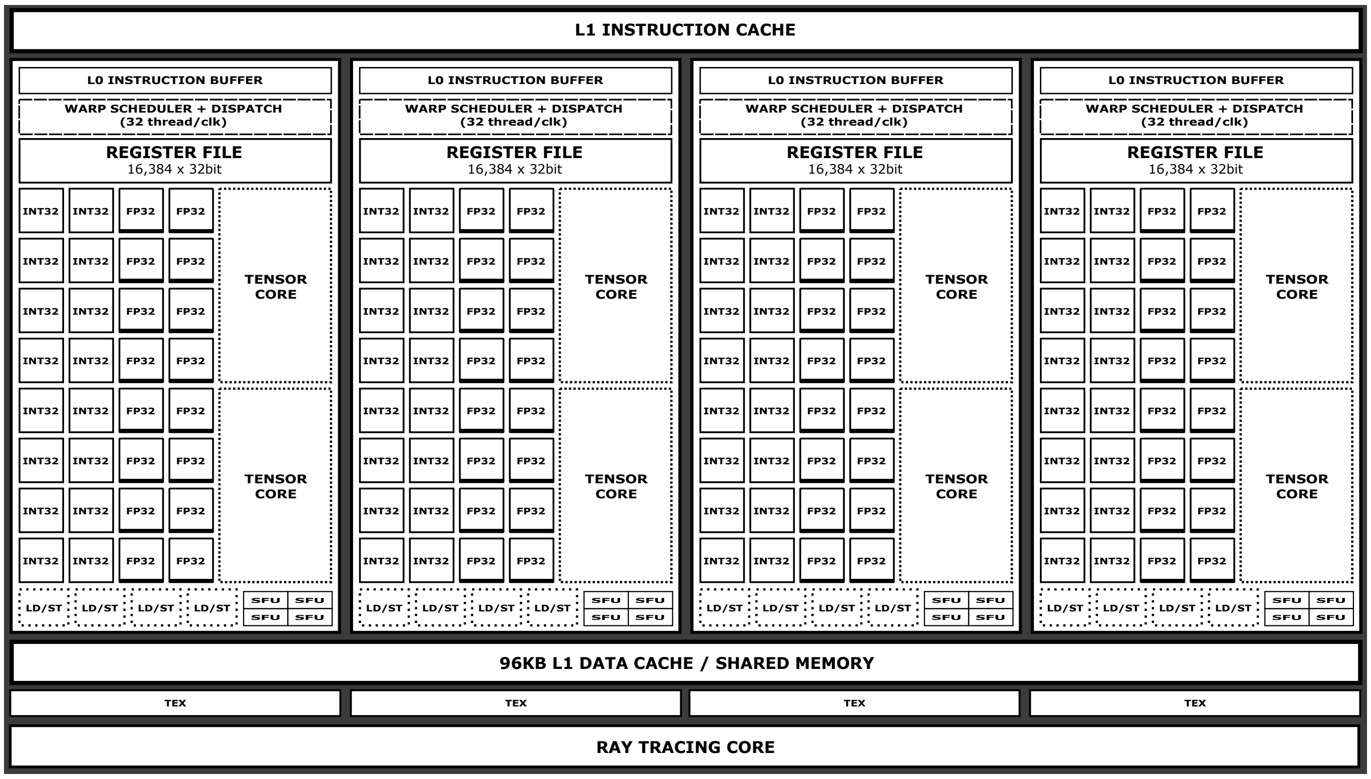
L1 INSTRUCTION CACHE
L0 INSTRUCTION BUFFER
WARP SCHEDULER + DISPATCH (32 thread/clk)
REGISTER FILE 16,384 x 32bit
INT32
FP32
TENSOR CORE
LD/ST
SFU
96KB L1 DATA CACHE / SHARED MEMORY
TEX
RAY TRACING CORE

Fig. 5. Simplified scheme of a Streaming Multiprocessor built in CUDA 7.5 Compute Capability (Turing TU102/104/106)

logic instructions still use GPU cores main ALU pipeline cannot be overlapped between different warps. When it is necessary to wait for the load/store memory operation to complete, SM can switch a warp to another to continue computations, and afterwards, return to the abandoned warp, see Fig. 6. Such an approach is beneficial to performance because it keeps the cores of the device more occupied. Ratio between number of active warps and maximum number of warps per SM (64 since architecture 3.0) is called the *occupancy*.

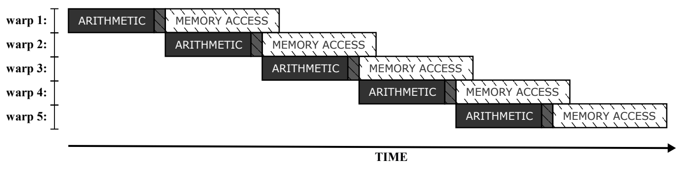

Fig. 6. Example of partial latency hiding by warp switching in an SM

There are of course more aspects of GPU algorithms improvement, which we will briefly discuss. These are: memory access patterns, unavoidable synchronizations, execution paths separation, etc., which also should be taken into account when developing effective GPU implementations. Each SM has its own shared memory and registers. Registers are partitioned among threads and shared memory is partitioned among blocks, so threads can cooperate and communicate within single block. Maximum number of active warps is limited by resources of the device. The state of each warp is always stored in the SM that it is executed on, therefore, such switching does not bring any additional time overhead. When each thread consumes more registers, fewer warps can be placed on an SM, and also when each block consumes more memory, fewer blocks can be stored on that SM as well. Consequently, general parallelism decreases. Therefore, the reduction of registers and memory usage directly increases the overall efficiency. Synchronizations can be done at the system-level and block-level. System-level synchronization waits for the work on whole GPU device to complete, while block-level synchronization waits for all threads within a single block to reach the same point. There is no possibility to synchronize threads from different blocks. Conditional instructions inside kernel functions play also an important role in GPU performance optimization. The basis of CUDA architecture is to execute the same instruction by all threads within a single warp. This is problematic when different threads have different execution paths, again, such situation can lead to reduction of parallelism and decrease overall computational efficiency.

The described, major aspects of general CUDA program execution model form the basis, which our model is built on. In the next section the proposed model will be presented in detail.

5. PROPOSED EXECUTION TIME PREDICTION MODEL

In this section we will present execution time prediction model for parallel GPU realizations of discrete transforms computation algorithms. We regard the presented model to be a set of algorithms allowing for evaluation of the overall execution time for a particular DTCA, when properly fed with the parameters describing the chosen GPU device and the considered DTCA execution configuration.

5.1. General kernel function execution time

Let us first consider a problem of calculation of a general kernel function execution time. Here we assume that the only information we have, regarding a given user's kernel function, is its properly defined computation time (in a sense which will be explained later in this section), and we are not in possession of any knowledge of the internal kernel function implementation. As explained in the previous section, the user invokes a kernel function with two main parameters, namely, the number of blocks N_b and the number of threads per block N_t, which are fed into the GPU host interface from the user's host application level. Having the number of blocks defined, GPU host interface schedules the blocks evenly between all N_{sm} multiprocessors present on the GPU device. Because of this fact, and since the activities undertaken by all the GPU SMs are assumed to be performed simultaneously and symmetrically, i.e. in an identical fashion for all the SMs present in a chosen GPU device, the overall kernel execution time can be determined effectively by considering the respective kernel execution process for a single multiprocessor only (i.e. it would be exactly equal to the time of the kernel execution process for a single SM, performed on the portion of blocks assigned to that particular multiprocessor). Having this in mind, let us consider in more detail the process of the kernel function execution for a single SM.

Following on the earlier considerations, once an evenly spread group of blocks is assigned to given SM it remains there until the very end of their execution process. Due to limited GPU resources not all of the blocks scheduled for execution on a particular SM can become simultaneously *active*. By active blocks we understand (see [10]) only those blocks for which compute resources, such as registers and shared memory, are currently allocated on a given SM. Because of these circumstances the SM execution process is forced to divide all blocks scheduled for execution on that SM, to smaller groups of active blocks, whose operations can be performed simultaneously, and execute those groups in a sequential manner. The maximum number of currently active blocks not only solely depends on the quantity of the GPU resources used by individual threads present in those blocks, but also, on the particular GPU device hardware limitations. In our model we consider two such limitations, namely $L_{b/sm}$, i.e. the maximum number of active blocks per SM and $L_{w/sm}$, i.e. the maximum number of active warps per SM. Those limitations are hardwired into a particular GPU architecture – see, e.g. columns 6th and 7th of Table 3 in Section 6, respectively. Other limitations also exist (e.g. maximum number of registers per SM or maximum amount of shared memory per SM), but they are rarely reached for the case of the DTCAs kernel functions analyzed in this paper, so we can safely omit them in our later considerations. Switching between warp contexts of the active warps (i.e. the warps within the currently active blocks) as well as switching to the next group of active warps blocks to be executed on an SM, generates no time overhead, since the compute resources of the considered warps blocks are kept on-chip during their entire activity time within an SM. All of the above activities lead to an execution of a sequence of consecutive *runs* of active block groups, which is progressively performed until all of the blocks scheduled for execution on a considered SM finish their computations. The size of those consecutive block groups (which we will later refer to as $N_{ab/sm}$, standing for the number of active blocks per SM), taking part in each of the mentioned runs (called the *full runs*), remains constant throughout all of the execution process, except for the last run (the *remaining run*), in which the number of active blocks $\overline{N}_{ab/sm}$ is just the reminder of the blocks scheduled for execution on the considered SM, which still have not been completed. To sum all the above considerations, we can conclude that the calculation of the kernel function overall execution time T_k can be performed using the following formula

$$T_k = t_p + N_r * t + \overline{N}_r * \bar{t}\,. \tag{12}$$

Here N_r is the number of full runs of groups of active blocks assigned to a single SM, $\overline{N}_r$ is the number of the remaining runs (which is either 0 or 1) and additional time t_p, which stands for kernel function execution preparation time, is included to account for the process of initial GPU configuration for the kernel launch, which is done by the GPU host interface before the kernel function is actually executed. This time has to be inevitably included in the model, since for small amount of GPU workloads, it overwhelmingly dominates the time of the whole kernel execution process.

Finally, let us consider the times t and $\bar{t}$ present in the above formula (12). They represent the execution times of actual computational operations of the kernel function, performed on GPU cores, for the full and the remaining runs, respectively. In order to calculate t and $\bar{t}$, we have to descent a step deeper into the analysis of the parallelism taking place in the kernel execution process for the single SM. For this matter we will introduce the notion of the *core package*. In our definition, a core package is a set of 32 GPU cores, which are able to perform a single computational operation (or launch a single memory operation), for all of the threads present in a given, individual warp, at a particular time instant. For the sake of simplicity, let us consider a single group of active blocks being executed on an SM in a full run at a chosen moment of time. Let us assume the number of currently active blocks to be equal to $N_{ab/sm}$, then the number of active warps can be determined by the following expression

$$N_{aw/sm} = N_{ab/sm} * N_{w/b}\,, \tag{13}$$

where $N_{w/b}$ is the number of warps per block, which can be directly determined from the value of the kernel launch parameter N_t, i.e. number of threads per block declared by the user in a kernel function call (see Section 5. 5.4.3). All active warps can now be evenly divided between all of the core packages available on a single SM. Thus, we obtain a set of active warp packages, each containing $N_{aw/cp}$ warps, where $N_{aw/cp}$ (standing for the number of active warps per core package), can be calculated using the following formula

$$N_{aw/cp} = \left\lceil N_{aw/sm}/N_{cp/sm} \right\rceil, \tag{14}$$

where $N_{cp/sm}$ is the number of core packages present in a single GPU SM and $\lceil\cdot\rceil$ is the ceiling function. The value $N_{cp/sm}$ of the number of core packages per SM can be determined directly from the GPU device specification by dividing the number of CUDA cores per SM by 32. Now, once again, we notice that since the operations undertaken by each active warp package are assumed to be performed simultaneously and symmetrically (i.e. in an identical fashion for all of the active warp packages) the overall execution time t of computational operations of the kernel, present in formula (12), can be effectively determined by considering the respective kernel execution process for a single warp package only (i.e. it would be exactly equal to the time of the kernel execution process for a single warp package within a single SM). This enables us to calculate the time t only in terms of the internal structure of the kernel function and $N_{aw/cp}$, i.e. the number of warps assigned to a single core package. The mentioned the internal structure of the kernel function is accounted for by introducing the function $T_c(N_w, \ldots)$, which returns the overall execution time of the actual kernel computation operations, given the number of warps $N_{aw/cp}$ assigned to a single core package. For input parameters, whose meanings are explained below:

- N_{sm} – number of SM present in the GPU device,
- $N_{c/sm}$ – number of GPU cores per SM,
- $L_{b/sm}$ – maximum number of active blocks per SM,
- $L_{w/sm}$ – maximum number of active warps per SM,
- N_b – kernel parameter – number of blocks,
- N_t – kernel parameter – number of threads per block,
- t_p – kernel function execution preparation time,
- $T_c(N_w, \ldots)$ – kernel execution time for a single core package.

Algorithm 1, presents the precise description of the procedure of general kernel function execution time calculation.

The formulation of the function $T_c(N_w, \ldots)$ for a general kernel program run on a single core package will be derived in the next subsection. Finally, it is worth mentioning that determi-

Algorithm 1 General kernel function execution time T_k calculation

```
 1: input:  $N_{sm}, N_{c/sm}, L_{b/sm}, L_{w/sm}, N_b, N_t, t_p, T_c(N_w, ...)$.   ▷ Input parameters.
 2: output:  $T_k$.   ▷ General kernel function execution time.
 3: begin
 4:     ▷ Auxiliary variables calculations.
 5:     $N_{sb/sm} \leftarrow \lceil N_b / N_{sm} \rceil$.   ▷ Number of blocks scheduled for execution on a single SM.
 6:     $N_{w/b} \leftarrow \lceil N_t / 32 \rceil$.   ▷ Number of warps per block.
 7:     $N_{cp/sm} \leftarrow N_{c/sm} / 32$.   ▷ Number of core packages per SM.
 8:     ▷ Kernel function single full run execution time calculation.
 9:     $N_{ab/sm} \leftarrow min\{N_{sb/sm}, \lfloor L_{w/sm} / N_{w/b} \rfloor, L_{b/sm}\}$.   ▷ Num. of active blocks per SM.
10:     $N_{aw/sm} \leftarrow N_{ab/sm} * N_{w/b}$.   ▷ Number of active warps per SM.
11:     $N_{aw/cp} \leftarrow \lceil N_{aw/sm} / N_{cp/sm} \rceil$.   ▷ Number of active warps per core package.
12:     $t \leftarrow T_c(N_{aw/cp}, ...)$.   ▷ Kernel function single full run execution time.
13:     ▷ Kernel function single remaining run execution time calculation.
14:     $\overline{N}_{ab/sm} \leftarrow N_{sb/sm} \, mod \, N_{ab/sm}$.   ▷ Number of active blocks per SM.
15:     $\overline{N}_{aw/sm} \leftarrow \overline{N}_{ab/sm} * N_{w/b}$.   ▷ Number of active warps per SM.
16:     $\overline{N}_{aw/cp} \leftarrow \lceil \overline{N}_{aw/sm} / N_{cp/sm} \rceil$.   ▷ Number of active warps per core package.
17:     $\bar{t} \leftarrow T_c(\overline{N}_{aw/cp}, ...)$.   ▷ Kernel function single remaining run execution time.
18:     ▷ Kernel function overall execution time calculation.
19:     $N_r \leftarrow \lfloor N_{sb/sm} / N_{ab/sm} \rfloor$.   ▷ Number of full runs.
20:     $\overline{N}_r \leftarrow \lceil (N_{sb/sm} \, mod \, N_{ab/sm}) / N_{ab/sm} \rceil$.   ▷ Number of the remaining runs.
21:     $T_k \leftarrow t_p + N_r * t + \overline{N}_r * \bar{t}$.   ▷ Kernel function overall execution time.
22:     return $T_k$.   ▷ Set the final result.
23: end
```

nation of the value of time $\bar{t}$, present in formula (12), can be accomplished by using exactly the same reasoning as the one regarding the time t, presented in the previous paragraph.

5.2. Kernel program execution time for single core package

As mentioned in the previous subsection, we will now present the simulation algorithm for determining kernel program execution time for a single core package, i.e. the calculation of the value of the function $T_c(N_w, \ldots)$, dependent on the number of warps assigned to a given SM single core package, which is one of the parameters of Algorithm 1 of general kernel function execution time evaluation.

Our simulation assumes only 3 general types of low-level kernel program instructions, namely:

- `calc` – arithmetic instructions,
- `load` – memory read instructions,
- `store` – memory write instructions.

This analogous to the models present in [22] and [21], and leads to the division of the whole kernel program to *computation periods* and *memory access periods*, which, if needed, can be grouped together and simulated with a single `calc`, `load` or `store` instructions instances of proper time durations.

We assume that a *kernel program* is an ordered list of pairs of the following form:

$$\texttt{kernel\_program} = [(\texttt{I}_1, d_1), (\texttt{I}_2, d_2), \ldots, (\texttt{I}_{N_i}, d_{N_i})],$$

where N_i is the number of low-level instructions contained in the kernel program source code, $\texttt{I}_k \in \{\texttt{calc}, \texttt{load}, \texttt{store}\}$, $k = 1, \ldots, N_i$, is the k-th low-level kernel program instruction and $d_k \in \mathbb{R}^+$ is its time duration, which will usually be given in GPU cores clock cycles; however, it can also be measured in any time units, suitable for a particular simulation. Such a `kernel_program` constitutes the input of the simulation. The remaining parameters the simulation has to be fed with are: N_w, which is the number of warps designated for execution on a single core package, and t_m – a memory access instructions time scaling factor, which is the model empirical parameter that has to be estimated for a considered kernel program running on a selected GPU from of a given major CC version family (GPU architecture major compute capability). To sum up, for input parameters, explained briefly below:

- `kernel_program` – kernel program low-level instructions,
- N_w – number of warps running on a single core package,
- t_m – memory access instruction time scaling factor,

Algorithm 2 presents the precise description of the procedure of calculation of the kernel program execution time T_c running on a single core package.

Let us comment the presented Algorithm 2 using exemplary ker- nel program from Listing 1. It is given in two variants, with `load` and `store` alternatives present in line 7 of Listing 1. Each operation is characterized by its duration in proper time units, e.g. in GPU cores clock cycles. Mentioned durations are exemplary and are only used for demonstration purposes.

In Figs. 7 and 8 warps execution profiles for a single GPU core package of both of the exemplary kernel program variants are shown, assuming the number of warps N_w running on that core package to be equal to 3 and memory access instructions time scaling factor t_m, depicted as black rectangle at the beginning of each memory access instruction, to be equal to 2. It is worth noting that real proportions of instructions durations of kernel programs from Listing 1 are preserved in Figs. 7 and 8.

Algorithm 2 Kernel program execution time T_c for a single core package

```
declare: struct instr_data { type ∈ { calc, load, store }; duration ∈ $\mathbb{R}^+$ }.
input: $N_w$, $t_m$, instr_data kernel_program[$N_i$].                ▷ Input parameters.
output: $T_c$.                  ▷ Kernel program execution time $T_c$ for a single core package.
begin
  ▷ Simulation initialization.
  $T_c \leftarrow 0$.
  for warp_index = $1, \ldots, N_w$ do instr_counter[warp_index] $\leftarrow 1$.
  ▷ Main simulation loop.
  do
    simulation_completed ← true.
    ▷ Round Robin warps loop.
    for (warp_index ← 1; warp_index $\leqslant N_w$; warp_index++)
      $T_c \leftarrow T_c +$ waitForAllPreviousLoadsToComplete(warp_index).
      ▷ Single warp instructions loop.
      while instr_counter[warp_index] $\leqslant N_i$:
        simulation_completed ← false.
        current_instr_counter ← instr_counter[warp_index].
        instr_counter[warp_index]++.
        if kernel_program[current_instr_counter].type = load:
          $T_c \leftarrow T_c + t_m$.
          if current_instr_counter + 1 $\leqslant N_i$:
            if kernel_program[current_instr_counter + 1].type ≠ load:
              break.  ▷ Switch to the next warp.
        else
          if kernel_program[current_instr_index].type = store:
            $T_c \leftarrow T_c + t_m$.
          else  ▷ The remaining case, i.e. the calc instruction.
            $T_c \leftarrow T_c +$ kernel_program[current_instr_counter].duration.
      end while
    end for
  while not simulation_completed.
  $T_c \leftarrow T_c +$ waitForAllMemoryTransactionsToComplete().
  return $T_c$.                                                    ▷ Set the final result.
end
```

Listing 1 Exemplary kernel program

```
1: $N_w = 3$.
2: $t_m = 2$.
3: kernel_program = (
4:   (load,  15),
5:   (calc,   5),
6:   (calc,   6),
7:   (load,  35), or (store, 35),
8:   (calc,  10),
9:   (store, 15)).
```

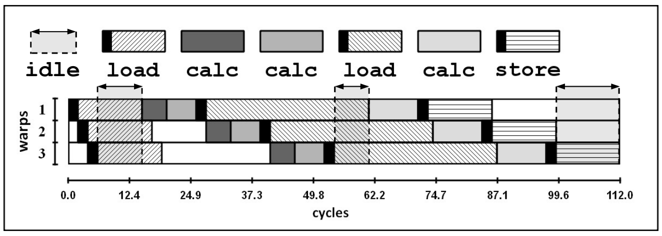

Fig. 7. Kernel program variant 1 from Listing 1 warps execution profile for a single core package within an SM

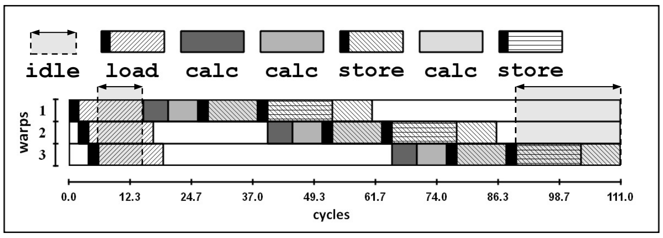

Fig. 8. Kernel program variant 2 from Listing 1 warps execution profile for a single core package within an SM

Algorithm 2 simulates the execution of consecutive warps in a *round Robin* fashion. Each warp has its own instruction counter, which points to the kernel program instruction to be executed at a given simulation step. When the warp is selected for execution (line 12 of Algorithm 2), it first waits until all preceding `loads`, present in that warp, have been completed. This is done with the use of the function `waitForAll PreviousLoadsToComplete()`, which returns the overall execution time of all `loads` preceding the current warp `calc` or `store` instruction (which can be 0 if no preceding `loads` were present in the current warp instruction chain). After that, the selected warp is executed until the last `load` preceding `calc` or `store` instruction is encountered within a kernel program, or the last instruction of the kernel program has been identified (line 15). When the last `load` preceding `calc` or `store` instruction in the current warp instruction chain has been reached or the last instruction of the kernel program has been identified, the simulation is switched to the next warp (lines 23 and 15, respectively) and the execution continues for that newly designated warp. Such procedure continues until all instructions have been simulated for all executed warps (what is indicated by the `simulation_completed` flag being equal to `true` after the main round Robin's loop – line 12, has finished without executing any of the warps instructions). When all instructions of all warps have been performed, in the last step, the emulation process is halted until all memory transactions, triggered earlier by all of the simulated warps, have finished their execution. This is done with the use of the `waitForAllMemoryTransactionsToComplete()` function, which returns the overall execution time of all `loads` and `stores`, triggered earlier by all of the simulated warps, which have not been completed yet. The simulation returns then the total execution time T_c of kernel program running on a single core package of the GPU single SM, ordained with completion of N_w warps.

It can be seen from the exemplary warps execution profiles, depicted in Figs. 7 and 8, that, according to our simulation model, calculation periods[1] (CPs) as well as the *SIMT Front End* parts (SFEs) of memory access periods (MAPs), depicted as black rectangles, are executed exclusively on a whole core package basis, and thus never mutually overlap. On the other hand, the transactional parts (TPs) of memory access periods (depicted with hatched rectangles) of a given warp, may overlap without any limitations with other CPs and MAPs of different warps. This effectively implements the *latency hiding* mechanism, which is one of the most important aspects of mass-parallel GPU computations. However, we can also see that in case our exemplary programs the number of warps is not enough to fully utilize GPU cores pipeline and thus idle periods appear, depicted in Figs. 7 and 8 as gray-shaded rectangles with top double-sided arrows. For a particular warp, the instructions dependencies are more restricted, i.e. for a single warp, in our model we assume that only *Read after Write* (RaW) instruction dependencies take place, all of which all are briefly summarized by the chart shown in Table 2. No further instruction dependencies for a single warp, as well as for all warps executing on a whole core package basis, other than those described above, are assumed to hold. Such choice of instruction dependencies applied in our model comes from the facts that: (i) we assume that all warps running on a single core package share a common ALU pipeline, (ii) we consider memory access operations to be realized by memory transaction subsystem, external to the GPU cores, (iii) we only apply typical algorithmic dependencies occurring in most of the DTCAs implementations we are interested in. All of those assumptions are well motivated and have their counterparts in the existing literature, see e.g. [21, 22, 33].

Table 2

Read after Write dependencies chart for a single warp

RaW instruction dependencies		Following instruction		
		`calc`	`load`	`store`
Preceding instruction	`calc`	*yes*	*yes*	*yes*
	`load`	*yes*	*no*	*yes*
	`store`	*no*	*no*	*no*

At last, once again, let us have a brief look to warps execution profiles, shown in Figs. 7 and 8. In the first variant of the kernel program from Listing 1, all calculation and memory access periods are arranged strictly sequentially on a single warp basis. On the other hand, a simple replacement of a second `load` operation with a `store` instruction in the second variant of the kernel program from Listing 1, causes overlapping of the last 3 operations, i.e `store`, `calc` and second `store` instruction, within all of the 3 considered warps. What is more, since the first `store` operation TP duration is longer than the sum of overall durations of the subsequent `calc` and `store` periods for all of the considered warps, the simulation has to be halted for the last warp on that particular operation to complete. This is a clear demonstration of the need for application of `waitForAllMemoryTransactionsToComplete()` function, which works globally on the whole core package basis. The presented examples also show how different execution profile patterns can become for almost identical kernel programs, which makes the argument on using the simulation method even stronger. At last, its also somewhat surprising, that although there is a huge amount of overlapping taking place for the second variant of the considered kernel program in com-

[1]here, and in later parts of the paper we follow the terminology present in [21, 22] and [33]

parison to its first variant, the overall execution times of both variants are almost equal (i.e. 112 and 111 cycles).

With this last remark we will end the presentation of the simulational part of the proposed model, in the next section we will show how to apply the model practically in case of the particularly simple DTCA example of the *naive* parallel implementation of a vector by matrix multiplication algorithm.

5.3. The application methodology of the model – a case study

In this section we will illustrate how to apply our model practically in the case of the *naive* parallel implementation of a vector by matrix multiplication algorithm, which can be considered one of the simplest DTCAs examples.

We start with presenting CUDA C kernel function code of the examined algorithm, which is our departure point. It is given below, in Listing 2.

Listing 2. CUDA C kernel for vector by matrix multiplication

```
__global__ void mtx_mul_vec(int N, float *mtx, float *vec_in, float *vec_out)
{
 int i = threadIdx.x + blockDim.x * blockIdx.x;
 for (int j = 0; j < N; j++)
  vec_out[i] += mtx[i * N + j] * vec_in[j];
}
```

Here, N stands for the input vector `vec_in` size (and also the output vector `vec_out` size), to be multiplied by $N \times N$-element matrix, whose rows have been flattened and stored in the one-dimensional `mtx` array. Each thread of the kernel function is responsible for multiplication of a single row of the `mtx` matrix with the whole input vector `vec_in`. Kernel function has to be invoked once with the total number of threads equal to N to obtain the output vector values, which are all assumed to be initially set to zeros on the CPU side to avoid additional kernel operations.

The next step is the kernel function compilation to CUDA PTX (parallel thread execution instruction set) assembly source, to enable creation of the kernel program model representation based on the static analysis of the generated PTX code. For the kernel function from Listing 2, the obtained assembly source code is given in Listing 3. The compilation can be done with the CUDA `nvcc` compiler with `--keep` option enabled to keep all intermediate files that are generated during internal compilation steps. The obtained PTX source might require some additional rearrangement, since `nvcc` applies various optimizations, like loop unrolling, etc., so that the resulting code exposes kernel main execution path while still being logically equivalent to its optimized form (such postprocessing was also applied in case of the code presented in Listing 3).

In the next step we perform static analysis of the obtained PTX source, which results in formulation of the model representation of the initial kernel program, whose parameters will be later adjusted to match the time execution measurements taken in the subsequent analysis steps. Listing 4 presents the resulting model representation of the kernel program (here we demonstrate the final form of the model representation of the kernel program, obtained after all adjustments of the memory

Listing 3. PTX code for vector by matrix multiplication kernel

```
 1
 2   // mtx_mul_vec kernel function PTX //           //   +-------------------------+
 3                                                  //   |       mtx_mul_vec       |
 4   ld.param.u32        %r12,   [param_0];         // ↑ +-------------------------+
 5   ld.param.u64        %rd11,  [param_1];         // | | Initial calculations.   |
 6   ld.param.u64        %rd12,  [param_2];         // | +-------------------------+
 7   ld.param.u64        %rd10,  [param_3];         // | | Integer arithmetical    |
 8   cvta.to.global.u64  %rd1,   %rd12;             // | | instructions count: 18. |
 9   cvta.to.global.u64  %rd2,   %rd11;             // | +-------------------------+
10   mov.u32             %r1,    %ctaid.x;          // | |                         |
11   mov.u32             %r2,    %ntid.x;           // | |                         |
12   mov.u32             %r3,    %tid.x;            // | |                         |
13   setp.lt.s32         %p1,    %r12,  1;          // | |                         |
14   @%p1 bra            BR2;                       // | |                         |
15   mad.lo.s32          %r15,   %r1,   %r2,  %r3;  // | |                         |
16   cvta.to.global.u64  %rd13,  %rd10;             // | |                         |
17   mul.lo.s32          %r4,    %r15,  %r12;       // | |                         |
18   mul.wide.s32        %rd14,  %r15,  4;          // | |                         |
19   add.s64             %rd3,   %rd13, %rd14;      // | |                         |
20   and.b32             %r14,   %r12,  3;          // | |                         |
21   mov.u32             %r26,   0;                 // ↓ |                         |
22                                                  //   +-------------------------+
23  BR1:                                            //   | Main loop.              |
24                                                  //   +-------------------------+
25   ld.global.f32       %f18,   [%rd28];           //   | Load.                   |
26   ld.global.f32       %f19,   [%rd27];           //   | Load.                   |
27   fma.rn.f32          %f20,   %f19,  %f18, %f31; //   | Multiply and add.       |
28   st.global.f32       [%rd3], %f20;              //   | Store.                  |
29   add.s64             %rd28,  %rd28, 4;          // ↑ +-------------------------+
30   add.s64             %rd27,  %rd27, 4;          // | | Loop iterators update.  |
31   add.s32             %r26,   %r26,  4;          // | +-------------------------+
32   setp.lt.s32         %p6,    %r26,  %r12;       // | | Integer arithmetical    |
33   @%p6 bra            BR1;                       // ↓ | instructions count: 5.  |
34                                                  //   +-------------------------+
35  BR2:                                            //   |                         |
36                                                  //   |                         |
37   ret;                                           //   | Retrun.                 |
38                                                  //   +-------------------------+
```

Listing 4 `mtx_mul_vec` kernel program

```
 1: $N$ – vector size.
 2: $t_m = 31$.
 3: kernel_program = (
 4:   (calc, 27),
 5:   repeat $N$ times
 6:     (load,  60),
 7:     (load,  60),
 8:     (calc,  10),
 9:     (store, 60),
10:     (calc,  14),
11:   end repeat
12: ).
```

access instructions durations have been made, on the basis of time execution measurements performed later for the analyzed kernel function).

Formulation of the model representation of the kernel program is derived from the PTX kernel source with the use of the following rules. First, we identify calculation periods present in the kernel PTX code, i.e. the continuous sequences of arithmetic, register transfer and branch instructions not interrupted by any of the global `load` or `store` instructions (PTX `ld.global` and `st.global` instructions). In the kernel code from Listing 3, these are sequences present in lines from 4 to 21, lines from 29 to 33, and a single `fma` instruction present in line 27 (we treat initial `ld.param` instructions as regular register transfer instructions, since they perform kernel parameters loads form a fast constant GPU memory). We count then the number of PTX instructions present in each of the identified calculation periods and estimate the cycle count of each such period as the number of its instructions decreased by 1 with a constant number of 10 cycles added to account for the completion of the first instruction in the given calculation period (this is motivated by assumptions holding for the ALU pipelines operation of GPU cores and the estimation of the average cycle count for general arithmetical instructions, see [10, 21] and [22]). According to the stated rules the first calculation period in the analyzed kernel code would take up 27 cycles, c.f. line 4 in Listing 4, i.e. 18 instructions – 1 + 10 cycles = estimated 27 cycles. The second calculation period takes 14 cycles, c.f. line 10 in Listing 4, i.e. 5 instructions – 1 + 10 cycles = estimated 14 cycles. At last, using the stated rules, for a single `fma` instruction present in line 27 of Listing 3, we assign 10 cycles, c.f. line 8 in Listing 4. We then include all calculation periods, interleaved by all identified global `loads` and `stores` (c.f. lines 25, 26 and 28 in Listing 3 and their counterparts from Listing 4, i.e. lines 6, 7 and 9, respectively), obtaining the the model representation of the kernel program, which structurally corresponds to the analyzed PTX kernel source code.

Calculation periods are modeled with the stated above, constant rules of determination of their durations. In the case of memory access operations, i.e. global `loads` and `stores`, their durations are also determined once (for all GPU devices), but now with the help of actual time execution measurements performed for the analyzed kernel function on a selected GPU device. For this purpose we invoke our kernel function for a range of different input data sizes and measure its time performance with the NVIDIA `nvprof` profiling tool. Based on the measurements results we refine durations of memory access instructions in the the model representation of the kernel program (Listing 4, lines 6, 7 and 9) and the memory access instructions time scaling factor t_m (Listing 4, line 2, see also Algorithm 2, Listing 1 and Figs. 7 and 8), to make the model predictions as close as possible to the measured kernel execution times in the whole range of the analyzed input data sizes. Once found, the durations of determined memory instructions can be used, without any modifications, to model execution times of the considered kernel function for other GPUs. In this way we obtain the fixed model representation of the kernel program, valid for predicting execution times for various GPU devices, with only one varying parameter t_m, which has to be adjusted once for a given GPU compute capability class, to account for different memory access time characteristics appropriate for the devices belonging to that class. That is the way our model copes with different memory access time characteristics for different GPU device classes. Having done all the described refinement steps, we obtain a complete model of the analyzed kernel function, ready to be used in the task of execution time prediction of the considered computation algorithm for various GPU devices.

Let us now present the details of the described process of memory access instructions and time scaling factor adjustment for the case of our exemplary kernel function. Using GeForce RTX 2060 GPU (see Table 3 for the device characteristics), for each size N of the input/output vectors, ranging from $N = 2^5$ (warp size) to $N = 2^{14}$ (maximum problem size we could cope with the selected GPU device), we have performed 10 experiments and gathered `nvprof's` execution time statistics. A part of the exemplary `nvprof's` profiling log file, generated for the analyzed kernel function for the input/output vector size $N = 2^{10}$, is presented below in Listing 5. In order to complete

Listing 5. A part of the exemplary profiling log file generated for the analyzed kernel function for input vector size $N = 2^{10}$

```
 1  ==62684== NVPROF is profiling process 62684, command: mtx_mul_vec.exe 30 96 10
 2  ==62684== Profiling application: mtx_mul_vec.exe 30 96 10
 3  ==62684== Profiling result:
 4     Start  Duration  Device                 Name
 5       ...  ...       ...                    ...
 6  315.51ms  7.6000us  ... -                  cudaSetDevice
 7  315.52ms  1.7013ms  ... -                  cudaMalloc
 8  317.22ms  240.20us  ... -                  cudaMalloc
 9  317.46ms  6.9000us  ... -                  cudaMalloc
10  317.47ms  8.8035ms  ... -                  cudaMemcpy
11  317.86ms  8.7098ms  ... GeForce RTX 2060  [CUDA memcpy HtoD]
12  326.27ms  312.30us  ... -                  cudaMemcpy
13  326.59ms  44.500us  ... -                  cudaMemcpy
14  326.62ms  1.9840us  ... GeForce RTX 2060  [CUDA memcpy HtoD]
15  326.63ms  30.400us  ... -                  cudaLaunchKernel (mtx_mul_vec(...))
16  326.66ms  922.50us  ... -                  cudaDeviceSynchronize
17  326.69ms  2.4000us  ... GeForce RTX 2060  [CUDA memcpy HtoD]
18  326.70ms  872.94us  ... GeForce RTX 2060  mtx_mul_vec(...)
19  327.58ms  58.400us  ... -                  cudaMemcpy
20  327.62ms  1.7600us  ... GeForce RTX 2060  [CUDA memcpy DtoH]
21  327.64ms  153.00us  ... -                  cudaFree
22  327.80ms  14.400us  ... -                  cudaFree
23  327.81ms  95.900us  ... -                  cudaFree
24       ...  ...       ...                    ...
```

our model, we need to determine the values of the following model parameters:

- t_p – preparation time of the kernel function execution (see Algorithm 1),
- t_m – preparation time of the kernel function execution (Listing 4),
- d_1–d_3 – durations of memory accesses (Listing 4, lines 6, 7, 9).

Determination of the t_p parameter is fairly straightforward – we set t_p to a median value of the CUDA driver API `cudaLa-unchKernel` function execution times (see line 15 of the exemplary profiling log file in Listing 5), determined on the basis of a profiling information gathered for all of the conducted experiments. We proceed in such a way, since the execution time of the function in question can be considered to be almost constant for all of the analyzed data sizes.

Next, we collect measurements of the considered `mtx_mul-vec` execution times of kernel function (c.f. exemplary `mtx-mul_vec` kernel function invocation measurement in line 18 of Listing 5). On the basis of the gathered `nvprof's` information, for each of the 10 measurements, performed individually for each of the examined data sizes $N = 2^{k+4}$, $k = 1, \ldots, 10$, we extract a single median value and report it as the measured `mtx_mul_vec` execution times of kernel function $t_k^{(d)}$ for that particular input data size $N = 2^{k+4}$.

In the following step, we set the durations of memory access instructions d_1, d_2 and d_3 to the chosen initial values, feasible in the average sense, and run our model to obtain the estimated overall kernel function execution times $t_k^{(m)}$ for all of the considered input data sizes $N = 2^{k+4}$, $k = 1, \ldots, 10$.

After gathering all the measurements and the respective model estimates we calculate model to measured execution times ratios characteristics:

$$r_k = t_k^{(m)} / t_k^{(d)}, \quad k = 1, \ldots, 10, \tag{15}$$

and additionally, based on the obtained characteristics, we determine the *mean absolute percentage error* $\overline{E}$ (i.e. MAPE, see e.g. [40]) of the model prediction performance along with max. prediction mismatch $\widehat{E}$ for the computed characteristics:

$$\overline{E} = \frac{1}{M}\sum_{k=1}^{M}|1-r_k|, \quad \widehat{E} = \max_{k=1,\dots,M}|1-r_k|, \tag{16}$$

where M is the number of measurements (in our case $M = 10$).

In the final step we seek feasible durations d_1, d_2 and d_3, which allow to obtain flat characteristics (15), and adjust time scaling factor of memory access instructions t_m, so that mean absolute percentage error $\overline{E}$ in (16) is minimized.

After completion of all of the described refinements, we obtain the DTCA kernel program model representation, dependent on only two parameters, namely, execution preparation time t_p of kernel function and time scaling factor t_m of memory access instructions, both of which have to be adjusted for each of the classes of GPU devices, belonging to a particular CUDA compute capability architecture.

Going back to our example, the identified memory access instructions durations were all equal to 60 cycles, while the value of time scaling factor of memory access instructions was found to be equal to 31 cycles (see Listing 4). For such values of the adjusted model parameters the obtained mean absolute percentage error $\overline{E}$ was equal to 2,4% with a maximum prediction mismatch $\widehat{E}$ of 6.9%. For demonstrational purposes, in Fig. 9(a) the measured and predicted overall kernel execution times (in milliseconds) are depicted. However because of the high range of the values the considered times cover (c.f. $t_{\min}$ and $t_{\max}$ in Fig. 9(b), denoting the minimum and maximum measured overall kernel execution times, obtained in the course of the conducted experiments), the presented graph might not be sufficiently informative, due to its limited accuracy resolution. Thus, in the later part of the paper, we would describe the obtained results with a lot more instructive accuracy characteristics (15) graphs, example of which is presented in Fig. 9(b), which also include information about the range of the examined execution times (i.e. $t_{\min}$ and $t_{\max}$ values), as well as about the respective errors $\overline{E}$ and $\widehat{E}$ values.

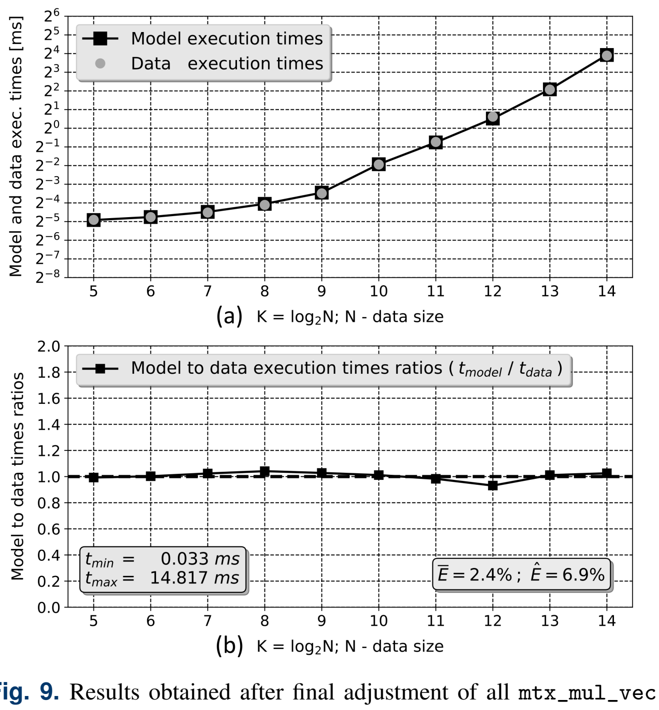

Fig. 9. Results obtained after final adjustment of all `mtx_mul_vec` kernel program parameters – Listing 4, (a) modeled and measured overall execution times [ms], (b) modeled to measured execution times ratios

At last it is worth mentioning that the average time required by the model to generate a single execution time estimate was equal to 0.68 s[2]. The presented model development as well as experimental methodology will be used throughout the rest of the paper for the considered DWT computation algorithms.

5.4. Execution time prediction model representations of parallel DWT computation algorithms

In this section we will formulate the detailed model representations of parallel DWT computation algorithms, which are of our main interest in the context of general confirmation of validity of the proposed approach. For this purpose, CUDA kernel functions of the DWT computation algorithms, described in Section 3, along with their model counterparts and suboptimal parallelism maximization heuristic will be presented and discussed. The considered parallel DWT implementations are not meant to be in any way even close to optimal, their primary purpose is solely the demonstration of our model fidelity.

5.4.1. Model representation of the parallel matrix-based DWT computation algorithm

Let us first present CUDA C implementation of a kernel function for the matrix-based DWT calculation, discussed in the first part of Section 3, it is given in Listing 6. There, the input argument K is the DWT filters length, N is the transform size (the dimensions of the DWT transform matrix, given by (2)), `idata` and `odata` are N-dimensional input and output vectors, respectively, and finally, `coefs` is the flattened $2 \times K$-dimensional DWT filters coefficients matrix. Each thread of the kernel function `mtx` calculates a single DWT output coefficient for each of the individual rows of the transform matrix given in equation (2).

Listing 6. CUDA C kernel of a matrix-based DWT calculation

```
__global__ void mtx(int K, int N, float *idata, float *odata, float *coefs)
{
 int id, k1, k2, i;                             //Definitions of local variables
 float v = 0.0f;                                //stored in GPU register file.
 id = blockIdx.x * blockDim.x + threadIdx.x;    //First calculation
 k1 = 2 * (id / 2);                             //period with associated
 k2 = (id % 2) * L;                             //execution time of 33 cycles.
 for (i = 0; i < K; i++)                        //Loop through all DWT
 {                                              //filter coefficients.
  v = v + idata[k1] * coefs[k2];                //Two load operations with exec.
  k1 = (k1 + 1) % N;                            //times of 120 and 160 cycles
  k2++;                                         //and second calc. period with as-
 }                                              //sociated exec. time of 17 cycles.
 odata[id] = v;                                 //Store operation with associated
}                                               //execution time of 100 cycles.
```

For any even filter length K and transform size N, computation of DWT coefficients with a matrix-based approach, requires only one Listing 6 kernel function launch, invoking to-

[2]for Python 3.7.4 model implementation on Intel Core i7, 2.2 GHz CPU

Listing 7 `dwt_mtx_kernel` program

```
 1: input: K - filter length.
 2: dwt_mtx_kernel = (
 3:  (calc, 33),
 4:  repeat K times
 5:    (load, 120),
 6:    (load, 160),
 7:    (calc, 17),
 8:  end repeat
 9:  (store, 100)
10: ).
```

tal number of N threads. Using methodology discussed in Section 5. 5.3, we arrive at the DWT matrix-based computation kernel model representation, derived from the function in Listing 6 PTX kernel source, which is presented below in Listing 7.

It can be observed that the above kernel program model representation is structurally similar to the one of a parallel vector by matrix multiplication algorithm, given in Listing 4. This should not be a considered to be surprising, since the presented matrix-based DWT computation algorithm might be viewed a slightly modified version of the *naive* vector by matrix multiplication parallel computation method analyzed in Section 5. 5.3, and, as such, is far from being optimal in the time effectiveness sense.

5.4.2. Model representation of the parallel lattice structure-based DWT computation algorithm

We proceed further to the second of the considered DTCA, namely the lattice stru- cture-based DWT calculation algorithm, discussed in the second part of Section 3. CUDA C realization of its kernel function is presented below in Listing 8.

Listing 8. CUDA C kernel function for lattice structure-based DWT calculation

```
 1  __global__ void ltt(int k, int L, int N, float *data, float *coefs)
 2  {
 3   int k0, k1, k2;                                      //Definitions of local variables
 4   float t, s, a, b;                                    //stored in GPU register file.
 5   k0 = 2 * k;                                          //First calculation period with
 6   k1 = 2 * (threadIdx.x + blockDim.x * blockIdx.x) + (k % 2);    //associated
 7   k2 = (k1 + 1) % N;                                   //execution time of 30 cycles.
 8   t = coefs[k0];                                       //Four load operations with
 9   s = coefs[k0 + 1];                                   //associated execution times of
10   a = data[k1];                                        //10, 10, 120 and 120 cycles.
11   b = data[k2];                                        //For all, but the last stage:
12   if (k < L - 1)                                       //second calculation period with
13   {                                                    //associated time of 18 cycles,
14    data[k1] = a + t * b;                               //and two store operations
15    data[k2] = s * a + b;                               //with the associated
16   }                                                    //times of 100 cycles each.
17   else                                                 //The last computational stage:
18   {                                                    //second calculation period with
19    data[k1] = t * a;                                   //associated time of 18 cycles,
20    data[k2] = s * b;                                   //and two store operations
21   }                                                    //with the associated
22  }                                                     //times of 100 cycles each.
```

All threads executed within a single launch of the kernel function from Listing 8, perform a single stage of the lattice structure-based DWT algorithm, depicted in Fig. 1 in Section 3. Complete calculation of the DWT coefficients requires $K/2+1$ consecutive launches of the presented kernel function, where K is the transform filter length. Each of the individual threads implements a single butterfly operation, given by formula (3) in Section 3. Kernel input parameters have the following meaning: L stands for the total number of computational stages of the DWT computation algorithm and is equal to $K/2+1$. Parameter k, varying from 0 to $L-1$, inclusive, is the number of the current computational stage, which a given thread is executed in. Parameter N is the transform size, `data` is the N-element array, which initially holds the values of the input vector, and, since the transformation is done in-place, at the end of the whole computational process, contains the output transform coefficients. Finally, the array `coefs` is holding all of the $2(K+1)$ butterfly operations coefficients.

Once again, using methodology discussed in Section 5. 5.3, we obtain DWT lattice structure-based computation kernel model representation, which is shown above in Listing 9. It is worth commenting on the conditional instruction present in the considered kernel code from Listing 8 throughout lines 12-21 and the associated time of the second calculation period present in its model representation from Listing 9 in line 7. Since a given kernel function launch is related to a single stage of the DWT computation, within such launch, only one execution path will be chosen by all running threads. This means that no warp divergence (as explained in [10]) will occur during all consecutive kernel calls. The fact that both execution paths of the considered conditional statement perform roughly similar computations, enables us to associate the common time of about 18 cycles with both of those paths, simplifying considerably the model representation. Once again we would like to emphasize that the presented DWT lattice structure-based implementation is far from being optimal in the sense of time effectiveness, but because of its computational structure, it serves well as the firm representative of a typical multi-stage DTCA.

Listing 9 `dwt_ltt_kernel` program

```
 1: dwt_ltt_kernel = (
 2:  (calc,  30),
 3:  (load,  10),
 4:  (load,  10),
 5:  (load,  120),
 6:  (load,  120),
 7:  (calc,  18),
 8:  (store, 100),
 9:  (store, 100)
10: ).
```

5.4.3. Total execution times of matrix and lattice structure-based DWT computation methods

Execution time prediction models of both, matrix and lattice structure-based DWT computation methods, are almost complete by means of definitions of the Algorithms 1 and 2, and kernel functions model representations from List's 7 and 9. The only parameters left, which can still be chosen freely by the user in order to optimize the effectiveness of the presented implementations, are kernel launch parameters N_b and N_t (c.f. Algorithm 1), determining the number of blocks and threads within the block for a single kernel call. In this section we will provide heuristic strategy of making suboptimal choice of the values of the considered parameters for both DWT computation approaches, which will then be used in the experimental study, and which effectively complete the considered execution time prediction models.

Let us first present the algorithm for the determination of suboptimal values N_b^* and N_t^* of the kernels launch parameters for both of the considered DWT calculation methods, which will be then discussed in detail.

Here, the input parameters definitions are as follows:

- N_{sm} – number of SMs present in the GPU device,
- $N_{t/w}$ – number of threads per warp (i.e. 32),
- $L_{t/b}$ – maximum number of threads per block (i.e. 1024),
- $\overline{N}_t$ – total number of threads in a single kernel call.

First, it is worth commenting that for both DWT computation methods, since we assume that the size N of the transform is always a power 2, all the input, as well as all of the intermediate, auxiliary and output values present in Algorithm 3, can be verified to be strict integers for any choice of the input parameters values.

Algorithm 3 Kernel launch parameters suboptimal values $N_{bt}^* = \{N_b^*, N_t^*\}$

```
1: input:  $N_{sm}, N_{t/w}, L_{t/b}, \overline{N}_t$ .                ▷ Input parameters.
2: output: $N_{bt}^* = \{N_b^*, N_t^*\}$   ▷ Suboptimal values of the kernel launch parameters.
3: begin
4:    ▷ Auxiliary variables calculations.
5:    $\overline{N}_{sm} \leftarrow N_{sm} + (N_{sm}\, mod\, 2)$ .         ▷ Auxiliary, even number of SMs.
6:    $\overline{N}_{t/sm} \leftarrow \overline{N}_t / \overline{N}_{sm}$ .      ▷ Auxiliary number of threads per SM.
7:    $N_{\min} \leftarrow N_{t/w}, N_{\max} \leftarrow L_{t/b}$ .   ▷ Auxiliary min. and max. numbers of threads.
8:    ▷ Main calculations - testing for complete, mutually exclusive set of conditions.
9:    if $\overline{N}_t < N_{\min}$ then $N_b^* \leftarrow 1, N_t^* \leftarrow \overline{N}_t$ .
10:   if $\overline{N}_t \geq N_{\min}$ and $\overline{N}_{t/sm} < N_{\min}$ then $N_b^* \leftarrow \overline{N}_t / N_{\min}, N_t^* \leftarrow N_{\min}$ .
11:   if $\overline{N}_{t/sm} \geq N_{\min}$ and $\overline{N}_{t/sm} \leq N_{\max}$ then $N_b^* \leftarrow \overline{N}_{sm}, N_t^* \leftarrow \overline{N}_t / \overline{N}_{sm}$ .
12:   if $\overline{N}_{t/sm} > N_{\max}$ then $N_b^* \leftarrow \overline{N}_t / N_{\max}, N_t^* \leftarrow N_{\max}$ .
13:   ▷ Final result evaluated, return it to the calling function.
14:   return $N_{bt}^* = \{N_b^*, N_t^*\}$                    ▷ Set the final result.
15: end
```

The adopted heuristic strategy for making suboptimal choice $N_{bt}^* = \{N_b^*, N_t^*\}$ of the kernels launch parameters, defined by Algorithm 3, is to maximally exploit physical hardware parallelism by making the division of the undertaken DWT computational task into blocks and the respective threads in such a way that the workload imposed on each of the GPU multiprocessors is spread as evenly as possible amongst as large as possible number of GPU SMs. This can be accomplished by the division heuristic, operating under GPU warp granularity regime (which is 32 physical threads per warp), which can be stated in terms of the four following rules:

1) if the total number of threads designated for execution is less than the GPU warp granularity, create one block containing all of the designated threads,
2) if the number of threads per SM is smaller than GPU warp granularity, create maximally numerous set of blocks, i.e. blocks containing minimum possible number of threads each (which happens to be 32 threads per block) and designate them for execution, thus making the maximum number of SMs evenly occupied,
3) if the number of threads per SM is greater than or equal to GPU warp granularity, but smaller than or equal to maximum allowable number of threads per block (i.e. 1024 for all of the considered GPUs), create a set of blocks equipotent with the number of GPU SMs, consisting of the blocks containing equipotent number of threads each,
4) at last, if the number of threads per SM is greater than the maximum allowable number of threads per block, create a set of blocks consisting of the necessary number of blocks containing maximum allowable number of threads each.

Algorithm 3 may be considered to be precise, formal expression of the rules stated above, comprising the strategy of heuristically suboptimal choice $N_{bt}^* = \{N_b^*, N_t^*\}$ of the kernel launch parameters for both of the analyzed DWT computation methods.

We are now ready to formulate the final expressions for total execution times of both of the considered DWT computation methods. Since for any filter length K and transform size N, computation of DWT coefficients with matrix-based approach requires only one Listing 6 kernel function launch, invoking total number of N threads, we can write the expression for matrix-based approach DWT computation total execution time T_m in the following way:

$$T_m(\mathbf{G}_p, N, K, t_p, t_m) = T_k\Big(\mathbf{G}_p,\, N_{bt}^*(\mathbf{G}_p, N),\\ t_p,\, T_c\big(N_w,\, t_m,\, \texttt{dwt\_mtx\_kernel}(K)\big)\Big), \tag{17}$$

where functions T_k and T_c denote Algorithms 1 and 2, respectively, with their consecutive input parameters, similarly N_{bt}^* refers to the stated above Algorithm 3, the set $\mathbf{G}_p = \{N_{sm}, N_{c/sm}, L_{b/sm}, L_{w/sm}, N_{t/w}, L_{t/b}\}$ denotes GPU device parameters specification (see Table 3 and Algorithms 1 and 3 parameters descriptions), t_p stands for the preparation of the kernel function execution time (see equation 12 explanation), t_m is the memory access instruction time scaling factor (see Algorithm 2 parameters descriptions, and also Listing 1 and the discussion of Figs. 7 and 8 discussion), and finally `dwt_mtx_kernel` is the matrix-based DWT computation kernel model representation defined in Listing 7, which comprises the T_c simulation input program.

Similarly, let us formulate the final expression for total execution time of the lattice structure-based DWT computation method as follows

$$T_l(\mathbf{G}_p, N, K, t_p, t_m) = \left(\frac{K}{2}+1\right) T_k\left(\mathbf{G}_p,\, N_{bt}^*\left(\mathbf{G}_p,\, \frac{N}{2}\right),\\ t_p,\, T_c\big(N_w, t_m,\, \texttt{dwt\_ltt\_kernel}\big)\right). \tag{18}$$

In the above equation all symbols have the same meaning as that explained earlier, except that now Algorithm 1 is invoked for total number of threads equal to $N/2$ and `dwt_ltt_kernel` is the lattice structure-based DWT computation kernel model representation defined in Listing 9, which comprises the T_c simulation input program. In (18) the function T_k is multiplied by the additional factor dependent on the transform filter length K, because in case of the lattice structure approach, the complete computation of DWT coefficients requires $\left(\frac{K}{2}+1\right)$ invocations of the kernel function from Listing 8.

To sum up, equations (17) and (18) comprise together a complete description of the proposed execution time prediction model of the parallel GPU implementations of the matrix and lattice structure-based DWT computation methods. Practical verification of the presented model will be carried out in the following section.

6. EXPERIMENTAL RESULTS AND MODEL VERIFICATION

In this section we will present experimental results which enable validation of the effectiveness of our model in terms of its prediction accuracy and computation time efficiency.

First, let us list the full set of the model parameters used in the conducted experiments for the tested GPU devices, they are all gathered in Table 3 and, for clarification purposes, their meanings are briefly reminded in the following chart:

- CC – GPU architecture CUDA *Compute Capability* class,
- F – GPU cores clock frequency measured in MHz,
- N_{sm} – number of SMs present in the GPU device,
- $N_{c/sm}$ – number of GPU cores per SM,
- $L_{b/sm}$ – maximum number of active blocks per SM,
- $L_{w/sm}$ – maximum number of active warps per SM,
- $N_{t/w}$ – number of threads in a single warp,
- $L_{t/b}$ – maximum number of threads in a single block,
- N_b – kernel parameter – number of blocks,
- N_t – kernel parameter – number of threads per block,
- t_p – kernel function execution preparation time in µs,
- t_m – memory access instructions time scaling factor in cycles.

For our experiments, we have used 6 different NVIDIA graphics cards, whose model names are given in the second column of Table 3, representing a fairly broad spectrum of GPUs compute architectures, specified by compute capability indices (CCs), listed in the first column of the above table. According to their types, the model parameters are divided into two main groups. The first group consists of the device dependent parameters, which can be determined directly on the basis of the particular GPU model technical specification. The meanings of all of those parameters were explained in detail in Sections 6. 5.1 and 6. 5.4.3, except for the first parameter F, which is the selected GPU cores clock frequency, reported in MHz. The second group of parameters, listed in Table 3, are architecture (i.e. CC) dependent ones, which means that their particular values, valid for a specific DTCA, are approximately equivalent across all of the GPU devices, belonging to the same CC class (c.f. the considered parameter values for the last two GPU devices listed in Table 3). These parameters are: t_p, which is the execution preparation time of the kernel function, analyzed in Sections 6. 5.1 and 6. 5.3, and measured in microseconds, and t_m, i.e. memory access instruction time scaling factor, discussed in Sections 6. 5.2 and 6. 5.3, and measured in GPU cores clock cycles. It is worth commenting that the t_p parameter has to be measured with the time units independent of GPUs cores clock frequency, since it translates to the respective CUDA driver API function call time (see Section 6. 5.2) handled by the GPU host interface, external to the GPU cores unit and synchronized with the separate clock. The parameters gathered in Table 3 constitute the fully comprehensive list of inputs, needed for our model to perform estimates of execution times for a selected GPU architecture and the chosen parallel computation algorithm.

Let us now present the applied methodology, enabling verification of our model in terms of prediction accuracy and computation time effectiveness. For each of 6 of the tested GPU devices we have performed a series of 4 experiments involving DWT calculation for 4 different, practical filter lengths, $K = 8$, 10, 12 and 14, with the use of both of the considered parallel computational methods. Their results are depicted on charts in Figs. 10–15 as `Exp.(M.`i`)` and `Exp.(L.`i`)`, where symbol M stands for the matrix-based and L indicates lattice structure-based parallel DWT computation algorithm, while $i = 1,\ldots,N_e$, denotes the individual experiment number out of total $N_e = 24$ conducted tests for each of the considered computational methods. For a given individual experiment `Exp.(`μ`.`i`)`, $\mu \in \{\text{M}, \text{L}\}$, $i \in \{1,\ldots,N_e\}$, $N_m = 15$ execution time measurements $t_k^{(d)}$ of the respective DWT computation method running on the selected GPU device, were performed and collected, for the range of the input vector sizes of $N = 2^{k+5}$, $k = 1,\ldots,N_m$ elements. At the same time, employing model parameters gathered in Table 3, valid for the selected GPU device and the considered DWT computation method, the respective estimates of execution times $t_k^{(m)}$, $k = 1,\ldots,N_m$, were calculated, depending on the DWT computation method in question, with the use of formulas (17) or (18). After all the measurements $t_k^{(d)}$ and the respective model estimates $t_k^{(m)}$ have been gathered for the

Table 3
Model parameters for the matrix and lattice structure-based DWT computation methods for the tested GPU devices

Model parameters		Device dependent							Architecture dependent			
									DWT matrix		DWT lattice	
CC	Device	F [MHz]	N_{sm}	$N_{c/sm}$	$L_{b/sm}$	$L_{w/sm}$	$N_{t/w}$	$L_{t/b}$	t_p [µs]	t_m [cc]	t_p [us]	t_m [cc]
2.1	**GT 720M**	1550	2	32	8	48	32	1024	18.7	9.5	11.0	33.0
3.0	**K1000M**	706	2	192	16	64	32	1024	11.2	33.0	11.2	47.0
5.0	**GTX 860M**	1020	5	128	32	64	32	1024	8.7	17.0	7.9	15.0
6.1	**GTX 1070**	1760	10	128	32	64	32	1024	6.3	18.5	5.8	19.0
7.5	**RTX 2060**	1200	30	64	16	32	32	1024	5.2	1.1	5.0	12.5
7.5	**RTX 2080**	1545	68	64	16	32	32	1024	5.2	1.1	5.6	12.3

particular experiment Exp.(μ.i), using the formula (15), the corresponding modeled to measured execution times ratios r_k were calculated (depicted as black squares, each corresponding to a particular input data size, in the experiment graphs present in Figs. 10–15), and based on their values, with the use of formulas (16), the mean absolute percentage error $\overline{E}_i^{\mu}$ of the model prediction performance along with the maximum prediction mismatch $\widehat{E}_i^{\mu}$, were determined for the considered individual experiment. Their values are also presented, for each of the individual experiments, on the experiment graphs in Figs. 10–15, along with the values $t_{\min}$ and $t_{\max}$, indicating the minimum and the maximum measured execution times encountered in the course of the corresponding experiment.

To evaluate a general characteristics of the proposed model execution time prediction performance, we have calculated several global estimates of the mean absolute percentage error and the maximum prediction mismatch, corresponding both, individually to each of the considered DWT computation methods, as well as to the overall model prediction efficiency. They are defined below in equations (19):

$$\overline{E}^{\mu} = \frac{1}{N_e}\sum_{i=1}^{N_e} \overline{E}_i^{\mu}, \qquad \overline{E} = \frac{1}{2}\left(\overline{E}^{M} + \overline{E}^{L}\right), \tag{19a}$$

$$\widehat{E}^{\mu} = \max_{i=1,\ldots,N_e} \widehat{E}_i^{\mu}, \qquad \widehat{E} = \max\left\{\widehat{E}^{M},\, \widehat{E}^{L}\right\}, \tag{19b}$$

where, as earlier, $\mu \in \{\text{M, L}\}$ indicates the matrix or lattice-structure based parallel DWT computation method and $N_e = 24$ denotes the total number of the conducted experiments. All of the defined above, 6 global measures of the model prediction performance, are reported on the left-hand side of the results Table 4, in its *Inaccuracy measures* section.

Additionally, to confront the model prediction accuracy against its computation time costs, we have calculated several global model computation time effectiveness measures corresponding, similarly as in the previous case, to both, each of the considered DWT computation methods alone, as well as to the overall model computation time efficiency characteristics. They are defined as follows:

$$\overline{T}^{\mu} = \frac{1}{N_e N_m}\sum_{i=1}^{N_e}\sum_{k=1}^{N_m} \overline{T}_{i,k}^{\mu}, \qquad \overline{T} = \frac{1}{2}\left(\overline{T}^{M} + \overline{T}^{L}\right), \tag{20a}$$

$$\widehat{T}^{\mu} = \max_{\substack{i=1,\ldots,N_e \\ k=1,\ldots,N_m}} \widehat{T}_{i,k}^{\mu}, \qquad \widehat{T} = \max\left\{\widehat{T}^{M},\, \widehat{T}^{L}\right\}, \tag{20b}$$

where, as before, $\mu \in \{\text{M, L}\}$ indicates the matrix or lattice-structure based parallel DWT computation method, $N_e = 24$ denotes the total number of the conducted experiments, $N_m = 15$ stands for the number of execution time measurements performed for each of the individual experiments Exp.(μ.i), $i \in \{1,\ldots,N_e\}$ and $\overline{T}_{i,k}^{\mu}$ is the duration of the model output estimate calculation process, reported for the $\{1,\ldots,N_m\} \ni k$-th measurement of the i-th experiment, conducted for the μ-th DWT computation method. All of the defined above, 6 global measures of the model computation time effectiveness, are reported on the right-hand side of the results in Table 4, in its *Simulation durations* section, for the model implementation developed in Python 3.7.4 programming language, running on the Intel Core i7, 2.2 GHz CPU-based system.

At last, to investigate the model accuracy in predicting the characteristics of execution time comparisons between different DTCAs, for each pair of the corresponding individual measurements $t_k^{(d)}$, $k \in \{1,\ldots,N_m\}$ performed for the matrix-based and the lattice structure-based DWT computation methods within each of the respective Exp.(M.i) and Exp.(L.i), $i = 1,\ldots,N_e$ experiments, the ratio $R_{i,k}^{(d)}$ between the reported matrix-based DWT method computation time and its respective lattice structure-based DWT method counterpart has been calculated. The analogous ratios $R_{i,k}^{(m)}$ have been determined for each pair of the corresponding individual model execution times estimates $t_k^{(m)}$, calculated for the considered matrix-based and the lattice structure-based DWT computation methods in all of the conducted experiments Exp.(M.i) and Exp.(L.i), $i = 1,\ldots,N_e$. All of the mutually corresponding ratios $R_{i,k}^{(d)}$ and $R_{i,k}^{(m)}$ characteristics are depicted in graphs Exp.(R.i), $i = 1,\ldots,N_e$, presented in Figs. 10–15, where the ratios regarding the GPU measurements data are indicated by the grayish squares, while their model estimate counterparts are represented by the black ones. In each of the mentioned graphs the average $\overline{E}_i^{R}$ and maximum $\widehat{E}_i^{R}$ comparison prediction errors are reported, calculated for each of the respective experiments Exp.(R.i), $i = 1,\ldots,N_e$, in the following way:

$$\begin{aligned} \overline{E}_i^{R} &= \frac{1}{N_m}\sum_{k=1}^{N_m} \frac{\left|R_{i,k}^{(d)} - R_{i,k}^{(m)}\right|}{R_{i,k}^{(d)}}, \\ \widehat{E}_i^{R} &= \max_{k=1,\ldots,N_m} \frac{\left|R_{i,k}^{(d)} - R_{i,k}^{(m)}\right|}{R_{i,k}^{(d)}}. \end{aligned} \tag{21}$$

Based on all of the collected data, gathered in the way described above, we have finally calculated the global estimates of the considered errors, characterizing the model general comparison

Table 4
Model inaccuracy and simulation durations results for the matrix and lattice structure-based DWT computation algorithms

Inaccuracy measures				Simulation durations			
Inaccuracies	DWT matrix	DWT lattice	Overall	Durations	DWT matrix	DWT lattice	Overall
Average	2.6%	3.0%	**2.8%**	**Average**	3.1 ms	3.8 ms	**3.5 ms**
Maximum	11.4%	14.5%	**14.5%**	**Maximum**	12.6 ms	19.3 ms	**19.3 ms**

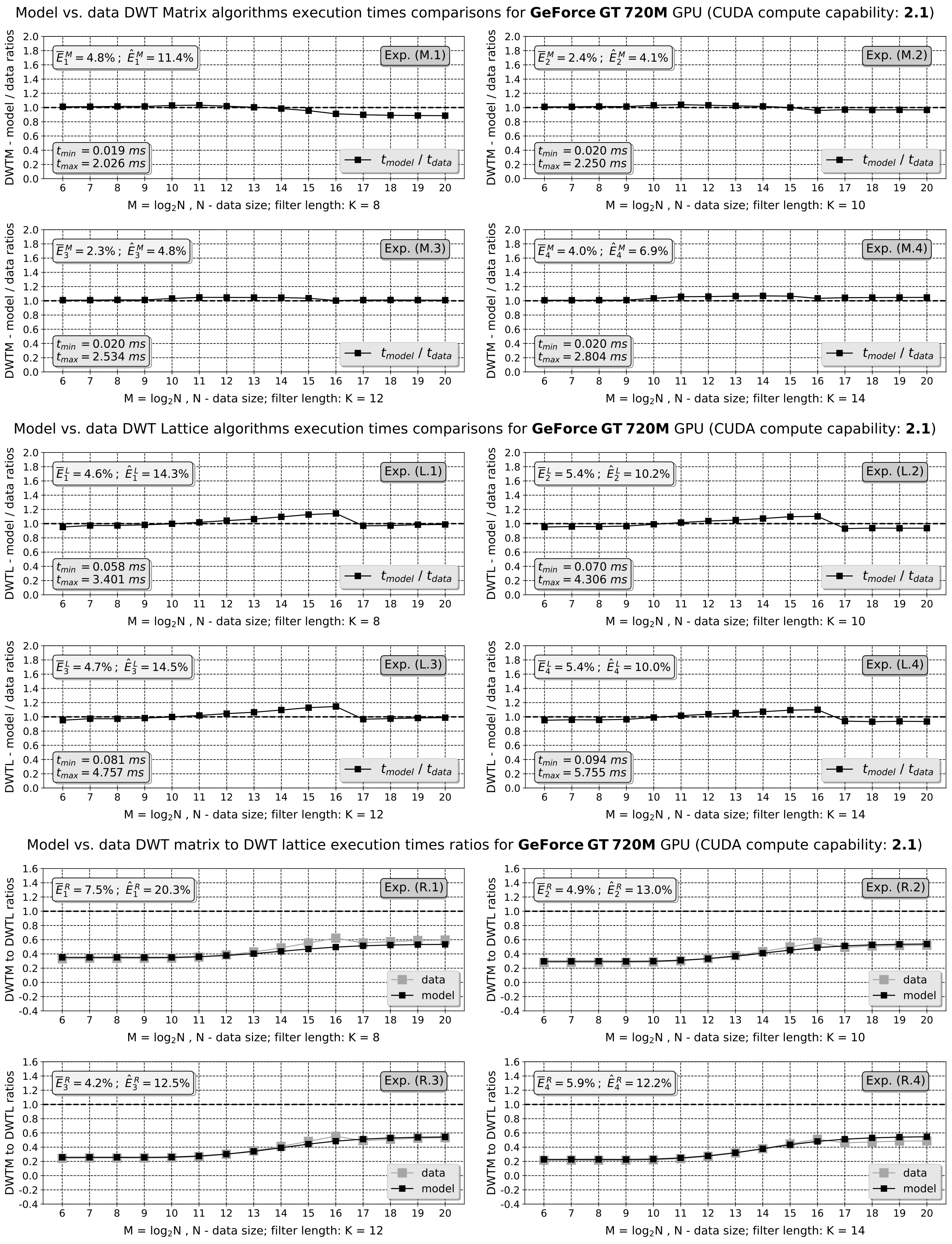

Fig. 10. Experimental results for GeForce GT 720M GPU with DWT input data sizes $M = 2^6, \ldots, 2^{20}$ and filter lengths $K \in \{8, 10, 12, 14\}$.
Experiments (M.1)–(M.4): DWT matrix algorithm modeled to measured execution times ratios.
Experiments (L.1)–(L.4): DWT lattice algorithm modeled to measured execution times ratios.
Experiments (R.1)–(R.4): DWT matrix to DWT lattice algorithms modeled to measured execution times ratios

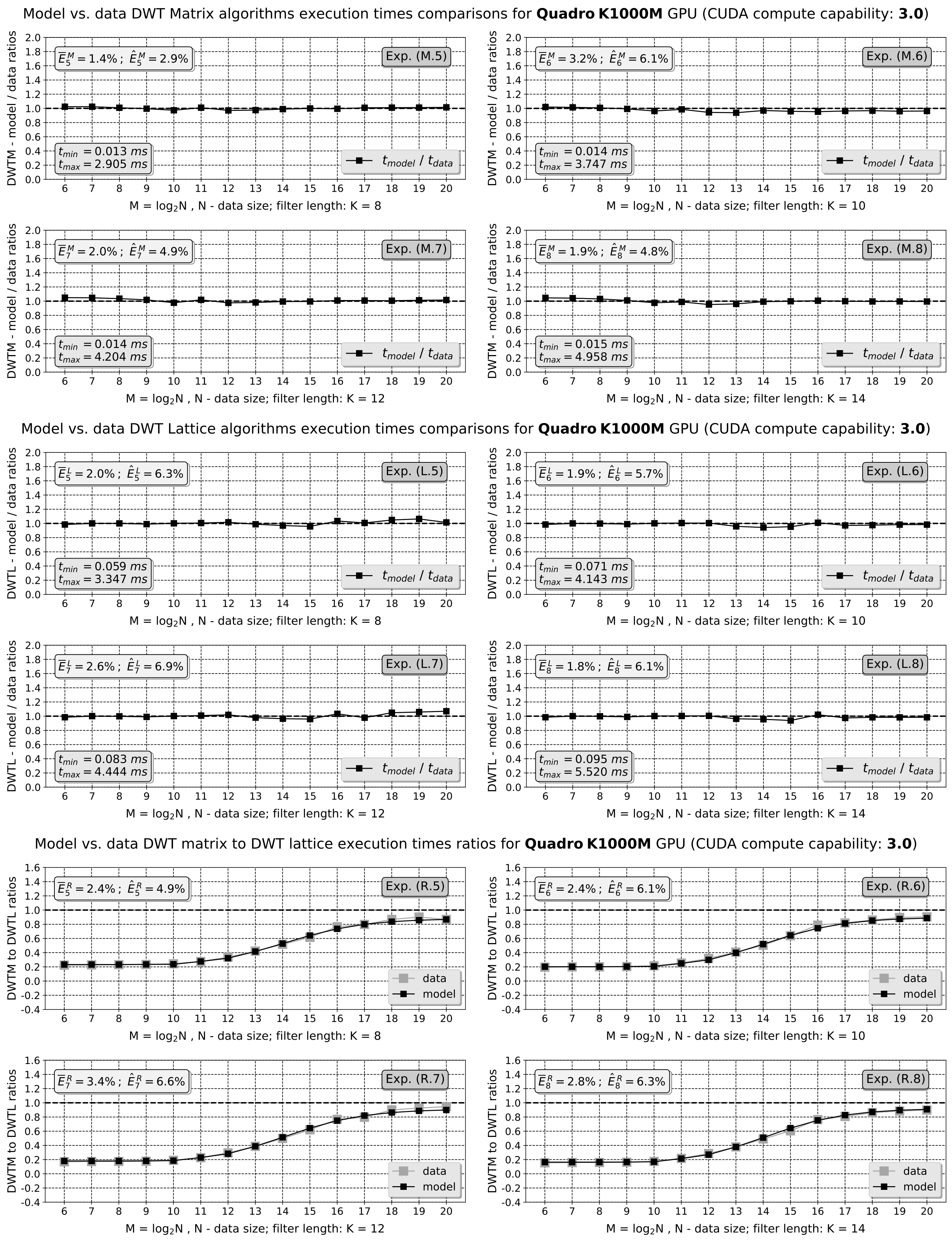

Fig. 11. Experimental results for Quadro K1000M GPU with DWT input data sizes $M = 2^6, \ldots, 2^{20}$ and filter lengths $K \in \{8, 10, 12, 14\}$.
Experiments (M.5)–(M.8): DWT matrix algorithm modeled to measured execution times ratios.
Experiments (L.5)–(L.8): DWT lattice algorithm modeled to measured execution times ratios.
Experiments (R.5)–(R.8): DWT matrix to DWT lattice algorithms modeled to measured execution times ratios

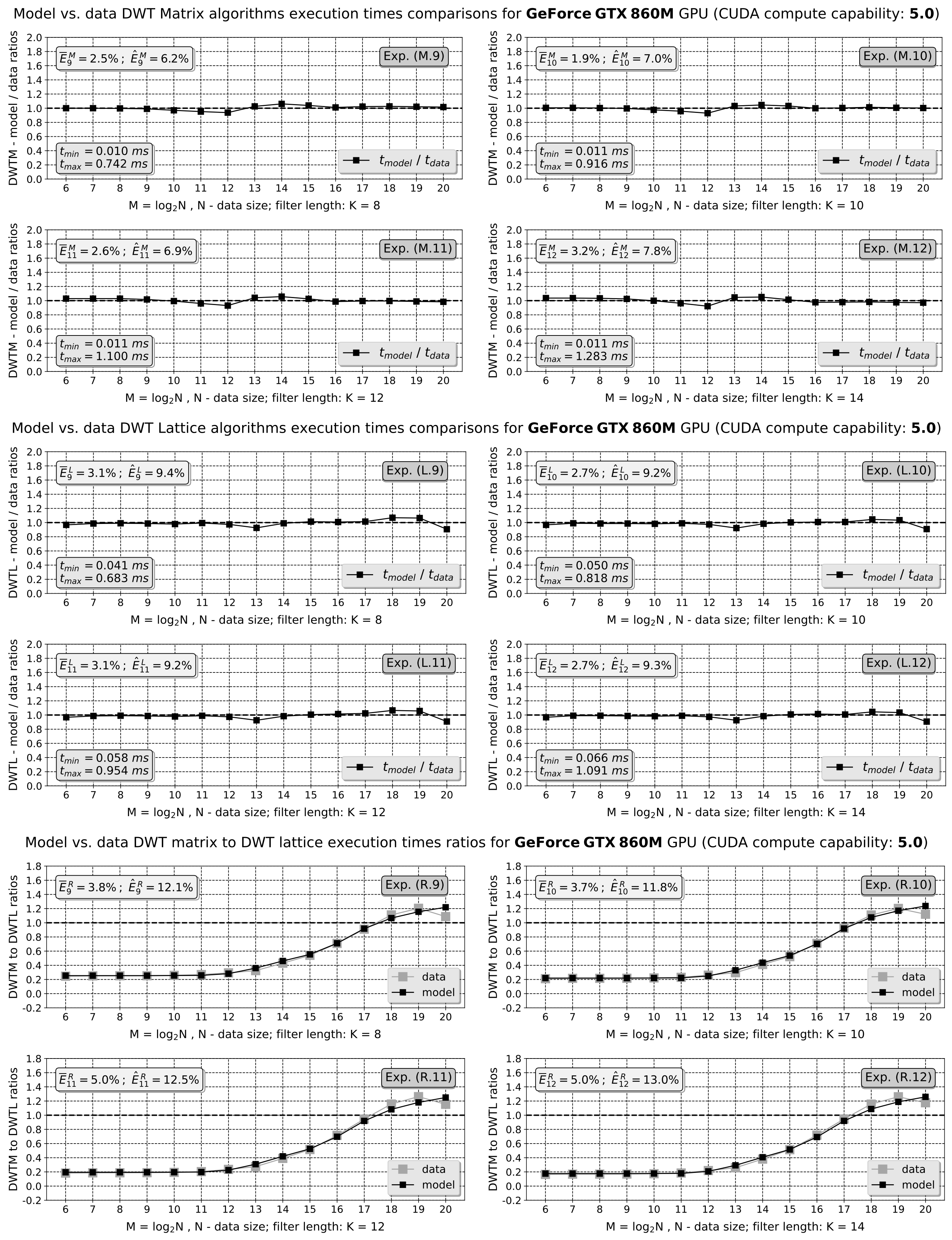

Fig. 12. Experimental results for GeForce GTX 860M GPU with DWT input data sizes $M = 2^6, \ldots, 2^{20}$ and filter lengths $K \in \{8, 10, 12, 14\}$.
Experiments $(\mathtt{M.9})$–$(\mathtt{M.12})$: DWT matrix algorithm modeled to measured execution times ratios.
Experiments $(\mathtt{L.9})$–$(\mathtt{L.12})$: DWT lattice algorithm modeled to measured execution times ratios.
Experiments $(\mathtt{R.9})$–$(\mathtt{R.12})$: DWT matrix to DWT lattice algorithms modeled to measured execution times ratios

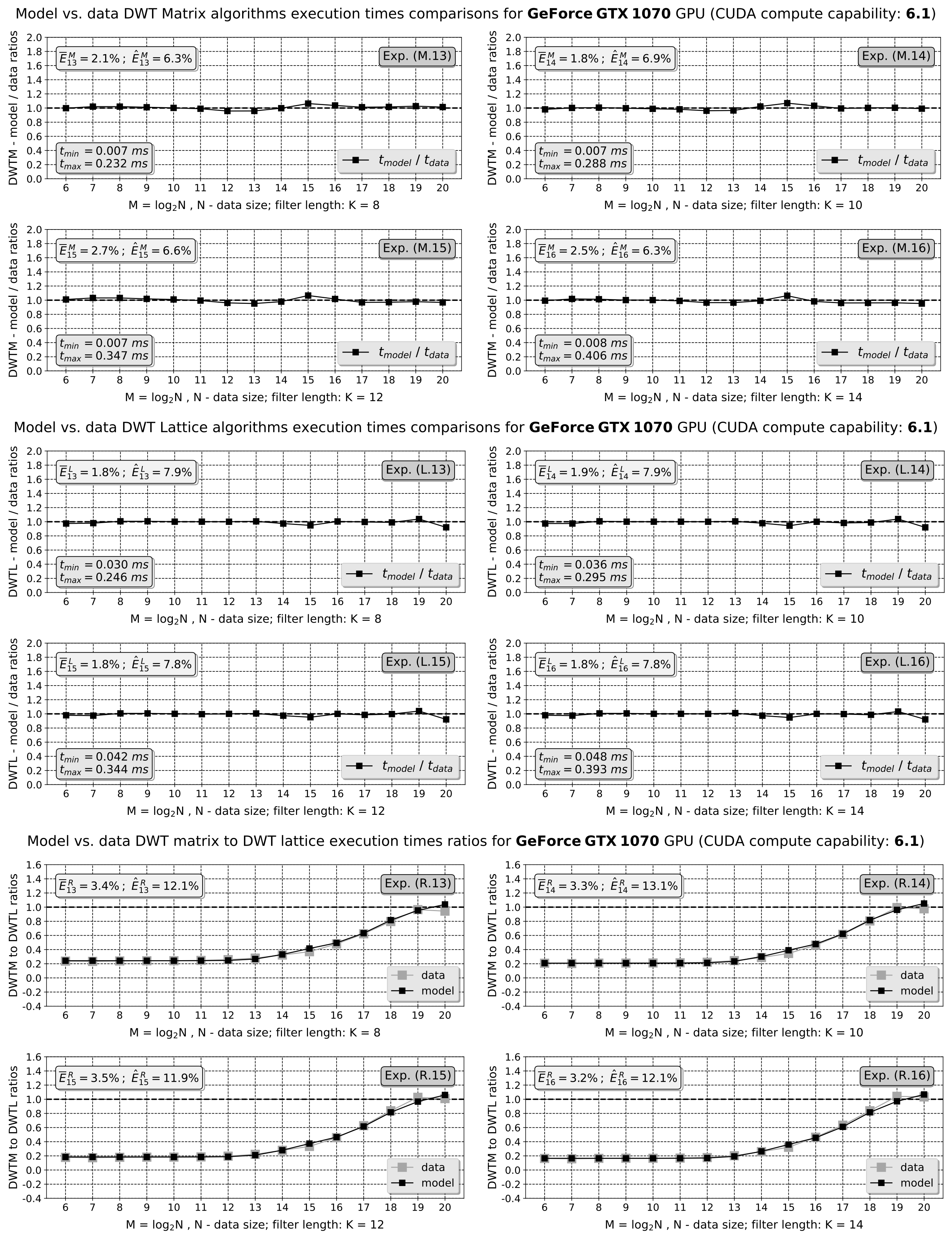

Fig. 13. Experimental results for GeForce GTX 1070 GPU with DWT input data sizes $M = 2^6, \ldots, 2^{20}$ and filter lengths $K \in \{8, 10, 12, 14\}$.
Experiments (M.13)–(M.16): DWT matrix algorithm modeled to measured execution times ratios.
Experiments (L.13)–(L.16): DWT lattice algorithm modeled to measured execution times ratios.
Experiments (R.13)–(R.16): DWT matrix to DWT lattice algorithms modeled to measured execution times ratios

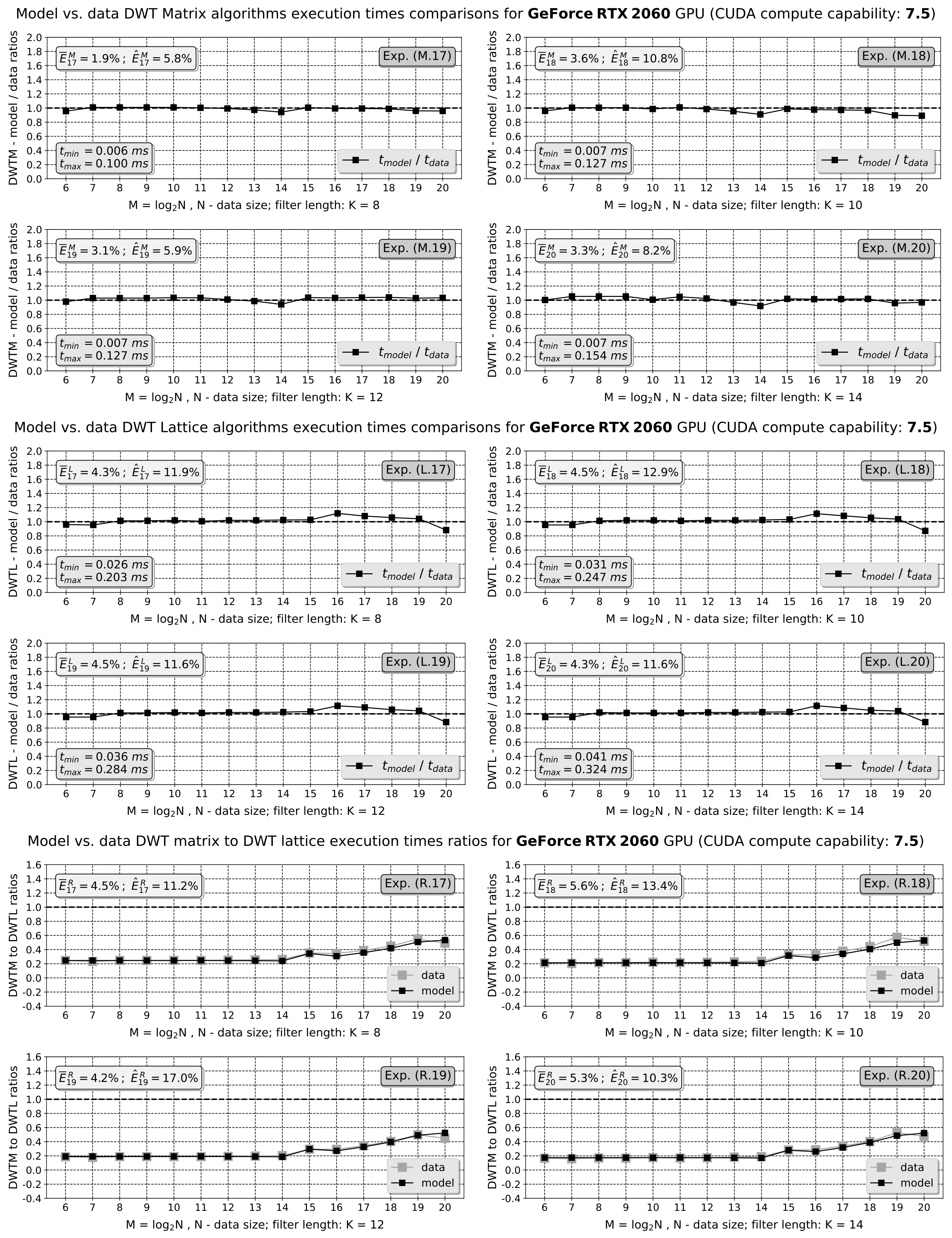

Fig. 14. Experimental results for GeForce RTX 2060 GPU with DWT input data sizes $M = 2^6, \ldots, 2^{20}$ and filter lengths $K \in \{8, 10, 12, 14\}$.
Experiments $(\mathtt{M.17})$–$(\mathtt{M.20})$: DWT matrix algorithm modeled to measured execution times ratios.
Experiments $(\mathtt{L.17})$–$(\mathtt{L.20})$: DWT lattice algorithm modeled to measured execution times ratios.
Experiments $(\mathtt{R.17})$–$(\mathtt{R.20})$: DWT matrix to DWT lattice algorithms modeled to measured execution times ratios

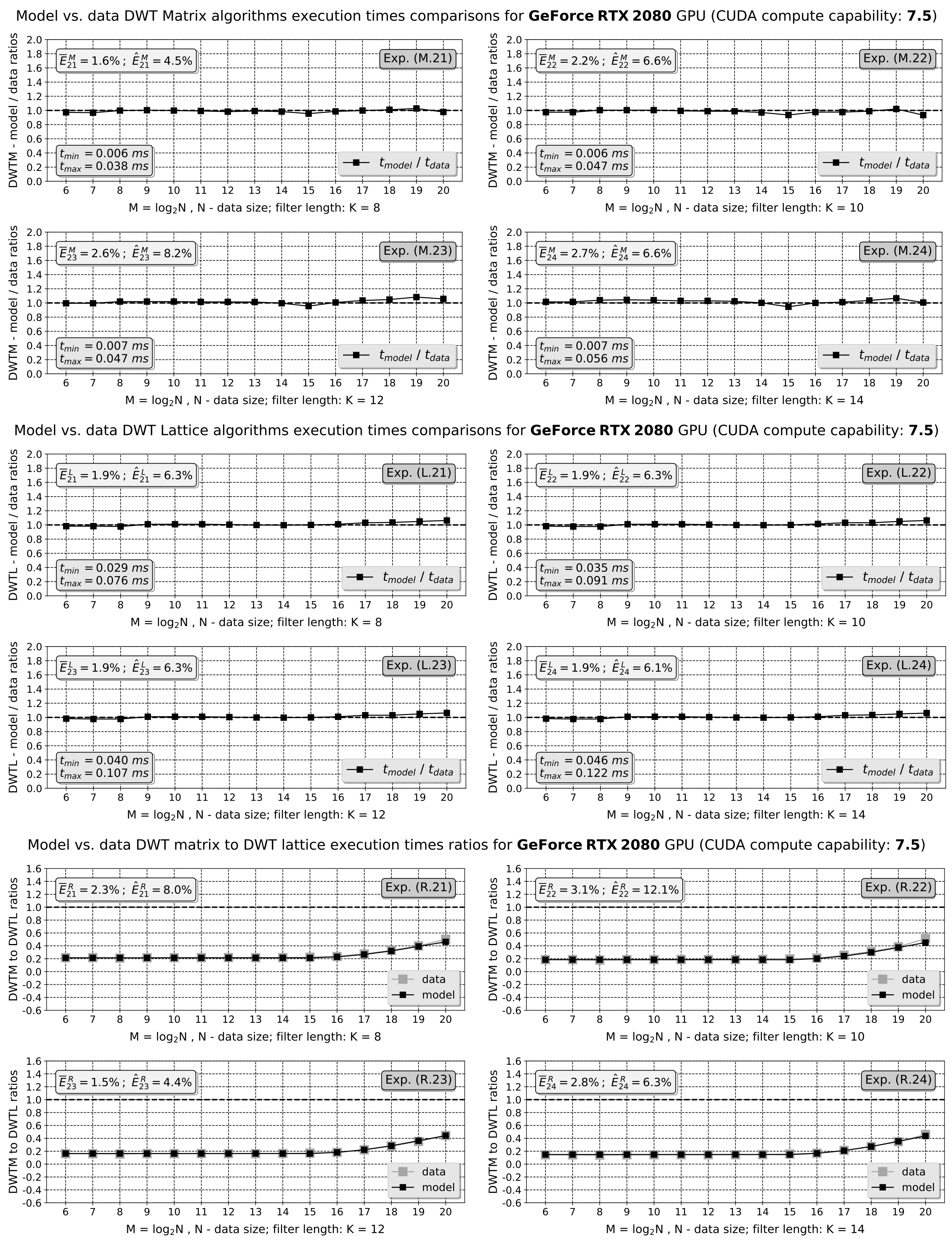

Fig. 15. Experimental results for GeForce RTX 2080 GPU with DWT input data sizes $M = 2^6, \dots, 2^{20}$ and filter lengths $K \in \{8, 10, 12, 14\}$.
Experiments $(\mathtt{M.21})$–$(\mathtt{M.24})$: DWT matrix algorithm modeled to measured execution times ratios.
Experiments $(\mathtt{L.21})$–$(\mathtt{L.24})$: DWT lattice algorithm modeled to measured execution times ratios.
Experiments $(\mathtt{R.21})$–$(\mathtt{R.24})$: DWT matrix to DWT lattice algorithms modeled to measured execution times ratios

prediction accuracy, using the stated below formulas:

$$\overline{E}^R = \frac{1}{N_e}\sum_{i=1}^{N_e}\overline{E}_i^R, \quad \widehat{E}^R = \max_{i=1,\dots,N_e}\widehat{E}_i^R, \tag{22}$$

which in the end happened to take the following values:

$$\overline{E}^R = 3.9\% \quad \text{and} \quad \widehat{E}^R = 20.3\%. \tag{23}$$

In the following section we will discuss the obtained results.

6.1. Results discussion

Graphs `Exp.(M.`i`)` and `Exp.(L.`i`)`, $i = 1,\dots,24$, depicted in Figs. 10–15, indicate that the execution times prediction errors, evaluated by our model, rarely exceed 5% relative difference with regard to their measured counterparts, throughout all of the experiments conducted on all 6 tested GPU devices for both of the considered DCTAs across all the examined input vectors sizes and DWT filter lengths. This is confirmed by the results shown in Table 4, where the overall average model relative inaccuracy has been evaluated to be on the level of 2.8%, with maximum prediction mismatch of 14.5%. Along with the average time, needed by the model to perform a single prediction evaluation, being equal to 3.5 ms, the presented results have to be considered highly satisfactory. In order to put them into broader perspective, let us gather all the major aspects of the efficiency characteristics of GPU execution time prediction models, discussed earlier in Section 2 and collected in Table 1, along with the respective features of the proposed solution, aggregated in Table 4. In Table 5 a suitable summary is presented.

Table 5
The proposed model characteristics comparison

Model type	Average prediction error	Maximum prediction mismatch	Average evaluation slowdown	Algorithm type
Analytical	5%	15%	negligible	DTCA-like
Statistical	9%	40%	negligible	general
ML-based	6%	45%	negligible	general
Simulation-based	1%	5%	10^8	DTCA-like
Hybrid	4%	16%	10^6	general
Proposed	**2.8%**	**14.5%**	$\mathbf{3.2\cdot 10^2}$	DTCA-like

It is worth explaining that the average evaluation slowdown factor, characteristic to our model, was calculated on the basis of the results obtained by the matrix by vector multiplication algorithm, discussed in detail in Section 6. 5.3.

From the characteristics summary shown in Table 5 one can conclude that in terms of the average prediction error and the respective maximum prediction mismatch, our model positions itself close to the simulation-based models, while maintaining prediction evaluation time similar to that, characteristic to the analytical predictors. It also significantly outperforms all considered hybrid models, to which category our model adheres to, but once again we have to emphasize the restrictive character of the model of ours in terms of the algorithm types it is able to process, in comparison to the general character of the considered hybrid models, capable of performing the respective evaluation for a broad class of parallel computation tasks. On the other hand we have to note that the proposed model can be applied to any kinds of GPU implementations of parallel algorithms, as long as they adhere to the assumptions holding for DTCAs, explained in detail in Section 1. The next feature of the proposed model, worth emphasizing, is the ease of practical application, example of which was demonstrated in Section 6. 5.3, which positions it close to simulation-based models and makes it far more facile in use compared to analytical predictors. What is more, for a given DTCA, the model is dependent only two parameters (apart from GPUs architectures specific constants), whose values may be reused for GPU devices of the same architecture (i.e. the same compute capability), what can be observed in Table 3, on the example of the parameters t_p and t_m values for DWT matrix and lattice structure-based methods, in the case of RTX 2060 and RTX 2080 GPU devices. Additionally, from graphs `Exp.(R.`i`)`, $i = 1,\dots,24$, depicted in Figs. 10–15, and relationships present in equation (23), it can be concluded that the model can also be successfully used as a fairly precise comparison tool between different DTCAs parallel implementations across a broad range of GPU architectures. This last feature appears to be particularly valuable in the case of DTCAs, since the mentioned results show that despite theoretical advantage of the lattice-based DWT computation method over its matrix-based counterpart, in all respects discussed in Section 3, such superiority is rarely the case for GPU realizations of the considered algorithms.

An issue worth further discussion is the general effectiveness relationship between the proposed model and the most accurate models class, which are the simulation-based predictors, since such relationship appears to be crucial in the evaluation of the overall fidelity of the proposed solution. Here we have to admit the undoubtable superiority of the simulation-based models in terms of the accuracy characteristics over the proposed prediction method. However, we should also note, what was briefly discussed earlier in Section 1 and can be concluded on the basis average evaluation slowdown factors present in column 4 of Table 5, the vast prediction evaluation speed difference between the analyzed approaches, which in many cases might effectively prohibit practical usage of the simulation-based methods in the process of verification of the time effectiveness validation of certain computational tasks. For example, as reported in [36], the simulation time of typical GPU algorithmic benchmarks extend from 3.8 days, up to even 17 days of the continuous simulation run on the typical consumer segment CPU-based system, for algorithms whose real GPU execution times vary from 0.524 ms up to 0.918 ms, respectively. Similarly, in his work [64] the author reports simulation time of the typical parallel GPU implementation of the 1024×1024 – element matrix by matrix multiplication algorithm realized with the use of the GPGPU-Sim emulator to be about 18 hours of the continuous simulation run on consumer segment CPU-based system. Taking into consideration the fact that employment methodology characteristic to our model and simulation-based approaches are similar in terms of their application difficulty and in light of the aforementioned results, the inaccuracy difference between our solution and simulation-based methods may be considered a relatively fair price in case of algorithms whose effectiveness validation requires the use of extensive data sizes or conducting numerous tests.

At last, it is also worth adding that for relatively moderate data sizes and filter lengths, the tested parallel GPU DWT implementations, which are not in any way even close to optimal, significantly outperform their CPU counterparts in terms of computation times. For example, for computation server system with quite powerful 8-threaded Intel® Xeon Silver 4112 processor with 256GB DDR RAM equipped with GeForce RTX 2080 GPUs, our additional comparisons indicate 20% up to 40% speedup of the overall, single GPU computation time (including all memory transfers) for DWT matrix GPU implementations over their CPU counterparts for data sizes N ranging from 2^{20} up to 2^{23}, respectively. For DWT lattice implementations, the considered speedups are of similar orders.

To sum up our discussion, since the computational structures of the studied parallel DWT algorithms can be considered typical for many other kinds of known DTCAs, such as multiple variants of e.g. DFT, FFT, DCT, FCT, DHT or FHT and others (see [1] or [4]), we can conclude that the obtained accuracy and time effectiveness results, along with those presented earlier in Section 6. 5.3 for the case of parallel implementation of a vector by matrix multiplication algorithm, should also extend, at least, to the mentioned DTCA classes with similar accuracy and time effectiveness characteristics. Such conclusion is also supported by means of the methodology used in many reputed execution time prediction models present in the literature (e.g. [21,22] or [23]), where only typical variants of chosen computational algorithms verified on the reduced pool of GPU devices families are analyzed in the process of validation the mentioned models fidelities. However, we still are aware of the fact that full confirmation of the proposed model general effectiveness might require further research involving other DTCAs classes, what is the scope of the, already undertaken, authors future work.

7. CONCLUSIONS

In this paper the authors presented a novel, hybrid execution time prediction model for parallel GPU realizations of discrete transforms computation algorithms. The model was precisely defined in the form of the set of 3 simple and time-efficient algorithms, allowing for its effective realization. The practical example of its application to a selected matrix by vector multiplication parallel computation method was given in the form of a detailed case study. The model was extensively validated through the experimental research conducted on 6 different GPU devices, covering a broad range of compute capability architectures for 2 exemplary structurally different, but theoretically close in terms of their parallel computational complexities, DWT computation methods. The results have revealed that its overall prediction accuracy of 97.2% positions the model near the simulation-based approaches, while its time-effectiveness is close to that characteristic to analytical solutions (3.5 *ms* on average for single prediction evaluation conducted for the considered DWT computation methods). It has also been proved to be much simpler in practical application than is the case for the last of the mentioned models categories, since the user is only required to supply the model with a properly converted kernel code, with almost no necessity of its prior analysis, and the model itself is dependent on only 2 free parameters, characteristic to a given GPU architecture and a selected computation algorithm, which are easy to determine during measurements. Additionally, it has been stated that the proposed model can be used for prediction of execution times of broader class of parallel computational tasks, as long as they adhere to, not particularly restrictive, structural assumptions holding for discrete transforms computation algorithms. At last, the authors want to point out that the potential extension of the model capabilities, regarding more detailed inclusion of memory access timing effects, shouldn't be excessively problematic. However, it would undoubtedly require a more profound investigation. To sum up, the proposed model appears to be highly precise, time-effective and feasible in practical application, and as such, in the authors' opinion, is a very interesting alternative to the related existing solutions.

REFERENCES

[1] U.N. Ahmed and K.R. Rao, *Orthogonal Transforms for Digital Signal Process*. Secaucus, NJ, USA: Springer-Verlag, New York, Inc., 1974.

[2] Y. Su and Z. Xu, "Parallel implementation of wavelet-based image denoising on programmable pc-grade graphics hardware," *Signal Process.*, vol. 90, pp. 2396–2411, 2010, doi: 10.1016/j.sigpro.2009.06.019.

[3] P. Lipinski and D. Puchala, "Digital image watermarking using fast parametric transforms," *Bull. Pol. Acad. Sci. Tech. Sci.*, vol. 67, pp. 463–477, 2019.

[4] K.R. Rao and P. Yip, *Discrete cosine transform: algorithms, advantages, applications*. San Diego, CA, USA: Academic Press Professional, Inc., 1990.

[5] D. Salomon, *A Guide to Data Compression Methods*. New York: Springer-Verlag.

[6] D. Puchala and M. Yatsymirskyy, "Joint compression and encryption of visual data using orthogonal parametric transforms," *Bull. Pol. Acad. Sci. Tech. Sci.*, vol. 64, pp. 373–382, 2016.

[7] M. Akay, *Time Frequency and Wavelets in Biomedical Signal Process.*, ser. IEEE Press Series in Biomed. Eng. Wiley-IEEE Press, 1998.

[8] S. Babichev, J. Skvor, J. Fiser, and V. Lytvynenko, "Technology of gene expression profiles filtering based on wavelet analysis," *Int. J. Intell. Syst. Appl.*, vol. 10, pp. 1–7, 2018.

[9] Z. Jakovljevic, R. Puzovic, and M. Pajic, "Recognition of planar segments in point cloud based on wavelet transform," *IEEE Trans. Ind. Inf.*, vol. 11, no. 2, pp. 342–352, 2015.

[10] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Indianapolis, IN 46256: John Wiley & Sons, Inc., 2014.

[11] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[12] G. Barlas, *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann Publishers, 2015.

[13] K. Stokfiszewski and K. Wieloch, " Time effectiveness optimization of cross correlation methods for OCR systems with the use of graphics processing units," *J. Appl. Comput. Sci.*, vol. 23, no. 2, pp. 79–100, 2015.

[14] A. Wojciechowski and T. Gałaj, "GPU supported dual quaternions based skinning," in *Computer Game Innovations*. A. Wojciechowski, P. Napieralski (Eds.), Lodz University of Technology Press, 2016, pp. 5–23.

[15] M. Wawrzonowski, D. Szajerman, M. Daszuta, and P. Napieralski, "Mobile devices' GPUs in cloth dynamics simulation," in *Proceedings of the Federated Conference on Computer Science and Information Systems*. M. Ganzha, L. Maciaszek, M. Paprzycki (Eds.), 2017, pp. 1283–1290.

[16] D. Puchala, K. Stokfiszewski, B. Szczepaniak, and M. Yatsymirskyy, "Effectiveness of fast fourier transform implementations on GPU and CPU," *Przegląd Elektrotechniczny*, vol. 92, no. 7, pp. 69–71, 2016.

[17] K. Stokfiszewski, K. Wieloch, and M. Yatsymirskyy, "The fast Fourier transform partitioning scheme for GPU's computation effectiveness improvement," in *Advances in Intelligent Systems and Computing II (CSIT)*, N. Shakhovska and V. Stepashko (Eds.), Springer, Cham, 2017, vol. 689, no. 1, pp. 511–522.

[18] B.H.H. Juurlink and H.A.G. Wijshoff, "A quantitive comparison of parallel computation models," *ACM Trans. Comput. Syst.*, vol. 16, no. 3, pp. 271–318, 1988.

[19] S.G. Akl, *Parallel computation. Models and methods*. Upple Saddle River, NJ: Prentice Hall, 1997.

[20] A. Madougou, S. Varbanescu, C. Laat, and R. van Nieuwpoort, "The landscape of GPGPU performance modeling tools," *Parallel Comput.*, vol. 56, pp. 18–33, 2016.

[21] H. Sunpyo and K. Hyesoon, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *ACM SIGARCH Comput. Architect. News*, vol. 37, pp. 152–163, 2009.

[22] C. Luo and R. Suda, "An execution time prediction analytical model for GPU with instruction-level and thread-level parallelism awareness," *IPSJ SIG Tech. Rep.*, vol. 2011-HPC-130, no. 19, pp. 1–9, 2011.

[23] M. Amaris, D. Cordeiro, A. Goldman, and R.Y. de Camargo, "A simple BSP-based model to predict execution time in GPU applications," in *Proc. IEEE 22nd International Conference on High Performance Computing (HiPC)*, 2015, pp. 285–294.

[24] L. Ma, R.D. Chamberlain, and K. Agrawal, "Performance modeling for highly-threaded many-core GPUs," in *Proc. IEEE 25th International Conference on Applica- tion-Specific Systems, Arch's and Processors*, 2014, pp. 84–91.

[25] K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P.J. Narayanan, and K. Srinathan, "A performance prediction model for the CUDA GPGPU platform," in *Proc. International Conference on High Performance Computing (HiPC)*, 2009, pp. 463–472.

[26] M. Amaris, R.Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram, "A comparison of GPU execution time prediction using machine learning and analytical modeling," in *Proc. 15th IEEE International Symposium on Network Computing and Applications (NCA)*, 2016, pp. 326–333.

[27] A. Karami, S.A. Mirsoleimani, and F. Khunjush, "A statistical performance prediction model for OpenCL kernels on NVIDIA GPUs," in *Proc. 17th CSI Int. Symposium on Computer Architecture & Digital Systems (CADS)*, 2013, pp. 15–22.

[28] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili, "Eiger: A framework for the automated synthesis of statistical performance models," in *Proc. 19th Int. Conference on High Performance Computing*, 2012, pp. 1–6.

[29] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: A statistical approach," in *Proc. 6th IEEE International Conference on Networking, Architecture, and Storage*, 2011, pp. 149–158.

[30] G. Wu, J.L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proc. 21st IEEE Int. Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 564–576.

[31] E. Ipek, B. Supinski, M. Schulz, and S. McKee, "An approach to performance prediction for parallel applications," in *Proc. 11th International Euro-Par Conference on Parallel Processing*, 2005, pp. 196–205.

[32] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *Proc. 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 725–737.

[33] "GPGPU-Sim project." [Online]. Available: http://www.gpgpu-sim.org.

[34] A. Bakhoda, W.L. Fung, H. Wong, and G.L. Yuan, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. ISPASS International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[35] "GPUSimPow – AES LPGPU Group Power Simulation Project." [Online]. Available: https://www.aes.tu-berlin.de/menue/forschung/projekte/gpusimpow_simulator/.

[36] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L.K. John, H. Jin, C. Xu, and J. Wu, "Accelerating GPGPU micro-architecture simulation," *IEEE Trans. Comput.*, vol. 64, no. 11, pp. 3153–3166, 2015.

[37] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," in *Proc. 21st International Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 335–344.

[38] G. Malhotra, S. Goel, and S. Sarangi, "GpuTejas: a parallel simulator for GPU architectures," in *Proc. 21st International Conference on High Performance Computing*, 2014, pp. 1–10.

[39] Y. Arafa, A.A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, "PPT-GPU: Scalable GPU performance modeling," *IEEE Comput. Archit. Lett.*, vol. 18, no. 1, pp. 55–58, 2019.

[40] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate GPU performance estimation through source-level analysis and trace-based simulation," in *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 506–518.

[41] K. Punniyamurthy, B. Boroujerdian, and A. Gerstlauer, "GATSim: Abstract timing simulation of GPUs," in *Proc. Design, Automation & Test, Europe Conf. & Exhibition (DATE)*, 2017, pp. 43–48.

[42] M. Khairy, Z. Shen, T.M. Aamodt, and T.G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proc. 47th IEEE/ACM Int. Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[43] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for GPGPU," in *Proc. IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 351–360.

[44] "GPU Ocelot project: a dynamic compilation framework for GPU computing." [Online]. Available: http://www.gpuocelot.gatech.edu

[45] J. Power, J. Hestness, M.S. Orr, M.D. Hill, and D.A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Comput. Archit. Lett.*, vol. 14, no. 1, pp. 34–36, 2015.

[46] "FusionSim GPU simulator project." [Online]. Available: https://sites.google.com/site/fusionsimulator/

[47] A. Nakonechny and Z. Veres, "The wavelet based trained filter for image interpolation," in *Proc. IEEE 1st International Conference on Data Stream Mining & Processing*, 2016, pp. 218–221.

[48] G. Strang and T. Nguyen, *Wavelets and Filter Banks*. Welleslay, UK: Welleslay-Cambridge Press, 1996.

[49] P. Lipiński and J. Stolarek, "Improving watermark resistance against removal attacks using orthogonal wavelet adaptation," in *Proc. 38th Conference on Current Trends in Theory and Practice of Computer Science*, vol. 7147, 2012, pp. 588–599.

[50] D. Bařina, M. Kula, and P. Zemčík, "Parallel wavelet schemes for images," *J. Real-Time Image Process.*, vol. 16, no. 5, pp. 1365–1381, 2019.

[51] D. Bařina, M. Kula, M. Matýšek, and P. Zemčík, "Accelerating discrete wavelet transforms on GPUs," in *Proc. International Conference on Image Processing (ICIP)*, 2017, pp. 2707–2710.

[52] D. Bařina, M. Kula, M. Matýšek, and P. Zemčík, "Accelerating discrete wavelet transforms on parallel architectures," *J. WSCG*, vol. 25, no. 2, pp. 77–85, 2017.

[53] W. van der Laan, A. Jalba, and J. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 132–146, 2011.

[54] M. Yatsymirskyy, "A novel matrix model of two channel biorthogonal filter banks," *Metody Informatyki Stosowanej*, pp. 205–212, 2011.

[55] M. Yatsymirskyy and K. Stokfiszewski, "Effectiveness of lattice factorization of two-channel orthogonal filter banks," in *Proc. Joint Conference NTAV/SPA*, 2012, pp. 275–279.

[56] M. Yatsymirskyy, "Lattice structures for synthesis and implementation of wavelet transforms," *J. Appl. Comput. Sci.*, vol. 17, no. 1, pp. 133–141, 2009.

[57] J. Stolarek, "Adaptive synthesis of a wavelet transform using fast neural network," *Bull. Pol. Acad. Sci. Tech. Sci.*, vol. 59, pp. 9–13, 2011.

[58] D. Puchala, K. Stokfiszewski, K. Wieloch, and M. Yatsymirskyy, "Comparative study of massively parallel GPU realizations of wavelet transform computation with lattice structure and matrix-based approach," in *Proc. IEEE International Conference on Data Stream Mining & Processing*, 2018, pp. 88–93.

[59] M. Harris, S. Sengupta, and J.D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3, Part VI: GPU Computing*, H. Nguyen, Ed. Addison Wesley, 2007, pp. 851–876.

[60] S. Sengupta, A.E. Lefohn, and J.D. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Proc. Workshop on Edge Computing Using New Commodity Architectures*, 2006, pp. D–26–27.

[61] J. Franco, G. Bernabe, J. Fernandez, and M.E. Acacio, "A parallel implementation of the 2d wavelet transform using CUDA," in *Proc. 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 111–118.

[62] H. Bantikyan, "CUDA based implementation of 2-D discrete Haar wavelet transformation," in *Proc. International Conference Parallel and Distributed Computing Systems*, 2014, pp. 20–26.

[63] M.J. Flynn and S.F. Oberman, *Advanced Computer Arithmetic Design*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[64] Ł. Napierała, "Effectiveness measurements of linear transforms realized on graphics processing units with the use of GPGPUSim emulator" – MSc thesis, Institute of Information Technology, Łódź University of Technology, Poland, 2020.