A generic parallel computational framework of lifting wavelet transform for online engineering surface filtration

Yuanping Xu[a]

Chaolong Zhang[a, b, *]

18668289@qq.com

Zhijie Xu[b]

Jiliu Zhou[c]

Kaiwei Wang[d]

Jian Huang[a]

[a]School of Software Engineering, Chengdu University of Information Technology, No.24 Block 1, Xuefu Road, Chengdu 610225, People's Republic of China

[b]School of Computing & Engineering, University of Huddersfield, Queensgate HD1 3DH, Huddersfield, UK

[c]School of Computer Science, Chengdu University of Information Technology, No.24 Block 1, Xuefu Road, Chengdu 610225, People's Republic of China

[d]School of Optical Science and Engineering, Zhejiang University, Hangzhou 310027, People's Republic of China

*Corresponding author at: School of Software Engineering, Chengdu University of Information Technology, No.24 Block 1, Xuefu Road, Chengdu 610225, People's Republic of China.

Abstract

Nowadays, complex geometrical surface texture measurement and evaluation require advanced filtration techniques. Discrete wavelet transform (DWT), especially the second-generation wavelet (Lifting Wavelet Transform – LWT), is the most adopted one due to its unified and abundant characteristics in measured data processing, geometrical feature extraction, manufacturing process planning, and production monitoring. However, when dealing with varied complex functional surfaces, the computational payload for performing DWT in real-time often becomes a core bottleneck in the context of massive measured data and limited computational capacities. It is a more prominent problem for the areal surface texture filtration by using 2D DWT. To address the issue, this paper presents a generic parallel computational framework for lifting wavelet transform (GPCF-LWT) based on Graphics Process Unit (GPU) clusters and the Compute Unified Device Architecture (CUDA). Due to its cost-effective hardware design and the powerful parallel computing capacity, the proposed framework can support online (or near real-time) engineering surface filtration for micro- and nano-scale surface metrology through exploring a novel parallel method named LBB model, the improved algorithms of lifting scheme and three implementation optimizations on the heterogeneous multi-GPU systems. The innovative approach enables optimizations on individual GPU node through an overarching framework that is capable of data-oriented dynamic load balancing (DLB) driven by a fuzzy neural network (FNN). The paper concludes with a case study on filtering and extracting manufactured surface topographical characteristics from real surfaces. The experimental results have demonstrated substantial improvements on the GPCF-LWT implementation in terms of computational efficiency, operational robustness, and task generalization.

Keywords: Lifting wavelet; LBB; Surface texture Measurement; Multi-GPU; CUDA; FNN

1 Introduction

Engineering workpieces are designed, manufactured and measured to meet tolerance specifications of various geometrical characteristics (e.g., surface texture, size, angle, radius). After a workpiece has been manufactured, it is necessary to verify the derived characterization (tolerance) parameters to ensure it performs its designed functions. Though the designed functions of a workpiece are the comprehensive effect of several characterization parameters, surface texture is often the dominant one. The roughness, waviness and form are three main characteristics on functional surfaces (e.g., roughness may affect the lifetime, efficiency and fuel consumption of a part) [1,2].

Obtaining roughness, waviness and form frequency bands relevant to a surface is normally accomplished by using filtration techniques to extract a surface texture signal. In principle, filtration must be carried out before suitable numerical surface texture parameters can be identified. It means that geometrical characteristic information relating closely with functional surfaces can be extracted precisely from measured datasets by using appropriate filters. This ensures the potential of establishing functional correlations between tolerance specification and verification [1–3].

The rapid progress of filtration techniques has presented metrologists a set of advanced tools, e.g. Gaussian filters, morphological filters and wavelet filters [4–7]. Gaussian filtering is commonly used in surface texture measurements and has been recommended by ISO 16~~,~~610 and ASME B46 standards for establishing a reference surface [8–10]. However, using Gaussian filter to extract surface topographical characteristics must be based on a presupposition that a raw surface texture signal is a combination of a series of harmonic components [10]. Unfortunately, areal engineering surface texture measurement contains not only regular signals but also nonperiodic and random signals, e.g. deep valleys, high peaks and scratches that are generated during various manufacturing processes, so that extraction, filtration and analysis of these signals are indispensable methods for monitoring how process information (e.g. surface texture parameters) relate to variability of the manufacturing processes [2]. Thus, straight applications of Gaussian filter struggle to satisfy the challenging requirements of surface texture measurement and analysis. In contrast, the discrete wavelet transform (DWT) has shown great potential in handling the complex issues efficiently. DWT has a great diversity of mother wavelets (e.g. Haar, Daubechies, CDF biorthogonal, Coiflet) that can be applied according to different application purposes. Moreover, DWT has advantages on multi-resolution analysis and lower computational complexity.

Recently, surface texture measurement has entered the era of miniaturization, e.g. various explorations on micro- or nano-machining and micro-electromechanical systems (MEMS) manufacturing [1]. Hence the increasing demands for efficient handling of big (measurement) data, processing accuracy, and real-time performance have been treated as paramount. As the status quo, the computational performance of DWT is a core bottleneck for handling high-throughput in online (or near real-time) manner [11], especially for two-dimensional DWT (2D DWT) in areal surface texture measurement [12].

To overcome this bottleneck, pure algorithmic acceleration solutions have been hotly investigated in the last decade. Sweldens proposed a new DWT model, it known as lifting wavelet transform (LWT) or second-generation wavelet that improved computational performance and optimized the memory consumption compared to the first-generation wavelet~~–~~–Mallet algorithm [13]. Although promising, lifting scheme still has limited satisfaction performance in accelerating computations of DWT for online applications when facing large-scale data size. Based on that, hardware accelerations have become a research focus. Graphics Processing Units (GPU) enables those solutions and has become the standard accessory for computation-intensive applications. Unfortunately, current lifting scheme struggles on a single GPU for meeting the demands of online process, especially when dealing with large-scale and dynamic datasets to define high precision requirements for surface texture metrology.

To tackle the challenges, this paper presents a generic parallel computational framework for LWT (GPCF-LWT) based on the off-the-shelf consumer-grade CUDA multi-GPU systems. The proposed GPCF-LWT can accelerate computational-intensive wavelets, so that it can support online engineering surface filtration for high precision surface metrology through exploring an innovative computational model (named the LBB model) for improving the usage of GPU shared memory and an improved lifting scheme for reducing the inherent synchronizations in the classic lifting scheme. The GPCF-LWT also contains three implementation optimizations based on the hardware advancements of an individual GPU node and the novel dynamic load balancing (DLB) model by using the improved fuzzy neural network (FNN) for heterogeneous multi-GPUs. Therefore, it supports intelligent wavelet transform selections in accordance with different surface characterization requirements and measurement conditions.

The rest of this paper is organized as follows. Section 2 gives a brief review of the previous related works, which leads to further clarify the contributions of this research. An in-depth analysis of the feasibility of algorithm improvements for adapting parallel computation are discussed in Section 3 following on a brief introduction of the lifting scheme based second-generation wavelet transform. Section 4 presents the GPCF-LWT framework and its three key components. The LBB model for GPCF-LWT has been explored based on a detailed evaluation of the three classic parallel computation models for lifting scheme implementations to solve the problem of GPU shared memory shortage during LWT computation. To increase the parallelizability of LWT, an improved lifting scheme for reducing the inherent synchronizations has been defined, and to further accelerate the LWT computation, three complementary implementation optimizations in accordance with the latest hardware advancements of GPU cards have been discussed. Moreover, to maximize parallelization of GPCF-LWT on a heterogeneous multi-GPU platform, Section 5 introduces a novel dynamic load balancing (DLB) model, in which a more effective FNN has been designed based on the previous work of authors. A case study is demonstrated and evaluated in Section 6 to test and verify the research ideas. Finally, Section 7 concludes this paper and summarizes the future works.

2 Previous related works

Modern GPUs are not only powerful graphics engines, but also highly parallel arithmetic and programmable processors. GPU is a SIMD (Single Instruction Multiple Data) parallel device containing several streaming multiprocessors (SM), and each SM has multiple GPU cores (also named CUDA cores), i.e., stream processors (SP) which are the computational cores for processing multiple threads simultaneously (each GPU core manages a thread). Besides for visualization and rendering, other more general-purpose computations using GPU (so called GPGPU) have also prosperous since 2002 when consumer-grade graphics cards became truly ‘‘programmable’’, ranging from numeric computational operations [14–16] to computer aided design, tolerance and measurement areas. More specifically, GPGPU has been proposed to solve numerous compute-intensive problems which cannot be solved by using CPU based serial processing mode, e.g., Mccool introduced traditional GPU hardware and GPU programming models for signal processing and summarized some classic signal processing applications based on GPU [17]. Su et al.

proposed a GPGPU framework to accelerate 2D Gaussian filtering, and it achieved about 4.8 times speedup without reducing the filtering quality [18]. However, traditional GPGPU programming must be coded with graphical programming interfaces such as OpenGL and DirectX, so heavy manual works were involved in the whole procedure of developing GPGPU programs, e.g., texture mappings, manual and static threading, memory management, and shading program execution. Thus, this obsolete GPGPU programming procedure is very complicated, and it also limited the parallel computing ability of GPU cards.

To alleviate these application difficulties, in 2007, NVIDIA introduced CUDA that is a programing framework for general-purpose computation and provides a scalable and integrated programming model for allocating and organizing processing threads and mapping them into SMs and CUDA cores with dynamical adaption ability for all mainstream GPU architectures [19]. In addition, CUDA maintains a sophisticated memory hierarchy (e.g. registers, local memory, shared memory, global memory and constant memory) in which different memories can be applied according to particular applications and optimization circumstances, and CUDA GPU cards had been optimized with higher memory bandwidth and fast on-chip performance [19–21]. Better yet, it embedded a series of APIs to program directly on GPU instead of transferring application codes to various obscure graphics APIs and the shading language manually. In CUDA programs, any function running on a GPU card is called “kernel”, and launching a kernel will generate multiple threads running on the GPU [19]. CUDA threads are organized into a hierarchy structure, i.e., multiple threads compose a block, and multiple blocks compose a grid such as a 2D matrix (see Fig. 11) [22]. When a kernel running on a GPU, SMs are in charge of creating, managing, scheduling and executing CUDA threads in groups (called warps), and each group handles 32 parallel threads. However, the memory capacity and parallel computing ability of a single GPU are still limited to integrally process a dataset with tens of gigabytes (GB) in size that is the normal size required for current large-scale scientific applications [23]. For example, a measured dataset extracted from the micro-scale surface area of a ceramic femoral head articulate is 49,~~152~~ 152 ~~×~~ × 49,152 in size (i.e. total ~~12~~ 12 GB data with single precision), whereas a single modern GPU GeForce GTX 1080 selected in this study has only 8 GB GPU memory capacity, so this ~~12~~ 12 GB dataset cannot be loaded and processed at a same time. Although the more expensive high-end GPUs such as NVIDIA TITAN Xp (~~12~~ 12 GB memory) and NVIDIA TITAN RTX (~~24~~ 24 GB memory) can store larger datasets, they will still struggle with real-life measurement datasets often coming in its hundreds and more GB of data sizes. Moreover, the GPU cards have fixed CUDA cores (e.g., 2560, 3840 and 4608 for GTX 1080, TITAN Xp and TITAN RTX, respectively), and each one can execute a hardware thread, so threads are also fixed to support certain number of concurrent computations. Consequently, a single GPU must divide, and process massive smaller chunks of a dataset iteratively, which may greatly increase the latency of intensive computations. In conclusion, scientific computations with big data, especially for applications of online and high precision surface texture measurement, demand distributed computations on multi-GPUs. With complementary optimization strategies, multi-GPUs can seamlessly work together to achieve amazing computational performance, but the multi-GPU architecture is more complicated than the single GPU, and it brings time consuming and error prone processes on workload partitions and allocations, their corresponding synchronizations and communications of divided data chunks, recombination of the processed data chunks, and data transfers across PCI-E (Peripheral Component Interconnection - Express) buses. It is worth mentioning that the NVLink connections introduced recently have approximate 10 times higher bandwidth than the PCI-E [24]. However, NVLink supports the high-end GPUs only (e.g., TITAN RTX and TESLA P100), and it is normally equipped only in the supercomputers.

Historically, although plentiful and concrete researches on wavelet-based filtering theories aroused in the last decades, in comparison, their practical implementations on computers achieved limited success due to the key challenge of computational costs for wavelet transforms. In general, the intensive computation of DWT through its inherent multi-level data decomposition and reconstruction will cause drastically reduction of its performance for online applications when facing large data size. With the rapid development of GPU parallelism, several single GPU acceleration solutions have been explored as responses to this challenge. For example, Hopf et al. accelerated 2D DWT on a single GPU card by using OpenGL APIs in 2002, and consequently this resulted in the beginning of GPU based DWT acceleration [25]. This early practice adopted the traditional “one-step” method rather than the separable two steps of 1D DWT, so that this programming model was not flexible and generalizable at all, i.e., each kind of wavelet transform must have a custom implementation. In 2007, Wong et al. optimized Hopf's work by using shading language (named as “JasPer”), and JasPer used shaders to handle multiple data streams in pipelines and had been integrated in JPEG2000 codec [26]. Notably, JasPer decomposes 2D DWT into two steps of 1D DWT, such that it supports dozens of wavelet types and boundary extension methods in convolution stage. It also unified the calculation methods of forward and inverse 2D DWT. For Wong's work, due to the limitations on GPU programmability and hardware structure at the time, the convolution, down and up sampling operations were executed in sequence on fragment processors of an early GPU, and each texture coordinate in texture mapping must be pre-defined in separate CPU programs. These solutions require manual mapping operations on GPU hardware, and task partitioning that decides which part of the work should be conducted on a CPU and which part will be fit for a GPU is indispensable in these practices prior to the emerging of unified GPU languages such as CUDA.

Recently, there has been a growing interest in CUDA based GPGPU computations to parallelize the DWT for scientific and big data applications. Franco et al. parallelized a 2D DWT on the CUDA enabled GPU NVIDIA Tesla C870 for processing images and achieved a satisfactory speedup of 26.81 times faster than OpenMP method with the fastest CPU setting at the time [27]. Song et al. accelerated CDF (9, 7) based SPIHT (set partitioning in hierarchical trees) algorithm for decoding real-time satellite images based on the NVIDIA Tesla C1060, which achieved speedup of about 83 times compared to a single thread CPU implementation [28]. Later, Zhu et al. improved 2D DWT computation on a GPU and verified it with a case study of the ring artifact removal, which achieved a significant performance increase [29]. Compared with previous CUDA enabled GPU implementations of DWT that only parallelized a very specific type of wavelet under a specific circumstance individually, Zhu's work supports different types of wavelets for adapting different application conditions. In addition, the latest constant memory technique has been applied in Zhu's work to optimize the overall GPU memory access. However, the shared memory access and CUDA streams based hide latency mechanism were ignored for further performance optimization in this research [19]. More importantly, the first-generation wavelet theory and its corresponding algorithmic operations applied in the convolution scheme of these researches had increased the computational complexity and imposed a great deal of memory and computational time consumption, e.g., researchers need to consider the troublesome issue of boundary destruction inherent in the Fourier transform [10]. The second-generation wavelet theory adopts the lifting scheme for wavelet decomposition and reconstruction, which preserves all application characteristics of the first-generation wavelet transform while bringing higher performance and lower memory demand due to its in-place algorithm property [13]. These previous CUDA acceleration studies chose the convolution scheme rather than lifting scheme because the execution order of lifting scheme has heavy data dependencies, such that it is impossible to be fully parallelizable. To explore CUDA based LWT implementation, in 2011, Wladimir et al. accelerated the LWT algorithm with three-level decompositions and reconstructions of an ~~1920~~ 1920 ~~×~~ × 1080 image by using CUDA version 2.1 on NVIDIA GeForce 8800 GTX 768 MB, and this practice achieved a speedup of 10 times compared with the corresponding CPU (AMD Athlon ~~64~~ 64 ~~×~~ × 2 Dual Core Processor 5200+) implementation [4]. ~~Recently,~~ Later, Castelar et al. optimized the (5, 3) biorthogonal wavelet transform by using a CUDA in application of the parallel decompression of seismic datasets [5]. Unfortunately, the further parallel optimization of LWT in these studies has not been considered.

Previous works focus on parallelizing data processing through using a single GPU that had witnessed a moderate performance gain across board. However, due to the limitation of data storage format and memory space, as well as the fixed number of CUDA cores available on a single GPU die, previous works are still struggling to fulfill the online data processing requirements from many large-scale computational applications, which is especially problematic for processing large measured surface texture data engaging complex filtrations with various tolerance parameters [1,30]. In comparison, multi-GPU based acceleration solutions can be flexible and extensible for achieving higher performance with relatively low hardware costs. Numerous computational-intensive issues that cannot be resolved by using a single GPU model have been making steady progress in the context of multi-GPUs. For instance, FFT (Fast Fourier Transform) based multi-GPU solution has become the norm in current online large-scale data applications [31,32]. In the meantime, several multi-GPU programming libraries (e.g. MGPU) [33] and multi-GPUs based MapReduce libraries (e.g. GPMR and HadoopCL) [34,35] had been developed by researchers. Moreover, deep learning which is very popular in the current decade also has benefited from the fast development of multi-GPU techniques [36,37]. These pilot multi-GPU studies are based on the assumptions that all GPU nodes equipped in a multi-GPU platform have equal computational capacity. In addition, task-based load balancing schedulers that these approaches relied upon fall short to support applications with huge data throughputs but limited processing function(s) since there are very few “tasks” to schedule. To maximize the data parallelism in a heterogeneous multi-GPU system, this study has also devised an innovative data-oriented dynamic load balancing (DLB) model based on an improved FNN for rapid measured data division and allocation on heterogeneous GPU nodes [38].

In summary, the combination acceleration optimizations merging both the parallelized architecture of multi-GPUs and improvements from the aspects of LWT and software routines seamlessly together (e.g., the algorithm optimization of lifting scheme, comprehensive memory usage optimization based on the advanced hierarchal architecture of CUDA memory, and the latency hide mechanism based on CUDA multi-streams) were largely ignored. To solve these shortcomings, this study explores and implements a generic parallel computational framework for the lifting wavelet transform (GPCF-LWT) based on a heterogeneous CUDA multi-GPU system. It further optimized the acceleration of 2D LWT through the improved processing model of lifting scheme (LBB and improved algorithm of lifting scheme), the devised three implementation optimization strategies and the improved DLB model on a heterogeneous multi-GPU system. Moreover, GPCF-LWT can support the whole kind wavelet transforms, thereby supporting intelligent wavelet transform selections in accordance with different surface texture characterization requirements and measurement strategies.

3 The second-generation wavelet

The fundamental idea behind wavelet analysis is to simplify complex frequency domain analyses into simpler scalar operations. The first-generation wavelet applies fast wavelet transform to enable decomposition (forward DWT) and reconstruction (inverse DWT) in the form of a two-channel filter banks, or the so-called Mallat convolution scheme [39]. Mallat wavelet decomposition and reconstruction convolve the input and output data with the corresponding decomposition and reconstruction filter banks respectively, so it has become a popular choice for many engineering applications in recent years. However, Mallat algorithm has a certain degree of computational complexity (e.g. it has inherent boundary destruction when executing FFT). Thus, Sweldens proposed a new implementation of DWT, i.e. lifting scheme or the second-generation wavelet.

The second-generation wavelet that uses the lifting scheme for wavelet decomposition and reconstruction preserves all properties of the first-generation wavelet, but brings higher computational performance and lower memory demand because of its in-place algorithm attribute [13]. The lifting scheme based 1D forward (i.e. decomposition) DWT (abbreviated as forward LWT or FLWT) contains four operation steps: split, predict, update and scale [4,13,40]:

1) *Split:* This step splits the raw signal into two subset coefficients, i.e. *even* and *odd*, and the former corresponds to the even index values while the latter is the odd index values. The split method is expressed in Eq. (1), and it is also called as “lazy wavelet”.

$$\begin{cases} even[i] = X[2i] \\ odd[i] = X[2i+1] \end{cases} \tag{1}$$

2) *Predict:* For the coefficients of *even* and *odd*, it can predict *odd* coefficients from the *even* by using the prediction operator *PO*, and then replace the old *odd* values by the predicted result as the next new *odd* coefficients, recursively, and this step can be expressed as Eq. (2). The *odd* is the same as the detail coefficients *D* in Mallat algorithm.

$$odd^i = odd^{i-1} - PO\left(even^{i-1}\right) = odd^{i-1} - even^{i-1} * p^i \tag{2}$$

3) *Update:* Similarly, it can update *even* by using the update operator *UO*, and then replace the old *even* values by the updated result as the next new even coefficients, recursively, and this step can be expressed as Eq. (3). The *even* is the same as *A* in Mallat algorithm.

$$even^i = even^{i-1} + UO\left(odd^i\right) = even^{i-1} + odd^i * u^i \tag{3}$$

4) *Scale:* Normalize *even* and *odd* coefficients with factor *K* respectively by using Eq. (4) to get the final approximation coefficients (*evenApp*) and the detail coefficients (*oddDet*).

$$\begin{cases} evenApp = even \times (1/K) \\ oddDet = odd \times K \end{cases} \tag{4}$$

All operators are completed in the in-place computational mode based on these four steps. Fig. 1 illustrates the split operator with in-place mode, in this case, the raw data should be split into *even* and *odd* subsets which are organized and stored in the first half and the second half of the memory space respectively that is the same space for raw data storage. Similarly, prediction, update and scale results are stored in their original memory spaces rather than allocating new memories, such that lifting scheme requires less memory consumption than the classic Mallat algorithm. Furthermore, the 1D inverse lifting scheme (i.e. reconstruction) DWT (abbreviated inverse LWT or ILWT) just inverse the computational sequence of operation steps of FLWT and switch the corresponding plus and subtraction operators:

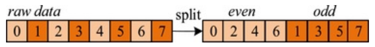


Fig. 1 The computation of split operation with in-place mode.

alt-text: Fig 1

Scale:

$$\begin{cases} even = evenAPP \times K \\ odd = oddDet \times (1/K) \end{cases} \tag{5}$$

Update:

$$even = even - UO(odd) \tag{6}$$

Predict:

$$odd = odd + PO(even) \tag{7}$$

Merge: This step is also called as “inverse lazy wavelet”, and it can be expressed by:

$$\begin{cases} X[2i] = even[i] \\ X[2i+1] = odd[i] \end{cases} \tag{8}$$

The lifting scheme can dynamically update configuration parameters (i.e. iterative number and wavelet filter coefficients) in each computational step of DWT based on factorizations of the polyphase matrixes of wavelet filter coefficients, such that the generalization of this framework can be guaranteed to realize the parallel computations for all kinds of wavelet transforms dynamically. Generally, the factorization of polyphase matrix $P(z)$ for all wavelets can be described as the following:

$$P(z) = \prod_{i=1}^{n} \begin{bmatrix} 1 & u_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ p_i(z) & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \tag{9}$$

where coefficients of $u_i(z)$ are update coefficients in the update step and coefficients of $p_i(z)$ are prediction coefficients in the prediction step in Figs. 2 and 3, and *K* is the scale factor. In a conclusion, the computational results of both forward and inverse LWT for arbitrary wavelet can be obtained by applying several steps of prediction and update operations and imposing the final normalization with K on the outputs of factorization of $P(z)$. Figs. 2 and 3 illustrate the main computational processes of single-level 1D forward and inverse LWT respectively.

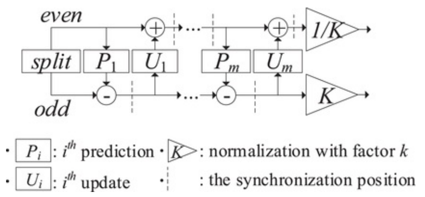

even
split
P_1
U_1
P_m
U_m
$1/K$
K
odd
P_i : i^{th} prediction
K : normalization with factor k
U_i : i^{th} update
: the synchronization position

Fig. 2 The main computational procedure of single-level 1D forward LWT.

alt-text: Fig 2

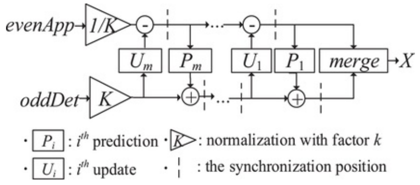

evenApp
$1/K$
U_m
P_m
U_1
P_1
merge
X
oddDet
K
P_i : i^{th} prediction
K : normalization with factor k
U_i : i^{th} update
: the synchronization position

Fig. 3 The main computational procedure of single-level 1D inverse LWT.

alt-text: Fig 3

4 CUDA based GPCF-LWT implementation

Based on the fundamental LWT concepts summarized in Section 3, this study proposes GPCF-LWT by using CUDA enabled GPUs for online and high precision surface measurement applications. This study assumes that factorizations of polyphase matrixes of ~~a~~ the wavelet filter coefficients have been prepared before applying the GPCF-LWT, so that we have got a series of update coefficients $U = \{u_0, u_1, \ldots, u_n\}$, a series of prediction coefficients $P = \{p_0, p_1, \ldots, p_n\}$, and also the scale factor K.

The traditional parallel computational platforms (e.g. OpenMP, MPI and FPGA) have empowered three classic LWT parallel computational models, namely, the RC (Row-Column), LB (Line-Based) and BB (Block-Based) models [41]. Generally speaking, the LB model has the best instruction throughput, followed by the BB and RC models [4]. Unfortunately, except for the shortest wavelets (e.g., the Haar wavelet), both LB and BB models had failed to be transplanted on the single CUDA GPU due to the space limitation of current shared memory, and the generalization of LWT (i.e., supporting all kinds of wavelet transforms) also cannot be satisfied. Although RC model is the only feasible CUDA GPU transplantation that supports the generalization of LWT, the processing time of CUDA based RC model is approximately 4 times slower than the Mallat algorithm solution running on the same GPU card [42]. Thus, based on the in-depth study of these three models, this study devised a generic LWT parallel computational model for the CUDA enabled GPU architecture.

4.1 The improved parallel computational model

- *The LB model.* On traditional parallel computational platforms (e.g., OpenMP, MPI and FPGA), the overall framework of LB based LWT is depicted in Fig. 4. The LB model divides a raw dataset (e.g., 1024×1024 in size) into N (N equals to the number of computational nodes) smaller subsets (e.g., if N is 4, each subset has 256 rows and 1024 columns), and each subset will be allocated on a computational node where LB performs the following operations:

1) Lunching m threads, and each one performs horizontal 1D LWT for each row of the target subset concurrently (i.e., each thread processes a row of the target subset.) and then the LB caches the temporary results in the local memory (e.g., a RAM—random access memory);

2) Once the m rows are completely processed, the m threads perform vertical 1D LWT on m columns of temporary results concurrently and repeatedly until all columns of the cached temporary results have been processed. In this step, the iterative times are $(width + m-1) / m$, and each iteration processes m columns.

3) Steps 1 and 2 maintain an integrated iteration that will be repeated until all rows and columns of the target subset are processed.

The last step of LB based LWT is that one node (usually called a master node) merges the results of *N* subsets and then obtains the final result. The minimum value of *m* is determined by a particular type of wavelet, e.g., 3 for Haar and 6 for Deslauriers-Dubuc (13, 7). Generally, *m* can be unified as a fixed value (e.g., 10 or 20) that larger than all types of wavelets needed. Taking $m = 10$ for a float dataset having1024 columns as an example, each computational node requires 40 KB local memory space, so LB model can be easily implemented on OpenMP, MPI and FPGA since they have large RAM spaces. However, a CUDA block has only 16 KB local memory (shared memory) space that cannot satisfy the memory demand of LB model except for the shortest wavelet (e.g., Haar wavelet requires only 12 KB shared memory space). In addition, columns of a dataset cannot be divided by LB, so it is a coarse-grained division approach, which cannot operate well with the SIMD of CUDA.

- *The BB model.* Unlike LB, the BB model divides a raw dataset into *N* subsets (data blocks) from both row and column directions, see Fig. 5. Each subset is allocated on a computational node where RC routine performs horizontal 1D LWT for all rows concurrently at first, and then vertical 1D LWT for all columns will be computed concurrently. Like LB model, the master node will merge all results of subsets at the last step of BB model. The BB model is also a coarse-grained division approach, and it also requires a great deal of local memory space, e.g., if a raw float dataset with 1024×1024 in size is divided into four subsets (each subset has 512 rows and 512 columns), each computational node requires 1024 KB local memory space that considerably exceed the size of a shared memory space of a CUDA GPU, such that classic RC model based CUDA solutions must use a GPU global memory space as the temporary caches, and it has become the key bottleneck for data access.
- *The innovative LBB model for CUDA LWT.* To solve the problem of shared memory shortage on CUDA GPUs, this study devised an innovative parallel computational model for 2D LWT-LBB by combining both LB and BB. The ~~main~~ overall framework of LBB model is shown as in Fig. 6. The LBB model divides a raw dataset into several subsets from the column direction, and each subset can be assigned with a CUDA thread block (e.g., 8 rows and 32 columns in size) where the in-built threads perform the following operations:

1) Each group of concurrent row threads loads a row of a dataset into the shared memory of a GPU in parallel. Accesses of a row data can be coalesced since all threads in a block access the continuous relative addresses [19], so it is very efficient to load a dataset into a shared memory space.

2) Each group of concurrent row threads in a block performs horizontal 1D LWT on a data row in parallel, and then the GPU caches the temporary results in its shared memory space.

3) *m* groups of row threads get *m* row temporary results in parallel by executing steps 1 and 2 (*m* is a preset value, and it is 8 here).

4) Each group of concurrent column threads in a block performs vertical 1D LWT on a data column of the cached temporary results in parallel, and *width/n* groups of column threads process *width/n* columns of cached temporary results in parallel (*width/n* is the width of a subset).

5) The corresponding column thread groups save the results into a GPU global memory space and delete the *m* row temporary results in the shared memory space got by step 3.

6) Steps 1–5 maintain a complete iteration that will be repeated until all rows and columns of a dataset are processed.

The two differences of data division methods for BB and LBB are: 1) In order to adapt the advanced architecture of CUDA threads, the height of a subset in LBB is larger than BB whereas the width for LBB is smaller than BB, and the number of subsets of LBB is far larger than BB as the number of thread blocks is far more than computational nodes in those traditional parallel platforms. Steps 1–6 of LBB also integrated the LB idea with the one major update: Instead of providing a thread for a row, LBB constructs the thread block structure based on numerous CUDA threads (each thread block organize *m* groups of concurrent row threads, and each group of row thread has *width/n* CUDA threads, e.g. 32 CUDA threads in each row), such that a thread block can perform both horizontal 1D LWT on multiple rows by using groups of row threads and vertical 1D LWT on multiple columns by using groups of column threads in parallel.

Although LBB divides a large dataset into several smaller subsets, the problem of shared memory shortage is still notable, e.g., it requires 64 KB shared memory space if a subset has size 128×128. Thus, the sliding window structure has been designed in LBB, see Fig. 6. In LBB, the size of a sliding window is equal to a constructed thread block (e.g. 8×32 for a subset 128×128), i.e., a sliding window is built on a thread block, and each subset has a sliding window. Once a sliding window is full (e.g., there are 8 rows of temporary results cached in a shared memory space), column-based vertical 1D LWT will be carried out (i.e. steps 4 and 5), and then LBB moves down a sliding window on the target subset to compute and cache other data points (or pixels) of other thread blocks concurrently by moving the ~~th~~row block to process the next 8 rows (starting from Step 1).

- *Performance analysis.* Taking the CDF (9, 7) wavelet as an example to evaluate the CGMA (Compute to Global Memory Access) (i.e., an index to estimate the radio of instruction throughput and global memory accesses for performance analysis [22]) with LBB, it requires 18 times of computational operations for performing 2D DWT on a data point, including 16 times of multiplications and 2 times of additions, and 4 times of global memory accesses (2 times of read and write respectively), by assuming that the size of a slide window is 8×32 and each thread processes a data point in parallel, such that the CGMA is:

$$CGMA = C/M = ((16 + 2) \times 2) / (2 + 2) = 9 \qquad \textbf{(10)}$$

Compared with the RC model (CGMA is 0.9), LBB has been increased by 10 times.

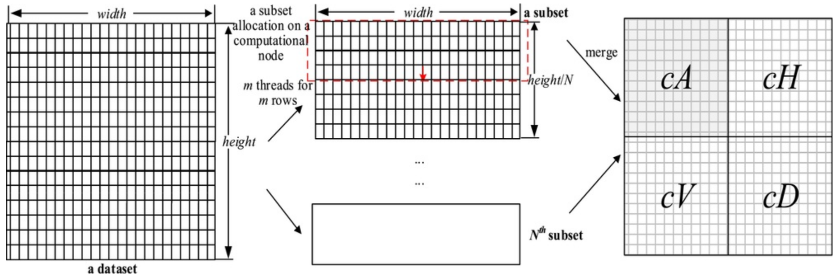


Fig. 4 The overall framework of LB model.

alt-text: Fig 4

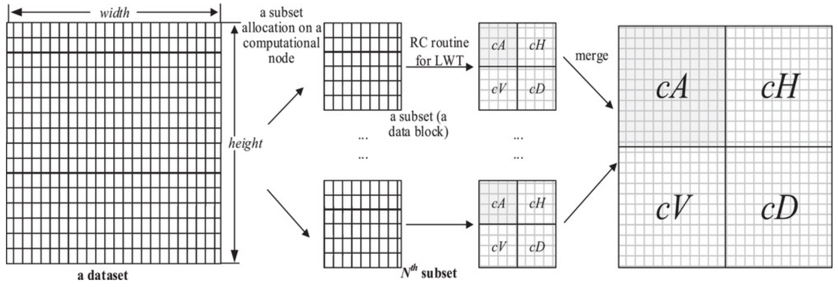


Fig. 5 The overall framework of BB model.

alt-text: Fig 5

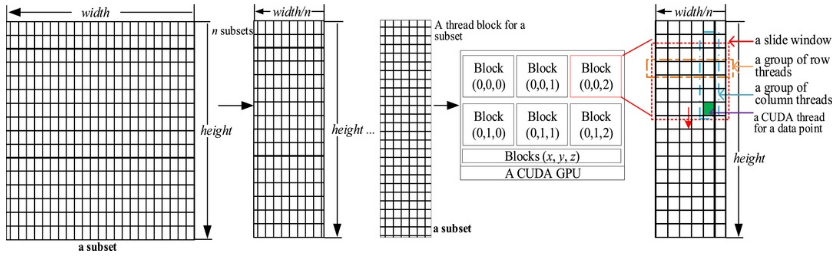


Fig. 6 The overall framework of LBB model.

alt-text: Fig 6

4.2 The improved lifting scheme

In a GPU routine, the LWT requires lots of synchronizations as every implementation procedure of the lifting scheme has different functions computing in sequence, and it decreases the parallelizability of LWT computations. To solve this problem, this study devised an improved approach to decrease the inherent synchronizations of lifting scheme. The factorization of polyphase matrix $P(z)$ (see ~~Eq. (11)~~Eq. (9)) can be converted into the following form:

$$P(z) = \prod_{i=0}^{m} \begin{bmatrix} 1 & u_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ p_i(z) & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix}$$
$$= \begin{bmatrix} 1 & u_0(z) \\ 0 & 1 \end{bmatrix} \prod_{i=1}^{n} \begin{bmatrix} 1 & 0 \\ p_i(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & u_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \quad \textbf{(11)}$$

where $n = m$ if $u_0(z) = 0$ and $n = m$-1 if $u_0(z) \neq 0$. According to ~~Eq. (13)~~Eq. (11), the optimized operation steps of the lifting scheme based 1D forward LWT are shown as the follows:

(1) *Split:*

$$\begin{cases} even^0[l] = x[2l] \\ odd^0[l] = x[2l+1] \end{cases} \tag{12}$$

(2) *Preprocessing:*

$$\begin{cases} s^0 = even^0 - u_0 \times odd \\ d^0 = odd^0 \end{cases} \tag{13}$$

(3) *Lifting (predict and update):* for $i = 1, \ldots, n$, then:

$$\begin{cases} d^i = d^{i-1} - p^i \times s^{i-1} \\ s^i = s^{i-1} - u^i \times d^i \end{cases} \tag{14}$$

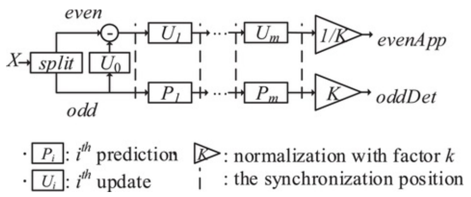
(4) *Scale:*

$$\begin{cases} evenApp = s^n / K \\ oddDet = Kd^n \end{cases} \tag{15}$$

The step 3 is performed iteratively. However, synchronization operations still exist in step 3 since the computation of s^i can only begin after the d^i computation due to s^i relies on d^i. Thus, ~~Eq. (16)~~Eq. (14) should be modified as the follow~~s~~ing:

$$\begin{cases} d^i = d^{i-1} - p^i \times s^{i-1} \\ s^i = s^{i-1} - u^i \times d^i \end{cases} \Rightarrow \begin{cases} d^i = d^{i-1} - p^i \times s^{i-1} \\ s^i = s^{i-1} - u^i \times \left(d^{i-1} - p^i \times s^{i-1}\right) \end{cases} \tag{16}$$

It can be seen from ~~Eq. (18)~~Eq. (16), the computation of s^i no longer relies on d^i, and s^i and d^i only relay on s^{i-1} and d^{i-1} respectively, so that the corresponding computational operations can be performed concurrently to decrease half of the synchronizations required in lifting scheme. The main computational procedure of single-level 1D forward LWT is demonstrated in Fig. 7 that is optimized from Fig. 2. The inverse LWT has the similar optimization approach.



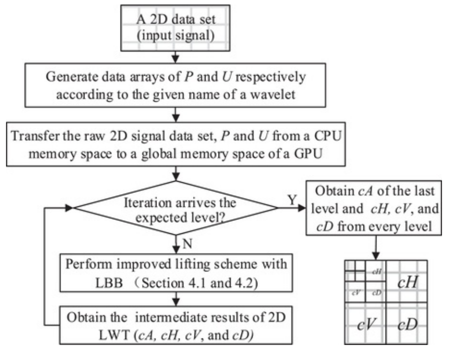
even
X
split
U_0
U_1
U_m
1/K
evenApp
P_1
P_m
K
oddDet
odd
P_i: i^{th} prediction
U_i: i^{th} update
K: normalization with factor k
: the synchronization position

Fig. 7 The improved lifting scheme for single-level 1D forward LWT.

alt-text: Fig 7

4.3 The construction of the single GPU-based GPCF-LWT

Before realizing parallel computations of ~~the single-level 1D~~ LWT by using the GPCF-LWT, a series of prediction coefficients $P = \{p_0, p_1, \ldots, p_n\}$ and update coefficients $U = \{u_0, u_1, \ldots, u_n\}$ must be generated successively according to the given name of a wavelet in order to ensure the generalization of the GPCF-LWT. P and U are 2D arrays with the same length n, and $P[i]$ and $U[i]$ correspond to prediction and update coefficients for the ith predication and update operations respectively. Within the GPCF-LWT, P and U serve as input parameters reflecting different wavelet types, such that GPCF-LWT can support the whole family of LWT. Based on LBB and the improved lifting scheme, Fig. 8 illustrates the main computational procedure of multi-level 2D forward LWT proposed in this study. It needs to generate a series of P (a 2D array) and U (a 2D array) according to the given name of a wavelet, and then copy these prediction and update coefficients together with the raw 2D signal data from a CPU memory space to a GPU global memory space, and a specific designed CUDA stream overlapping strategy for increasing the concurrency of this step is presented in Section 4.4. In Fig. 8, the output results *cA, cH, cV*, and *cD* correspond to the approximation coefficients and detail coefficients along horizontal, vertical, and diagonal orientations respectively. Multi-level 2D LWT applies the approximation coefficients *cA* as a 2D ~~raw~~ input signal for the next computational level, and then a new 2D forward LWT iteration with the same computational steps and computational parameters as its previous level will be started on this level. In this way, this procedure will be executed recursively until the expected level has reached, and the final results are the *cA* of the last level and all detail coefficients *cH, cV*, and *cD* collecting from each computational level. The computational framework of LBB based inverse LWT can be constructed in the similar way of Fig. 8.

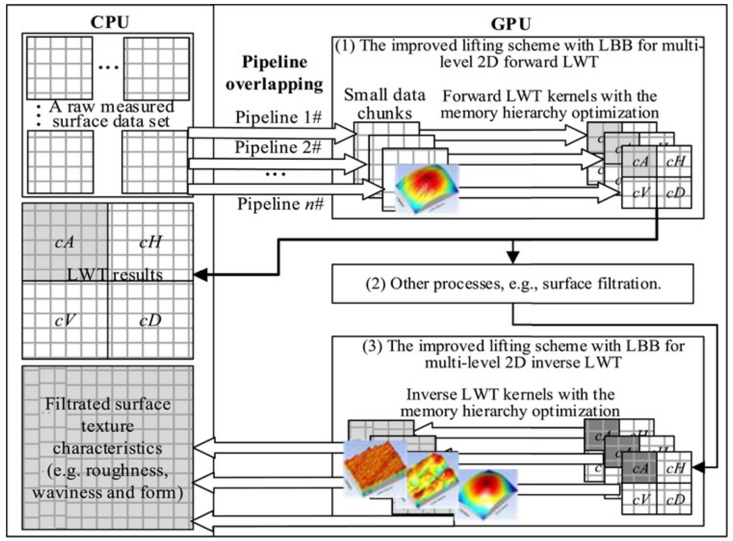


A 2D data set (input signal)
Generate data arrays of P and U respectively according to the given name of a wavelet
Transfer the raw 2D signal data set, P and U from a CPU memory space to a global memory space of a GPU
Iteration arrives the expected level?
Y
N
Obtain cA of the last level and cH, cV, and cD from every level
Perform improved lifting scheme with LBB (Section 4.1 and 4.2)
Obtain the intermediate results of 2D LWT (cA, cH, cV, and cD)
cH
cV
cD

Fig. 8 The main computational procedure of improved lifting scheme with LBB for multi-level 2D forward LWT.

alt-text: Fig 8

In this case, a raw surface measured dataset will be divided into several small data chunks which will be further parallelized by applying the pipeline overlapping strategy (see Section 4.4). A host function (i.e., the forward and inverse scheduler) executing on a CPU is responsible for launching the *Predict*() and *Update*() kernels on a GPU iteratively according to the series of prediction and update coefficients. Fig. 9 shows that a 2D LWT on GPCF-LWT has been realized through separating it into two 1D LWT from the row and column orientations respectively, and multi-level LWT has been implemented by applying the single~~-~~level LWT recursively. Once the final results are obtained, inverse LWT can be performed to reconstruct the filtrated engineering surfaces (e.g. roughness and waviness).



CPU
GPU
A raw measured surface data set
Pipeline overlapping
Pipeline 1#
Pipeline 2#
Pipeline n#
(1) The improved lifting scheme with LBB for multi-level 2D forward LWT
Small data chunks
Forward LWT kernels with the memory hierarchy optimization
cA
cH
cV
cD
LWT results
(2) Other processes, e.g., surface filtration.
(3) The improved lifting scheme with LBB for multi-level 2D inverse LWT
Inverse LWT kernels with the memory hierarchy optimization
Filtrated surface texture characteristics (e.g. roughness, waviness and form)

Fig. 9 An implementation overview of the GPCF-LWT on a single GPU card.

alt-text: Fig 9

GPCF-LWT supports the whole family of wavelet transforms based on LBB and the improved lifting scheme. This paper presents two examples (Haar and CDF (9, 7) wavelets) to show the realization of LWT with any specific wavelet by using GPCF-LWT generically. ~~One of~~The factorizations of polyphase matrixes of wavelet filter coefficients conforming to the improved lifting scheme can be written as the ~~Eqs. (19)~~Eqs. (17) and ~~(20)~~(18) for Haar and CDF (9, 7) respectively.

$$P(z) = \begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix}\begin{bmatrix}1 & -1/2\\ 0 & 1\end{bmatrix}\begin{bmatrix}\sqrt{2} & 0\\ 0 & 1/\sqrt{2}\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix}\begin{bmatrix}1 & -1/2\\ 0 & 1\end{bmatrix}\begin{bmatrix}\sqrt{2} & 0\\ 0 & 1/\sqrt{2}\end{bmatrix}, \tag{17}$$

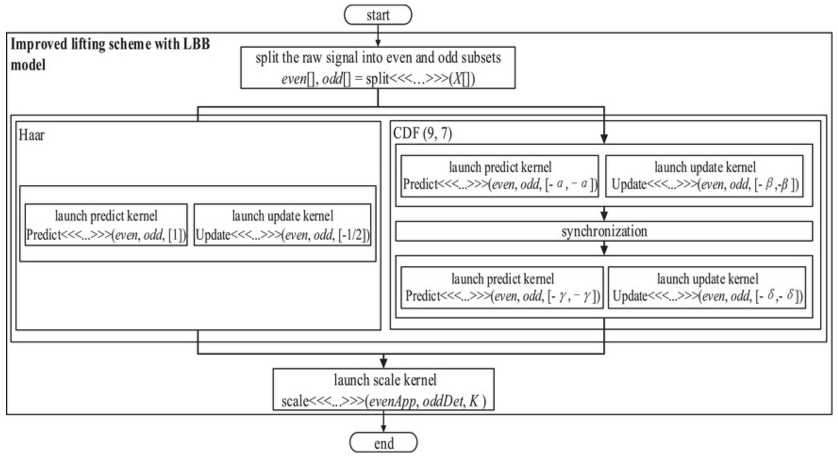
$$\begin{aligned}P(z) &= \begin{bmatrix}1 & 0\\ -\alpha(1+z) & 1\end{bmatrix}\begin{bmatrix}1 & -\beta\left(1+z^{-1}\right)\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ -\gamma(1+z) & 1\end{bmatrix}\begin{bmatrix}1 & -\delta\left(1+z^{1}\right)\\ 0 & 1\end{bmatrix}\begin{bmatrix}1/\varepsilon & 0\\ 0 & \varepsilon\end{bmatrix}\\ &= \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ -\alpha(1+z) & 1\end{bmatrix}\begin{bmatrix}1 & -\beta\left(1+z^{-1}\right)\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ -\gamma(1+z) & 1\end{bmatrix}\begin{bmatrix}1 & -\delta\left(1+z^{1}\right)\\ 0 & 1\end{bmatrix}\begin{bmatrix}1/\varepsilon & 0\\ 0 & \varepsilon\end{bmatrix},\end{aligned}\quad \begin{cases}\alpha = 1.5861343420\\ \beta = -0.0529801185\\ \gamma = 0.8828110755\\ \delta = 0.4435068520\\ \varepsilon = 1.1496043988\end{cases} \tag{18}$$

According to ~~Eqs. (19)~~Eqs. (17) and ~~(20)~~(18), *P*, *U* and *K* for Haar and CDF (9, 7) respectively can be acquired from the following ~~Eqs. (21)~~Eqs. (19) and ~~(22)~~(20).

The coefficients U_0 of both Haar and CDF (9, 7) are zero, so the optimized computational procedure of single-level 1D forward LWT can be devised as ~~Fig. 13~~ Fig. 10 where a host function is responsible for scheduling the executions of predict and update kernels and providing different configuration and computational parameters for different types of wavelets, and the corresponding predict and update kernels are executed on a GPU card in parallel. The left side of Fig. 10 illustrates the computational procedure of Haar wavelet transform, and CDF (9, 7) wavelet is illustrated on the right side. The inverse LWT implementations of Haar and CDF (9, 7) are similar as the forward implementations, which just inverse the corresponding sequences of operation steps of forward computations and switch the corresponding plus and subtraction operators.

$$\text{Harr}: \begin{cases}U_0 = 0\\ P_1 = \{1\}\\ U_1 = \{-1/2\}\\ K = 1/\sqrt{2}\end{cases}, \tag{19}$$

$$\text{CDF }(9,7): \begin{cases}U_0 = 0\\ P_1 = \{-\alpha, -\alpha\}\\ U_1 = \{-\beta, -\beta\}\\ P_1 = \{-\gamma, -\gamma\}\\ U_2 = \{-\delta, -\delta\}\\ K = \varepsilon\end{cases} \tag{20}$$



start
Improved lifting scheme with LBB model
split the raw signal into even and odd subsets
even[], odd[] = split<<<...>>>(X[])
Haar
launch predict kernel
Predict<<<...>>>(even, odd, [1])
launch update kernel
Update<<<...>>>(even, odd, [-1/2])
CDF (9, 7)
launch predict kernel
Predict<<<...>>>(even, odd, [- α, - α])
launch update kernel
Update<<<...>>>(even, odd, [- β ,-β])
synchronization
launch predict kernel
Predict<<<...>>>(even, odd, [- γ, - γ])
launch update kernel
Update<<<...>>>(even, odd, [- δ ,- δ])
launch scale kernel
scale<<<...>>>(evenApp, oddDet, K)
end

Fig. 10 The scheduling algorithm of a single-level 1D forward LWT.

alt-text: Fig 10

4.4 Three optimization strategies for GPCF-LWT implementation on a single GPU card

The GPCF-LWT further explores three implementation optimizations on a single GPU by seamlessly integrat~~e~~ing the GPCF-LWT implementation with the latest CUDA GPU hardware advancements to maximize the parallelization of generic 2D LWT computations:

1) *GPCF-LWT provides an optimization strategy that enable dynamic configurations of execution parameters of kernels based on the CUDA thread hierarchy.* In CUDA programs, a thread block must be loaded into a SM inseparably, and threads in a block will be further encapsulated into several warps. A SM executes all its warps (e.g., 4 warps in Fig. 11) conforming to SIMD (Single Instruction, Multiple Data) model, such that all threads in a warp share same program instructions but to process different data at the same time. Another inactive warp will be activated from the inactive state when an active warp in the same SM is pending (e.g., a warp will be pending when it needs to access data from the global memory as it generally requires 400–600 cycles) in order to hide memory latency [20,21]. Thus, it is of vital importance to configure the number of CUDA threads in a block and organize them in an appropriate hierarchy structure (i.e. setup applicable block and grid sizes) for each kernel, such that SMs can have just enough warps to schedule while limiting the complexity of thread management. Ideally, the number of threads in a block must be an integral multiple of 32 to match the fact that each warp has 32 threads for 32 CUDA cores in CUDA GPUs, so each SM can be possibly arranged with just enough number of warps. In practice, SM may reduce the number of blocks during run-time when too many CUDA threads are organized in a block and each thread occupies too many registers or a large shared memory space, and it often leads to the vital problem that there are not enough warps to be scheduled. With these considerations, this study had tested and evaluated the underlying relationship between the different number of threads in a block and the hardware consumption of GPCF-LWT, which found that the occupancy of CUDA cores in a SM can come up to 100% when we configure 256 threads per block in the GPCF-LWT. Thus, the block number (grid size, *bn*) of LWT is $bn = (sn+256-1)/256$ where *sn* is the length of input data, such that the *bn* with each block having 256 threads can keep all SMs of a GPU card busy all the time, and hide the delay caused by data transfers. To test and evaluate the validity and practicality of this implementation, this study has compared the effects of different number of threads in a block to find the inherent correlation between the number of configured threads in a block and the processing time. Fig. 12 lists the processing time for different numbers of threads in a block for 4-level 2D LWT with data size of 4096×4096 running on a single GTX 1080 GPU card (the hardware specification can be seen in Table 3). Fig. 12 shows that GPCF-LWT can gain the minimum processing time when the number of threads in a block is set to 256.

2) *GPCF-LWT has developed a hierarchical memory usage scheme for generic LWT computations based on the CUDA GPU memory hierarchy.* CUDA adopts a SPMD (Single Program, Multiple Data) programming model and provides a sophisticated memory hierarchy (i.e. register, local memory, shared memory, global memory, texture memory and constant memory, etc.). Hence, a GPU can achieve high data accesses through elaborately designed CUDA codes empowered by the efficient usages of different memories according to the respective data features, including access mode, size and format. GPCF-LWT calls the CUDA API function *cudaHostAlloc*() rather than *malloc*() to allocate a CPU host memory in page-locked memory or pinned memory mode that can support DMA (Direct Memory Access) without requiring to allocate additional buffers for caching data. Compared with page-able memory mode, the page-locked mode transfers data directly from a CPU host memory space to a GPU global memory space, so the write speed of it is 1.4 times faster and read speed is 1.8 times faster than page-able mode [21]. However, the pinned memory should not be overused for its scarcity that leads to reduced performance. With this in mind, this study pre-allocates a fixed size pinned memory in the initialization stage and reuse it during the whole computational life-cycle, such that it maintains high memory performance throughout. Moreover, GPCF-LWT uses the shared memory of a GPU for executing kernels, and instead of direct data accesses from a GPU global memory, a single data point will be loaded into a shared memory space of a SM once and only once from the global memory, and then any loaded data point can be accessed by several CUDA threads simultaneously in a very fast speed (generally it only requires 1–32 cycles). In comparison, data access speed of a global memory will be dramatically reduced when several threads access the same data point in a global memory space repeatedly, and registers were also not applied here for the similar reason [20]. Another major reason for using the shared memory is that row addresses in a global memory are aligned, such that the operation that 32 CUDA threads in a warp loading data continuously from the global memory can be coalesced, i.e., loading 32 data points into a warp needs the global memory access only once rather than 32 accesses in sequence, so this coalescing can enhance the data transfer bandwidth. Another important issue is that where to store and access *P* and *U*, the constant memory or the global memory. Although both the constant memory and global memory are in the same physical location of the GPU hardware architecture, the time consumption for accessing a data point from a constant memory can be much faster than a global memory due to its inherent cache and broadcast mechanisms. For instance, a data point can be loaded from the cache of a constant memory when it is a cache hit, and it just costs one cycle in general. Broadcast is another very useful tool to reduce data access delay, which means that a data point loaded by a thread will be broadcasted to all the other threads of a wrap. Thus, accessing a data point from a constant memory can be as fast as a register when all threads of a warp access the same data point [21,43]. In this study, all CUDA threads use the same values of *P* and *U*, and they are constant and will not be changed during performing *PO* and *UO*, such that the fast data accesses can be achieved by storing and accessing *P* and *U* sets in a constant memory space. All in all, the memory hierarchy of GPCF-LWT for generic LWT computations can be organized as Fig. 13.

To test and evaluate the validity and practicality of this hierarchical memory usage scheme of GPCF-LWT, this study analyzed the performance factors influencing the 2D LWT computation on different memory spaces. Firstly, this experiment tested and compared the bandwidth by using the pinned and page-able memory mode respectively by transferring 1GB measured data from both H2D (from a CPU host memory space to a GPU global memory space) and D2H (from a GPU global memory space to a CPU host memory space) directions. In Fig. 14, the bandwidth of pinned memory mode gains approximately 55% improvement compared with the page-able memory mode. Then, the experiment tested and compared the computational performances between the proposed hierarchical memory usage scheme and the pure global memory usage. In the pure global memory solution, all data (raw measured datasets, *P*, *U*, intermediate results and the filtration output results) are stored in the global memory. In contrast, the proposed scheme stores different datasets in different memory spaces, i.e., it applies the global memory space to store the raw measured datasets and the filtration results, and the shared memory space to cache subsets and the temporary results, which put *P* and *U* coefficient instances in the constant memory space. Fig. 15 illustrates the processing times of these two solutions, and it can be seen from the experimental results that the proposed scheme gains about 50% computational performance enhancement.

3) *GPCF-LWT enables a pipeline overlapping strategy by using CUDA streams.* CUDA GPU provides a non-blocking asynchronous execution mode for GPU kernels, such that the host codes on a CPU and CUDA codes on a GPU can be executed concurrently. Based on this consideration, this study derives a CUDA stream overlapping strategy that can effectively alleviate the delay imposed by scheduling the sequential order of execution of the lifting scheme. Each pipeline manages a CUDA stream, and a pipeline can organize *n* kernels running on it in different time slices, i.e., a pipeline will active and run another kernel automatically when a kernel is pending [19]. Pure CUDA programs based on the non-overlapping model are variants of the following fundamental three stages: (1) transferring a raw dataset from a CPU host memory space to a GPU global memory space through PCI-E buses (i.e., H2D data transfers); (2) processing the raw dataset via a specific algorithm—GPU kernel executions (e.g., 2D LWT); (3) transferring output results from the GPU global memory space to the CPU host memory space (D2H data transfers). Obviously, both stages of data transfers (i.e. H2D and D2H) and kernel executions for data processing are time-consuming. More significantly, these three time-consuming stages are performed in sequence in the non-overlapping mode, and the sequential timelines for three pipeline stages are illustrated in Fig. 16. Thus, data transfers in Fig. 16 are blocking transfers and the host function running on a CPU can only launch GPU kernels after completion of the corresponding data transfers, so that both the PCI-E buses and GPU waste a lot of time in idle state during the entire workflow, which sharply decreases the processing performance of CUDA programs. To reduce these idle time slices, GPCF-LWT overlaps the three major time-consuming stages by using CUDA streams in the pipelines. In accordance to the hardware development that both GPU and PCI-E support bidirectional data transfers, each pipeline in GPCF-LWT handles a repeated process of transferring a small data chunk to a GPU memory first and then launching LWT kernels to process this data chunk until the whole dataset is processed. Fig. 17 shows the timelines of two data transfer stages and LWT kernel execution stage with the overlapping model, in which LWT kernels can be performed during stages of data transfers (i.e., the data transfers are hided), such that a GPU can always stay in the busy state during the entire data processing workflow to gain higher performance than the non-overlapping model. This strategy can be extended to hide data transfer delays in multi-GPU systems gracefully. The CUDA API *cudaMemcpyAsync*() had been applied in this study to asynchronously transfer data from a CPU host memory to a GPU global memory in order to realize the overlapping between data transfers and kernel executions. The pseudocodes of the devised overlapping strategy are listed in Algorithm 1, and the LBB based 2D LWT kernel is listed in Algorithm 2. Moreover, this aggregated strategy still applies on the systems using NVLink technology with two features: (1) both H2D and D2H data transfers are efficient, so stage 1 and 3 consume far less time; (2) these two stages can transfer a relative large data chunk each time, such that stage 2 maintains high concurrency. With these new features, the GPCF-LWT can achieve better performance without extra effort.

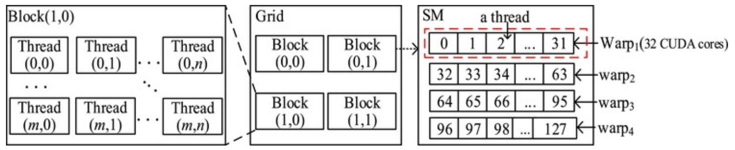


Fig. 11 The hierarchy structure of CUDA threads.

alt-text: Fig 11

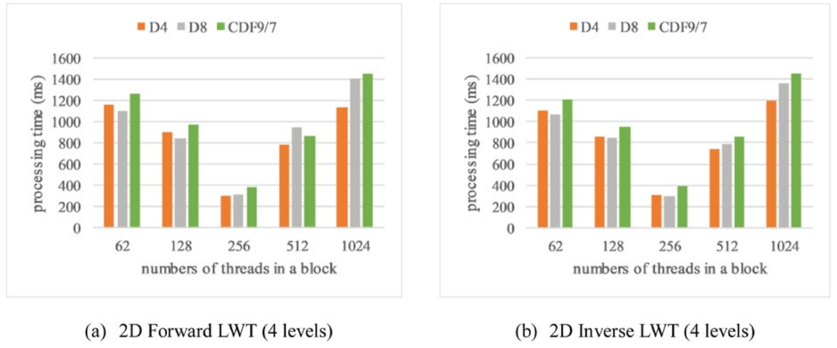


Fig. 12 Processing times of 2D LWT on GPCF-LWT for different numbers of threads in a block.

alt-text: Fig 12

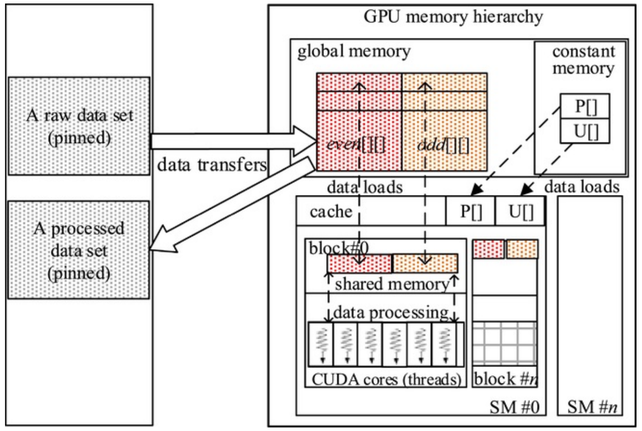


Fig. 13 The hierarchical memory ~~hierarchy~~ usage of GPCF-LWT.

alt-text: Fig 13

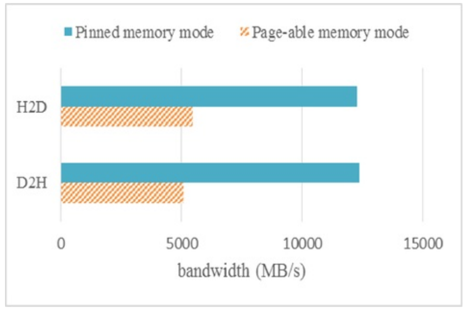


Fig. 14 The bandwidth of pinned and page-able memory mode respectively.

alt-text: Fig 14

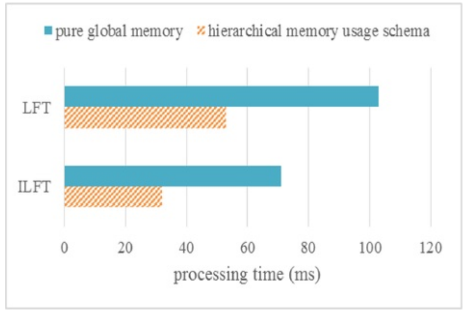


Fig. 15 The performance comparison of the two solutions.

alt-text: Fig 15

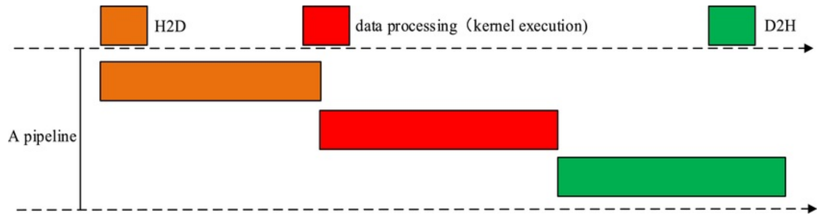


Fig. 16 The timelines of the data transfers and LWT kernel executions in the non-overlapping model.

alt-text: Fig 16

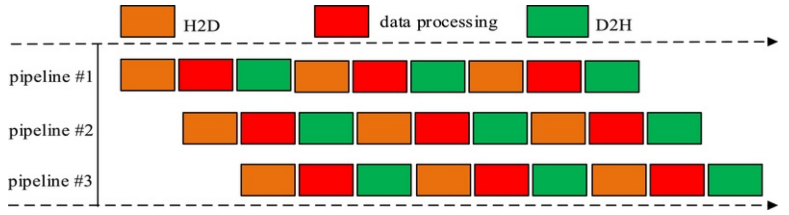


Fig. 17 The timelines for overlapping of three pipelines.

alt-text: Fig 17

Algorithm 1 Overlapping of three stages—two data transfers and kernel executions.

alt-text: Algorithm 1

Inputs: a 2D raw data *Data*[][], and row number *rows*, column number *cols*.

```
1 stream <- cudaStreamCreate(&stream[i])     // create n streams
2 a_h = &Data[0][0]                          // get the start address of Data[][]
3 size= N * cols / nStreams                  // calculate the data size for each stream
4 for i = 0 to nStreams −1 do
5       offset = i * N / nStreams            // calculate the address offset for a stream pipeline
6       cudaMemcpyAsync(a_d+offset, a_h+offset, size, H2D, stream[i]) // transfer data to a GPU in the ith stream
7       LBB_LWT <<<…, stream[i]>>>(a_d+offset)     // launch a wavelet transform kernel in the ith stream
8       cudaMemcpyAsync(a_d+offset, a_h+offset, size, D2H, stream[i]) // transfer data to a CPU in the ith stream
```

Algorithm 2 LBB based 2D LWT kernel.

alt-text: Algorithm 2

Inputs: a 2D raw data *Data*[][], and row number *rows*, column number *cols*.

Outputs: *cA, cH, cV*, and *cD.*

```
1       // for each slide window
2          for(row=0; row < height; row+=SLIDE_HEIGHT) do
3                __shared__ float sData <- load raw data from a GPU global memory space to a GPU shared memory space
4                // each group of row threads performs horizontal 1D LWT in parallel to process a subset and caches
```

```
            _// the temporary results
             even, odd <- split(sData);
             Preprocessing(even, odd);
             __syncthreads();        // synchronization operations
             predict(even, odd);
             update(even, odd);
            __syncthreads();
            // each group of column threads performs vertical 1D LWT in parallel to process the temporary results
            even, odd <- split(sData);
            Preprocessing(even, odd);
            __syncthreads();
            predict(even, odd);
            update(even, odd);
            __syncthreads();
            save(even, odd, output);        // save output results back to a GPU global memory space
```

To test and evaluate the validity and practicality of this pipeline overlapping strategy for LWT computations, this study compared the performance between the overlapping and non-overlapping implementations by using a single GTX 1080 GPU card and performing 4-level 2D LWT with D4 wavelet (data size 4096 × 4096). This experiment recorded the processing time of each step, i.e., H2D, data processing on a GPU card (CUDA kernel executions), and D2H. Based on the experimental results listed in Table 1~~Table 3~~, it can be seen that the overall computational time of non-overlapping implementation is the sum of time consumptions of these three steps since these steps are performed in a sequential order. By contrast, the overall computational times of overlapping implementations are steadily less than the sum of these three steps as they are performed concurrently. Thus, the overlapping optimization strategy can reduce the overall computational time~~ (Table 1)~~.

Table 1 The performance comparison between overlapping and non-overlapping implementation (ms).

alt-text: Table 1

Program routines	Strategies	H2D	Kernel executions	D2H	Overall processing time
FLWT	Non-overlapping	35	193	32	260
	Overlapping	33	197	32	213
ILWT	No-overlapping	33	215	38	286
	Overlapping	34	206	35	225

5 The multi-GPU solution of GPCF-LWT

The powerful computing capability of single GPU satisfies the computational demands of many applications. However, the limited memory spaces and CUDA cores of a single GPU card make it impossible to process a complete dataset in online surface texture measurement applications. Thus, dealing with large-scale dataset requires the multi-GPU processing mode. At present, there are two categories of commonly used multi-GPU systems, i.e., the standalone computers (a single CPU node with multiple GPU cards) and the cluster distributed systems (multiple CPU nodes and each accompanied by one or more GPU cards). Both can accelerate data processing greatly and gain better computational performance. However, limited to the low-speed PCI-E buses and network connections, big data transferring between a CPU host memory space and a GPU global memory space becomes the key bottleneck of multi-GPU systems. Based on this consideration, reducing or hiding data transfer delays are the most important optimization strategy to obtain high performance. More importantly, since the major consumption of LWT is multiplication, addition and subtraction, and it shares only nearby data (i.e. GPCF-LWT does not need to share datasets across the network), so communication requirements among distributed computer nodes in LWT applications are very low. From this point of view, a standalone heterogeneous multi-GPU system has been adopted for GPCF-LWT. This study tries to reduce idle time slices of PCI-E bus and GPU kernel executions, and improves the computational load balancing over the multiple GPU cards for the multi-GPU GPCF-LWT implementation. To reduce idle time slices, this study made two improvements on GPCF-LWT: (1) maximizing the utilization rate of the PCI-E bus bandwidth; (2) hiding data transfer delays through making every GPU card always in the busy state during the whole data processing workflow. For the first point, GPCF-LWT applies bidirectional data transfers on the bandwidth of a PCI-E bus that can support both H2D and D2H at the same time. In terms of the second point, the pipeline overlapping strategy presented in Section 4.4 has been applied on every GPU card in the heterogeneous multi-GPU system. Furthermore, in GPCF-LWT, data transfers between a CPU and a GPU just need to read once, and it does not require any intermediate computational results transferring between a CPU and a GPU or among different GPUs. Thus, this multi-GPU based GPCF-LWT does not require any extra communication, and it has not any data transfer delay during the whole computational procedure. Nevertheless, the traditional dataset division approaches are likely to cause load unbalancing problem and low rate of GPU hardware utilization.

5.1 Load unbalancing problem

Basically, due to the separable property of 2D LWT, a simple load balancing model based on the pure dataset division method can be derived [23]. This solution is simple and useful, but it may lead to load unbalancing problem when a multi-GPU system contains heterogeneous GPU cards with unequal computing capability. As a result, the overall performance of a multi-GPU system depends on the GPU node that has the lowest performance. Moreover, 2D LWT applications are data-oriented computations, i.e., huge data processing but limited CUDA kernels. Thus, this study devised a data-oriented dynamic load balancing (DLB) model for rapid and dynamic dataset division and allocation on heterogeneous GPU nodes by using a fuzzy neural network (FNN) framework [38]. The data-oriented DLB model can minimize the overall processing time by dynamically adjusting the number of data subsets in a group for each GPU node according to runtime feedbacks. Here, two important improvements on FNN of the previous work of authors ([3~~9~~8]) to enhance the effectiveness of DLB model are presented.

5.2 Improved FNN model

The overall framework of the FNN based DLB model devised in the previous work is illustrated in Fig. 18 ~~[39]~~[38]. In this model, five state feedback parameters (fuzzy sets), i.e., floating-point operation performance (*F*), global memory size (*M*), parallel ability (*P*), the occupancy rate of computational resources of a GPU (*UF*) and the occupancy rate of the global memory space of a GPU (*UM*), relating closely to the overall computational performance are defined. The predictor in FNN is devised to predict the relative computational ~~performance~~ability under different workload scenarios, and the scheduler can reorganize the data groups for each GPU card respectively according to dynamic feedbacks of the predictor. In the beginning, a raw dataset is divided into several equal-sized data chunks (the size is relatively small, and the number of data chunks should be far more than the number of GPU nodes) and these chunks are organized into *n* groups (*n* is equal to the number of GPU cards in a multi-GPU system) by using the scheduler. In the runtime, the number of data chunks in a group assigned to each GPU card is different, and it is determined dynamically by the real-time feedbacks (i.e., five state feedback parameters) of a single GPU card. Thus, the DLB model minimizes the overall processing time by dynamically adjusting the number of data chunks in a group for each GPU at runtime.

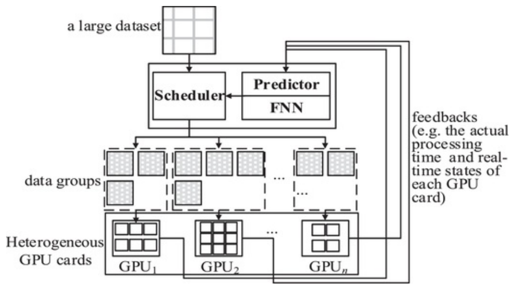


Fig. 18 The overall framework of the FNN based DLB model.

alt-text: Fig 18

Based on the earlier outputs, this study further explored two improvements on FNN to increase the efficiency and predication accuracy of the devised DLB model:

1) This study has constructed other three state feedback parameters for improving the accuracy of the n^{th} relative computational ability CP^n_i predication for i^{th} GPU node, i.e., memory speed (*MS*), boost clock (*BC*) and the last relative computational ability (CP^{n-1}_i or LCP^n_i). Both memory speed and boost clock are core hardware parameters affecting the computing capability of CUDA GPU cards. For example, the memory speed of GeForce GTX 1080 GPU card is 10 Gbps (GBs per second) and the boost clock is 1607 MHz (Mega

Hertz), and the GeForce 750 Ti GPU card is 1.02 Gbps and 1085 MHz, respectively. There is a strong correlation between CP^{n-1}_i and CP^n_i, i.e., when the current CP^{n-1}_i is very high, more subsets will be organized in the data group of CPU$_i$, and it will keep CPU$_i$ busy with high *UM* and *UF* values, which in turn leads to the low CP^n_i. Thus, the balance between CP^{n-1}_i and CP^n_i needs to be considered in predications. Currently, the improved DLB model has eight feedback parameters and they are all fuzzified as "high" and "low" fuzzy subsets, these three kinds of fuzzy sets and subsets are listed in Table 2.

2) This study improves the fuzzy subset descriptions by providing a comprehensive membership function adaption mechanism. In the former DLB model, all fuzzy sets are fuzzified by one membership function–the sigmoid. Here, the static parameters (i.e., *F, M, P, MS* and *BC*) still use sigmoid membership function, whereas the dynamic feedback parameters (i.e., *UF, UM, CP* and *LCP*) adopt an innovative membership function. This membership function has been explored based on the softplus that has been processed by shift and scalar operations to satisfy the fundamental property of a fuzzy membership function. The softplus became an mainstream activation function in deep leaning networks, and it can be formulated as ~~Eq. (23)~~Eq. (21) and its curve is illustrated in Fig. 19 [44]. It grows very slow when *x* is relatively small, and it will be rapidly increased after *x* greater than a threshold value (e.g., 1 in Fig. 19). This phenomenon can model the change of computational performance of a GPU node as when *UF* and *UM* are relatively low, the computing capability is almost unchanged, and it also will drop sharply when *UF* and *UM* are larger than the threshold values. However, the softplus cannot be used as a membership function directly due to ~~Eq. (23)~~Eq. (21) fails to satisfy $f(x) \in [0, 1]$ and $x \in [0,1]$.

$$f(x) = \ln(1 + e^x) \quad \textbf{(21)}$$

Table 2 Three kinds of fuzzy sets and subsets.

alt-text: Table 2

Sets	Descriptions	Fuzzy subsets	Descriptions of fuzzy subsets
MS	Memory speed	*MSL*	Low
		MSH	High
BC	Boost clock	*BCL*	Low
		BCH	High
LCP	The last relative computational ability	*LCPL*	Low
		LCPH	High

Table 3 The specification of a heterogeneous multi-GPU system.

alt-text: Table 3

Property	Description
CPU	Intel Core i7-4790 3.6 GHZ
Memory	16G
GPU	GPU_1: NVIDIA GeForce GTX 750 Ti, 2 G, 5 × SM, 128 SP/SM, 1.02 Gbps memory speed, 1085 MHz boost clock
	3 × GPU_2: NVIDIA GeForce GTX 1080, 8 G, 20 × SM, 128 SP/SM, 10 Gbps memory speed, 1607 MHz boost clock
OS	Windows 10 64 bit
CUDA	Version 8.0

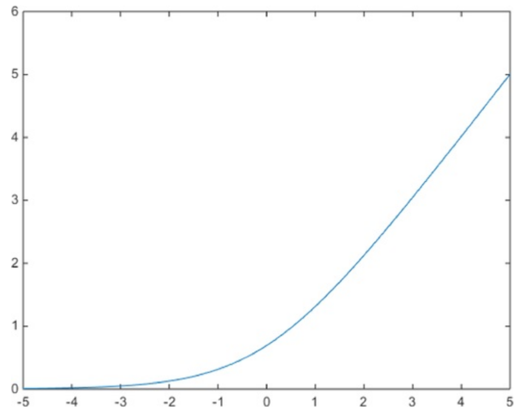


Fig. 19 The curve of softplus function.

alt-text: Fig 19

To remedy this defect, this study performs shift and scalar operations on the softplus function, so the softplus can be extended as follows:

$$f(x) = \ln\left(1 + e^{ax+b}\right) \tag{22}$$

where a is a scalar factor and b is a shift factor, and ~~Eq. (24)~~ Eq. (22) must satisfy:

$$\begin{cases} \lim\limits_{x \to 0} f(x) = 0 \\ \lim\limits_{x \to 1} f(x) = 1 \end{cases} \tag{23}$$

By adding a normalization factor, ~~Eq. (24)~~ Eq. (22) can be redefined as ~~Eq. (26)~~Eq. (24), and a and b can take the value of 5 and −3, respectively. Fig. 20 shows the curve of the improved softplus.

$$f(x) = \ln\left(1 + e^{ax+b}\right) / \ln\left(1 + e^{a+b}\right), x \in [0, 1]\,. \tag{24}$$

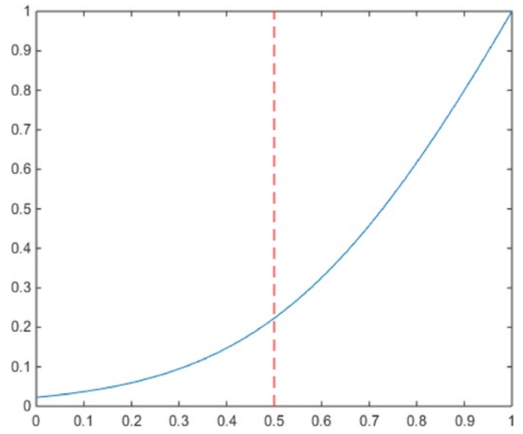


Fig. 20 The curve of the improved softplus function.

alt-text: Fig 20

Based on the three new state feedback parameters and the improved softplus, this study constructed a more effective DLB model that integrates the fuzzy theory, artificial neural networks (ANN) and the back propagation algorithm. The comparison experiment between the improve DLB model and the conventional DLB model in ~~[39]~~ [38] is presented in Section 6.2 (the ~~fourth~~ fifth experiment).

6 Test and performance evaluation

6.1 Hardware and testbed

Table 3 specifies a heterogeneous multi-GPU system constructed for testing and evaluating GPCF-LWT with experiments and the applicable case study, and it contains four different types of GPU cards~~–~~—a middle-low range GPU (NVIDIA GeForce GTX 750 Ti) and three high-end GPUs (NVIDIA GeForce GTX 1080). GTX 750 Ti and GTX 1080 contain 640 and 2560 CUDA cores, respectively.

6.2 Benchmarking experiments

This section tests and evaluates the computational performance and practicability of GPCF-LWT under both a heterogeneous multi-GPU setting and a high-end single GPU setting in five encompassing experiments:

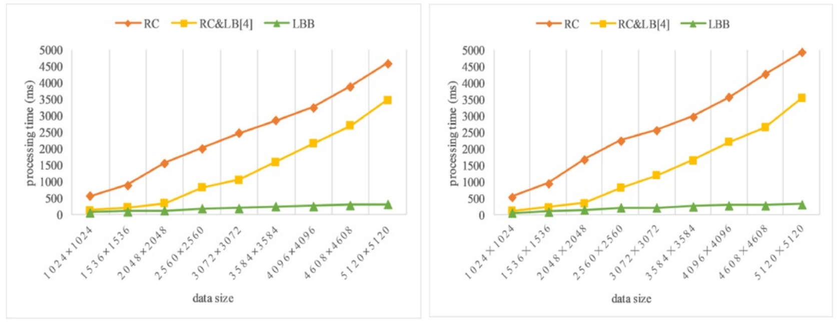
1) The first experiment compares the computational performances of the LBB, RC and RC&LB [4] based on the classic lifting scheme by using the single GPU setting (a GTX 1080 GPU card). The LBB, RC and RC&LB have been tested on the 4-level CDF (9, 7) forward and inverse 2D LWT. The data size ranges from ~~1024~~ 1024 ~~×~~ × 1024 to ~~5120~~ 5120 ~~×~~ × 5120. Fig. 21 shows that with increasing data size, the performance from RC is plunging quickly due to intrinsic synchronization operations. RC&LB shows high computational performance when the data sizes are relatively small (e.g., equal to or smaller than ~~2048~~ 2048 ~~×~~ × 2048). The processing time will increase drastically once the data size goes up for extra synchronization calls. In contrast, the devised LBB model can maintain high-performance for larger data range. This experiment~~al~~ demonstrated that the LBB model in the GPCF-LWT framework is superior and gains up to 14 times speedup in comparison with RC, and up to 7 times when compared with RC&LB under a single GPU configuration.

2) The second experiment tests the effectiveness of the improved lifting scheme by using the single GPU setting. It also performs the 4-level CDF (9, 7) forward and inverse 2D LWT. In Fig. 22, the improved lifting scheme gains higher computational performance since it reduces half of the required synchronization in lifting step (step 3) (see Section 4.2). It substantially increases the parallelizability of LWT computations.

3) To evaluate the generalization feature of CPCF-LWT, the third experiment tests the computational performance of six types of wavelets, namely Haar, D4, D8, D20, CDF (7, 5) and CDF (9, 7), using the classic lifting scheme with RC~~;~~, the classic lifting scheme with RC&LB~~;~~, and the improved lifting scheme with LBB. A single GTX 1080 GPU card is used for ~~a~~the 4-level 2D LWT ~~of the~~ with data size 4096 × 4096. In this experiment, different wavelet transforms have been performed in a generic manner with different *U, P* and *K* for different wavelets. The experimental results highlight that the computational performance of CPCF-LWT is significantly higher than the corresponding RC and RC&LB based implementations, see Fig. 23.

4) The fourth experiment evaluates the effectiveness of the three optimization strategies ~~equipment~~ in GPCF-LWT, again on a single GTX 1080 GPU card. Since each strategy has been tested separately in Section 4.4, this experiment focuses on the comparison between the LBB with the hybrid strategy-driven (the combination of three optimizations) and LBB without any optimization. Fig. 24 shows that the hybrid approach gain noticeable improvements when comparing to the pure LBB.

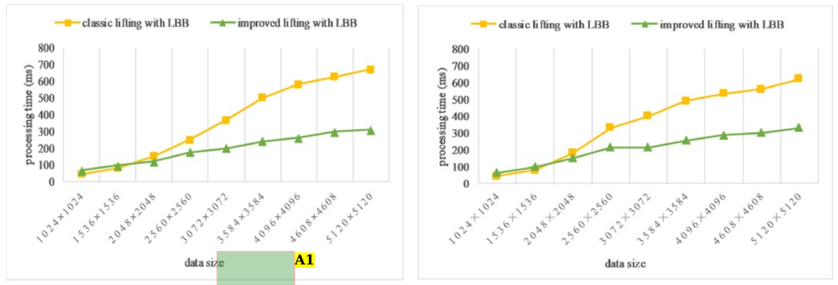
5) The fifth experiment further examined the computational performances of GPCF-LWT when applied on two single GPU settings and three multi-GPU settings by using the heterogeneous GPU system listed in Table 3. This experiment has adapted a CDF (9, 7) wavelet to perform 4-level 2D forward and inverse LWT of large data sizes ranging from 10,~~240~~ 240 ~~×~~ × 10,240 to 16,~~384~~ 384 ~~×~~ × 16,384. Fig. 25 lists the processing time of the five test settings, i.e., the single GTX 750 Ti GPU setting ($setting_1$), the single GTX 1080 GPU setting ($setting_2$), pure dataset division based multi-GPU setting ($setting_3$), the original DLB based multi-GPU setting ($setting_4$) [38] and the improved DLB based multi-GPU setting ($setting_5$). It can be seen from Fig. 25 that $setting_1$ implementation has the lowest computational performance, followed by $setting_2$ since the GTX 1080 GPU contains more CUDA cores (2560), larger global memory space (8GB) and higher data transfer speed (10 Gbps) than the GTX 750 Ti GPU (640, 2GB, 5.4 Gbps, respectively). However, the computational performance improvement of $setting_2$ is very limited, which proves that it will not lead to the linear computational performance improvement for large-scale computational applications accompanied by extremely large input volume and highly repetitive simple operational procedures or iterations, specifically for 2D LWT by simply changing a better GPU card. The $setting_3$ implementations can gain a slightly higher computational performance improvement than the two single GPU implementations for about 2 and 3 times. In contrast, the DLB-driven GPCF-LWT multi-GPU implementations ($setting_4$ and $setting_5$) achieve significant computational performance enhancement since the FNN based DLB can allocate data tasks dynamically during runtime according to the relative computational ~~performance~~ability across all GPU nodes. Moreover, the improved DLB ($setting_5$) has gained better performance than the original version ($setting_4$) for all data sizes due to the softplus based membership function that is more suitable for adapting the dynamic change of computational performance of heteronomous GPU nodes. The peak performance gain (speedup on data size of 16,384 × 16,384) of the $setting_5$ has reached approximately 10 times more than $setting_3$; and when compared with the two single GPU implementations, the speedup from $setting_5$ has reached 15 and 20 times than $setting_1$ and $setting_2$ respectively.



(a) **2D forward LWT** (b) **2D inverse LWT**

Fig. 21 The performance comparison among LBB, RC and RC&LB models.

alt-text: Fig 21



A1

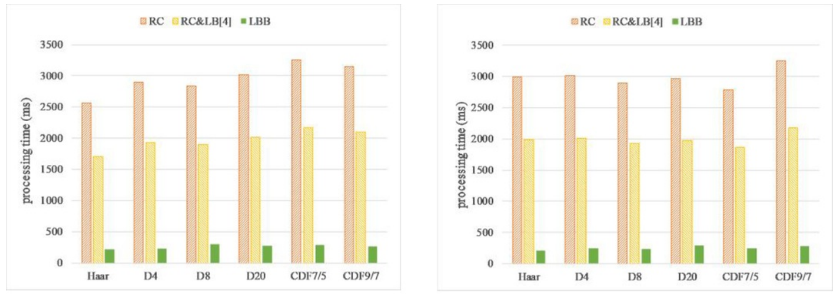
(a) **2D forward 2D LWT** (b) **2D inverse LWT**

Fig. 22 The performance comparison between the classic and improved lifting scheme.

alt-text: Fig 22

Annotations:

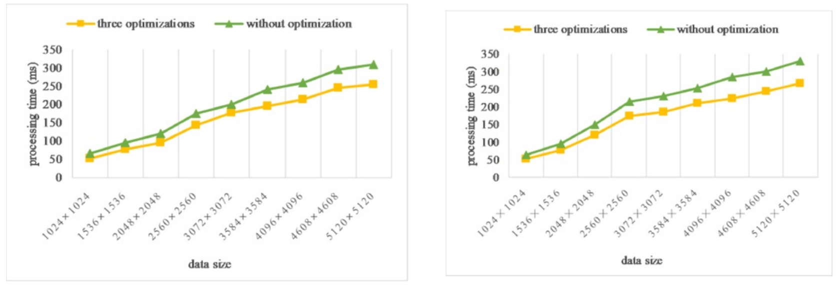
A1. delete the extra "2D"



(a) **2D forward LWT** (b) **2D inverse LWT**

Fig. 23 The generalization feature evaluation of CPCF-LWT.

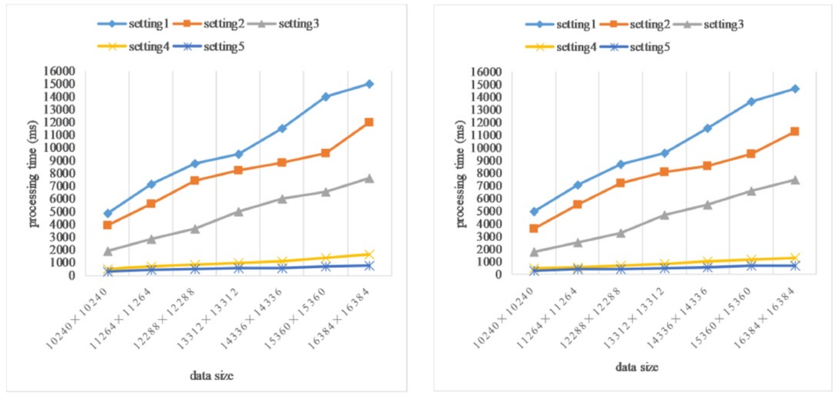
alt-text: Fig 23



(c) **2D forward LWT** (d) **2D inverse LWT**

Fig. 24 The effectiveness experiment of the three optimization strategies in GPCF-LWT.

alt-text: Fig 24



(a) 2D forward LWT (b) 2D inverse LWT

Fig. 25 The comparison of processing times of *setting*$_1$, *setting*$_2$, *setting*$_3$, *setting*$_4$ and *setting*$_5$ implementation~~s~~.

alt-text: Fig 25

6.3 A case study on online engineering surface filtration

This section carries a case study on online engineering surface filtration to test the usability and efficiency of the devised multi-GPU GPCF-LWT in comparison with other state-of-the-art multi-GPU approaches. In the field of areal surface texture characterization~~ extraction~~s, geometrical characteristic signals relating closely with functional requirements can be extracted precisely from wavelet coefficients by using different transmission bands. The 2D forward LWT computations should be performed on raw 2D surface measured datasets to obtain instances of detail coefficients D_i at all levels and the approximation coefficient cA instances from the last level, and the instances of cA_i of each level can be obtained during performing 2D inverse LWT. Based on instances of D_i and cA_i, the geometrical characteristics (i.e., roughness, waviness and form) of a surface texture can be obtained by calculating the following inverse LWT defined in ~~Equation~~ Eq. ~~(27)~~Eq. (25) [10,40].

$$\begin{cases} Roughness: \eta(x,y) = ILWT\left(D_{i,\frac{j}{2^{j-1}}}\right) = \sum_{i=1}^{l}\left\{D_{i,\frac{j}{2^{j-1}}}(x,y) + \rho\left[A_{i-1,\frac{j}{2^{j-1}}}(x,y)\right]\right\} \\ Waviness: \eta'(x,y) = ILWT\left(D_{i,\frac{j}{2^{j-1}}}\right) = \sum_{i=l}^{i}\left\{D_{i,\frac{j}{2^{j-1}}}(x,y) + \rho\left[A_{i-1,\frac{j}{2^{j-1}}}(x,y)\right]\right\} \\ Form: \eta''(x,y) = ILWT\left(A_{i,\frac{j}{2^{j-1}}}\left(x, \frac{k}{2^{j-1}}\right)\right) \end{cases} \quad (25)$$

In order to extract the roughness $\eta(x, y)$ and waviness $\eta'(x, y)$ from a surface texture profile, the cA_i must be setup to zero. Similarly, the D_i must be setup to zero for extracting the form characteristic.

This study selected two kinds of raw engineering surfaces (surfaces on ceramic femoral heads and milled metallic workpieces) to test and evaluate the GPCF-LWT. The GPCF-LWT integrates the devised LBB, improved lifting scheme, the three implementation optimizations, and the improved data-oriented DLB. This analysis evaluates the SURFSTAND that is a surface characterization system employing modern CPU core [10], and two other multi-GPU implementations of RC&LB and the method proposed by Ikuzawa ~~[46]~~[45]. It should be noted that the pure dataset division method was applied for the two multi-GPU implementations. The heterogeneous multi-GPU listed in Table 3 is used for all three multi-GPU implementations.

Figs. 26a and 27a show a raw measured dataset from the functional surface of a ceramic femoral head and a milled metallic workpiece respectively with the sampling size ~~4096~~ 4096 ~~×~~ × 4096. In Fig. 26a, the measured surface texture has two different types of scratches produced by the grinding manufacturing process, and one of them is regular and shallow scratches while the other is random deeper scratch. The six-level D4 wavelet has been used in this case to perform both forward and inverse 2D LWT. The Fig. 26b-d demonstrate the roughness, waviness and form characteristics extracted from a femoral head surface by using the inverse LWT defined in ~~Equation~~ Eq. ~~(27)~~Eq. (25). In the case of the milled metallic surface (see Fig. 27a), the six-level CDF (9, 7) wavelet has been used to perform 2D LWT, and Fig. 27b–d demonstrate the roughness, waviness, and form characteristics extracted from a milled metallic surface by using the inverse LWT defined in ~~Equation~~ Eq. ~~(27)~~Eq. (25).

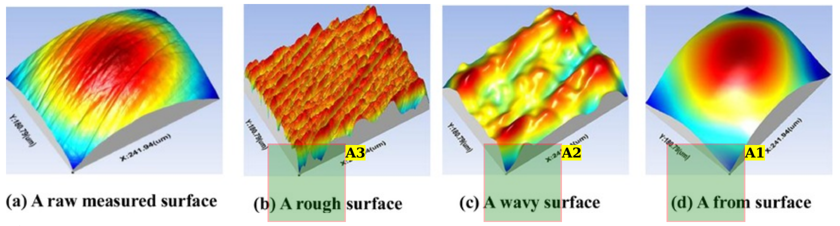


Fig. 26 The functional surface texture of a ceramic femoral head.

alt-text: Fig 26

Annotations:

A1. The form surface

A2. The wavy surface

A3. The rough surface

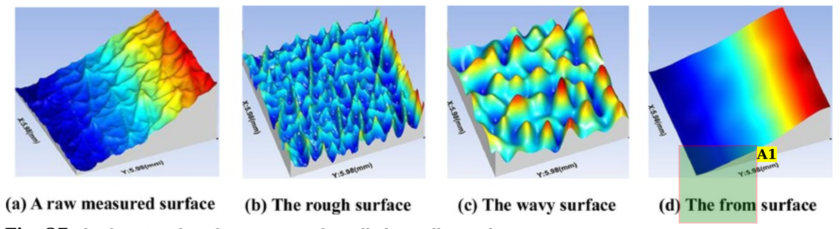


Fig. 27 The functional surface texture of a milled metallic workpiece.

alt-text: Fig 27

Annotations:

A1. The form surface

It is worth mentioning that CUDA supports the single float (32-bit, abbreviated as FP32) and the half float (16-bit, FP16) precisions ~~[19,45]~~[19,46]. This experiment tested both of these two precisions. From this test, it is discovered that float precision settings do not affect the computational performance of 2D LWT on CPU due to the Intel Core i7-4790 CPU uses 64-bit instruction. Both FP16 and FP32 operations can be executed on the 64-bit mode in negligible difference. In contrast, FP16 based multi-GPU implementations show higher computational performance than the FP32 implementations. However, the lower precision of FP16 fails to fulfil the high precision requirement of micro- and nano-surface texture metrology, such that this case study only applies the FP32 for evaluation. The results show that the multi-GPU GPCF-LWT and pure CPU (Intel Core i7-4790 3.6GHZ) implementations have same filtration effects on both precision and accuracy aspects. Table 4 shows that multi-GPU implementations can gain over 20 times speedup over CPU implementations.

Table 4 The performance comparisons between a CPU and the multi-GPU implementation (ms).

alt-text: Table 4

Datasets	LWT		ILWT	
	CPU	GPCF-LWT	CPU	GPCF-LWT
Ceramic femoral head (4096 × 4096)	10,165	415	12,590	450
Ceramic femoral head (12,288 × 12,288)	28,569	756	34,681	832
Milled metallic workpiece (4096 × 4096)	12,850	510	12,685	511
Milled metallic workpiece (12,288 × 12,288)	30,256	748	35,647	786

Fig. 28 shows the performance comparison among GPCF-LWT, RC&LB and Ikuzawa's solution on the heterogeneous multi-GPU system detailed in Table 3. In this case, the testing dataset has been expanded to the size of 12,~~288~~ 288 ~~×~~ × 12,288 to evaluate whether GPCF-LWT can deal with large-scale datasets in the online manner. The experimental results show that RC&LB has the minimum computational performance due to its heavy synchronization calls. Ikuzawa's solution gains better performance due to its higher efficiency on memory usage. Though both still suffer from the heavy synchronization, pipeline cluttering, and data overload problems. In contrast, the GPCF-LWT with advanced optimizations gains up to 12 times on processing speed over other two multi-GPU approaches.

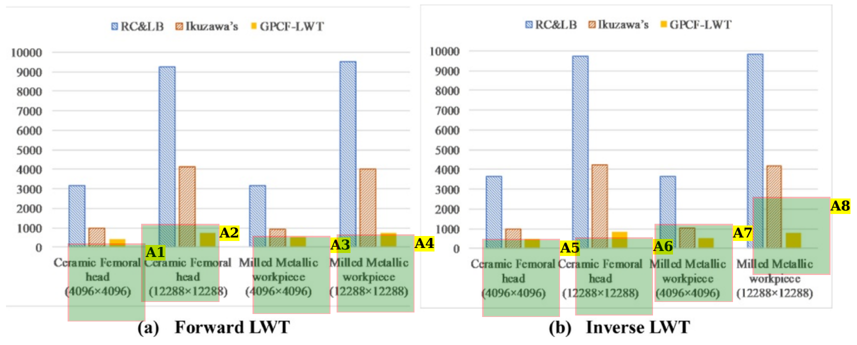


Fig. 28 The performance comparisons of three multi-GPU implementations (ms).

alt-text: Fig 28

Annotations:

A1. Ceramic femoral head

A2. Ceramic femoral head

A3. Milled metallic workpiece

A4. Milled metallic workpiece

A5. Ceramic femoral head

A6. Ceramic femoral head

A7. Milled metallic workpiece

A8. Milled metallic workpiece

In summary, multi-GPU based GPCF-LWT has gained significant improvement on the computational performance for 2D LWT. It can process a very large dataset (e.g., 12~~,288~~ 288 ~~×~~ × 12~~,~~288) in less than one second, so that performance requirements of online (or near real-time) and ~~large-scale~~ data intensive surface texture filtration applications can be satisfied gracefully.

7 Conclusions and future work

To fulfill the online and big data requirements in micro- and nano-surface metrology, the parallel computational power of modern GPUs is explored. This research devised a generic parallel computational framework for lifting wavelet transform (GPCF-LWT) acceleration based on a heterogeneous multi-GPU system. One of the outcomes of this innovative infrastructure is a new parallel computational model named LBB to alleviate the vital problem of shared memory shortage in a single GPU card while ensuring the generalization of LWT in the context of CUDA memory hierarchy. To increase the parallelizability of LWT, an improved lifting scheme has also been developed. Three optimization strategies on the single GPU configuration are validated and then integrated. To further improve the computational performance of the GPCF-LWT, it has been expanded to compute on a heterogeneous multi-GPU system with a Fuzzy Theory-based load balancing model. This study further explored two improvements on the previous devised FNN DLB model to increase the efficiency and predication accuracy. Experiments show that the proposed GPCF-LWT can achieve superior and substantial computational performance gain when compared with other state-of-the-art techniques. Over 20 times speedup has been monitored over the pure CPU solutions, and up to 12 times over other benchmarking multi-GPU solutions with large-scale surface measurement datasets. The innovative framework and its corresponding techniques have addressed the key challenges from 2D LWT applications that are vital for big surface dataset processing.

One avenue opened up during the study for future exploration is to introduce the deep learning idealism across the GPU and CPU boundary. Especially when facing cell-CPU or cluster CPUs, a hybrid and efficient data distribution scheme, as well as an online surface texture analysis platform with smart recommendation functions for filter selection will be of great value for practitioners.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the NSFC (61203172), the Sichuan Science and Technology Programs (2018JY0146 and 2019YFH0187), and “598649-EPP-1-2018-1-FR-EPPKA2-CBHE-JP” from the European Commission.

References

[1] X.J. Jiang and D.J. Whitehouse, Technological shifts in surface metrology, *CIRP Ann* **61** (2), 2012, 815–836, https://doi.org/10.1016/j.cirp.2012.05.009.

[2] D.A. Axinte, N. Gindy, et al., Process monitoring to assist the workpiece surface quality in machining, *Int. J. Mach. Tools Manuf.* **44** (10), 2004, 1091–1108, https://doi.org/10.1016/j.ijmachtools.2004.02.020.

[3] X. Jiang, P.J. Scott, et al., Paradigm shifts in surface metrology. Part I. Historical philosophy, *Proc. R. Soc. A Math. Phys. Eng. Sci.* **463** (2085), 2007, 2049–2070, https://doi.org/10.1098/rspa.2007.1874.

[4] W.J. van der Laan, A.C. Jalba, et al., Accelerating wavelet lifting on graphics hardware using CUDA, *IEEE Trans. Parallel Distrib. Syst.* **22** (1), 2011, 132–146, https://doi.org/10.1109/TPDS.2010.143.

[5] J.A. Castelar, C.A. Angulo, et al., Parallel decompression of seismic data on GPU using a lifting wavelet algorithm, Signal Process. Images Comput. Vis., 2015, IEEE.

[6] S. Lou, W.H.H. Zeng, et al., Robust filtration techniques in geometrical metrology and their comparison, *Int. J. Autom. Comput.* **10** (1), 2013, 1–8, https://doi.org/10.1007/s11633-013-0690-4.

[7] H.S. Abdul-Rahman, S. Lou, et al., Freeform texture representation and characterisation based on triangular mesh projection techniques, *Measurement* **92**, 2016, 172–182, https://doi.org/10.1016/j.measurement.2016.06.005.

[8] ISO 16610-61:2015 Geometrical product specification (GPS) – filtration – part 61: linear areal filters – Gaussian filters, (2015).

[9] ASME B46.1-2009, Surface texture: surface roughness, waviness, and lay, (2009).

[10] P. Nejedly, F. Plesinger, et al., CudaFilters: a signalplant library for GPU-accelerated FFT and FIR (The original reference No.29 becomes No.10, original No.10 becomes the No.11, No.11 becomes No.12, and No.12 becomes No.13, etc.) filtering, *Softw. - Pract. Exp.* **48** (1), 2018, 3–9, https://doi.org/10.1002/spe.2507.

[11] X. Jiang, P.J. Scott, et al., Paradigm shifts in surface metrology. Part II. The current shift, *Proc. R. Soc. A Math. Phys. Eng. Sci.* **463** (2085), 2007, 2071–2099, https://doi.org/10.1098/rspa.2007.1873.

[12] W. Sweldens, The lifting scheme: a construction of second generation wavelets, *SIAM J. Math. Anal.* **29** (2), 1998, 511–546.

[13] S. Xu, W. Xue, et al., Performance modeling and optimization of sparse matrix-vector multiplication on NVIDIA CUDA platform, *J. Supercomput.* **63** (3), 2013, 710–721, https://doi.org/10.1007/s11227-011-0626-0.

[14] H. Wang, H. Peng, et al., A survey of GPU-based acceleration techniques in MRI reconstructions, *Quant. Imaging Med. Surg.* **8** (2), 2018, 196–208, https://doi.org/10.21037/qims.2018.03.07.

[15] S. Mittal and J.S. Vetter, A survey of CPU-GPU heterogeneous computing techniques, *ACM Comput. Surv.* **47** (4), 2015, 1–35, https://doi.org/10.1145/2788396.

[16] M. Mccool, Signal processing and general-purpose computing and GPUs, *IEEE Signal Process. Mag.* **24** (3), 2007, 109–114, https://doi.org/10.1109/MSP.2007.361608.

[17] Y. Su, Z. Xu, et al., GPGPU-based Gaussian filtering for surface metrological data processing, In: *2008 12th Int. Conf. Inf. Vis.,* 2008, IEEE, 94–99, https://doi.org/10.1109/IV.2008.14.

[18] NVIDIA, CUDA C Programming Guide v9.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[19] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 2012, Newnes.

[20] NVIDIA, CUDA C Best Practices Guide v9.0, NVIDIA. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[21] D.B. Kirk and W.H. Wen-Mei, Programming Massively Parallel Processors: A Hands-on Approach, 2012, Morgan Kaufmann.

[22] R. Couturier, Designing Scientific Applications On GPUs, 2013, CRC Press.

[23] D. Foley and J. Danskin, Ultra-performance Pascal GPU and NVLink interconnect, *IEEE Micro* **37** (2), 2017, 7–17, https://doi.org/10.1109/MM.2017.37.

[24] M. Hopf and T. Ertl, Hardware accelerated wavelet transformations, Data Vis., 2000, Springer, 93–103, https://doi.org/10.1007/978-3-7091-6783-0_10.

[25] Tien-Tsin Wong, Chi-Sing Leung, et al., Discrete wavelet transform on consumer-level graphics hardware, *IEEE Trans. Multimed.* **9** (3), 2007, 668–673, https://doi.org/10.1109/TMM.2006.887994.

[26] J. Franco, G. Bernabé, et al., The 2D wavelet transform on emerging architectures: gPUs and multicores, *J. Real-Time Image Process* **7** (3), 2012, 145–152, https://doi.org/10.1007/s11554-011-0224-7.

[27] C. Song, Y. Li, et al., A GPU-Accelerated wavelet decompression system with SPIHT and reed-solomon decoding for satellite images, *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens* **4** (3), 2011, 683–690, https://doi.org/10.1109/JSTARS.2011.2159962.

[28] L. Zhu, Y. Zhou, et al., Parallel multi-level 2D-DWT on CUDA GPUs and its application in ring artifact removal, *Concurr. Comput. Pract. Exp.* **27** (17), 2015, 5188–5202, https://doi.org/10.1002/cpe.3559.

[29] ~~L. Blunt and X. Jiang, Advanced Techniques For Assessment Surface Topography: Development of a Basis For 3D Surface Texture Standards Surfstand~~(Here original No.28 becomes No.29, and the rest is numbered successively.)~~, 2003, Elsevier.~~

[30] Wang Jianjun, Lu Wenlong, et al., High-speed parallel wavelet algorithm based on CUDA and its application in three-dimensional surface texture analysis, In: *Int. Conf. Electr. Inf. Control Eng.,* 2011, IEEE, 2249–2252, https://doi.org/10.1109/ICEICE.2011.5778225.

[31] Y. Chen, X. Cui, et al., Large-scale FFT on GPU clusters, In: *Proc. 24th ACM Int. Conf. Supercomput.,* New York, New York, USA2010, ACM Presshttps://doi.org/10.1145/1810085.1810128.

[32] A. Nukada, Y. Maruyama, et al., High performance 3-D FFT using multiple CUDA GPUs, In: *Proc. 5th Annu. Work. Gen. Purp. Process. with Graph. Process. Units - GPGPU,* New York, New York, USA2012, ACM Press, 57–63, https://doi.org/10.1145/2159430.2159437.

[33] S. Schaetz and M. Uecker, A multi-GPU programming library for real-time applications, In: *Int. Conf. Algorithms Archit. Parallel Process,* 2012, Springer, 114–128.

[34] J.A. Stuart and J.D. Owens, Multi-GPU MapReduce on GPU clusters, In: *2011 IEEE Int. Parallel Distrib. Process. Symp.,* 2011, IEEE, 1068–1079, https://doi.org/10.1109/IPDPS.2011.102.

[35] M. Grossman, M. Breternitz, et al., HadoopCL: MapReduce on distributed heterogeneous platforms through seamless integration of hadoop and OpenCL, In: *2013 IEEE Int. Symp. Parallel Distrib. Process,* 2013, IEEE, 1918–1927, https://doi.org/10.1109/IPDPSW.2013.246.

[36] M. Abadi, A. Agarwal, et al., TensorFlow: large-Scale machine learning on heterogeneous distributed systems, (2016). doi:10.1038/nn.3331.

[37] S. Chintala, An overview of deep learning frameworks and an introduction to Pytorch, 2017. https://smartech.gatech.edu/handle/1853/58786(accessed October 5, 2017).

[38] C. Zhang, Y. Xu, et al., A fuzzy neural network based dynamic data allocation model on heterogeneous multi-Gpus for large-scale computations, *Int. J. Autom. Comput.* **15** (2), 2018, 181–193, https://doi.org/10.1007/s11633-018-1120-4.

[39] K.K. Shukla and A.K. Tiwari, Filter banks and DWT, SpringerBriefs Comput. Sci., 2013, Springer, 21–36, https://doi.org/10.1007/978-1-4471-4941-5_2.

[40] X.Q. Jiang, L. Blunt, et al., Development of a lifting wavelet representation for surface characterization, *Proc. R. Soc. A Math. Phys. Eng. Sci.* **456** (2001), 2000, 2283–2313, https://doi.org/10.1098/rspa.2000.0613.

[41] M.E. Angelopoulou, K. Masselos, et al., Implementation and comparison of the 5/3 lifting 2D discrete wavelet transform computation schedules on FPGAs, *J. Signal Process. Syst.* **51** (1), 2008, 3–21, https://doi.org/10.1007/s11265-007-0139-5.

[42] C. Tenllado, J. Setoain, et al., Parallel implementation of the 2D discrete wavelet transform on graphics processing Units: filter bank versus lifting, *IEEE Trans. Parallel Distrib. Syst.* **19** (3), 2008, 299–310, https://doi.org/10.1109/TPDS.2007.70716.

[43] W. Nicholas, The CUDA Handbook: A Comprehensive Guide to GPU Programming, 2013.

[44] Y. LeCun, Y. Bengio, et al., Deep learning, *Nature* **521** (7553), 2015, 436–444, https://doi.org/10.1038/nature14539.

[45] T. Ikuzawa, F. Ino, et al., Reducing memory usage by the lifting-based discrete wavelet transform with a unified buffer on a GPU, *J. Parallel Distrib. Comput.* **93-94**, 2016, 44–55, https://doi.org/10.1016/j.jpdc.2016.03.010.

[46] NVIDIA, Whitepaper - NVIDIA GeForce GTX 750 Ti, 2014.

Highlights

- A new parallel computation model for LWT named LBB was devised to tackle the limitation of current shared memory bottleneck on modern GPUs.
- The improved lifting scheme has reduced approximately half of the synchronizations needed when computing the lifting wavelets in a single CUDA thread.
- Three optimization strategies were benchmarked for GPCF-LWT implementation.
- The multi-GPU based GPCF-LWT supports online large-scale measured data filtration through an improved FNN based dynamic load balancing (DLB) model.

Queries and Answers

Query: Please confirm that givennames and surnames have been identified correctly.
Answer: Yes

Query: Please confirm that the provided email 18668289@qq.com is the correct address for official communication, else provide an alternate e-mail address to replace the existing one.
Answer: Please change email address to creekline017@hotmail.com

Query: Please check the citation of Table 1 in the text. It has been put randomly. Please fix it in proper place.
Answer: The citation of Table 1 has been corrected in the text.

Query: Please check the citation of Eq. (26) in the text, But there is no Eq. (26) in the text. Please correct if necessary.
Answer: Eq. (24) is actually Eq. (22), and Eq.26 is actually Eq. (24) . Other wrong equation number had been corrected in text.

Query: Please check the citation of Eq. 27 in the text. It is not in the text. Please correct it if necessary.

Answer: There are 15 places that the equation number is wrong. We corrected and marked in the text. Thank you.

Query: Please check funding information and confirm its correctness.
Answer: I confirm its correctness

Query: Please provide detail of reference nos. 8 and 9.
Answer: [8]ISO 16610-61:2015, Geometrical product specification (GPS) – filtration – part 61: linear areal filters – Gaussian filters, An ISO/TC 213 dimensional and geometrical product specifications and verification standard, 2015, 1-18, https://www.iso.org/standard/60813.html.

[9]ASME B46.1-2009, Surface texture: surface roughness, waviness, and lay, An American national standard, 2009, 1-120, https://www.asme.org/products/codes-standards/b461-2009-surface-texture-surface-roughness.

Query: Please provide detail of reference no. 36.
Answer: M. Abadi, A. Agarwal, et al., TensorFlow: large-Scale machine learning on heterogeneous distributed systems, TensorFlow whitepaper, 2016, 1-19, https://doi.org/10.1038/nn.3331.

Query: Please provide Publisher name and place for reference no. 45.
Answer: No.45 is a journal paper, and You mean No.46?
NVIDIA, NVIDIA GeForce GTX 750 Ti, Whitepaper, NVIDIA Corporation, 2014, 1-10, https://international.download.nvidia.cn/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.