A fast discrete wavelet transform using hybrid parallelism on GPUs

Tran Minh Quan, *Student Member, IEEE,* and Won-Ki Jeong, *Member, IEEE*

Abstract—Wavelet transform has been widely used in many signal and image processing applications. Due to its wide adoption for time-critical applications, such as streaming and real-time signal processing, many acceleration techniques were developed during the past decade. Recently, the graphics processing unit (GPU) has gained much attention for accelerating computationally-intensive problems and many solutions of GPU-based discrete wavelet transform (DWT) have been introduced, but most of them did not fully leverage the potential of the GPU. In this paper, we present various state-of-the-art GPU optimization strategies in DWT implementation, such as leveraging shared memory, registers, warp shuffling instructions, and thread- and instruction-level parallelism (TLP, ILP), and finally elaborate our hybrid approach to further boost up its performance. In addition, we introduce a novel mixed-band memory layout for Haar DWT, where multi-level transform can be carried out in a single fused kernel launch. As a result, unlike recent GPU DWT methods that focus mainly on maximizing ILP, we show that the optimal GPU DWT performance can be achieved by hybrid parallelism combining both TLP and ILP together in a mixed-band approach. We demonstrate the performance of our proposed method by comparison with other CPU and GPU DWT methods.

Index Terms—Wavelet Transform, Hybrid Parallelism, Lifting Scheme, Bit rotation, GPU Computing

1 Introduction

The discrete wavelet transform (DWT) has been actively studied in the image processing domain. One example is image compression: One can discard less-important information from a wavelet-transformed image more effectively, as in JPEG2000 [1]. The Federal Bureau of Investigation (FBI) uses a wavelet-based compression method for their fingerprint image database, which is one of the most effective biometric authentication techniques [2]. Another example is feature detection: One can develop an edge-detection filter via leveraging the high-pass filter of a wavelet transform [3]. Lastly, the wavelet transform is also commonly used as a sparsifying transform in the Compressed Sensing reconstruction theory [4] in order to reduce noise or outliers in the reconstructed signal. Figure 1 illustrates the results of applying DWT to a typical 2D image and 3D medical data. More comprehensive reviews of wavelet transforms and their applications can be found in [5] and [6].

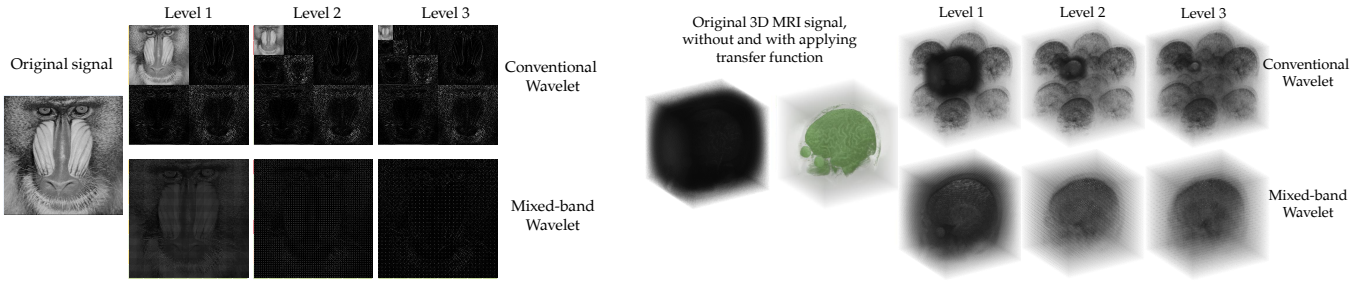
Not only has DWT been widely used in various disciplines, but also many of wavelet's applications are time-critical, such as in processing streaming video or the real-time reconstruction of sensor data. In order to accelerate wavelet transforms for such those purposes, special hardware or accelerators have been used. These include Field Programmable Gate Arrays (FPGAs), [7], [8], Intel Many Integrated Core (MIC) architecture, [9], and Graphics Processing Units (GPUs). Among them, GPUs have gained much attention due to their superior performance in terms of cost and energy consumption. High-level GPU programming APIs, such as NVIDIA CUDA [10] and OpenCL [11], have also promoted wide adoption of GPUs by lowering the learning curve.

The motivation of this work stems from our recent research on GPU-accelerated Compressed Sensing MRI reconstruction where the wavelet is used as a sparsifying transform. We observed that the main bottleneck of conventional wavelet transforms is global memory transactions because intermediate results must be stored in global memory during multi-level wavelet transformation. Meanwhile, we realized that the spatial location of wavelet coefficients is not important in many applications, such as compressed sensing reconstruction. Because $\ell 1$ minimization in compressed sensing is basically a thresholding process to suppress small non-zero wavelet coefficients, we are interested only in frequency statistics of the wavelet domain, i.e., the histogram of the wavelet transformed image. Therefore, it is acceptable to rearrange the location of wavelet coefficients as long as the forward and inverse transforms can reproduce the exact input data without any loss, which is the main idea of our mixed-band wavelet transform.

Another observation we made was that no single optimization technique works best for GPU DWT implementation. Many-core processors were originally designed to exploit thread-level parallelism (TLP), but the previous study [12] showed that instruction-level parallelism (ILP) plays an important role in hiding memory latency and increasing throughput of the GPU. Along that line, Enfedaque et al. [13] recently reported that maximizing ILP using registers only without using shared memory showed good performance in their GPU DWT implementation. In the proposed work, however, we discovered that hybrid parallelism by combining register-based ILP and shared memory-based TLP actually performs better for GPU DWT.

The main contribution of this work is several-fold. First, we explore various optimization strategies for the GPU implementation of the discrete wavelet transform, such as

• *Tran Minh Quan, and Won-Ki Jeong are with Ulsan National Institute of Science and Technology (UNIST).*
E-mail: {quantm, wkjeong}@unist.ac.kr



(a) 2D image (b) 3D volume

Fig. 1: Examples of conventional (top row) and mixed-band (bottom row) discrete wavelet transform. Unlike the conventional wavelet transform, the mixed-band method does not separate high- and low-frequency coefficients in the transformed domain.

using shared memory, registers, warp shuffling instructions, and maximizing ILP. Second, we provide rigorous performance analysis of the presented strategies on NVIDIA's Fermi and Kepler GPU architectures, and propose a hybrid approach by exploiting both ILP and TLP. Last, we introduce a mixed-band memory layout for Haar DWT that allows in-place filtering and fused multi-level transformation for further acceleration.

The rest of this paper is organized as follows: Section 2 provides a survey of related literature, both in theoretical approaches and in their recent GPU implementations. Section 3 provides an overview of lifting schemes for the Haar, CDF 5/3 and CDF 9/7 wavelet families, which is necessary to explain the proposed method. It also includes the mixed-band memory layout as well as the conversion between conventional and mixed-band approach, i.e., bit-circular shift (bitwise rotation) operation to determine the locations of wavelet coefficients. Section 4 discusses the various GPU optimization strategies for DWT on different GPU architectures and our proposed hybrid approach. Section 5 provides an in-depth analysis and runtime comparisons to other GPU DWT methods together with CPU methods. Performance of special optimization for multi-level Haar DWT is also included. Finally, Section 6 concludes the paper by discussing the advantages as well as the potential drawbacks of the proposed work and suggesting future research directions.

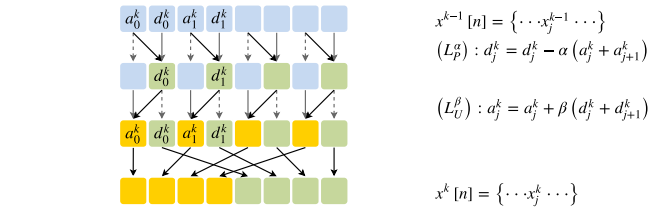
2 Related Work

In this section, we survey the recent research trends in discrete wavelet transforms and GPU computing.

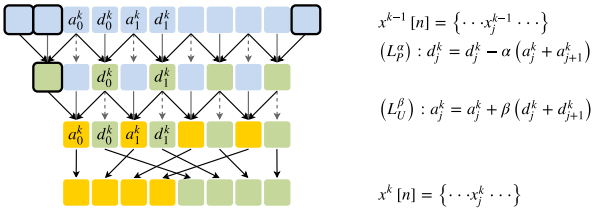
Discrete wavelet transforms: In the history of DWT development, the *first-generation* wavelet [14] initiated the basic idea of transforming the input signal to *approximation* (low frequency) and *detail* (high-frequency) signals by convolving a translated/dilated basis function, i.e., a mother wavelet. A decade later, Sweldens [15] introduced the *second-generation* wavelet, i.e., a lifting scheme that allows in-place filtering to reduce computations. It was later generalized by Sole et al. [16] as the *lazy* wavelet. Daubechies [6] built a theoretical foundation of constructing wavelet basis [6], including CDF 5/3 and CDF 9/7. A more in-depth review of the wavelet theory and applications in signal processing research can be found in [5].

GPU-accelerated DWT: An early work by Tenllado et al. [17] was based on graphics APIs (e.g., OpenGL [18]) in order to map computing problems into the graphic pipeline. Due to the hardware limitation and inflexible programmability, their approach yielded only a moderate speed improvement. Later, general computing APIs (e.g., NVIDIA CUDA [10] and OpenCL [11]), and advanced user-controllable hardware features (e.g., caches, shared memory, register, texture, etc.) accelerated wider adoption of GPUs for general computing problems. including DWT. Matela [19] introduced an early work of optimized wavelet lifting scheme implementation on NVIDIA GPUs using CUDA. The proposed method constructed a special mapping on shared memory so that the 2D filtering could be performed without a transposing step. Franco et al. [20], [21] proposed a row-based 1D filtering followed by an image transpose to process a 2D image in two steps, which is accelerated by shared memory. Laan et al. [22] ported the efficient CPU lifting scheme implementation [15] on the GPU. In the following work [23], Laan et al. proposed a sliding window technique to store intermediate values in shared memory during the vertical pass and reuse them to perform the filtering without an image transpose, which led to a large performance leap. Song et al. [24] proposed a method of column segmentation and establish a fast vertical transform strategy. Among them, [20], [21], [22], [23] and [24] can be classified as 1D row/column-based methods with various handling of the vertical wavelet transform. A common problem with those methods is the expensive global memory transaction between each 1D filtering.

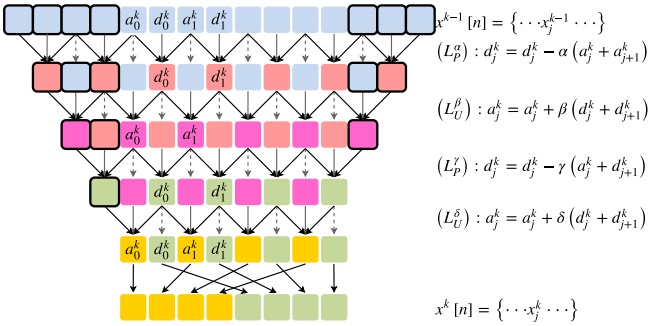
On the other hand, the block-based methods [25], [26] suggested different strategies to perform the wavelet transform in both horizontal and vertical directions in only one pass. Song et al. [25] overcame the limitations of Matela [19] (i.e., incorrect neighboring pixel values due to mirroring) by holding the actual pixels from neighboring blocks in shared memory. Quan et al. [26] proposed a modification of wavelet coefficient layout to allow the intermixing of low- and high-coefficients and also allowed a register-based lifting scheme on the GPU. Most recently, Enfedaque et al. [13] exploited warp shuffling instructions and completely removed shared



(a) Haar wavelet



(b) CDF 5/3 wavelet



(c) CDF 9/7 wavelet

Fig. 2: Lifting scheme for various wavelet families.

memory by employing registers instead, which significantly increased the performance.

Most of the existing GPU DWT work, except Song et al. [25] and Enfedaque et al. [13], have not fully exploited ILP. Instead, they required the entire thread block to perform memory operations while only half of the threads participated in the lifting steps. In addition, extending Enfedaque et al. [13] to 3D volumetric data is not straightforward because the required number of per-thread registers increases sharply as the dimension increases. In our proposed work, we address these issues by exploiting ILP and TLP together.

3 Background

3.1 Wavelet lifting schemes

In this section, we briefly review the basic ideas of wavelet lifting schemes. As introduced earlier, the lifting scheme [15] is a well-known technique to implement a *second-generation* wavelet as a computationally-efficient successor of the *first-generation* wavelet based on a filter scheme. Figure 2 is a pictorial description of 1D DWT with a lifting scheme on several wavelet families, such as Haar, CDF 5/3, and CDF 9/7. Those wavelets have a biorthogonal basis, which allows us to have more freedom in designing a pair of analysis and synthesis functions (i.e., forward and inverse transforms). They are also *separable*, which means multi-dimensional DWT can be implemented by successively applying a 1D transform along each dimension.

Wavelet lifting schemes consist of multiple steps (e.g., two for Haar and CDF 5/3, four for CDF 9/7) with an optional normalization step at the end. First, the input signal is *split* into two streams of coefficients: *even* and *odd* arrays, or conventionally called by *approximation* and *detail* series, which indicate the prospective low- and high-frequencies of the wavelet spectrum. In other words, the multirate input signal is decomposed into a polyphase representation as follows:

$$a^k[n] = x^{k-1}[2n], \quad d^k[n] = x^{k-1}[2n+1] \tag{1}$$

where $a^k[n]$ and $d^k[n]$ are the approximation and detail coefficients of the successive transform level k from $k-1$, respectively. Next, the *detail* part is *predicted* from the current values of both streams (Equation 2) and the *approximation* part is then *updated* (Equation 3) accordingly:

$$d^k[n] = d^k[n] - L_P\left(a^k[n]\right) \tag{2}$$

$$a^k[n] = a^k[n] + L_U\left(d^k[n]\right) \tag{3}$$

Simply speaking, L_P and L_U (illustrated as the black arrow in Figure 2), which are the prediction and updating lifting operators in Equation 2 and Equation 3, can be represented as the local weighted sum of the lifting coefficients. Both Haar and CDF 5/3 lifting schemes use one pair of weights (α and β) while CDF the 9/7 scheme requires two pairs of weights, (α and β), and (γ and δ), as listed in Table 1. Finally, the resulting coefficients are sent to the appropriate locations in the *merge* step, with or without normalization, and one level of transform is completed (Equation 4). The inverse transform can be by applying the previous steps in reverse order with weights in opposite signs.

$$x^k[n] = a^k[n], \quad x^k[n + dim/2] = d^k[n] \tag{4}$$

TABLE 1: Lifting coefficients of selected wavelet families

	Haar	CDF 5/3	CDF 9/7
α	-1.0	-0.50	-1.58613434
β	$+0.5$	$+0.25$	-0.05298012
γ	N/A	N/A	$+0.88291108$
δ	N/A	N/A	$+0.44350685$

Note that CDF 9/7 is lossy, meaning that the complete round of forward and inverse transforms does not reproduce the input signal This is due to the non-exact lifting coefficients (see Table 1). In addition, CDF 9/7 is more computationally expensive because it requires four lifting steps, as opposed to only two steps in Haar and CDF 5/3. Figure 2 also shows that L_P and L_U of the Haar wavelet lifting transform can be done locally without reading neighboring values across the block boundary because both predicting and updating steps use a one-side support, either left or right, within a group of two pixels for a one-dimensional transform. CDF 5/3 and CDF 9/7, however, need to predict and update the current element using the neighboring values from both sides. This means that the implementation of a CDF wavelet on the GPU should follow the stencil-based computing paradigm [27] because each thread block needs to access a portion of data slightly larger than the

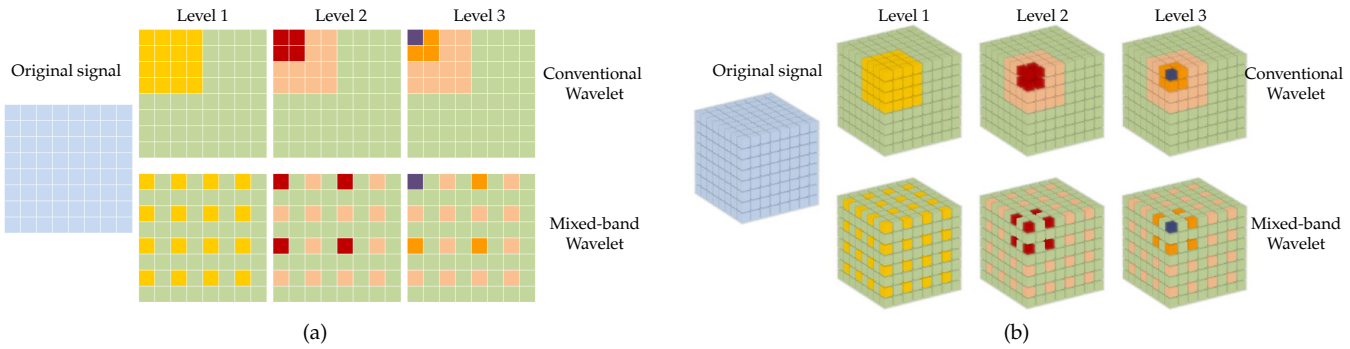


Fig. 3: Memory layouts of conventional and mixed-band wavelets in (a) 2D and (b) 3D.

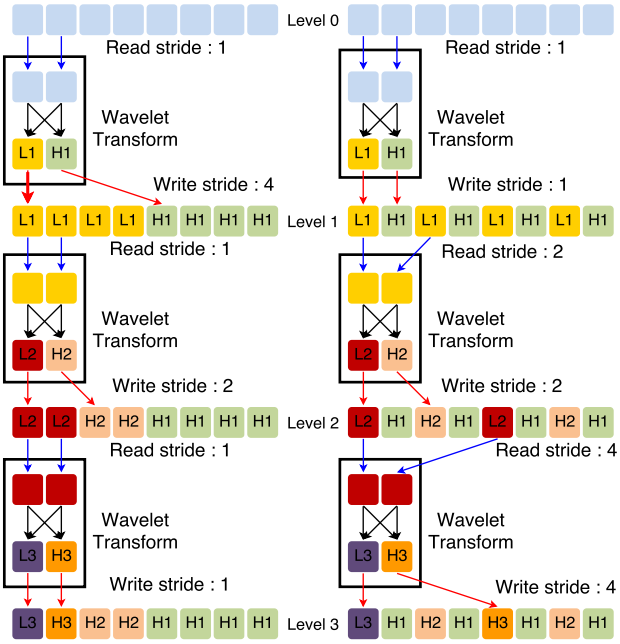


Fig. 4: Comparison of memory access patterns in conventional (left) and mixed–band (right) 1D Haar wavelet transforms.

output region, which is called *halo*. More details about block dimensions and thread assignments for handling halo will be discussed in Section 4.

3.2 Mixed-band wavelet transform

The motivation of developing a mixed-band algorithm is that the separation of different frequency bands may not be necessary for many applications as long as forward and inverse transforms work as expected. For example, the main reason that the wavelet transform is used in compressed sensing MRI reconstruction is to sparsify the image in wavelet domain and suppress close-to-zero signals. Hence, for this purpose, the location of each coefficient does not matter. Therefore, the main idea in the proposed mixed-band algorithm is to allow reading from and writing to the same memory location without rearranging the result into low- and high-frequency coefficients. This fits especially well for GPU Haar DWT because the entire single-level transform can be implemented fully *in-place* using shared memory without writing intermediate lifting results to global memory.

Figure 3 shows the memory layout of wavelet coefficients in conventional (upper rows) and mixed-band (bottom rows) wavelet transforms. In the conventional DWT (top rows in Figure 3), low- and high-frequency wavelet coefficients are spatially separated (for example, in Figure 3 (a), the upper image of Level 1, the yellow region is the low-frequency coefficients and the green region is the high-frequency coefficients). In the following level, the low-frequency region of the previous level will be used as the input to the wavelet transform. In the mixed-band wavelet, however, high- and low-frequency coefficients are inter-mixed, as shown in Figure 3's bottom rows. This unconventional layout does not alter the histogram of the wavelet coefficient values, and the inverse transform can reconstruct the input image losslessly.

In order to explain the mixed-band approach in detail, without loss of generality, we use a 1D wavelet example as shown in Figure 4. Let us assume that the input is a 1D array consisting of eight pixel values. Each wavelet transform converts the input into two half-size arrays: one stores low-frequency coefficients and the other stores high-frequency coefficients. Therefore there can be up to $\log_2(N)$ levels when N is the size of input array (there are three levels in this example). Blue arrows are reading transactions and red arrows are writing transactions. If we assume that this process runs on the GPU, then you can consider the input and output pixel arrays as global memory and the dotted box for wavelet transform as shared memory. As shown in Figure 4 left, reading and writing strides are different in the conventional wavelet transform while those are identical in the mixed-band wavelet transform. The benefits of mixed-band approach include 1) less memory usage due to in-place filtering, 2) *fused* multi-level transformation in a single kernel call without synchronizing using global memory, and 3) reducing the index computation overhead for shuffling the location of wavelet coefficients.

3.3 Bit rotation permutation

In order to convert a mixed-band wavelet layout to a conventional wavelet layout, we need to recompute the

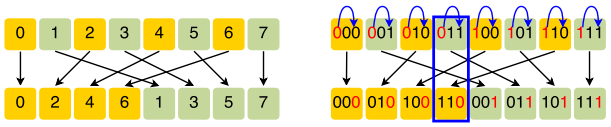


Fig. 5: Bit rotation permutation on an array of 8 pixels: decimal indexing (left) and binary indexing (right).

index of each coefficient. Specifically, after performing the lifting step, the wavelet coefficients are divided into two groups, i.e., low- and high-frequency coefficients, as shown in Figure 5. A naive method to compute a new index value involves integer division and addition operations, which affects the performance. A 2D DWT is even more expensive because we need to manage four groups of coefficients. In order to reduce this overhead, we borrow the idea of index shuffling strategy in Fourier transform to rearrange wavelet subbands more efficiently. The fast DFT introduced in [28] uses *bit reversal permutation* to rearrange the Fourier frequency bands. Similar to this, we propose a procedure to perform fast reindexing on the wavelet spectrum, i.e., *bit rotation permutation*. Figure 5 right illustrates the 3-bit rotation to-the-right of each index on an array of 8 pixels. As shown here, the index of the target location after wavelet transform can be simply computed by rotating (or shifting) source index's bits. As a result, only bitwise operations are required to calculate output indices and the computational cost is lower. For an image of arbitrary size, the number of rotation bits can be determined by evaluating $log(\lceil dim \rceil)/log(2)$ where $\lceil dim \rceil$ is rounded to the optimal value for efficient calculation (as in DFT). This pre-calculation can be performed explicitly outside GPU kernel calls.

4 GPU Optimization Strategies

In this section, various optimization strategies for GPU implementation of lifting DWT are discussed in detail, and an optimal implementation strategy using hybrid parallelism will be proposed at the end.

4.1 Using shared memory

Shared memory is commonly used in GPU programs to reduce the longer memory latency of global memory (DRAM), and it is effective if data is reused multiple times. Another important role of shared memory is to serve as a communication medium between threads. The lifting scheme requires multiple computation steps, in which each step updates only a half of the input data (either even- or odd-indexed data), and newly updated data is used as an input to the next update step. Figure 7 (a) shows CDF 5/3 lifting wavelet implementation using shared memory for interthread communication. In the figure, red arrows represent shared memory read/write transactions and gray dotted arrows represent no memory transaction. Because only even- or odd-indexed values are updated concurrently at any given lifting step, there is no read-write conflict. Shared memory is even more effective for wavelets with large supports, such as CDF 9/7, because many update iterations must be performed.

Special care should be taken when data is copied from global memory to shared memory, since global memory access is an expensive operation on the GPU. Most wavelet bases other than Haar have a large support that extends beyond the computing domain, which is called *halo*. Due to halo, the tile dimension is not aligned with that of the thread block. For example, as shown in Figure 6, for a $\{32, 16\}$ thread block, the required tile size including the halo for CDF 5/3 will be $\{35, 19\}$ because the halo size along one axis is 3 (2 for left/top, and 1 for right/bottom). In order to load the entire tile region from global memory, we first linearize 2D thread indices to 1D, and then assign threads to appropriate global memory locations in a row-major order (Figure 6). Because our tile dimension is not a multiple of 128 Bytes (L1 cache line size), we disable L1 cache and use the non-caching global memory load to reduce the memory transaction granularity to 32 Bytes. The entire tile can be copied from global to shared memory in two loading steps for 512 threads (only 153 threads participate the data copy in the second step).

4.2 Using registers

In this optimization, we reduce shared memory transactions by loading necessary neighboring values to registers and perform computation only on registers. The main idea behind is that register's memory bandwidth is much higher than that of shared memory even though both are on-chip memory [12], so reducing shared memory transactions is beneficial although more registers are required.

Figure 7 (a) is a shared memory-based CDF 5/3 implementation, in which in-place computation of the lifting scheme is done via writing back and forth from shared memory. This is a commonly used technique for interthread communication, but we can optimize even further by using registers as shown in Figure 7 (b), in which registers are used as a temporary per-thread buffer to hold the result of each lifting step computation. As shown in the illustration, the total number of shared memory transactions (drawn in red arrows) is reduced in the register-based implementation. The shared memory-based implementation shown in Figure 7 (top) (a) requires 6 reads (two sets of three reads in lifting with α are done in parallel), 3 writes to shared memory and two `__syncthreads()` calls. But the register-based implementation shown in Figure 7 (bottom) requires only 5 reads and 2 writes of shared memory per thread with only one `__syncthreads()` call at the end. Although there is extra data copy between registers in the register-based implementation, we observed an overall performance improvement over shared memory-based implementation about 20%. Note that the strategy described here does not exploit instruction level parallelism (ILP) yet. It can further improve the performance as shown in the following sections.

4.3 Exploiting Instruction level parallelism (ILP)

In the previous optimization strategies, about a half of GPU cores are idling during DWT lifting steps. We can further reduce that inefficiency by leveraging ILP, which reduces the number of threads and let each thread do more work. For example, if we reduce the thread block size by half, then

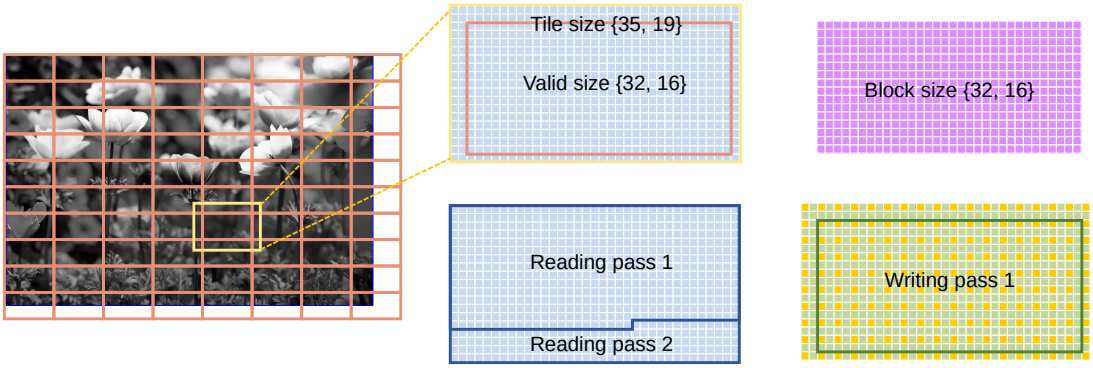


Fig. 6: Region of reading and writing on using only shared memory strategy.

the thread block can cover the same output region without idling by doubling each thread's workload. Therefore, all the GPU cores will participate in each even- or odd-lifting step.

The ILP setup for CDF 5/3 is shown in Figure 8 where the thread block size is {32, 16} and the tile size, which is equivalent to the shared memory size, is {64, 32}. In this setup, the effective output size becomes {60, 28} due to the halo of size 2 on each side. Note that in this setup we match the shared memory dimension to a multiple of thread block dimension, which is different from the shared memory strategy. Each thread covers a {2, 2} region, which is 4× more memory transactions and 2× more lifting computations per thread. For the CDF 9/7 DWT case, we can keep the same thread block and tile dimension by reducing the output size to {56, 24} because the halo size is 4 on each side. The thread block size is then tuned empirically by fixing the x dimension to 32 and testing various y dimensions from 2 to 60. In our experiment, 20 produced the best result, i.e., 20 warps per a thread block.

Note that in this approach we did not use a register technique. Even so, we observed about 5× performance boost compared to the shared memory-based approach introduced in Section 4.1, which is due mainly to hiding memory and instruction latency and reducing unnecessary idling in the lifting steps.

4.4 Exploiting warp shuffles on Kepler GPUs

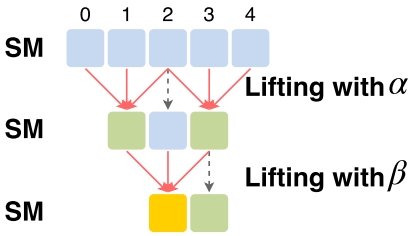
Recent NVIDIA GPUs (Kepler and later) support warp shuffle instructions, which allow for the rapid exchange of data between threads in a same warp. Currently, four warp shuffle instructions are provided: `__shfl_up`, `__shfl_down`, combined with `__shfl_xor`, and `__shfl` [10]. Using warp instructions, a thread can directly read another thread's register values without explicitly exchanging them via shared memory. In the shared memory-based strategy introduced above, each lifting step was required to write the intermediate result back to shared memory so that neighboring threads can use it for the next lifting step. Therefore, by organizing the tile size as a multiple of warp size and declaring registers to hold the intermediate pixel values, we can eliminate expensive synchronization via shared memory in lifting steps.

In order to exploit warp shuffles, we need to align the warp and the lifting direction, e.g., a horizontal warp for horizontal lifting step (Figure 9 left). In addition, the y dimension of the tile size should be a multiple of a warp size so that a warp can process columns without thread idling. In our setup, the tile size is {64, 32} and the thread block size is {32, 32}. As shown in Figure 9, a warp processes one row for a horizontal lifting and two columns for a vertical lifting so that each thread is processing {2, 1} output regions. In each lifting step, *every* thread must read one register value from its neighboring thread using a warp shuffling instruction, so we call this strategy *Full-Shuffles*. Note that this warp shuffle can eliminate shared memory access during lifting steps, but all the data must be written back to shared memory twice: one is between horizontal and vertical lifting and the other is immediately before the final output is written to global memory. The former instance is for inter-warp communication, and the latter instance is for coalesced global memory writes.

4.5 Combining all: hybrid approach

In this hybrid approach, we combine all the previously discussed techniques, such as leveraging shared memory, registers, warp shuffles and ILP to maximally hide the latency of memory operations. This method declares more registers per thread, e.g., an array of 4 by 2, and uses a {32, 8} thread block (8 warps) to cover a {64, 32} tile. Hence, there are 8× of memory operation-ILP (MILP) and 4× of computation-ILP (CILP). Regarding the lifting steps during the horizontal pass, it operates similar to *Full-Shuffles*. The slight difference is that each thread must perform DWT 4× the amount of work compared to *Full-Shuffles*. Thereafter, the intermediate result of the horizontal transform is written back into shared memory so that warps can later access the vertical pixels in aligned orders.

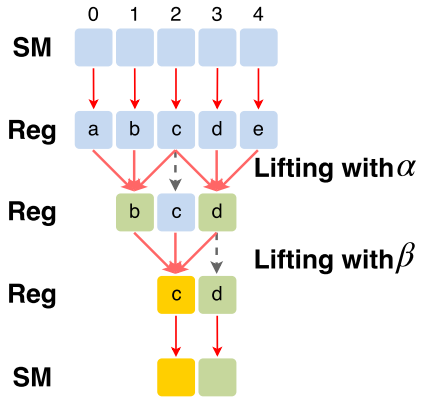
Subsequently, the lifting steps along the vertical pass will take place as a combination of registers and warp shuffles. The computation, which happens inside each thread, requires register-read only (Figure 9 center, orange arrows). At the top or bottom of the thread's array, neighboring pixel values must be fetched via warp shuffles (Figure 9 center, red arrows). This configuration is called *hybrid* or *Semi-Shuffles*. In fact, the degree of ILP can be modified by adjusting the *number of warps per tile*. Several variants of *hybrid* will be discussed in Section 5. Further, when the number of warps per tile is reduced to one and each thread holds enough registers to proceed lifting steps vertically (an




```
if((threadIdx.x&1)==0) //Encode lifting with alpha
    sharedMem[at(index_2d.x + 1, index_2d.y)]
    -= alpha*(sharedMem[at(index_2d.x + 0, index_2d.y)]
              +sharedMem[at(index_2d.x + 2, index_2d.y)]);
__syncthreads();

if((threadIdx.x&1)==0) //Encode lifting with beta
    sharedMem[at(index_2d.x + 2, index_2d.y)]
    += beta *(sharedMem[at(index_2d.x + 1, index_2d.y)]
              +sharedMem[at(index_2d.x + 3, index_2d.y)]);
__syncthreads();
```

(a) Shared memory-based approach




```
if((threadIdx.x&1)==0){
    a = sharedMem[at(index_2d.x + 0, index_2d.y)];
    b = sharedMem[at(index_2d.x + 1, index_2d.y)];
    c = sharedMem[at(index_2d.x + 2, index_2d.y)];
    d = sharedMem[at(index_2d.x + 3, index_2d.y)];
    e = sharedMem[at(index_2d.x + 4, index_2d.y)];
    //Encode lifting with alpha
    b -= alpha*(a+c);
    d -= alpha*(c+e);
    //Encode lifting with beta
    c += beta *(b+d);
    sharedMem[at(index_2d.x + 2, index_2d.y)] = c;
    sharedMem[at(index_2d.x + 3, index_2d.y)] = d;
}
__syncthreads();
```

(b) Register-based approach

Fig. 7: Two lifting scheme implementations of biorthogonal CDF 5/3 wavelet and their NVIDIA CUDA code snippets.

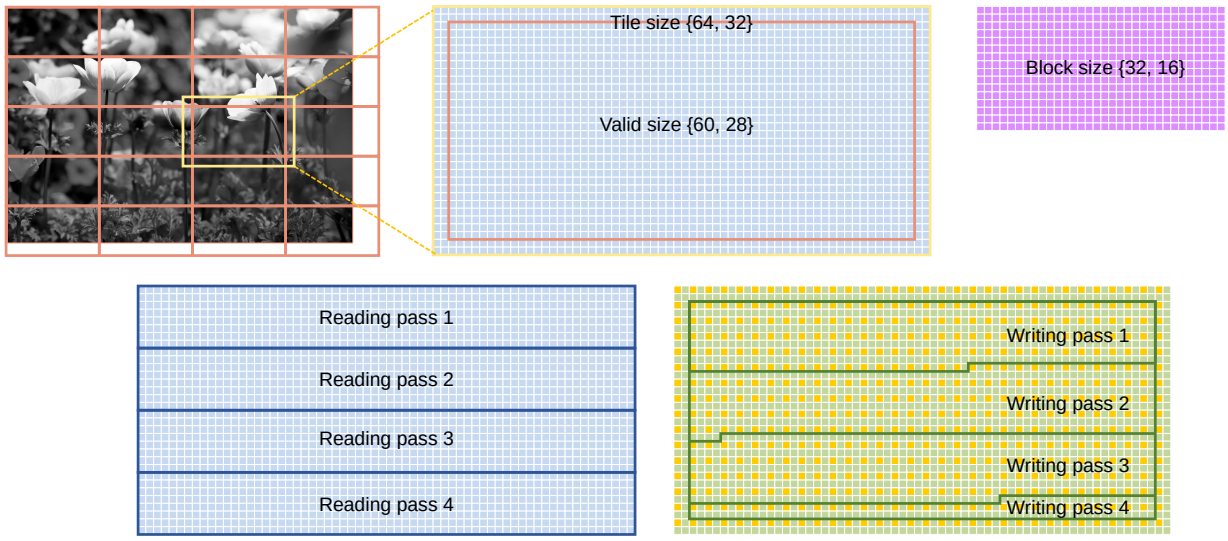


Fig. 8: Region of reading and writing on using instruction level parallelism strategy.

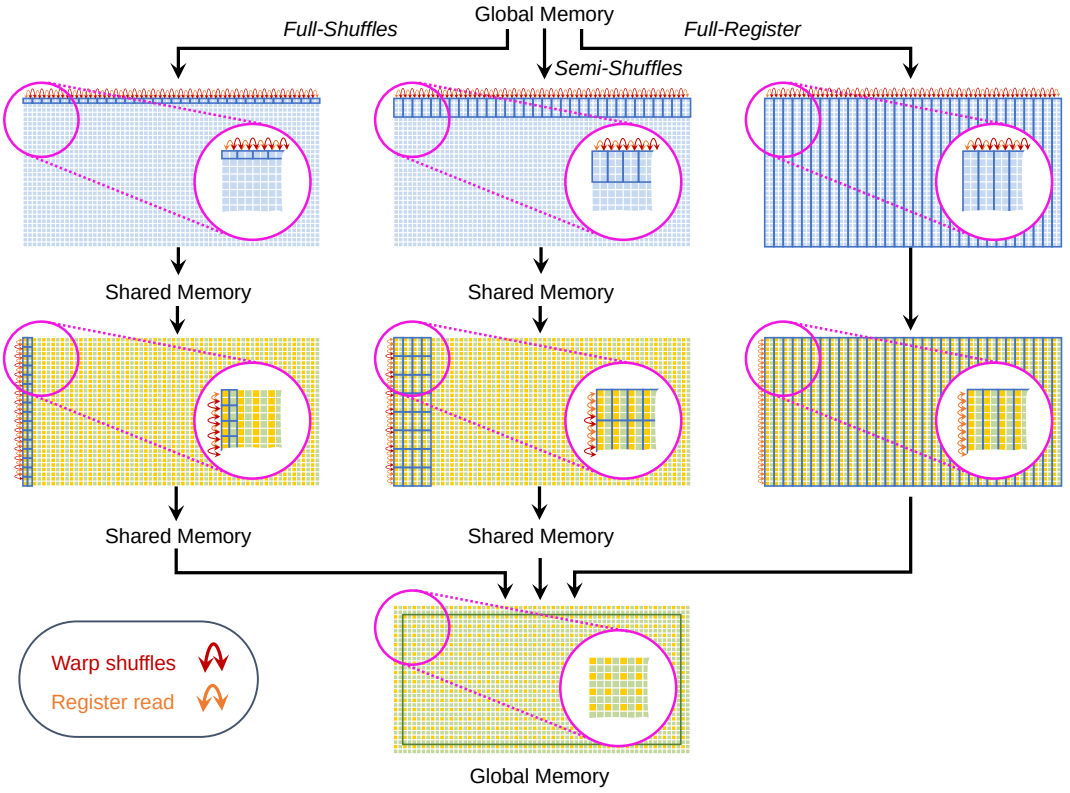


Fig. 9: Three variants of hybrid parallelism of DWT: *Full-Shuffles*, *Semi-Shuffles* and *Full-Register*.

array of 32 by 2), this will be similar to the setup used by Enfedaque et al. [13]. This particular setup is named *Full-Register* (Figure 9 right). Note that in *Full-Register*, shared memory is completely removed and more warps can be grouped to form a single block.

4.6 Fused multi-level Haar DWT

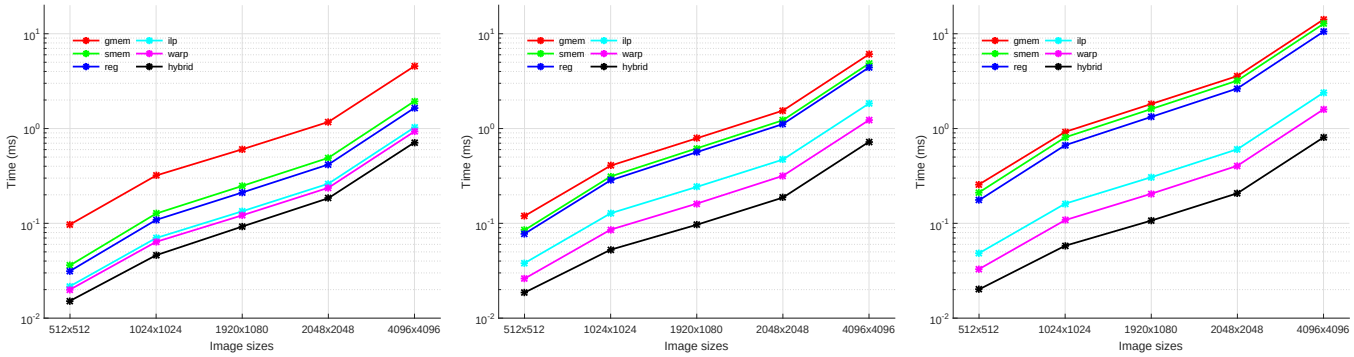
As mentioned earlier, Haar DWT has only one-side local supports and its lifting-scheme is completely fit onto CUDA blocks because halos across the block boundary are not needed. If we modify the memory layout of the conventional Haar DWT to that of the mixed-band approach, multiple levels of Haar wavelet can be processed in a single kernel call, i.e., *fused* multi-level transformation. The main idea behind this is that the mixed-band approach allows Haar DWT to be processed in-place, where a group of four pixel is read, lifted and written onto the same memory location. Therefore, once a chunk of processing block is loaded to shared memory, wavelet transform can be applied on shared memory iteratively without writing intermediate results back to global memory. The first level of transform takes place as normal mixed-band Haar DWT where implementation is exactly same as the hybrid approach except we do not reshuffle the location of wavelet coefficients. In the following levels, the method simply reads the appropriate coefficients from shared memory, performs lifting steps, and stores the result back to shared memory. Note that ILP will decrease after each level because less number of coefficients need to be processed. In addition, there is a limit for the tile size per block, so the maximum DWT level should be chosen appropriately. In our implementation, we chose four levels of transformation to be fused into a single kernel call. The details of the experimental results will be discussed in Section 5.5.

5 Results and Discussions

5.1 Comparison of various strategies

We verified the performance of the proposed method by performing one level of 2D DWT on images of different sizes. We selected a computer equipped with an NVIDIA GK110 GPU to collect the running times for all measurements. Note that in this evaluation, all of the pre- and post-transfers from CPU to GPU (and vice versa) were omitted (only the kernel timing was taken into account). We conducted the experiment using various strategies, in both directions (analysis and synthesis kernels, and then averaging the running times) and on three commonly distinct orthogonal wavelet families (Haar, CDF 5/3 and CDF 9/7). Each strategy was tested multiple times (at least 200 times) to measure the average running time. The proposed optimization strategies are summarized as follows:

Using global memory only (*gmem*): In this strategy, global memory was used to hold all buffers and perform the 2D DWT without considering any support of the GPU's on-chip memory (i.e., shared memory, cache, or registers). This approach is a naive GPU implementation, and shows a moderate increase in speed compared to a serial CPU implementation. The results of using only global memory were considered as the baseline to evaluate the efficiency of other optimization techniques.



(a) Haar DWT (b) CDF 5/3 DWT (c) CDF 9/7 DWT

Fig. 10: Running times (in msecs) of various strategies, on NVIDIA Kepler GPUs.

Using shared memory (*smem*): It is essential to move all of the frequently accessed data to a common place, which is shared by a thread block, in order to increase the performance. All the threads within a block copy a tile of the image (plus halo) from global memory and transfer it to shared memory. This helps to reduce the tremendous cost of accessing global memory.

Using registers (*reg*): Instead of invoking `__syncthreads()` after each level of a lifting step in shared memory as above, registers are used for inter-thread communication within a block. This approach allows each GPU thread to work more independently of its neighbors without synchronization.

Using instruction level parallelism (*ilp*): The total number of threads is reduced by a factor of 4 and each of them performs more work on either memory operations or wavelet lifting steps. In this case, the instruction pipeline suffers less from stalls, which leads to higher computational throughput. In addition, it allows thread blocks to leverage more on-chip resources such as registers and shared memory / L1 caches.

Using warp shuffles (*warp*): With the thread block configured as $\{32, 32\}$ on a $\{64, 32\}$ tile, the input pixels are first read directly from global memory to registers (declared as an array of 2 by 1 per thread) without `__syncthreads()`. The horizontal pass can proceed with the support of warp shuffles and save the temporary results into shared memory. Then 32 warps can process 64 columns in the vertical pass to complete one level of DWT.

Using hybrid method (*hybrid*): We further extended our strategy to the hybrid parallelism which is a fusion of using registers, warp shuffles, TLP and ILP together. The degree of TLP/ILP was chosen empirically after testing potential configurations in order to have the best choice of representation (see Section 5.2).

As shown in Table 2 and illustrated in Figure 10, the running times of various strategies (measured in milliseconds, visualized on a logarithmic scale) are decreasing in the order of approaches (*gmem*, *smem*, *reg*, *ilp*, *warp*, *hybrid*). The results show that those steps we followed leads to a better DWT performance on the GPU.

TABLE 2: Running times (in msecs) of various optimization strategies for CDF 9/7 DWT on NVIDIA Kepler GPUs

CDF 9/7	*gmem*	*smem*	*reg*	*ilp*	*warp*	*hybrid*
512x512	0.2572	0.2119	0.1751	0.0484	0.0329	0.0201
1024x1024	0.9219	0.8072	0.6667	0.1612	0.1081	0.0580
1920x1080	1.8157	1.6061	1.3269	0.3065	0.2052	0.1069
2048x2048	3.5802	3.2048	2.6476	0.6046	0.4049	0.2076
4096x4096	14.1514	12.7280	10.5169	2.3916	1.5946	0.8066

TABLE 3: Different block and tile configurations of *hybrid* approach.

TLP \ILP	1	4	8	16	32
32×1	64×1	64×4	64×8	64×16	64×32
(32×4)	–	–	–	64×40	64×104
32×4	64×4	64×8	64×16	64×32	64×64
32×8	64×8	64×16	**64×32**	64×64	64×128
32×16	64×16	64×32	64×64	64×128	64×256
32×32	64×32	64×64	64×128	64×256	64×512

TABLE 4: Running times (in msecs) of *hybrid* for CDF 9/7 DWT on an NVIDIA Kepler GPU, image size 1024×1024.

TLP \ILP	1	4	8	16	32
32×1	–	–	–	0.08728	0.23954
32×4	–	–	0.08594	0.06403	0.11589
32×8	–	0.09809	**0.06033**	0.08215	0.14014
32×16	0.14144	0.06940	0.07224	0.12643	–
32×32	0.11320	0.07807	0.10729	–	–

TABLE 5: Running times (in msecs) of *hybrid* for CDF 9/7 DWT on an NVIDIA Kepler GPU, image size 1920×1080.

TLP \ILP	1	4	8	16	32
32×1	–	–	–	0.16205	0.45436
32×4	–	–	0.16094	0.11829	0.21642
32×8	–	0.18526	**0.11145**	0.15333	0.26044
32×16	0.27010	0.12919	0.13411	0.23910	–
32×32	0.21490	0.14499	0.20212	–	–

TABLE 6: Running times (in msecs) of CPU implementations (MATLAB and GSL) and various optimization strategies for CDF 9/7 DWT on an NVIDIA Kepler GPU (including data transfer time).

CDF 9/7	*gsl*	*matlab*	*gmem*	*smem*	*reg*	*ilp*	*warp*	*hybrid*
512x512	3.309	22.600	0.539	0.501	0.491	0.372	0.380	0.356
1024x1024	16.911	91.800	2.222	2.172	2.085	1.589	1.566	1.539
1920x1080	-	167.400	4.632	4.345	4.308	3.325	3.310	3.195
2048x2048	132.794	349.700	9.051	8.846	8.531	6.614	6.416	6.389
4096x4096	635.031	1202.800	35.222	34.090	33.311	25.779	24.961	24.776

5.2 Comparison with different hybrid configurations

In this section, we show that combining the advantages of TLP (by increasing number of threads) and ILP (by assigning more work per thread) will result in a better DWT performance. As mentioned earlier, the degree of ILP can be adjusted on the basis of the amount of work eligible for one thread, which is the interpolation of *Full-Shuffles* and *Full-Register*. Table 3 specifies the sizes of thread block and the amount of computational work per thread on DWT CDF 9/7, where the position from left to right indicates an increasing degree of ILP (or a decreasing degree of TLP), and the position from top to bottom indicates the thread block size. Note that some of the configurations (in gray) will be invalid due to either the number of necessary pixels along the vertical axis for a valid tile or the hardware limitations (shared memory resource per block). The first row indicates the configuration of the *Full-Register* without the involvement of shared memory. We can assign either one or more warps per blocks to increase the occupancy. The first column covers the *Full-Shuffles* case where 16 warps handle a {64, 16} tile and 32 warps handle a {64, 32} tile. The other configurations belong to the *Semi-Shuffles* where lifting along y is a mixture between intra-thread read and intra-warp data exchange.

We conducted the experiment on various image sizes, e.g., 512×512, 1024×1024 (Table 4), 1920×1080 (Table 5), 2048×2048 and 4096×4096, and observed that the best configuration was using eight warps per block while letting each thread do eight times the amount of computation, i.e., 32×8 block size and 64×32 tile size (marked in boldface).

5.3 Comparison with CPU implementations

In this evaluation, we compare the proposed GPU DWT with several existing CPU DWT implementations to see how much speed up can be achieved from the GPU. The wall-clock running times of each method, which include all the data transfer overhead on GPU implementation for fair comparison, are collected. We chose the DWT implementations available in MATLAB 2015b [29] and GNU Scientific Library (GSL), version 1.16 [30] for this experiment. Table 6 lists the running time of each method measured on various image sizes for one level of CDF 9/7 DWT. As shown in this table, our *hybrid* approach running on a single NVIDIA Kepler GPU K40 (745 MHz) outperforms *gsl* and *matlab* running on an Intel i7-4790K CPU (4.20 GHz) by a large margin – our GPU DWT achieved up to 25× and 65× speed up, respectively.

5.4 Comparison with other GPU DWT methods

On NVIDIA Fermi architecture

We compared our method with the most recent GPU wavelet work using Fermi GPUs [25] and its references therein. In other to make direct comparisons, we ran our CDF 9/7 DWT on an NVIDIA GF100 GPU (Tesla C2050) and compared our results with the results of methods from [25].

To be more specific, Matela [19] and our early work [26] used a symmetric block-based approach, which disregards the halo across a block boundary, and hence acts as an approximation of the classical filter scheme. The other methods, including our proposed method, are an exact implementation of non-Haar DWT. Among those, Franco et al. [20], Song et al. [24], and Laan et al. [22] split the horizontal and vertical pass into two cascaded stages, which differs in the handling of the intermediate result before performing the vertical transform. Franco et al. [20] included the transposition after each pass so that on the next stage of wavelet transform, data would already be laid out the same as on the horizontal pass. Song et al. [24] did not have the transposition step, but instead processed the vertical pass in column segment fashion. As showed in Table 7, Franco et al. [20] was faster than Song et al. [24] only on small image (512×512) where the data was actively cached on the transpose operation. It suffered from the memory bottleneck, however, due to the data movement for larger images. Laan et al. [22] proposed a slab approach (sliding window) to achieve a fast column transform where enough data would be stored in the top and the bottom of the slab, and multi-column could be processed at the same time to maximize the use of shared memory. Overall, those three methods involved expensive global memory in-between and that impaired their performance.

The recent improvement from Song et al. [25] employed the block-based (or tiling) technique, which is same as ours, to eliminate the intermediate results being stored in global memory. This approach performs a two-pass transform in one kernel call, and hence establishes a 2D stencil-based processing procedure. Song et al. also implicitly used the ILP where the block was configured as {60, 1} and the tile was set as {60, 32}. Equivalently, they assigned 2 warps (64 threads) for one tile of the image and hence 4 threads out of 64 were idling. Note that this approach did not fully leverage the GPU resource when the block size was not divisible by the warp size (32), which left more room to optimize further.

Our *ilp* approach is faster (Table 7) than the recent method from Song et al. [25] because we employed ILP and used fast bitwise computation to calculate the subband's

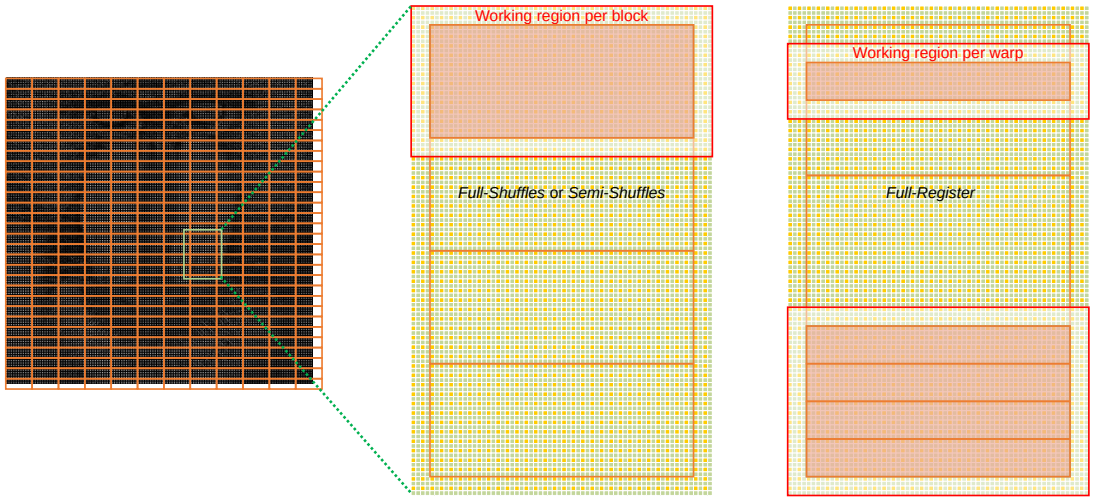


Fig. 11: Overlapping regions of *Full-Shuffles* or *Semi-Shuffles*, compared to those of *Full-Register*.

TABLE 7: Running times (in msecs) of other GPU DWT methods and the proposed methods for CDF 9/7 DWT.

CDF 9/7	GF100 (Fermi)						GK110 (Kepler)	
	Matela [19]	Franco et al. [21]	Song et al. [24]	Laan et al. [23]	Song et al. [25]	*ilp*	Enfedaque et al. [13]	*hybrid*
512 × 512	0.51	0.35	0.46	0.25	0.16	0.107	0.0545	0.0204
1024 × 1024	1.31	1.08	1.05	0.66	0.46	0.387	0.1452	0.0594
1920 × 1080	2.36	2.14	1.79	1.21	0.86	0.747	–	0.1114
2048 × 2048	4.51	3.86	2.54	1.98	1.68	1.489	0.4812	0.2227
4096 × 4096	17.25	16.38	14.15	7.18	6.44	5.924	1.7919	0.8678

locations. It also showed that the advantages of giving more work per thread will lead to better performances. Because we just need to store the valid coefficients (neglect the halo) to global memory, there are still idling threads during the last pass of writing (see Figure 8). The ratio between the active and the total threads of our method is 88.5%, however, which is much higher than 79.4% from Song et al. [25].

On NVIDIA Kepler architecture

We also compared our method with the most recent register-based GPU DWT algorithm [13]. In order to make a direct comparison, we downloaded their source code from the web repository (available at [31]) and compared it with our *hybrid* version on an NVIDIA K40 GPU. Again, one level of CDF 9/7 DWT was used as a test case to stress the computing power of the GPU. As shown in Table 7, our *hybrid* method consistently outperformed Enfedaque et al. [13].

The main reason behind that result is that once ILP is saturated and the maximum performance is reached, then assigning more work to a thread cannot increase the performance. Since ILP and TLP are inversely proportional to each other for the fixed problem size, if ILP is increased then TLP is decreased, and vice versa. Just maximizing ILP would hurt TLP significantly. In [12], Volkov et al. showed that maximizing ILP can result in an optimal performance even though occupancy becomes very low, i.e., the degree of TLP is low. Along this line, Enfedaque et al. [13] introduced a GPU DWT that maximized ILP using registers only. As shown in the recent study by Fatehi and Gratz [32], however, there exists an upper bound of ILP for total memory and computation instructions. Therefore, we believe that combining TLP with ILP can be more effective because memory and instruction latency can be further hidden by concurrent threads once ILP reaches its peak. We also believe that there should be enough concurrent warps running on a Streaming Multiprocessor (SMX) of the GPU for optimal performance. For example, on NVIDIA Kepler architecture, up to two independent instructions from each of the four concurrent warps can be issued per clock cycle [33], which makes the setup of eight warps per block in the *hybrid* optimization fit better to Kepler GPUs than did the four warps per block in Enfedaque et al [13].

Another important point is that our *hybrid* method used both warp shuffles and registers for vertical (along y axis) lifting steps, which allowed fewer overlapping regions than Enfedaque et al [13]. For example, consider the case of performing CDF 9/7 DWT on an image: its $\{64, 96\}$ portion must use four blocks (each has 8 warps) to proceed (as *Full-Shuffles* or *Semi-Shuffles*). This results in a ratio of the overlapping halo region to the output size of 45.83%. By comparison, *Full-Registers* requires three blocks (each has 4 warps) and hence, has 112.5% of the overlapping ratio. This is mainly because the working area of *Full-Shuffles* or *Semi-Shuffles* is per-block while *Full-Registers* or Enfedaque et al. [13] is per-warp.

5.5 Comparison on multi-level GPU DWT

In this section, we assess the actual application-level running time of the proposed GPU DWT for multi-level transformation to see how the proposed method performs under realistic conditions. Table 8 shows the running times of

TABLE 8: Running times (in msecs) of [13] and *hybrid* for multi-level CDF 9/7 DWT, on an NVIDIA Kepler GPU.

CDF 9/7	Enfedaque et al. [13]				*hybrid*			
level	1	2	3	4	1	2	3	4
512x512	0.0545	0.0835	0.1107	0.1364	0.0204	0.0407	0.0544	0.0670
1024x1024	0.1452	0.2006	0.2259	0.2533	0.0594	0.1015	0.1197	0.1334
2048x2048	0.4812	0.6251	0.6770	0.7064	0.2227	0.3505	0.3878	0.4057
4096x4096	1.7919	2.2756	2.4170	2.4733	0.8678	1.3525	1.4717	1.5086

TABLE 9: Running times (in msecs) of *hybrid* and *mb-hybrid* for multi-level Haar DWT, on an NVIDIA Kepler GPU.

Haar	*hybrid*				*mb-hybrid* (fused kernel)			
level	1	2	3	4	1	2	3	4
512x512	0.0150	0.0298	0.0414	0.0524	0.0129	0.0152	0.0182	0.0192
1024x1024	0.0458	0.0753	0.0901	0.1016	0.0333	0.0417	0.0522	0.0560
2048x2048	0.1810	0.2810	0.3106	0.3251	0.1139	0.1472	0.1888	0.2039
4096x4096	0.7162	1.1165	1.2181	1.2479	0.4470	0.5690	0.7336	0.7934

CUDA kernels measured using a wall-clock timer for up to four levels of CDF 9/7 DWT in our *hybrid* approach and Enfedaque et al. [13]. We used test images at various resolutions ranging from 512×512 to 4096×4096. For accuracy, we ran the same test multiple times and collected the average running time. Each test includes running times for both forward and inverse transformations. As shown here, our proposed method achieved higher performance compared to the state-of-the-art method [13] measured on an NVIDIA Kepler GPU (K40). We observed that our *hybrid* method runs up to 2.6× faster than Enfedaque et al., and its performance gap is slowly reduced as the number of transform levels and image size increase.

Table 9 shows the running times of our previous hybrid optimization on the conventional memory layout (*hybrid*) and the mixed-band layout (*mb-hybrid*) for fused multi-level Haar DWT. In the *hybrid* approach, the transformation result at every level is written to global memory, but the *mb-hybrid* approach transforms the input image up to four levels without accessing global memory. This results in a significant performance improvement, up to 2.7× speed up over the *hybrid* approach that is the most optimized version for single level wavelet transformation.

6 Conclusion

In this paper, we introduced various optimization strategies for 2D discrete wavelet transforms on the GPU. The proposed strategies leverage fast on-chip memories (shared memory and registers), warp shuffle instructions, and thread- and instruction-level parallelism. We showed that, unlike other state-of-the-art GPU DWTs, hybrid parallelism that exploits both ILP and TLP together results in the most optimal performance. We also showed that the mixed-band layout of Haar DWT outperformed the conventional DWT especially when multi-level transformation is taken into account. For future work, we plan to apply our proposed GPU DWT to various wavelet applications, e.g., compressed sensing MRI reconstruction and sparse coding using dictionary learning, and investigate the performance benefits in large-scale data processing applications.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments on the paper. This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development), the R&D program of MOTIE/KEIT (No. 10054548, Development of Suspended Heterogeneous Nanostructure-based Hazardous Gas Microsensor System), and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2058773). The authors would like to thank NVIDIA for their hardware support via NVIDIA GPU Research Center Program.

References

[1] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG 2000 still image compression standard," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 36–58, 2001.

[2] J. N. Bradley, C. M. Brislawn, and T. Hopper, "FBI wavelet/scalar quantization standard for gray-scale fingerprint image compression," vol. 1961, pp. 293–304, 1993.

[3] S. Mallat and W. Hwang, "Singularity detection and processing with wavelets," *IEEE Transactions on Information Theory*, vol. 38, pp. 617–643, Mar. 1992.

[4] M. Lustig, D. Donoho, J. Santos, and J. Pauly, "Compressed sensing MRI," *IEEE Signal Processing Magazine*, vol. 25, no. 2, pp. 72–82, 2008.

[5] S. Mallat, *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way*. Academic Press, 3rd ed., 2008.

[6] I. Daubechies, *Ten Lectures on Wavelets*. CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, 1992.

[7] C. Desmouliers, E. Oruklu, and J. Saniie, "Discrete wavelet transform realisation using run-time reconfiguration of field programmable gate array (FPGA)s," *IET Circuits, Devices Systems*, vol. 5, pp. 321–328, July 2011.

[8] C.-H. Hsia, J.-S. Chiang, and J.-M. Guo, "Memory-efficient hardware architecture of 2-d dual-mode lifting-based discrete wavelet transform," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, pp. 671–683, Apr. 2013.

[9] G. Bernabe, J. Cuenca, L. P. Garca, and D. Gimnez, "Improving an autotuning engine for 3d fast wavelet transform on manycore systems," *The Journal of Supercomputing*, vol. 70, pp. 830–844, Nov. 2014.
[10] http://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[11] http://www.khronos.org/opencl/.
[12] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–11, Nov. 2008.
[13] P. Enfedaque, F. Auli-Llinas, and J. Moure, "Implementation of the DWT in a GPU through a register-based strategy," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2014.
[14] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693, July 1989.
[15] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM J. Math. Anal.*, vol. 29, pp. 511–546, Mar. 1998.
[16] J. Sole and P. Salembier, "Generalized lifting prediction optimization applied to lossless image compression," *IEEE Signal Processing Letters*, vol. 14, pp. 695–698, Oct. 2007.
[17] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, and F. Tirado, "Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 299–310, Mar. 2008.
[18] http://www.opengl.org/.
[19] J. Matela, "GPU–Based DWT acceleration for JPEG2000," *Annual Doctoral Workshop On Mathematical and Engineering Methods in Computer Science*, pp. 136–143, 2009.
[20] J. Franco, G. Bernabe, J. Fernandez, and M. Acacio, "A parallel implementation of the 2D wavelet transform using CUDA," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 111–118, 2009.
[21] J. Franco, G. Bernab, J. Fernndez, and M. Ujaldn, "Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs," *Procedia Computer Science*, vol. 1, pp. 1101–1110, May 2010.
[22] W. van der Laan, J. B. T. M. Roerdink, and A. Jalba, "Accelerating wavelet-based video coding on graphics hardware using CUDA," in *Proceedings of 6th International Symposium on Image and Signal Processing and Analysis, 2009. ISPA 2009*, pp. 608–613, 2009.
[23] W. van der Laan, A. Jalba, and J. B. T. M. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 132–146, 2011.
[24] C. Song, Y. Li, and B. Huang, "A GPU-accelerated wavelet decompression system with SPIHT and reed-solomon decoding for satellite images," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, pp. 683–690, Sept. 2011.
[25] C. Song, Y. Li, J. Guo, and J. Lei, "Block-based two-dimensional wavelet transform running on graphics processing unit," *IET Computers Digital Techniques*, vol. 8, pp. 229–236, Sept. 2014.
[26] T. M. Quan and W.-K. Jeong, "A fast mixed-band lifting wavelet transform on the GPU," in *IEEE International Conference on Image Processing*, pp. 1238–1242, Oct. 2014.
[27] P. Micikevicius, "3d finite difference computation on GPUs using CUDA," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, (New York, NY, USA), pp. 79–84, ACM, 2009.
[28] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.
[29] http://www.mathworks.com/products/wavelet/.
[30] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi, *GNU Scientific Library: Reference Manual*. Network Theory Ltd., Feb. 2003.
[31] http://github.com/PabloEnfedaque/.
[32] E. Fatehi and P. Gratz, "ILP and TLP in Shared Memory Applications: A Limit Study," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 113–126, ACM, 2014.
[33] http://docs.nvidia.com/cuda/kepler-tuning-guide/.

Tran Minh Quan is currently a PhD Student in the school of electrical and computer engineering at UNIST, Korea. His research interests are GPU computing and biomedical image processing. He received his BS degree in Electrical Engineering at KAIST, Korea.

Won-Ki Jeong is currently an associate professor in the school of electrical and computer engineering at UNIST. Before joining UNIST, he was a research scientist in the Center for Brain Science at Harvard University. His research interests include scientific visualization, image processing, and general purpose computing on the graphics processor in the field of biomedical image analysis. He received a Ph.D. Degree in Computer Science from the University of Utah in 2008, and was a member of the Scientific Computing and Imaging (SCI) institute. He is a recipient of the NVIDIA Graduate Fellowship in 2007, and is currently the PI of the NVIDIA GPU Research Center at UNIST.