Reducing Memory Usage by the Lifting-based Discrete Wavelet Transform with a Unified Buffer on a GPU

Takuya Ikuzawa[a,b], Fumihiko Ino[a,*], Kenichi Hagihara[a]

[a]*Graduate School of Information Science and Technology, Osaka University, 1-5 Yamada-oka, Suita, Osaka 565-0871, Japan*
[b]*Industrial ICT Solutions Company, Toshiba Corporation, 72-34 Horikawa-cho, Saiwai-ku, Kawasaki, Kanagawa 212-0013, Japan*

Abstract

In this study, to improve the speed of the lifting-based discrete wavelet transform (DWT) for large-scale data, we propose a parallel method that achieves low memory usage and highly efficient memory access on a graphics processing unit (GPU). The proposed method reduces the memory usage by unifying the input buffer and output buffer but at the cost of a working memory region that is smaller than the data size n. The method partitions the input data into small chunks, which are then rearranged into groups so different groups of chunks can be processed in parallel. This data rearrangement scheme classifies chunks in terms of data dependency but it also facilitates transformation via simultaneous access to contiguous memory regions, which can be handled efficiently by the GPU. In addition, this data rearrangement is interpreted as a product of circular permutations such that a sequence of seeds, which is an order of magnitude shorter than input data, allows the GPU threads to compute the complicated memory indexes needed for

*Corresponding author. Tel.: +81 6 6879 4353; fax: +81 6 6879 4354.
Email address: `ino@ist.osaka-u.ac.jp` (Fumihiko Ino)

parallel rearrangement. Because the DWT is usually part of a processing pipeline in an application, we believe that the proposed method is useful for retaining the amount of memory for use by other pipeline stages.

Keywords: Discrete wavelet transform, lifting scheme, memory-saving computation, in-place algorithm, GPU

1. Introduction

The discrete wavelet transform (DWT) [1] is a well-known method for time-frequency analysis, which analyzes a signal in both the time and frequency domains using various time-frequency representations. The DWT employs wavelets as these representations, or scaled and translated versions of a finite-length or fast-decaying oscillating waveform called a mother wavelet. The DWT is useful for classifying the input signal into high-frequency components and low-frequency components in the same manner as the traditional fast Fourier transform (FFT). Given a signal with n samples (we assume that n is a power of 2), the DWT can be processed within $\mathcal{O}(n)$ time, whereas the FFT requires $\mathcal{O}(n \log n)$ time; consequently, the DWT is more computationally efficient than the FFT. Similar to the FFT, the DWT is a separable filter. For example, a two-dimensional (2-D) DWT filter can be realized by row-wise and column-wise (horizontal and vertical) 1-D DWT filters. The DWT is useful in various fields, such as data compression for images [2, 3, 4], artifact removal [5], data decoding for earth observation satellites [6] and clustering analysis for geometrical data [7]. These typical applications usually employ multiresolution analysis [1], where the DWT is applied recursively to low-frequency components at each resolution level of the hierarchy. Therefore, a fast DWT implementation is required to process DWT-based applications in real-time.

Acceleration methods for the DWT can be classified into two types: algorithm-based and accelerator-based approaches. An example of the former is the lifting scheme [8], which halves the computational requirements of the traditional filter bank scheme [9]. Furthermore, the lifting scheme is an in-place algorithm, which is capable of overwriting the algorithm's output onto the input buffer; thus, the lifting scheme reduces the amount of memory usage by half compared with the filter bank scheme. In this study, an in-place algorithm is defined as an algorithm that requires $\mathcal{O}(1)$ memory space, excluding the input buffer of size n.

The DWT has also been implemented in many accelerators, such as graphics processing units (GPUs) [10], field programmable gate arrays [11], and cell broadband engines [12, 13]. The GPU is one of the most popular commodity devices among these hardware types, where it provides a much higher memory bandwidth than the CPU. For example, the latest Maxwell architecture [14] provides 336 GB/s of memory bandwidth, which is one magnitude higher than that of the double data rate fourth generation (DDR4) memory. Consequently, many memory-bound applications have been successfully accelerated on GPUs [15, 16, 17]. The DWT is a memory-bound operation rather than a compute-bound operation, so we consider that the GPU is a promising accelerator for the DWT.

Laan *et al.* [10] accelerated the lifting scheme on a GPU that is compatible with the compute unified device architecture (CUDA) [18]. They implemented row-wise and column-wise 1-D DWTs as kernel functions, which are then offloaded from the CPU to the GPU. Their kernels facilitate efficient memory access by using a data layout that stores high-frequency components in a contiguous memory region. After the high-frequency components, low-frequency components are also stored in a contiguous memory region. This data layout is useful for maximizing the effective memory bandwidth on a GPU because memory transactions are performed per warp, i.e., a series of GPU threads that executes the same instruction every clock cycle [18], where the threads in a warp are allowed to access a contiguous memory region simultaneously via a 128-byte memory transaction.

However, this layout distinguishes the output buffer from the input buffer, thereby eliminating the advantage of in-place transformation. For signal data larger than half of the GPU memory capacity, the data must be swapped from the GPU memory to the CPU memory. This data transfer between the CPU and GPU degrades the overall transformation performance because the peak bandwidth of the PCI Express (PCIe) bus is 15.8 GB/s, which is one magnitude lower than that of the GPU memory. In addition, the DWT is usually part of a processing pipeline in an application, so it is better to retain the amount of memory for use by other pipeline stages.

In this study, in order to obtain a fast lifting-based DWT for large-scale data, we propose a parallel method that reduces the amount of memory usage and that facilitates highly efficient memory access on a CUDA-compatible GPU. The proposed method achieves memory-saving transformation by unifying the input buffer and the output buffer but with the cost of a tiny working memory space, which is less than the input size n. Naive unification avoids parallelizing the lifting scheme because the unified buffer causes many flow dependencies during transformation. To address this issue, the proposed method partitions the input data into small chunks, which are then rearranged in an appropriate manner so the parallel DWT can be performed over the unified buffer. Thus, this data rearrangement can be regarded as a preprocessing phase that classifies the input data into groups so different groups are independent of each other, where data-dependent tasks within the same group are processed sequentially, but different groups can be processed in parallel. Furthermore, chunk operations can be processed in a single-instruction, multiple-data (SIMD) manner because each chunk comprises multiple data elements.

In addition to the buffer unification mentioned above, the proposed method performs the DWT via simultaneous access to contiguous regions. Thus, after the chunks have been generated using this GPU-friendly access pattern, the generated chunks can also be placed in appropriate positions by this favorable access pattern. Allowing the GPU threads to compute the complicated memory addresses where their responsible chunks must be placed is a challenging issue. In particular, irregular addresses must be computed from very small amounts of information with a data size less than n to maintain the advantage of the in-place transformation. To tackle this issue, the proposed method represents a sequence of memory addresses as a circular permutation. This interpretation allows us to precompute a small sequence of numbers, which can then be used as seeds to produce the sequence of memory addresses for use in data rearrangement.

The remainder of this paper is organized as follows. Section 2 introduces related research on GPU-accelerated DWT implementations. Section 3 summarizes the lifting scheme, which forms the basis of the proposed method. Section 4 describes the proposed method and Section 5 presents the experimental results. Finally, Section 6 gives the conclusions.

2. Related Work

Table 1 compares related research into the acceleration of the DWT on GPUs. Each study employed different hardware so the peak performance is presented with the transformation throughput to allow standardization. To the best of our knowledge, no previous GPU-based implementation has achieved a compact data representation and data arrangement that enables GPU-friendly memory accesses for the lifting-based DWT.

Wong *et al.* [19] implemented the filter bank scheme on a GeForce 7800 GTX GPU, where their five-level DWT implementation was based on OpenGL [20], or a graphics application programming interface (API). They selected the filter bank scheme rather than the lifting scheme because the data dependencies in the lifting scheme are more complex than those in the filter bank scheme. These complex data dependencies cannot be handled efficiently by OpenGL, which is specialized for graphics applications, because OpenGL has many restrictions on its programming capability compared with CUDA. For example, OpenGL prohibits the explicit management of an on-chip cache called *shared memory* [18]. Despite these restrictions, Tenllado *et al.* [21] implemented the lifting scheme using OpenGL. However, they found that the filter bank scheme was faster than the lifting scheme due to the complex data dependencies mentioned above.

Franco *et al.* [22] implemented the filter bank scheme on a Tesla C870 GPU. They adopted a row-column approach, which reuses a horizontal DWT kernel as a vertical DWT kernel by transposition. Their implementation transformed an 8192×8192-pixel image within 71 ms and the transformation throughput reached 0.9 Gpixel/s. However, the overall performance was limited by data transfer between the CPU and GPU. Transferring the data between the CPU and GPU required 163 ms, so the transformation throughput dropped to 0.3 Gpixel/s after considering these overheads. A similar row-column approach was presented by Galiano *et al.* [23].

To the best of our knowledge, Matela *et al.* [3] first implemented the filter bank scheme on a CUDA-compatible GPU. Their implementation was based on a block-based approach, which allows the horizontal and vertical DWT kernels to be merged into a single kernel. Their implementation copies 2-D blocks into the shared memory, which can be commonly accessed from threads in the same thread block [18]. This on-chip memory is useful for saving the GPU memory bandwidth, so that the ratio R of the memory throughput over the peak memory bandwidth was more than 1.0. That is, on-chip data was efficiently shared among threads. In their implementation, they distinguished the output region from the input region.

Laan *et al.* [10] proposed a hybrid of the row-column and block-based approaches. Their horizontal process is same as that in the row-column approach. By contrast, their vertical process avoids transposition by applying the vertical DWT to vertically-long blocks. Furthermore, a sliding window mechanism was developed to reuse on-cache data fetched from the GPU memory. Similar to the block-based approach [3], this hybrid approach increased the ratio R to more than 1.0. Using a GeForce 8800 GTX card, their implementation achieved a transformation throughput of 1.5 Gpixel/s for a 4096×4096-pixel image. This throughput was roughly $\times 3$ higher than those achieved by a row-column scheme and an OpenGL-based implementation [21]. As presented theoretically in [8], Laan *et al.* demonstrated that the lifting scheme was two times faster than the filter bank scheme on the CUDA-compatible GPU. Kucis *et al.* [24] presented an extension of Laan's method that reduces synchronization by allowing redundant computation between threads. This extended method achieved a transformation throughput of 8.3 Gpixel/s on a GeForce GTX 580 card, which was 30% faster than the basic method. However, after considering the data transfer between the CPU and GPU, the transformation throughput decreased to 0.4 Gpixel/s.

Sharma *et al.* [25] used OpenCL [26] to implement the lifting scheme on a GeForce GTX 285 GPU. OpenCL is a standard for heterogeneous computing platforms, so their implementation ran on CUDA-incompatible GPUs and multicore CPUs. Their OpenCL-based implementation achieved a transformation throughput of 0.4 Gpixel/s for a 4096×4096-pixel image, but it was slower than that of their CUDA-based implementation, which was 0.7 Gpixel/s.

Table 1: Comparison of related research. P denotes the peak memory bandwidth for the GPU employed. ρ and ρ^+ denote the transformation throughputs without and with data transfer time between the CPU and GPU, respectively. For the L-level DWT, the number $\mathcal{N}$ of transformed pixels is given by $\mathcal{N} = \sum_{k=1}^{L} N^2/4^{k-1}$ because the level k transforms an $N/2^{k-1} \times N/2^{k-1}$-pixel region. $R = \mathcal{N}A/T/P$ is the ratio of the memory throughput over the peak memory bandwidth P, where A and T are the amount of memory access per pixel and the execution time, respectively. The proposed method assumes that a chunk comprises $b = 512$ elements.

Study	DWT scheme	API	GPU	Memory	Data size	Memory	P	R	Throughput (Gpixel/s)	
	(level)			capacity (MB)	(pixel)	usage	(GB/s)		ρ	ρ^+
Wong [19]	Filter bank (5)	OpenGL	7800 GTX	512	2K×2K	$2n$	38	—	—	0.003
Tenllado [21]	Lifting (5)	OpenGL	7800 GTX	512	2K×2K	$2n$	38	0.33–0.73	0.3–0.4	—
Tenllado [21]	Filter bank (5)	OpenGL	7800 GTX	512	2K×2K	$2n$	38	0.57–0.66	0.3–0.6	—
Franco [22]	Filter bank (1)	CUDA	Tesla C870	1536	8K×8K	$2n$	77	0.49	0.9	0.3
Laan [10]	Lifting (3)	CUDA	8800 GTX	768	4K×4K	$2n$	86	1.22	1.5	—
Matela [3]	Lifting (1)	CUDA	GTX 295	896	2K×2K	$2n$	112	1.67	2.6	—
Galiano [23]	Lifting (1)	CUDA	GTX 280	1024	4K×4K	$2n$	142	0.97	2.5	—
Sharma [25]	Lifting (3)	OpenCL	GTX 285	1024	4K×4K	$2n$	159	0.11	0.4	—
Sharma [25]	Lifting (3)	CUDA	GTX 285	1024	4K×4K	$2n$	159	0.18	0.7	—
Kucis [24]	Lifting (1)	CUDA	GTX 580	1536	4K×4K	$2n$	192	3.11	8.3	0.4
This paper	Lifting (1)	CUDA	GTX 580	1536	16K×16K	$n + \lceil n/2b \rceil$	192	1.27	2.8	0.7
This paper	Lifting (1)	CUDA	GTX 980 Ti	6192	32K×32K	$n + \lceil n/2b \rceil$	336	1.38	5.3	2.0

3. Lifting based DWT

Let $x_0, x_1, \ldots, x_{n-1}$ be the input signal with n samples. We assume that n is a power of two. Given the input signal, the DWT outputs its low-frequency components $c_0, c_1, \ldots, c_{n/2-1}$ and high-frequency components $d_0, d_1, \ldots, d_{n/2-1}$. In the lifting scheme [8], the t-th element c_t of low-frequency components and d_t of high-frequency components, where $0 \leq t < n/2$, are given by

$$c_t = x_{2t} + \sum_{i=-k_0}^{k_0-1} u_i d_t, \quad (1)$$

$$d_t = x_{2t+1} - \sum_{j=-k_1+1}^{k_1} p_j x_{2(t+j)}, \quad (2)$$

where u_i $(-k_0 \leq i < k_0)$ and p_j $(-k_1 < j \leq k_1)$ are coefficients determined by the mother wavelet.

The data dependencies in Eqs. (1) and (2) indicate that in-place transformation can be achieved by overwriting d_t to x_{2t+1}, then c_t to x_{2t}. This access pattern implies a cyclic layout, which stores the low-frequency element c_t and high-frequency element d_t alternately: $c_0, d_0, c_1, d_1, \ldots, c_{n/2-1}, d_{n/2-1}$. Using this layout, the DWT can be implemented with two kernels, a high-pass filter kernel for computing high-frequency components and a low-pass filter kernel for computing low-frequency components.

One drawback of the cyclic layout is that it requires data packing to obtain the filtered result, where the high- and low-frequency components are stored in odd and even elements, respectively (note that the indexes start from zero). Furthermore, the cyclic layout cannot maximize the memory throughput during transformation because both kernel functions cause a memory stride of two, where the high- and low-pass kernels output odd and even elements, respectively. As a further issue, the memory stride doubles at every hierarchical level of the multiresolution analysis, where the DWT is applied recursively to low-frequency components.

3.1. Laan's Previous Method

Laan *et al.* [10] presented a solution to the data layout issue. Their method increases the efficiency of memory access by using a block layout, which stores low-frequency components and high-frequency components in contiguous regions: $c_0, c_1, \ldots, c_{n/2-1}, d_0, d_1, \ldots, d_{n/2-1}$. However, this method separates the output buffer from the input buffer due to the data dependencies inherent in Eqs. (1)

and (2). Consequently, it eliminates the advantage of in-place transformation and consumes double the memory space, as mentioned in Section 1.

Consider here a block layout that naively unifies the input buffer and the output buffer. As shown in Fig. 1(a), the high-pass filter kernel outputs high-frequency element d_t to $x_{t+n/2}$, where $0 \leq t < n/2$. However, this execution eliminates part of the original elements, $x_{n/2}, x_{n/2+2}, \ldots, x_{n-2}$, which are needed by the second kernel to compute $c_{n/4}, c_{n/4+1}, \ldots, c_{n/2-1}$, as indicated in Eq. (1).

Two alternative approaches can be employed to avoid this elimination: a serialization approach that sequentially processes data-dependent tasks or a duplication approach that copies all of the elements, $x_0, x_1, \ldots, x_{n-1}$, to another memory region before outputting the computational results. The former approach fails to take advantage of massively parallel computing on the GPU. The latter approach corresponds to the previous method [10], which separates the output buffer from the input buffer.

4. Proposed Method

Similar to Laan's previous method [10], the proposed method employs a block layout to exploit the high peak memory bandwidth of the GPU. Furthermore, the proposed method rearranges the data elements so the odd elements follow the even elements, as shown in Fig. 1(b). This rearrangement is performed on a per-chunk basis, where each chunk contains b elements. Next, the method computes Eqs. (1) and (2) to complete the lifting operation [10]. For simplicity, we assume 1-D data in the following discussion.

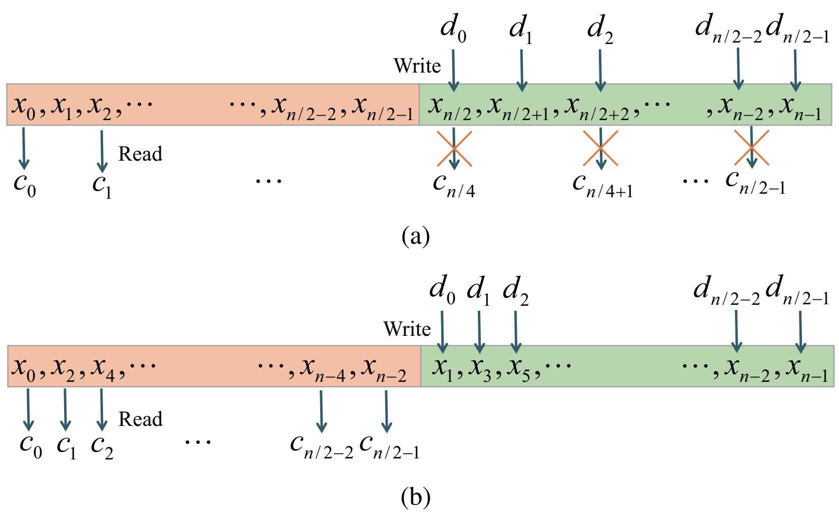

d_0 d_1 d_2 $d_{n/2-2}$ $d_{n/2-1}$
Write
$x_0, x_1, x_2, \cdots$ $\cdots, x_{n/2-2}, x_{n/2-1}$ $x_{n/2}, x_{n/2+1}, x_{n/2+2}, \cdots$ $, x_{n-2}, x_{n-1}$
Read
c_0 c_1 $\cdots$ $c_{n/4}$ $c_{n/4+1}$ $\cdots$ $c_{n/2-1}$
(a)
d_0 d_1 d_2 $d_{n/2-2}$ $d_{n/2-1}$
Write
$x_0, x_2, x_4, \cdots$ $\cdots, x_{n-4}, x_{n-2}$ $x_1, x_3, x_5, \cdots$ $\cdots, x_{n-2}, x_{n-1}$
Read
c_0 c_1 c_2 $\cdots$ $c_{n/2-2}$ $c_{n/2-1}$
(b)

Figure 1: Memory access pattern in lifting-based DWT. (a) When writing high-frequency components, a naive unification scheme eliminates the original elements needed for the latter half of the low-frequency components. (b) By contrast, the proposed scheme maintains the original elements after writing the high-frequency components.

4.1. Data Rearrangement

The proposed method rearranges the input data according to the following two steps.

1. Chunk generation: Given an input signal $x_0, x_1, \ldots, x_{n-1}$ with n samples, the method constructs an even chunk and an odd chunk within every segment of $2b$ elements (see Fig. 2). An even chunk contains even elements whereas an odd chunk contains odd elements. In the following discussion, let m $(= \lceil n/2b \rceil)$ be the number of even chunks. For simplicity, the number of odd chunks is assumed to be the same as that of even chunks; thus, the total number of chunks is given by $2m$. To facilitate highly efficient memory access, the chunk size is assumed to be a multiple of 128 bytes be-

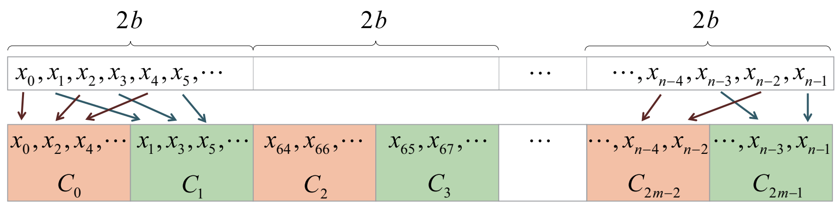


Figure 2: Chunk generation. Even and odd chunks are generated within every segment of $2b$ elements. After generation, even and odd chunks appear alternately.

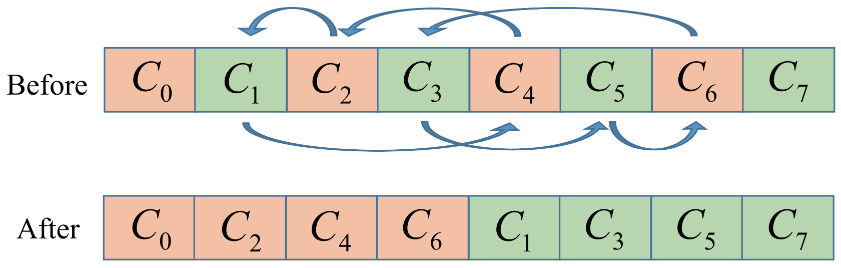


Figure 3: Chunk rearrangement ($m = 4$). Odd chunks follow even chunks after rearrangement. Note that C_0 is an even chunk because the indexes start from zero.

cause the current CUDA accesses the global memory via 128-byte memory transactions [18].

2. Chunk rearrangement: $2m$ chunks are rearranged such that m odd-chunks appear after m even-chunks, as shown in Fig. 3. More formally, the $2i$-th chunk and the $(2i + 1)$-th chunk, where $0 \leq i < m$, are placed in the i-th chunk and the $(i + m)$-th chunk after rearrangement.

Algorithm 1 shows the proposed chunk generation algorithm, which describes the kernel function that needs to be executed by massive GPU threads in parallel. This algorithm assumes that a thread block [18] containing $2b$ threads is responsible for generating two chunks within a segment of $2b$ elements. Each thread loads its responsible element from the global memory to a shared array s of size $3 \times b$

(line 3), before performing local synchronization within the thread block. Note that the last column of this array is a dummy region padded for avoiding bank conflicts [18] when writing the loaded element back to the appropriate location in the global memory (line 10). This row-wise algorithm can easily be extended to a column-wise algorithm without using the shared memory. As described in [10], column-wise operations can be achieved by assigning a row of 32 neighboring columns to a warp, where every warp is allowed to access a contiguous memory region.

Algorithm 1 Chunk generation kernel.

Input: Input signal $x_0, x_1, \ldots, x_{n-1}$ with n samples and b elements in a chunk
Output: Per-chunk rearranged elements $x_0, x_2, \ldots, x_{n-1}$ containing alternate odd and even chunks

```
1:  $i \leftarrow$ thread index;
2:  $j \leftarrow$ thread block index;
3:  $s_{i\%2,\lfloor i/2 \rfloor} \leftarrow x_{2bj+i}$;   ▷ Load responsible element into a shared array of size $3 \times b$
4:  __synchthreads();                                   ▷ Local synchronization
5:  **if** $i < b$ **then**                              ▷ Even chunk generation
6:      $k \leftarrow 2i$;
7:  **else**                                             ▷ Odd chunk generation
8:      $k \leftarrow 2(i - b) + 1$;
9:  **end if**
10: $x_{2bj+i} \leftarrow s_{k\%2,\lfloor k/2 \rfloor}$;
```

Chunk rearrangement can be represented as a product of l circular permutations: $\sigma_0 \circ \sigma_1 \circ \ldots \circ \sigma_{l-1}$, where σ_j $(0 \le j < l)$ represents the j-th circular permutation. Let $(i\ j)$ be a transposition representing an exchange between the i-th chunk and the j-th junk. The chunk rearrangement for $m = 4$ shown in Fig. 3

can then be formulated as a product of two circular permutations, $(1\ 4\ 2) \circ (3\ 5\ 6)$. According to this interpretation, a circular permutation $(1\ 4\ 2)$ represents a data-dependent task that circulates chunks C_1, C_4 and C_2. Thus, a task is a series of chunk movements that must be processed sequentially. However, different circular permutations are independent of each other, so they can be processed in parallel. Therefore, the proposed method assigns a circular permutation to a thread block, which then sequentially processes a series of chunk movements that corresponds to the assigned circular permutation. Note that a chunk contains multiple data elements; thus, the data elements within a chunk can be processed simultaneously by b threads from the same thread block in a SIMT manner.

A critical issue is how to formulate circular permutations according to the number n of samples. To address this issue, we introduce the following lemma.

Lemma 1. *Let s be the size of an arbitrary circular permutation σ_j $(0 \leq j < l)$, which represents a series of chunk movements in step 2. Then, $s \leq \lceil \log 2m \rceil$.*

Proof . *Let $(a_0\ a_1\ a_2\ \ldots\ a_{s-1})$ be an arbitrary circular permutation, which represents a series of chunk movements in step 2, where this permutation indicates that chunks $C_{a_0}, C_{a_1}, \ldots, C_{a_{s-1}}$ must be circulated. Then, for all $0 \leq k < s$, $0 \leq a_k < 2m$ because there are $2m$ chunks in total. According to step 2, we have Eq. (3).*

$$a_k = \begin{cases} a_{k+1}/2, & \text{if } a_{k+1} \text{ is even,} \\ \lfloor a_{k+1}/2 \rfloor + m, & \text{otherwise.} \end{cases} \tag{3}$$

Note that for all $0 \leq k < s$, a_k can be represented as a binary number with $\lceil \log 2m \rceil$ bits. In this binary representation, applying a circular right shift to a_{k+1} gives a_k, as indicated by Eq. (3). For example, the permutation $(1\ 4\ 2)$ has three elements, 1, 4, and 2, which correspond to binary representations of 001, 100, and 010, respectively, and the last two elements can be obtained by applying a circular right shift to the first element 001 (see Fig. 4). Therefore, for an arbitrary a_k, applying a circular right shift at most $\lceil \log 2m \rceil$ times to a_k produces itself; thus, $s \leq \lceil \log 2m \rceil$. □

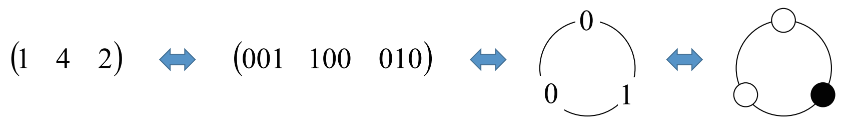


Figure 4: Interpretation of chunk rearrangement with a circular permutation. A circular permutation of length M can be mapped to a necklace with M beads and two colors, where rotations are considered equivalent. In this example, $M = 3$.

According to Eq. (3), we have Eq. (4):

$$a_{k+1} = 2a_k \bmod (2m-1), \tag{4}$$

where $0 \leq k < s$. Thus, the representative element a_0 of a circular permutation produces all the elements of that circular permutation. This means that the final memory location for an arbitrary chunk can be computed from a_0, i.e., a seed.

Algorithm 2 describes the proposed method for rearranging chunks according to step 2. This algorithm assumes that the process uses l thread blocks of size b and three inputs: (1) $2m$ chunks that need to be rearranged, (2) the number l of circular permutations, and (3) their representative elements $p_0, p_1, \ldots, p_{l-1}$, where p_j is the representative element for circular permutation σ_j $(0 \leq j < l)$. Given these inputs, the thread block with index j loads a chunk that corresponds to the representative element p_j in a register (line 2). Next, another chunk that corresponds to the next element in circular permutation σ_j is moved into the empty memory location where the copied chunk was present (line 5). Similar chunk movements are iterated until the representative element is reached (lines 4–7). Finally, the algorithm moves the representative element p_j from the register into the last empty location (line 8). Note that this rearrangement is optimal in terms of the amount of memory access because all of the chunks that must move to a different location are loaded once and stored once.

Algorithm 2 Chunk rearrangement kernel based on a product of circular permutations.

Input: Chunks $C_0, C_1, \ldots, C_{2m-1}$ that need to be rearranged, number $2m$ of chunks, and representative elements $p_0, p_1, \ldots, p_{l-1}$ of circular permutations
Output: Rearranged chunks $C_0, C_1, \ldots, C_{2m-1}$

```
1: $j \leftarrow$ thread block index;
2: $W \leftarrow C_{p_j}$;                                  ▷ $b$ elements are copied in parallel
3: $i \leftarrow p_j$;
4: **while** $2i \bmod (2m-1) \neq p_j$ **do**
5:     $C_i \leftarrow C_{2i \bmod (2m-1)}$;                ▷ $b$ elements are copied in parallel
6:     $i \leftarrow 2i \bmod (2m-1)$;
7: **end while**
8: $C_i \leftarrow W$;                                      ▷ $b$ elements are copied in parallel
```

4.2. Precomputation of Representative Elements

In order to perform the in-place DWT, the GPU threads must compute the number l of circular permutations and all the representative elements $p_0, p_1, \ldots, p_{l-1}$, for an arbitrary problem size n. In particular, the representative elements must be assigned to thread blocks without duplication. However, it is not easy to parallelize this assignment to GPU threads, mainly due to the lack of a global synchronization mechanism.

Our solution to this issue is to precompute the number l of circular permutations and representative elements $p_0, p_1, \ldots, p_{l-1}$ on the CPU. Algorithm 3 describes our sequential algorithm for computing all the representative elements. This algorithm enumerates all of the elements in a circular permutation σ_j and stores the minimum value as p_j. The representative elements are computed per circular permutation, so an array X is used to avoid duplicated enumeration.

Algorithm 3 Precomputation of representative elements for circular permutations.

Input: Number $2m$ of chunks

Output: Number l of circular permutations and representative elements $p_0, p_1, \ldots, p_{l-1}$ of circular permutations

```
 1: for $i \leftarrow 0$ to $2m - 1$ do                         ▷ Initialization
 2:     $X_i \leftarrow 0$;
 3: end for
 4: $l \leftarrow 0$;
 5: for $i \leftarrow 0$ to $2m - 1$ do                         ▷ Compute $p_l$
 6:     if $X_i = 0$ then                                    ▷ Not registered yet
 7:         $p_l \leftarrow i$;
 8:         $X_i \leftarrow 1$;
 9:         $j \leftarrow 2i \bmod (2m - 1)$; ▷ Generate number $j$ such that $j$ and $i$ belong
    to the same circular permutation
10:         while $j \neq i$ do
11:             $X_j \leftarrow 1$;
12:             $j \leftarrow 2j \bmod (2m - 1)$;
13:         end while
14:         $l \leftarrow l + 1$;
15:     end if
16: end for
```

The representative elements $p_0, p_1, \ldots, p_{l-1}$ are then transferred to constant memory [18] (i.e., GPU memory) before generating chunks on the GPU. Note that the capacity of constant memory is 64 KB in the current CUDA. This capacity is sufficiently large to store representative elements for the largest problem size n that can be processed on a GPU. A detailed analysis is provided in Section 5.1.

4.3. Space Complexity Analysis

The proposed method requires two different working spaces to perform the DWT: (1) constant memory used for representative elements, and (2) shared memory and registers used for chunk generation and circulation.

The constant memory usage depends on the number l of representative elements. The following lemma [27] is introduced to formulate l for arbitrary n.

Lemma 2. *According to Burnside's lemma [28], the number of necklaces with M beads and K (> 1) colors (see Fig. 4), where rotations are considered to be equivalent, is*

$$\frac{1}{M}\sum_{i|M}\mu\left(\frac{M}{i}\right)K^{i} = \frac{1}{M}\sum_{i=1}^{M}K^{\gcd(i,M)}, \tag{5}$$

where μ is the classical Möbius function and $\gcd(i, M)$ is the greatest common divisor of i and M.

Thus, the number $l+2$ is equivalent to that of necklaces with $M = \lceil \log 2m \rceil$ beads and $K = 2$ colors, where rotations are considered to be equivalent. Note that "+2" considers two exceptions, permutations $(0\ 0\ \dots 0)$ and $(1\ 1\ \dots 1)$, which never move chunks during the arrangement process. These exceptions correspond to the head and tail chunks. Lemma 2 gives the following theorem.

Theorem 1. *The number l of representative elements is smaller than $m = \lceil n/2b \rceil$ if $m \geq 128$.*

Proof . *Eq. (5) can be simplified further as follows.*

$$\begin{aligned} l+2 = \frac{1}{M}\sum_{i=1}^{M}K^{\gcd(i,M)} &\leq \frac{1}{M}\sum_{i=1}^{M}K^{i} && (6)\\ &= \frac{K(K^{M}-1)}{M(K-1)} && (7)\\ &< \frac{K^{M+1}}{M(K-1)} && (8)\end{aligned}$$

$M = \lceil \log 2m \rceil$, $K = 2$ and $x \leq \lceil x \rceil \leq x + 1$ for an arbitrary real number x, so we have

$$\begin{aligned} l &< \frac{8m}{\log 2m} && (9) \\ &\leq m, \text{ if } m \geq 128. && (10) \end{aligned}$$

$\square$

Eq. (10) indicates that $2m$ chunk indexes are reduced to at least half because a circular permutation comprises at least two elements (i.e., two chunk indexes). Therefore, the number l of representative elements is smaller than $m = \lceil n/2b \rceil$. Asymptotically, l is bounded by $\mathcal{O}(n/\log n)$ from Eq. (9).

In addition, the proposed method generates $\mathcal{O}(n/\log n)$ thread blocks during chunk circulation because the method uses l thread blocks in total. The usage of registers per thread block (i.e., per circular permutation) is equivalent to the chunk size, where the chunk size is $4b$ in bytes if the signal data is stored in a 4-byte data type. Therefore, the proposed algorithm consumes $\mathcal{O}(n/\log n)$ registers if all of the thread blocks are processed simultaneously. However, due to the limited amount of computational resources, such as the shared memory and registers, CUDA restricts the maximum number of active thread blocks that can be processed simultaneously [18]. Therefore, our CUDA-based implementation runs with $\mathcal{O}(1)$ registers. Note that the proposed method achieves this restriction by data arrangement, which eliminates the data dependencies between different thread blocks. Such independent thread blocks are necessary to reduce the number of active thread blocks.

In summary, the proposed method consumes $\lceil n/2b \rceil$ additional space locations to perform the lifting-based DWT. Currently, we use $b = 512$, so this additional memory usage is an order of magnitude lower than the n needed by existing methods that distinguish the output buffer from the input buffer.

4.4. Tiling-based Transformation

Tiling-based approaches [29, 30] are useful for dealing with large-scale data that exceeds the GPU memory capacity. These approaches split the data into tiles that are transformed separately. However, Eqs. (1) and (2) indicate that each element needs its neighbors to update its value. Consequently, separated transformation creates blocking artifacts at the tile boundaries, thereby excessive tiles can degrade the transformation quality.

This degradation can be avoided by having halo regions for each tile (see Fig. 5); the input data is decomposed into small overlapping segments (with halo regions) to exactly transform all elements inside the tiles. Both the proposed and Laan's previous methods can be enhanced with a tiling-based approach by specifying a segment as the input. However, the proposed memory-saving method accepts roughly twice larger segments than those accepted by the previous method.

As for tile creation, a 1-D block decomposition scheme is deployed to minimize the overhead of data decomposition. An alternative is a 2-D block decomposition scheme, but it complicates data transfer between the CPU and GPU because elements of a 2-D tile is usually scattered throughout memory. In contrast, elements of a 1-D tile is stored in a contiguous memory region, which facilitates data transfer between the CPU and GPU.

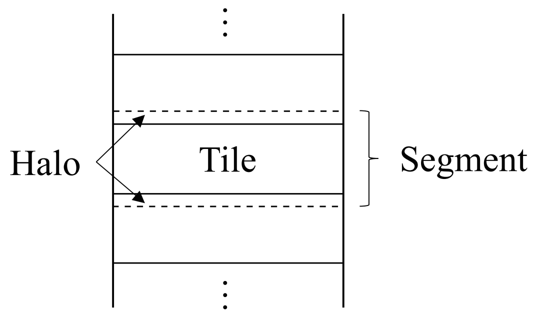


Figure 5: A tiling-based approach with halo regions.

Table 2: Specifictions of hardware devices deployed for experiments. Effective bandwidths were measured using the bandwidthTest program [18]. DMA stands for direct memory access.

GPU card	980 Ti	780 Ti	680	580
Compute capability	5.2	3.5	3.0	2.0
Architecture	Maxwell	Kepler	Kepler	Fermi
GPU memory capacity (GB)	6	3	2	2
Peak memory bandwidth (GB/s)	336.5	336.0	192.2	192.4
Effective memory bandwidth (GB/s)	249.1	236.3	152.5	169.2
Peak PCIe bandwidth (GB/s)	15.8	15.8	15.8	8.0
Effective PCIe bandwidth (GB/s)	11.5	11.8	12.2	6.3
Number of DMA engines	2	1	1	1

Note that tiling-based approaches can be efficiently processed in a pipeline; the data segments are streamed through the pipeline to overlap CPU-GPU data transfer with kernel execution. Such a pipeline can be implemented with CUDA streams [18].

5. Experimental Results

To evaluate the proposed method, we performed comparisons with Laan's previous method [10], which enabled GPU-friendly memory accesses with a block layout scheme (see Section 3.1). In the experiments, we investigated the memory usage and execution time for the Deslauriers-Dubuc (13,7) wavelet [31] and $N \times N$-pixel images, where $2\mathrm{K} \leq N \leq 32\mathrm{K}$. Each pixel comprised 4-byte data. Table 2 summarizes the specifications for our experimental machine, which had an Intel Core i7-3770K CPU and either a GeForce GTX 980 Ti, 780 Ti, 680 or 580. The machine ran with Windows 8.1 (64 bit), CUDA 7.5 [18] and the driver version 355.82. All of the implementations were compiled using the Visual Studio 2013 with the `-O2` option.

5.1. Memory Consumption

Figure 6 shows the amount of memory usage. Compared with the previous method, the proposed method halved the amount of memory usage by buffer unification. Consequently, on a GeForce GTX 980 Ti GPU, the proposed method processed an $N \times N$-pixel image of up to $N = 32\mathrm{K}$ pixels without data decomposition. By contrast, the previous method processed a maximum size of $N = 16\mathrm{K}$ pixels without data decomposition. Note that the entire GPU memory cannot be used for the input/output buffer because part of the GPU memory is reserved for the frame buffer.

Next, we investigated the additional memory size $\alpha\ (= 4l)$ required to store all of the representative elements (Fig. 7). As shown in this figure, the size α reached 3.09 KB for a large image of $32\mathrm{K} \times 32\mathrm{K}$ pixels, which consumed 4 GB of the GPU memory. Consequently, we consider that the proposed method is useful for minimizing additional memory usage with a practical data size, although it does not achieve in-place transformation.

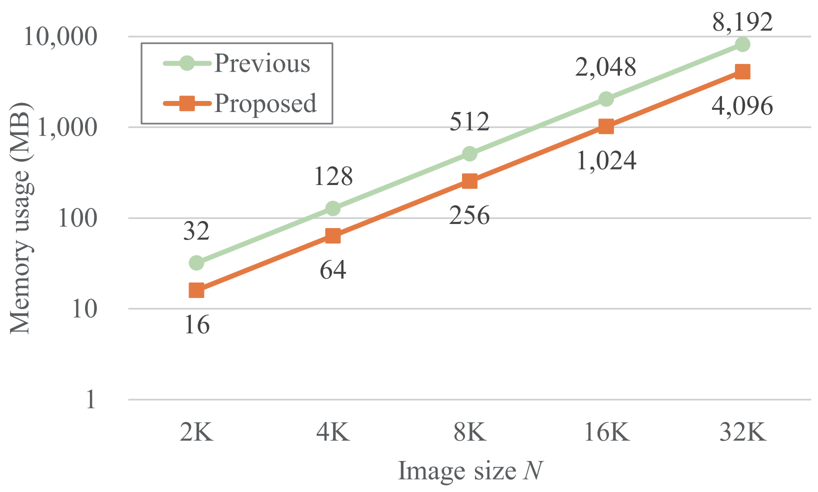


Figure 6: Comparison of memory usage with different image sizes N. An image comprises $N \times N$ pixels.

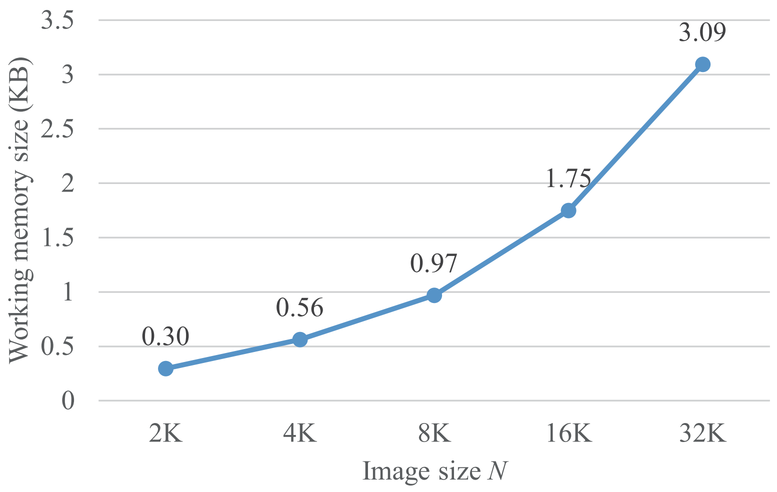


Figure 7: Working memory size required for representative elements. An image comprises $N \times N$ pixels.

5.2. DWT Performance

The DWT is part of a processing pipeline in a practical application. Therefore, many GPU-accelerated applications typically implement the DWT as well as other pipeline stages on the GPU to accelerate the overall pipeline. Consequently, we measured the execution time for four configurations where the input data and output data were stored in either the CPU memory or the GPU memory. In the following, the notation $x \to y$ denotes a configuration, where x and y represent the locations of the input data and the output data, respectively, and $x, y \in \{\mathrm{C}, \mathrm{G}\}$. The notations C and G correspond to the CPU memory and the GPU memory, respectively.

Figure 8 shows the execution times using the proposed method (without tiling) and the previous method (with tiling) for large data. Note that the previous method was extended to transform this large image successfully, where we decomposed the image into two segments to transform them in order, as presented in Section 4.4. However, the pipelining capability was switched off, so that data segments were processed in a synchronous manner.

The proposed method minimized the amount of memory usage, and it avoided data transfer between the CPU and GPU. Thus, the proposed method was 2.4–3.9 times faster than the tiled previous method when both the input and output data were stored in the GPU memory ($\mathrm{G} \to \mathrm{G}$). However, the proposed method requires a preprocessing phase, which increased the kernel execution time by 2.0–2.6 times. Similarly, the proposed method achieved speedups of 1.2–1.8 times compared with the tiled previous method when either the input data or output

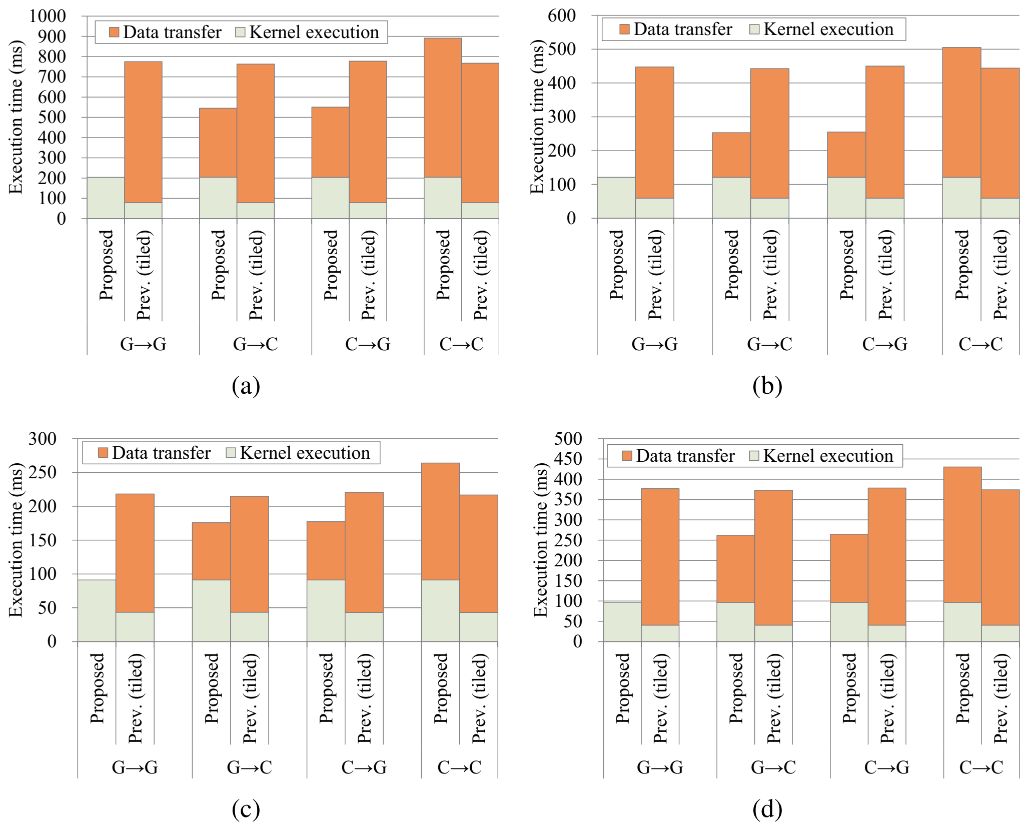


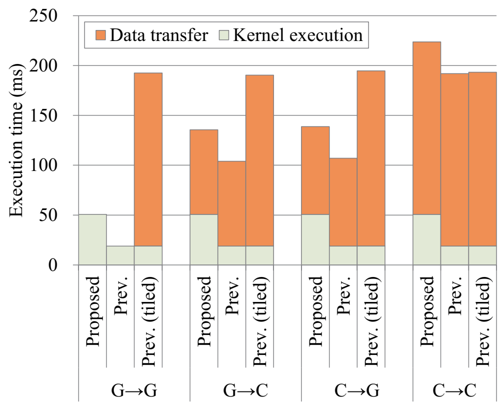
Figure 8: Execution time with different input/output configurations for large data. Results for (a) GeForce GTX 980 Ti with $N = 32\text{K}$, (b) 780 Ti with $N = 24\text{K}$, (c) 680 with $N = 16\text{K}$, and (d) 580 with $N = 16\text{K}$. The data sizes were 4.0 GB, 2.3 GB and 1.0 GB for $N = 32\text{K}$, $N = 24\text{K}$ and $N = 16\text{K}$, respectively.

data were stored in the GPU memory ($\mathrm{G} \rightarrow \mathrm{C}$ and $\mathrm{C} \rightarrow \mathrm{G}$). These configurations required data transfer between the CPU and GPU regardless of the data size. Moreover, the tiled previous method swapped half of the input data before the second transformation because its input and output buffers exhausted the GPU memory. This swapping operation increased the data transfer volume, and thus the tiled previous method was slower than the proposed method. Finally, the proposed method was 12–18% slower than the tiled previous method when both the input and output data were stored in the CPU memory ($\mathrm{C} \rightarrow \mathrm{C}$). However, CPU–GPU data transfer determines the transformation throughput for this configuration, so that these overheads can be partially hidden by pipelined execution, as described in Section 5.4.

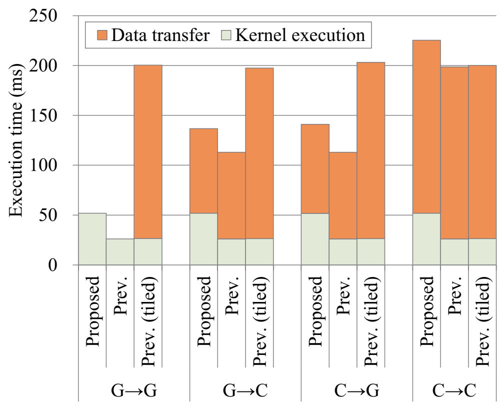
Figure 9 shows the execution times for small data, which could be transformed without tiling. Compared with the previous method, the proposed method required 1.1–2.7 more time to transform these small data. In this case, the previous method completed the DWT in almost the same data transfer time as the proposed method. Consequently, the preprocessing overheads increased the execution time for the proposed method. As mentioned for large data, these overheads can be partially addressed by pipelined execution.

5.3. Kernel Execution Efficiency

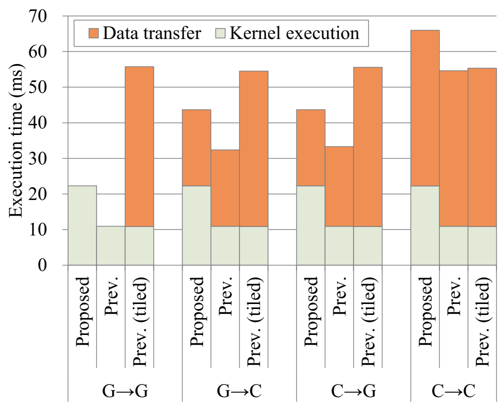
The DWT is a memory-bound operation, so we evaluated the execution efficiency of our kernel functions in terms of the effective memory throughput. Figure 10 shows the effective memory throughput $E = M/T$ for the proposed method,



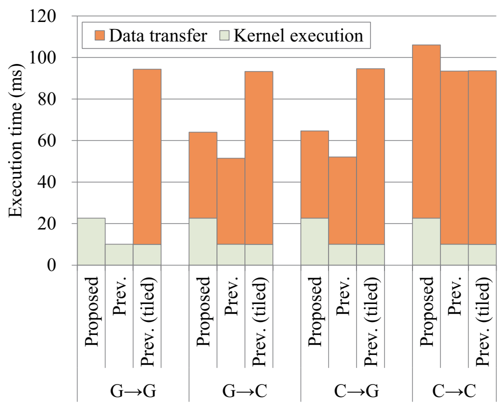
(a)



(b)



(c)



(d)

Figure 9: Execution time with different input/output configurations for small data. Results for (a) GeForce GTX 980 Ti with $N = 16\text{K}$, (b) 780 Ti with $N = 16\text{K}$, (c) 680 with $N = 8\text{K}$, and (d) 580 with $N = 8\text{K}$. The data sizes were 1 GB and 256 MB for $N = 16\text{K}$ and $N = 8\text{K}$, respectively.

Table 3: Number of data elements accessed. We used $r = 256$ and $h = 6$ for the Deslauriers-Dubuc (13,7) wavelet employed in this study.

Breakdown	Amount of memory access
Chunk generation	$4N^2$
Chunk rearrangement	$4N(N - 2b)$
Lifting DWT	$4(1 + h/r)N^2$

where M and T are the amount of memory accesses and the kernel execution time, respectively.

The amount M of memory accesses is determined as follows. First, the number of elements accessed in the chunk generation step is given by $2N^2$ because every chunk is read and written once during this step. This step is repeated twice, each for the horizontal 1-D DWT and the vertical 1-D DWT, so that $4N^2$ memory accesses are required for an $N \times N$-pixel image. Similarly, the chunk rearrangement step incurs $2N(N-2b)$ memory accesses, where $2b$ corresponds to the head and tail chunks, which never move during this step, and thereby $4N(N - 2b)$ locations are accessed for an $N \times N$-pixel image. Finally, the thread blocks in the lifting step access halo regions because computation for an element requires its neighboring elements. For simplicity, we considered 1-D data with a size of n. For 1-D data, n/r thread blocks referred to $r + 2h$ elements to output n elements, where r and $h = k_0 + k_1$ are the numbers of responsible elements per thread block and neighboring elements in the halo region, respectively. Consequently, the lifting step for 1-D data accessed $(r + 2h)n/r + n = 2(1 + h/r)n$ memory locations. By extending this result to 2-D data, which requires a row-wise 1-D DWT and a column-wise 1-D DWT, we obtained $4(1 + h/r)N^2$.

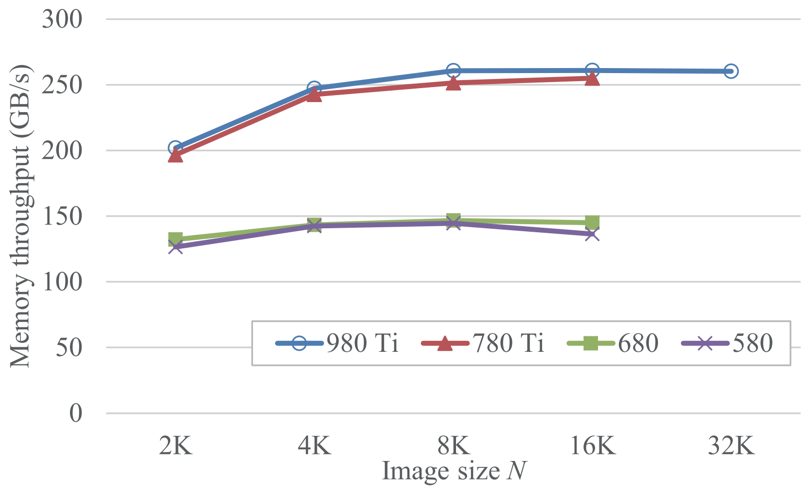


Figure 10: Effective memory throughputs obtained without tiling. Owing to memory exhaustion, kernels on 780 Ti, 680 and 580 cards failed to transform a 32K $\times$ 32K-pixel image.

As shown in Fig. 10, the effective memory throughputs were bounded by the peak GPU memory bandwidth of the deployed card (Table 2). The proposed method achieved high memory throughputs, although it changes the order of data elements in the preprocessing phase. This highly efficient arrangement is attributable to regular memory access in contiguous regions. However, Table 3 shows the main drawback of the proposed method, which is the increased amount of memory accesses. Compared with the previous method, the proposed method required three times more memory accesses to perform the lifting-based DWT with a unified buffer. This theoretical result explains why the proposed method required 2.0–2.6 times longer kernel time than the previous method (see Section 5.2).

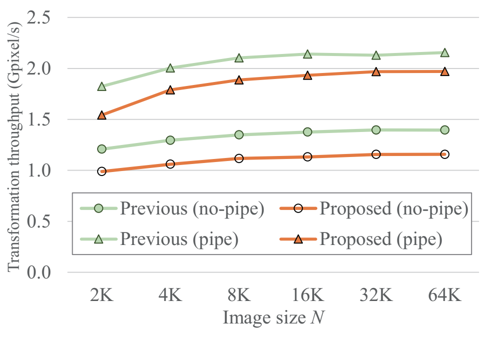
5.4. Pipelined DWT Throughput

We next evaluated tiling-based pipelined versions presented in Section 4.4. We measured the transformation throughput for configuration $\mathrm{C} \to \mathrm{C}$, where we assumed that an image of size $N \times N$ was stored in the CPU memory. The image size N ranged from 2K to 64K, so that the largest image could not be stored entirely in the GPU memory.

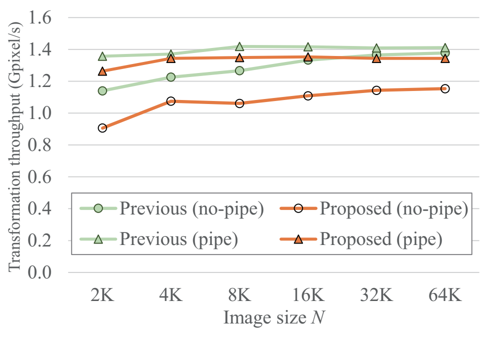
Figure 11 shows the throughputs measured for different image sizes N, where the number of tiles was set to N/b. Except for the 980 Ti, the throughputs achieved by the pipelined proposed method were close to those achieved using the pipelined previous method. By contrast, the non-pipelined proposed version was 12–25$\%$ slower than the non-pipelined previous version due to the preprocessing phase. Thus, the proposed method degraded the transformation throughput for on-memory data, but this performance degradation can be partially hidden by overlapping, particularly for large-scale data that cannot be stored entirely in the GPU memory.

With respect to the 980 Ti card, this card is equipped with two direct memory access (DMA) engines, which enable simultaneous bidirectional transfer between the CPU and GPU. Consequently, as compared with the non-pipelined version, the pipelined version reduced the data transfer time. However, this bidirectional transfer caused memory access contention, so there was a gap between the pipelined and non-pipelined throughputs, as shown in Fig. 11(a).

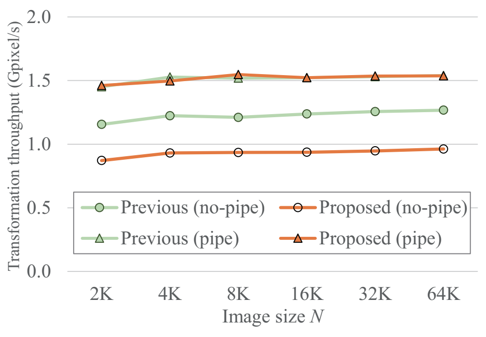
Figure 12 investigates this contention issue in more detail. This figure explains (1) why the pipelined proposed version was slower than the pipelined previous



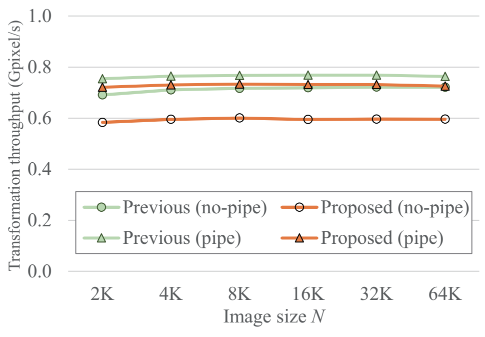
(a)



(b)

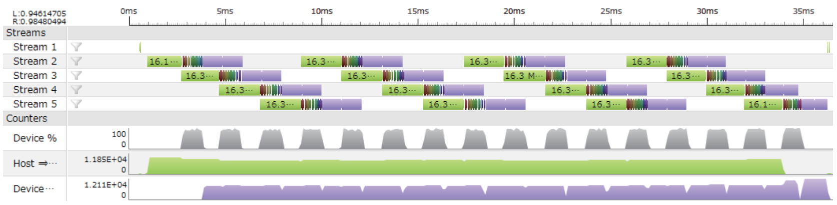


(c)

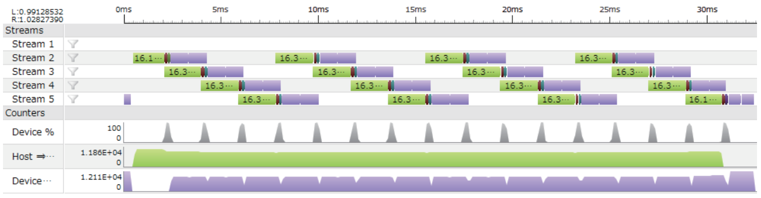


(d)

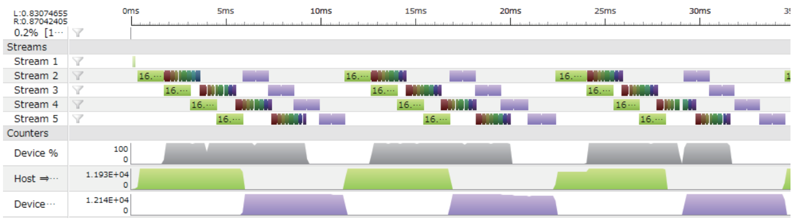
Figure 11: Comparison of transformation throughputs with different image sizes N. Results for (a) GeForce GTX 980 Ti, (b) 780 Ti, (c) 680, and (d) 580. The number of segments was fixed to N/b.



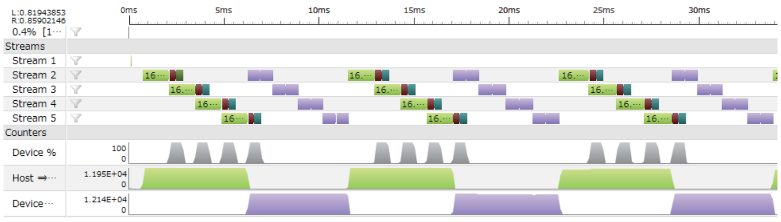
(a)



(b)



(c)



(d)

Figure 12: Timeline views of tiling-based pipelined implementations. In each subfigure, the upper four lines show the activities of CUDA streams. The lower three lines summarize the activities in terms of kernel execution (gray), CPU$\rightarrow$GPU data transfer (green) and GPU$\rightarrow$CPU data transfer (purple). GeForce GTX 980 Ti has two DMA engines, so that (a) the kernel execution overhead of the proposed method increased data transfer time compared to (b) the previous method. In contrast, GTX 680 has a single DMA engine, so that there was no significant gap between (c) the proposed method and (d) the previous method.

version on the GTX 980 Ti card and (2) why the former was as fast as the latter on the GTX 680 card. With multiple DMA engines, the GPU memory can be simultaneously accessed by three CUDA operations: (1) kernel execution, (2) CPU$\rightarrow$GPU data transfer and (3) GPU$\rightarrow$CPU data transfer, as shown in Figs. 12(a) and 12(b). This simultaneous access slightly affected the performance of the pipelined proposed version, which took longer kernel execution time than the pipelined previous version. In fact, the proposed version declined the data transfer rate by 10% compared to the previous version. Therefore, the declined rate determined the transformation time though kernel execution was fully overlapped with CPU-GPU data transfer. In contrast, such a decline was not observed on the GTX 680 card, which exclusively transfers data between the CPU and GPU owing to a single DMA engine; at most two CUDA operations can access the GPU memory simultaneously: (1) kernel execution and (2) either CPU$\rightarrow$GPU data transfer or GPU$\rightarrow$CPU data transfer, as shown in Figs. 12(c) and 12(d).

Thus, the pipelined DWT throughput for configuration $\mathrm{C} \rightarrow \mathrm{C}$ was mainly determined by the peak PCIe bandwidth. Because the 580 card has a PCIe generation 2 interface, its DWT throughput was half of that achieved with the 780 ti or 680 cards, which has a PCIe generation 3 interface. Multiple DMA engines further increased the throughput with a full overlap but declined the data transfer rate due to memory contention. The pipelined scheme is useful for partially hiding the overhead of the proposed method.

As for the practicality of the proposed method, the method is useful for processing larger data at a time. For example, a decoding system [6] for satellite images adopted a series of decoder accelerated on a GPU. Another example is a clustering system [7] that groups similar objects into disjoint classes. Because a cluster algorithm was applied to tiles separately, the proposed memory-saving method allows more objects to be grouped from an approximately twice larger tile.

6. Conclusion

In this study, we proposed a parallel DWT method that achieves both a compact data representation and GPU-friendly memory accesses for the lifting scheme. The proposed data representation unifies the output buffer with the input buffer. The key idea of unification is chunk-based data rearrangement, which separates data-independent tasks from data-dependent tasks, and thus massively parallel threads are allowed to move data elements according to circular permutations. The proposed method also achieves highly efficient memory access by allowing GPU threads to access contiguous memory regions. However, the proposed method requires at most $\lceil n/2B \rceil$ numbers as seeds for a series of memory addresses that are needed for parallel rearrangement.

The results of experiments showed that the proposed method was a maximum of 3.9 times faster than a previous method that separates the input buffer from the output buffer. In particular, the proposed method was useful when either the input buffer or output buffer existed in the GPU memory. We also found that the proposed method achieved a high memory throughput that was close to the peak memory bandwidth. However, our kernel was affected by the increased amount of memory accesses, i.e., three times that of the previous kernel. This disadvantage can be partially resolved by overlapping kernel execution with CPU-GPU data transfer. This pipelined implementation achieved a transformation throughput that was comparable to the previous method.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15K12008 and 15H01687, and the Japan Science and Technology Agency CREST program, “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems.” We are also grateful to the anonymous reviewers for their valuable comments.

References

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

[2] I. V. Šimek, R. R. ASN, GPU acceleration of 2D-DWT image compression in MATLAB with CUDA, in: Proc. 2nd UKSIM European Symp. Computer Modeling and Simulation (EMS’08), 2008, pp. 274–277.

[3] J. Matela, GPU-based DWT acceleration for JPEG2000, in: Proc. Doctoral Workshop Mathematical and Engineering Methods in Computer Science (MEMICS’09), 2009, pp. 136–143.

[4] J. Ao, S. Mitra, B. Nutter, Fast and efficient lossless image compression based on CUDA parallel wavelet tree encoding, in: Proc. Southwest Symp. Image Analysis and Interpretation (SSIAI'14), 2014, pp. 21–24.

[5] L. Zhu, Y. Zhou, D. Zhang, D. Wang, H. Wang, X. Wang, Parallel multi-level 2D-DWT on CUDA GPUs and its applications in ring artifact removal, Concurrency and Computation: Practice and Experience.

[6] C. Song, Y. Li, B. Huang, A GPU-accelerated wavelet decompression system with SPIHT and Reed-Solomon decoding for satellite images, IEEE J. Selected Topics in Applied Earth Observations and Remote Sensing 4 (3) (2011) 683–690.

[7] A. A. Yıldırım, C. Özdoğan, Parallel wavecluster: A linear scaling parallel clustering algorithm implementation with application to very large datasets, J. Parallel and Distributed Computing 71 (7) (2011) 955–962.

[8] W. Sweldens, The lifting scheme: A construction of second generation wavelets, SIAM J. Mathematical Analysis 29 (2) (1998) 511–546.

[9] S. G. Mallat, A theory for multiresolution signal decomposition: The wavelet representation, IEEE Trans. Pattern Analysis and Machine Intelligence 11 (7) (1989) 674–693.

[10] W. J. van der Laan, A. C. Jalba, J. B. Roerdink, Accelerating wavelet lifting on graphics hardware using CUDA, IEEE Trans. Parallel and Distributed Systems 22 (1) (2011) 132–146.

[11] M. E. Angelopoulou, P. Y. K. Cheung, K. Masselos, Y. Andreopoulos, Implementation and comparison of the 5/3 lifting 2D discrete wavelet transform computation schedules on FPGAs, J. Signal Processing Systems 51 (1) (2008) 3–21.

[12] D. A. Bader, V. Agarwal, S. Kang, Computing discrete transforms on the Cell Broadband Engine, Parallel Computing 35 (3) (2008) 119–137.

[13] M. Błażewicz, M. Ciżnicki, P. Kopta, K. Kurowski, P. Lichocki, Two-dimensional discrete wavelet transform on large images for hybrid computing architectures: GPU and CELL, in: Proc. 17th European Conf. Parallel Computing Workshops (Euro-Par'11 Workshops), Part I, 2011, pp. 481–490.

[14] NVIDIA Corporation, NVIDIA GeForce GTX 980 (Nov. 2014).
URL `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF`

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU computing, Proceedings of the IEEE 96 (5) (2008) 879–899.

[16] F. Ino, Y. Munekawa, K. Hagihara, Sequence homology search using fine grained cycle sharing of idle GPUs, IEEE Trans. Parallel and Distributed Systems 23 (4) (2012) 751–759.

[17] Y. Okitsu, F. Ino, K. Hagihara, High-performance cone beam reconstruction

using CUDA compatible GPUs, Parallel Computing 36 (2/3) (2010) 129–141.

[18] NVIDIA Corporation, CUDA C Programming Guide Version 7.5 (Sep. 2015).
URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[19] T.-T. Wong, C.-S. Leung, P.-A. Heng, J. Wang, Discrete wavelet transform on consumer-level graphics hardware, IEEE Trans. Multimedia 9 (3) (2007) 668–673.

[20] D. Shreiner, M. Woo, J. Neider, T. Davis, OpenGL Programming Guide, 5th Edition, Addison-Wesley, Reading, MA, 2005.

[21] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, F. Tirado, Parallel implementation of the 2D discrete wavelet transform on graphics processing units: Filter bank versus lifting, IEEE Trans. Parallel and Distributed Systems 19 (3) (2008) 299–310.

[22] J. Franco, G. Bernabé, J. Fernández, M. Ujaldón, The 2D wavelet transform on emerging architectures: GPUs and multicores, J. Real-Time Image Processing 7 (3) (2012) 145–152.

[23] V. Galiano, O. López, M. P. Malumbres, H. Migallón, Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs, J. Supercomputing 64 (1) (2013) 4–16.

[24] M. Kucis, D. Barina, M. Kula, P. Zemcik, 2-D discrete wavelet transform using GPU, in: Proc. 26th Int'l Symp. Computer Architecture and High Performance Computing Workshop (SBAC-PADW'14), 2014, pp. 1–6.

[25] B. Sharma, N. Vydyanathan, Parallel discrete wavelet transform using the open computing language: a performance and portability study, in: Proc. 24th IEEE Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW'10), 2010.

[26] Khronos OpenCL Working Group, The OpenCL specification version 1.1, `http://www.khronos.org/registry/cl/` (Jun. 2011).

[27] N. Metropolis, G.-C. Rota, Witt vectors and the algebra of necklaces, Advances in Mathematics 50 (2) (1983) 95–125.

[28] W. Burnside, Theory of Groups of Finite Order, 1897.

[29] T. Acharya, P.-S. Tsai, JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures, Wiley-Interscience, 2004.

[30] J.-W. Kim, J. Song, S. Lee, I.-C. Park, Tiled interleaving for multi-level 2-D discrete wavelet transform, in: Proc. 40th IEEE Int'l Symp. Circuits and Systems (ISCAS'07), 2007, pp. 3984–3987.

[31] G. Deslauriers, S. Dubuc, Symmetric iterative interpolation processes, Constructive Approximation 5 (1) (1989) 49–68.