FAST AND EFFICIENT LOSSLESS IMAGE COMPRESSION BASED ON CUDA PARALLEL WAVELET TREE ENCODING

by

Jingqi Ao, B.S.E.E, M.S.E.E

A Dissertation

In

ELECTRICAL AND COMPUTER ENGINEERING

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dr. Sunanda Mitra
Committee Chair

Dr. Brian Nutter, Co-chair

Dr. Ranadip Pal

Dr. Mark A. Sheridan
Dean of the Graduate School

May, 2014

Copyright 2014, Jingqi Ao

ACKNOWLEDGMENTS

First, my gratitude is due to my advisor Dr. Sunanda Mitra. Dr. Mitra is always there when I need help and she is more like a mentor rather than a professor to me. One of most valuable things I learned from her is her spirit, never stop learning new things and never quit. I would also like to thank Dr. Brian Nutter and Dr. Ranadip Pal. They are like my elder friends. Besides knowledge, they selflessly share their experiences and ideas with me. I also really appreciate their valuable time spent as my committee members.

Then, I would like to thank my friends at CVIAL: Dr. Enrique Corona, Bharath Lohray and Xiangyuan Zhao. From you, I learnt about different cultures, different countries, and different opinions. You let me know how huge the world is and how broad our field is.

Last but not the least, I would like to thank my parents. I miss you so much. Thank you for your support and understanding during my six-year study. We all made sacrifices but I always know that without you, there is no chance for this dissertation.

TABLE OF CONTENTS

ABSTRACT

Lossless compression is still in high demand in medical image applications despite improvements in the computing capability and decrease in storage cost in recent years. With the development of General Purpose Graphic Processing Unit (GPGPU) computing techniques, sequential lossless image compression algorithms can be modified to achieve more efficiency and speed. Backward Coding of Wavelet Trees (BCWT) is an efficient and fast algorithm, utilizing Maximum Quantization of Descendants (MQD) and it is quite suitable for lossless parallel compression because of its intrinsic parallelism and simplicity. However, the original implementation of BCWT is a CPU-based sequential codec and that implementation has multiple drawbacks which hinder the parallel extension of BCWT. Parallel Coding of Wavelet Trees (PCWT) modifies the BCWT from theoretical workflow to implementation details. PCWT introduces multiple new parallel stages, including parallel wavelet transform stage, parallel MQD calculation stage, parallel Qmax search stage, parallel element encoding stage and parallel group encoding stage, and change the encoding sequence from backward to forward. All those stages are designed to accelerate the compression process. PCWT implementation is designed with the consideration of Compute Unified Device Architecture (CUDA) hardware constrains and implementation scalability. With newly designed workflow and highly optimized parallel stages, PCWT performs faster than the lossless JPEG-XR algorithm, the current standard, with comparable compression ratios. Multiple possible improvements in speed and flexibility on PCWT are also proposed as future work.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The explosion of image data nowadays has demanded advanced image compression algorithms to be implemented in both image transmission and image storage stages. For example, for medical images, 600 million imaging procedures are performed in United States every year and they are taking more space than before since more complex imaging techniques have been applied [1]. Cloud storage has been proposed to address the efficiency problem of image storage and access and image compression algorithm is one of key techniques for cloud-based solution. Cloud-based solution requires scalability and consistency, which implies multiple copies of large image datasets [2]. Cloud-based solution also allows flexible data access, which means data should be able to be accessed conveniently with Wi-Fi and 3G/4G connections [2]. However, those accesses are relatively slow and expensive. In general, data transmission time during the access should be short, which requires the image data should be compressed and reusable.

One of most popular standards for image compression is Joint Picture Expert Group (JPEG) [3]. It has been widely used in digital cameras and on Internet. However, JPEG is intrinsically lossy because it is based on the lossy Discrete Cosine Transform (DCT) [3] and it is not suitable for many high-precision-required applications. The successor of JPEG is JPEG2000 [4], with the ambition of improving JPEG with another transform, i.e. Discrete Wavelet Transform (DWT) [4]. However, JPEG 2000 is overly complicated and is not widely applied in image processing field. The latest image compression standard is JPEG-XR [5], utilizing so called hierarchical two-stage Lapped Biorthogonal Transform (LBT) [6]. LBT is similar to DCT except it can be lossless. JPEG-XR is quite new and still needs time to be widely accommodated.

One distinguishing feature of DWT is the pyramid structure of its result [7]. DWT has the concept “level” and pyramid structure of its result reflects the

independency within one level and dependency between neighboring levels. Dependency between neighboring levels can be utilized in compression and more importantly, independency within one level can be utilized to achieve high degree of parallelism, which implies fast compression. In this dissertation, Parallel Coding of Wavelet Trees (PCWT) is proposed, as a DWT-based compression algorithm, to achieve high speed compression, with competitive compression efficiency to JPEG-XR standard.

DWT-based compression generally includes two procedures: compression procedure and the corresponding decompression one. The compression procedure usually has two phases: forward transform phase and encoding phase. The transform phase is to generate wavelet coefficients via forward DWT and the encoding phase is to encode those coefficients into bitstream. The decompression procedure also includes two phases: decoding phase and inverse transform phase. The decoding phase is to restore wavelet coefficients from bitstream and the inverse transform phase is to restore original data by performing inverse DWT.

DWT-based compression can be either lossy or lossless, depending on whether restored data are mathematically equal to original data. "Mathematical equivalency emphasizes on the numerical identity between restored data and original data. There is a type of lossy compression, called visually lossless compression [8], in which restored data, i.e. restored images, are not the same as original data, i.e. original images, but differences between restored images and original images are difficult to be perceived by human visual system.

Lossless DWT-based compression is the basis of its lossy counterparts. Any change on any part of lossless compression can lead to a lossy compression. Nonetheless, not all types of losses are acceptable. Note that PCWT is implemented as a lossless compression, which can provide a base-line compression for any further modification or improvement. However, it can also be easily implemented in a lossy compression mode.

For PCWT, in transform phase, the reversible (5,3) wavelet transform is used for lossless compression [9]. (5,3) wavelet transform is an integer to integer transform so that integer pixel values from images can be transformed to integer coefficients losslessly. Lifting scheme can be used in the implementation of (5,3) wavelet transform, which requires less memory and computation than the traditional convolution method [10].

In encoding phase, Backward Coding of Wavelet Trees (BCWT) [11] is chosen among other wavelet coefficient encoding algorithms such as SPIHT [12], or EBCOT [13]. BCWT is an efficient and fast algorithm, by utilizing Maximum Quantization Descendant (MQD) in its one-pass encoding procedure. However, the original BCWT implementation is a CPU-based sequential program. In PCWT, its main concepts have been maintained but several main steps have been modified to achieve high parallelism.

Recently, General Purpose Graphic Processing Unit (GPGPU) computing has been increasingly used to accelerate traditionally sequential procedures [14]. GPGPU computing has been proposed to be utilized in some image compression standards, such as JPEG2000 [15] and JPEGXR [16]. However, some aspects of those standards were not designed with the consideration of parallelism, e.g. entropy coding, which are difficult to be parallelized. On the other hand, BCWT can be a good candidate for lossless parallel wavelet transform based compression because of its intrinsic parallelism and simplicity [11]. That is also an important reason why BCWT is chosen as the encoding phase.

The main research contributions of this dissertation are 1) development of an accelerated compression algorithm i.e., PCWT within the context of GPGPU demonstrating a GPGPU approach to solve existing sequential problems, 2) modifications and re-designs made to eliminate or at least relieve the impacts of GPGPU disadvantages on our final algorithm, without largely sacrificing its advantages thus including Block-wise Lifting-scheme Discrete Wavelet Transform with Boundary processing (B-LDWT+B) and two-stage encoding and excluding line-

based implementation of BCWT [17] due to its nonadaptive, irregular result generation pattern and intensive I/O accesses, 3) backward coding for MQD computation only, otherwise, forward coding of wavelet trees, and 4) superior performance of PCWT over current standard, JPEG-XR, for large images (over 20 mega pixels) in encoding speed.

This dissertation is organized as follows: Chapter 1 briefly introduces the importance of image compression and reviews the development of image compression standard. It also provides an overview of the proposed PCWT and relevant techniques and outlines the main contributions of this dissertation work. Chapter 2 illustrates background knowledge about BCWT, i.e. the predecessor of PCWT, and the GPGPU computing model used in this dissertation. Chapter 3 and 4 describe the implementation details of PCWT in terms of compression procedure and decompression procedure, respectively. Chapter 5 demonstrates test results of PCWT, which is compared with those of a mainstream JPEG-XR implementation. Chapter 6 concludes the dissertation by providing the conclusion of dissertation and possible future work.

CHAPTER II

BACKGROUND OF BCWT AND GPGPU MODELS

2.1 The Sequential BCWT

The BCWT was originally implemented as a sequential CPU-based algorithm. One major improvement brought by BCWT is the simplified encoding algorithm, in which multiple wavelet tree searches are eliminated. The encoding is only based on MQD, which allows the local encoding. MQD is defined in [17] as:

$$m_{i,j} = \begin{cases} q_{O(i,j)}, \\ \quad \text{if } (i,j) \text{ is in level 2 subbands}, \\ \max\{q_{O(i,j)}, \max\limits_{(k,l)\in O(i,j)}(m_{k,l})\}, \\ \quad \text{otherwise} \end{cases}$$

in which

$c_{i,j}$: The wavelet coefficient at coordinate (i, j).

O(i, j) : A set of coordinates of all the offspring of (i,j).

D(i, j) : A set of coordinates of all descendants of (i,j).

L(i, j) = D(i, j) – O(i, j) : A set of coordinates of all the leaves of (i,j).

$q_{i,j} = \begin{cases} \left\lfloor \log_2 \left| c_{i,j} \right| \right\rfloor & ,\text{if } \left| c_{i,j} \right| \geq 1 \\ -1 & ,\text{otherwise} \end{cases}$: The quantization level of the coefficient $c_{i,j}$.

$q_{O(i,j)} = \max\limits_{(k,l)\in O(i,j)} \{q_{k,l}\}$: The maximum quantization level of the offspring of (i,j).

$q_{L(i,j)} = \max\limits_{(k,l)\in L(i,j)} \{q_{k,l}\}$: The maximum quantization level of the leaves of (i,j).

Local encoding has two meanings: one is within each coefficient band, to encode each MQD based unit, i.e. one MQD and its corresponding four coefficients, only the MQD based unit and one higher-level MQD are involved. Neither extra neighboring coefficient nor MQD is needed. The other meaning is to encode bands on

any level, only two levels, i.e. the current level and the adjacent higher level, are involved. There is no operation that involves more than two levels.

Local encoding introduces parallelism in BCWT because multiple local encoding operations can run simultaneously. However, even though the parallelism of BCWT was mentioned [11], there is no parallel implementation of BCWT, to the best of knowledge of the author.

One possible difficulty that hinders parallel implementation of BCWT is the parallel generation of bitstreams. Bitstream generation is usually the bottleneck of most image processing algorithms, because no matter how many parts of an image can be processed in parallel, the final bitstream is always sequential and can only be generated sequentially. Another possible difficulty is the cost of penalization. In this dissertation, GPGPU computing model is chosen as the parallel computing model. However, there are multiple new implementation issues for GPGPU computing model to implement BCWT, which do not exist or have little effects in CPU-based sequential implementation. For example, one of most significant issues is the decomposition level of forward wavelet transform used in BCWT. In CPU-based sequential implementation, the number of decomposition levels can be very high, which generates small size of bands on the top level. But in GPGPU computing model, those small bands can jeopardize the overall performance of algorithm. Another example is the use of MQD in BCWT. In one BCWT CPU-based sequential implementation [17], MQD is used to encode coefficients for all bands except top-level bands, i.e. LL, HL, LH and HH bands on the top level. Those bands have their own encoding rules. However, in this dissertation's implementation, pre-allocation of GPU memory prefers estimable and unified MQD-based coefficient encoding rules, by which all coefficients are encoded in the same way so that the overall required size of generated bitstream is easy to be estimated.

In BCWT CPU-based sequential implementation, the encoding sequence is backward, which is from the lowest level to the highest level of coefficient matrix. The backwardness of BCWT aims to decrease resource usage of the algorithm, more specifically, low memory usage during the compression procedure. The encoding of one band follows the generation of that band so that the band can be cleared from the memory after it has been encoded. This allows memory re-use, which is quite suitable for limited memory situations. However, in the implementation described in this dissertation, the running speed of the implementation has the highest priority when any trade-off needs to be made. Memory efficiency is considered when the implementation is designed but only because of hardware constraints. GPGPU computing model emphasizes on launching massive "workers" on the task to achieve high throughput. Even if each worker in GPGPU computing model has the same memory efficiency as the single worker in the sequential implementation, the overall memory usage of GPGPU computing model is still much higher than that of its CPU sequential counterpart because of the massive number of workers. In practice, a single worker of GPGPU implementation usually has less memory efficiency than its CPU counterpart because of GPU's memory fetching mechanism and synchronization limit, which will be explained in details in Section 2.2. Since the memory does not have the top priority during the implementation design in this dissertation, the forward encoding sequence becomes promising. The thumbnail image encoding can be achieved by forward encoding, with much less difficulties than backward encoding.

BCWT has made improvements over many existing wavelet-tree based algorithms by simplifying encoding mechanism, without decreasing compression efficiency [11]. However, BCWT needs further modifications on itself to be efficiently parallelized. Those modifications, which constitute PCWT, will be explained in details in Chapter 3 and 4.

2.2 General Purpose Graphic Processing Unit (GPGPU) Computing

General Purpose Graphic Processing Unit (GPGPU) Computing is to utilize the programmable graphics processing unit to perform general computing task, rather than its designated graphics related work.

GPGPU computing has two major models: Computation Unified Device Architecture (CUDA) and OpenCL. CUDA [18] is a proprietary model which is solely supported by Nvidia and OpenCL [19] is managed by Khronos Group which includes multiple vendors. However, CUDA has generally better supports than OpenCL, in terms of books, documentations and forums. In this dissertation, CUDA is chosen as the GPGPU computing model to implement PCWT.

CUDA includes a large range of topics and multiple books are even dedicated to CUDA itself. In this dissertation, CUDA is utilized but not the subject itself, so only directly relevant CUDA features are introduced in this section, which are CUDA memories, threads and synchronization.

2.2.1 CUDA Memories

Figure 2.1 shows the memory structure of CUDA [20], which include multiple different types of memories: global memory, shared memory, constant memory, local memory, etc. Among them, global memory, shared memory and local memory are used in the PCWT implementation.

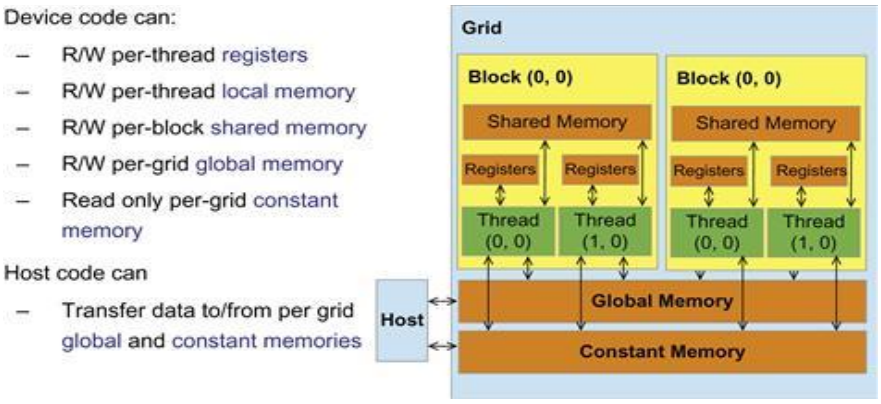


Figure 2.1 CUDA memory architecture [20]

Global memory has the largest volume among all those three types of memories but the slowest access speed. The number of global memory accesses in any algorithm should be as few as possible. Furthermore, the access of global memory should be coalesced, which means multiple consecutive bytes should be accessed at the same time by algorithm. For CUDA computation capability 2.0 devices, in each global memory read/write, 128 bytes are accessed [21]. Figure 2.2 shows a comparison between coalescing and non-coalescing global memory access. Suppose one algorithm needs interlaced bytes from two 128-byte blocks at different places. If one implementation does not consider coalescing access and fetch the needed data only when the data is required, four memory reads are required. For each read, only 64 bytes out of fetched 128 bytes are actually used, however, totally 128 bytes are still fetched from global memory. On the other hand, if coalescing access is considered and all 256 bytes are pre-fetched by an implementation, only two global memory accesses, rather than four, are needed.

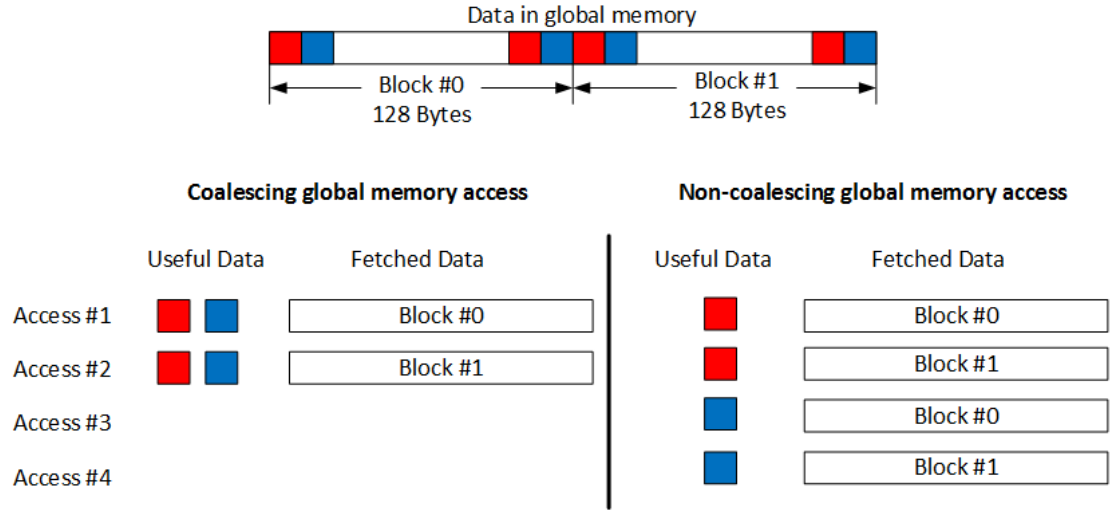


Figure 2.2 Comparison between coalescing and non-coalescing memory access of 256-byte data in the global memory

Shared memory is attached to each CUDA block and has less size than the global memory but higher access speed. CUDA block is a software term that will be explained with threads and synchronization. It can be understood as a working unit

with limited resources, e.g. limited shared memory attached to it. Different from global memory, shared memory does not require coalescing access. Shared memory is usually the place where intensive, random memory accesses happen. Note that shared memories belonging to different CUDA blocks are independent, meaning that "workers" in one block can only access to the shared memory of that block. "Workers" from one block cannot access the shared memory of another block. Shared memory will be further discussed with synchronization.

Local memory has the fastest access speed but highly limited volume. Each worker (i.e. thread) has its unique local memory. The limited volume of local memory usually constrains the use of it. For CUDA computation capability 2.0 devices, each worker has very limited local memory, i.e. only 63 32-bit registers and many of them are used by compiler without any notice to programmers. The available registers for programmers are much less than 63 and the exact number depends on how a specific program is compiled. Local memory should be used for distinguishing values of each thread, which are unique for each thread and need to be stored for further use. Use local memory to store input or output is neither efficient nor even possible in many cases.

2.2.2 Threads and Thread Divergence

After data is stored in memories of devices, "workers" are sent to process those data and generate the desired output. In CUDA, those workers are defined as threads. Threads are software abstract of working items supported by hardware, so it has both software and underlying hardware meanings. On hardware level, each thread represents a basic working unit in a CUDA core resident in a GPU multi-processor. One GPU card usually has multiple multi-processors. For example, Nvidia GTX 570 GPU has 15 multiprocessors, in each of which there are 32 CUDA cores (also called "wraps" in some CUDA literatures) and each CUDA core launches 32 threads at the same time. However, calculating the maximum parallel threads running at any given time by $15 \times 32 \times 32 = 15360$ is not accurate, because not all 32 cores on one multi-processor can run at the same time. CUDA hardware scheduling details are never

disclosed thoroughly, to the best of knowledge of the author. Based on some available materials [22], only a small number of 32 CUDA cores on each multi-processor can run at the same time. The reason why there are more CUDA cores existing than the number they can run simultaneously is that GPU can switch between CUDA cores when one specific core is idling for resources or data, e.g. waiting for a large amount of data read from memory.

In our implementation, the thread is more often referred as its meaning on software level. On software level, the whole task is divided into multiple CUDA blocks and in each block multiple threads execute their designated work independently. When all threads in one block finish their work, that block is done. When all blocks are done, the task is finished. In CUDA, each block has both thread and shared memory limits: for computation capability 2.0 device (e.g. GTX570), each block can launch at most 1024 threads, with 48kB shared memory. For example, if one block is used to process a 100x100 monochrome 32-bit image (i.e. single color per pixel and each pixel is 32-bit), data of that image can be copied to the block's shared memory and at most 1024 threads can be used to process all 10,000 pixels, which is about 10 pixels per thread.

The number of blocks used to process a specific task and the number of threads per block are determined by multiple factors, such as hardware constraints and algorithm design. In some cases, multiple blocks are mandatory. For example, if one image has large size, the shared memory of one block cannot hold the complete image and the whole processing has to be divided into multiple blocks. For 1000x1000 monochrome 32-bit images, the number of blocks should not be less than 976. In other cases, multiple blocks are the result of optimization. For example, multiple blocks are required if more threads are needed. To process the same 100x100 image to achieve higher processing speed, ten blocks with total 10,000 threads can be launched so that one thread in each block only needs to process one pixel. In practice, 10 times thread launching can achieve 2-5 times acceleration due to the fixed overhead of memory IO.

Note that CUDA threads are not completely independent. Each CUDA core launches 32 threads at the same time and those 32 threads are expected to do the same work at any given time. If any thread of those 32 threads performs different work from other threads, thread divergence happens. For example, if 32 threads of one CUDA core are used to process 32 elements (each thread for one element), even index threads (#0, #2, #4, … #30) adds one to their corresponding elements and odd index threads (#1, #3, #5, …, #31) subtracts one from their corresponding elements, thread divergence happens between even- and odd-index threads. To react to this divergence, CUDA runtime launches those 32 threads twice. In the first run, 32 threads are launched but only even-index threads do the real work i.e., adding one to elements, and odd-index threads are idling. In the second run, 32 threads are launched again, but only odd-index threads do the real work i.e. subtracting one from elements and even index ones are idling. It can be found that thread divergence affects the executing speed adversely. If all 32 threads perform adding or subtracting task, there is no thread divergence and only one run is needed. But with thread divergence, one extra run is needed. If the thread divergence is more severe, e.g. threads are sent to do 32 different tasks, the performance could be further degraded. Note that it is difficult to achieve zero thread divergence in most algorithm implementations but explicit thread index based branches, like the odd- and even-index thread example above, should be avoided whenever possible.

2.2.3 Synchronization

Blocks are not only used to provide parallel thread launching and fast shared memory, but they are also the synchronization basis of any task. Synchronization provides a deterministic state of execution of multi-thread tasks. Generally, the execution state of multi-thread task is undetermined. For example, if 128 threads are used to add one to each element of a 128-element array, thread #0 is not necessarily the first thread to start or finish. Thread #127 could be the first thread to launch due to the CUDA core scheduling. If the following execution depends on the completion of all 128 threads, synchronization is needed. Synchronization of all 128 threads means

the execution will halt until all 128 threads finish their work, so that the following execution can start without the risk that any thread within those 128 threads has not finished yet. However, only threads within the same blocks can be synchronized. For example, if 10 blocks with 1,000 threads in each block are used to process 10,000 pixels, only 1,000 threads within the same block can be synchronized. Thread #0 can be synchronized with thread #999 in the block #0; but thread #0 in block #0 cannot be synchronized with thread #0 in block #1. There is no guarantee that which block runs or finishes first after multiple blocks are launched (e.g. block #0 could run before block #1 and block #1 could run before block #0). This limited synchronization within the same block affects the design of algorithm, meaning tasks assigned to each block should not have any dependency on each other. Threads from different blocks can be synchronized only after one kernel is finished. Kernels can be viewed as parallel functions with block configuration attached to it. It can be understood as threads from different blocks can be synchronized only after all those blocks are finished. Kernel launching has its own overhead so launching too many kernels only for thread synchronization is not efficient. Parallelization of sequential algorithms is usually involved with execution flow change or even theoretical modification because of the limited synchronization of CUDA framework.

GPU memories and their accesses, thread launching, execution mechanisms and synchronization are the three main CUDA factors which affect re-design and implementation of the PCWT algorithm. These factors will be pointed out when the detailed implementation is explained in following chapters.

CHAPTER III

PCWT COMPRESSION PROCEDURE

The compression procedure includes six main stages: parallel Block-based Lifting-scheme Discrete Wavelet Transform with Boundary processing (B-LDWT+B), parallel MQD calculation, parallel QMAX search, parallel element encoding, parallel group encoding and sequential bitstream output.

3.1 Parallel B-LDWT+B

Parallel B-LDWT+B is to generate Lifting-scheme Discrete Wavelet Transform (LDWT) coefficients for following encoding stages. It is based on (5,3)-LDWT, with boundary processing and block configurations. Before explaining B-LDWT+B, the traditional (5,3)-LDWT is reviewed.

3.1.1 (5,3)-LDWT

For an array with $2N$ elements, 1D (5,3) forward LDWT can be calculated by [9]:

H-component (Odd-index elements): $\mathrm{D[2i+1]} = \mathrm{D[2i+1]} - \lfloor(\mathrm{D[2i]} + \mathrm{D[2i+2]})/2\rfloor$, $\mathrm{i} = 0,..,\mathrm{N}-1$. (Eq. 3-1)

L-component (Even-index elements): $\mathrm{D[2i]} = \mathrm{D[2i]} + \lfloor(\mathrm{D[2i-1]} + \mathrm{D[2i+1]} + 2)/4\rfloor$, $\mathrm{i} = 0,\dots,\mathrm{N}-1$. (Eq. 3-2)

And the corresponding inverse 1D LDWT can be calculated by:

L-component (Even-index elements): $\mathrm{D[2i]} = \mathrm{D[2i]} - \lfloor(\mathrm{D[2i-1]} + \mathrm{D[2i+1]} + 2)/4\rfloor$, $\mathrm{i} = 0,\dots,\mathrm{N}-1$. (Eq. 3-3)

H-component (Odd-index elements): $\mathrm{D[2i+1]} = \mathrm{D[2i+1]} + \lfloor(\mathrm{D[2i]} + \mathrm{D[2i+2]})/2\rfloor$, $\mathrm{i} = 0,..,\mathrm{N}-1$. (Eq. 3-4).

The 2D (5,3) LDWT can be divided into two phases, i.e. 2D horizontal LDWT and 2D vertical LDWT, which are both depends on 1D (5,3) LDWT. Generally, 2D forward LDWT performs 2D forward horizontal LDWT first and 2D forward vertical

LDWT second. Correspondingly, the 2D inverse LDWT performs 2D inverse vertical LDWT first and 2D inverse vertical LDWT second.

Four implementation issues are needed to be addressed when implement 2D LDWT with CUDA:

(1) 1D forward LDWT expects even number of working elements on each decomposition level. For N-level 2D forward LDWT, for the working matrix with size $R \times C$, one common step is to pad the row and column number of the working matrix to the smallest power of 2 integer (2^N) that is not less than R and C. However, this type of padding causes unnecessary computation. For example, to perform 2-level 2D forward LDWT on matrix 129x129, the matrix is padded to 256x256. Only about 25% elements are effective elements.

(2) When (5,3)-LDWT is performed on an array with finite $2N$ elements $D[j]$ ($j = 0, \ldots, 2N-1$), two elements $D[2N]$ and $D[-1]$ do not exist, when compute H-components and L-components, respectively.

(3) CUDA can only provide limited shared memory for either horizontal or vertical (5,3)-LDWT. For Computation Capability 2.0 devices, the shared memory limit for each block is 48kB. Using shared memory is critical for obtaining the correct forward LDWT or inverse LDWT result fast. Shared memory allows much faster memory access than the global memory and synchronization of threads from the same block. Synchronization ensures that H- and L-component computations are performed deterministically and guarantees the correctness of the generated result.

If a specific GPU hardware does not have enough shared memory to process a complete row or column of 2D (5,3)-LDWT, without proper modifications, the direct block-wise LDWT implementation introduces block effects on the boundary of each block [15].

(4) Forward (5,3)-LDWT generates interlaced decomposition results in which L-components are interlaced with H-components. In practice, generated L- and H-

components are expected to be clustered as the intermediate result for the next level decomposition or as the final result to be presented.

To solve the issues mentioned above, Block-wise LDWT with Boundary processing (B-LDWT+B) is proposed to perform fast and efficient LDWT without loss and block effect.

3.1.2 Parallel B-LDWT+B

Figure 3.1 shows an example of forward 1D B-LDWT+B transform with boundary processing and I2C operation, on a 12 elements with 3 blocks. The rest of this section will explain how steps in 1D B-LDWT+B are introduced to solve the issue (1) to (4) mentioned in the last section (Section 3.1.1).

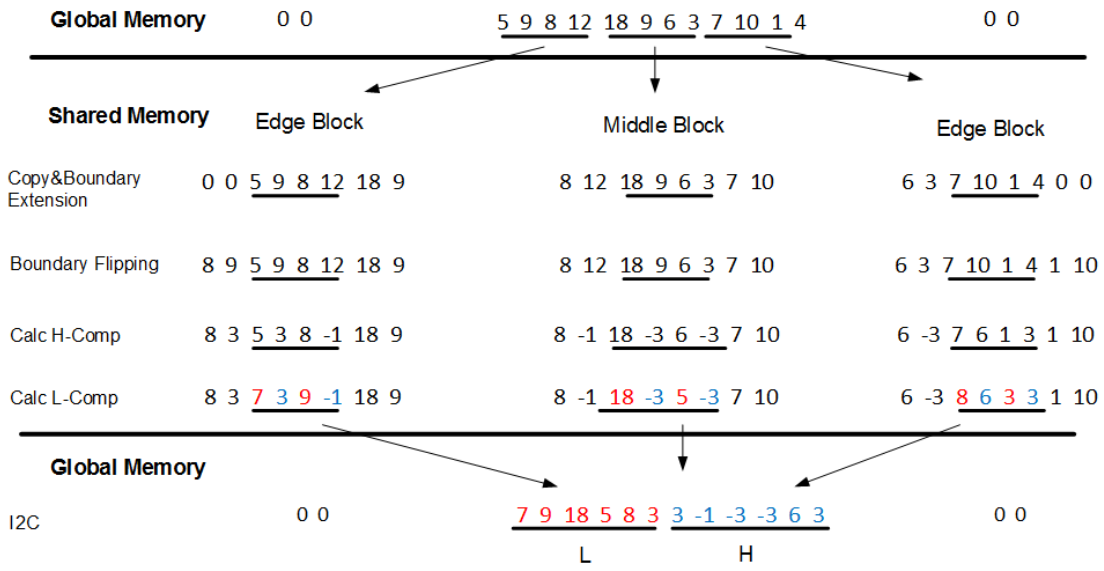


Figure 3.1 Forward 1D B-LDWT+B transform on 12 elements with 3 blocks

To solve issue (1), the Maximum Allowed Decomposition Level (MADL) of 2D forward LDWT and band/level size list are proposed.

Theoretically, with proper padding, 2D forward LDWT can be performed continuously until 1x1 LL band is achieved. However, more levels 2D forward LDWT performs with, smaller sizes of top level bands are generated. Lacking of enough elements in those bands limits the parallelism of computation of those bands. The

limited benefits from parallelism of small highest level bands can be overwhelmed by cost of transforming data between host (CPU) and device (GPU) in terms of running time, which means small sizes of top bands affect the performance of our parallel algorithm adversely. The MADL is proposed to constrain the level used for 2D forward LDWT, based on the size of working matrix. It means matrices with different sizes have different MADLs.

The MADL can be calculated based on the pre-set minimum row and column number of the highest level band size LL_MIN_R and LL_MIN_C. For example, if LL_MIN_R is set to 64, which means the row number of LL band should be equal or larger than 64. For images with size 1024x1024, the maximum allowed decomposition level of DWT is 4, because $\frac{1024}{2^4} = 64$ but $\frac{1024}{2^5} = 32 < 64$.

The MADL aims at improving performance of B-LDWT+B implementation, but it also helps the implementation to avoid over-computation by not performing unnecessary decompositions that is above MADL.

Note that over-computation still exists in decompositions within MADL, and level/band size is proposed to handle that. “Level size” is defined as the row number and column number the input matrix for one level 2D LDWT. “Band size” is defined as the row number and column number of bands generated by 2D LDWT. In standard 2D LDWT implementation, the level size and band size are “interchangeable”, which are only different with a factor of 2. However, in our algorithm, level sizes and band sizes are relative but calculated differently. Level sizes are used in both forward and inverse LDWT and band sizes are mainly used for encoding (i.e. band-based encoding) and decoding.

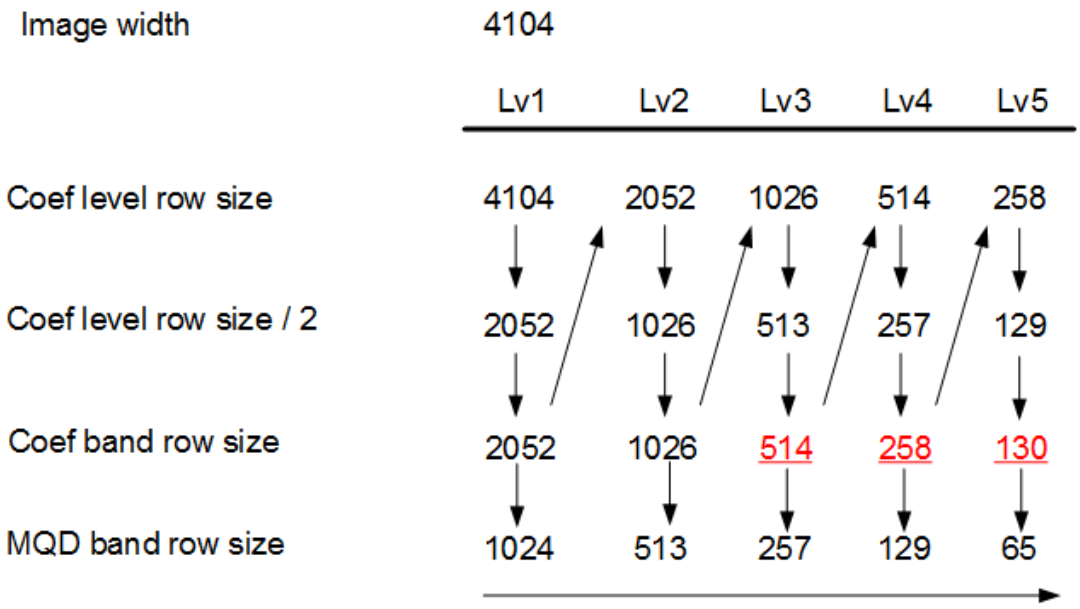


Figure 3.2 Band and level size calculation

Figure 3.2 shows an example of difference between level size and band size.

Differences rise from different rules on which level sizes and band sizes are calculated.

For level size, the level size of any level must be even. In Figure 3.2, the level row size at Lv4 would be 513 if directly calculated from Lv3 ($\frac{1026}{2} = 513$). The actual level size at Lv4 is padded to 514 with one row of 0, to ensure the Lv4 LDWT decomposition can be done. However, the level row size at Lv3 is still maintained as 1026 rather than 1028 ($2 \times 514 = 1028$). The potential over-computing of forward DWT on Lv3 is avoided.

On the other hand, for band size, the band size of any band must be even. This is from the assumption of following encoding stages, which is "every band should have corresponding MQD band". In Figure 3.2, the band's row size at Lv5 would be 129 ($\frac{258}{2} = 129$) but is padded to 130. However, the band's row size at Lv4 is still 258 rather than 260 ($2 \times 130 = 260$) so the potential over-computing and further over-encoding in following encoding stages on Lv4 is avoided.

There is another parameter related with level and band size, i.e. the band offset. It will be explained in Section 4.4.

To address issue (2), boundary processing is needed to substitute the missing element $D[2N]$ and $D[-1]$ in the data array. In B-LDWT+B, two types of boundary processing methods, boundary flipping and boundary extension, are used. Choosing these methods instead of using simple padding zero or mirroring boundary is because of the block constraints in solution of issue (3). The details of boundary flipping and boundary extension will be explained when addressing the issue (3).

To address the issue (3), both horizontal 2D B-LDWT+B and vertical 2D B-LDWT+B have to be processed block by block. Blocks here refer to CUDA blocks but they can be conceptual blocks of image processing fields. Directly dividing the input 2D matrix into multiple blocks and processing those blocks independently without proper boundary processing brings the boundary effect to the DWT decomposition result. Such rough block division will also cause the "fixed block size" problem that is the block size used in forward transform could hinder the execution of inverse transform. For example, if the hardware used in forward transform has enough memory to process blocks with size 512x512 but the hardware used in inverse transform does not have enough memory to process the same size blocks, the lossless inverse transform is not able to proceed.

Boundary flipping and boundary extension are used in B-LDWT+B to achieve block-wise lossless LDWT with arbitrary block configuration. As shown in Figure 3.1, blocks in our B-LDWT+B implementation can be classified into two categories: edge blocks and middle blocks. For edge blocks, boundary-flipping (mirroring the boundary elements) is used on one end and boundary extension (reading elements outside the current block) is used on the other end. For middle blocks, boundary extensions are used on both ends. Figure 3.1 shows an example of boundary flipping and boundary extension of edge and middle blocks.

To solve the issue (4), embedded Interlacing-to-Clustering (I2C) operation is introduced to cluster the L- and H-components. "Embedded" means this operation is embedded in both L- and H-component calculation so that there is no extra clustering step after L- and H-component calculations are finished. As shown in Figure 3.1, the clustered result (L-components are marked in red and H-components are marked in blue) is stored in the global memory as the result of 1D B-LDWT+B, without an extra clustering operation involved.

Comparing to the alternative independent I2C operation, there are several advantages of embedded I2C operation:

(1) The kernel launching overhead [23] of extra I2C operation can be avoided;

(2) Implementation of embedded I2C operation is simpler than independent I2C operation. Each embedded I2C operation is performed in either horizontal or vertical direction, which means it needs to handle only two types of components, i.e. L-components and H-components. The independent I2C operation clusters elements in both horizontal and vertical direction, so it has to process four types of components, i.e. LL-, HL-, LH- and HH-components;

(3) The number of global memory operations of the embedded I2C operation is less than those of independent I2C operation. Independent I2C operation needs extra global memory access to read the interlaced result into shared memory, process it and write the clustered result back to global memory. Those extra global memory accesses will decrease the overall running speed of B-LDWT+B.

2D B-LDWT+B is a variant of (5,3)-LDWT, with boundary processing, block configurations and embedded clustering operations. Different from traditional CPU-based 2D (5,3)-LDWT, in which identical 1D LDWTs are performed on different directions i.e. horizontal and vertical directions sequentially, implementations of horizontal and vertical part of 2D B-LDWT+B has significant differences. Section 3.1.3 and 3.1.4 will describe the details of horizontal and vertical part of 2D B-LDWT+B and demonstrate those differences.

3.1.3 Implementation of Horizontal 2D B-LDWT+B

Horizontal 2D B-LDWT+B includes six steps:

(1) The input matrix, i.e. the original image for the first level decomposition or the LL band from the previous decomposition, is divided into blocks. One block contains a part of row. For input matrix size $R \times C$, the block configuration is $R \times \left\lceil \frac{C}{b} \right\rceil$, in which b is the number of element processed by one block.

(2) Blocks are read from GPU global memory (global memory 2D array #1) into blocks' corresponding shared memories. For each block, four extra elements outside the block are also read for boundary extension, as shown in Figure 3.1. The elements designated to be processed by the block are called effective elements and the 4 extra elements are called boundary elements. Effective elements and boundary elements constitute the input section of shared memory of that block.

(3) Optional boundary flipping is performed on one end of the input section, if the block is a boundary block.

(4) Calculate H-components.

(5) Calculate L-components. Note the interlaced result is generated after this step and generated H- and L-components are stored in the output section of shared memory.

(6) Perform embedded I2C operations on generated H-elements and L-elements. The interlaced result is copied from 2D array #1 to another global memory 2D array (2D array #2). Extended band offsets, which will be explained in section 4.4, are used to relocate H- and L-elements in 2D array #2.

Note that there are cases that the rightmost blocks has fewer elements than the preset block size. For example, if the width of image is 640 and the block size is 256, then each row is divided into 3 blocks, with 256, 256, and 128 elements, respectively. With proper boundary processing, i.e. boundary extension on the left end and boundary flipping on the right end, those rightmost blocks can proceed through step

(2) to (6) without any padding. It means that the rightmost block with 128 elements does not have to be padded to 256 elements.

Figure 3.3 shows an example of horizontal 2D B-LDWT+B. Each row of the input matrix in global memory 2D array #1 has been divided into three blocks. Block #0 and #1 are complete block and block #2 is the incomplete block which has fewer elements than block #0 and #1. After step 1 and 2, each block has been read into shared memory, with boundary extension that is marked by broken lines. After step 3, 4 and 5, the L- and H-components in each block has been generated. After step 6, clustered L- and H-components are stored in global memory 2D array #2. Note there is a horizontal band offset between generated L and H bands, which will be explained in section 4.4.

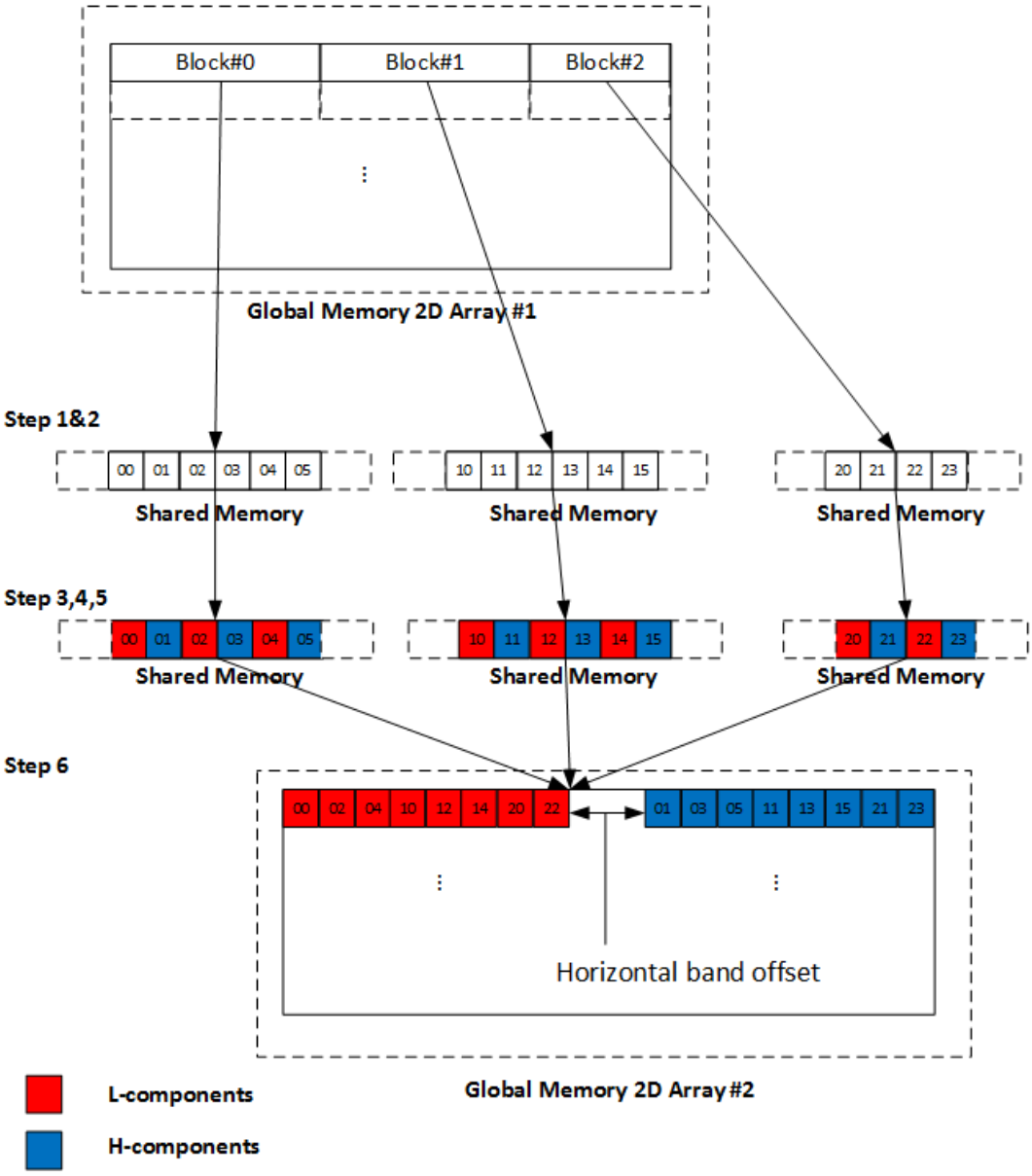


Figure 3.3 Example of horizontal 2D B-LDWT+B

3.1.4 Implementation of Vertical 2D B-LDWT+B

Vertical 2D B-LDWT+B has different implantation from its horizontal counterpart because of the 2D array storage mechanism in GPU global memory and global memory coalescing access. The 2D array is stored row by row in GPU's global memory and 2D array can be only accessed efficiently in a coalescing pattern which

requires threads in one CUDA block to read consecutive global memory (generally 128 bytes [21]).

The efficient way to access elements in 2D array in global memory is to read elements row by row, and the direct access a single column of 2D array is not efficient. Figure 3.4 shows the comparison between column-wise and row-wise 2D matrix processing. Assume element processing in both methods are the same so that the difference is majorly introduced by the coalescing global memory access. To process the input matrix with size 64x32, each fetch of column-wise processing reads a complete column by using 64 memory accesses and most of elements read are dropped. Overall, column-wise 2D matrix processing performs 2048 global memory accesses. On the other hand, the row-wise processing fetches the data by coalescing accesses. Suppose each fetch only reads two rows, i.e. 64 elements which fit the number of elements processed in column-wise processing. Overall, row-wise 2D matrix processing performs only 64 global memory accesses. It can be found that by using row-wise 2D matrix processing method, temporal cost of data I/O, which is notorious performance bottleneck of GPGPU computing [24], is decreased to 1/32. Nevertheless, in some cases, row-wise 2D matrix processing method requires modifications on algorithms to accommodate the data structure fetched into the processing unit, especially when the original algorithm is column-wised, like vertical 2D B-LDWT+B.

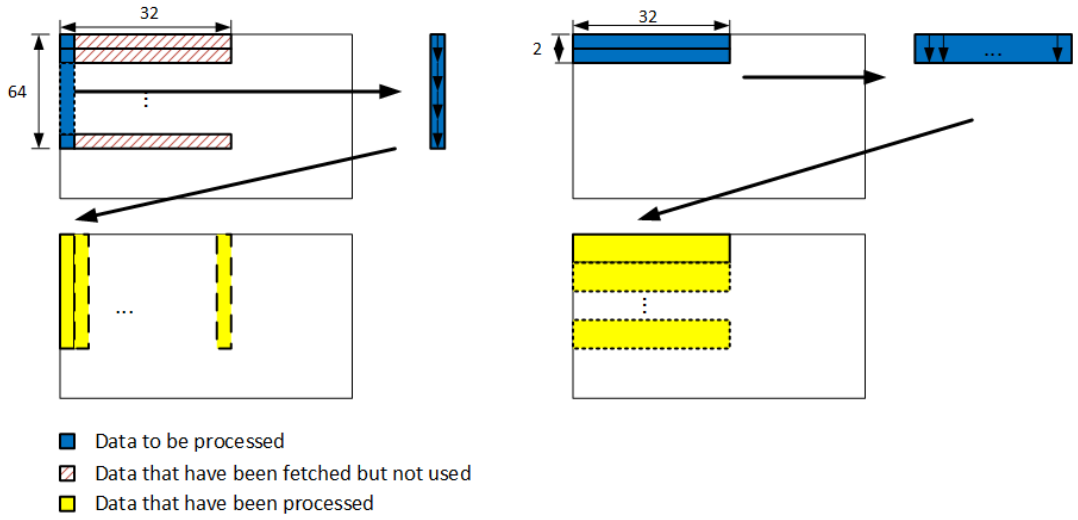


Figure 3.4 Comparison between column-wise and row-wise 2D matrix processing

To increase the efficiency of global memory access in vertical 2D B-LDWT+B, the block used in vertical 2D B-LDWT+B is 2D, which means each block includes elements from multiple rows and columns.

Vertical 2D B-LDWT+B includes six steps:

(1) The input matrix, i.e. the result of the previous horizontal 2D B-LDWT+B, is divided into multiple blocks. One block contains multiple partial columns. For input matrix size $R \times C$, the block configuration is $\left\lceil \frac{R}{r} \right\rceil \times \left\lceil \frac{C}{c} \right\rceil$, in which r and c are the number of elements processed by one block in horizontal and vertical direction, respectively.

(2) Similar to horizontal 2D B-LDWT+B, blocks are read from GPU global memory (global 2D array #2) into the blocks' corresponding shared memory. For each block, four extra rows of elements outside the block are also read for boundary extension.

(3) Optional boundary flipping is performed if the block is boundary block.

(4) Perform H-step of 1D B-LDWT+B.

(5) Perform L-step of 1D B-LDWT+B.

(6) Perform embedded I2C operations on generated H-elements and L-elements. The interlaced result in global 2D array #2 is clustered and copied to global 2D array #1. The clustered result in 2D array #1 can be used for the next level decomposition or output of the whole 2D B-LDWT+B stage.

Figure 3.5 shows an example of vertical 2D B-LDWT+B. The result of horizontal 2D B-LDWT+B has been stored in global memory 2D array #2. Each 2D block contains 4x3 elements. The row number is larger than the column number for each 2D block reflects that in each block transform is performed along each column. Each block should contain as many elements in vertical direction as possible. Similar to the example in Figure 3.3, Block #0 and #1 are complete block and block #2 is the incomplete block which has fewer elements than block #0 and #1. After step 1 and 2, each block has been read into shared memory, with boundary extension that is marked by broken lines. After step 3, 4 and 5, the L- and H-components in each block have been generated. After step 6, clustered L- and H-components are stored in global memory 2D array #1. Note there is also a vertical band offset between generated L and H bands, which will be explained in section 4.4.

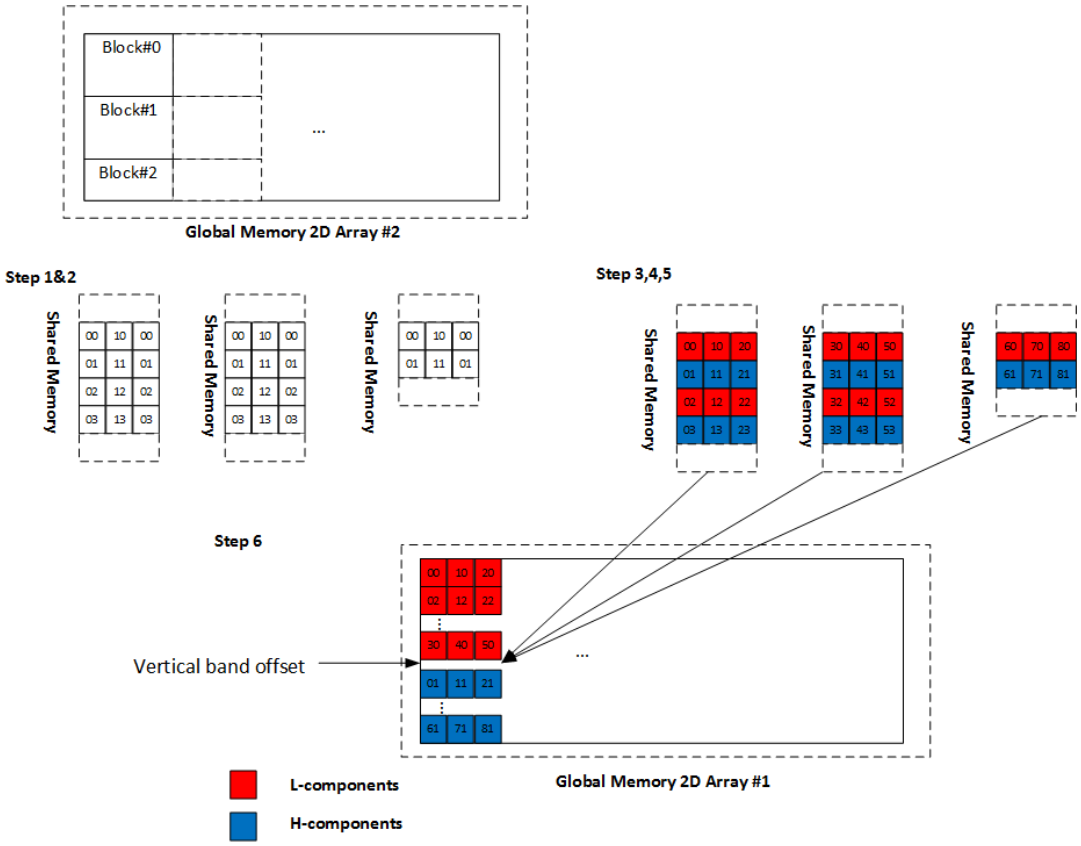


Figure 3.5 Example of vertical 2D B-LDWT+B

Note that in the previous horizontal 2D B-LDWT+B, the block could be also set as 2D. Processing one partial row in 2D horizontal B-LDWT+B needs extra elements i.e. boundary elements, which usually causes extra global memory accesses. To decrease the temporal cost from those extra global memory accesses, each row should be divided into as few partial rows as possible. Since each block's shared memory is limited, in 2D horizontal B-LDWT+B, the block is chosen as 1D, which requires less block number for each row than 2D block.

Similarly, in vertical 2D B-LDWT+B, block number that is used to process each column is expected as little as possible. However, the block's height cannot be arbitrarily large because of limited block shared memory and 128 bytes (thrity-two 32-bit integers) recommended block row width.

3.2 Parallel MQD calculation

3.2.1 Modified MQD calculation

There are multiple methods to encode the generated 2D B-LDWT+B coefficients. In our algorithm, Maximum Quantization Descent (MQD) is used to encode coefficients because of its inherited parallelism. In general, every four coefficients with their corresponding MQDs can be encoded independently.

Different from the original BCWT [11], in our algorithm, all coefficient bands have their corresponding MQD matrix, called MQD bands. It means that all HL, LH, HH and LL (only exists on the top level) bands have their corresponding MQD bands. This simplifies the encoding procedure and decreases the difficulties in implementing our algorithm.

The generation of MQD matrix is from the lowest level to the highest level. There are two types of MQD bands:

(1) MQD band without corresponding lower level MQD band's support and

(2) MQD band with corresponding lower level MQD band's support.

Type (1) MQD bands are MQD bands corresponding to coefficient bands on the lowest level and the LL band. Type (2) bands are MQD bands corresponding to the remaining coefficient bands. Two types of bands have different calculation rules: For type (1) MQD band, only coefficients are involved in the calculation. More specially, for each MQD element in type (1) MQD band, only 4 coefficients are used. For type (2) MQD band, both coefficients and MQD elements from supportive MQD bands are involved. It means that for each MQD element in type (2) MQD band, 4 coefficients from its corresponding coefficient band and 4 MQD elements from its supportive MQD band are used.

Figure 3.6 shows MQD calculation with and without lower-level MQD band on Lv2 and Lv1, respectively. "Qmax()" is a function used to find the maximum Quantization Level (QL) of current level four-coefficient unit [11]. The QL of one coefficient can be understood as the minimum number of bits that are needed to

losslessly store the absolute value of that coefficient. However, there are differences between QL of one coefficient and actual minimum number of bits needed by the coefficient. $QL = 0$ means the minimum number of bits is 1 bit. And the 0's QL is defined as -1. "Max()" is a function used to find the maximum value among a group of numbers.

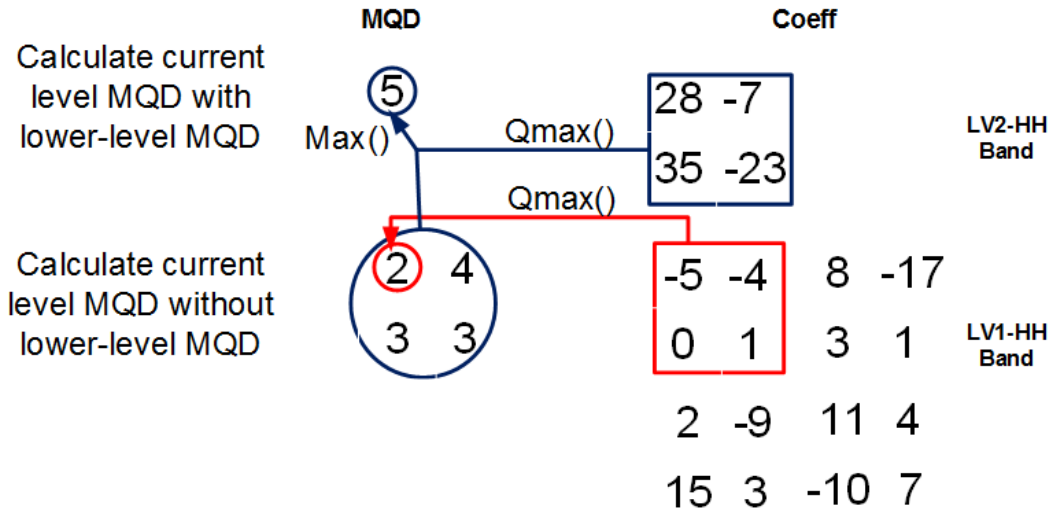


Figure 3.6 Calculation of two types of MQD bands

To calculate type (1) MQD, for example, to calculate four coefficients (-5, -4, 0, 1)'s MQD, QL of each coefficient is calculated first:

$$QL(-5) = \lfloor \mathrm{Log}_2(|-5|) \rfloor = 2,$$

$$QL(-4) = \lfloor \mathrm{Log}_2(|-4|) \rfloor = 2,$$

$$QL(0) \stackrel{\text{def}}{=} -1,$$

$$QL(1) = \lfloor \mathrm{Log}_2(|1|) \rfloor = 0.$$

The maximum QL out of those four QLs is 2, which means the MQD of those four coefficients is 2.

To calculate type (2) MQD, for example, the MQD of four coefficients (28, -7, 35, -23), QL of each coefficient is calculated first:

$MQL(28) = \lfloor \mathrm{Log}_2(|28|) \rfloor = 5,$

$MQL(-7) = \lfloor \mathrm{Log}_2(|-7|) \rfloor = 2,$

$MQL(35) = \lfloor \mathrm{Log}_2(|35|) \rfloor = 5,$

$MQL(-23) = \lfloor \mathrm{Log}_2(|-23|) \rfloor = 4,$

Note the candidate MQD, but not the final MQD, is 5.

Then four MQDs from the supportive MQD band are obtained as (2, 4, 3, 3) and the maximum value among those four MQDs and the candidate MQD is 5. The final MQD is determined as 5.

3.2.2 Implementation of Parallel MQD Calculation

In parallel MQD calculation, each MQD band is divided into multiple blocks. Currently, there is no parallelism between bands, which means one CUDA kernel processes one band and bands of MQD matrix are processed sequentially. Parallelism can be expended from within-band to between-band. To process multiple bands in one CUDA kernel, different band offsets need to be sent to the kernel and branches for different bands are introduced into the kernel, which could degrade the performance of parallel MQD calculation.

The processing sequence of bands is from the lowest level to the highest level. Within each level, it follows HL-, LH- and HH-band sequence. The last band to be processed is the LL band.

The parallel MQD calculation for one band includes four steps:

(1) The MQD band is divided into multiple blocks. For the target MQD band with size $R \times C$, the block configuration is $R \times \left\lceil \frac{C}{N} \right\rceil$, in which N is the number of MQDs generated by each block.

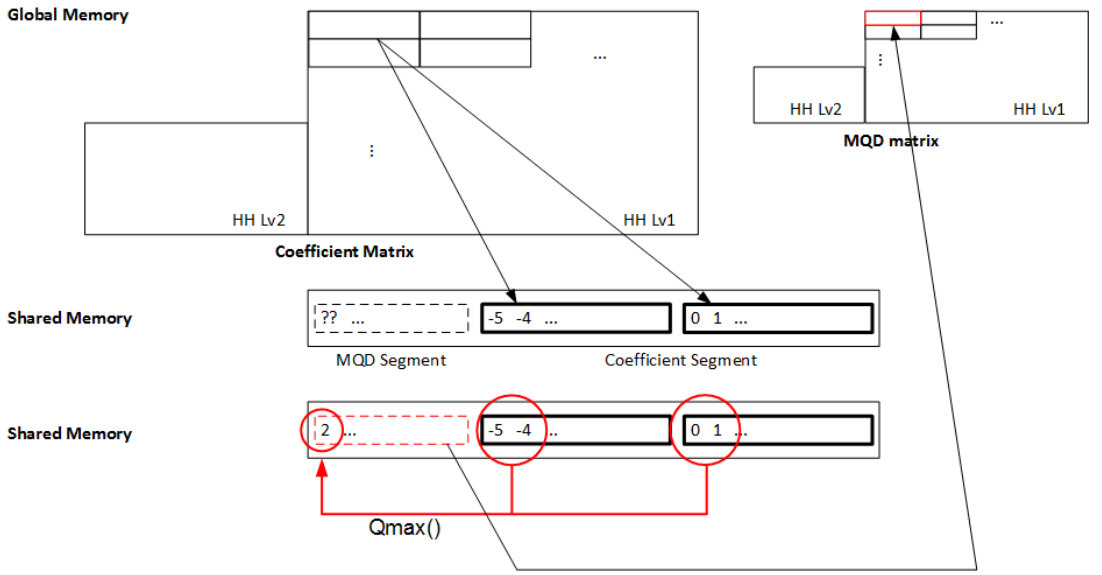
(2) For type (1) MQD band, each block takes $2 \times 2N$ coefficients from the current level coefficient band in the global memory. For type (2) MQD band, each

block takes $2 \times 2N$ coefficients from the current level coefficient band and $1 \times N$ MQDs matrix from the supportive MQD band in the global memory.

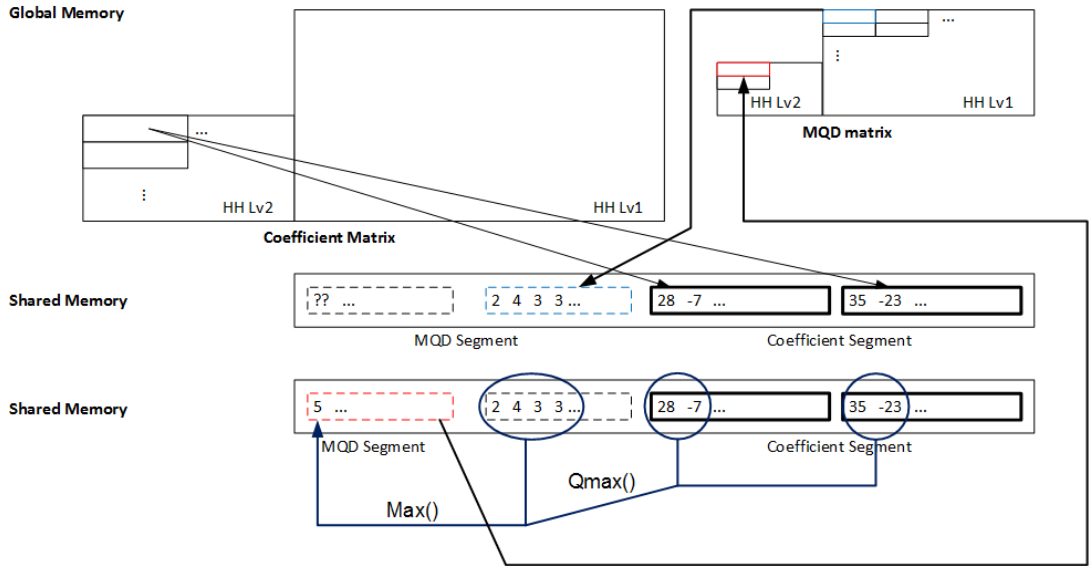
(3) Each thread generates one MQD, called the target MQD. The target MQD is generated from four coefficients, with or without four supportive MQDs. The thread obtains the maximum coefficient of these four coefficients and calculates its QL. If the band is type (1), the calculated QL is the target MQD; if the band is type (2), the calculated QL is taken as the candidate MQD and compared with four supportive MQDs. The maximum value among them is taken as the target MQD.

(4) After all $1 \times N$ MQDs of one block are generated, the block copies those MQDs back to MQD matrix in global memory, based on the offset and size of the band. The calculation of each MQD band's offset and size will be demonstrated in section 4.4.

Figure 3.7 shows an example of parallel MQD calculation. It is based on the same numerical example in Figure 3.5. It can be found that the processing band has been divided into blocks. As shown in Figure 3.7 (a), for type (1) MQD band, only coefficients are read into the shared memory and calculated MQD is copied back to corresponding MQD band. For type (2) MQD band, as shown in Figure 3.6 (b), both coefficients and MQDs from supportive MQD band are copied into the shared memory. Note the distance between coefficients from two rows, for example, the offset between coefficients -5, -4 and 0, 1 in Figure 3.6 (a), which reflects the row-wise storage in the shared memory.



(a)



(b)

Figure 3.7 Example of parallel MQD calculation

3.3 Parallel Qmax search

3.3.1 Qmax and parallel maximum value search

Qmax is defined as the maximum value within the highest level of MQD matrix, called "search matrix". The search matrix includes MQD bands corresponding to LL-, HL-, LH- and HH-band on the highest level of coefficient matrix. Qmax is used to encode the highest level MQD bands, which will be explained in section 3.4.

The temporal complexity of sequential comparison-based maximum value search is at least O(n), which means it is linearly proportional to the amount of data to be processed. "Comparison-based" means that the maximum value is obtained as the result of a series of comparisons. If any new value is larger than the current maximum value, the maximum value is updated with the new value; otherwise the maximum value remains unchanged. O(n) reflects the fact that it is impossible to tell the maximum value of a list without at least checking each element in the list. Parallel Qmax search utilizes more hardware resources, i.e. more threads on more data within a given time internal, to decrease the overall temporal complexity of search.

Maximum value search is parallelizable to a certain degree. The parallel maximum value search used in PCWT is a comparison-based reduction operation on the search matrix. "Reduction" means the number of elements being operated decrease gradually along with the proceeding of search.

As shown in Figure 3.8, there are three phases in Qmax parallel search: horizontal reduction, vertical reduction and final reduction. Their implementation details will be explained in following sections 3.3.2 to 3.3.4.

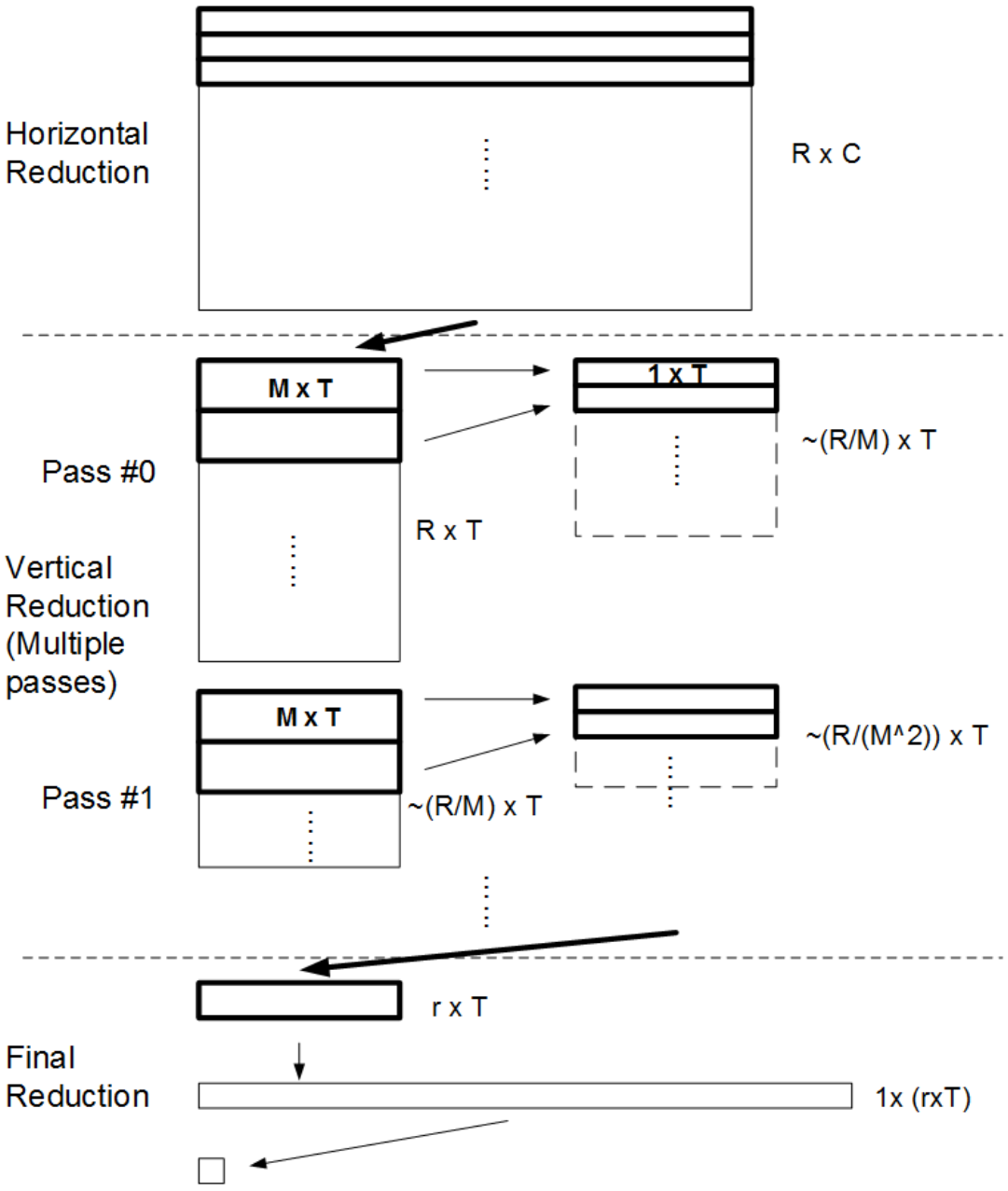


Figure 3.8 Three main phases of parallel Qmax search

3.3.2 Implementation of horizontal reduction

Horizontal reduction is a block-based parallel operation to reduce the horizontal dimension of the search matrix, i.e. the number of columns in the search matrix. Each block processes one row of search matrix, which implies that the shared

memory of one block can hold a complete row of the search matrix. For CUDA, this assumption is reasonable. Use 10000x10000 images as an example, after 5-level B-LDWT+B is performed, the width of search matrix is less than 256. The typical size of CUDA shared memory per block is 48kB which can hold more than 10,000 32-bit integers. Therefore, the assumption is valid for most of images. The horizontal reduction having multiple blocks for one row can be implemented in a multi-pass style that is similar to the following vertical reduction, which will be described in details in section 3.3.3.

The horizontal reduction has four steps:

(1) The search matrix is divided into multiple blocks. For the search matrix with size $R \times C$, the block configuration is $R \times 1$ and each block processes C elements.

(2) Each block copies one row from the original search matrix into the block's shared memory.

(3) In each block, T threads are launched and thread t performs comparison-based maximum value search on element$t$, 2t, 3t until the end of row. The element t, $2t$, $3t$, … are called the thread t's search range within the block. The maximum value within that search range is updated at element t, meaning that the current maximum value is always stored at element t.

(4) Copy the reduced row with size T, which consists of maximum values from each thread's search range, back to the original search matrix. Note that the horizontal reduction generates a horizontally compacted search matrix with size $R \times T$.

Figure 3.9 shows an example of horizontal reduction on a 5x5 search matrix. Suppose each block can hold maximum 6 elements and $T = 3$. It means that three threads are used in each block.

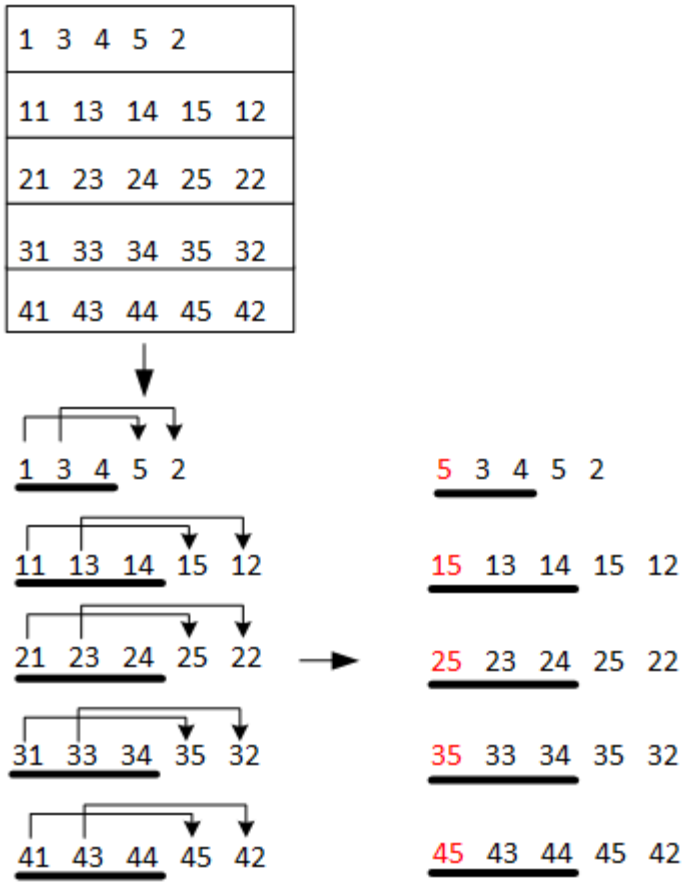


Figure 3.9 An example of horizontal reduction

Each row of the search matrix is assigned to a block and each thread compares the first element with the rest elements in its search range. Note that the third thread in each block has only one element in its search range, for example, 4 in the first row or 14 in the second row.

The maximum value within each thread's search range replaces the first element in its search range. For example, for the first thread in the first block, element 5 replaces element 1. But for the second thread in the same block, the first element 3 is the larger than element 2, so that there is no update.

The resulting 5x3 matrix (whose elements are marked by blank underscore lines) is the generated horizontally compacted matrix, which is the input for the next phase.

3.3.3 Implementation of vertical reduction

After the horizontal reduction is finished, the vertical reduction is performed to reduce the vertical dimension of the search matrix i.e. the row number of the search matrix. The goal of vertical reduction is to reduce the number of horizontally compacted rows, so that the final reduction, which will be explained in section 3.3.4, can use only one block to process all remained rows and obtain the final maximum value. The vertical reduction is a block-based multi-pass parallel operation. "block-based" means in each pass, the working matrix is divided into multiple blocks. Each block reduces multiple rows into one row to achieve in-block vertical reduction. "Multi-pass" indicates that in many cases, one pass cannot reduce the number of elements in the resulting matrix below the maximum allowable size of final reduction, so the vertical reduction needs multiple passes to achieve its goal.

In general, if one pass in the vertical reduction can approximately reduce the vertical size of the working matrix to $\frac{1}{M}$, the number of passes P to reduce the matrix row number from R to r can be calculated as: $\frac{R}{M^P} = r$, that is $P = log_M \frac{R}{r}$. It is an efficient reduction, for example, if the vertical size needs to be decreased from R = *10000* to r = *10* and each block can contain $M = 10$ rows, only three passes are required.

The vertical reduction includes multiple passes. In each pass, the vertical reduction has four steps:

(1) The current working matrix is divided into multiple blocks. For the matrix with size $R \times T$, the block configuration is $\left\lceil \frac{R}{M} \right\rceil \times 1$, in which M is the number of rows per block. Note that after each pass, the row number of the current working matrix decreases.

(2) Each block copies M rows from the working matrix into the block's shared memory. These M rows merge into a 1D search array with the size TM.

(3) In each block, T threads are launched and thread t performs comparison-based maximum value search on element$t$, $2t$, $3t$ until the end of row. Similarly to the horizontal reduction, the element t, $2t$, $3t$, … are called the thread t's search range within the block. The maximum value within that search range is updated at element t.

(4) Copy the search results (the first T elements of the 1D search array) back to the current search matrix.

After each pass, the current working matrix shrinks from $R \times T$ to $\left\lceil \frac{R}{M} \right\rceil \times T$. If $\left\lceil \frac{R}{M} \right\rceil < r$, in which r is the pre-set maximum number of rows processed in the final reduction, the vertical reduction stops; otherwise, another pass starts.

Figure 3.9 shows a follow-up example based on the result in Figure 3.8. Each block contains two horizontally compacted rows so the 5x3 matrix is divided into three blocks. In each block, a comparison-based maximum search, which is similar to the one in horizontal reduction, launches. After the 1st pass, a 3x3 vertically reduced matrix (whose elements are marked by blank underscore lines) is generated. It still contains 9 elements, which exceeds the maximum allowed number of elements per block i.e. 6. After the 2nd pass, a 2x3 vertically reduced matrix (whose elements are marked by blank underscore lines) is generated. It can fit into a single block and vertical reduction stops.

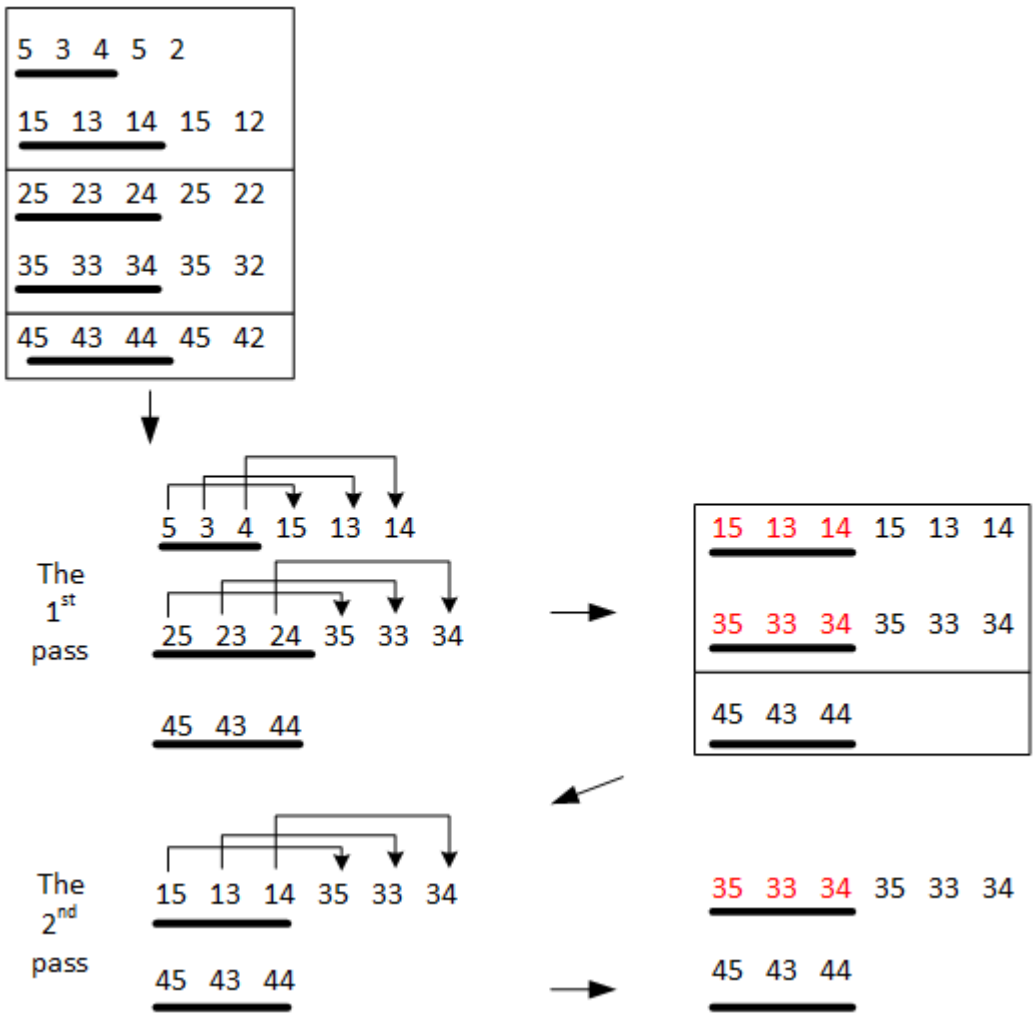


Figure 3.10 Example of vertical reduction

3.3.4 Implementation of final reduction

After the vertical reduction, the size of search matrix has been decreased so that the remained search matrix can fit into a single block's block memory. The final reduction is a single-block, multi-thread operation to find the maximum value of the whole MQD matrix.

The final reduction can be viewed as a variant of vertical reduction. It includes four steps:

(1) Only one block is used. Copy the result of vertical reduction to the block's shared memory row by row and form a concatenated row.

(2) Similarly to the vertical reduction, T threads are launched and thread t performs comparison-based maximum value search on element t, $2t$, $3t$ until the end of row. This step reduces the concatenated row to a compacted row with size T.

(3) One thread (usually the thread $t = 0$) performs comparison-based maximum value search on the compacted row from step (2) and the global maximum value is stored at the first element of the compacted row.

(4) Copy the final result back to the global matrix.

Figure 3.11 shows an example of final reduction based on the result in Figure 3.10. One block is used to process the remained 2x3 matrix. First, three threads are used to find and relocate maximum values within their search range. Second, one single thread is used to search through three remained maximum values. The final maximum value i.e. 45, is found as the result of the single-thread search.

It can found that most parts of parallel Qmax search are multi-block, multi-thread operations, which have high parallelism. The only single-block, multi-thread operation is the final reduction and the only single-thread operation is the step (3) in the final reduction. This parallelism decrease trend reflects the intrinsic of reduction operation, in which number of elements being operated are decreasing gradually. The influence of lacking of parallelism in the final phase of the maximum MQD search is neglectable. For example, for an MQD with 256x256, if 32 threads per block are used to process them, in the final stage, there are only 32 elements left for step (3), which means less than 0.1% MQDs are operated by a single thread.

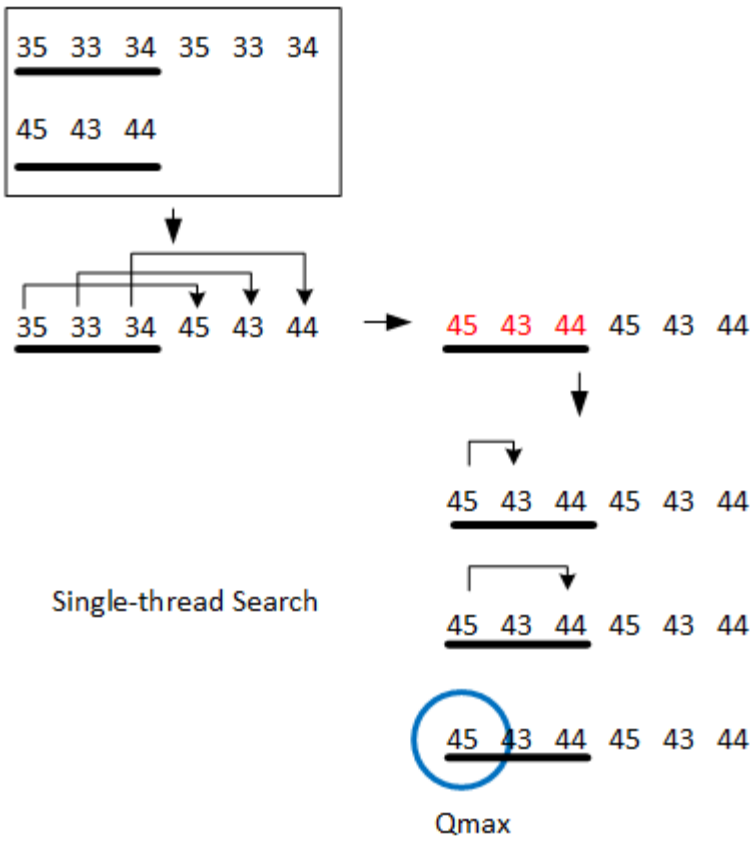


Figure 3.11 Example of final reduction

3.4 Parallel Element Encoding

3.4.1 Elements in Parallel Element Encoding

Parallel element encoding is to encode the generated B-LDWT+B coefficient to bitstream. This stage is the core stage of the whole algorithm. “Element” in parallel element encoding refers to the basic unit of encoding, which includes one MQD and four corresponding coefficients in one band at one level. In Figure 3.6, one element is (5; 28, -7, 35, -23) and another element is (2; -5, -4, 0, 1).

The procedure of encoding elements is similar to the procedure of calculation MQD. In BCWT, they are combined so that high memory efficiency can be achieved. However, those two procedures are separated in PCWT, because of the limited resources available in CUDA. As shown in previous sections, most of stages in PCWT

are implemented in the unit of block. Because of limited resources of each block (e.g. limited shared memory, limited number of threads, etc.), a common trade-off is between number of threads that can be launched per block and complexity of task performed by each thread. If element encoding is combined to MQD calculation, one obvious consequence is the high data demand from each thread, in which data required by element encoding and data required by MQD calculation merge. Data from three levels i.e. the current level, the neighboring higher level, and the neighboring lower level, are involved in element-encoding-MQD-calculation-combined operation, which would causes fewer available threads than separated element encoding or MQD calculation operation. Operations in the current separated MQD calculation and element encoding stage need data only from two levels. In PCWT, the complexity of task is decreased to allow more parallelism. Besides that, the forward encoding sequence is an important reason to separate MQD calculation stage from element encoding stage.

3.4.2 Element Encoding Sequence and Rules

Figure 3.12 shows the encoding sequence of element encoding.

The parallel element encoding stage has the following encoding sequence:

(1) The level encoding sequence is from the highest level to the lowest level.

(2) For the highest level, the band encoding sequence is top-left, top-right, bottom-left and bottom-right, i.e. LL band, HL band, LH band and HH band. For the rest of levels, the sequence is top-right, bottom-left and bottom-right, i.e. HL band, LH band and HH band.

(3) Within a band, elements are encoded row by row from top-left corner to right-bottom corner.

(4) For one element, the MQD in the element is encoded by higher-level corresponding MQD or Qmax first. If the element belongs to bands on the top level, the MQD is encoded by Qmax; otherwise the MQD (x, y) is encoded by its

corresponding higher level MQD $\left(\left\lfloor\frac{x}{2}\right\rfloor, \left\lfloor\frac{y}{2}\right\rfloor\right)$. The four coefficients in the element are encoded by the MQD in the element second. The encoding sequence is top-left, top-right, bottom-left and bottom-right.

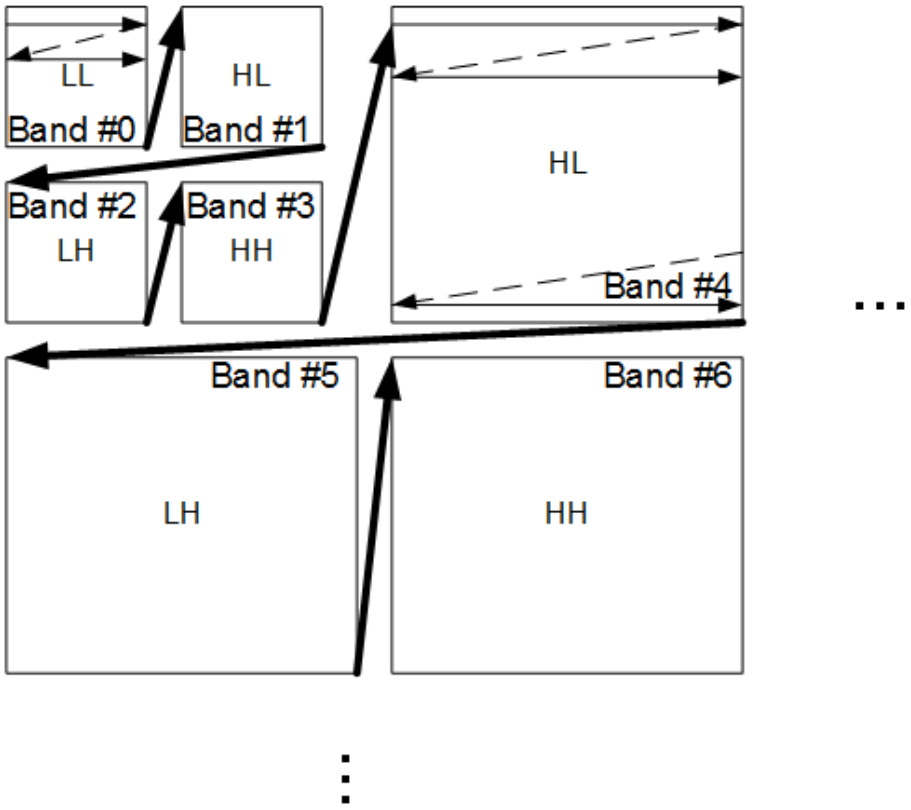


Figure 3.12 The encoding sequence of element encoding

Different from sequential CPU encoding procedure, the result of parallel element encoding includes two arrays. One array, called element encoding bitstream, contains bitstreams of elements. Another array, called element encoding length record, records length of each element's bitstream, i.e. the number of bits used by each element bitstream.

Figure 3.13 shows element encoding with Qmax. The "8 7 6 5" and "5 4 3 2 1 0" are bit index of coefficient's absolute value and "S" represents the bit index for coefficient's sign.

Element encoding with Qmax includes three steps:

(1) The element's MQD ("5") is encoded by Qmax ("8")

(2) Four coefficients corresponding to the MQD are encoded in the sequence of "top-left, top-right, bottom-left and bottom-right". i.e. "28, -7, 35, -23". The absolute value is encoded first, and the sign is encoded second.

(3) The generated bitstream ("0x 1723 A3AF") is stored in element encoding bitstream and the number of bits used to encode this element, i.e. 32, is recorded in element encoding length record.

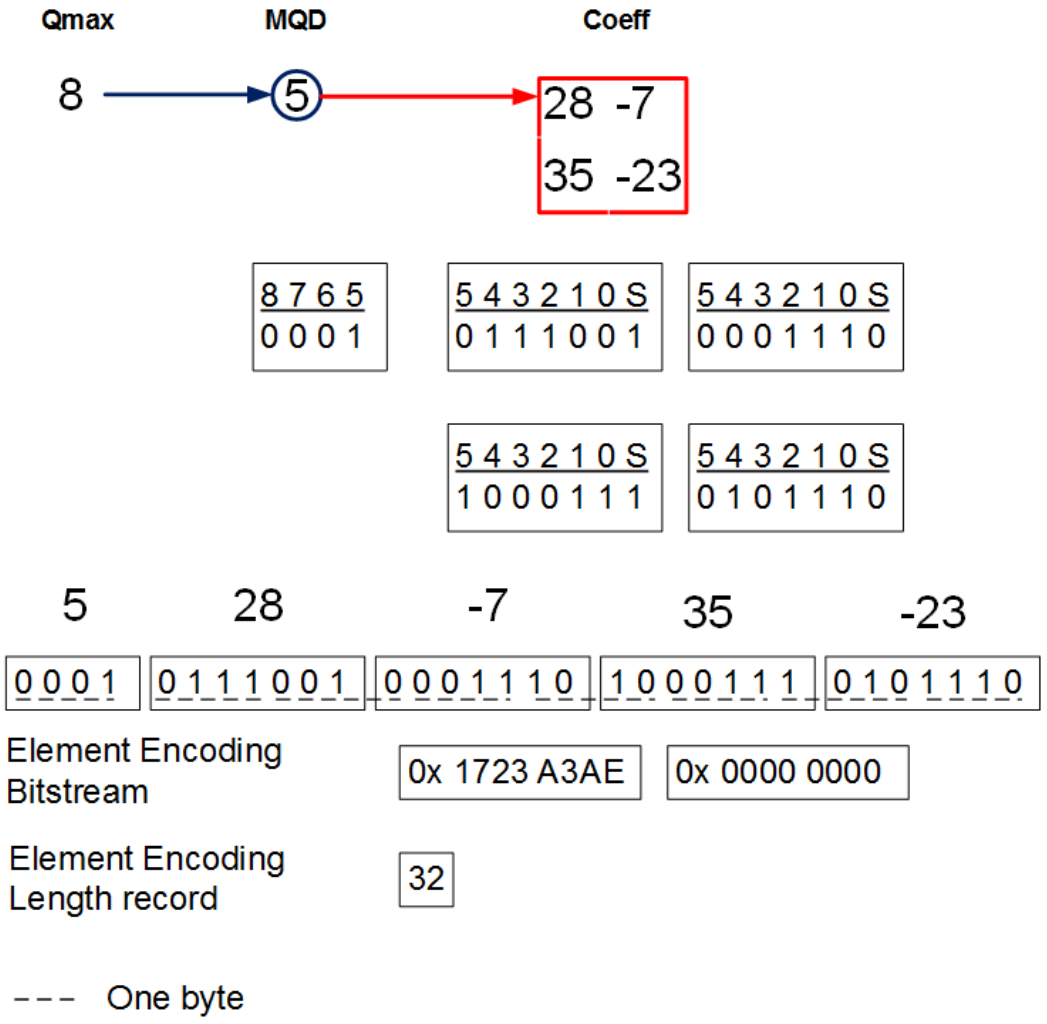


Figure 3.13 Example of element encoding with Qmax

Figure 3.14 shows an example of element encoding with the higher-level MQD. Similarly, it contains three steps:

(1) The element's MQD ("2") is encoded by higher-level MQD ("5")

(2) Four coefficients corresponding to the MQD are encoded in the sequence of "top-left, top-right, bottom-left and bottom-right" ("-5, -4, 0, 1").

(3) The generated bitstream ("0x 1A80 3") is stored in element encoding bitstream and the number of bits used to encode this element, i.e. 20, is recorded in element encoding length record.

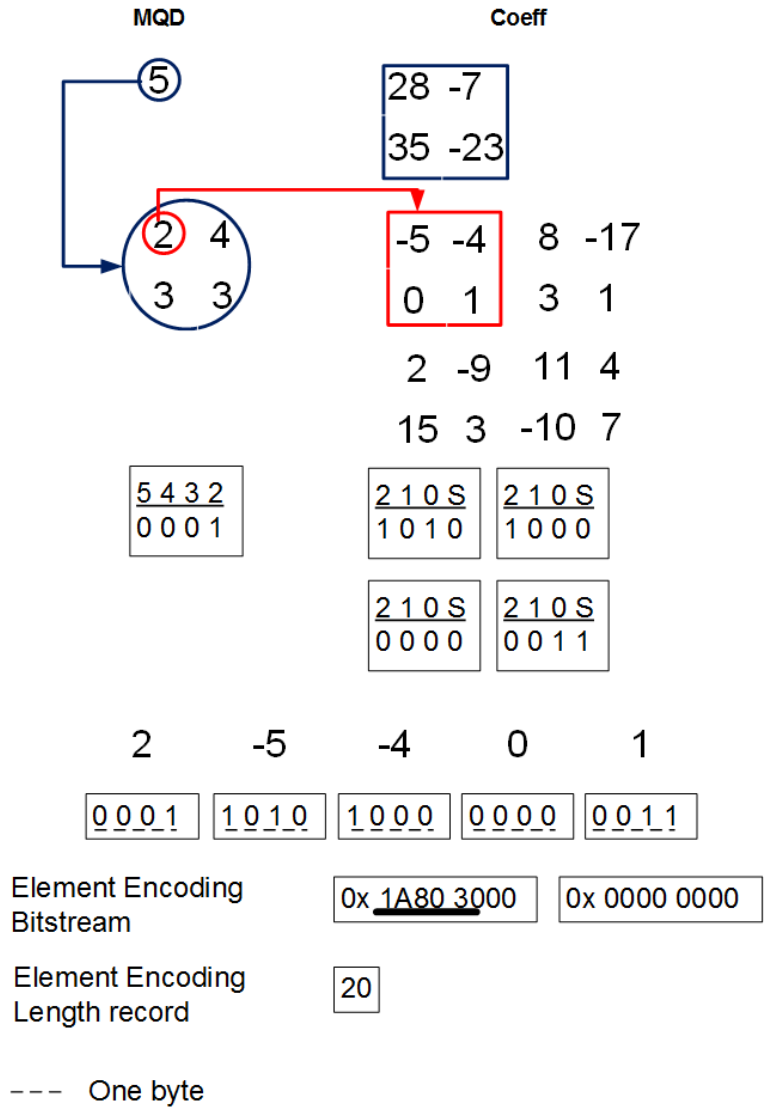


Figure 3.14 Example of element encoding with higher-level MQD

Note that the bitstream of element encoded in Figure 3.14 only needs 20 bits but still uses a 32-bit container, which causes nearly 30% wasted space. In practice, this "waste of space" problem is common in element encoding stage and a further stage, called parallel group encoding (section 3.5), is used to alleviate the problem.

3.4.3 Implementation of Parallel Element Encoding

In PCWT implementation, each GPU hardware access is in the unit of 32-bit integer. For example, one coefficient or MQD is stored as one 32-bit integer and element bitstream of one element is stored as multiple 32-bit integers. Each read/write of coefficient, MQD or bitstream is in the unit of 32-bit integer. There is no 16-bit or 8-bit access in our algorithm. The reason why the PCWT implement uses 32-bit access is because of the dynamic memory allocation restriction in CUDA [21]. In CUDA, only one dynamically allocated array is allowed in each kernel. Dynamic memory allocation allows the implementation to use the hardware resources efficiently. However, the unique dynamic array in the kernel restricts the data access format within the kernel. Since the coefficients are stored as 32-bit integers, the implementation access is determined as 32-bit. Multi-type data access is allowed in CUDA and is considered as part of future work. For example, coefficients can be accessed as 16-bit short integer, MQDs as 8-bit byte and bitstreams as 8-bit byte.

Two implementation issues in parallel element encoding are needed to be solved: one is the opposite direction of indexing of coefficients and bitstreams. As shown in Figure 3.15, for coefficient or MQD 32-bit access, the bit index is from rightmost (index = 0) to leftmost (index = 31); for bitstream 32-bit access, the bit index is from leftmost (index = 0) to rightmost (index = 31). The other issue is the fast bit extraction and insertion. Coefficient and MQD needs fast bit extraction and bits from coefficient are extracted from high index to low index, i.e. from 31 to 0. Bitstream access needs fast bit insertion. Bits that are extracted from coefficients or

MQDs are inserted into a 32-bit integer of bitstream. The bit insertion is from low index to high index (i.e. from 0 to 31).

To implement the fast bit extraction and insertion with opposite index directions of coefficient and bitstream, two groups of bit masks “Zero Index on the Left” (ZIL) group and “Zero Index on the Right” (ZIR) group are used. Each group contains 32 bit-masks, in which only one bit out of 32 bits are 1.

Figure 3.15 shows ZIR and ZIL mask groups. ZIR group is used for fast bit extraction from coefficients and MQDs and ZIL group is used for fast bit insertion to bitstream.

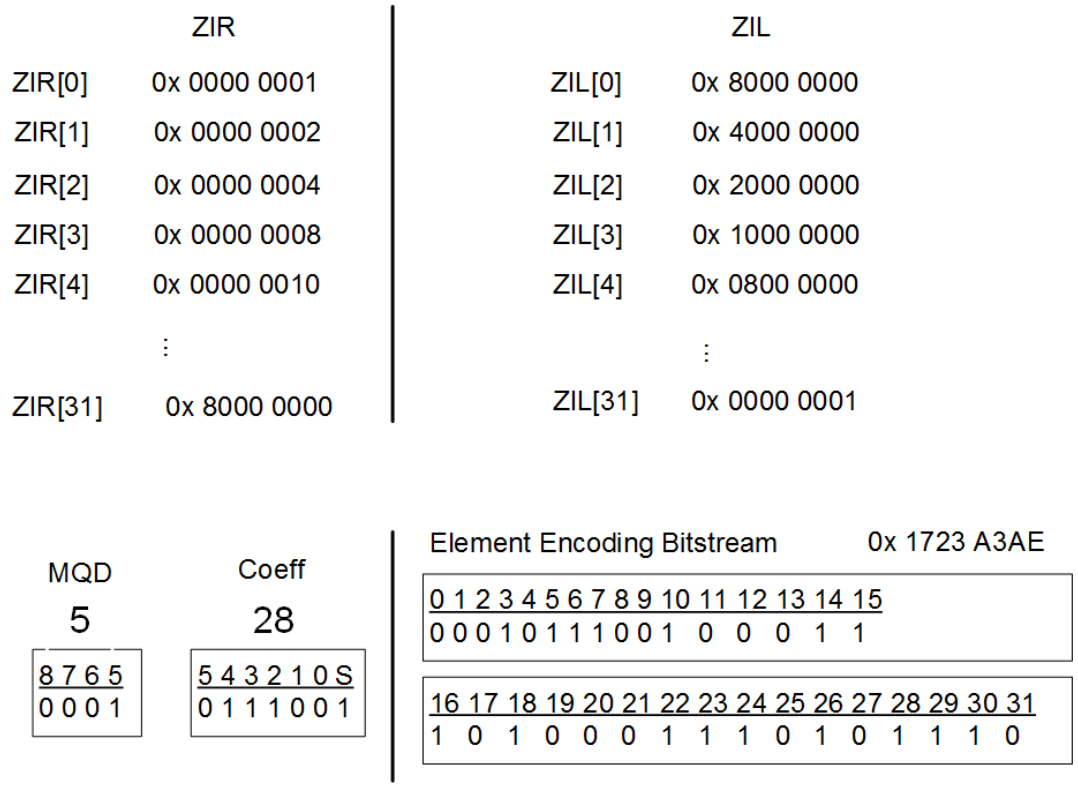


Figure 3.15 ZIR and ZIL groups and opposite index directions in MQD, coefficients and bitstream

Note that parallel element encoding collaborates with the following parallel group encoding stage. A pre-group operation exists in parallel element encoding to

reallocate the generated bitstream of each element, based on the group encoding configuration, which will be explained in section 3.5.4.

As mentioned in section 3.4.1, parallel element encoding is performed band by band. Each band is divided into multiple blocks. Each block processes multiple elements in one row. In each block, one thread processes one element. The shared memory of one block is divided into six or seven sections:

(1) ZIL segment: contains ZIL group,

(2) ZIR segment: contains ZIR group,

(3) Optional higher-level MQD segment: contains$1 \times \frac{N}{2}$ MQDs from higher level. For bands from the highest level, there is no such group.

(4) MQD segment: contains $1 \times N$ integer array,

(5) Coefficient segment: contains $2 \times N$ integer array (stored as $1 \times 2N$),

(6) Bitstream segment: contains 1D integer array. It is used to store the element bitstream.

(7) Length record segment: 1D integer array. It is used to store the bit number of each element after encoded.

For each block, the parallel element encoding includes four steps

(1) Copy data from global memory to shared memory, including ZIL group, ZIR group, optional higher-level MQD array, MQD array and coefficient array. MQD and coefficient arrays correspond to N elements.

(2) Clear the bitstream and length record group with 0

(3) Launch T threads. Each thread encodes one element.

First, the MQD is encoded. If MQD is -1, only MQD is encoded and it is encoded as 0. If MQD is not -1, the MQD is encoded as itself and it is encoded by Qmax or its corresponding MQD on the higher-level. The ZIR group is used to extract

bits from the MQD and ZIL group is used to write those bits to bitstream segment. The number of bits used to encode MQD is recorded to length record segment.

Following MQD encoding, four coefficients of the current element are encoded, under the condition that the MQD is no less than Qmin. Qmin is a threshold for future PCWT lossy compression extension and currently it is set as 0 for lossless compression. Four coefficients are encoded in the order of top-left, top-right, bottom-left and bottom-right. The absolute value of each coefficient is encoded, followed by the sign. The sign is encoded as 1 if positive, or 0 if negative. The number of bits used to encode coefficients is accumulated to length record array.

Note that when the bitstreams are written back to the shared memory, each element has its unique offset within the bitstream section. This offset is determined by "int32_per_element", which is calculated as $\left\lceil 5 * \frac{Qmax+1}{32} \right\rceil$.

(4) Copy the encoded bitstream and corresponding length record in shared memory to global memory.

The pre-group operation happens in this step. As shown in Figure 3.16, elements within the same block do not necessarily belong to the same group. Only the blue section in block #(bkx, bky) belongs to the group #g. Bitstreams and length records of elements from different groups need to be placed into the element encoding bitstream and length record array, with different group-involved offset. Because the bitstream of one element can use multiple 32-bit integers, the copy of bitstream from shared memory to global memory is different from that of length record.

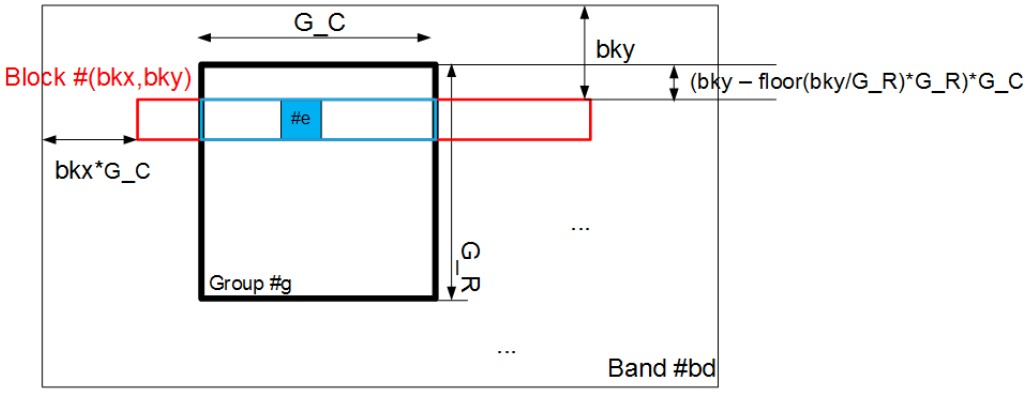


Figure 3.16 One element in band #bd, group #g is processed by thread t in block #(bkx, bky)

To copy the bitstream to the global memory, each thread handles one 32-bit integer copy. The key point of copying one 32-bit integer from one block's shared memory to the element encoding bitstream in the global memory is to find the correspondence between one 32-bit integer in the shared memory (called "source integer") and one 32-bit integer in the global memory ("destination integer").

On the shared memory side, multiple threads can process bitstream from the same element. Bitstream of one element can have multiple 32-bit integers and one thread only handles one integer.

The element that is processed by thread #t has element index within the block:

element_id_in_block = t/int32_per_element,

in which int32_per_element is number of 32-bit integers allocated for bitstream of one element.

As shown in Figure 3.16, each block (marked as the red rectangle) processes multiple complete rows from multiple groups, one complete row from each group. The row marked by blue in Figure 3.16 is encoded into the bitstream in shared memory whose ends are also marked by blue in Figure 3.17.

As shown in Figure 3.17, for thread #t, the source integer has the in-row int32 index in the shared memory is:

*int32_id_in_row = t – floor(element_id_in_block/G_C)*G_C*int32_per_element,*

in which G_C is the number of elements per row in one group.

On the bitstream side, finding the destination integer is more complicated.

The destination integer in element encoding bitstream is from the group with the global group index:

*group_id_global = floor(bky/G_R)*group_per_band_c + bkx*group_per_block + floor(element_id_in_block/G_C) + band_group_offset,*

in which *group_per_band_c* is the number of groups per row in one band, *group_per_block* is the pre-set number of group processed by one group.

The group's corresponding 32-bit integer offset is:

*group_offset_global = group_id_global*int32_per_group,*

in which *int32_per_group* is the number of 32-bit integers allocated in one group.

The destination integer also has in-group integer offset:

*int32_offset_in_group = (bky-floor(bky/G_R)*G_R)*G_C*int32_per_element + t – floor(element_id_in_block/G_C)*G_C*int32_per_element,*

in which *G_R* is the number of elements per column in one group.

Overall, the 32-bit integer processed by thread #t in block (#bkx, #bky) should be copied to the element encoding bitstream with offset *dest_int32_offset*:

dest_int32_offset = group_offset_global + int32_offset_in_group.

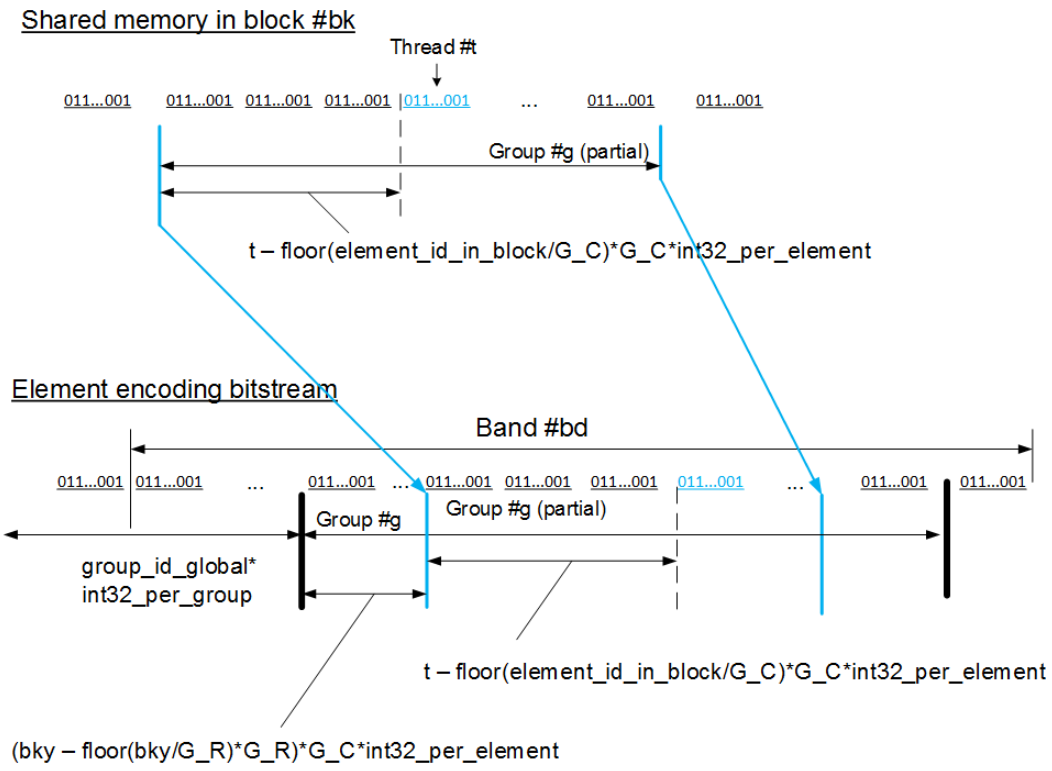


Figure 3.17 Relationship between one 32-bit integer in shared memory of block #bk and one 32-bit integer in the element encoding bitstream

To copy the length record to the global memory, one thread handles one 32-bit integer, which is the length record of one element.

On the shared memory side, as shown in Figure 3.18, for thread #t, in the shared memory, the source integer has the in-row int32 index is:

*int32_id_in_row = t – floor(t/G_C)*G_C*int32_per_element.*

On the global memory side, as shown in Figure 3.18, the destination integer is from the group with the global group offset in element encoding length record:

*group_offset_global = floor(bky/G_R)*group_per_band_c + bkx*group_per_block + floor(t/G_C) + band_group_offset.*

Overall, the 32-bit integer processed by thread #t in block (#bkx, #bky) should be copied to the element encoding length record with offset *dest_int32_offset*:

dest_int32_offset = group_offset_global + int32_id_in_row.

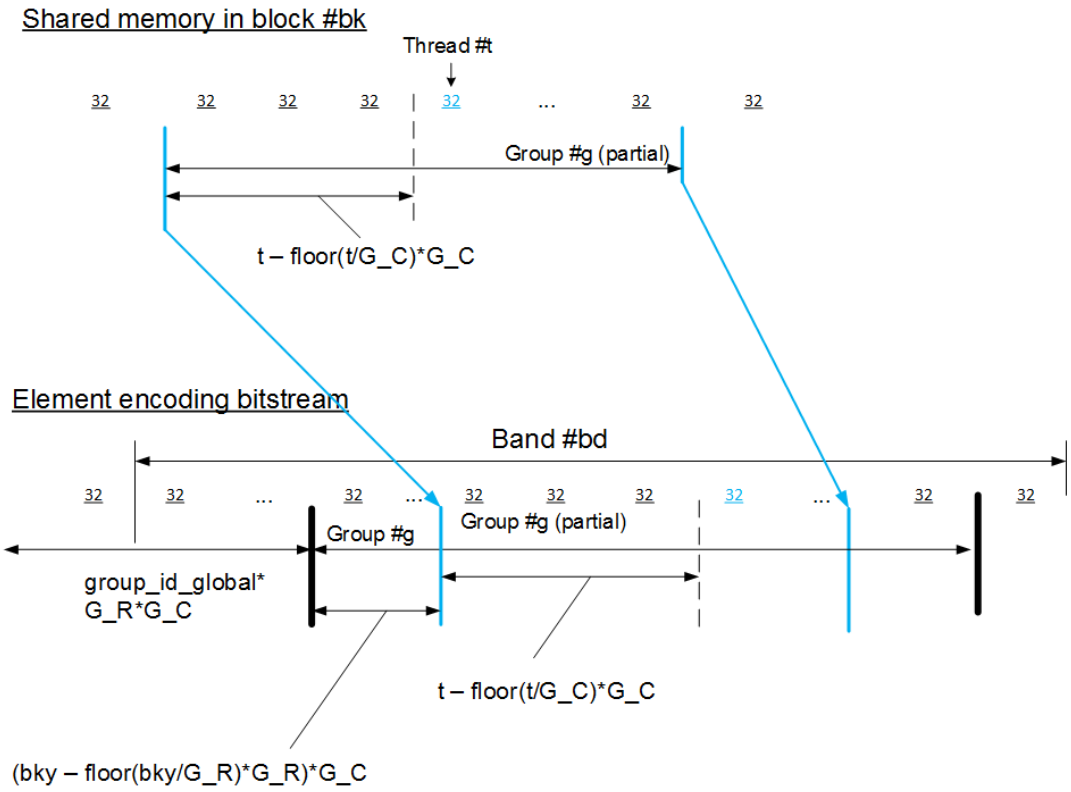


Figure 3.18 Relationship between one 32-bit integer in shared memory of block #bk and one 32-bit integer in the element encoding length record

Figure 3.19 shows an example of processing one block in element encoding and it is based on the numerical result in Figure 3.13. The whole coefficient and MQD bands have been divided into multiple blocks. Note it is the element encoding with Qmax. After step 1, ZIR and ZIL mask groups, MQDs, coefficients are copied into the block's shared memory and bitstream and length record segments have been initialized with 0. After step 2, MQDs have been encoded by Qmax, which is provided as one kernel parameters. After step 3, all four coefficients are encoded by the MQD. Note the distance between coefficients from the same element reflects the row-wise storage of coefficient matrix. The generated bitstream is recorded into bitstream sengement and the accumulated bitstream length is recorded into length record segment, as the step 2 and 3 proceeds. After step 4, the bitstream and length record segments are

copied to element encoding bitstream and element encoding length record, with the calculated offsets.

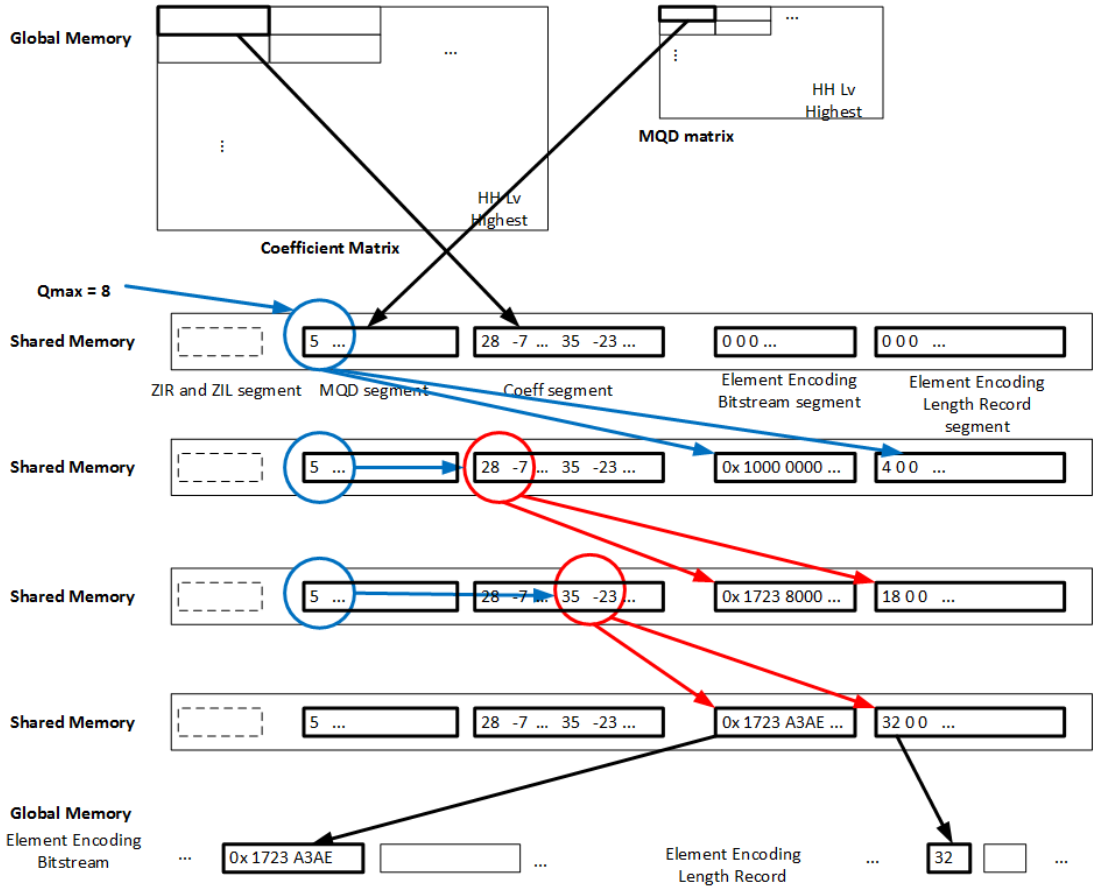


Figure 3.19 Example of processing one block in element encoding stage

3.5 Parallel Group Encoding

3.5.1 Group Encoding

Group encoding is to encode bitstreams of elements that are from the same group into a new bitstream. As shown in Figure 3.14, there are numerous unused bits in element encoding result. Group encoding is to concatenate bits used by elements in the same group together and squeeze out unused bits. Group encoding can increase the compression ratio by eliminating unused bits among bitstreams of elements in the

same group. However, group encoding decreases the independence of elements, which means that elements in the same group can be only decoded in sequence.

One issue in group encoding is the limited parallelism caused by sequentiality of bitstream of each group. If multiple threads are used to process one group, the race condition will cause the incorrect group encoding bitstream.

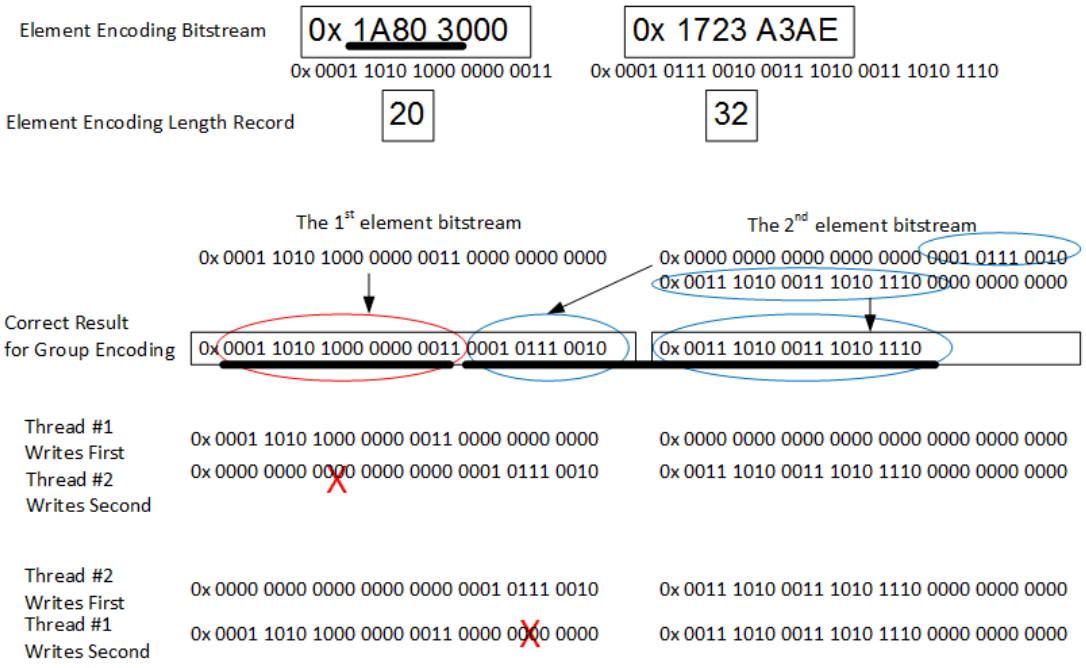


Figure 3.20 Example of correct group encoding bitstream and incorrect group encoding bitstream caused by race condition

As Figure 3.20 shows, The correct group encoding bitstream can be obtained by concatenating the second element bitstream to the end of first element bitstream. The second element bitstream involves two 32-bit integers: the first 32-bit integer contains part of the second element bitstream, with enough offset to avoid overwriting the first element bitstream stream. The second 32-bit integer contains remained bits of the second element bitstream, with necessary bit shift.

Suppose the thread #1 handles the writing of the first element bitstream to the group element bitstream and the thread #2 handles the writing of the second one. It

can be found in Figure 3.20, no matter thread #1 or thread #2 executes first, the generated group encoding bitstream is incorrect. The error happens in the first 32-bit integer, which does not contains bits from both element bitstream #1 and element bitstream #2. The first 32-bit integer contains only bits from element bitstream #1 or #2 because of the overwriting between two threads.

To avoid the overwriting problem, one group is processed by only one thread. However, even if the shared memory of one block can hold multiple completed groups, the actual thread number is limited by the number of groups processed by that block. For example, if the group size is 32x32, the shared memory of one block in computation capability 2.0 devices can hold maximum 10 groups, which only allows 10 threads to process at the same time. To increase the parallelism of group encoding, one block is used to process large number of groups (e.g. 64 or 128 groups). With the limited shared memory of one block, the assumption that one block is able to hold all bitstreams of a large number of groups is usually invalid. In another word, one block can only take a large number of "partial" groups in one pass. For each group, multiple passes are needed to complete the processing. This solution is the trade-off result between high parallelism and limited hardware resource.

Recently, the atomic operations [21] are introduced into GPGPU model to simplify the implementation of general purpose algorithms. The atomic operations are a type of thread-blocking operations and "atomic" reflects its "uninterruptable" characters. If atomic operations are used in our group encoding stage, only one thread can operate on a specific 32-bit integer exclusively. Other threads that are launched to operate the same integer have to wait until the current working thread finishes it work and release the access lock. However, atomic operation is slower than non-atomic operation because of mutual exclusive mechanism and atomic operations on the shared memory are simply to put in a queue to be sequential. For example, if two threads want to independently write several bits to the same 32-bit integer by using atomic bit-operation OR, those two OR operations are executed sequentially, with the cost of releasing and retaining the access lock on the 32-bit integer between those two

threads. In our current implementation, single-thread writing is simply executed twice to write bits to 32-bit integer, without any exclusive access cost. The atomic operation based implementation is not necessarily faster than the current implementation but it is still listed as the future work.

3.5.2 Complete and incomplete group

"Complete group" is defined as the fully occupied group and "Incomplete group" is defined as the group without full occupancy. Incomplete group usually appears near the right or bottom edge of each band. Figure 3.21 shows an example of complete and incomplete groups in one band.

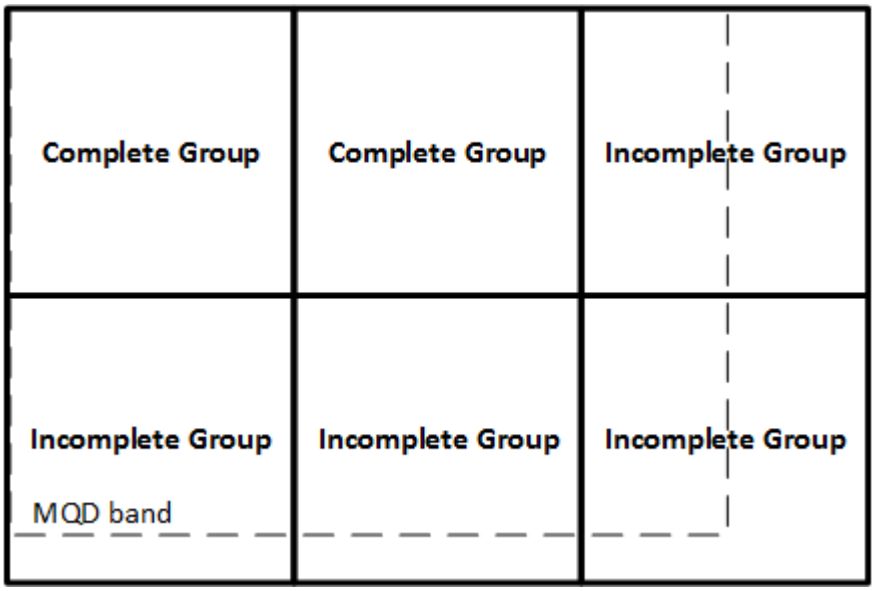


Figure 3.21 Complete and incomplete groups for MQD band

Handling incomplete group is critical for achieving high compression ratio. Simply extending the size of band by padding place-holders (such as 0) to eliminate incomplete groups introduces unused elements, which are irrelevant to the DWT and should not be encoded. This cost increases with the size of group because more place-holders are needed to form a larger complete group.

In our implementation, missing elements in incomplete group are handled by both element encoding stage and group encoding stage. In element encoding stage,

missing elements still occupy their pre-allocated spaces in element encoding bitstream but their corresponding elements in element encoding length record are set to 0. Figure 3.21 demonstrates an example of handling the incomplete group #(n+1). There are two missing elements per row in group #(n+1), which still occupy empty spaces in element encoding bitstream. However, their corresponding elements in element encoding length record are 0.

In group encoding stage, missing elements are eliminated based on their zero-use of bits. Those missing elements do not occupy any space in the group encoding bitstream so that they do not have any negative effect on the compression ratio.

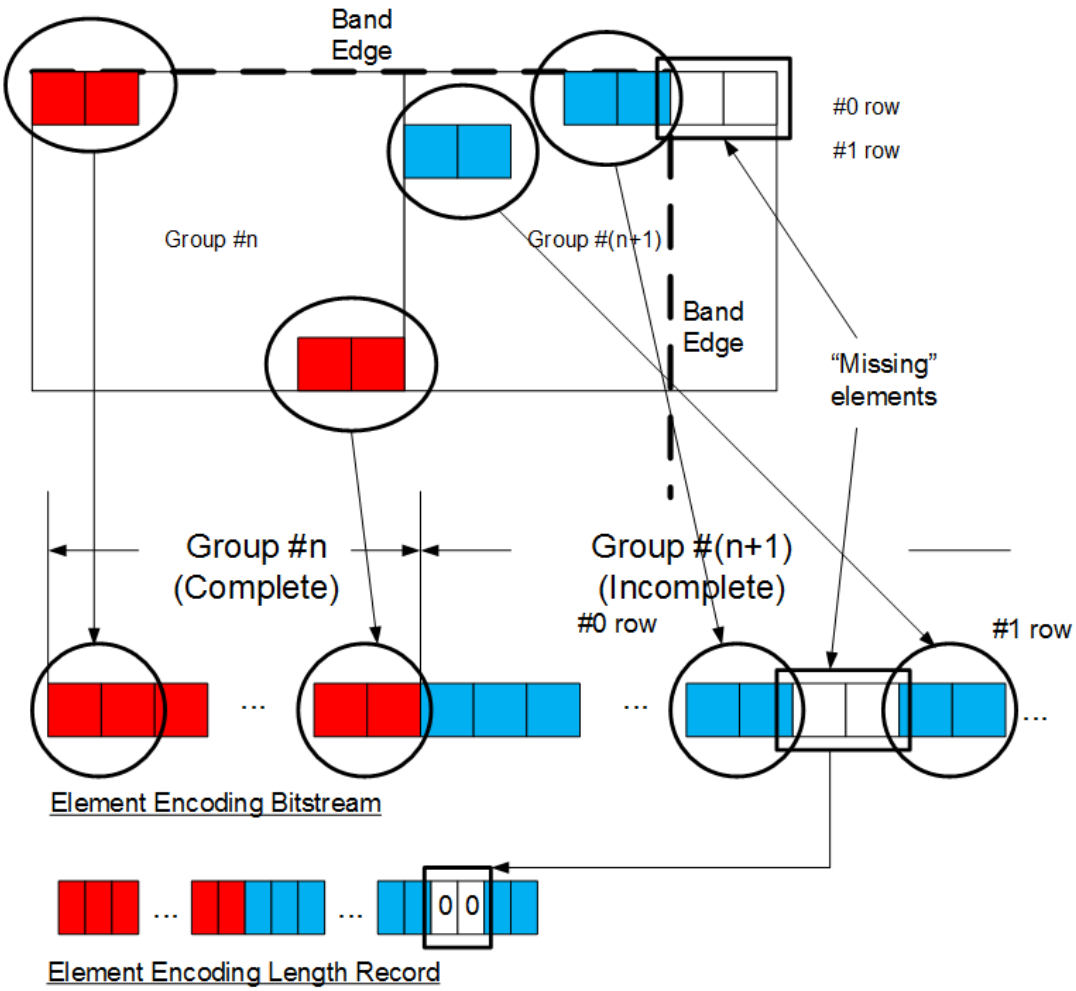


Figure 3.22 Handling missing elements in element encoding stage

3.5.3 Complete and incomplete blocks

In group encoding, each block processes multiple groups. However, the number of groups might not be the multiple of block number. For example, if there are 60x60 groups in a specific band and each block processes 128 groups, the group encoding stage needs 29 blocks, in which 28 blocks are "complete blocks" and the remaining block is a "incomplete block". Each complete block processes 128 groups and the incomplete block processes 16 groups. Based on the complete and incomplete blocks, the group encoding stage can be divided into two phases: the first phase is to run all complete blocks by one CUDA kernel and the second phase is to run the remaining incomplete block by another CUDA kernel.

The reason why the group encoding stage has two phases is to decrease the divergence between blocks within the same kernel. If both complete and incomplete blocks are included in the same kernel, in-kernel condition branches, which are used to determine how many actual groups are needed to process by a specific block, is unavoidable. Those branches introduce additional complexity and overhead to the kernel.

3.5.4 Pre-group operation in Element Encoding Stage

Pre-group operation is to reallocate element bitstreams in element encoding bitstream so that bitstreams belonging to the same group can be clustered.

Figure 3.23 shows the effect of pre-group operation on element encoding bitstream. As mentioned in element encoding stage (section 3.4.3), elements in the same CUDA block do not necessarily belong to the same group. In this example, each group has size 10x100 and one block processes two rows of elements i.e. one row from each group. Block #0 processes elements 000 to 099 in group #n and elements 000 to 099 in group #(n+1). Block #1 processes elements 100 to 199 in group #n and elements 100 to 199 in group #(n+1). Without pre-group operation, bitstreams from group #n and group #(n+1) are interlaced. If one block in group encoding stage is set to process 8 elements from each group in one pass, for pass #p, bitstreams of element 096, 097, 098, 099, 100, 101, 102, 103 are located in two separate sections in element

encoding bitstream and two global memory accesses are required when group encoding is performed. However, with pre-group operation, those bitstreams are located with consecutive addresses and only one global memory access is needed, which makes the encoding operations on pre-grouped element encoding bitstream faster. Pre-group operation in element encoding stage enhances the performance of group encoding stage.

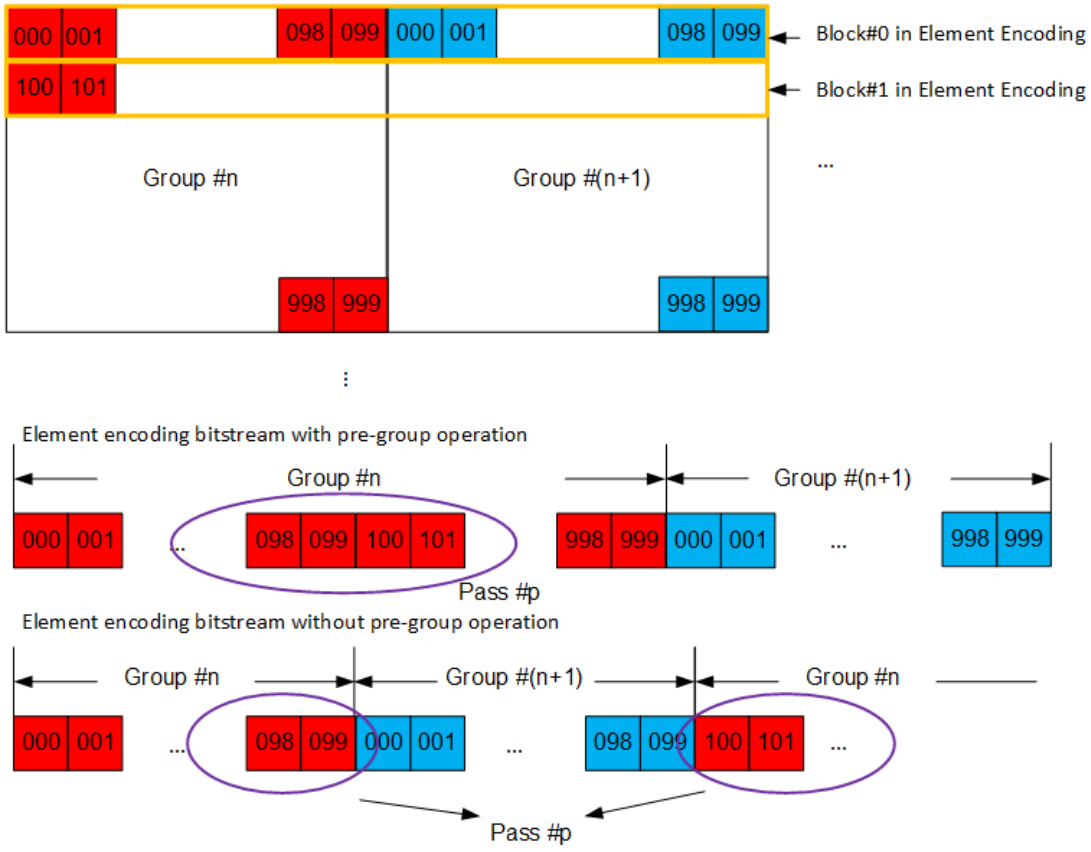


Figure 3.23 Comparison between element encoding bitstream with and without pre-group operation

3.5.5 Implementation of Parallel Group Encoding

The shared memory structure of one block in parallel group encoding, which includes six parts:

(1) ZIL segment: contains ZIL group

(2) Group offset segment: 32-bit offset of each group processed the block

(3) Element encoding bitstream segment: contains partial bitstream from element encoding bitstream

(4) Element encoding length record segment: contains partial length record from element encoding length record

(5) Group encoding bitstream segment: contains generated partial bitstream from group encoding

(6) Group encoding length record: contains generated “partial” length record from group encoding

Figure 3.23 shows the main steps in group encoding stage. The group encoding includes multiple passes.

The core of group encoding is implemented as a double-loop. The outer loop is to control the number of passes for each group. The number of passes for each group is determined by the number of elements processed in each pass. For example, if for each 32x32 group, one pass processes 4 elements, the number of passes is 256. Therefore the outer loop runs 256 times.

In outer loop, a partial element encoding bitstream and element encoding length record are copied to the shared memory, to form element bitstream segment and element length record segment, respectively. One 32-bit integer is also copied from group encoding bistream into the beginning of group bitstream segment. In some cases, the size of the partial group bitstream generated in one pass is not the multiple of 32-bit integers. In order to concatenate the bitsteam to be generated from the following pass to the already-generated bitstream correctly, one 32-bit integer at the end of processed group encoding bitstream needs to be read back for the current pass.

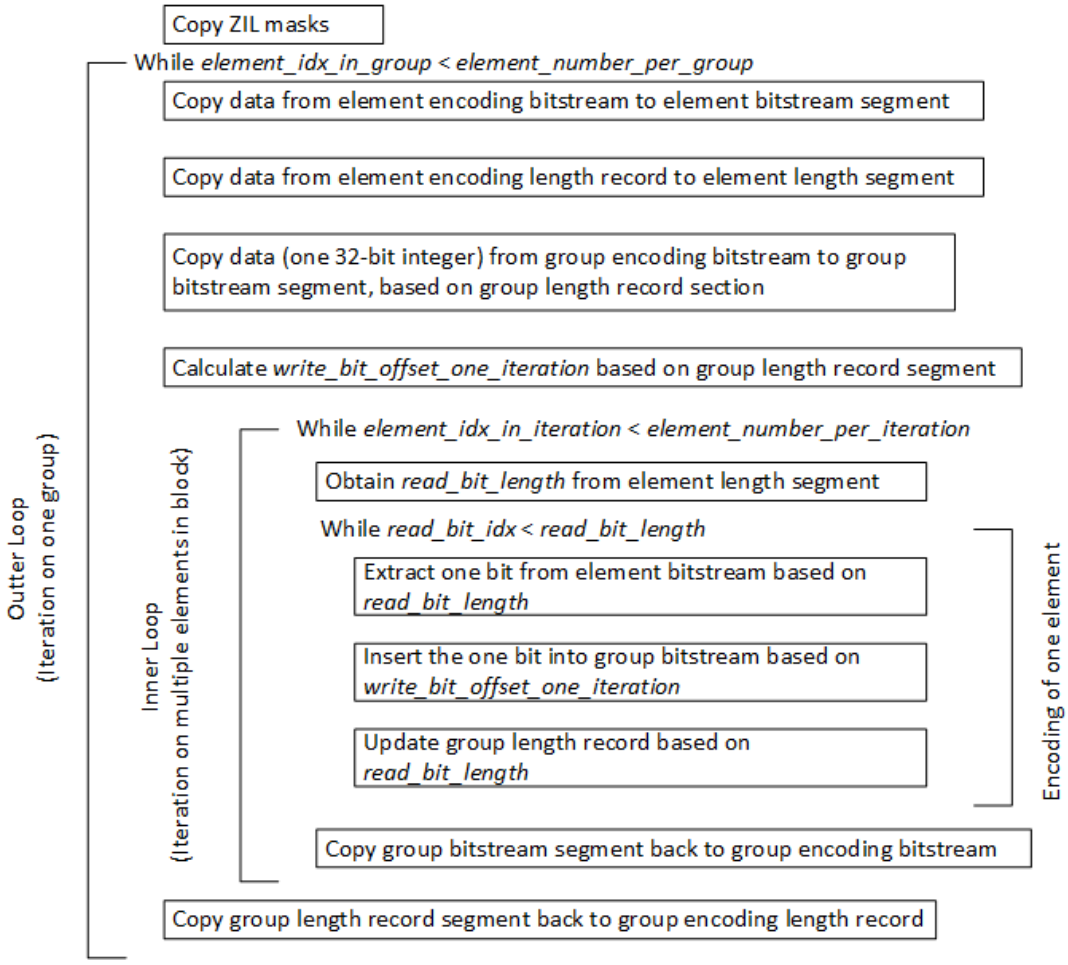


Figure 3.23 Main steps in group encoding stage

Parameter *write_bit_offset_one_iteration* is used to indicate the position of the first bit to write in the current pass. And it is calculated by:

*write_bit_offset_one_iteration = group_length – floor(group_length/32)*32*, in which group_length is the accumulated bit number of already-processed element bitstreams.

The inner loop is to merge element bitstreams fetched in the current pass into the single bitstream of the group. Each iteration of inner loop processes one element bitstream. For example, if one pass processes four elements for one group, the number of iteration of inner loop is four.

The processing of one element is similar to that in element encoding stage, in which bits from one element in element bitstream segment are extracted and written to the group bitstream segment. The difference is that in element encoding stage, both ZIL and ZIR groups are used. However, in group encoding stage, both source and destination bits all have zero index at the leftmost end so that only ZIL group is used.

After all elements assigned for the one inner loop iteration are processed, the generated group bitstream segment is copied back to group encoding bitstream in the global memory.

After all elements in one group are processed i.e. the outer-loop is completed, the generated group length record segment is copied back to group encoding length record in the global memory.

3.5 Sequential Bitstream Output

3.5.1 Sequential Output

The previous parallel group encoding stage provides two data structures: group encoding bitstream that includes group bitstreams and group encoding length record that contains lengths of group bitstreams. The sequential output stage is to combine those group bitstreams with group bitstream lengths, into a new bitstream called the final bitstream and output the final bitstream to a file on storage devices.

Each group bitstream and its corresponding group bitstream length are combined first. The group bitstream length is stored in front of group bitstream, which allows the fast location of a specific group. Then those combined group bitstreams are merged on the byte (8-bit) level, not on bit level, to form the final bitstream. Nevertheless, mergence on the bit level can provide further compression, on the top of element encoding and group encoding stages. However, bit-level mergence complicates the extraction of bitstreams from different groups in the decoding procedure. Bytes need to be split and shifted before bitstreams are ready for decoding. Mergence on the byte level can avoid those bit operations. Mergence on byte level can also avoid the bit wasting that is introduced by combination on 32-bit level. Group

bitstreams and their length are stored in the unit of 32-bit integer, as the result of element encoding and group encoding stages. Directly store all 32-bit integers will unavoidably introduce unused bits at the end of each combined group bitstream because no matter 1 bit or 31 bits are actually used, 32 bits are occupied. Mergence on byte level is the tradeoff between compression ratio and decoding complexity.

3.5.2 Implementation of Sequential Output

Besides the final bitstream, there are at least six parameters about the compressed image and compression procedure, which need to be written to the final output file:

(1) Qmax: is required to decode bands on the highest level.

(2) Qmin: currently is set as 0.

(3) dwt_level: the actual DWT levels performed by PCWT, which is required for inverse B-LDWT+B in decompression procedure.

(4) element_per_group_R: the number of rows in one group

(5) element_per_group_C: the number of columns in one group

(6) channel_number: currently not used.

Six parameters are written to the file first, followed by the final bitstream. The wiring of the final bitstream to file is implemented as a loop. In each loop, the group bitstream and its length of one group are output in two steps:

(1) The first 32 bits are the group bitstream length, which is obtained from length_record_group

(2) The number of effective bytes of the group bitstream is calculated by: (length_record_group*8 + 7)/8 and those bytes from each group bitstream are written to the output file.

CHAPTER IV

PCWT DECOMPRESSION PROCEDURE

Decompression procedure is to read files that contain PCWT bitstream and restore the compressed images. It includes three main stages: sequential bitstream input, sequential decoding, and parallel inverse B-LDWT+B.

4.1 Sequential Bitstream Input

4.1.1 Sequential Input

Sequential bitstream input is to read compressed image file in bytes and reallocate those bytes to form the proper data structures for the sequential decoding stage, which will be explained in details in the following section.

The structure of compressed image file is defined in section 3.5.2 and it contains two major parts:

(1) Metadata of the image and compression procedure

(2) Group bitstreams.

The part (1) is critical for following stages in the decompression procedure. The part (2) has the similar memory structure to the final bitstream in Section 3.5, whose memory spaces need to be pre-allocated. The pre-allocation of final bitstream requires two parameters:

(a) int32_per_group: the number of 32-bit integers per group and

(b) group_number_total: the total number of groups processed in the decompression procedure.

Note that both parameters can be derived from the metadata from part (1).

4.1.2 Implementation of Sequential Bitstream Input

The sequential bitstream input includes three steps:

(1) Read the metadata of the image and compression procedure: these metadata are the ones written in Section 3.5.2, including: Qmax, Qmin, dwt_level,

element_per_group_R, element_per_group_C and channel_number. Each parameter is a 32-bit integer.

(2) Calculate the codec properties, which will be explained in section 4.4 to ensure the decompression procedure is strictly inversed to the compression procedure. Based on the obtained codec properties, necessary memory allocations are performed, including: the sequence decoding bitstream, the sequence decoding length record, MQD matrix for sequential decoding stage and two GPU 2D arrays for inverse B-LDWT+B transform.

(3) Read group bitstreams into the group sequence decoding bitstream and their corresponding bit number into sequence decoding length record.

4.2 Sequential Decoding

4.2.1 Sequential decoding and its comparison with encoding stages

Sequential decoding is to decode the sequence decoding bitstreams, with the help of sequence decoding length record, to re-generate the B-LDWT+B coefficient matrix. The generated coefficient matrix will be used as the input matrix for inverse 2D B-LDWT+B (section 4.3) to restore the compressed image.

Different from the compression procedure that contains two encoding stages i.e. element encoding and group encoding stages, the decompression procedure only contains one decoding stage i.e. the sequential decoding stage. In sequential decoding stage, sequential decoding bitstream is directly decoded to 2D coefficient matrix. There is no intermediate bitstream, such as the element encoding bitstream in the compression procedure, in decompression procedure.

MQD matrix is still involved in the sequential decoding as the auxiliary matrix, which means the sequential decoding stage needs to decode two matrices, coefficient matrix and MQD matrix from the sequential decoding bitstream.

The decoding stage is chosen as sequential because strong dependencies within the bitstream impede performance of any parallel decoding stage: (1) Level-dependency, i.e. the decoding of lower-level bands depends on the completion of decoding of higher-level bands, limits the parallelism among different levels; (2) Element dependency, i.e. elements within the same group can only be decoded sequentially, limits the parallelism within each band.

Similarly to the element encoding stage, the basic unit in the sequential decoding stage is the element, which includes one MQD and four coefficients. The sequential decoding stage has the same level/band/group/element processing sequence as that of compression procedure.

(1) The decoding is from the top level to the lowest level

(2) Within each level, bands are decoded in sequence of (LL), HL, LH and HH. The sequential decoding stage has two types of decoding: decoding element with Qmax and decoding element with higher-level MQD, depending on the band being decoded.

(3) Within each band, groups are decoded sequentially from the group on top-left corner to the group on bottom-right corner.

(4) Within each group, elements are decoded sequentially from the element on top-left corner to element on the bottom-right corner.

Figure 4.1 shows an example of decoding one element with its corresponding higher-level MQD. The bitstream is the same one used in Figure 3.20. The decoding procedure contains two parts:

(1) The element's MQD, i.e. 2, is decoded from one group bitstream by higher-level MQD, i.e. 5.

(2) Four coefficients are decoded with the decoded MQD from the same element bitstream in the sequence of top-left, top-right, bottom-left and bottom-right

coefficients. For each coefficient, the absolute value is encoded first, and the sign is encoded second

The decoded MQD and coefficients can be verified by the numerical example in Figure 3.14.

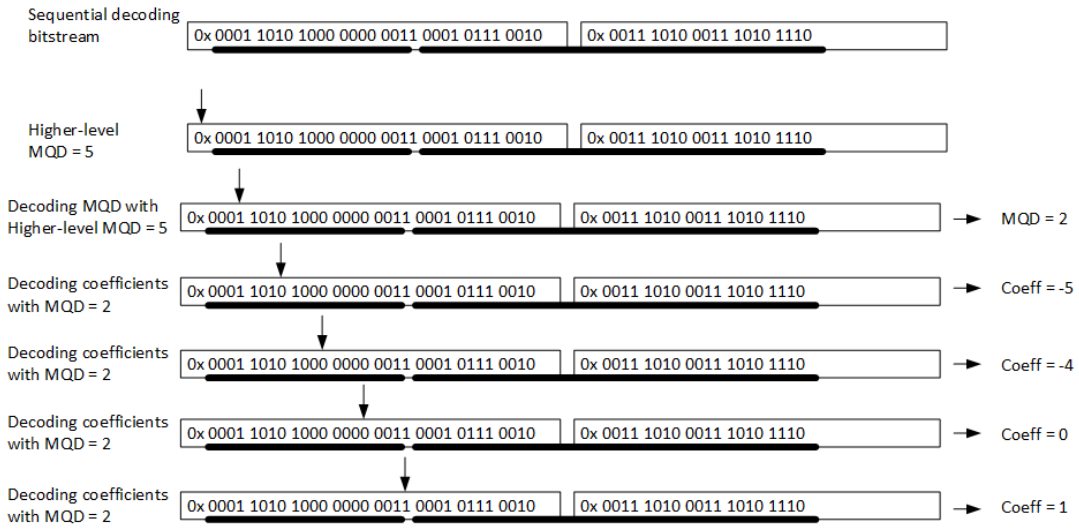


Figure 4.1 Example of decoding one element with the corresponding higher-level MQD.

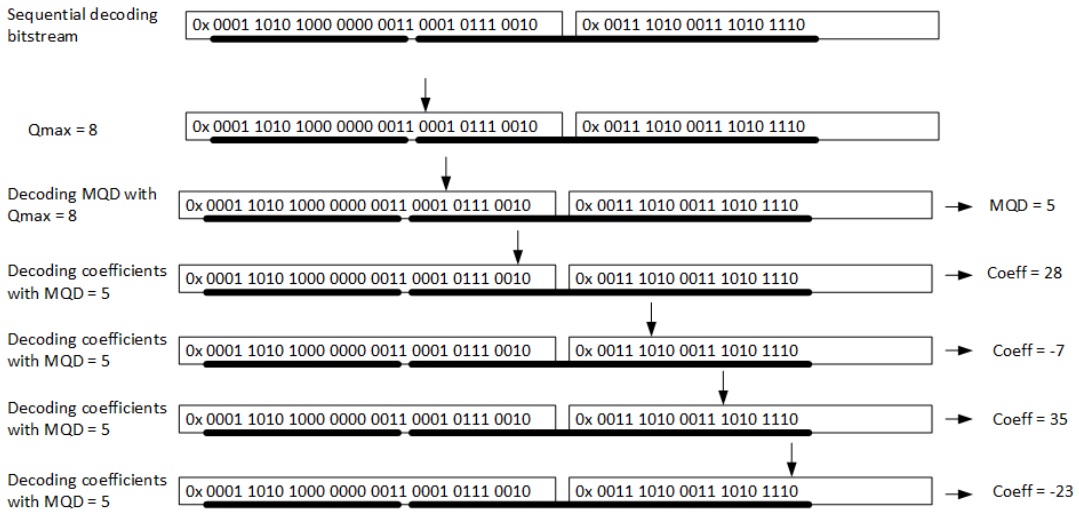


Figure 4.2 Example of decoding one element with QMAX.

Figure 4.2 shows an example of decoding one element with Qmax. Similarly, it contains two parts:

(1) The element's MQD i.e. 5, is decoded from one group bitstream by Qmax, i.e. 8.

(2) Four coefficients are decoded from the same bitstream in the sequence of top-left, top-right, bottom-left and bottom-right elements, i.e. 28, -7, 35, -28.

The decoded MQD and coefficients can be verified by the numerical example in Figure 3.13.

4.2.2 Implementation of sequential decoding

In the sequential decoding stage, the decoding has four levels (section 4.2.1). Figure 4.3 shows the implementation of sequential decoding. Note that level decoding and band decoding are merged because each band has its unique level and more importantly, each band has its unique offsets and sizes, which are sufficient for placing decoded MQDs and coefficients in the correct positions.

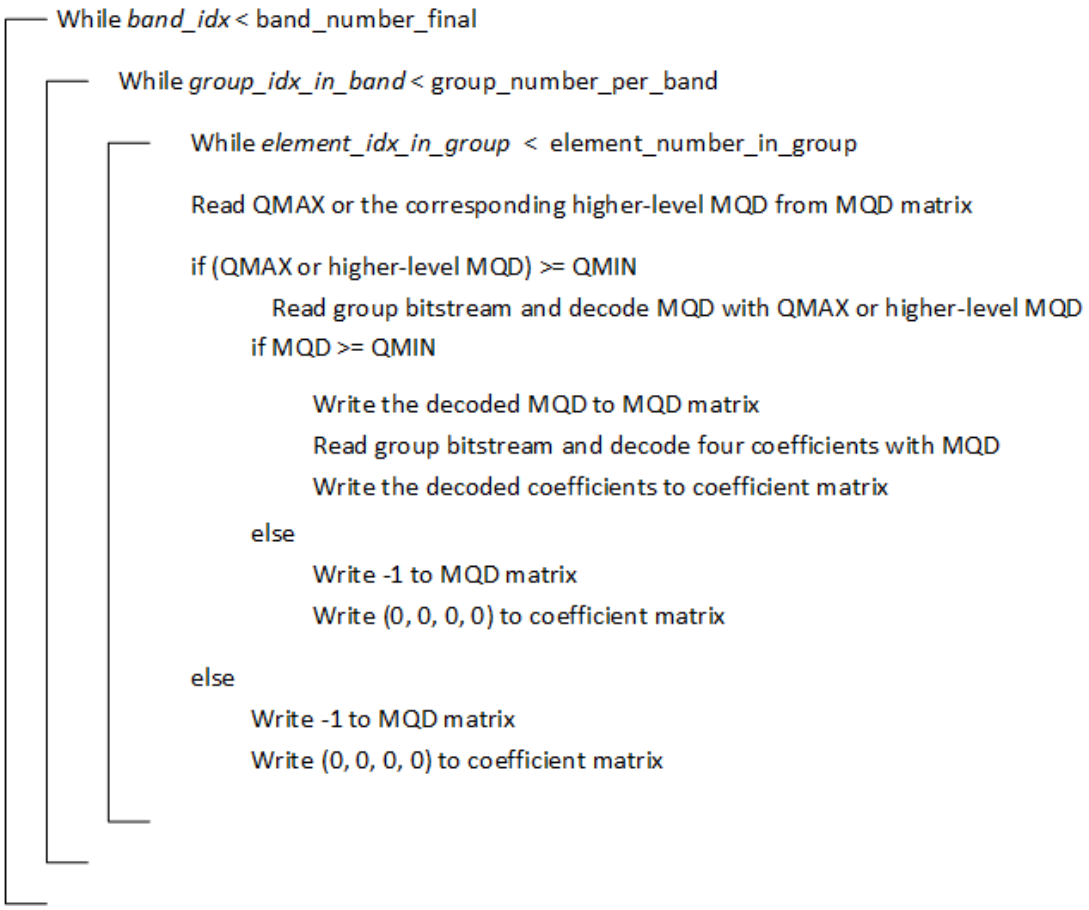


Figure 4.3 The implementation of sequential decoding

Each band has its unique offset and size, which can be obtained from corresponding codec properties (section 4.4.3) such as *coeff_band_offset_list*, *coeff_band_size_list*, *mqd_band_offset_list* and *mqd_band_size_list*.

Each group within one band has its unique offset and size, which can be also obtained from corresponding codec properties (Section 4.4.3), such as *mqd_group_number_per_band_list* and *group_size*.

Each element within one group has its unique position, which cannot be obtained from codec properties directly. To decode one element in group decoding, its in-group element index can be obtained by accumulating the index from 0 when group encoding starts.

After the element index in the group is obtained, based on the valid group column number *valid_group_size_c*, the element position within the group can be determined by:

element_idx_r_in_group = element_idx_in_group/valid_group_size_c

element_idx_c_in_group = element_idx_in_group - element_idx_r_in_group valid_group_size_c*, in which *element_idx_r_in_group* and *element_idx_c_in_group* are the element row index and element column index within the group, respectively.

The *valid_group_size_c* is used instead of pre-defined group size is because of the group encoding stage in the compression procedure (section 3.5). For complete group, *valid_group_size_c* equals the pre-defined group column number. However, for an incomplete group, *valid_group_size_c* equals the actual column number in that group, because the missing elements are not included in sequential decoding bitstream.

After *element_idx_r_in_group* and *element_idx_c_in_group* are obtained, the element's MQD position is *element_idx_r_in_group* and *element_idx_c_in_group* with necessary band and group offsets. Coefficients' positions are (*element_idx_r_in_group**2, *element_idx_c_in_group**2),

(*element_idx_r_in_group**2 + 1, *element_idx_c_in_group**2),

(*element_idx_r_in_group**2, *element_idx_c_in_group**2 + 1),

and (*element_idx_r_in_group**2 + 1, *element_idx_c_in_group**2 + 1), with necessary band and group offsets.

4.3 Parallel Inverse B-LDWT+B

Parallel inverse LDWT is to restore the compressed image from decoded LDWT coefficients. As the inverse operation of parallel forward B-LDWT+B, inverse B-LDWT+B includes two phases: parallel inverse horizontal 2D B-LDWT+B and parallel inverse vertical 2D B-LDWT+B, with embedded Clustering-to-Interlacing (C2I) operation are performed.

C2I operation is the inverse operation of the I2C operation in forward transforms. The decoded coefficient matrix is in the cluster form, which is the result of the I2C operation in the compression procedure. To perform inverse LDWT properly, the clustered L- and H-components needs to be interlaced and C2I operation is to generated interlaced row or column before horizontal or vertical inverse transform is performed.

Parallel inverse 2D B-LDWT+B is also block-based, thus it has similar configuration to parallel forward 2D B-LDWT+B. Each block of parallel inverse horizontal 2D B-LDWT+B processes a partial row and each block of parallel inverse vertical 2D B-LDWT+B process multiple partial columns. However, there are several differences between parallel forward and inverse transforms:

(1) In inverse transform, vertical inverse transform is performed before horizontal inverse transform, which is different from forward transform, in which horizontal forward transform is performed before the vertical one.

(2) In inverse transform, C2I operation is performed before any L- or H-component calculation is performed. In forward transform, I2C operation is performed after both L- and H-component calculation is finished.

(3) In inverse transform, the "over-computed" element is D[2N], which is calculated as a L-component and is used to calculated the H-component $D[2N-1]$. In forward transform, the "over-computed" element is $D[-1]$, which is calculated as a H-component and is used to calculate the L-component $D[0]$.

Figure 4.4 shows an example of 1D inverse B-LDWT+B. It is based on the numerical example in Figure 3.1.

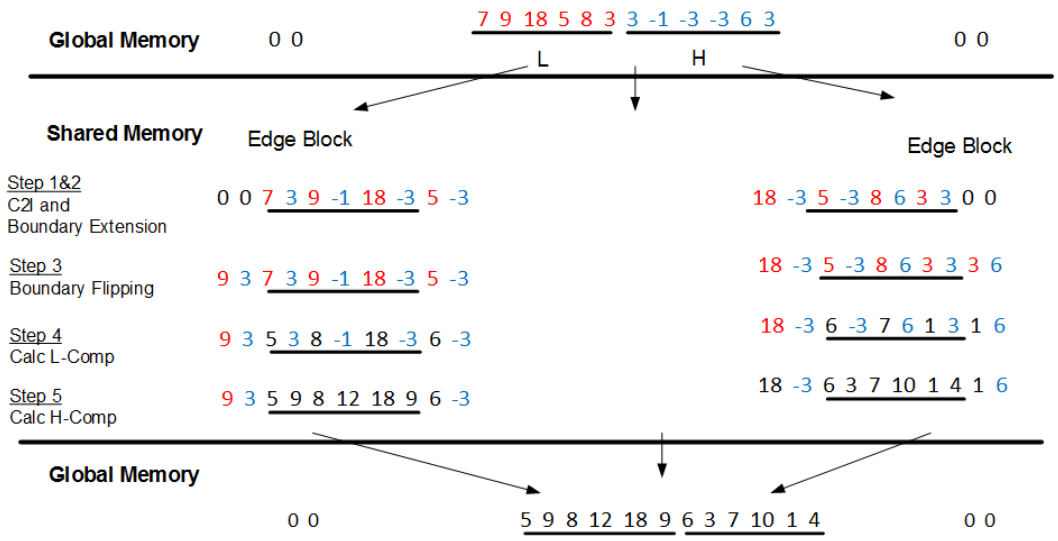


Figure 4.4 Example of 1D inverse B-LDWT+B

4.3.1 Implementation of Vertical 2D B-LDWT+B

Vertical inverse 2D B-LDWT+B is to perform 1D inverse B-LDWT+B on columns of the decomposition result on any level. Vertical inverse 2D B-LDWT+B has the similar block configuration to the vertical forward one.

The complete Vertical 2D B-LDWT+B includes six steps:

(1) The input matrix, i.e. the decomposition result on any level which includes four bands LL, HL, LH and HH, is divided into blocks. One block contains multiple partial columns. For input matrix size $R \times C$, the block configuration is $\left\lceil \frac{R}{r} \right\rceil \times \left\lceil \frac{C}{c} \right\rceil$, in which r and c are number of elements processed by one block in horizontal and vertical direction, respectively.

(2) Blocks are read from GPU global memory (global 2D array #1) into the blocks' corresponding shared memory. Embedded C2I operation is implemented by two global memory reads. One is for L-components and the other is for H-components.

Similar to the 2D forward vertical B-LDWT+B, for each block, 4 extra rows of elements outside the block are also read for boundary extension.

(3) Optional boundary flipping is performed on one end of the shared memory, if the block is boundary block.

(4) Perform L-step of 1D B-LDWT+B.

(5) Perform H-step of 1D B-LDWT+B.

(6) Copy the generated result back to global memory array #2.

4.3.2 Implementation of Horizontal Inverse 2D B-LDWT+B

Horizontal inverse 2D B-LDWT+B is to perform 1D inverse B-LDWT+B on each row of the result of vertical inverse 2D B-LDWT+B. It includes six steps:

(1) The input matrix, i.e. the result from inverse vertical 2D B-LDWT+B, is divided into blocks. One block contains a part of row. For input matrix size $R \times C$, the block configuration is $R \times \left\lceil \frac{C}{b} \right\rceil$, in which b is the number of element processed by one block.

(2) Blocks are read from GPU global memory (global 2D array #2) into the blocks' corresponding shared memory. Embedded C2I operation also is performed.

(3) Optional boundary flipping is performed on one end of the shared memory, if the block is boundary block.

(4) Perform L-step of 1D inverse B-LDWT+B.

(5) Perform H-step of 1D inverse B-LDWT+B.

(6) Copy the generated result back to global memory array #1.

4.4 Codec Properties Calculation

4.4.1 Codec Properies

Codec properties are defined as all the necessary parameters used in PCWT compression and decompression procedure.

Codec properties can be classified into four categories:

(1) Coefficient matrix related properties:

(a) Coefficient level offset

(b) Coefficient level effective size

(c) Coefficient level extended size

(d) Coefficient band offset

(e) Coefficient band effective size

(f) Coefficient band extended size

(2) MQD matrix related properties:

(a) MQD band offset

(b) MQD band effective size

(c) MQD band extended size

(3) Element encoding related properties:

(a) int32_number_per_element: number of 32-bit integers per element

(3) Group encoding related properties:

(a) Group number per band

(b) Total group number

(c) int32_number_per_group: number of 32-bit integers per group

(b) Group bitstream offsets

(4) DWT related properties:

(a) The final number of decomposition levels in forward B-LDWT+B

(a) Total band number

4.4.2 Codec Properties Calculation in Compression Procedure

Note that those codec properties are obtained from different stages because there are dependencies between some of properties.

In compression procedure, following parameters are required from users:

(1) Input image size (image_size_R, image row number and image_size_C, image column number)

(2) User preferred decomposition level (user_dwt_level)

(3) Group size (group_size_R, number of rows per group and group_size_C, number of columns per group)

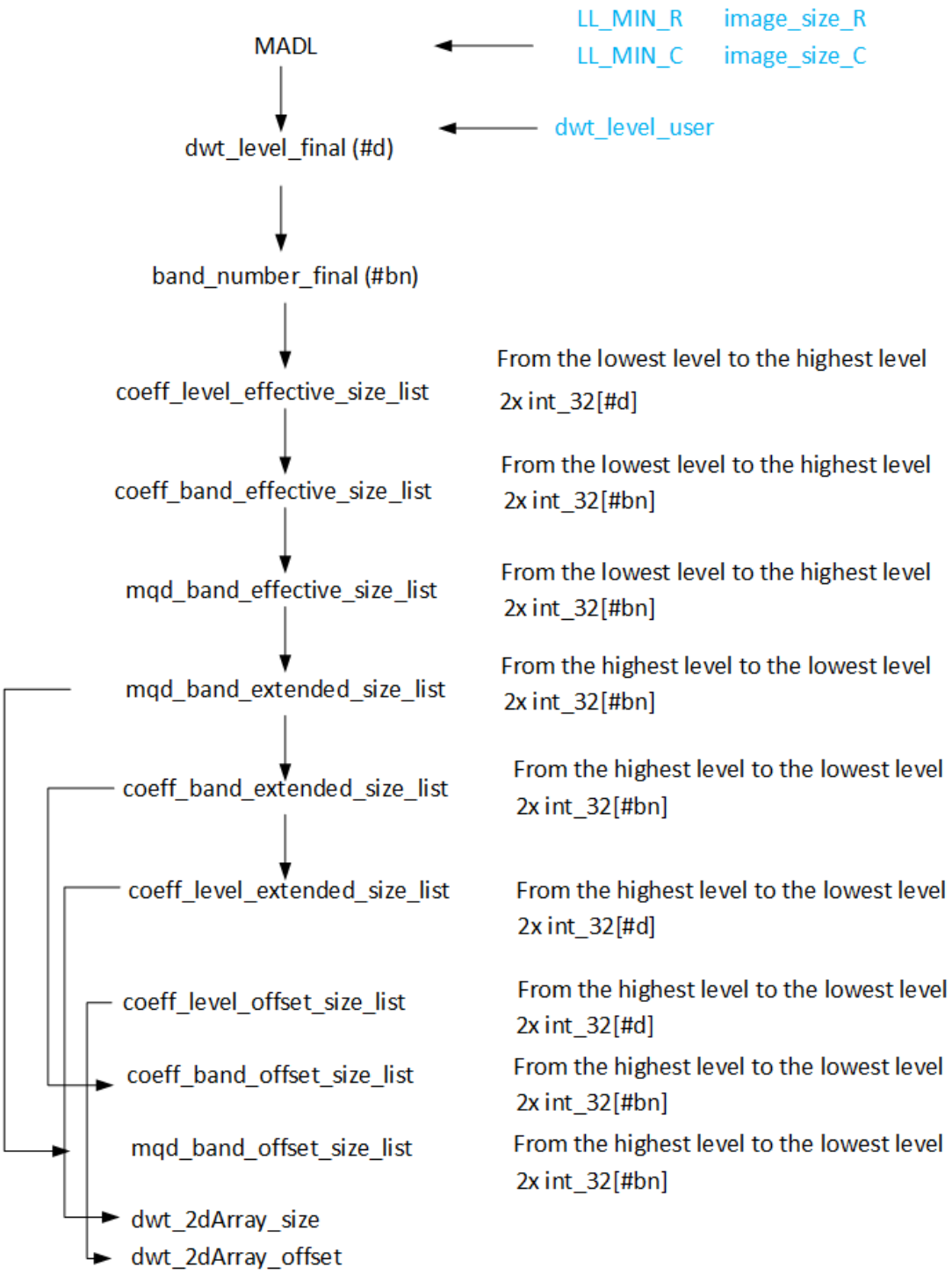


Figure 4.5 Calculation of level size, level offset, band size and band offset

Figure 4.5 shows calculation of size and offset of each level and band. Note the arrows means the dependencies between different properties. Properties without dependencies can be calculated independently.

(1) *MADL*: the detailed calculation has been describe in section 3.1.2, which depends on the minimum row number of LL band (*LL_MIN_R*), the minimum column number of LL band (*LL_MIN_C*), the input image row number (*image_size_R*) and the input image column number (*image_size_C*).

(2) *dwt_level_final*: the actual B-LDWB+B decomposition level number used in B-LDWT+B (section 3.2), which is the minimum value between *MADL* and user preferred decomposition level number *dwt_level_user*.

(3) *band_number_final*: the total number of bands generated by B-LDWT+B, which is calculated by:

*band_number_final = dwt_level_final*3 + 1;*

(4) *coeff_level_effective_size*: the effective level size of each decomposition level, which is described in section 3.1.2.

(5) *coeff_band_effective_size*: the effective band size of each band, which is described in section 3.1.2.

(6) *mqd_band_effective_size*: the effective band size of MQD matrix, which is described in section 3.1.2. Note the calculation is from the lowest level to the highest level for (4) to (6).

(7) *mqd_band_extended_size*: the extended band size of MQD matrix, which is calculated by:

mqd_band_extended_size [i] = *mqd_band_effective_size*[i], i = 0…3

mqd_band_extended_size [i] = *mqd_band_extended_size*[i-3], i = 4,…#bn – 1.

(8) *coeff_band_extended_size*: the extended band size of coefficient matrix.

coeff_band_extended_size [i] = *mqd_band_extended_size*[i]*2, i = 0,..,#bn – 1.

(9) *coeff_level_extended_size*: the extended level size of coefficient matrix.

coeff_level_extended_size [j] = *coeff_level_extended_size* [3*(*dwt_level_final* - j)]*2, j = 0,..,#d-1.

(10) *coeff_level_offset*: the offset of each coefficient level. Coefficient level offsets are used to normalize the memory fetching of boundary extension in LDWT forward and inverse transform, which has be explained in section 3.1.3. The current LDWT algorithm requires level offset to be equal to 2 in both horizontal and vertical direction.

(11) *coeff_band_offset*: the offset of each coefficient band. Coefficient band offsets are introduced to ensure the correctness of fetching any individual band's element. It is determined by both level offsets and band extended size.

Coefficient band offset is used to cluster L- and H-components after they have been calculated. Since the coefficient band can be "over-sized" i.e. the sum of sizes of L- and H-band can be larger than its original level size, the band's offset cannot be directly derived from the level size. Use the same example demonstrated in Figure 3.2, as Figure 4.6 shows, if the band's row offset at Lv5 was set to 129 rather than 130 based on the level row size, in I2C operation of horizontal 2D B-LDWT+B stage, the end of L-band will overwrite the head of H-band. To avoid possible overwriting, the band offsets is derived from coefficient level offset and band extended size:

For top-right band:

coeff_band_offset_R[i] = *coeff_level_offset_R*;

coeff_band_offset_C[i] = *coeff_level_offset_C* + *coeff_band_size_extended_C*[i],

i = 1,4 …, #bn-3.

For buttom-left band:

coeff_band_offset_R[j] = *coeff_level_offset_R* + *coeff_band_size_extended_R*[j],

coeff_band_offset_C[j+1] = *coeff_level_offset_C*;

i = 2,5 …, #bn-2.

For buttom-right band:

coeff_band_offset_R[k] = *coeff_level_offset_R* + *coeff_band_size_extended_R*[k],

coeff_band_offset_C[k] = *coeff_level_offset_C* + *coeff_band_size_extended*[k],

i = 3,6 …, #bn-1.

For LL band:

coeff_band_offset_R[0] = *coeff_level_offset_R*,

coeff_band_offset_C[0] = *coeff_level_offset_C*.

coeff_band_offset_R is the row offset along the vertical direction and *coeff_band_offset_C* is the column offset along the horizontal direction.

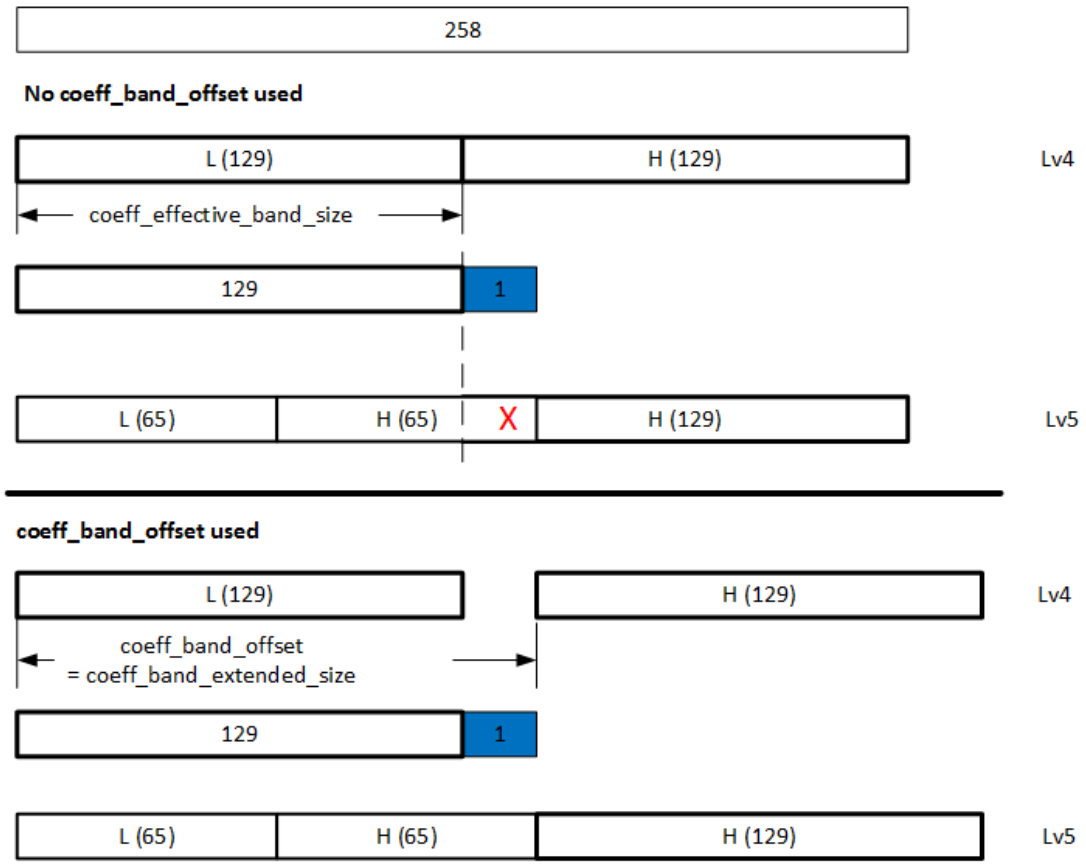


Figure 4.6 Effect of coefficient band offset in B-LDWT+B

(12) *mqd_band_offset_list*: a list containing offsets of each MQD band. MQD band offsets are introduced to ensure the correctness of fetching any individual band's element. Different from coefficient band offsets, MQD band offsets is only determined by MQD band extended size.

For top-right band:

mqd_band_offset_R[i] = 0;

mqd_band_offset_C[i] = *mqd_band_size_extended_C*[i],

i = 1,4 …, #bn-3.

For buttom-left band:

mqd_band_offset_R[j] = *mqd_band_size_extended_R*[j],

mqd_band_offset_C[j+1] = 0;

j = 2,5 …, #bn-2.

For buttom-right band:

mqd_band_offset_R[k] = *mqd_band_size_extended_R*[k],

mqd_band_offset_C[k] = *mqd_band_size_extended*[k],

k = 3,6 …, #bn-1.

For LL band:

mqd_band_offset_R[0] = 0,

mqd_band_offset_C[0] = 0.

Similarly, *mqd_band_offset_R* is the row offset along the vertical direction and *mqd_band_offset_C* is the column offset along the horizontal direction.

(13) *dwt_2dArray_size* and *dwt_2dArray_offset*: the size and offset of GPU 2D array allocated in GPU's global memory and the horizontal and vertical offset of data within the GPU 2D array.

dwt_2DArray_offset = 2

dwt_2DArray_size = 2**coeff_band_size_extended*[#bn-1] + 4

After Qmax is obtained (section 3.3), as Figure 4.7 shows, following codec properties are calculated for bitstream-related operations:

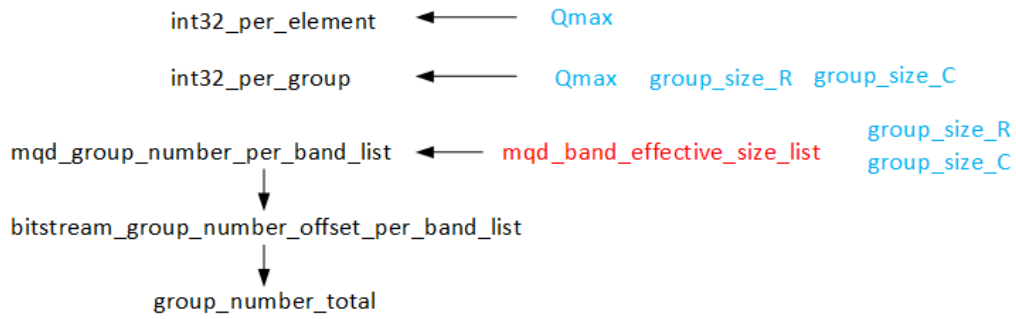


Figure 4.7 Bitstream related codec properties

(1) *int32_per_element*: the number of 32-bit integers allocated for bitstream of one element. It is calculated by

ceil((*Qmax* + 1)*5/32),

in which ceil(x) is the ceiling function that provides the nearest integer no less than x.

(2) *int32_per_group*: the number of 32-bit integers allocated for bitstream of one group. It is calculated by:

*int32_per_element*group_size_R*group_size_C*

(3) *mqd_group_number_per_band_list*: a list with #bn integers to record the number of groups per band. Note this list is used in both element encoding stage (section 3.4) and group encoding stage (section 3.5).

4.4.3 Codec Properties Calculation in Decompression Procedure

The codec properties calculation in decompression procedure is similar to that in compression procedure. The difference is in decompression procedure, all metadata, such as image size, group size, *Qmax*, etc are obtained together at the beginning stage of decompression procedure (Section 4.1). As shown in Figure 4.8, the codec properties can be calculated in one step, rather than separated in two steps as in compression procedure.

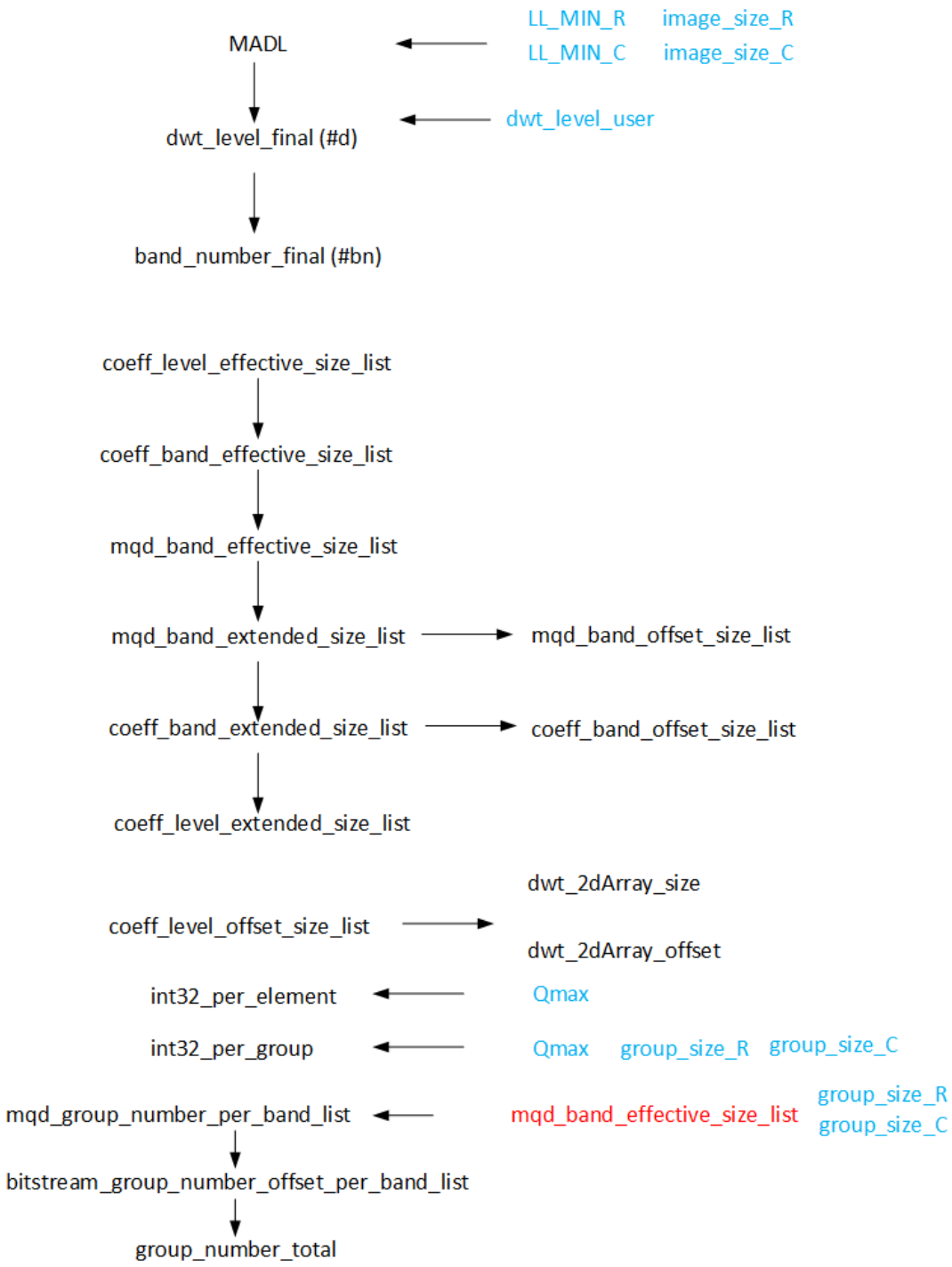


Figure 4.8 The codec properties calculation in decompression procedure

CHAPTER V

COMPARATIVE EXPERIMENTAL RESULTS OF PCWT AND JPEG-XR

The purpose of our PCWT algorithm is to achieve fast compression while maintaining a decent compression ratio. JPEG2000 is not chosen to compare with our algorithm even though it is wavelet transform based because of its high complexity and the corresponding low compression speed [25]. Our PCWT algorithm is compared with CPU version JPEG-XR compression algorithm [26], which is much faster and still achieved a competitive compression effect [27].

JPEG-XR algorithm is a Photo-transform based algorithm [26]. Similar to PCWT, its compression procedure can be divided into two phases: transform phase and encoding phase. The current standard only defines the decompression procedure with details [28], but the corresponding compression procedure is not difficult to derive as shown in Figure 5.1.

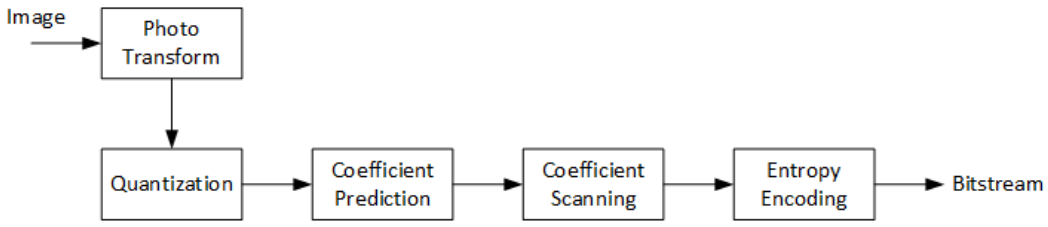


Figure 5.1 The workflow diagram of JPEG-XR algorithm

Compared to JPEG-XR, PCWT has following differences:

(1) PCWT uses a different transform. JPEG-XR uses Photo transform but PCWT uses B-LDWT+B.

(2) PCWT lossless implementation does not have quantization. JPEG-XR's quantization should be disabled in lossless mode.

(3) PCWT does not have coefficient prediction. Coefficient prediction is to increase encoding efficiency, with the assumption that relatively high similarity between blocks exists. Corresponding coefficients from different blocks have similar

values and can be encoded efficiently by entropy encoding. Coefficient prediction is not implemented in PCWT because the entropy encoding is not used in PCWT.

(4) PCWT has different coefficient scanning order than JPEG-XR. JPEG-XR has a similar Zig-Zag scanning order like JPEG because of the intrinsic low- and high-frequency component pattern in the result of Photo transform. The result of B-LDWT+B transform does not have that pattern so coefficients are scanned line by line.

(5) PCWT does not use entropy encoding. PCWT utilized MQD-based encoding, which has much higher parallelism than entropy encoding. Entropy encoding is implemented by searching the bitstream and replacing target patterns with new strings based on entropy coding table. One significant difficulty for implementing parallel entropy encoding is that entropy encoding prefers complete bitstream rather than a section of that bitstream so that it can locate as many target patterns as possible and further achieve high compression efficiency. However, this "search one whole bitstream" cannot lead to high parallelism. MQD-based encoding is highly localized and can achieve higher parallelism than entropy encoding.

Our experimental PC equips Intel i7 2.93GHz CPU with 6GB memory. The GPU hardware is Nvidia GTX570 on the same PC.

JPEG-XR compression algorithm is implemented by using Microsoft .NET 4.0 framework [29]. PCWT compression algorithm is implemented by using native C++ (host-side) and CUDA C 4.0 (device-side).Both algorithms are compiled into the release mode executable file by Microsoft Visual Studio 2010. Both executable files run on Microsoft Windows 7 64-bit.

The test is divided into two categories: compression procedure and decompression procedure. The starting point of compression procedure is that the raw image is already in host (CPU) memory and the ending point is the generated bitstream has been written on the host harddisk. The starting point of decompression procedure is that there is a compressed file on the host harddisk and the ending point is to the image has been restored in the host memory. Correspondingly, there are

compression running time and decompression running time. It is noted that both running times include the time for data transfer between host and device and memory allocation on both GPU and CPU. Those full-scale running times can provide a comprehensive measurement of our algorithm and provide an accurate comparison between our algorithm and JPEG-XR algorithm.

As listed in Table 5.1, twenty publicly accessible medical images have been tested with PCWT and JPEG-XR algorithms. All images are divided into four categories based on their pixel size: 2~5 Mega Pixels (MP) from Img #1 to #5, 5~10 MP from Img #6 to #10, 10~20MP from Img #11 to #15, and more than 20MP from Img#16 to #20. All images are transformed to 8pp gray-level image to obtain the baseline performance of both algorithms. Figure 5.2 shows four test images. One image from each category.

PCWT and JPEG-XR are compared in compressed size overhead in terms of compression ratio and encoding/decoding acceleration in terms of running time.

For compressed size overhead (generally PCWT file's size is larger than JPEG-XR's file), each image i in one category has a compression ratio (CR) from PCWT $\mathrm{CR_{PCTW}(i)}$ and another CR from JPEG-XR $\mathrm{CR_{XR}(i)}$ $\mathrm{i} = 1, \ldots 5$. The size overhead of PCWT comparing to JPEGXR can be calculated as:

$$S = \frac{\sum_{i=1}^{5} w(i) \cdot h(i) \cdot CR_{PCWT}(i)}{\sum_{i=1}^{5} w(i) \cdot h(i) \cdot CR_{XR}(i)}, i = 0,...,5,$$

in which $\mathrm{w(i)}$ and $\mathrm{h(i)}$ are the image i's width and height, respectively. $\mathrm{CR_{PCWT}}$ and $\mathrm{CR_{JPEGXR}}$ are compression ratios from PCWT and JPEG-XR, respectively.

Similarly, the acceleration of PCWT comparing to JPEG-XR can be calculated as:

$$A = \frac{\sum_{i=1}^{5} w(i) \cdot h(i) \cdot T_{PCWT}(i)}{\sum_{i=1}^{5} w(i) \cdot h(i) \cdot T_{XR}(i)}, i = 0,...,5, \quad (3)$$

in which $T_{PCWT}(i)$ and $T_{XR}(i)$ are image i's encoding/decoding running time from JPEG-XR and PCWT, respectively.

Table 5.1 Test image information

	Width (pixel)	Height (pixel)
2~5 MP		
M2to5_01	2133	2133
M2to5_02	2048	2500
M2to5_03	1800	1643
M2to5_04	1600	1322
M2to5_05	1728	1638
5~10 MP		
M5to10_01	2876	2894
M5to10_02	2048	2500
M5to10_03	3744	1640
M5to10_04	2241	2492
M5to10_05	2400	3020
10 ~ 20 MP		
M10to20_01	4280	3520
M10to20_02	5352	3000
M10to20_03	4000	3000
M10to20_04	3543	3543
M10to20_05	3047	3386
> 20 MP		
M_Larger20M_01	6144	4990
M_Larger20M_02	6098	4404
M_Larger20M_03	7024	5532
M_Larger20M_04	6000	5290
M_Larger20M_05	6070	4104

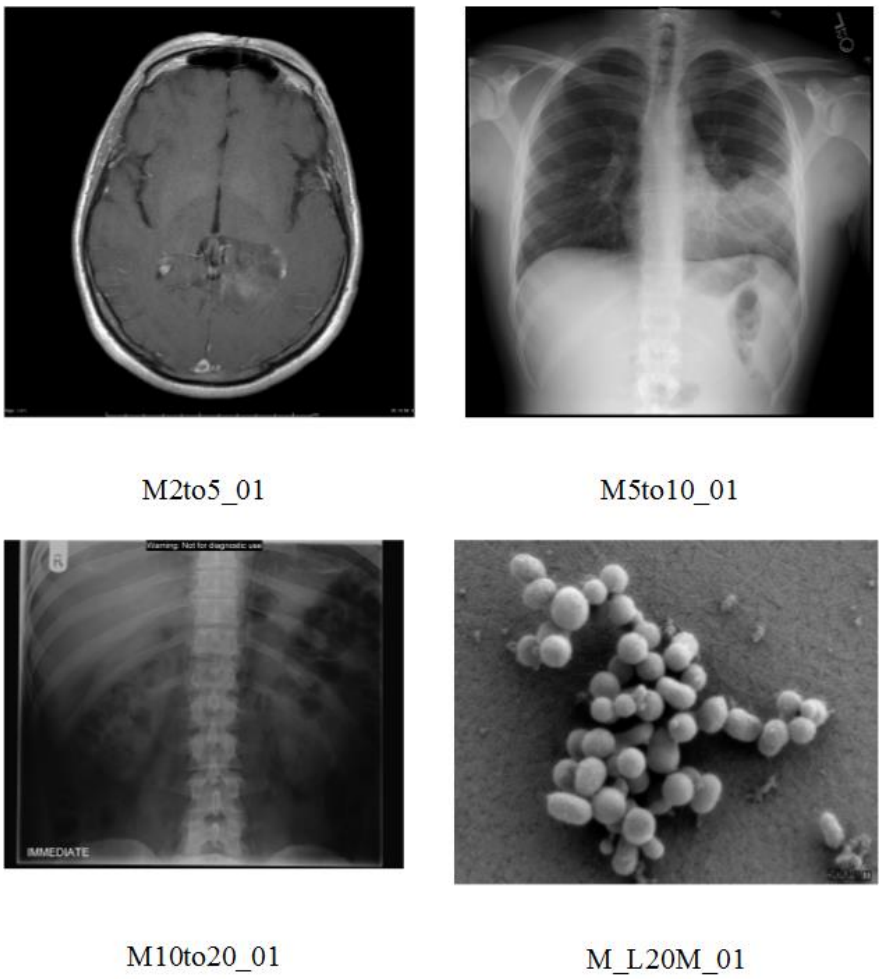


Figure 5.2 Four sample images used in the comparison

It is noted that the storage size overhead and acceleration are calculated with weights. The weighted comparison puts more weights on larger images' results, which reflects our criterion that a specific size overhead or acceleration on large images is usually more significant (for size overhead, "worse", for acceleration, "better") than the same number on small images.

Table 5.2 lists the compression ratio ("CR"), compression running time ("Comp Time") and decompression running time ("Decomp Time") of all 20 images with PCWT and JPEG-XR. Table 5.3 shows the image-size-weighted compression and compression acceleration brought by PCWT and corresponding image-size-

weighted size overhead. From Table 5.2 and Table 5.3, it can be found that PCWT is suitable for processing large image. For testing images with size larger than 20MP, PCWT can achieve more than 6 times compression speed faster than JPEG-XR, with less than 5% extra storage overhead. There is an insignificant improvement on decompression speed from PCWT, because the decoding phase in the decompression procedure is sequential.

Table 5.2 Compression ratio, compression running time and decompression running time of 20 test images by using PCWT and JPEG-XR

	PCWT CR	PCWT Comp Time (ms)	PCWT Decomp Time (ms)	JPEG-XR CR	JPEG-XR Comp Time (ms)	JPEG-XR Decomp Time (ms)
M2to5_01	6.24	41	66	6.78	106	117
M2to5_02	3.20	44	96	3.66	152	167
M2to5_03	3.15	39	59	3.39	97	102
M2to5_04	2.48	40	47	2.61	76	78
M2to5_05	2.87	40	57	2.97	98	100
M5to10_01	2.83	62	170	3.36	274	271
M5to10_02	3.20	45	95	3.66	151	165
M5to10_03	4.35	51	102	4.17	173	180
M5to10_04	1.95	55	135	2.01	212	211
M5to10_05	3.82	53	129	4.71	198	215
M10to20_01	2.70	81	299	2.99	519	534
M10to20_02	3.60	94	292	4.41	464	453
M10to20_03	2.28	70	268	2.43	459	451
M10to20_04	4.06	72	212	4.07	352	366
M10to20_05	3.08	68	202	3.09	348	335
M_L20M_01	1.36	195	940	1.37	1211	1186
M_L20M_02	4.85	136	459	5.85	692	681
M_L20M_03	1.59	234	1241	1.61	1510	1507
M_L20M_04	1.79	176	962	1.68	1239	1244
M_L20M_05	2.50	124	604	2.77	936	938

Table 5.3 Weighted compression acceleration, weighted decompression acceleration and weighted size overhead for four image categories.

	PCWT to JPEG-XR weighted compression acceleration	PCWT to JPEG-XR weighted decompression acceleration	PCWT to JPEG-XR weighted size overhead
2~5 MP	2.69	1.73	1.08
5~10 MP	3.77	1.65	1.12
10~20 MP	5.55	1.68	1.09
> 20 MP	6.44	1.34	1.04

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

PCWT is a fast and efficient wavelet tree based encoding algorithm. With carefully designed parallel lossless wavelet transform and coefficient encoding algorithm, PCWT can achieve more than 6 times faster encoding speed than the standard JPEG-XR implementation, with less than 5% extra storage overhead. Scalability of PCWT is high because most of the parallel steps do not have any hardware specification limitations. PCWT is also extensible. The current PCWT implementation is lossless, which meets the requirement of most medical image application. However, it is not difficult to change it into a lossy mode of operation.

The main research contribution of this dissertation is that it did not only describe an accelerated compression algorithm within the context of GPGPU but also tried to demonstrate a GPGPU approach to solve existing sequential problems. GPGPU approach has its characteristic advantages and disadvantages, which rise from the GPU hardware and programming models. In this research, limited hardware resource and relatively slow global memory I/O access speed of CUDA are identified as disadvantages. All the modifications and re-designs in this research are made to eliminate or at least relieve the impacts of those disadvantages on our final algorithm, without largely sacrificing its advantages. These modifications and re-designs include B-LDWT+B and two-stage encoding and exclude line-based implementation of BCWT [17] because of its nonadaptive characteristics. The experimental result is reported in a comprehensive manner by including data transfer time to demonstrate the author's acknowledgement of GPGPU's unique overhead and to propose a more accurate performance evaluation method of GPGPU solution.

There are still a few possible improvements which can be addressed in the future as listed below:

(1) Parallel multi-band MQD calculation. MQD bands within the same level can be calculated independently but in the current implementation, they are calculated

sequentially. Replacing the current sequential band calculation with parallel multi-band calculation can introduce higher parallelism into the algorithm, which can achieve higher compression speed.

(2) Multi-pass Qmax horizontal reduction. The horizontal reduction in Qmax search is the only place in the current implementation that assumes the data being processed can fit into one block's shared memory. Multi-pass horizontal reduction will bring more generalized solution and it can be implemented similarly to the multi-pass vertical reduction.

(3) Atomic operation based group encoding. Multiple passes in group encoding stage in compression procedure are used to avoid race condition that could generate false group encoding bitstream. Atomic operation can eliminate race condition between threads and further eliminate multiple passes in the group encoding stage. Each thread can handle one element rather than one group. However, atomic operation is generally slower than non-atomic operation, the atomic operation based group encoding is not necessarily faster than the current implementation.

In summary, the newly developed PCWT offers an efficient and fast wavelet based lossless encoding/decoding algorithm which outperforms the current standard, lossless JPEG-XR. PCWT can be easily modified to operate in lossless or lossy mode as required and will keep evolving with the algorithm improvement and hardware progress.

BIBLIOGRAPHY

[1] L. Landro, "Where Do You Keep All Those Images?," 2013. [Online]. Available: http://online.wsj.com/news/articles/SB10001424127887323419104578374420820705296.

[2] AT&T, "Medical Imaging in the Cloud," [Online]. Available: http://www.corp.att.com/healthcare/docs/medical_imaging_cloud.pdf.

[3] International Telecommunication Union, "Information Technology Digital Compression and Coding of Continuous-tone Still Images," 1993. [Online]. Available: http://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[4] International Telecommunication Union, "Information Technology - JPEG 2000 Image Coding System," 2002. [Online]. Available: http://www.ic.tu-berlin.de/fileadmin/fg121/Source-Coding_WS12/selected-readings/20_T-REC-T_1_.800-200208-I__PDF-E.pdf.

[5] International Telecommunication Union, "Information Technology - JPEG-XR Image Coding System," 2012. [Online]. Available: http://www.itu.int/rec/T-REC-T.832-201201-I/en.

[6] C. Tu, S. Srinivasan, G. Sullivan, S. Regunathan and H. Malvar, "Low-complexity hierarchical lapped transform for lossy-to-lossless image coding in JPEG XR / HD Photo," in Applications of Digital Image Processing XXXI, 2008.

[7] J. Shapiro, "Embedded Image Coding Using Zerotree of Wavelet Coefficients," IEEE Transactions on Signal Processing, vol. 41, no. 12, pp. 3445-3462, 1993.

[8] R. Slone, D. Foos, B. Whiting, E. Muka, D. Rubin, T. Pilgram, K. Kohm, S. Young, P. Ho and D. Hendrickson, "Assessment of visually lossless irreversible image compression," Radiology, vol. 215, no. 2, pp. 543-553, 2000.

[9] M. Adams and R. Ward, "Wavelet transforms in the JPEG-2000 standard," in IEEE Pacific Rim Conference on Comunications, Computers and Signal Processing, 2001.

[10] W. van der Laan, A. Jalba and J. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," IEEE Transactions on Parallel and Distributed System, vol. 22, no. 1, pp. 132-146, 2011.

[11] J. Guo, S. Mitra, B. Nutter and T. Karp, "A fast and low complexity image codec based on backward coding of wavelet trees," Data Compression Conference Proceedings, pp. 292-301, 2006.

[12] A. Said and W. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," IEEE Transactions on Circuits and Systems for Video Technology, vol. 6, no. 3, pp. 243-250, 1996.

[13] D. Taubman, "High performance scalable image compression with EBCOT," IEEE Transaction on Image Processing, vol. 9, no. 7, pp. 1158-1170, 2000.

[14] Nvidia, "GPU Application," [Online]. Available: http://www.nvidia.com/object/gpu-applications.html.

[15] J. Matela, "GPU-Based DWT acceleration for JPEG2000," Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, pp. 136-143, 2009.

[16] M.-C. Che and J. Liang, "GPU implementation of JPEG XR," in Proc. of SPIE-IS&T Electronic Imaging, 2010.

[17] L. Ye, "Fast and low memory usage coding for image and video based on wavelet transform," Texas Tech University, 2007.

[18] Nvidia, "CUDA Zone," [Online]. Available: https://developer.nvidia.com/category/zone/cuda-zone.

[19] Khronos, "The open standard for parallel programming of heterogeneous systems," [Online]. Available: https://www.khronos.org/opencl/.

[20] D. Kirk and H. Wen-mei, Programming massively parallel processors (2nd Ed), Morgan Kaufmann, 2012.

[21] Nvidia, "CUDA Toolkit Document," [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[22] Stackoverflow, [Online]. Available: http://stackoverflow.com/questions/6490572/cuda-how-many-concurrent-threads-in-total.

[23] M. Boyer, "CUDA kernel overhead," [Online]. Available: http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html.

[24] Mathworks, "GPU programming in MATLAB," [Online]. Available: http://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html.

[25] R. Veerla, Z. Zhang and K. Rao, "Advanced image coding and its comparison with various still image codecs," American Journal of Signal Processing, vol. 2, no. 5, pp. 113-121, 2012.

[26] S. Srinivasan, C. Tu, S. Regunathan and G. Sullivan, "HD photo: a new image coding technology for digital photography," in Proc. of SPIE Applications of Digital Image Processing XXX, 2007.

[27] F. Fiorucci, G. Baruffa and F. Frescura, "Objective and subjective quality assessment between JPEG XR with overlap and JPEG 2000," Journal of Visual Communication and Image Representation, vol. 23, no. 6, p. 835–844 , 2012.

[28] F. Dufaux, G. Sullivan and T. Ebrahimi, "The JPEG XR image coding standard," IEEE Signal Processing Maganize, vol. 26, no. 6, pp. 195-200, 2009.

[29] Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/system.windows.media.imaging.wmpbitmapencoder(v=vs.100).aspx.