GPU-Based Implementation of JPEG2000 Encoder

M. Ahmadvand¹, and A. Ezhdehakosh²
¹ Department of Computer Engineering and IT, Hamedan University of Technology, Hamedan, Iran
² Department of Computer Engineering and IT, Amirkabir University of Technology, Tehran, Iran

Abstract - JPEG2000 has become one of the most rewarding image coding standards. It provides a practical set of features which weren’t necessarily available in the previous standards. The features were realized as a result of two new techniques, namely the Discrete Wavelet Transform (DWT), and Embedded Block Coding with Optimized Truncation (EBCOT). The complexity of EBCOT Tier-1 makes its implementations very difficult and time consuming.

In this paper, we focus on accelerating JPEG2000 encoder by using general-purpose processing on Graphical Processing Unit (GPU). We use CUDA platform to implement DWT and EBCOT Tier-1 as the most important sections of JPEG2000. Resulting implementation of proposed architecture performs very well compared to other available implementations.

Keywords: JPEG2000, GPU Computing, CUDA

1 Introduction

JPEG2000 has become one of the most rewarding image coding standards. It provides a practical set of features which weren’t necessarily available in the previous standards. JPEG2000 [1][2] offers numerous advantages over JPEG. These advantages include: ROI (Region Of Interest) coding, quality vs. resolution compression, lossless and lossy compression, progressive image compression/transmission by resolution/quality, random code-stream access and error resilience. Such characteristics add to the functionality of a system that is employing JPEG2000 as an image compression technique. The features and performance of JPEG2000 make this standard superior to JPEG. The features were realized as a result of two new techniques, namely the Embedded Block Coding with Optimized Truncation (EBCOT) [3]-[6] and Discrete Wavelet Transform (DWT) [7]. The complexity of EBCOT Tier-1 makes its implementations very difficult and time consuming.

During the process of encoding, an image is partitioned into data matrices called Tile-components. Each Tile-1 component is then coded separately. The process of coding is made up of different sections. These sections are depicted in Figure 1 and each is described below.

1.1 Component Transform

This section is optional in JPEG2000 and is used to improve compression efficiency [3]. The transform converts the RGB data into another color representation, with a luminance (or intensity) channel and two color difference channels.

![Figure 1. JPEG2000 encoder block diagram](image)

1.2 Discrete Wavelet Transform (DWT)

DWT [7] is a domain transform that transforms an image Tile-component from special domain to frequency domain and provides a special decorrelation. This transform can be executed for as many levels as necessary. The spatial decorrelation provided by the DWT improves as the number of transform levels increases. The output of each level of DWT is categorized into four sub-bands. DWT can be performed either by the traditional convolution based filter or by the lifting scheme based filter which has lower computational complexity compared to the former filter.

1.3 Quantization

Quantization [4] is the process by which the sub-band samples generated by the DWT are mapped onto quantization indices for coding.

1.4 EBCOT Tier-1

This section receives the quantized wavelet coefficients and encodes them into bit-streams. These coefficients are sliced into code-blocks before they are fed into the EBCOT Tier-1 [4]. EBCOT Tier-1 is composed of two parts: Bit-Modeler and MQ-Coder [5][6]. Bit-Modeler is a bit-plane (a matrix that contains all the bits of the same order of all the coefficients of a code block) coder. A Bit-Modeler exploits the symmetries and redundancies within and across the bit-planes and generates corresponding contexts for each bit. After the context is generated, the MQ-Coder will code the bits (decisions) based on their associated contexts.
Table 1 Run time percentage for different modules in JPEG2000 encoder

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lossy</th>
<th>Lossless</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Transform</td>
<td>10.1</td>
<td>3.64</td>
</tr>
<tr>
<td>DWT</td>
<td>25.14</td>
<td>10.41</td>
</tr>
<tr>
<td>Quantization</td>
<td>6.4</td>
<td>N.A.</td>
</tr>
<tr>
<td>EBCOT Tier-1</td>
<td>44.86</td>
<td>67.35</td>
</tr>
<tr>
<td>EBCOT Tier-2</td>
<td>13.5</td>
<td>18.6</td>
</tr>
</tbody>
</table>

1.5 EBCOT Tier-2

EBCOT Tier-2 [2] is for rate allocation. The rate allocation is responsible for acquiring the highest quality for the output while maintaining a predetermined resolution, or acquiring the highest resolution while maintaining a predetermined image quality.

The execution time of different modules in the JPEG2000 algorithm is presented in Table I. It is noted from this table that DWT and EBCOT Tier-1 algorithms, as the main modules in JPEG2000 standard, occupies %70 of the execution time of the whole procedure. In this paper, we focus on accelerating JPEG2000 encoder by using general-purpose processing on Graphical Processing Unit (GPU) [9]. We used CUDA [10] platform to implement DWT and EBCOT Tier-1 as the most important sections of JPEG2000. Resulting implementation of proposed architecture performs very well compared to other available implementations.

This paper is organized as follows: in the next section a deep analysis of the EBCOT Tier-1 will be presented. In section III DWT transform will be described. In the next section GPU computing using CUDA will be explained. Our proposed implementation is introduced in sections V-VI. Experimental results are presented in section VII followed by the conclusion.

2 EBCOT Algorithm

EBCOT [3] is a two-tiered coder, where the first tier is a block coder and the second tier is for rate-distortion optimization and bitstream formation. Although Tier 2 is part of EBCOT algorithm, the practical implementation detail is not defined in the standard and not restricted in the design of an encoder.
passes. Pass 1 is named “Significant Propagation Pass.” During pass 1 scanning, those samples that are currently insignificant, but have at least one immediate significant neighbor are coded first. Clearly, these samples are most likely to become significant. Pass 2 is called “Magnitude-Refinement Pass.” Samples that have become significant in previous bit-planes are coded in this pass. The last pass, pass 3, is “Clean-Up Pass.” Samples not coded in the first two passes are coded in this pass. The two bits that are coded in pass 2 in this bit-plane have become significant in the previous bit-plane. Those bits coded in pass 1 are near those two pass 2 positions due to the significance propagation characteristic.

Once a bit is checked and decided to be coded in one pass, its context is generated according to the status of its neighbors using four coding primitives, Zero Coding (ZC), Run-Length Coding (RLC), Sign Coding (SC), and Magnitude Refinement (MR) primitives. There are total 19 contexts defined in JPEG 2000 [3]. The context of a bit should be generated and sent to arithmetic encoder along with the bit to be coded. Among the four coding primitives, ZC and SC primitives are used in the first and the third pass, MR primitive is used in the second pass only, and RLC primitive is used in the third pass. ZC and MR primitives check the eight immediate neighbors’ significant states to decide the context, and the RLC primitive is applied when the four bits in a column do not have any significant neighbors. The SC primitive has to check the four immediate neighbors’ sign and significant states.

After the context is generated, the arithmetic encoder will code the bits (decisions) based on their associated contexts, called MQ-Coder. Generally, a BAC encodes a code-stream consisting of a sequence of symbols. Each symbol (logic ‘0’ or logic ‘1’) is classified into one of these categories: the More Probable Symbols (MPS), and the Less Probable Symbols (LPS), based on the probability of their occurrence. In BAC an interval is considered as a probability model. This interval is divided into two subintervals, each one, corresponding to the probability of each symbol. When a symbol occurs, the subinterval associated with that symbol becomes the new interval. The recursive splitting of the current interval continues until all symbols are received. When the last symbol is received the characteristics of the last subdivided interval represents the encoded data.

As indicated above, BAC algorithm requires many multiplication operations in order to encode each symbol, and multiplication is time consuming. In addition, since a compressed data will only be generated when the last symbol of an input stream has been received by the encoder, serious loss of data occurs when the last symbol of a stream is not received. Finally after each subdivision of the probability model, the precision required for presenting the new interval increases. This leads to an increase of the required storage space for the interval values.

The MQ-Coder is an adaptive BAC implementation used in the JPEG2000 standard. MQ-Coder has eliminated multiplications by choosing special extremes for the intervals that are used in the probability model. In addition, the MQ-Coder periodically sends out the last byte of the stream which represents the encoded data, therefore addressing the problems associated with the increasing precision and compressed data being generated only after receiving the last symbol.

3 Discrete Wavelet Transform (DWT)

Discrete Wavelet Transform (DWT) [7] is a broadly used digital signal processing technique with application in diverse areas. DWT allows us to study a digital signal in different resolutions as sets of coarse and fine values. Wavelet transforms are used in domains of digital speech recognition, multi-resolution video processing or data compression. In the context of JPEG2000 standard, DWT is the key prerequisite of the compression process. Most of advanced features of JPEG2000 rely on DWT as well as the superior low-bitrate performance does.

JPEG2000 standard species use of LeGall (CDF) 5/3 DWT filter-banks [8] for lossless compression process and Daubechies-Feauveau (CDF) 9/7 DWT filter-banks [8] for lossy processing. Wavelet transforms can be implemented by convolution or by lifting scheme.

The advantage of lifting scheme over convolution is in reduced memory and computational complexity. Lifting scheme allows for in-place data manipulation and reduces memory dependencies.

Lifting scheme analysis [7] proceeds as follows. An input signal is split into even and odd subsequences denoted as and respectively. These values are further modified using alternating prediction (denoted as p) and update (denoted as u) steps. In the prediction step, the algorithm takes an odd sample in a turn and subtracts a linear combination of its (even) neighbors from it; a prediction error is formed:

\[ d^1_i = d^0_i - p(s^0_i + s^0_{i+1}) \]  
\[ (1) \]

In the update step, a linear combination of already modified adjacent odd samples is added to each even sample and updated even sequence is formed:

\[ s^1_j = s^0_j + u(d^1_{j-1} + d^1_j) \]  
\[ (2) \]

The output of the last update stage, , is actually a low-pass output of DWT filter and similarly output of the last prediction stage, is a high-pass output of the filter. So the result of the wavelet transformation is a signal divided into low-pass and high-pass subbands.

2D signals (e.g., images) are usually transformed in both dimensions. 1D DWT transform is first applied to all rows then to all columns resulting in four subbands LL, HL, LH, and HH. The LL subband is an approximation of the original signal and can be further transformed recursively.
4 GPU computing based on CUDA platform

CUDA [10] is software and hardware platform designed for general purpose computing on GPUs. GPUs have a parallel architecture capable of running thousands of threads in parallel. In CUDA computing model, such threads are grouped into so called thread blocks. Threads within a block can cooperate among themselves by sharing data through a shared memory. As opposite to a large global memory, shared memory is relatively small and very fast. The advantage of global memory is that it is accessible to all threads, whereas shared memory is visible only to threads of the block. Common work flow is to copy data from RAM to global memory of GPU. Once data is ready in global memory, a GPU program can be executed. Each thread block initially fetches a small portion of data from the global memory into the shared memory. Data is then processed by threads in the block and the result is moved back to the global memory.

The global memory access pattern is perhaps the most important performance consideration in programming for the CUDA architecture. In a nutshell, when 16 adjacent threads access adjacent locations in global memory then memory loads and stores are coalesced in one transaction.

5 GPU-Accelerated implementation of DWT

The key part of our GPU-accelerated DWT is the design how to split the work between thread blocks in order to provide maximum utilization of the GPU. Since we need to compute the transform in both dimensions, it is natural to choose a 2D partitioning of source image data. Also size and shape of thread blocks needs to be determined. Because the lifting scheme algorithm alternately works with even and odd samples, an efficient approach is to have one even and one odd data sample per each thread in a block, i.e., to have twice as much data samples as threads in each block. Resulting partitioning of image data. Each thread block has its dimensions Bx×By where Bx = Dx, By = Dy/2 , and Bx,By and Dx,Dy denote number of threads and samples in horizontal and vertical direction respectively and s[x][y] is the shared memory 2D array. Note that we propose transposed thread mapping for efficient data processing as follows. Threads are directly mapped into the upper half of block only, so that we have to change the thread mapping to be able to process whole block. In (3) and (4), we have swapped 3 thread indices Tx, Ty so that the threads cover the left half of the data block instead of the upper half which was covered originally. (3) then predicts all odd samples and (4) updates all even samples in the block. To calculate the second dimension of the transform, we just apply same filters to the columns.

The result of the application of lifting filters to rows and columns is composed of coefficients of four DWT subbands. Coefficients of LL and HL subbands are alternately located on even rows and LH and HH coefficients on odd rows of the shared memory s.

The final step of the CUDA-based transform is to move result from shared memory back to global. Particular subbands, however, needs to be stored separately in global memory. Because there are twice as much data samples as threads in the block, we store even lines first. Even lines contain all LL and HL samples and because those are interleaved, we use first half of threads to store all LL samples and second half to store HL samples. The access to the global memory is hereby coalesced.

Note that proposed implementation of DWT is optimized for maximum performance and its limitation is that it does not take into account sample values exchange between blocks borders. The proposed algorithm does not introduce any visual artifacts provided both forward and reverse transformations work with the same data blocks dimensions.

6 GPU-Accelerated implementation of EBCOT Tier-1

Architecturally, the parallel algorithm for the bit plane coder is optimized to match the existing graphics hardware. A number of code blocks are processed independently and each code block samples should be processed in parallel using the
state prediction method. Hence, each work group is in charge of handling one code block, and multiple processing elements (PE) of the work group can process the samples in parallel. In particular, the usage of memory resources has to be carefully optimized, control flow operations must be minimized as they result in costly processor stalls, and the workload must be distributed so as to maximize hardware resources occupancy and hide memory latency when stalls are unavoidable.

In GPGPUs any flow control instruction (i.e. if, switch...) can significantly affect the instruction throughput by causing threads of the same parallel thread block to diverge; that is, to follow different execution paths. GPGPUs provide great arithmetic capability at low hardware cost, but to achieve this goal, the cores in each multiprocessor often share only one instruction decoder and one small branch control unit. Therefore a single instruction is executed over N threads in parallel, where N is specific to the hardware chip and often has value of 32 or 64. As a result, control instructions and branch divergences on GPGPUs tend to be very expensive.

Unfortunately, the context formation process in JPEG2000 requires many control operations. For example, when the BPC scans a 3×3 window of neighboring samples (8 neighbors of a given sample), the algorithm may take 256 different execution paths. Additionally, the BPC needs to select one out of 19 contexts based on that information. If the bitplane coder (BPC) is implemented with a standard switch/case construct its performance would be bound to be very low. Fortunately, the context decision rules are predefined so look-up-tables (LUT) for context formation can be constructed to avoid the branching control flow. A LUT should have 256 entries, where the indices for the entries are formed from the 8 neighbor state bits and the value is selected based on the context rule. However it is inefficient to concatenate the state bits to form a LUT index every time. Therefore the state bits are stored in a 16-bit state flag instead, as shown in Figure 3. Each sample has one corresponding state flag that stores the state information itself and the state bits of its neighbors. This organization allows the BPC to easily retrieve an LUT index by applying a binary mask.

![Figure 3. 16-bit state flag of a sample. The index of ZC or SC LUT can be extracted from the flag by applying a binary mask](image)

It is very critical to optimize memory usage for the BPC where the context formation process executes massive numbers of memory and arithmetic operations. Particularly on GPGPUs, an efficient memory utilization can not only significantly reduce the latency but also can increase the resource occupancy of computational resource to speedup the computing time.

The first optimization considered is to efficiently allocate different sets of data into the most suitable of memory blocks based on the application’s demand to reduce latency and conflict. When the BPC processes one sample, it not only refers to the sample but also to its 8 neighbors. Consequently, there is a high degree of memory conflicts in the BPC, particularly in parallel BPC where multiple threads concurrently access different samples. However, both the memory conflict rate and memory latency can be dramatically reduced with very fast, multi-way shared memory that resides locally on chip. The shared memory on modern graphics cards has from 16 to 32 banks which can be accessed independently with a latency of only one clock cycle. It is also very important to optimize allocation of the context output buffer since it is the intermediate buffer for context formation and arithmetic coder. Storing this buffer in the global memory would be very inefficient since the BPC would write to global memory and then the arithmetic coder would have to read back from global memory immediately after BPC write. Therefore this output buffer should be also stored in on-chip shared memory. Additionally, since the BPC refers to the LUTs and the state flags very frequently, these data structures should also be placed in the shared memory as well. The LUTs are read-only and small enough to reside in fast the constant cache memory. The code blocks are initially stored in the off-chip global memory then each multiprocessor will copy its respective code block into its shared memory. The LUTs are stored in constant memory and fetched to multiprocessors’ constant cache at runtime.

After the data sets are efficiently allocated into selected memory regions, the utilization of memory, especially shared memory, should be minimized to increase the multiprocessor occupancy of the GPGPUs.

The multiprocessor occupancy is defined as the ratio of the number of resident warps to the maximum number of warps supported on a multiprocessor of a GPU. Typically the higher occupancy the better multiprocessor can hide the warps latency and increase ALU utilization which will yields better speedup. Each multiprocessor on a GPU has a set of registers and a small amount of on-chip shared memory. These resources are shared among the active thread warps. Therefore the lower shared resources are utilized by a particular warp, the higher number of warps can reside in a multiprocessor. The compiler can attempt to minimize register usage but the utilization of shared memory must be optimized by the programmer. Additionally, the number of threads per work group should be large enough, at least 4× of the warp size, to achieve the best performance.

However, it is not simple to reduce shared memory utilization in the parallel EBCOT Tier-1 coder since it...
depends on the code block size which is one of the key parameters that determine JPEG2000 compression efficiency. The code block size is often varied from 8×8 to 64×64 samples, but using the largest allowed configuration is preferred because the larger code block size the better compression efficiency. On the other hand, a code block size requires a large shared memory buffer which may reduce the multiprocessor occupancy and hence reduce the performance speedup. In a naïve implementation, at least 24 Kbytes of shared memory are needed to store a 64 × 64 code block and the two respective state flag, while a 8 × 8 code block requires only a small 256-byte buffer. The results show that by changing code block size from 64 × 64 to 8 × 8, the compression efficiency drops about 25% on image Bike. But by decreasing code block size the multiprocessor occupancy can be significantly improved from 17% to 50%. There is an unusual result with the 8×8 code block where the occupancy no longer increases.

It clearly shows that the large code block significantly decreases the multiprocessor occupancy which results in significant speedup drop. To overcome this problem, previous studies had to compromise compression efficiency to use a GPU-affordable code block size such as 8 × 8 but this is an impractical size. This study therefore manages to design a special strategy that can handle large code blocks with low utilization of shared memory.

7 Experimental results

This section presents the result of our proposed implementation. The parallel JPEG2000 coder is implemented using CUDA on the Nvidia SDK 3.2 running on an Nvidia GTX480 graphic card. The reference CPU platform uses an Intel Core i7, with 12GB RAM running at 2.8GHz.

There are several popular versions of JPEG2000 compression software running on CPUs, including JasPer [11], and OpenJPEG [12]. JasPer is chosen to compare against the GPU implementation since it is an open source program, with fully accessible source code, and very good performance. The image test set includes the most popular JPEG2000 test images (bike, woman, cafe, and lena).

Table II compares runtime for JasPer and the GPU implementation JPEG2000 coder. The GPU-based JPEG2000 encoder implementations are more than 17× faster than the JasPer implementation.

<table>
<thead>
<tr>
<th>Image</th>
<th>Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>lena</td>
<td>15.23</td>
<td>238.74</td>
</tr>
<tr>
<td>cafe</td>
<td>267.36</td>
<td>5214</td>
</tr>
<tr>
<td>bike</td>
<td>284</td>
<td>4398.28</td>
</tr>
</tbody>
</table>

8 Conclusion

In this paper, the design and development of a novel PEG2000 encoder are presented. The parallel algorithm can process data at the sample-level. In particular, this paper is the first to presents a fully parallel solution for the arithmetic coder and DWT in JPEG2000. The implementation of the JPEG2000 coder leverages widely available and massively parallel GPGPU hardware and provides a 17× performance speedup compared to the JasPer software implementation. It is believed that even greater speedup is possible with full 64-bit hardware support. Additionally, the proposed parallel algorithms are potentially applicable to a wide range of image processing and data compression applications.

For future work, the emphasis will be on further improved implementations of the arithmetic coder. In addition, the Tier-2 routines can be parallelized in order to have a complete JPEG2000 encoding flow running on a GPU platform. Another research direction is that of implementing the proposed parallel solutions on different parallel hardware platforms to compare the different architectures on the performance and optimization strategies.

9 References


