Optimized GPU Implementation of JPEG 2000 for Satellite Image Decompression

Derviş Utku Ufuk*‡, Alptekin Temizel†, Ahmet Murat Özbayoğlu‡
*Space Technologies Research Institute, TÜBİTAK, Ankara, Turkey
utku.ufuk@tubitak.gov.tr
†Graduate School of Informatics, METU, Ankara, Turkey
atemizel@metu.edu.tr
‡Department of Computer Engineering, TOBB ETÜ, Ankara, Turkey
{dufuk, mozbayoglu}@etu.edu.tr

***Abstract*—JPEG 2000 is a powerful yet computationally complex image compression algorithm which is widely used in remote sensing applications. In this work, we focus on speeding-up the decompression algorithm by applying various GPU optimization techniques. We have conducted numerous experiments on high-resolution satellite images in two operational modes; a synchronous mode and an asynchronous batch mode. We share our experiment results and make performance evaluations regarding each operational mode and optimization method separately. Finally we propose an optimized GPU architecture for satellite image decompression and compare the achieved performance with a multi-threaded CPU architecture.**

***Index Terms*—JPEG 2000, GPU, CUDA, image compression, parallel processing, satellite imagery**

I. Introduction

JPEG 2000 is an image compression algorithm which is widely used in remote sensing, medical imaging, and many other fields due to its high compression performance, error resilience, scalability and support for both lossy and lossless modes. However, the underlying algorithmic complexity is the biggest drawback of this algorithm and thus it can become a bottleneck in some applications such as satellite image processing.

There have been quite a few studies on increasing the performance of JPEG 2000. While the focus had been on FPGA and ASIC architectures until 2011 prior to popularity of GPUs [1]–[4], it has shifted to GPUs in the recent years. Hovewer, most of the recent studies only focus on the encoder algorithm and not on the decoder [5]–[7]. On top of that, some of the proposed architectures even deviate from the standard and break compatibility for the sake of increased efficiency and data-level parallelism [8]–[10].

There are only a few studies that both cover the decoder and retain compatibility with the original standard. Some of these proposed architectures therein solely rely on GPU parallelism [11]–[13]; and some involve CPU-GPU heterogeneous parallelism [14]. One thing these studies share in common is that they cannot go beyond a certain granularity of parallelism, because of the fact that the JPEG 2000 algorithm in its nature is not very suitable for the SIMD (single instruction multiple data) architecture. This limits the potential speed-up that could be achieved with GPU parallelism.

In remote sensing applications, satellite images are encoded onboard before they are downloaded from a ground station. Since the encoder is embedded within the satellite and cannot be modified, compliance with standards is mandatory in most cases. This is why we stick to the original JPEG 2000 standard in this work.

In Section II, we present an overview of the JPEG 2000 algorithm. We give detailed information about test data and computation environment in Section III. Then in Section IV, we analyze the impacts of a variety of GPU optimization techniques on the decoder performance. We make an overall performance evaluation and also compare the obtained performance results with a reference multi-threaded CPU implementation in Section V. Finally we discuss the future work arising from our studies and conclude our discussion in Section VI.

II. JPEG 2000 Overview

The JPEG 2000 standard is made up of several and incremental parts. In this paper we focus on the Part I (i.e. Tier I) of the standard, because it defines the minimum compliant codec and the core coding system. In the remainder of this section, we briefly go over the encoding and decoding processes and the underlying algorithms.

During encoding, firstly the input image is divided into square tiles. Then those tiles are further divided into square codeblocks after a discrete wavelet transform (DWT) as shown in Figure 1. In our design, a 3-level DWT is applied on tiles with 256×256 pixel dimensions. Consequently, each tile is divided into 64 codeblocks of 32×32 pixels after DWT.

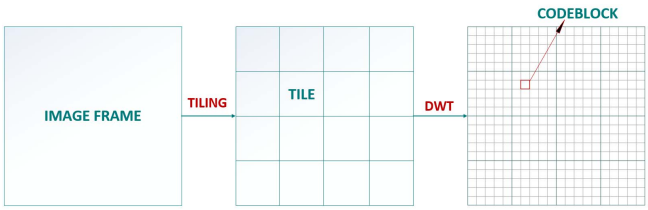

Fig. 1. JPEG 2000 Tiles and Codeblocks

978-1-5386-7649-3/18/$31.00 ©2018 IEEE
DOI 10.1109/CSE.2018.00014

The default reversible DWT in JPEG 2000 is implemented by the Le Gall 5-tap/3-tap filter [15]. This is the filter that we use in our implementation, because we only experiment with lossless decompression in order to analyze the worst-case performance scenario. The high-pass and low-pass coefficients produced by this filter are shown in equations 1 and 2, respectively.

$$y(2n+1) = x(2n+1) - \lfloor \frac{x(2n) + x(2n+2)}{2} \rfloor \quad (1)$$

$$y(2n) = x(2n) + \lfloor \frac{y(2n-1) + y(2n+1) + 2}{4} \rfloor \quad (2)$$

Next, bit-plane coding (BPC) is applied on each codeblock. This is a context modeling operation which involves serially scanning the wavelet coefficients according to a certain scan pattern. Finally the MQ-Coder — the binary arithmetic coder (BAC) of JPEG 2000 — uses this model to generate the encoded bitstream serially. This bitstream includes predefined markers indicating the start of each codeblock (and possibly each tile) as demonstrated in Figure 2. BPC and BAC together make up the Embedded Block Coding with Optimal Truncation (EBCOT) algorithm as shown in Figure 3, which is the most computationally intensive stage of JPEG 2000.

During decompression, these operations are inverted in order to reconstruct the encoded image. As demonstrated in Figure 4, we can break the Tier I decoding process into three main steps:

1) Codeblock Extraction: The first decoding step in a parallel architecture is to parse and extract the codeblock codewords and assign each thread a codeword buffer to process. Using the codeblock markers (and tile markers if applicable), it is quite straightforward to locate the codeword buffer of each codeblock in the bitstream. Since this is a highly serial operation, we do it exclusively on CPU. Nonetheless, it is trivially possible to employ a CPU-level parallelism while parsing multiple encoded image frames.

2) EBCOT Decoding: As tiles and codeblocks are encoded independently from each other, it is possible to decode them independently and in parallel. However due to its strictly sequential nature, EBCOT cannot be parallelized beyond a codeblock-level granularity. Therefore each thread is responsible for processing an entire codeblock in our EBCOT decoding scheme.

3) IDWT: This final step is much more suitable for data-level parallelism compared to the codeblock extraction and EBCOT steps. This is because there are no strictly serial operations in IDWT, except that the 3 levels must be performed one after another. As a result, we have been able to assign each CUDA thread 2 wavelet coefficients in our IDWT implementation, while in EBCOT each thread is responsible for processing all 1024 wavelet coefficients per codeblock. Therefore the IDWT stage takes much less time than EBCOT in our GPU architecture, as we will discuss in Section V.

III. Experimental Setup

Our test data consists of 7 panchromatic satellite image frames with spatial resolutions of 8192×8192 pixels and bit depths of 11-bits/pixel. The average compression ratio is 2.07, which means that the codeword buffers approximately take up 42.5MB on average. On the other hand, each decoded frame takes up 128MB, because each pixel is represented by 16 bits (2 bytes) instead of 11.

Tier I encoder has been implemented only for CPU in order to prepare the encoded images for experimentation. On the other hand, the Tier I decoder has been implemented for both CPU and GPU using C++ along with CUDA.

During experiments, we tested our multi-threaded CPU decoder on processing single image frames. The GPU decoder has been tested on both single frame processing and batch processing modes, because the efficiency of the GPU decoder increases when the image frames are processed in batch.

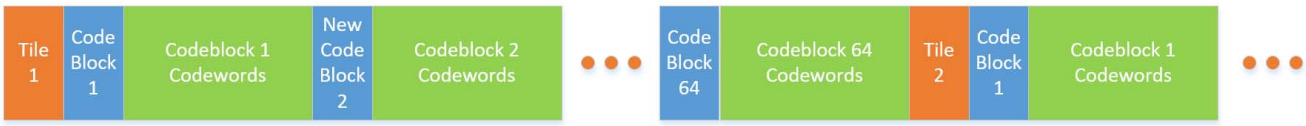

Fig. 2. JPEG 2000 Encoded Bitstream

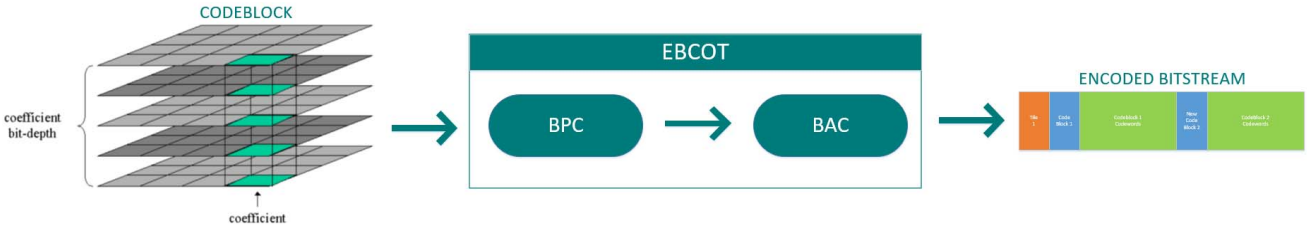

Fig. 3. EBCOT Decoder Flowchart

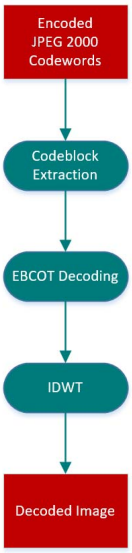

Fig. 4. JPEG 2000 Decoder Flowchart

The kernel execution times and achieved multiprocessor occupancy rates for the GPU decoder have been measured using the NVIDIA Visual Profiler (NVVP) tool [16].

All experiments have been performed on a workstation with the following setup:

- CPU: Intel Xeon E5-2637 v2, 3.5GHz with 128GB RAM
- GPU: NVIDIA GeForce GTX 1060, 6GB, 1280 CUDA Cores, Compute Capability 6.1

IV. GPU Optimization Strategies

In this section, we present the GPU optimization techniques that we have applied for speeding-up the EBCOT decoding stage. We do not analyze the IDWT stage in detail because it has very little impact on the overall performance as it takes much less time compared to EBCOT. Therefore we only share our design choices and performance measurements regarding IDWT in the next subsection.

A. IDWT

As stated earlier, our implementation has been designed for inversing a DWT which is applied first horizontally and then vertically for 3 levels. In Figure 5, the output images and subbands are shown as a result of applying this transformation on a sample image. Hence we launch 6 successive pairs of IDWT and transpose kernels per image frame, as can be seen from the GPU timeline demonstrated in Figure 7.

Among the techniques that we will discuss in the next subsection, we utilized shared memory and read-only cache in IDWT and transpose kernels. As a result, we have measured the total execution time of the 12 successive kernels as 36 miliseconds on average.

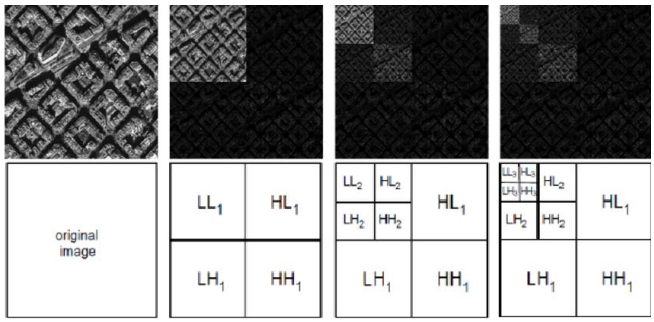

Fig. 5. Output Images and Subbands for a 3-Level DWT

B. EBCOT Decoding

We experimented with a number of GPU optimization techniques in order to speed-up the EBCOT decoding stage. In the following subsections, each of these techniques and their impact on the performance are explained in detail.

1) Registers per Thread: Number of registers used by a single thread can easily become the occupancy limiter in complex algorithms. As stated earlier, EBCOT is a very computationally intensive algorithm which is carried out by a single thread per codeblock in our architecture. As an inevitable consequence, each thread ends up using a large number of registers. In our initial implementation, the number of registers per thread was 62. In this case the number of resident warps per multiprocessor is 32, which corresponds to a theoretical occupancy of 50%. When tested on a single frame, the achieved occupancy was 40.7% and the EBCOT kernel took 4.9 seconds.

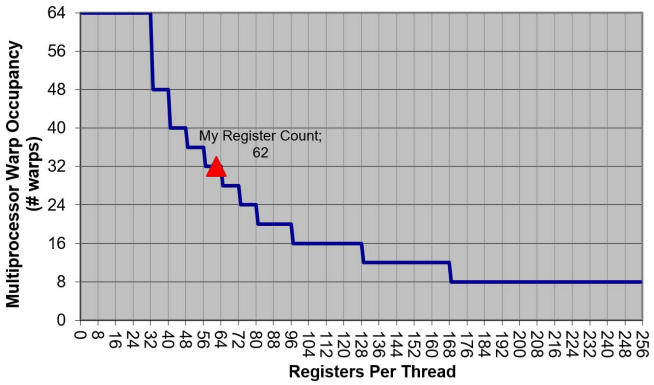

Fig. 6. Register Usage vs. Multiprocessor Occupancy

CUDA Occupancy Calculator [17] suggests that it is possible to increase the number of resident warps per multiprocessor by decreasing the register count to 56, as visualized in Figure 6. We achieved this after some optimizations that involve removing some redundant class members and constants. Consequently, the theoretical and achieved occupancy rates increased to 56.2% and 46.4%, respectively. Also the kernel execution time decreased from 4.9 seconds to 4.7 seconds.

Fig. 7. GPU Timeline of the IDWT Stage

2) Page-Locked Host Memory: Memory allocations on the host (CPU) side are pageable by default. By registering a host buffer as page-locked memory, the operating system guarantees that this buffer will always be in physical memory until it is unregistered. This can potentially increase the throughput of the memory transfers between host and device. In addition, the memory transfers between page-locked host memory and device memory can be performed concurrently with kernel execution, as we will discuss in one of the following subsections.

Two kinds of memory transfers take place while decompressing an image frame:

- The encoded codeword buffer (42.5MB on average) is copied from host to device before launching the EBCOT kernel.
- The decoded image buffer (128MB) is copied from device to host after the IDWT stage has been completed.

In our experiments, we initially used default memory allocations for both host buffers. In this case, copying the encoded codewords from host to device took 9.5 miliseconds and copying the decoded buffer from device to host took 25.5 miliseconds. Then we registered both buffers as page-locked host memories and repeated our measurements. As a result, copy times of the encoded and decoded codewords dropped to 3.7 miliseconds and 10.3 miliseconds, respectively. Therefore we have approximately achieved a 2.5× speed-up on device-host memory transfers.

3) Read-Only Cache: The read-only data cache was introduced with CUDA compute capability 3.5 in order to speed up reads from device memory. As a prerequisite, the cached data has to be read-only for the entire lifetime of the kernel [18]. In our case, the only buffer that the EBCOT kernel does not modify is the encoded codeword buffer.

In our experiments, the EBCOT kernel took 4.7 seconds in single frame decompression mode when we did not use the read-only cache at all. When the read-only cache is utilized for the encoded codewords, this duration decreased to 4.5 seconds.

4) Asynchronous Kernel Launch and Memory Transfers: We have only covered the single frame decompression mode so far. In this subsection, we compare this mode with the batch mode, where we utilize CUDA streams to launch kernels and perform memory transfers asynchronously.

In single frame decompression mode, we sequentially launch a new kernel for each frame if there are multiple frames to process. EBCOT decoding of 7 frames in this mode takes 33.5 seconds including kernel launches and memory transfers. On the other hand, this operation takes 32.4 seconds in batch mode where the kernel launches and memory transfers overlap with each other as demonstrated in Figure 8.

5) State Plane Storage and Shared Memory: Shared memory is a faster yet smaller memory area compared to global memory, which can be accessed and shared by all threads in a thread block. In this subsection, we present the performance impacts of utilizing global memory versus shared memory for the 4 state planes $(\chi, \sigma, \sigma', \eta)$ that are used in EBCOT decoding stage. These planes consist of 1024 binary bits (flags) each, since the codeblock dimensions are 32×32 pixels.

In a naive implementation, these buffers can be stored in boolean arrays in the C++ language. However boolean arrays take up 8 bits per element in device memory, which results in 7 wasted bits per flag. This is especially problematic for shared memory utilization, because it corresponds to a shared memory demand of 256KB per thread block (64 threads). Obviously it is not possible to fit this volume inside shared memory, since the maximum amount of shared memory available in most devices is 48KB per thread block, including GTX 1060.

Although we had initially stored the state planes in boolean arrays, we later converted them to byte arrays in order to experiment with shared memory utilization. Since this modification eliminated the storage inefficiency, the shared memory demand reduced from 256KB to 32KB per thread block. This modification introduced a significant amount of speed-up on its own. As a result, the EBCOT kernel duration for decoding a single frame decreased from 4.5 seconds to 4.2 seconds; and the total time to decode 7 frames in batch decreased from 32.4 seconds to 29.4 seconds.

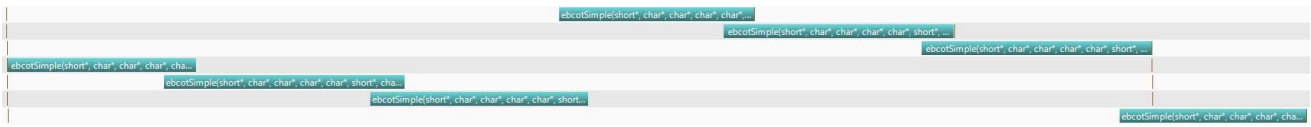

Fig. 8. Asynchronous Kernel Launch and Memory Transfers with CUDA Streams

The next step in our experiments was to store the state planes inside shared memory. This, on the other hand, resulted in a significant performance loss as the utilization of 32KB of shared memory per thread block drastically decreased the achieved multiprocessor occupancy from 56.2% to 9.4%. The decrease in the theoretical occupancy can also be observed by looking at Figure 9, which shows the visualization obtained from the CUDA Occupancy Calculator. Due to this loss in occupancy, the EBCOT kernel duration for decoding a single frame increased from 4.2 seconds to 8.7 seconds, and the total time to decode 7 frames in batch increased from 29.4 seconds to 60 seconds.

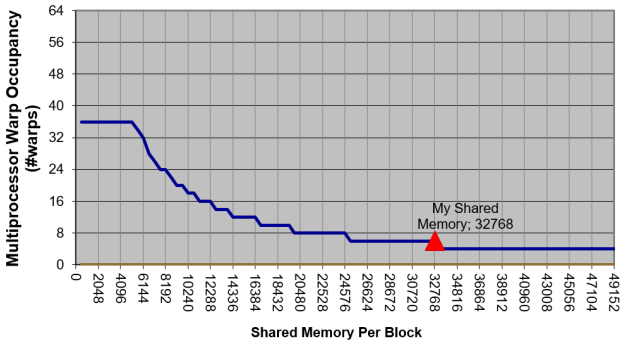

Fig. 9. Shared Memory Usage vs. Multiprocessor Occupancy

6) Dynamic Parallelism: Dynamic parallelism is a CUDA feature that makes it possible to launch child kernels within parent kernels. It is available on devices of compute capability 3.5 or higher. We have experimented with this capability in our implementation of JPEG 2000 decoder.

After the EBCOT stage, the decoded wavelet coefficients are converted from signed-magnitude representation to 2's complement representation and copied to their corresponding positions in the image buffer to be reconstructed. Our initial approach was to do this operation at the end of the EBCOT kernel. In this case, 1024 coefficients are mapped serially by a single thread which is responsible for processing the corresponding codeblock.

We have experimented with dynamic parallelism to see if it can speed-up this operation. Firstly we have removed this serial operation from the EBCOT kernel and replaced it with a dynamic kernel launch. Thus, each of the 1024 coefficients in the codeblock are handled by a separate thread in parallel. However, the experiments showed that this modification was not effective, as it reduced the achieved multiprocessor occupancy from 46.4% to 38.5%. As a result, the EBCOT kernel execution time for decoding a single frame increased from 4.2 seconds to 4.6 seconds and the total time to decode 7 image frames in batch increased from 29.4 seconds to 40.2 seconds. As the operations in the child kernels involved low arithmetic complexity, overhead of launching child kernels became more dominant and this effectively decreased the overall occupancy and resulted in poorer performance.

V. Performance Evaluation

In this section, we evaluate the performances of the optimal GPU and CPU decoders and compare them with each other.

A. GPU Performance

After having experimented with all the GPU optimization techniques we mentioned in Section IV, we achieved the best performance in the following configuration:

- The number of registers per thread is limited to 56.
- Encoded codeword buffer and decoded frame buffer are both registered as page-locked host memory.
- Read-only cache is utilized for accessing the encoded codeword buffer from the EBCOT kernel,
- CUDA streams are utilized for asynchronous kernel launches and memory transfers when processing multiple frames in batch.
- State planes $(\chi, \sigma, \sigma', \eta)$ are stored in byte arrays instead of boolean arrays.
- No shared memory is utilized by the EBCOT kernel.
- Wavelet coefficients are mapped into the decoded frame buffer serially in the end of the EBCOT kernel as opposed to employing dynamic parallelism.

With this configuration, we have measured an EBCOT kernel execution time of 4.2 seconds when decoding a single frame. As for processing 7 frames in batch, we have measured a total execution time of 29.4 seconds including memory transfers, EBCOT decoding and IDWT.

B. CPU Performance

We have utilized the OpenMP library [19] for CPU parallelism, where all 1024 tiles in an image frame are processed independently from each other and in parallel. We have measured an average EBCOT decoding time of 7.2 seconds and an average IDWT time of 507 miliseconds while decoding a single frame. Codeword extraction takes 43 miliseconds for both CPU and GPU, as it is done using CPU in both implementations.

C. Comparison

Table I indicates the average execution times of different decoding stages for each decoder type. The asynchronous batch GPU mode is slightly faster than the synchronous mode considering the average decoding time of an image frame. On the other hand, both GPU modes are more than 1.8× faster than the multi-threaded CPU decoder.

TABLE I
Performance Comparison

Decoder	Codeblock Extraction	EBCOT Decoding	IDWT	Device Memory Transfers	Total
CPU	43 ms	7220 ms	507 ms	N/A	7770 ms
GPU-Sync	43 ms	4180 ms	36 ms	13 ms	4272 ms
GPU-Async	43 ms	4140 ms	36 ms	13 ms	4232 ms

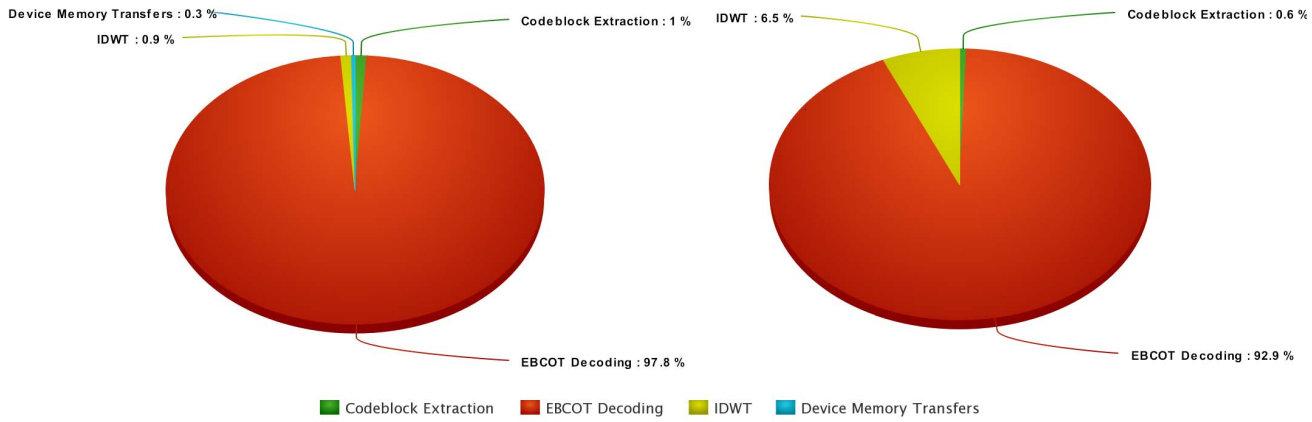

Fig. 10. Computational Intensities of Different Decoding Stages. (a) GPU-Async (b) CPU

VI. Conclusion and Future Work

It is clear from our studies that EBCOT decoding is by far the most computationally intensive stage in JPEG 2000 decompression, regardless of whether it is performed using CPU or GPU. However it becomes more of a bottleneck when it comes to GPU architectures, as this stage cannot be optimized as efficiently as the IDWT stage which is very suitable for the SIMD model.

This can be clearly seen from Figure 10, where the execution times of these two stages are compared percentage-wise for our CPU and GPU decoders. While the GPU decoder is 14× faster in the IDWT stage, it is only 1.7× faster in the EBCOT decoding stage compared to the CPU decoder. As a result, a maximum overall speed-up of 1.8× has been achieved with the GPU decoder after numerous optimizations that we have detailed in Section IV.

In our future studies, we aim to enrich our experiments by benchmarking different types of processors. Our another goal is to implement a CPU-GPU heterogeneous architecture and divide the processing tasks in order to achieve the optimal utilization of computing resources. Finally we aim to replace our batch decompression mode with a stream decompression mode, where the downloaded satellite images can be pipelined in real-time so as to increase the efficiency of satellite imaging missions.

References

[1] Chiang, J. S., Lin, Y. S., Hsieh, C. Y. (2002). Efficient pass-parallel architecture for EBCOT in JPEG2000. In Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on (Vol. 1, pp. I-I). IEEE.

[2] Andra, K., Chakrabarti, C., Acharya, T. (2003). A high-performance JPEG2000 architecture. IEEE Transactions on Circuits and Systems for video technology, 13(3), 209-218.

[3] Fang, H. C., Chang, Y. W., Wang, T. C., Lian, C. J., Chen, L. G. (2005). Parallel embedded block coding architecture for JPEG 2000. IEEE Transactions on Circuits and Systems for Video Technology, 15(9), 1086-1097.

[4] Sarawadekar, K., Banerjee, S. (2011). An efficient pass-parallel architecture for embedded block coder in JPEG 2000. IEEE Transactions on Circuits and Systems for Video Technology, 21(6), 825-836.

[5] Matela, J., Rusnak, V., Holub, P. (2011, March). Efficient JPEG2000 EBCOT context modeling for massively parallel architectures. In Data Compression Conference (DCC), 2011 (pp. 423-432). IEEE.

[6] Matela, J., Šrom, M., Holub, P. (2011, October). Low GPU occupancy approach to fast arithmetic coding in JPEG2000. In International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (pp. 136-145). Springer, Berlin, Heidelberg.

[7] Lee, J. W., Kim, B., Yoon, K. S. (2014, February). CUDA-based JPEG2000 encoding scheme. In Advanced Communication Technology (ICACT), 2014 16th International Conference on (pp. 671-674). IEEE.

[8] Le, R., Bahar, I. R., Mundy, J. L. (2011, June). A novel parallel Tier-1 coder for JPEG2000 using GPUs. In Application Specific Processors (SASP), 2011 IEEE 9th Symposium on (pp. 129-136). IEEE.

[9] Auli-Llinas, F., Enfedaque, P., Moure, J. C., Sanchez, V. (2016). Bitplane image coding with parallel coefficient processing. IEEE Transactions on Image Processing, 25(1), 209-219.

[10] Enfedaque, P., Auli-Llinas, F., Moure, J. C. (2017). GPU Implementation of Bitplane Coding with Parallel Coefficient Processing for High Performance Image Compression. IEEE Transactions on Parallel and Distributed Systems, 28(8), 2272-2284.

[11] Ciznicki, M., Kurowski, K., Plaza, A. J. (2012). Graphics processing unit implementation of JPEG2000 for hyperspectral image compression. Journal of Applied Remote Sensing, 6(1), 061507.

[12] Ciżnicki, M., Kierzynka, M., Kopta, P., Kurowski, K., Gepner, P. (2014). Benchmarking JPEG 2000 implementations on modern CPU and GPU architectures. Journal of Computational Science, 5(2), 90-98.

[13] Wu, X., Li, Y., Liu, K., Wang, K., Wang, L. (2014). Massive parallel implementation of JPEG2000 decoding algorithm with multi-GPUs. Satellite Data Compression, Communications, and Processing X, 9124, 91240S.

[14] Le, R., Mundy, J. L., Bahar, R. I. (2012, July). High performance parallel JPEG2000 streaming decoder using GPGPU-CPU heterogeneous system. In Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on (pp. 16-23). IEEE.

[15] Skodras, A., Christopoulos, C., Ebrahimi, T. (2001). The JPEG 2000 still image compression standard. IEEE Signal processing magazine, 18(5), 36-58.

[16] https://developer.nvidia.com/nvidia-visual-profiler

[17] https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[18] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[19] https://www.openmp.org/