Sample-Parallel Execution of EBCOT in Fast Mode

Volker Bruns, Miguel Á. Martínez-del-Amor
Fraunhofer Institute for Integrated Circuits
Erlangen, Germany
{volker.bruns, miguel.martinez}@iis.fraunhofer.de

***Abstract*— JPEG 2000's most computationally expensive building block is the *Embedded Block Coder with Optimized Truncation* (EBCOT). This paper evaluates how encoders targeting a parallel architecture such as a GPU can increase their throughput in use cases where very high data rates are used. The compression efficiency in the less significant bit-planes is then often poor and it is beneficial to enable the *Selective Arithmetic Coding Bypass* style (fast mode) in order to trade a small loss in compression efficiency for a reduction of the computational complexity. More importantly, this style exposes a more finely grained parallelism that can be exploited to execute the raw coding passes, including bit-stuffing, in a sample-parallel fashion. For a latency- or memory critical application that encodes one frame at a time, EBCOT's tier-1 is sped up between 1.1x and 2.4x compared to an optimized GPU-based implementation. When a low GPU occupancy has already been addressed by encoding multiple frames in parallel, the throughput can still be improved by 5% for high-entropy images and 27% for low-entropy images. Best results are obtained when enabling the fast mode after the fourth significant bit-plane. For most of the test images the compression rate is within 1% of the original.**

I. Introduction

JPEG 2000 is an immensely flexible still image compression standard released jointly by ITU and ISO [1]. Perhaps the most dominant field of use is in media entertainment where JPEG 2000 was chosen by the *Society of Motion Picture Technology Experts* (SMPTE) to be employed in both the *Digital Cinema Package* (DCP) format [2] and the more recent *Interoperable Master Format* (IMF) [3]. In this context, JPEG 2000 is employed to compress resolutions between 2K and 4K with very high data rates (DCP: up to 250 Mbit/s, IMF: up to 800 Mbit/s). Due to its high complexity, JPEG 2000 compression is a very computationally expensive task and doing so fast is a challenge. The color- and wavelet-transforms as well as the quantization are perfectly suited to the parallel architecture of a GPU. However, the situation is quite different for the entropy coder, named *Embedded Block Coder with Optimized Truncation* (EBCOT) [4], which operates on blocks of quantized wavelet coefficients and processes them bit-plane by bit-plane.

Keeping in mind the high bit rates employed in DCPs or IMF packages, the reality is that only frames comprising a lot of details actually reach the maximum data rate limit. Looking at the effectiveness of EBCOT, especially in the less significant bit-planes, the cost of arithmetically coding every single bit-plane in terms of increased run-time will in many use-cases outweigh the benefit, which is a marginally higher compression ratio. The JPEG 2000 standard addresses exactly this issue by defining the *Selective Arithmetic Coding Bypass* style, sometimes also referred to as the *Fast Mode* or *Lazy Mode* [4].

Different parties, including the authors of this paper, have set out to outsource many or all processing steps to the GPU using programming interfaces such as OpenCL or CUDA [5]-[7]. With their focus on DCI or IMF profiles, these works don't support the fast mode, though. The authors of [8] propose a drop-in replacement for EBCOT that exposes more parallelism, but sacrifices context adaptiveness as well as compliance with the standard. To the same end, but without parallel architectures in mind, [9] proposes an ultrafast mode, where EBCOT is replaced with a fixed codebook Huffman coder preceded by an optional prediction step.

In contrast, this work will stay within the bounds of the standard and evaluate the potential of the selective arithmetic coding bypass style. Even if its use is prohibited in the DCP or IMF profiles, a faster codec would still be useful in scenarios where a DCP or IMF package will be created downstream, especially when the codestreams are partially identical and can be efficiently transcoded. The research question posed here is: can an implementation of EBCOT that targets parallel architectures such as GPUs benefit from the more finely grained parallelism exposed by this coding style and thereby increase its throughput?

Chapter 2 will present a brief overview of JPEG 2000 with focus on EBCOT and its fast mode. In chapter 3 a sample-parallel algorithm will be proposed. A large part will focus on how to execute the required bit-stuffing in parallel. Chapter 4 will discuss implementation strategies and experimental results. Finally, the findings are summarized.

II. The JPEG 2000 standard

A. Encoder Overview

The first step is an RGB-to-YUV color transform that decorrelates colors from brightness levels. In lossy mode, it is denoted as *Irreversible Color Transform* (ICT) whereas in the lossless mode an integer-approximation named *Reversible Color Transform* (RCT) is used. Due to its pixel-parallel nature, this step is very suited for execution on a GPU. Each thread can transform one or more pixels.

Next, the individual color channels are each decomposed with a *Discrete Wavelet Transform* (DWT) in a *Mallat* pyramid fashion. Subsequently, all coefficients are quantized, with a step-size that may vary for each subband. Again, both steps are sample- parallel and well suited to be executed by a GPU.

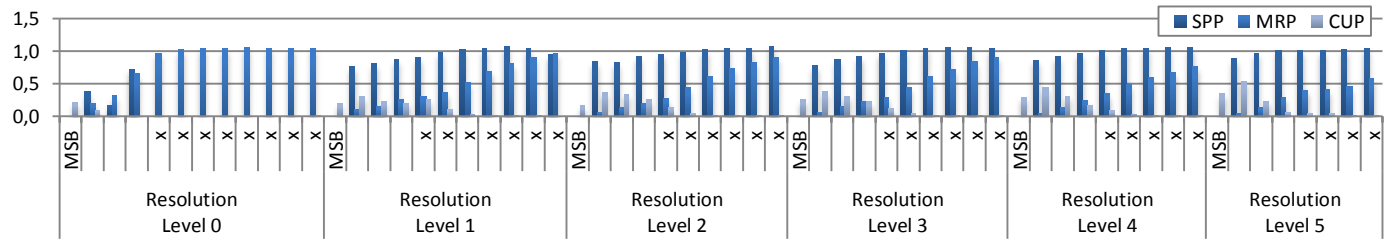

Figure 1. Compression efficiency for a frame from the DCI SteM sequence ("MM_2K_XYZ_01571", Y channel, DCI 2K profile). Values below 1 indicate compression, values above 1 expansion. If fast mode starting after fourth bit-plane were enabled, symbols from SPP and MRP in bit-planes marked with an "x" would bypass the MQ-encoder, reaching a value of 1.

B. EBCOT

Each quantized subband is split into code-blocks that are independently entropy-coded. Both the DCP and IMF specifications define a code-block size of 32 by 32 pixels. A smaller code-block size would be preferable in order to increase the available parallelism, but this came at the cost of a reduced compression efficiency, since the adaptive coder then had less opportunity to adjust to the signal. The impact of code-block sizes is further discussed in [8].

EBCOT comprises two tiers. In the first one, a code-block is compressed into an embedded bitstream. While the bitstream is constructed, the coder creates truncation points and records them on a rate-distortion plot. In case the total accumulated size of all code-blocks' bit streams exceeds the given codestream size budget, the post-compression rate distortion optimization (PCRD-opt.), carried out in EBCOT's second tier, can make an informed decision on which code-blocks' bit streams to truncate and where.

A code-block's bit stream is created by traversing from the most significant magnitude bit-plane (planes with only zero bits are skipped) to the least significant bit-plane. Each plane is scanned in three passes in a column-wise order by columns of four coefficients (stripe column). The first pass, denoted *Significance Propagation Pass* (SPP), codes symbols from those samples that are not yet significant (no one bit in a previous bit-plane), but are likely to become significant in this plane based on their neighborhood. The next pass, denoted *Magnitude Refinement Pass* (MRP), codes symbols from all those samples that have already turned significant. Finally, the last pass, denoted *Clean-up Pass* (CUP), codes all remaining symbols. A sample's sign-bit is coded immediately following its first significant bit. The spatial correlation among samples is modelled by assigning each coded symbol to one of 19 contexts, depending on the significance state of the sample itself and its immediate neighborhood.

EBCOT employs a context-adaptive binary arithmetic coder (AC), named *MQ-coder*. All symbols of a code-block are sequentially fed to the AC in combination with a context. The AC adjusts the probability whether the next symbol is expected to be a one or zero independently for each context. Within a bit-plane, context modelling and arithmetic coding can be regarded as separate processes in the encoder. While arithmetic coding does not present any parallelism within a code-block, contexts can be modelled in a sample-parallel fashion as presented in [6].

C. Selective Arithmetic Coding Bypass Style

The compression efficiency of any given code-pass can be defined as the inverse ratio of the number of symbols that are fed into the MQ-encoder to the number of bits that are emitted by the coder. Ratios smaller than one indicate that the MQ-encoder has compressed the signal while ratios larger than one indicate an expansion. Especially for the SP- and MR-passes in the less significant bit-planes of high-energy bands the ratio is often close to or even above one (see Figure 1). The reason is that the Laplacian distribution assumed by the coder is not actually present anymore, but instead the signal is distributed uniformly [10]. To address this issue, JPEG 2000 defines the *Selective Arithmetic Coding Bypass* style. When enabled, symbols from the SP- and MR-passes starting from a code-block's fifth significant bit-plane are directly appended to the bitstream, bypassing the MQ-encoder. Symbols from CU-passes must still be fed into the MQ-encoder as usual. A bitstream needs to be terminated whenever switching between non-bypassed and bypassed mode so that in the end it consists of multiple segments. This coding style is signaled globally for either the entire image or image component – it cannot be set on a code-block by code-block basis. Part 2 of the standard enhances the mode by enabling the user to select whether the AC in the SP- and MR-passes should already be bypassed as early as in the second, third or fourth bit-plane.

III. Sample-Parallel Algorithm

A. Parallel Context Modelling and Raw Coding

A high-level presentation of a parallel EBCOT's tier 1 in fast mode is presented in Algorithm 1**.** It comprises three stages: context modelling (1), the raw-coded SPP and MRP (2) and the MQ-encoded CUP (3). The first two stages can be carried out by a kernel where each thread is in charge of a stripe column. Thus, a 32 by 8 thread-block is mapped to a 32 by 32 code-block. The MQ-encoded CUP does not allow to process samples in parallel, so a one-to-one thread-to-code-block mapping is an obvious choice. In Algorithm 1 all three stages are combined into a single kernel, but this is an implementation choice. Different strategies are discussed later on.

The context-modelling related routines are summarized in the algorithm's first line. A detailed description of this approach is laid out in [6]. In short, threads execute a collaborative pre-processing stage in order to collect information that will allow them to perform the passes in

Algorithm 1 - EBCOT tier-1 with bypassed SPP and MRP

one thread per stripe-column

```
1.  contextModelling() // SPP+MRP+CUP
2.
3.  numLocalSyms[] = countLocalSymbols()
4.  shared sharedMem[] // byte buffer
5.
6.  // ---- SPP ----
7.  startIdx = exclScan(numLocalSyms[SPP])
8.  putBitsAtomic(sharedMem, startIdx,
9.    getLocalSppSyms(), numLocalSyms[SPP])
10.
11. // ---- MRP ----
12. startIdx = exclScan(numLocalSyms[MRP])
13. putBitsAtomic(sharedMem, startIdx,
14.   getLocalMrpSyms(), numLocalSyms[MRP])
15.
16. // ---- SPP + MRP bitstuff & copy ----
17. totalNumSyms = scanTotalNumSyms(numLocalSyms)
18. numBytes = bitstuff(sharedMem,
19.   totalNumSyms[SPP] + totalNumSyms[MRP])
20. if (tid == 0) // is block leader?
21.  terminateRawSegm(sharedMem)
22. if (id < numBytes)
23.   bitstream[tid] = sharedMem[tid]
24.
25. // ---- CUP ----
26. startIdx = exclScan(numLocalSyms[CUP])
27. putBits(sharedMem, startIdx,
28.   getLocalCupCtxds(), numLocalSyms[CUP])
29.
30. shared mqenc;
31. if (tid == 0) { // is block leader?
32.   startMqSegm(mqenc)
33.   for (i = 0; i < numLocalSyms[CUP]; i++)
34.     mqEncode(mqenc, bitstream, sharedMem[i])
35.   terminateMqSegm(mqenc, bitstream);
36. }
```

parallel for each stripe column. After this first stage, each thread computes all the context-decision pairs generated by the three passes (SPP, MRP and CUP) for its stripe. These symbols are stored into registers to be used later.

Following the context-modelling stage, the collected symbols need to be fed into the MQ- or raw-coder in the proper order: first symbols from the SPP, then those from the MRP, and finally all remaining symbols from the CUP. Symbols from the first two passes bypass the MQ-coder so that the threads can collaboratively create the bitstream segment in parallel. Each thread knows how many symbols it must code for the current pass (line 3). For the SPP, it can be anywhere from zero (none of the four positions belong to the SPP) to eight symbols (all four positions are zero-coded and require sign-coding); for the MRP between zero and four symbols (each of the four positions' magnitude is refined); and for the CUP the worst case scenario is that ten context-decision pairs get produced for a single-stripe column (run-mode is entered, but gets immediately interrupted and all remaining symbols turn their sample significant so that the sign bits have to be coded as well). The offset from the start of the bitstream segment to which a thread needs to put the current pass's symbols is defined by the sum of all previous threads' symbols. It can be efficiently calculated by executing a block-wise exclusive prefix scan (lines 7, 12) [11][12]. Next, each thread can copy its symbols to the bitstream. Since multiple threads will have to write to the same byte, they must use an atomic bitwise *or* operation. An implementation is described in detail

Algorithm 2 - Parallel Bit-stuffing

one thread per byte

```
1.  insert = false
2.  numPrevInserts = 0
3.  forever {
4.    lftFF = read8Bits(bitstream, (tid-1)*8-
5.      numPrevInserts) == 0xFF
6.    headOfSegm = !lftFF
7.    n = segmExclScan(lftFF, headOfSegm)
8.    insertNew = lftFF && isEven(n)
9.    myVote = insertNew != insert
10.   anyChanges = blockVote_any(myVote)
11.   if (!anyChanges)
12.     break
13.   numPrevInserts = exclScan(insertNew?1:0)
14. }
15. val = read8Bits(bitstream, tid*8-numPrevInserts)
16. if (insert)
17.   val = (val>>1)&0x8F // insert 0-bit in MSB
18. bitstream[tid] = val
```

in [13]. The proposed algorithm first creates the bitstream in an intermediate buffer in shared memory. This is advantageous because bit-stuffing is separated into a subsequent separate function that will have to read and write the bitstream at least one more time. A side-effect is that the implementation benefits from the native support for shared atomics introduced in Nvidia's Maxwell architecture [14].

Next, the same steps need to be repeated for the SPP so that afterwards the entire raw bitstream segment has been written to the intermediate buffer in shared memory.

B. Parallel Bit-Stuffing

The JPEG 2000 standard requires that whenever there is a 0xFF byte in the bit stream, the next byte must include an extra zero-bit stuffed into the MSB. The original motivation behind bit stuffing is to avoid a carry-over in the MQ-encoder, which would in theory render bit stuffing unnecessary for raw bit stream segments. However, the fact that after bit-stuffing the byte following an 0xFF is guaranteed to be in the range of 0x00 to 0x7F is additionally exploited to introduce marker codes, consisting of a 0xFF byte followed by a byte in the range of 0x90 to 0xFF. For this reason, bit-stuffing applies to raw bit stream segments just the same. Bit-stuffing is trivial when building the bit stream serially. However, when done in parallel, it requires some effort. Algorithm 2 provides an overview of the basic idea and figure 2 visualizes an example.

Each thread is in charge of writing one byte of the bit-stuffed output bit stream. In case the byte to the left is 0xFF and the byte before that is not 0xFF, a thread needs to insert a zero bit in the MSB (see thread 3 in figure 2). To ensure that bits are inserted only after every other consecutive 0xFF byte, a segmented exclusive scan (see [12]) is used, where a segment represents back to back 0xFF bytes. The head flag, which indicates that a new segment must be started, is set whenever the previous byte is not 0xFF. Thread 4 in figure 2 obtains a scan result of $n=1$, for example, because the previous $n+1=2$ bytes consist entirely of one-bits. A thread needs to insert a zero bit only if the segmented scan result is an even number. A consequence of inserting a bit is that all subsequent threads then need to reevaluate their decision: it might be that they, too, inserted a zero bit, but that this isn't actually necessary

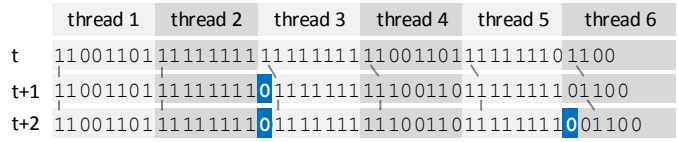

Figure 2. Parallel bit-stuffing example. Blue bits are inserted.

anymore given that a bit was inserted earlier on in the bit-stream and all other bits were right-shifted. Oppositely, it might be that they did not insert a zero bit, but now actually have to do it (see *thread 6* at *t+1*). And finally, it might be that their decision still stands. For this reason, the parallel bit-stuffing routine might take multiple iterations to finish. After the initial decision, a block vote is conducted. In the first iteration, each thread votes whether or not they think they need to insert a zero bit. If none of them vote positively, no bit-stuffing is necessary anywhere and the routine can finish. If at least one bit-stuff is required, the threads continue and re-evaluate their decisions, this time taking into account the previous threads' actions (after t+1, threads 4-6 know that one bit was inserted in an earlier byte). By running an exclusive scan over all threads' bit-stuffing decisions, each thread can determine the amount of bit-stuffs in all previous bytes in the bitstream. Considering the bit shifts required by the zero-bit inserts, they can then figure out which eight bits would eventually fall into the byte slot to their immediate left and again check, based on the result of a segmented scan, if they are required to insert a zero-bit in their byte's MSB. Once again each thread casts a vote, this time signaling if their decision of whether they need to include a zero-bit or not has changed compared to the last vote. The entire process is repeated for as long as any of the threads had to correct their previous vote. In the end, each thread reads the seven or eight bits from the unstuffed bit-stream that will fall into their output byte slot, insert a zero-bit in the MSB or not, depending on their last vote, and finally overwrite the output byte in the bit-stream.

Since in CUDA, registers are 32 bits wide, an optimization to the algorithm is for each thread to be in charge of writing four output bytes as opposed to only one byte. Special care has to be taken when participating in the scan or voting operations. Depending on the type of operation, a thread will have to recursively apply it to its four bytes and then contribute the overall result to the block-wide operation. The experimental results reported next include this optimization.

IV. Experimental Results

Three different strategies for EBCOT's first tier in fast mode have been evaluated. They differ in how the routines for (1) context modelling, (2) the two raw passes and (3) the MQ-encoded CUP are organized into separate kernels. The starting point was the optimized, non-bypassed EBCOT implementation used in [4], which launches separate context-modeling- (CM) and MQ-encoding-kernels (MQ) for every bit-plane so that all code-blocks are coded in parallel, but the processing of bit-planes is synchronized. Each code-block starts processing at its first significant magnitude bit-plane. Launching kernels individually for each bit-plane is beneficial for two reasons. First, the memory required for storing the context-decision pairs collected by the CM-kernel and input to

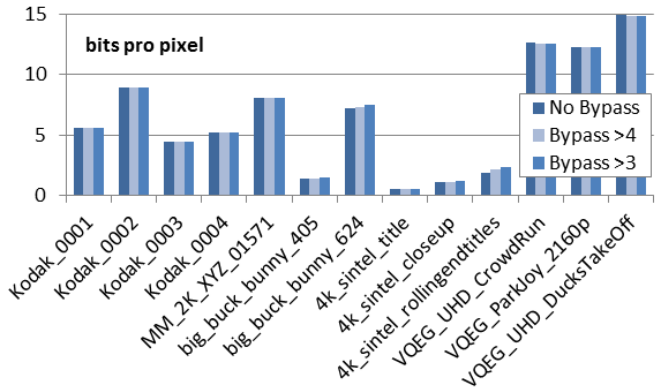

Figure 3. Impact of fast mode (kicking in after 3 or 4 magnitude bit planes) on compression efficiency

the MQ-kernel only has to be large enough to fit the worst-case amount of pairs from a single bit-plane. Second, the MQ-kernel for bit-plane N can be overlapped with the CM-kernel for bit-plane N-1.

Three-Kernel-Strategy – The first strategy splits the routines into three kernels: one for CM, one for the two raw passes and one for the CUP. The CUP-kernel cannot start before the raw-passes-kernel has finished, because it needs to know at which exact byte offsets to continue the code-blocks' bit streams. However, the next bit-plane's CM-kernel can be overlapped with the current bit-plane's CUP-kernel.

Two-Kernel-Strategy - A drawback of the first approach is that the CM-kernel needs to store the modeled context-decision-pairs to global memory in order to make them available to the coding kernels. Since the CM- and raw-passes-kernels both use an identical thread layout, the *Two-Kernel-Strategy* fuses them together into a single kernel. On the downside, the CUP kernel can then no longer be overlapped.

One-Kernel-Strategy - Still, the CM-kernel has to write the context-decision-pairs for the CUP to global memory. The *One-Kernel-Strategy* fuses all routines into a single kernel. A positive side-effect is that now the kernel can just as well loop over all bypassed bit-planes internally. The drawback is that most threads stay idle while the block's first thread feeds the CUP-context-decision pairs to the MQ-encoder. However, the regular MQ-kernel with one thread per code-block naturally suffers from high thread-divergence, which alleviates this drawback to some degree.

Experiments show that the *One-Kernel-Strategy* outperforms the previous two when encoding single images, yielding an up to 2x speed-up compared to EBCOT without fast mode (Figure 4). However, the speed-up is not equally high for all test images. By enabling the fast mode, only the execution of the less significant bit-planes (LSBs) is sped up. The number of code-blocks that still have outstanding significant bit-planes to be processed decreases with every bit-plane. By the time the coder gets to the bypassed LSBs, the number of remaining code-blocks has become so low, that a high-end GPU is severely under-occupied. At this point the duration for processing the remaining LSBs without fast mode is almost constant, irrespective of the test image's resolution and entropy. The execution time for the MQ-coded bit-planes,

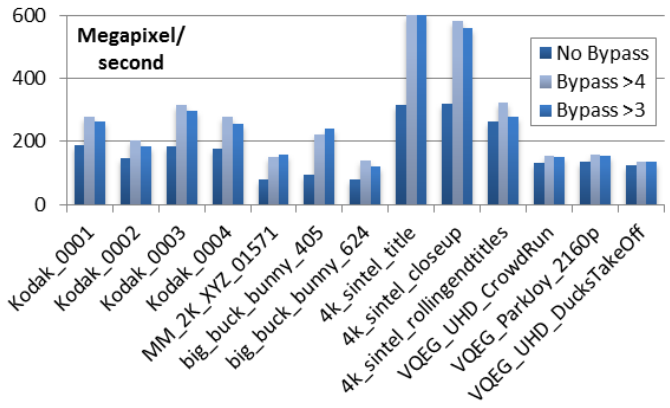

Figure 2. Performance increase for EBCOT when encoding single images (12 bit 4:4:4, no max. bit-rate).

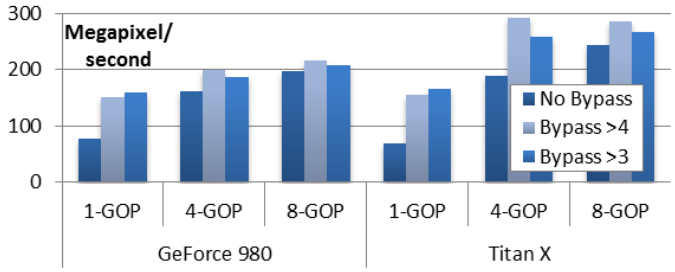

Figure 3. Performance increase for EBCOT when encoding frames in parallel from the (12 bit, 4:4:4, no max. bit-rate, 2.7 bit per sample) DCI SteM sequence ("MM_2K_XYZ_01xxx").

though, depends much more heavily on the resolution and entropy. Thus, the relative time spent in the bypassed bit-planes is lowest for the high-resolution, high-entropy images (VQEG, Kodak). The relative speed-up for only the bypassed bit-planes is highest for those images where the GPU was most severely under-occupied.

An alternative approach to increase the amount of parallel work units is to process code-blocks from multiple images in parallel. This concept is denoted as group-of-pictures (GOP)-coding here. Figure 5 compares the throughput for different GOP-sizes. The benefit of the fast mode decreases as the GOP-size, and with it the number of parallelizable code-blocks, increases. However, even at a GOP-size of eight, the parallel fast mode implementation yields an increase of throughput of 5-10% for the high-entropy UHD sequences, 10-15% for the 2K sequences and up to 20% for the low-entropy 4K sequences. Since now the overall GPU occupation is higher, the *three-kernel-strategy* performs best: the time savings from overlapping CM and CUP outweigh the cost of additional memory accesses.

The impact of the bit-stuffing routine was only measureable for the high-entropy 4K images, where it accounted for less than 3% of the total EBCOT tier-1 runtime. The number of inserted zero bits ranges between 30 ppm for the high-entropy *Kodak_0003* test image and around 300 ppm for the low-entropy *4k_sintel_title* image. The average number of loop iterations required for the proposed parallel bit-stuffing algorithm lies between two and three for all test images.

V. Conclusion

The research question, whether JPEG 2000's selective arithmetic coding bypass mode ("fast mode") is beneficial to parallel GPGPU implementations can be answered positively. Algorithms for a sample-parallel execution of the raw-coded passes and subsequent bit-stuffing as well as a strategy on how to best organize them into GPU kernels were proposed. The increased parallelism speeds up EBCOT tier-1 by a factor of two for low-entropy images where only very few code-blocks have more than four significant bit-planes. For high-entropy 4K images a more moderate speed-up of between 10% and 30% was measured. The compressed size is within 1% of that of the unmodified EBCOT, with the exception of those test images with text, especially "Sintel rolling titles" (14% larger) and "Sintel title" (6% larger), where the SPP compresses well even in the high-frequency bands and at lower bit-planes (Figure 3).

If the increased latency and memory consumption can be tolerated, multiple frames should be encoded in parallel in order to increase the GPU-occupancy. In this case, the benefit from the fast mode is diminished significantly to speed-ups between 5% for high-entropy images and 27% for low-entropy images.

References

[1] ISO/IEC 15444-1. JPEG2000 image coding system|part 1: Core coding system

[2] A. Bilgin, M.W. Marcelling, JPEG2000 for Digital Cinema, IEEE Int. Symp. on Circuits and Systems, pp. 3878-3881, May 2006

[3] SMPTE ST 2067-21:2014. Interoperable Master Format – Application #2 Extended

[4] D. S. Taubman, M. W. Marcellin, JPEG2000: Image Compression Fundamentals, Standards, and Practice. Springer, 2002

[5] Fraunhofer IIS, http://www.iis.fraunhofer.de/en/ff/bsy/leist/easydcp.html

[6] J. Matela, V. Rusnak, P. Holub, "GPU-Based Sample-Parallel Context Modelling for EBCOT in JPEG2000," Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10), pp. 77-84, 2010

[7] A. Weiß, M. Heide, S. Papandreou, N. Fürst, "CUJ2K: a JPEG2000 encoder in CUDA," https://sourceforge.net/projects/cuj2k/files/cuj2k-documentation.pdf/download, pp. 1-48, Sep. 2009

[8] F. Auli-Llinas, M.W. Marcellin, Stationary Probability Model for Microscopic Parallelism in JPEG2000, IEEE Trans. on Multimedia, Vol. 16, No. 4, pp. 960-970, Jun. 2014

[9] T. Richter, S. Simon, On the JPEG 2000 ultrast mode, IEEE Int. Conf. on Image Proc., pp. 2501-2504, Sep. 2012

[10] R. Yu, C. C. Ko, S. Rahardja, X.Lin, Bit-Plane Golomb Coding for Sources with Laplacian Distribution, Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing, Vol. 4, pp. 277-280, Apr. 2003

[11] G.E. Blelloch, Scans as Primitive Parallel Operations, IEEE Trans. On Computers, Vol. 38, No. 11, pp. 1526-1538, Nov 1989

[12] J. Luitjens, Faster Parallel Reductions on Kepler, https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler, Feb. 2014

[13] A. Balevic, "Parallel variable-length encoding on GPGPUs," Proc. of the 2009 int. conf. on Parallel processing, EuroPar'09. Berlin, pp. 26-35, 2010

[14] N. Sakharnykh, Fast Histograms Using Shared Atomics on Maxwell, https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell, Mar. 2015