Parallel efficient rate control methods for JPEG 2000

Martínez-del-Amor, Miguel Á., Bruns, Volker, Sparenberg, Heiko

Miguel Á. Martínez-del-Amor, Volker Bruns, Heiko Sparenberg, "Parallel efficient rate control methods for JPEG 2000," Proc. SPIE 10396, Applications of Digital Image Processing XL, 103960R (19 September 2017); doi: 10.1117/12.2273005

Event: SPIE Optical Engineering + Applications, 2017, San Diego, California, United States

Parallel Efficient Rate Control Methods for JPEG 2000

Miguel Á. Martínez-del-Amor[1], Volker Bruns[2], Heiko Sparenberg[3]
Moving Picture Technologies Department, Fraunhofer Institute for Integrated Circuits IIS
Am Wolfsmantel 33, 91058 Erlangen, Germany

ABSTRACT

Since the introduction of JPEG 2000, several rate control methods have been proposed. Among them, post-compression rate-distortion optimization (PCRD-Opt) is the most widely used, and the one recommended by the standard. The approach followed by this method is to first compress the entire image split in code blocks, and subsequently, optimally truncate the set of generated bit streams according to the maximum target bit rate constraint. The literature proposes various strategies on how to estimate ahead of time where a block will get truncated in order to stop the execution prematurely and save time. However, none of them have been defined bearing in mind a parallel implementation.

Today, multi-core and many-core architectures are becoming popular for JPEG 2000 codecs implementations. Therefore, in this paper, we analyze how some techniques for efficient rate control can be deployed in GPUs. In order to do that, the design of our GPU-based codec is extended, allowing stopping the process at a given point. This extension also harnesses a higher level of parallelism on the GPU, leading to up to 40% of speedup with 4K test material on a Titan X. In a second step, three selected rate control methods are adapted and implemented in our parallel encoder. A comparison is then carried out, and used to select the best candidate to be deployed in a GPU encoder, which gave an extra 40% of speedup in those situations where it was really employed.

Keywords: Image Compression, JPEG 2000, GPGPU, EBCOT, Rate Control

1. INTRODUCTION

JPEG 2000 (J2K) is a still image compression standard released jointly by ITU and ISO [4]. Its main applications today range from medical image compression to media entertainment. For the latter, JPEG 2000 was selected by the *Society of Motion Picture & Television Engineers* (SMPTE) to be the image coding baseline for both *Digital Cinema Packages* (DCP) and *Interoperable Master Format* (IMF). Among the advantages of the standard, J2K offers multiresolution capabilities, multilayer and embedded code streams, and rate-distortion optimization for high quality compression at low bitrates [22]. However, one of the major drawbacks of JPEG 2000 is its high complexity [21].

On the one hand, compressing high resolution images with JPEG 2000 is a very compute-demanding task [2]. This is the case for both the IMF and DCP formats. Specifically, the entropy coder, named EBCOT [20], is the main bottleneck (around 60% of the overall process [13][14]) and hence, the phase where many authors have invested most of the efforts [2]. In EBCOT, the output of the image transform is decomposed into code blocks, which can be processed independently [20]. Therefore, a coarse-grain level of parallelism is the encoding of code blocks, what is harnessed in most of hardware and CPU implementations [21].

Another approach to accelerate the execution of EBCOT is to offload it to parallel co-processor devices like *Graphics Processing Units* (GPUs) [5][13][14][15][24]. Today GPUs offer a highly parallel architecture with thousands of lightweight cores disposed in SIMD multiprocessors [17]. Programming GPUs has been eased over the years, and requires now a short learning curve [12] since the introduction of abstract programming models such as CUDA [27] and OpenCL [29]. GPUs are designed to support a massive level of fine-grained parallelism, so at least all code blocks of an image are processed simultaneously. In any case, efficient J2K codec implementations have been challenging and subject of many research works.

[1] miguel.martinez@iis.fraunhofer.de; phone +49 9131 776-5145; fax +49 9131 776-5109; www.iis.fraunhofer.de
[2] volker.bruns@iis.fraunhofer.de; phone +49 9131 776-5156; fax +49 9131 776-5109; www.iis.fraunhofer.de
[3] heiko.sparenberg@iis.fraunhofer.de; phone +49 9131 776-5143; fax +49 9131 776-5108; www.iis.fraunhofer.de

On the other hand, JPEG 2000 needs to compress the entire image (this is done in EBCOT tier 1) before truncating information in an optimal way in order to achieve the target file size (EBCOT tier 2) [22]. The recommendation of the standard is to use PCRD-opt [20], which follows this scheme. However, this introduces high redundancy when working at low bitrates, so an alternative is to stop the encoding of the image once a good compression is achieved. There are many complementary and alternative methods to PCRD-opt introduced in the literature [2]. For example, Chang et al. [9][10] propose to feedback information from tier 2 by constructing a table of accumulated budget per rate-distortion slope, stopping in this way the encoding of a code block, if its slope will already exceed the available budget. The most-used software for JPEG 2000, Kakadu, already implements a heuristic based on the same idea [21]. Aulí-Llinás et al. [1][2][3] propose a scan order over the code blocks where the passes over the bit planes are interleaved. When a desired budget is achieved, the coding is stopped.

Nevertheless, all the efficient rate control methods for JPEG 2000 are designed for a sequential processing of code blocks. Thus, incorporating any of them in a GPU J2K codec is not natural, and to our knowledge, has not been covered in the literature. In this paper, three methods to stop the encoding at EBCOT tier 1 on the GPU are introduced. To his end, a new design for the GPU encoder is also proposed, which in turn, provides better performance than the state of the art.

This paper is structured as follows. Section 2 gives an overview of the required background of GPU computing and JPEG 2000. In Section 3, the state of the art is discussed, with special focus on some existing rate control methods and on current architecture of J2K GPU codecs. Sections 4 and 5 introduce the proposed methods, for both a new EBCOT GPU architecture and for parallel strategies enabling early stop of encoding, respectively. Finally, Section 6 provides the obtained results of the methods, and Section 7 gives conclusion and proposal of future research lines.

2. PRELIMINARIES

2.1 GPU computing

The GPU is the heart of graphics cards [12]. They were conceived for rendering purposes only, allowing computing color attributes of pixels in a parallel way. After years of evolution and thanks to the fast-growing market of graphics, the GPU today is a highly parallel processor that can be employed for general purpose computations. Although its architecture looks powerful, the cores are much simpler and more lightweight than in commodity CPUs (although in larger number). In the specific case of NVIDIA's technology (the same concepts can be transferred to other vendors), a GPU consists of *Streaming Multiprocessors* (SMs), and SMs contain in turn *Streaming Processors* (SPs, or *cores*). Typically, the number of SPs per SM depends on the microarchitecture version, and the number of SMs on the device range.

SMs work on a *Single-Instruction Multiple-Thread* (SIMT) fashion [12], so that the threads scheduled on a SM execute the same instruction on different pieces of data. More specifically, SMs schedule threads in groups of 32 (called *warp,* or *wavefront* in AMD architectures), in such a way that a warp is the parallel granularity on the GPU. A warp is executed in *Single-Instruction Multiple-Data* (SIMD), so the threads in it have to be synchronized. Whenever threads in a warp run different instructions because of branching (the threads *diverge*), their execution is implicitly serialized (into subgroups of threads being synchronized).

In summary, GPUs provide a system optimized for data parallelism. As mentioned, their programmability is eased by programming models such as *Compute Unified Device Architecture* (CUDA) [27][28]. Introduced almost 10 years ago, it provides a model that abstracts the GPU architecture in a scalable way; that is, a program in CUDA can run on different GPU architectures and scale automatically to the resources available in there. CUDA is a proprietary technology of NVIDIA, while OpenCL [29] was first conceived to become a standard in GPU computing. The concepts in OpenCL and CUDA are similar, so in what follows, we will only focus on the description of the latter.

In CUDA programming model, the CPU (*host*) takes control of the execution flow, and offload to the GPU (*device*) pieces of code (*kernel* function) [28]. The execution of kernels is carried out by a *grid* of *threads*. Typically, a grid is composed of thousands of threads, allowing occupying the hardware resources. Launching a large amount of threads helps to hide stalls in their execution, given by dependencies, memory accesses, etc. The grid is a two-level hierarchy, where *threads* are arranged into *thread blocks*. All blocks have the same number and organization of threads. Each block is identified by a two dimensional identifier, and each thread within its block by a three dimensional identifier. In this way, any thread can be identified unequivocally by the combination of both thread and thread block identifiers. The execution of threads inside a block can be synchronized by a *barrier* operation (*__syncthreads()*), and threads of different blocks can be synchronized only by finishing the execution of the kernel (in CUDA versions older than 9 [27]).

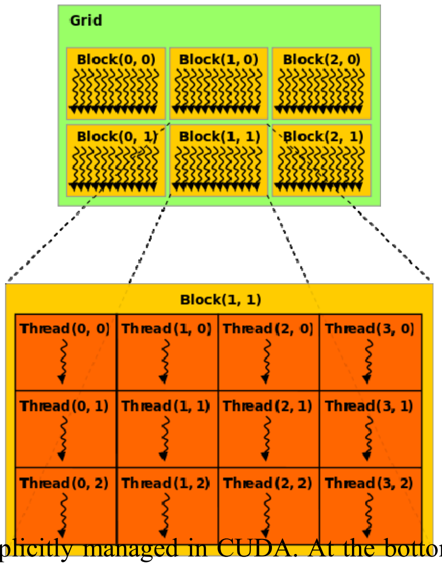


Figure 1. Threading model in CUDA. Threads are executed in grids organized by blocks. From [28].

Furthermore, the memory hierarchy is explicitly managed in CUDA. At the bottom of the hierarchy lies the *global (or device) memory,* which is the largest but the slowest memory in the system. It is accessible by the host since it is the communication channel between CPU and GPU, and all threads share the same memory space over it. A *L2 cache* memory system is built in recent GPUs to speedup accesses to global memory. *Shared memory* is smaller but faster than global memory, and accessed only by threads belonging to the same block. Next to shared memory, a *L1 cache* is also available. Normally, performance of CUDA applications depends on at what extend shared memory is exploited. Finally, every thread has access to its own variables in registers [12][28].

Therefore, the most efficient way to structure an algorithm is as follows: (1) threads of each block read their corresponding data portion from global memory to shared memory (if cooperation is required) or to registers (otherwise), (2) threads work with the data directly on local memory, and (3) threads copy these altered data back to global memory [17].

A good pattern to follow when doing parallel programming with GPUs is to use the well-known *primitives* [17]. They are functions that fit into the parallel architecture of GPUs, and for which exist already efficient implementations. These primitives can be used as building blocks for algorithms. Examples of primitives are: *reduce* (computing a value from an array, e.g. the sum of all elements), *scan* (accumulate partial results in an array, e.g. the accumulated sums of the elements in an array), *compact* (discard elements in an array according to some value), etc.

Finally, by using CUDA *streams* it is possible to overlap the execution of kernels and memory transfers, and also to launch several kernels at the same time to the same GPU (they will execute in parallel as long as resources are available).

2.2 JPEG 2000

In JPEG 2000, the coding pipeline is based on first applying a color transform to the images, followed by a frequency transform. The output of this transformation is processed by an entropy coder in order to reduce the size of the resulting bit stream (actually doing the compression) [22]. There are two modes for lossy or lossless compression. For the latter, a reversible path is required so that no information is missing in the resulting compressed file. For the former, some information can be discarded, so no reversibility is required. In this paper, we will focus on the lossy mode, since it is the one employed for the digital cinema (DCP) and IMF profiles.

The encoding of an image in lossy mode starts with the *Irreversible Component Transform* (ICT), where the pixels in RGB are converted into a YC_bC_r color space. This reduces the redundancy between color components. The following stage is the application of the *Discrete Wavelet Transform* (DWT). The DWT applies a 9/7 filter, both horizontally and vertically, to separate low and high frequencies into subbands. The wavelet decomposition follows a Mallat pyramid, in which for each resolution level, four subbands are obtained: LL, HL, LH and HH. The LL subband is a low resolution version of the compressed image, and can be further decomposed as long as more wavelet decompositions are applied. Next, subband samples are quantized, using a *deadzone scalar quantization*. Quantization step sizes are constant per subband and typically low frequencies are quantized less heavily than high frequencies [22].

The quantization indices are then sent to the entropy coder. In JPEG 2000, this is carried out by the *Embedded Block coder with Optimized Truncation* (EBCOT). Here, each quantized subband is split into code blocks, which contain NxM samples (DCP and IMF specifications state a size of 32x32). Each code block is entropy coded independently. For this aim, EBCOT is composed of two tiers: in tier 1, each code block is compressed into an embedded bitstream, and in tier 2, the set of all bitstreams is processed in order to find the set of truncation points (one per code block) that yields the best rate-distortion ratio [22].

EBCOT tier 1 [20] operates on a sign-magnitude representation of the quantization indices. A code block bitstream is created by traversing the bit planes (i.e. set of bits of all samples of a code block at a given position (offset) in the magnitude part), from the *most* (MSB) to the *least significant bit plane* (LSB). Note that MSB bit planes with only zeroes are skipped. Each bit plane is scanned by three passes, which follows an order based on stripe columns (4 sample column). The first pass is called *Significance Propagation Pass* (SPP) and codes symbols from samples that are not significant yet, but that will likely become significant (i.e. higher bit planes are still zero) according to the neighborhood context. The second pass is denoted as *Magnitude Refinement Pass (*MRP) and codes symbols from all samples that are already significant. At last, the third pass is the *Clean-up Pass* (CUP), which codes all remaining symbols. There are four *coding primitives* that are employed by the passes in order to generate the corresponding symbols. Signs are coded as long as the corresponding sample gets significant (found the first bit equals to 1). Moreover, the context (neighborhood states) is also taken into consideration, so that the symbols generated out of each pass are included into *Context-Decision (C-D) pairs* containing a context value out of 18 available and a decision bit (the symbol).

C-D pairs are further processed by a context-adaptive arithmetic coder named MQ-coder. It receives the context-decision pairs in the order they are generated in each pass; in other words, the MQ-coder processes the pairs sequentially in exactly the same order they were issued. Therefore, context modelling (CM) and MQ-coder (MQ) can be regarded as separate processes in the encoder.

According to the JPEG 2000 standard, a specific rate control strategy to achieve a target bitrate in EBCOT tier 2 is not defined. The recommended one is the *Post Compression Rate Distortion Optimization* (PCRD-opt). Rate-distortion optimization is used to minimize the overall distortion of an image under a target bitrate R_{max}. After EBCOT tier 1, along with the embedded code streams of each code block, a set of distortion (D(z)) and rate (R(z)) values are calculated after each pass z. Then, PCRD-opt uses the Lagrange multiplier technique to get an optimum solution to this problem: $D(\lambda)+\lambda R(\lambda)$, where λ minimizes the expression yielding $R(\lambda) = R_{max}$. λ is also associated to the Rate-Distortion curve, which represents in fact the slope in a given point. The process is as follows: first of all, for each code block, a set of feasible truncation points is calculated. These are those points whose slope lies on the convex hull formed by the R-D curve. That is, they are given by the following property: a pass z belongs to the set of feasible truncation points if and only if $\lambda(z) > 0$ and $\lambda(z) > \max(\frac{D(z)-D(t)}{R(t)-R(z)})$, with $t > z$ (i.e. the slope at that point is greater than the maximum possible computed slope from the rest of passes). Finally, the process finds the smallest slope (a.k.a. *threshold slope*) from all the code blocks yielding to the closest bitrate to the target [22]. It is noteworthy that the calculation of this optimal truncation point can be done only after fully encoding all the code blocks.

Finally, once the optimal truncation point is computed, the bitstreams are trimmed and packetized into the final codestream in packets. Each packet contains a header and a body part, and headers are further compressed using a tag tree scheme. Thus, in order to compute the optimal truncation point not exceeding the target budget, simulation of packet headers (and remaining markers as well as main headers defined in the standard) has to be also considered.

3. STATE OF THE ART

3.1 Efficient rate control methods for JPEG 2000

As discussed, EBCOT tier 1 has first to generate the whole compressed image to be truncated later by this method. In those scenarios with low bitrates, passes belonging to the least significant bit planes might be discarded by PCRD-opt. Therefore, it makes sense to stop early the encoding at tier 1 by using methods that can detect in advance this situation.

Although there is a plethora of methods for rate control [2], only some of them cope with avoiding the encoding of the entire image. In this section, a selection of techniques is described.

3.1.1 Kakadu's heuristic, Slope-Byte table (SB) and Minimal Slope Discarding (MSD)

Kakadu is the most widely-used software for JPEG 2000 that is developed by the main contributor to the standard. In it, a predictive truncation strategy is implemented. The heuristic is based on first calculating statistics from past encoded code blocks and use them to estimate, if a coding pass is going to be discarded in tier 2 [21]. The statistics and estimators computed in Kakadu's heuristic for predictive truncation are similar to the methods introduced in [9] and in [10]. Hence, in what follows, we will focus on these two methods.

The *Slope-Byte table* method (SB) [10] uses a table to store, for each possible slope value, an accumulated number of bytes given by the lengths achieved in each pass of the previously encoded code blocks. If after accumulating that length for a given slope it exceeds the target bitrate, the encoding of such a code block can be stopped. Figure 5 sketches this algorithm. Given that such a table can be very large and not manageable in hardware (and very expensive for CPU encoders), the number of entries of the table can be reduced by working per slope ranges. However, this will come at the cost of losing quality. According to the author's results, SB gives PSNR differences with respect to the Verification Model no worse than 0.06 dB.

The *Minimal Slope Discarding* method (MSD) [9] works similarly as SB, but when exceeding the bitrate, it goes to a second step in which the minimal slope value among the coded code blocks are found. Then, it discards the pass with this minimal slope (decreasing in this way the accumulated bitrate). If the code block with the minimal slope is the current one, the encoding of the code block can be stopped and the execution is moved to the next one. This method can be applied after computing the statistics with SB, so achieving perfect rate control without paying PSNR degradation.

3.1.2 Coding Passes Interleaving (CPI)

The aim of *Coding Passes Interleaving* (CPI) method defers from SB and MSD in that it aims to replace PCRD-Opt [1][2]. It also proposes to delete tier 2 from EBCOT, and perform everything inside tier 1. The three following premises are the theoretical background of the method: (1) almost all coding passes are feasible truncation points (because of small coding segment lengths); (2) a coding pass is more likely to be included in the codestream, if it belongs to a high (more significant) bit plane; and (3) PCRD-Opt includes all coding passes in higher bit planes, but differs in the lower ones. Bearing this in mind, the algorithm encodes the image by interleaving the passes of all the code blocks. In other words, instead of encoding block by block, it performs the first pass of all code blocks, then the second pass on all coe-blocks, and so on.

For this aim, a total order is applied to the coding passes that are applicable to all code blocks. This will depend on the bit plane where each code pass is being applied. First, the coding pass types are numerated as follows: SPP=2, MRP=1, CP=0. Second, the total order of the application of a coding pass is defined as *Significance Level (SL),* being $SL = (bitPlane \times 3) + CoPType$, where *CoPType* is the coding pass type. As mentioned in Section 2.2, insignificant bit planes (being all zero from the MSB) are discarded (so no coding passes are applied to them). Therefore, not all code blocks are active in some high significance levels.

Finally, the main loop of the CPI algorithm traverses the significance levels for every code block. The order in which the code blocks are processed is also important, and for that, the subbands are also sorted, so the LL subband is visited first. Moreover, the color components are interleaved. The pseudocode is depicted in Algorithm 1.

```
curRate ← 0
MSCoP ← (max_cblocks(MSB) + 1) * 3
LSCoP ← (min_cblocks(LSB)) * 3
for SL ← MSCoP … LSCoP
    for each Resolution Level
        for each Component
            for each Subband
                for each Code block CB
                    CoPType ← SL mod 3
                    bit plane ← floor(SL / 3)
                    rate ← Encode(CB, CoPType, bit plane)
                    curRate ← curRate + rate
                    if curRate >= targetRate
                        StopEncoding
```

Algorithm 1. CPI pseudocode, where it can be seen the order in which code blocks are visited, from [1].

CPI allows stopping the encoding once the target length is reached. Thanks to the special order, the lowest subbands have more preference than the higher, but higher sample values start to contribute to the length before lowest values. This way, the truncation point is found during encoding, without the need of creating the whole compressed imaged and running PCRD-opt afterwards.

This algorithm can also be used in the decoding process to achieve a desired target bitrate without the use of any quality layer. Finally, the quality delivered by CPI compared to PCRD-Opt is not well-balanced and for some bitrates is 0.5dB worse [3], and the decoder gives differences of 0.17dB. This method was improved by an extension called ROC [3], in which they identify that the inclusion or not of some passes leads to higher differences in PSNR. Using ROC, the average difference is 0.077dB.

3.1.3 Minimum RD (MRD)

MRD, presented in [19], takes advantage of reusing the information from the encoding of lower subbands in the higher ones. In this case, those code blocks at LL0 subband are first encoded with EBCOT tier 1, and their information is sent to tier 2 (with PCRD) to calculate the minimal slope λmin. This value is employed to obtain the optimal slope as

$$\lambda_{opt} = \frac{\lambda_{\min}}{-bitDepth}$$

The passes of code blocks of the remaining subbands are processed by tier1, if their RD slopes are greater than or equal to λopt.

The authors ensure that the quality of the decompressed image is very close to the one produced with PCRD-Opt. by just using the slope from the lowest subband. They conclude that the maximum difference considered in their work is up to 0.23dB.

3.1.4 Vikram's estimators for Length and Distortion

Finally, it is noteworthy to mention a method to estimate the slopes without the need of having the exact lengths computed after sending the C-D pairs to the MQ coder. In [23], two estimators for length and distortion obtained after encoding a bit plane are defined. The information to compute this only depends upon the bit values in a bit plane. However, the estimators need to use parameters of lines of best fit for each bit plane and subband, what the authors affirmed to have calculated, but not published yet.

Using these estimators, the above mentioned methods could be further approached by approximating the slopes in context modelling, not requiring to execute first the MQ coder in order to know the exact lengths after each pass.

3.2 JPEG 2000 encoder on the GPU

In a JPEG 2000 encoder, the most time consuming task is EBCOT, followed by DWT. An efficient implementation of them is, therefore, critical for a good performance. In this concern, the GPU has been successfully employed to accelerate JPEG 2000 coding implementations. While ICT is highly parallelizable, and DWT can be also parallelized with good performance [5][11][7], running entropy coding on the GPU becomes a bottleneck [18]. Hence, many efforts have been posed in order to extract more fine-grained parallelism from EBCOT.

The first published approach to implement a J2K encoder on GPUs was CUJ2K [25]. The design of the grid was to assign a single thread for each code block, so that they then traverses the bit planes applying the passes sequentially and feeding the symbols to the MQ coder. Best performance was achieved using 64 threads per thread block. In this very first design, using GTX280, and compared against a single-threaded execution of Kakadu, the GPU encoder was 2.6 times faster.

After many efforts, it has been shown that parallelizing the execution of the MQ coder over a sequence of context-decision pairs is not possible [16]. Therefore, a way to increase parallelism in EBCOT implementations is to detach the MQ coder from CM . In this kind of design, the MQ coder runs sequentially (using a similar grid design of a thread per code block, and 64 threads per thread block) in a separated kernel, so that CM can be further optimized. Some more efforts can be posed to improve the GPU code running the MQ coder [16], or even to replace it by a more classical arithmetic coder [13], but the execution still remains sequential.

In [13], [14] and [24] similar ways to bring more parallelism to CM are discussed. The basic ideas for this is to break the dependency between samples (because of the context formation) by execution a pre-processing stage, and to fuse the execution of coding passes in order to avoid warp divergence. The latter is accomplished by executing the four coding primitives instead of the passes. Each coding primitive gets effective when certain merged conditions from the passes hold [14][15]. A typical grid for this design is a 32x8 threads per thread block, assigning a thread to each column stripe (and assuming 32x32 code blocks, used for DCI profiles).

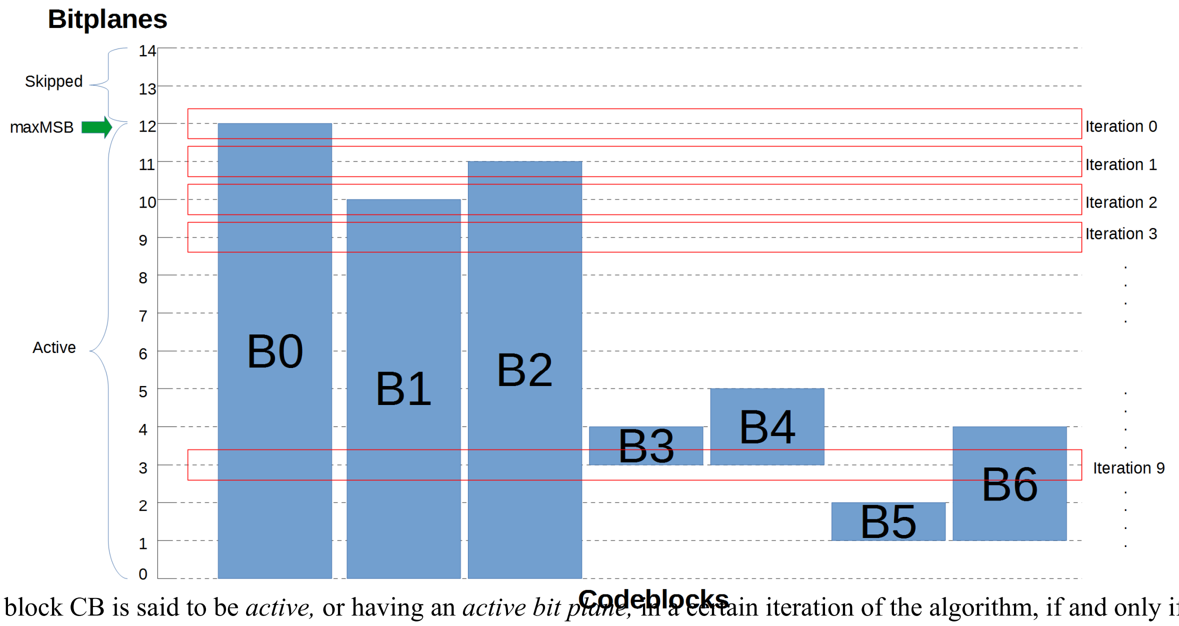
Thus, EBCOT tier 1 is executed in two stages: first by a CM kernel that runs context modelling to all code blocks at the same time, going through all bit planes and collecting all C-D pairs; and second, by a MQ kernel that receives as input the sequence of C-D pairs and generate the corresponding bitstreams. After that, EBCOT tier 2 is executed. Although PCRD-Opt can be also executed in parallel by probing different slopes for truncation points, this stage along with packetization is usually executed on the host. In this case, it is required to send the set of all bitstreams and metadata computed (lengths and slopes, for example) to the CPU.

When working with a sequence of images, and in order to increase the occupancy of resources in GPUs, a strategy is to encode several frames at the same time. Therefore, such a codec is said to work with GOPs (Group Of Pictures) [8]. GOP sizes range from 2 to more than 8 (depending on available memory on the GPU). When GOP size is bigger than 1, the code blocks of the frames are all handled at the same time by the EBCOT kernels. Moreover, using CUDA streams, it is possible to overlap the transmission of GOP data with the computation of the previous or next GOP.

4. PARALLEL ENCODING BY BIT PLANES (CIBO)

According to the SoTA GPU J2K encoder designs, kernels executing CM and MQ are separated, since more parallelism can be exploited in CM rather than in MQ. However, with this design, it would not be possible to bail the encoding in tier 1 prematurely for CM since the lengths, and hence the slopes, are only available after the MQ coder is executed. Although the lengths (and distortions) can be approached through Vikram's estimators, the accumulated errors when they are employed within the efficient rate control methods aforementioned can be enlarged.

In order to effectively compute the corresponding lengths achieved after each coding pass, and to be able to also stop the CM kernel prematurely, the MQ kernel should start its execution before the CM ends. In this proposal, both the CM and MQ kernels are redefined to work only in a bit plane basis; that is, instead of looping over all bit planes for every code block, they run CM and MQ for only a given bit plane. In this way, the host will have to implement the loop over the bit planes (instead of in the device), calling to CM and MQ kernel in each iteration. More precisely, the procedure starts by calculating the maximum MSB (*maxMSB*) over all code blocks, so that iterations over insignificant bit planes can be skipped. Then, the loop starts from the maxMSB, and executes the CM kernel over the code blocks being significant in bit plane MSB. Once CM is finished for this bit plane, the MQ kernel is launched and processes the C-D pairs generated. The loop then decreases the bit plane iterator variable and performs similarly as explained. Figure 4 shows the workflow followed by this design (assume that the “check stop” box in the figure is empty at this point). The order in which bit planes are visited for all code blocks is represented in Figure 2.



A code block CB is said to be *active,* or having an *active bit plane,* in a certain iteration of the algorithm, if and only if

Figure 2. Interleaved bit plane execution for 7 blocks. B3 to B6 have been weighted accordingly to the subband.

$$MSB_{CB} \leq maxMSB - iter \leq LSB_{CB},$$

where MSB_{CB} and LSB_{CB} are the most and least significant bit planes of the code block. Before executing CM or MQ kernel in an iteration, the device does not know which code blocks are active in that moment. A conservative strategy is to launch thread blocks and threads always for all code blocks, and checking later the active condition within CM and MQ kernels, finishing the execution of those threads working with not active code blocks. But this can be optimized by adjusting the number of threads in each iteration ahead of time and not spawning threads for inactive code blocks in the first place. In order to do this, it is required to compact the list of code blocks. On the one hand, a compaction will provide the amount of active code blocks; on the other hand, compacting will allow launching contiguous threads and thread blocks to all active code blocks, avoiding to launch threads whose execution will end right at the beginning of the kernels.

Hence, an extra code block-mapping kernel is launched at the beginning of each iteration, which calculates the active code block condition according to the iteration. This information is then processed by the *compact* primitive to compute the indexes of active code blocks. This list is passed to each kernel and is used to map threads to code blocks. It is worth to note that many libraries, such as Thrust and CUB, already provide efficient implementations to compact.

However, the strategy of having this loop over bit planes and compacting active code blocks achieves worse performance than just running CM and MQ at once like in the SoTA. The main reason for this is that in every iteration, the kernel needs to re-read the image information from global memory, and write it back also to global memory. Thus, another optimization is to overlap executions of CM and MQ as follows: MQ kernel for bit plane *b* with CM kernel for bit plane *b-1*. The goal is to have a similar execution as in Figure 3. This can be accomplished by using two CUDA streams, one for CM and other for MQ. Experimental results show that, although the overlapping is not fully accomplished for all iterations, the final performance is better than when not overlapping. In what follows, we will name this strategy as *CM-MQ interleaved bit planes overlapping* (*CIBO,* in short).

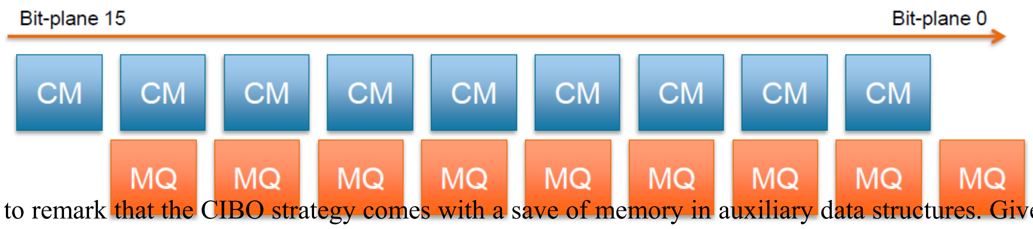


Figure 3: Overlapping the execution of CM and MQ kernels per bit plane/iteration.

It is important to remark that the CIBO strategy comes with a save of memory in auxiliary data structures. Given that the device needs to keep the computed C-D pairs for MQ kernel, running CM for all bit planes will required enough space to store all possible C-D pairs. However, with CIBO it is required to store the pairs for only 2 bit planes. This will allow launching the codec with larger GOPs, what becomes very important on GPUs with small amount of memory.

Last but not least, in [6], a variant of this design was employed in order to enable JPEG 2000 fast mode on a GPU encoder. Thanks to the iteration by bit planes, it was easier to implement a bypassing.

5. PARALLEL STRATEGIES FOR EARLY STOP OF ENCODING

In this section, three novel techniques for stopping a parallel implementation of EBCOT tier 1 prematurely are proposed. PCRD-opt is still employed in tier 2 for an optimal truncation point search, since the goal is to explore which strategies fit better to the GPU architecture, get more effective when applied in parallel, and minimize the impact in the resulting quality.

These strategies are based on the above mentioned efficient rate control methods (Section 3.1). Specifically, one method employs the SB and another the CPI algorithm. However, MRD was not used because encoding first LL subband and the rest afterwards is actually almost doubling the encoding time, since those code blocks belonging to LL cannot be processed in parallel with the rest.

Figure 4 summarizes the flow diagram of the execution of EBCOT's CM and MQ kernels with CIBO design, and using a stop checking procedure (employing one of the parallel methods discussed here). First we calculate the maximum of the MSBs of each code block; secondly, only active code blocks for that bit plane are compacted; third, CM kernel is launched; fourth, the execution is stopped until CM is finished; fifth, MQ kernel is launched; sixth, the stop criteria is executed after MQ; seventh, the stop condition is checked to exit the loop, if not, the execution flow continues back to compact and CM (that can be overlapped with MQ and check stop if using 2 CUDA streams).

5.1 Parallel Slope Threshold (PST)

A straightforward approach when working with picture sequences is to stop the encoding of images by using the threshold slope of previous frames. In order to do that, the calculation of slopes inside the MQ kernel has to be performed. In the SoTA version, this calculation is done at the end of MQ kernel only, once the whole code block is encoded, in order to avoid divergent threads (note that for this aim, all previous truncation points have to be computed). However, for this purpose, this calculation should take place incrementally pass by pass (or at least, by bit plane).

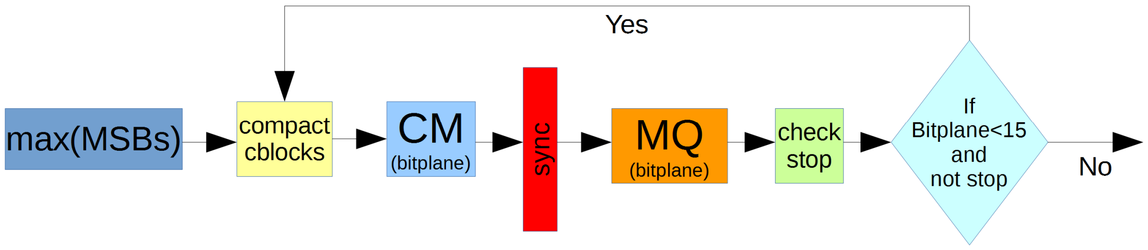


Figure 4. Flow diagram for the EBCOT kernels launch, with stop checking.

Thus, in this method, a compromise is adopted: the slopes are estimated without having to compute truncation points, so they might be a few bytes off, and by using the distortion and lengths computed in CM and MQ respectively for that pass. This reduces the complexity of the kernel and thread divergence.

While the main advantage of this solution is its simplicity, since it only needs to store the threshold from the previous frame and send it to the next one, there are some drawbacks:

- If there is an abrupt decrease of threshold values in the sequence, then we are stopping the encoding too early. That is, if we pass a high threshold value to a frame which will have almost threshold slope 0, we will then loose information. However, we do not expect a huge impact in quality, because a threshold slope 0 means that the bitrate was enough to code all the information contained in the image, so dropping some bit planes should not be very harmful. Moreover, this would affect only to a single GOP.
- When working with GOPs of large size (e.g. 8), we are decreasing the locality of the frames: one frame will give estimation to its (e.g. 8th) predecessor. The larger the size of GOP, the smaller the correlation of the frames.

In this method, the check stop phase is implemented with only one kernel, which computes the slopes at the end of the bit plane, and if it exceeds the threshold propagated from the previous one, encoding of the corresponding code block is stopped. This comparison also includes a conservative margin in case the reference slope was very high.

5.2 Parallel Slope-Byte (PSB)

This method adapts the idea of the Slope-Byte table to be executed in parallel. Therefore, PSB uses the same statistics collected with SB, but with not fully encoded code blocks (since they are processed in parallel). These gathered partial statistics can be enough to estimate a stop condition at low bit planes.

The procedure is as follows: after each iteration, the corresponding slopes and lengths are calculated using the same estimation as for PST (that is, without finding exact truncation points). Then, the slope is used to index the table and accumulate the obtained length.

In order not to overflow the memory with an entry per possible slope value, the size of the table employed here is the same as suggested in the original SB method [10], of 1024 entries. Moreover, the slopes are quantified to fit into this table by dividing their values by 64, what be achieved simply bit-wise right-shifting their values 6 times. Again, each frame has to be treated separately within a GOP, since this method is applied to still images. Thus, one SB table per frame in a GOP is required.

The overall behavior is sketched out in Figure 5. Specifically, the implementation consists of three kernels:

- Kernel 1: each thread computes the slope and the length obtained after the iteration/bit plane coding of a code block. This information is then used to accumulate values in the SB table, which resides in global memory.
- Kernel 2: a block of threads reads the SB table from global memory, and scans (prefix sum) the values locally. Once a value is detected to exceed the target bitrate, the corresponding index (bit-wise left-shifted 6 times) is annotated as the slope threshold for that image.

- Kernel 3: uses the slope threshold calculated in Kernel 2 to mark code blocks as prematurely stopped, similarly as the PST method.

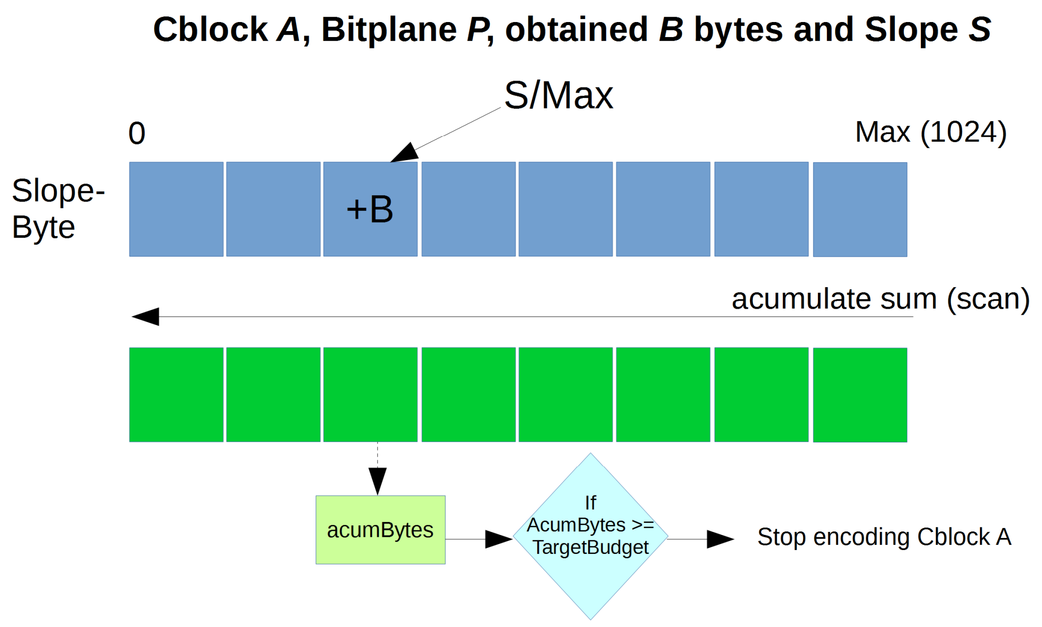

Cblock *A*, Bitplane *P*, obtained *B* bytes and Slope *S*
S/Max
0
Max (1024)
Slope-Byte
+B
acumulate sum (scan)
acumBytes
If AcumBytes >= TargetBudget
Stop encoding Cblock A

Figure 5. Summary of PSB method. For every code block, accumulate the obtained bytes for a biptlane for the entry corresponding to the slope.

5.3 Bit plane Parallel Interleaving (BPI)

This method uses the same base than the CPI algorithm. BPI takes advantage from the fact that the CIBO design of EBCOT kernels performs the same interleaving scheme than CPI, but in a coarse granularity: the bit planes are interleaved instead of each coding pass. Thus, each iteration of the CIBO kernels corresponds to a full scan iteration of three code passes for the corresponding bit plane. In other words, BPI is an approximation to CPI by using more conservative stopping criteria, because it is checked after coding a bit plane, what corresponds to three iterations of the outer-most loop of the CPI algorithm. This parallel approximation will give larger images as limited by the target bitrate, because we cannot stop the encoding at a fractional bit plane. Moreover, overhead from packet, tile and main headers are ignored. Subsequently, PCRD-opt is still used to find the optimum truncation point.

The implementation of this method is simpler than PST and PSB, given that there is no need to calculate slopes after every iteration. Instead, it requires computing the total length of all encoded bitstreams from all processed code blocks. If it exceeds already the target bitrate, then the execution is stopped and goes to PCRD-opt. Moreover, special care has to be taken with GOP sizes larger than 1, since, again, each image has to be treated separately. This requires running separately the computation of the total length for each image in the GOP separately.

In short, BPI is based on two kernels:

- Kernel 1: each thread calculates the accumulated bytes for a code block, and then each thread block reduces locally the amount calculated by its threads, what gives the total accumulated lengths for all code blocks covered by a thread block.
- Kernel 2: the amounts calculated in Kernel 1 are globally reduced, so calculating the total partial length of the image at that point. The termination of code blocks is marked for the frame that has exceeded the target bitrate within a GOP.

Note that if a frame within a GOP does not exceed the bitrate, its code blocks are still active and continue the process with the CIBO kernels. Finally, Figure 6 shows an overview of the BPI procedure.

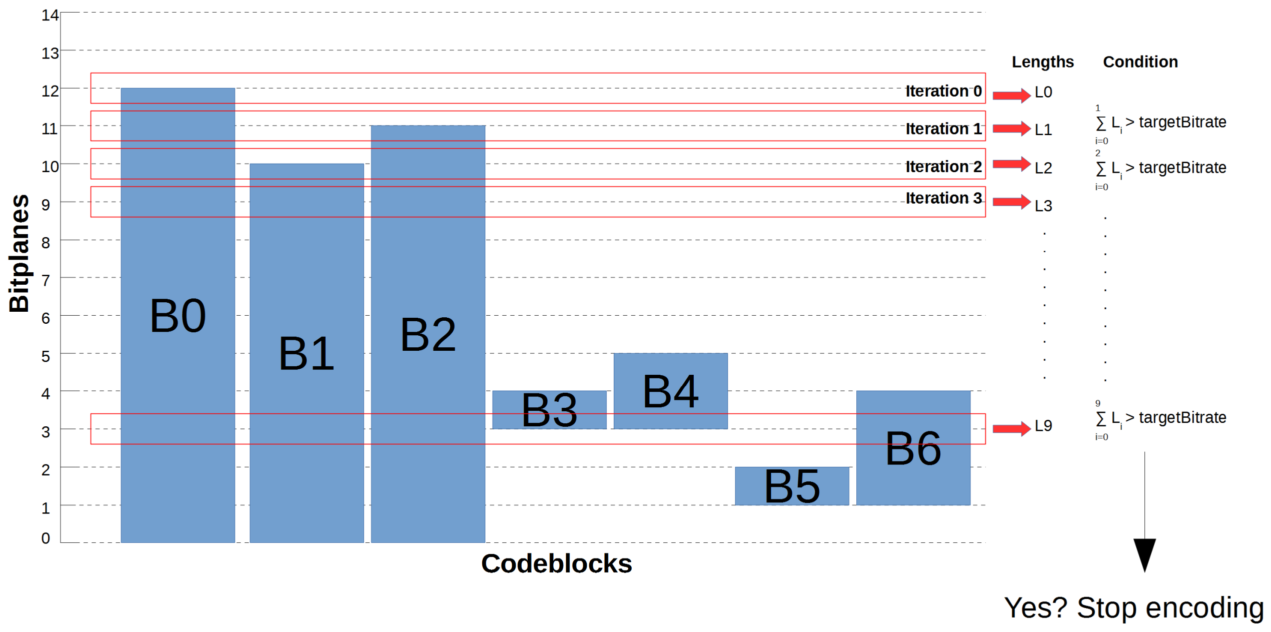


Figure 6. Summary of BPI method. For every bit plane, compute the lengths, and use the accumulated lengths so far for the condition.

6. RESULTS

The experiments carried out to test the methods here proposed have several dimensions: PSNR differences and speedup achieved with different bitrates, GOP sizes and GPU generations. This is covered by two benchmarks: one by looking to the drop of PSNR and obtained speedup with respect to different target bitrates on a GTX 470, and another by exploring the scalability of achieved speedups and drop of quality when increasing the GOP size on a Titan X.

Table 1. Average PSNR drops and speedups achieved for every parallel method using a 4K sequence from Sintel short movie at 24 fps.

Method	**Meas.**	**Bitrates (Mbit/s)**						
		250	**200**	**150**	**100**	**80**	**50**	**25**
PST	*PSNR dif*	0.541	0.334	0.262	0.528	0.401	0.506	0.507
	Speedup	0.994	1.003	1.008	1.022	1.021	1.021	1.048
PSB	*PSNR dif*	0	0	0	0	0	0	0
	Speedup	0.981	0.987	0.988	0.986	0.992	0.994	1.007
BPI	*PSNR dif*	1.134	2.024	2.587	2.609	2.691	2.827	2.832
	Speedup	1.054	1.067	1.122	1.176	1.188	1.233	1.294

Table 1 shows the first experiment carried out. It compares the three proposed methods PST, PSB and BPI when applied at different bitrates in Mbit/s (ranging from 250 to 25). The results were obtained for a sequence of 1000 frames from the short movie Sintel in 4K. The GPU employed was a GPU GeForce GTX 470 (448 cores, 14 SMs, 1.3 GB of memory, Fermi architecture). The GOP size was set to 1, and PSNR differences and speedups are calculated taking as reference the execution with CIBO kernels.

According to Table 1, the maximum speedup achieved by PST is up to 5% at 25 Mbits/s, and in average it is of 2%. From 250 to 150 there is barely any improvement of performance. From the PSNR point of view, the loss of quality is always around 0.5 dB. In this experiment, we can conclude that the PST brings a balance between quality loss and acceleration. For PSB, the difference in PSNR was always below 10^{-5} dB, so we can say the quality remains unchanged. However, there is no positive speedup. In fact, the performance was worse (around 1-2%) until the bitrate of 25, where the runtime of the kernels was very similar. Concerning BPI, the method achieves better results in efficiency, up to 29%. The method was always better for all bitrates, but the impact is more significant below 200 Mbits/s. However, this comes with a large drop of quality of around 2.5dB.

Concerning the second experiment, Figure 7 shows the tests carried out with the following four short sequences (48 frames long) which contain high amount of information:

- DCI 2: DCI's Standard Evaluation Material (StEM) in RGB 4K, frames 1560 to 1599.
- DCI 3: DCI's Standard Evaluation Material (StEM) in RGB 4K, frames 3400 to 3439.
- DCI 4: DCI's Standard Evaluation Material (StEM) in RGB 4K, frames 5100 to 5139.
- DCI 6: DCI's Standard Evaluation Material (StEM) in RGB 4K, frames 10000 to 10039.

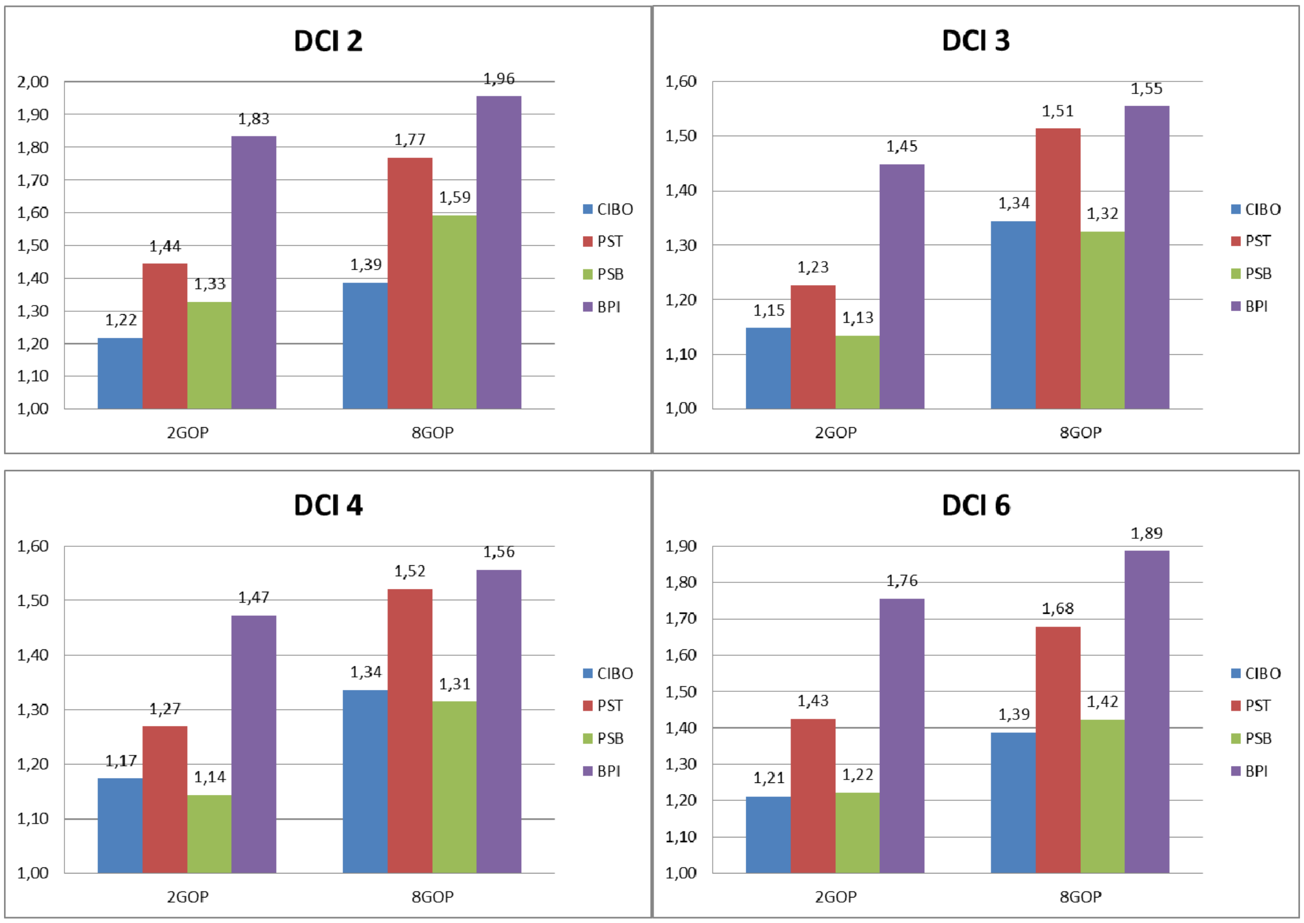


Figure 7. Speedup results (y-axis) compared against the SoTA kernels for CIBO kernels, and for parallel methods PST, PSB and BPI for early stop of encoding. GOP sizes were 2 and 8, and 5 different sequences. A TitanX (Maxwell) GPU was used.

The speedups shown in the figure correspond to the total time of the whole encoding process and are calculated with respect to the encoder using the SoTA kernels for EBCOT. This experiment was run for a target bitrate of 250 Mbit/s on a GPU TitanX having 3072 cores, 24 SMs, 12GB of GDDR5 and Maxwell architecture.

It can be seen, first, that the speedup scales along with the GOP size; that is, when dealing with more pictures at the same time, the impact of the methods is larger. This trend can be seen first on the speedup achieved by the CIBO strategy for EBCOT kernels, where the 15% in DCI 3 and 4 for GOP 2 goes to 34% for GOP 8, and from 20% in DCI 2 an 6 for GOP 2 to 39% with GOP 8. This shows that CIBO significantly accelerates the execution of the process thanks to the overlapping of CM and MQ kernels, given that the GPU gets more saturated at higher sizes of GOPs, distributing the workflow in a better way.

Looking to the methods for early stop of encoding, PSB offers an extra 10% of speedup with GOP 2 and 20% with GOP 8 in DCI 2. This sequence contains detail-rich images; hence, the method becomes effective in this situation. However, the downside of it is that the required overhead makes it worse with other sequences where the method does not take effect. For example, in DCI 4, PSB is 4% slower than the reference. Regarding PST, it brings in average an extra of 20% of acceleration. It shows that for the first case, the method becomes effective even though the overhead entailed. The low impact of the overhead in this experiment comes from the high amount of resources on the Titan X. In this case, it would be possible to overlap the execution of the kernels to update the S-B table with CM kernel, and they also run faster. BPI achieves an extra of up to 60% of speedup, being even almost twice as fast as the SoTA with DCI 2. For the rest of the sequences, this method improves by ranges between 20% and 50%, making it very attractive for further research.

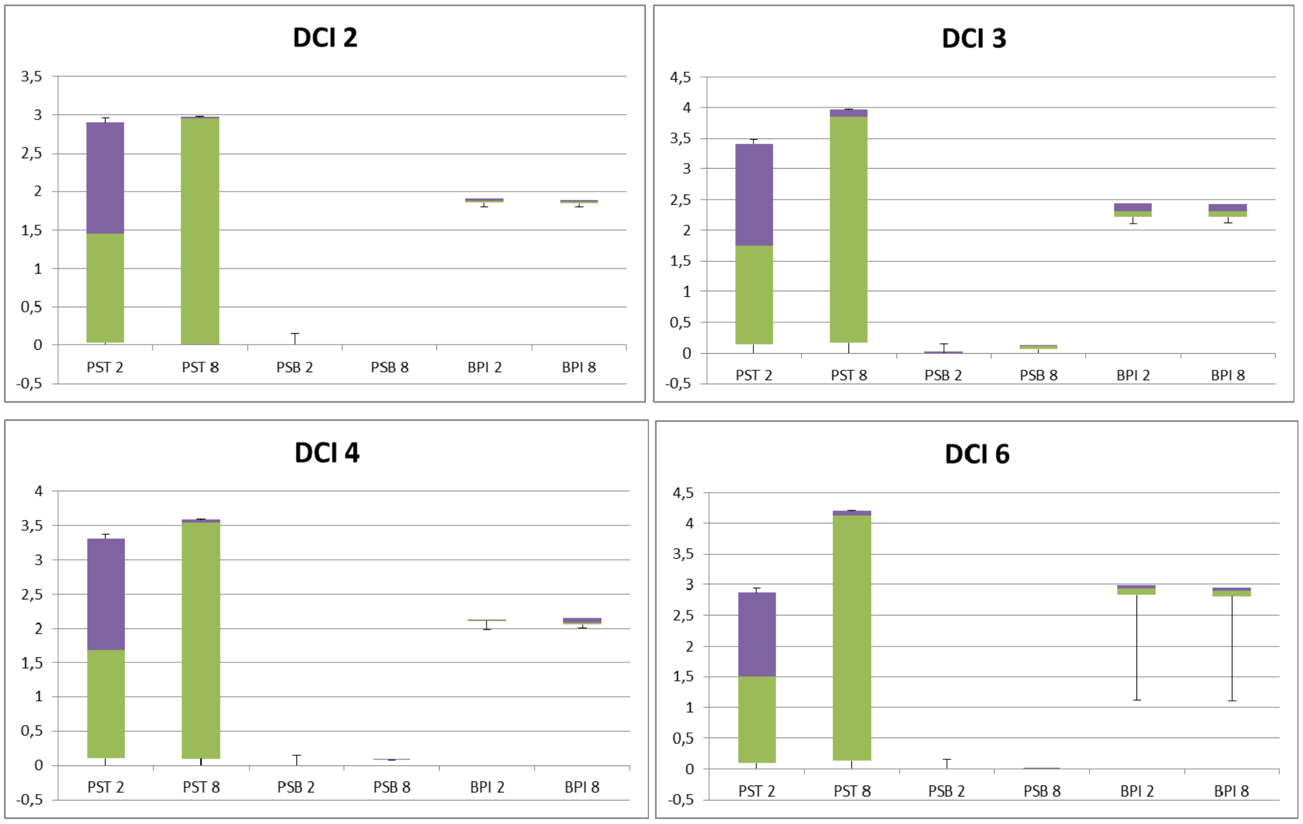


Figure 8. Distribution of PSNR drops (y-axis), with respect to CIBO strategy, for each method applied with 2 and 8 GOP (e.g. PST 2 is the method PST in 2GOP), to the four sequences tested in the experiment.

Finally, Figure 8 shows the distribution of the PSNR differences obtained after encoding the frames of the four tested sequences with 2 and 8 GOP, and with the three methods PST, PSB and BPI, with respect to CIBO. This shows that the major impact comes with PST that also has the largest dispersion of PSNR drops. The maximum drops are around 3dBs for 2-GOP and 4 dBs for 8-GOP. When increasing the GOP size, and given that we are using the same threshold slope to the whole group, the impact is larger, so that not only the maximum is larger, but the average drops are closer to the maximum in this situation. The increase of speedup with going to 8-GOP with PST was from 10% to 18% in average, so this means that using PST with large GOP sizes does not pay off.

The strong point of PSB is that the output images barely loose information. The maximum reported drop was of 0.18 dB for 8 GOP with sequence DCI 3. Recall that even no speedup was achieved in that situation. However, the largest speedup achieved of 20% for 8 GOP in DCI 2 brings a maximum of 0.01dB drop. For BPI, the drops of PSNR are more uniform than in PST, being almost constant between 2 and 3dB for all cases. However, for DCI 6 there are some minimum drops close to 1 dB. Given that BPI brings the largest speedups (up to 40%), this shows that it leads to a better efficiency – quality tradeoff.

7. CONCLUSION

In this paper, a new approach to accelerate the encoding of images with JPEG 2000 on GPUs is covered. Three methods to stop the encoding prematurely in EBCOT tier 1, by using feedback from tier 2, when this is executed in parallel to all code blocks, are introduced: Parallel Slope Threshold (PST), Parallel Slope-Byte (PSB) and Bit plane Parallel Interleaving (BPI). Actually, PSB feedbacks information from tier 2 of the very same encoded image, PST uses the information of the tier 2 applied to the previous frame in a sequence, and BPI estimates the truncation point coarsely.

In order to enable these methods on the SoTA designs of GPU J2K encoders, a new strategy is also proposed. This brings more efficiency thanks to the overlap of the execution of Context-Modelling and MQ-coder kernels when they are applied only per bit plane: *CM-MQ interleaved bit planes overlapping* (*CIBO*).

Experiments show that the CIBO strategy achieves speedups of up to 30% without a quality impact in the obtained images with respect to the starting point. PSB only becomes effective with sequences that are very rich in details, with speedups of up to 20% with GOP size of 8. Otherwise, the use of this method decelerates the execution of encoders by 4%. However, the maximum reported drop of PSNR is up to 0.18dB. PST, in turn, offers up to 20% of speedup in many more situations, even with GOP size of 2, but with the downside of reducing the quality of the obtained images by almost 3-4 dBs. Finally, BPI achieves boost speedups of up to 60%. However, it also has a bad impact in the obtained quality, of up to 2dB (but with low deviation; that is, the drop is always similar).

Future work will cover the improvement of the implementation of the mentioned methods, especially the BPI, which offered better speedups. One way will be better reproduce the behavior of CPI algorithm by further determining the code blocks to remain in the last visited bit plane. Moreover, BPI will be also studied as a parallel rate control method for JPEG 2000, by totally replacing PCRD-opt. For PST, smarter strategies can be considered in order to skip situations in which the propagated threshold slope is too high. For PSB, more efforts can be posed to reduce the overhead, so that it can be always applied in the encoder without impact on the performance when the stopping conditions do not hold.

ACKNOWLEDGEMENT

This work was developed during the fellowship tenure of *ERCIM Alain Bensoussan programme* at Fraunhofer IIS by M.A. Martínez-del-Amor.

REFERENCES

[1] Aulí-Llinàs, F., Serra-Sagristà, J., Moteagudo-Pereira, J.L. and Bartrina-Rapesta, J., “Efficient Rate Control for JPEG 2000 Coder and Decoder”. Proc. Data Compression Conference, 282-291 (2006).

[2] Aulí-Llinàs, F., “Model-based JPEG 2000 Rate Control Methods”. PhD Thesis, Universitat Autonoma de Barcelona, 2006.

[3] Aulí-Llinàs, F. and Serra-Sagristà, F., "Low Complexity JPEG 2000 Rate control through Reverse Subband Scanning Order and Coding Passes Concatenation". IEEE Signal Processing Letters, 14(4), 251-254 (2007).
[4] Boliek, M. (Ed.), "Information Technology – The JPEG 2000 image coding system: Part 1", ISO/IEC 15444-1, 2016.
[5] Bruns, V. "Acceleration of a JPEG 2000 Coder by Outsourcing Arithmetically Intensive Computations to a GPU". Master of Science Thesis, Tampere University of Technology (2008).
[6] Bruns, V. and Martínez-del-Amor, M.A., "Sample-parallel execution of EBCOT in fast mode", Proc. 2016 Picture Coding Symposium, 1-5 (2016).
[7] Bruns, V., Schmitt, A. and Sparenberg, H., "Accelerating a JPEG 2000 coder with CUDA", 45th JPEG Committee Meeting (2008).
[8] Bruns, V., Sparenberg, H. and Fößel, S., "Video decoder and method of decoding a sequence of pictures", patent US20130121421 A1 (2013).
[9] Chang, T-H., Chen, L-L., Lian, C-Jr., Chen, H-H., Chen, L-G., "Computation reduction Technique for Lossy JPEG 2000 Encoding through EBCOT Tier 2 Feedback Processing". Proc. International Conference on Image Processing 2002, Vol. 3, 85-88 (2002).
[10] Chang, T-H., Chen, L-L., Lian, C-Jr., Chen, H-H. and Chen, L-G., "Effective hardware-oriented technique for the rate control of JPEG 2000 encoding". Proc. International Symposium on Circuits and Systems 2003, Vol. 2, 684-687 (2003).
[11] Enfedaque, P., Aulí-Llinàs, F., and Moure, J.C., "Implementation of the DWT in a GPU through a Register-based strategy", IEEE Transactions on Parallel and Distributed Systems, 26(12), 3394-3406 (2015).
[12] Kirk, D. and Hwu, W-M., "Programming Massively Parallel Processors: A Hands-on Approach", MA (USA), (2010).
[13] Le, R., Bahar, I. and Mundy, J., "A novel parallel tier 1 coder for JPEG 2000 using GPUs". Proc. IEEE 9th Symposium on Application Specific Processors, 129-136 (2011).
[14] Matela, J., Rusnak, V., and Holub, P., "GPU-based Sample-Parallel Context modelling for EBCOT in JPEG 2000". Proc. 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Sciences, 77-84 (2010).
[15] Matela, J., Rusnak, V. and Holub, P., "Efficient JPEG 2000 EBCOT context modelling for massively parallel architectures", Proc.. IEEE Data Compression Conference 2011, 423-432 (2011).
[16] Matela, J., Srom, M. and Holub, P., "Low GPU Occupancy Approach to Fast Arithmetic Coding in JPEG 2000", Proc. 7th international conference on Mathematical and Engineering Methods in Computer Science, 136-145 (2011).
[17] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. and Phillips, J.C., "GPU computing". Proc. IEEE 96(5), 879–899 (2008).
[18] Richter, T., Simon, S., "Coding Strategies and Performance Analysis of GPU Accelerated Image Compression", Proc. 2013 Picture Coding Symposium, 125-128 (2013).
[19] Singh, S., Sharma, R.K. and Sharma M.K., "Efficient rate control approach for JPEG 2000 image coding". Journal of Electronic Imaging, 21(3), 033004 (2012).
[20] Taubman, D., "High Performance Scalable Image Compression with EBCOT", IEEE Transactions on Image Processing, 9(7), 1158-1170 (2000).
[21] Taubman, D., "Software architectures for JPEG 2000", Digital Signal Processing, Vol 1, 197-200 (2002).
[22] Taubman, D. and Marcellin, M., "JPEG 2000, Image Compression Fundamentals, Standards and Practice", Springer, 2002.
[23] Vikram, K.N., Vasudevan, V. and Srinivasan, S., "Rate-distortion estimation for fast JPEG 2000 compression at low bit-rates", Electronics Letters, 41(1), 2005.
[24] Wei, F., Cui, Q. and Li, Y., "Fine-Granular parallel EBCOT and optimization with CUDA for Digital Cinema image compression", Proc. IEEE International Conference on Multimedia and Expo, 1051-1054 (2012).
[25] Weiß, A., Heide, M., Papandreou, S. and Fürst, N., "CUJ2K: a JPEG 2000 encoder in CUDA", Software-Praktikum SS 2009, Universitäat Stuttgart (2009).
[26] Yeung, Y. M. and Au, O.C., "Efficient Rate Control for JPEG 2000 Image Coding", IEEE Transactions On Circuits And Systems For Video Technology, 15(3), 335-344 (2005), and in US patent US 7.970.224 B2 (2011).
[27] NVIDIA CUDA website, last accessed 2017. https://developer.nvidia.com/cuda-zone
[28] NVIDIA CUDA programming guide, last accessed 2017. http://docs.nvidia.com/cuda/cuda-c-programming-guide
[29] Khronos Group OpenCL website, last accessed 2017. https://www.khronos.org/opencl