Evaluation of GPU/CPU Co-Processing Models for JPEG 2000 Packetization

Volker Bruns, Miguel Á. Martínez-del-Amor, Heiko Sparenberg
Moving Picture Technologies
Fraunhofer Institute for Integrated Circuits IIS
Erlangen, Germany

***Abstract*—With the bottom-line goal of increasing the throughput of a GPU-accelerated JPEG 2000 encoder, this paper evaluates whether the post-compression rate control and packetization routines should be carried out on the CPU or on the GPU. Three co-processing models that differ in how the workload is split among the CPU and GPU are introduced. Both routines are discussed and algorithms for executing them in parallel are presented. Experimental results for compressing a detail-rich UHD sequence to 4 bits/sample indicate speed-ups of 200x for the rate control and 100x for the packetization compared to the single-threaded implementation in the commercial *Kakadu* library. These two routines executed on the CPU take 4x as long as all remaining coding steps on the GPU and therefore present a bottleneck. Even if the CPU bottleneck could be avoided with multi-threading, it is still beneficial to execute all coding steps on the GPU as this minimizes the required device-to-host transfer and thereby speeds up the critical path from 17.2 fps to 19.5 fps for 4 bits/sample and to 22.4 fps for 0.16 bits/sample.**

Keywords—JPEG 2000; PCRD-Opt.; Packetization; GPGPU

I. INTRODUCTION

JPEG 2000 is a still-image compression scheme standardized jointly by ISO and ITU [1]. It was selected by the *Society of Motion Picture & Television Engineers* (SMPTE) to be employed in both the *Digital Cinema Package* (DCP) format [2] and, more recently, in the mezzanine *Interoperable Master Format* (IMF) [3]. While cinema servers are equipped with FPGA-based hardware decoders, mastering stations and transcoders employed throughout the post-production workflow are more often software-based. The high computational complexity of JPEG 2000 is a challenge and can stand in the way of real-time processing. Therefore, much work has been invested into accelerating JPEG 2000 coders. One strategy has been to utilize graphics processing units (GPUs) which today have thousands of processing cores. The focus was spent mostly on both the *Discrete Wavelet Transform* (DWT) and the *Embedded Block Coder with Optimized Truncation* (EBCOT). Only little focus, however, has been devoted to executing the *Post-Compression Rate-Distortion Optimization* (PCRD-Opt.) [4] and packetization on a GPU. However, once all other coding blocks have been optimized, these operations can take up a significant portion of the overall coding time on a CPU, especially at high bit rates.

Figure 1 shows three possible co-processing models that differ in the points at which to switch the execution back from the GPU to the CPU. The research questions posed in this paper are: how can the *PCRD-Opt*. and packetization routines be efficiently computed on a GPU? Which co-processing model is the fastest?

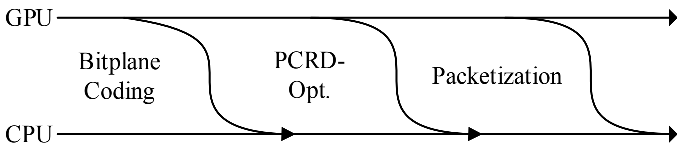


Fig. 1. GPU/CPU Co-Processing models for JPEG 2000 packetization

II. STATE OF THE ART

Considerable research has been devoted to re-formulating the individual JPEG 2000 coding-blocks for an execution on a GPU. Even before CUDA or OpenCL were released, Tenllado et al presented how to implement 2D DWTs efficiently on GPUs [5]. More recently it was shown how to leverage CUDA's register-shuffling intrinsics in a parallel DWT implementation [6]. The EBCOT algorithm poses a greater challenge as it does not inherently expose finely grained parallelism. A sample-parallel context modelling algorithm was presented by [7] and later in [8] and [9]. In [10] the authors evaluated if the selective arithmetic bypassing mode can yield a significant speed-up on a GPU. A CUDA-based open source encoder was released by the University Stuttgart, where PCRD-Opt. and packetization are executed on the CPU [11]. GPU-specifics implementation optimizations for the MQ-coder were presented in [13]. Aside from research related to parallel architectures, another related field is that of alternate rate-control algorithms, especially since the standard leaves some room for implementations here. Many algorithms have the goal of estimating prior to compression which passes will end up getting truncated. An excellent overview is given in [12]. In this paper, the *PCRD-Opt*.-based rate control is used. It was first proposed by Taubman in [4] and adopted as an example into the standard.

III. REVIEW OF JPEG 2000

Figure 2 shows a JPEG 2000 encoder's coding stages and data structures. After the color- and wavelet-transforms have been applied, each subband is first quantized and then split into non-overlapping code-blocks. Groups of spatially corresponding code-blocks from all subbands within a resolution level are grouped into *Precincts*. The most

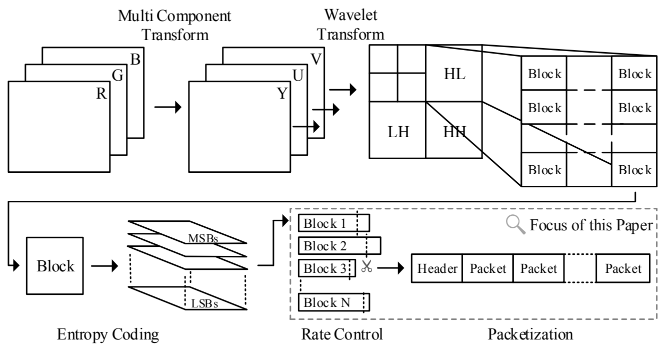


Fig. 2. JPEG 2000 Encoder Overview

predominant profiles employed in digital media production, the *DCI-*, *Broadcast-* and *IMF*-profiles [1][2], all define a code-block size of 32x32 samples and precinct size of 8x8 blocks (4x4 in the smallest resolution level). In EBCOT's first tier, each code-block is compressed independently into an embedded bit stream using a bit plane-wise scan pattern and a context adaptive binary arithmetic coder. To offer a finely grained bit stream embeddedness on a fractional bit plane level, each plane is further scanned in three passes: the first pass includes those samples that are not yet significant (magnitude of zero in decoder's point of view), but are likely to turn significant in this bit plane, because they have neighbors that are already significant and neighboring values in a subband tend to have similar magnitudes. The second pass refines those samples that have already turned significant in a previous bit plane. The last pass codes all remaining samples.

A. Post Compression Rate Control

In lossy coding, a maximum data rate defined as a property of the JPEG 2000 profile must not be exceeded. If the simulated code-stream is initially too large after block coding, EBCOT's second tier truncates the code-blocks' bit streams with the additional goal of retaining optimal quality. The standard proposes the *PCRD-Opt.* algorithm, which relies on the block encoder to produce a set of truncation points - one after each pass - along with an estimate of each point's slope on the code-block's rate-distortion plot as side information. It is then an optimization problem to find the set of truncation points n_j - one for every block's bit stream j - that yields the lowest overall distortion while still staying within the available data rate budget. The algorithm is explained in detail in the standard [1] and the original publication [4]. To give a brief overview here, it can be shown that only the truncation points that lie on the convex hull of their block's rate distortion plot are viable candidates and so all potential points for a single code-block have strictly decreasing slopes. It follows that a set of truncation points n_j^λ can be constructed by defining a slope threshold λ and then selecting, for each code-block, the truncation point that has a slope closest to, but not below the threshold. A bisection search will gradually lead to the best available slope threshold λ. It starts off by probing a threshold half way between the minimum and maximum possible slopes. The bit stream lengths and number of included passes for each code-block can be computed based on the current set of truncation points. Then the packet headers, tile-part headers and main headers are simulated and their lengths summed up to give the total codestream size. As long as this computed size exceeds (or is too far below) the specified budget, the procedure is repeated with a lower (higher) slope threshold.

B. Packetization

Structurally, a JPEG 2000 codestream consists of a set of markers that can be categorized into a main header and tile parts. A tile part is a sub-portion of the compressed image and its content depends on the progression order. When the major progression order is by color channel, each tile-part corresponds to a color channel. The main header comprises global properties and coding options required to decode and interpret the image. Tile-part headers contain further properties that can vary by tile-part, such as the bit depth. They are followed by the tile-part body, which is comprised of a set of packets. Each packet contains the compressed code-blocks from one particular precinct in one quality layer. Since the use of quality layers is prohibited in the *DCI-*, *Broadcast-* and *IMF*-profiles, we assume there is only a single quality layer going forward. Each packet header contains the code blocks' number of skipped all-zero bit planes, pass count and segment lengths. Since these values are likely to be similar for the code-blocks in one precinct, they are compressed with tag tree and comma codes. For improved random access, the tile-parts' lengths can be stated in the main header[1]. This is mandatory in the *DCI-*, *Broadcast-* and *IMF*-profiles.

IV. General Purpose Computing on GPUs

This section tries to give a very brief introduction into the concepts of GPU programming. A high-end GPU today has hundreds of cores and its architecture is designed to run thousands of lightweight threads. Tasks that can be broken down into many parallel sub-tasks are best suited to be computed on a GPU. A program consists of functions that are executed on the *device* (GPU) or on the *host* (CPU). A function that is run on the GPU is denoted as a *kernel* in this text. Input data to a kernel first needs to transferred into device memory and the results need to be transferred back into host memory. Memory transfers, kernels and host functions can be run in parallel. Popular toolkits are *OpenCL* [14], *NVIDIA CUDA* [15], *Apple Metal* [16] and *OpenMP* [17].

V. Proposed GPU/CPU Co-Processing Models

CPU-only - After the code-blocks' bit streams have been produced on the GPU the final *PCRD-Opt.* and packetization routines are executed by the CPU. From here on, this model is denoted as *CPU-only*. Since the GPU and CPU can operate in parallel, the packetization of frame *N* on the CPU are overlapped with the compression of frame *N+1* on the GPU. However, a packetization on the CPU entails that the required data needs to be transferred into the host memory first. Since the bit streams' lengths are not known ahead of time, they need to be produced into dedicated fixed-size slots of a large-enough memory block, so that bit streams are eventually interspersed with unused gaps. Either the gaps are included in the memory transfer or the bit streams need to be compacted first. Additionally, a set of per-code-block metadata is required for

[1]in the *Tile-Part Length in Main Header* (TLM)-marker

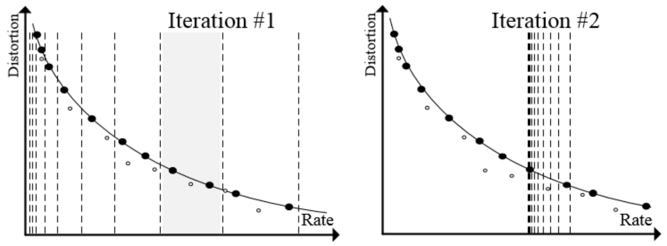
the packet header construction: the number of all-zero-bit planes, number of passes, pass lengths and pass slopes. Given that a UHD image without chroma-subsampling comprises around twenty thousand code-blocks and assuming up to 15 magnitude bit planes per block and two bytes per metadata value, this amounts to ~3.5 MB per image, which is often more than the compressed code-stream itself.

Hybrid – As part of the bisection search for the optimal slope threshold the entire codestream creation is simulated multiple times. By far the most computationally demanding sub-routine involved in this process is the packet header construction. In the *Hybrid* co-processing model, this search including the codestream simulation is executed on the GPU, and only the final packetization is left for the CPU. The pass lengths and slopes then do not have to be transferred, but instead only the total lengths of the truncated bit streams.

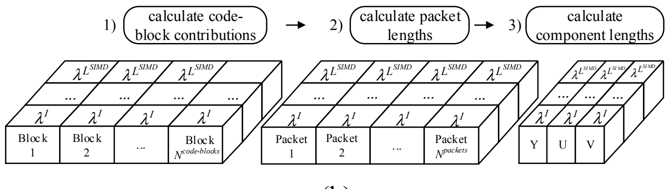
GPU-only – in this model, all processing steps are carried out on the GPU. A positive side effect is that the device-to-host transfer is decreased to the bare minimum: only the final codestream needs to be transferred, no additional per-code-block metadata.

VI. PCRD-Opt. on the GPU

In order to gain from a parallel architecture, the *PCRD-Opt.* algorithm is redefined to exploit two levels of parallelism. Figure 3 (a) shows the rate-distortion plot of a single code-block. The truncation points are laid out on a convex curve. Instead of probing only one slope threshold at a time, multiple slopes (dashed lines) that are equally spaced apart in the current search window are probed in parallel. Iteratively, the search window is narrowed until a slope threshold is found that yields a codestream size sufficiently close to the specified maximum size. A natural choice for the number of simultaneously probed thresholds is the GPU architecture's SIMD-group size L^{SIMD}. Tests with a detail-rich UHD sequence showed that for 32 parallel probes, two iterations are usually



(a)


1) calculate code-block contributions
2) calculate packet lengths
3) calculate component lengths
$\lambda^{L^{SIMD}}$
λ^1
...
Block 1
Block 2
Block $N^{code\text{-}blocks}$
Packet 1
Packet 2
Packet $N^{packets}$
Y
U
V

(b)

Fig. 3 Block diagram for proposed *PCRD-Opt.* algorithm with two dimensions of parallelism: (a) probe multiple slope thresholds in parallel and (b) process code-blocks and packets in parallel in a two-level reduction

sufficient to get within 5% of the desired codestream size. When probing only one threshold at a time it requires nine iterations to get within 5% and sixteen iterations to get as close as possible.

The second level of parallelism is to process code-blocks and packets in parallel. Figure 3 (b) shows how the overall codestream size is calculated in a two-level reduction. A first kernel 1) computes per code-block and probed slope threshold the number of passes and truncated bit stream length. To do that it counts the passes up to the truncation point and sums up their compressed lengths. Subsequently, a second kernel 2) with one thread per packet and threshold computes the packet lengths by accumulating the bit stream lengths of all code-blocks contained in that packet (up to 192) and adding to that the length of the simulated packet header. Finally, packet sizes are accumulated by color channel since for *DCI*-profiles the maximum size per channel is also constrained. The fixed overhead for the tile-part headers also counts towards the per-channel limit and needs to be accounted for. Finally, the total codestream size is the sum of all channels plus the fixed overhead of the main header.

VII. Packetization on the GPU

The codestream anatomy can be described by a table of cells, *C,* where each cell C_i^a represents a section of the final codestream that is constructed independently, *a* being the corresponding type abbreviation and *i* the occurrence of that type. Table I lists all cell types.

TABLE I. Codestream Anatomy Cell Types

Type	*Abbr.*	*Occurrences*	*Length*
Main Header	*MH*	1	fixed
Tile Part Length	*TLM*	1	fixed
Start of Tile and Start of Data	*SOT*	N^{TP}	fixed
Packet Header	*PH*	$N^{packets}$	dynamic
Packet Body Code-Block	*PB*	$N^{code\text{-}blocks}$	dynamic
End of Codestream	*EOC*	1	fixed

The codestream anatomy stays constant for an entire image sequence[2] and serves as a convenient indirection in order to hide the progression order from the GPU-kernels. The construction of each codestream then comprises three phases (as depicted in figure 4):

1. Populate a table $L[C_i^a]$ with the cells' exact sizes in bytes
2. Prefix-Sum over the cell lengths $L[C_i^a]$ in order to obtain a table $\Delta[C_i^a]$ containing the cells' offsets in the final codestream
3. Construct the cells' contents and write them to the appropriate positions, obtaining the final codestream in GPU memory

[2] JPEG 2000 is a still image format, but this paper focuses on movies

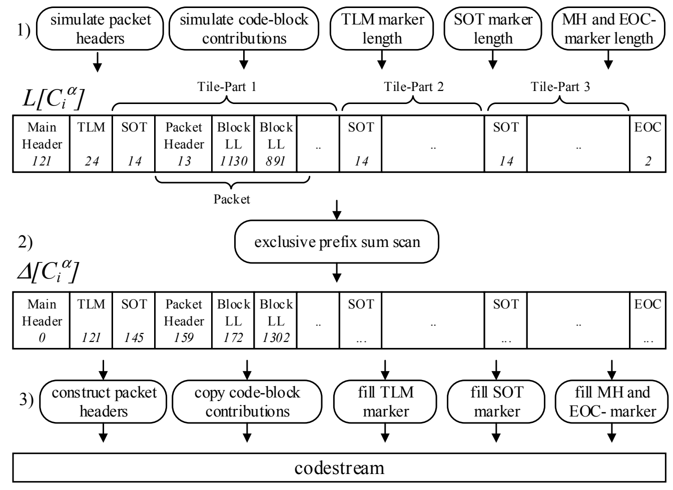


Fig. 4. Block diagram for proposed packetization algorithm

Given the number of tile-parts, N^{TP}, number of packets, $N^{packets}$,, and the number of code-blocks, $N^{code\text{-}blocks}$, the number of cells, C_N, is then

$$C_N = 1(C^{MH}) + 1(C^{TLM}) + N^{TP} + N^{packets} + N^{code\text{-}blocks} + 1(C^{EOC})$$

In the first phase, a table $L[C_i^\alpha]$, containing the lengths in bytes of each cell, is populated. $L[C^{MH}]$ comprises the combined lengths of the main header markers[3]. Instead of transferring all required information to the GPU and assembling the main header there, the *Kakadu* library [21] was used to build this cell. As long as the comments do not change, this cell's size remains fixed for an entire image sequence. The size of $L[C^{TLM}]$ depends only on N^{TP} and the precision chosen to store the tile-parts' lengths and tile-part indices. All these variables stay constant for an entire image sequence, and thus $L[C^{TLM}]$ remains fixed. The SOT and SOD markers have a fixed size per specification.

Let b_j denote the embedded bit stream for code-block B_j and b_j^z the prefix of b_j up to and including pass z. λ_j^z defines the slope for pass z of code-block B_j, and L_j^z the length of b_j^z. All $L[C_p^{PB}]$ can be filled with the lengths L_j^z for all code-blocks B_j that belong to precinct p. Thread coordinates can be conveniently mapped to cells C_p^{PB} via a look-up-table with $N^{code\text{-}blocks}$ entries. The mapping of code-blocks to precincts can be established by a combination of three look-up tables: let $N_p^{x,y}$ denote the number of code-blocks (in rows y and columns x), in precinct p and ΔB_p the position of the precinct's first code-block $x=0$, $y=0$, in a list of all code-blocks sorted by packet membership $\hat{B}$. The code-blocks belonging to precinct b are then $\hat{B}[\Delta B_p]$, …, $\hat{B}[\Delta B_p + N_p^{x,y} - 1]$.

All $L[C_p^{PH}]$ can be computed by simulating the header construction. In fact, the packet headers have already been simulated during the last *PCRD-Opt.* iteration, so an implementation can choose to store the packet header sizes and reuse the set for the selected slope threshold λ here.

[3] SOC, SIZ, COD, QCD and COM

Once all cells in the table $L[C_i^\alpha]$ have been filled, the cells' offset in bytes, $\Delta[C_i^\alpha]$, in the final codestream can be computed by executing a device-wide exclusive prefix-sum [18]

$$\Delta[C_i] = ExclusivePrefixSum\ (L[C_i^\alpha])$$

Parallel implementations are available through toolboxes like *thrust* or *cub* [19][20]. The final codestream size (excluding the length of the two-byte EOC-marker), $L^{codestream}$, can now be looked up in the last element of the table Δ:

$$L^{codestream} = \Delta[C_{N-1}^{EOC}] + L[C_{N-1}^{EOC}]$$

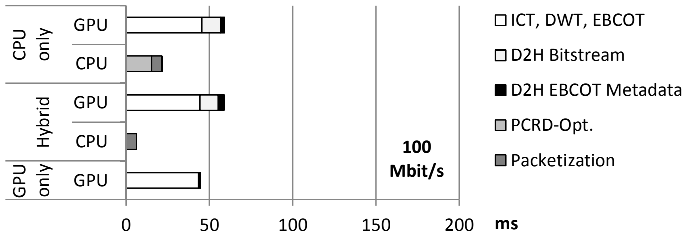
In the third phase, the codestream cells are filled with data. A memory block in GPU memory of size $L^{codestream}$ needs to be reserved, so that the individual cells can be directly written to their final locations. All cells except those of type C^{MH} and C^{EOC} are filled on the GPU. The authors opted to split the tasks into four kernels – one for each of the remaining cell types. They can operate in parallel.

The C^{TLM} cell contains the lengths, L_k^{TP}, of each tile part TP_k. Similarly, C_k^{SOT} cells contain the length and index of their respective tile parts. The length of a tile part can be obtained from $\Delta[C_i^\alpha]$:

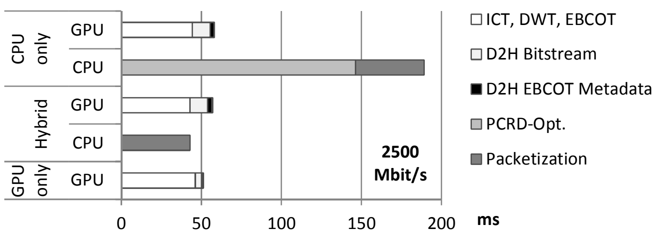
$$L_k^{TP} = \begin{cases} \Delta[C_{k+1}^{SOT}] - \Delta[C_k^{SOT}], & 0 \le k < N^{TP} - 1 \\ \Delta[C^{EOC}] - \Delta[C_k^{SOT}], & k = N^{TP} - 1 \end{cases}$$

The C^{PH} cells contain the side information required by the decoder to correctly interpret the compressed code-blocks. The construction of tag-trees and comma-codes is analog to an implementation for a CPU. Since packet headers are always padded to byte boundaries, parallel threads will not write to the same byte address. Therefore it is not necessary to write out bits with the atomic-or intrinsic and initialize the target cell with zeros. Instead, the GPU implementation can collect emitted bits in an intermediate register in order to reduce expensive accesses to global memory. The kernel for filling C^{PH} cells is almost identical to the simulation kernel used to compute $L[C^{PH}]$. They only differ in that the latter merely counts emitted bits, while the former actually writes them to memory.

Filling in the C^{PB} cells is essentially a device-to-device copy operation as the individual bit streams b_j are already located in device memory. However, they are interspersed with gaps and yet untruncated. The task at hand is to copy each bit stream's prefix b_j^z for the previously determined slope threshold λ into the corresponding cells C_j^{PB}. A naive approach would be to have one thread per code-block copying the bit stream prefix byte by byte. A faster, because more parallel, solution is to assign multiple threads to each code-block and have them copy the bit stream prefix chunk wise. Again, a good choice for the amount of threads per code-block is the architecture's SIMD-group size L^{SIMD}. Considering the size of a cache line $L^{cache\text{-}line}$, each thread should then copy $N = L^{cache\text{-}line} / L^{SIMD}$ bytes per iteration until L_j^z bytes have been copied by the SIMD group j. The destination address in global memory is defined by the codestream offset and therefore alignment



(a)



(b)

Fig. 5. Encoder runtimes, for VQEG Crowd Run UHD sequence, left: 0.16 bits/sample (100 Mbit/s @ 24 fps), right: 4.2 bits/sample (2500 Mbit/s @ 24 fps), DCI-compliant settings (except bit rate). GPU: GeForce 1080, CPU: Kakadu v6.4 single-threaded on Intel i7 3960X

constraints cannot be honored, but the source bit streams b_j should be placed at addresses that are a multiple of $L^{cache\text{-}line}$.

After all four kernels have finished populating their respective cells the codestream can be transferred into host memory where the remaining two cells C^{MH} and C^{EOC} can be filled.

VIII. Experimental Results

Figure 5 shows a comparison of the three evaluated co-processing models. Since the GPU and CPU can work on different frames in parallel, their bars are laid out side-by-side. The GPU implementation uses the NVIDIA CUDA framework [15]. The CPU implementation, *Kakadu v6.4*, does not employ multi-threading for either *PCRD-Opt.* or packetization since its architecture is optimized for streamlined line-by-line encoding, where the image does not have to be entirely loaded into memory at any point [21]. For this discussion, we will assume the measured CPU-runtimes can be divided by the number of CPU cores without overhead.

In the low-bit rate scenario, Figure 5 (a), executing both PC*RD-Opt.* and packetization on the CPU does not present a bottleneck as the GPU-portion of the encoder always takes longer. However, executing all stages on the GPU leads to an overall acceleration from 59 ms (16.9 fps) to 44 ms (22.4 fps). This gain stems from two reasons: 1) in order to run *PCRD-Opt.* on the CPU, the pass lengths and slopes of all code-blocks have to be transferred from the GPU to the CPU, which is no longer necessary in the *GPU-only* mode; and 2) the device-to-host transfer of the compressed data is also faster, because the portions of the bit streams discarded by the rate control are not included anymore and the bit streams are arranged back to back without gaps.

Figure 5 (b) shows the runtimes for the same test sequence, when compressed less heavily. Without multi-threading, the CPU-portion takes around four times as long as the GPU portion and thus poses a bottleneck. In this scenario, a multi-threaded CPU implementation running on at least four cores would be required to remove the bottleneck. Though this was neither examined nor the goal of this work, the parallel algorithms presented in this paper should also be applicable to a CPU implementation. However, the finely grained parallelism should be grouped into coarser work units, reducing the number of threads that need to be spawned to an amount that is closer to the number of virtual CPU cores present in modern CPUs.

The GPU portion's runtime in the *Hybrid* model stays almost unchanged as the search for the best threshold takes only 0.7 ms on the GPU compared to 146 ms on the CPU. Creating the codestream entirely on the GPU takes only an extra half a millisecond. Reducing the device-to-host data transfers from 13.7 ms to 3.9 ms, the *GPU-only* model yields an overall gain from 17.2 fps to 19.5 fps, assuming the CPU-portion were sufficiently sped up by multi-threading. With respect to the single-threaded implementation evaluated here, the encoder's throughput is increased from 5.3 fps to 19.5 fps.

IX. Conclusions

To answer the research questions, the *PCRD-Opt.* algorithm can be efficiently executed on a GPU, where parallelism in two dimensions can be exploited: (1) probe multiple truncation point sets in parallel and (2) process code-blocks and packets in parallel. At large bit rates, *PCRD-Opt.* executed on a CPU by a single-threaded implementation was shown to be the bottleneck in an otherwise entirely GPU-optimized JPEG 2000 encoder. It runs about two orders of magnitudes faster when run on a GPU.

Furthermore, a parallel table-driven algorithm for executing the packetization on a GPU was presented. It runs 80x faster compared to the single-threaded CPU implementation, but more importantly it leads to an overall speed-up of the encoder, because the amount of data that needs to be transferred into host memory is minimized. Table 2 lists a condensed summary of the findings.

TABLE II. Comparison of Results For Co-Processing Models

Model	*Discussion*
CPU-only	*PCRD-Opt.* requires multi-threading at high bit-rates or else poses bottleneck. Packetization uncritical, when done in parallel with GPU, but then leads to higher latency.
Hybrid	*PCRD-Opt.* 200x faster (4bits/sample). Discarded portions of bitstream can be omitted from transfer, but requires that bit streams are compacted first. Per-Code-Block metadata still needs to be transferred for packetization on CPU.
GPU-only	Packetization 100x faster (4 bits/sample) on GPU. Transfer is minimized to bare codestream, which leads to overall speed-up. Lower latency, because no coding steps executed on CPU anymore.

In conclusion, creating the codestream entirely on the GPU pays off. The time saved in the transfer outweighs the extra time it takes to run the rate control and packetization on the

GPU. This effect weighs in more heavily when compressing high-entropy images to low bit-rates, but could be alleviated by anticipating which passes will end up getting discarded by the rate control and not coding them in the first place.

When compressing to high bit rates, on the other hand, it was shown that rate control and packetization would need to be multi-threaded when run on the CPU or otherwise the CPU portion will present a bottleneck. Creating the codestream entirely on the GPU has the positive side effect that the overall runtime is decreased, albeit less significantly than at low bit rates, since the gain of transferring only the already truncated bit streams weighs in less heavily. Additionally, the latency is reduced, because the GPU and CPU do not need to operate on different frames in parallel anymore. Finally, the CPU is left free to do other tasks.

REFERENCES

[1] M. Boliek (Ed.), ”Information Technology - The JPEG2000 image coding system: Part 1”, ISO/IEC 15444-1, 2016

[2] A. Bilgin, M.W. Marcellin, “JPEG2000 for Digital Cinema”, IEEE International Symposium on Circuits and Systems, pp. 3878-3881, May 2006

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

[4] D. Taubman, “High performance scalable image compression with EBCOT”, IEEE Transactions on Image Processing, Vol. 9 No. 7, pp. 1151-1170, 2000

[5] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, F. Tirado, “Parallel implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter-Bank versus Lifting.”, IEEE Transactions on Parallel and Distributed Systems, Vol. 19, No. 3, pp. 299-310, March 2008

[6] P. Enfedaque, F. Auli-Llinas, J. C. Moure, “Implementatin of the DWT in a GPU through a Register-based Strategy”, IEEE Transation on parallel and distributed Systems, Vol. 26, No. 12, pp.3394-3406, 2015

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

[8] R. Le, I.R. Bahar and J.L. Mundy, “A novel parallel Tier-1 coder for JPEG2000 using GPUs”, IEEE 9th Symposium on Application Specific Processors, pp. 129-136, Jun 2011

[9] F. Wei, Q. Cui, Y. Li, “Fine-Granular Parallel EBCOT and Optimization with CUDA for Digital Cinema Image Compression”, IEEE Int. Conf. on Multimedia and Expo (ICME), Jul 2015

[10] V. Bruns and M. A. Martínez-del-Amor, “Sample-Parallel Execution of EBCOT in Fast Mode”, 32nd Picture Coding Symposium, Nuremberg, Dec. 2016

[11] A. Weiß, M. Heide, S. Papandreou, N. Fürst (2009, Sept 20), ”CUJ2K: a JPEG2000 encoder in CUDA” [Online], Available: http://sourceforge.net/projects/cuj2k/files/cuj2k-documentation.pdf/download

[12] F. Auli-Llinas, “Model based JPEG 2000 Rate Control Methods”, Ph.D. dissertation, Departament d’Enginyeria de la Informacio I de les Comunicacions, Universitat Autonoma de Barcelona, Barcelona, Spain, Oct 2006

[13] J. Matela, M. Srom, P. Holub, ”Low GPU Occupancy Approach to Fast Arithmetic Coding in JPEG2000”, 7th Int. Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS’11), Vol. 71, pp. 136-145, Oct. 2011

[14] Khronos Group (Aug 2009), OpenCL [Online], Available: http://www.khronos.org/opencl

[15] NVIDIA (Jun 2007), CUDA [Online], Available: http://www.nvidia.com/object/cuda_home_new.html

[16] Apple Inc. (Jun 2014), Metal [Online], Available: https://developer.apple.com/metal

[17] OpenMP ARB (Oct 1997), OpenMP [Online], Available: http://www.openmp.org

[18] G.E. Blelloch, “Scans as Primitive Parallel Operations”, IEEE Transactions on Computers, Vol. 38, No. 11, pp. 1526-1538, Nov 1989

[19] J. Hoberock, N. Bell, Thrust (May 2009) - Parallel Algorithms Library [Online], Available: https://thrust.github.io/

[20] D. Merrill, NVIDIA Research (2011), cub – CUDA Unbound [Online], Available: https://nvlabs.github.io/cub/

[21] D. Taubman (2014), Kakadu Software [Online], Available: http://kakadusoftware.com/