Taylor & Francis
Taylor & Francis Group

Automatika

Journal for Control, Measurement, Electronics, Computing and Communications

ISSN: 0005-1144 (Print) 1848-3380 (Online) Journal homepage: https://www.tandfonline.com/loi/taut20

Performance engineering for HEVC transform and quantization kernel on GPUs

Mate Čobrnić, Alen Duspara, Leon Dragić, Igor Piljić & Mario Kovač

To cite this article: Mate Čobrnić, Alen Duspara, Leon Dragić, Igor Piljić & Mario Kovač (2020) Performance engineering for HEVC transform and quantization kernel on GPUs, Automatika, 61:3, 325-333, DOI: 10.1080/00051144.2020.1752046

To link to this article: https://doi.org/10.1080/00051144.2020.1752046

Published online: 16 Apr 2020.

AUTOMATIKA
2020, VOL. 61, NO. 3, 325–333
https://doi.org/10.1080/00051144.2020.1752046

REGULAR PAPER

OPEN ACCESS Check for updates

Performance engineering for HEVC transform and quantization kernel on GPUs

Mate Čobrnić, Alen Duspara, Leon Dragić, Igor Piljić and Mario Kovač

Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia

ABSTRACT

Continuous growth of video traffic and video services, especially in the field of high resolution and high-quality video content, places heavy demands on video coding and its implementations. High Efficiency Video Coding (HEVC) standard doubles the compression efficiency of its predecessor H.264/AVC at the cost of high computational complexity. To address those computing issues high-performance video processing takes advantage of heterogeneous multiprocessor platforms. In this paper, we present a highly performance-optimized HEVC transform and quantization kernel with all-zero-block (AZB) identification designed for execution on a Graphics Processor Unit (GPU). Performance optimization strategy involved all three aspects of parallel design, exposing as much of the application's intrinsic parallelism as possible, exploitation of high throughput memory and efficient instruction usage. It combines efficient mapping of transform blocks to thread-blocks and efficient vectorized access patterns to shared memory for all transform sizes supported in the standard. Two different GPUs of the same architecture were used to evaluate proposed implementation. Achieved processing times are 6.03 and 23.94 ms for DCI 4K and 8K Full Format, respectively. Speedup factors compared to CPU, cuBLAS and AVX2 implementations are up to 80, 19 and 4 times respectively. Proposed implementation outperforms previous work 1.22 times.

ARTICLE HISTORY
Received 20 January 2020
Accepted 23 March 2020

KEYWORDS
Integer discrete cosine transform (DCT); high efficiency video coding (HEVC); Graphics processor unit (GPU); matrix multiplication; compute unified device architecture (CUDA)

Introduction

The share of video traffic in global Internet traffic will undergo significant growth, from 75 percent in 2017 to predicted 82 percent by 2022 [1]. It is estimated that ultra-high definition (UHD) or 4K video will account for 22 percent of that amount. To support fast delivery and inexpensive storage of video data of such a huge size, video compression with high coding efficiency is required. HEVC [2] is a state-of-the-art video coding standard that doubled compression efficiency compared to its predecessor Advanced Video Coding (AVC). This accomplishment was made at the cost of an increase in computational complexity of video encoding and decoding.

HEVC standard is devised for a hybrid video coding where transform, scaling, and quantization (TQ) are contained in an important functional block. In modern hybrid video coding systems that block follows the motion-compensated prediction and precedes entropy coding. Regardless of how effectively the preceding prediction process is exercised, there is typically a remaining prediction error residual signal that has to be further processed. In overall encoding time, TQ and its inverse process take about 25 percent for the all-intra and 11 percent for the random-access configurations in HM 8.0 encoder [3]. As a new feature, when compared to prior standards, HEVC introduced additional transform block (TB) sizes to make encoding of video files of 4K and 8K resolutions more efficient; replaced real-valued discrete cosine transform (DCT) with integer DCT to ensure device interoperability and avoid encoder-decoder mismatch; simplified (de)quantization process by turning it into scalar multiplication (division). Efficient implementation in software exploiting SIMD capabilities and parallel processing were set as design goals during HEVC TQ development [4].

To cope with increased computational complexity, which is especially challenging for the design of real-time applications, advanced computing architectures are required. Heterogenous multiprocessor computing architectures are one possible solution in such cases. There, the application is portioned in a way that tasks are distributed among coprocessors depending on their specialized processing capabilities. Currently, the most common heterogeneous systems are made of a multicore Central Processing Unit (CPU) and GPU [5]. Serial portions of applications are run on the former, while data-parallel, compute-intensive portions are offloaded to the latter. Programming paradigms for these two architectures are different, which has to be considered during the application design. In addition to GPUs, Field Programmable Gate Array (FPGA) is another possible hardware node in heterogeneous

CONTACT Mate Čobrnić mate.cobrnic@fer.hr

architectures. Very high energy efficiency, reconfigurability, and low latency put them ahead of other hardware solutions for compute-intensive signal-processing computations.

In this paper, we present a highly parallel HEVC transform and quantization kernel with AZB identification designed for execution on a GPU. Memory bandwidth, instruction throughput, and on-chip memory allocation were properly balanced to achieve efficient execution and high resource utilization. Different optimization techniques, from overlapping data transfers with computation to fine-tuning arithmetic operations sequences, were incorporated to reach high performance. By exploiting the Compute Unified Device Architecture (CUDA) programming model [6] experimental results exhibited processing times 6.03–8.88 ms for the Digital Cinema Initiatives (DCI) 4K video depending on block partitioning granularity for transform coding. Speedup factors which were obtained compared to the CPU, manufacturer's CUDA Basic Linear Algebra Subroutines (cuBLAS) and Advanced Vector Extensions (AVX2) implementations are 80, 19 and 4 respectively. In comparison with related work 1.22 times lower processing time is achieved.

The rest of the paper is organized as follows: second Section describes related work and motivation for this research. In Section 3 HEVC transform and quantization operations, which are the base for parallelization, are presented. In Section 4 the proposed GPU parallelization methods are described and justified using a mid-end GPU. Implementation results are discussed and evaluated using a high-end GPU of the same architecture in Section 5. Section 6 presents the conclusion of the paper.

Related work and motivation

Previous research in this field which focused on CPU + GPU heterogeneous platforms mainly tackled motion estimation as a functional block with the highest computational load [7,8]. A step further was taken with massive parallelization in [9] and all functional blocks of HEVC decoder, except entropy decoder, were ported to GPU. Quite the contrary, solutions with TQ acceleration using FPGAs rather than the GPU [10] are much more represented as a research topic.

Regarding the migration of HEVC TQ to the GPU in [11] two tables, one describing transform unit (TU) partitioning and the other quantization parameter (QP) value storing, together with the mapping algorithm at the CTU level, were proposed to achieve efficient implementation. In [12] authors dealt with a heterogeneous system for HEVC encoder where motion-compensated prediction processing already resides at the GPU side and additionally the TQ has to be ported there. Parallel TU address list construction and coefficient packing were proposed to get high processing speed. Benchmarking the performance of four different HEVC 2D transform kernel designs against its industrial solution, which combine assembly and AVX2 instructions, was carried out in [13] without conducting performance optimizations. In [14] the highly optimized parallel implementation of the HEVC dequantization and inverse transform is presented using the unified programming model for the CPU + GPU heterogeneous system.

Previously mentioned works mainly focus on the integration aspect of GPU implementation of HEVC TQ and do not reveal in detail the design of kernel function and optimizations which are made to efficiently balance between GPU resources, sum of registers, allocated on-chip memory per thread-block, number of threads per multiprocessor and global memory bandwidth. That is preventing proper validation of their performance gains and gaps. Additionally, it has to be mentioned that fair comparison using only TQ processing time is not feasible. In [11], GPU acceleration is done at the CTU block level and in [12] and [14] it is done at the frame level with transform blocks previously grouped for GPU acceleration. The latter approach allows much better use of GPU parallelism.

The highest workload stage during HEVC TQ is the 2D forward transform which is mathematically realized as double matrix multiplication. Many guidelines exist with general optimizations principles exampled in matrix multiplication and other basic linear algebra subroutines (BLAS) [15–17]. They mainly deal with the multiplication of two large size (one or both dimensions) matrices which are tiled to small size subblocks e.g. 16×16 and distributed among GPU thread-blocks. Values from input matrices are repeatedly loaded in subblocks and the resulting subblock is computed as the sum of products of these subblocks. HEVC TQ operates with batches of small size matrices i.e. TBs where the tiling approach would degrade performance. The efficient mapping of TBs and dot product computations to various components of the GPU subsystem with the adaptation of known performance optimization techniques was set as the main design objective. The final proposal presents a systematic and efficient solution for the multithreaded GPU kernel function for all supported transform sizes in HEVC.

HEVC transform and quantization TQ

HEVC TQ processes the TBs coming from the residual quadtree (RQT) structure. The output consists of quantized transform coefficients (levels). The process consists of three stages: the 2D integer DCT transform, quantization, and AZB identification. The standard specifies 4-, 8-, 16- and 32-point transform.

HEVC transform of size $N \times N, N \in \{4, 8, 16, 32\}$ is given by

$$Y = D \times X \times D^T \quad (1)$$

where D is the HEVC transform matrix with predefined values, X is the residual matrix and D^T is a transpose of the transform matrix. All matrices are of size $N \times N$. HEVC transform matrix is scaled by a predefined factor compared to orthonormal DCT but it keeps most of the properties which are useful for compression efficiency and efficient implementation [4]. To retain the norm of the residual block, additional scaling is needed after each step in the first process stage, each of two integer 1D transforms, and in the second quantization stage.

Quantization is a scalar operation. Transform coefficients are quantized by a QP value. HEVC also supports frequency-dependent quantization. Low frequency coefficients can be quantized with finer quantization step to adjust to the human visual system.

AZB identification is the last stage in the process. The information if all levels coming out from one TB are zero is used to set the value of the syntax element coded block flag (CBF). This helps to reduce the number of bits to be transmitted.

Decoder is built-in in HEVC encoder since the frame has to be fully decoded to be used as a reference in the estimation and prediction process for subsequent frames. As the first step in the reconstruction of the frame, quantized transform coefficients are dequantized and inverse transform is applied thereafter. Computation complexity of dequantization and inverse transform (DQIT) is equivalent or lower to that of the TQ process. Double matrix multiplication of input blocks with HEVC transform matrix and its transpose with intermediate scaling, the same number of scalar multiplications and bitwise shifts appear in both functional blocks. AZB identification is part of the TQ process only.

Proposed GPU parallelization methods

GPUs are powerful arithmetic engines suited to run thousands of threads in parallel. To obtain the best performance from a GPU and to achieve suitable low video latency, GPU processing is done at the frame level or frame's horizontal segment level. Compared with pure CPU implementation, heterogeneous accelerator-based architecture could potentially suffer from a large overhead in data transfer between the CPU (host) and the GPU (device). As shown later, that overhead could reach up to 26% of overall processing time in case of a mid-end GPU and 73% in case of a high-end GPU. Residual data blocks are copied asynchronously in groups from host to device as shown in Figure 1. One group includes all the blocks of the same size. Data transfers are optimized according to [18]. Once computed, blocks of quantized coefficients and their zero markings are returned to the host. The device receives data into its global memory and transfers them from there to streaming multiprocessors (SM). Transform matrices of all sizes are written to the global memory in advance. To reach the full SM utilization and decrease thread creation and destruction cost, the grid-stride loop technique [19] was employed for kernel design. Accesses to the device global memory are done in a coalesced way to maximize the effective memory bandwidth. A thread-block accesses a group of residual blocks, block by block in the raster scan order where the data belonging to a residual block are accessed row by row.

Residual image is dynamically partitioned into TBs whose size adapts to spatial and frequency characteristics of the corresponding image area. Transform operations performed with TBs of different sizes are different and TBs are grouped before the transfer to the device. Gathering the same sized TBs can be done by resuming our previous work [20]. The prediction stage can be modified in a way that residual values are stored to memory location for particular block size. The index of the addressing information for the block is written into the TU data structure.

Since matrix-matrix multiplications are the most compute-intensive part in the process, shared memory, located on the chip, is used to hold the input data to reduce global memory accesses. To achieve high bandwidth, it is divided into banks. Each bank provides a bandwidth of 64 bits per clock cycle. Buffered data, byte-aligned residuals and intermediate transform coefficients are each 16-bit wide. For efficient communication with the memory, vectorized memory access is used and four data are retrieved from the memory per bank per transaction. Bank size is reconfigured to 64 bits so that in a thread-block successive 64-bit vectors are assigned to successive banks.

Shared memory is additionally exploited as the single access point for AZB identification. Since mapping residual block to thread-block is $n : 1 (n \in \mathbb{N})$, an intermediate array of Booleans of length n is allocated in the shared memory. Group of threads in a thread-block that computes levels in a TB will initiate a write request to the same corresponding array element if the non-zero level was identified in its vector. As the last step, each array element is tested by a single thread in the group to set a related array element in the output AZB array in global memory. To ensure correct results of values in both arrays, the threads are synchronized two times. First time after the initialization of arrays in shared and global memory and the second time after thread wrote a corresponding value to the matching residual block's array element. The shared memory access pattern which enables maximum throughput of shared memory will be presented below. If the AZB identification stage is skipped in the process kernel, the

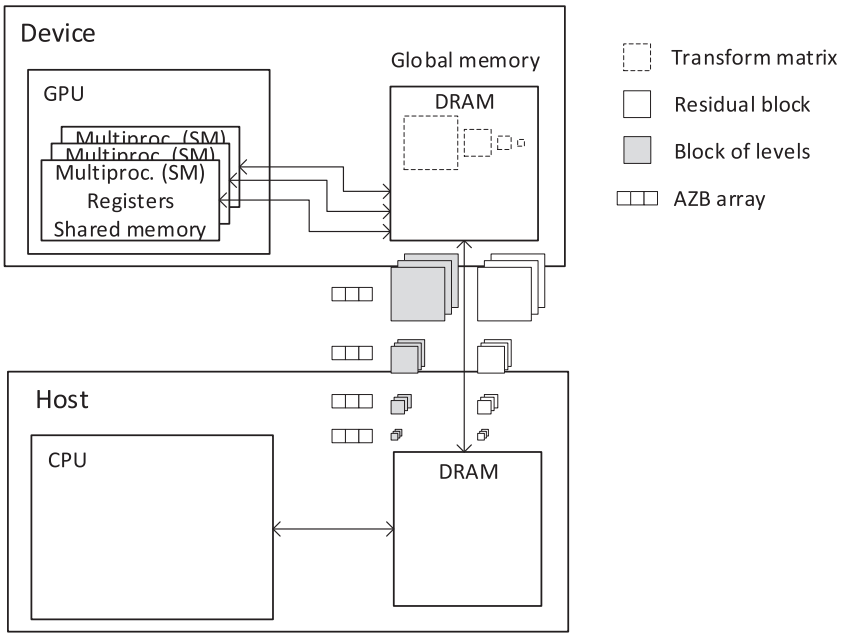

Figure 1. Data transfers between host and device.

Table 1. Processing times for different block mappings for 4×4 transform blocks.

TB size	No. of blocks	Block mapping	Kernel execution time [ms]	Regs per thread	Shared memory per block (bytes)	Occupancy [%]
4 × 4	829,440 (DCI 4K)	1:1	49.55	28	112	25
		64:1	5.02	32	4160	100
		128:1	5.60	32	8320	100
		192:1	7.59	31	12,480	75
		256:1	8.71	32	16,640	100

processing time is shortened 4% or 2% depending on the used GPU type.

Mapping transform blocks to thread-blocks

For performance optimized HEVC TQ with maximized parallel execution, the maximum utilization of available device resources has to be achieved. GPU physical limits relate to numbers of threads, thread-blocks, available registers, allocatable size of shared memory per SM. SM schedules and executes threads in warps, groups of 32 parallel threads. The number of active thread-blocks and warps depends on the amount of shared memory and registers used by the kernel but is constrained by their physical limits. The ratio of the number of active warps per SM to the maximum number of possible active warps is the manufacturer's metric for warp utilization called occupancy [21].

A straightforward approach for execution configuration is to map one residual block to one thread-block. Table 1 shows processing time for several configurations for 4×4 TBs including boundary cases when only one TB is processed within a thread-block and when the maximum thread-block size is configured. Configured number of threads depends on the mapping ratio. For example, as four threads process one TB, there are 512 threads configured for 128:1 block mapping.

Low occupancy for 1:1 block mapping is caused by the low utilization of thread-blocks. Though the maximum number of thread-blocks resides on every SM, there is only one warp allocated per thread-block. Moreover, this one warp is not fully utilized. Only four of its threads are active. In the case of 64:1 block mapping, higher thread-block utilization yields higher performance, with a kernel speed-up of 9.8 times. As the mapping ratio further increases, so does kernel execution time. This behaviour is caused by the progressive reduction of global memory throughput. As more TBs are mapped to a thread-block the stride of the kernel loop is larger and fewer memory transactions can be served from the L2 cache load. The L2 cache is shared by all SMs and used to cache accesses to global memory. By using vectorized memory access the kernel becomes computation-bound and therefore lower occupancy in case of 192:1 block mapping is not penalized with additionally longer execution time. The block mapping of ratio 64:1 with TB and thread indexing is illustrated in Figure 2. This thread-block size will be employed for all transform sizes in the rest of the paper.

Considering the selected thread-block size, the number of TBs which are mapped to one thread-block is determined for TBs of larger size by the equation:

$$n_{TB} = \frac{256}{N \times N/4} \tag{2}$$

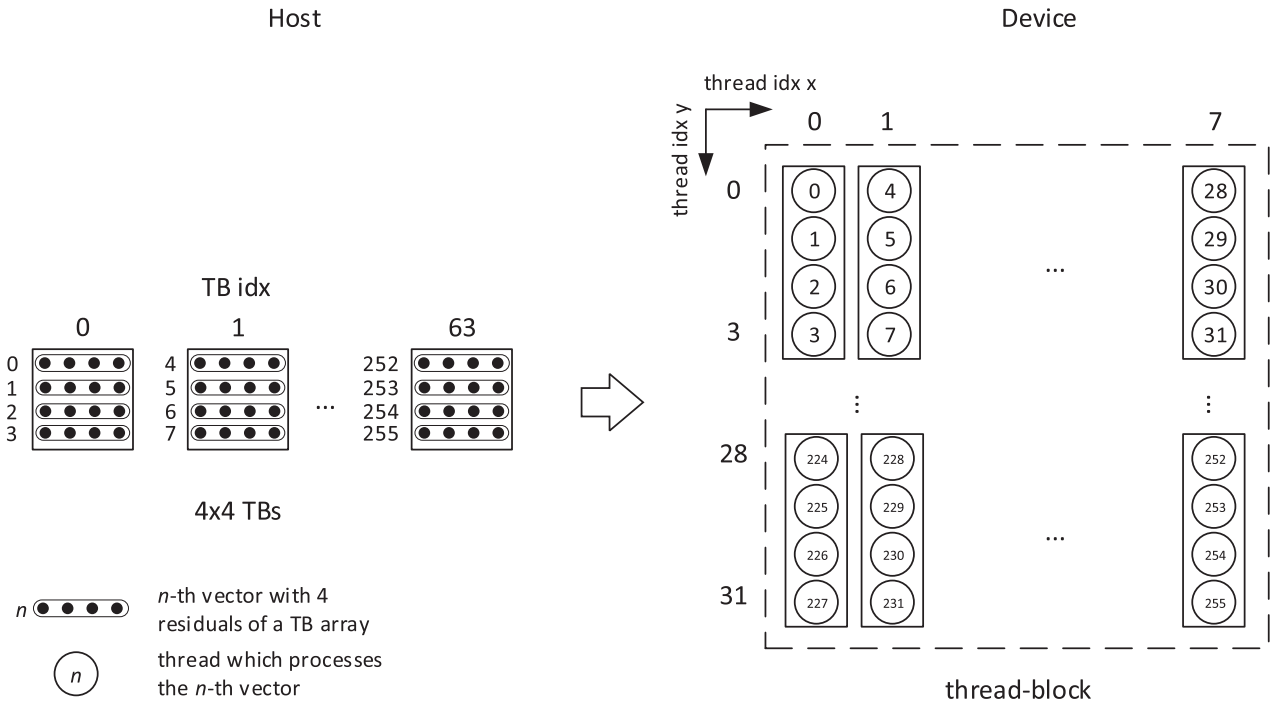

Figure 2. 64:1 residual to thread-block mapping from host to device for 4 × 4 transform block.

where 256 is the selected thread-block size, $N \times N$ the size of TB and divisor 4 appears due to vectorized access, as each thread processes 4 residuals.

Efficient vectorized access pattern to shared memory

By using 64-bit vectors, an adjustment to the bus width of the shared memory bank is accomplished. To achieve overall high memory bandwidth, an appropriate access pattern for all TB sizes has to be created. In the 2D transform application, every thread will execute a dot product of a matching row in multiplicand block with a matching column in the multiplier block. Reading row data is performed as reading along the banks and as such is naturally parallel. In the case of the widest row, which appears for 32 × 32 TB, row data will reside in eight different banks. Data load can be done within one transaction.

Loading of column data within a warp with an access pattern which results in reading from the same bank causes bank conflicts. With an array width of 8, which is the least common multiple for all array widths in vectorized access (1, 2, 4, 8), and regular access pattern bank conflict would happen for transform sizes larger than four. For that particular transform size memory locations, belonging not only to column data from one TB but to complete row of TBs in thread-block, always map to different memory banks. For example, in the case of transform size 8 × 8, 32 memory locations in separate banks are sufficient to store only half of one row of TBs in a thread-block.

Memory requests to one bank are split into as many requests as there are requested 64-bit words in that bank. To prevent bank conflicts, shared memory 2D array is padded with an additional column. The additional column will cause data to shift right to the new bank but this technique is not efficient for all transform sizes. Access pattern to padded shared memory array for one thread-block is shown for 4 × 4 TBs and 32 × 32 TBs in Figure 3. Padded array locations are marked with striped squares. In the case of a 4-point 1D transform, threads which process eight consecutive TBs compose one warp. As can be seen, when padding is used, some data which are processed by the first warp are spilled to the next memory location in the already used banks. Such data couldn't be accessed in the same transaction as one from the upper memory location. Without padding, the first warp would access memory location 0 in all 32 banks in the same cycle. Similar happens for 8×8 TBs where warps would retrieve necessary data from shared memory in three requests when padding is applied instead of two requests when padding is omitted. For 16 × 16 and 32 × 32 TBs there are four and eight serialized memory requests to column data respectively when there is no padding. With padding, those accesses are conflict-free. For the latter transform size, the padding is shown in Figure 3(b).

It is obvious that padding is not appropriate for smaller transform size and as such it is not applied there. The impact on the kernel execution time for different sizes is confirmed with measurements presented in Table 2. For every test case, it is assumed that the frame is split to same sized TBs. Manufacturer's quality metric shared efficiency, defined as the ratio

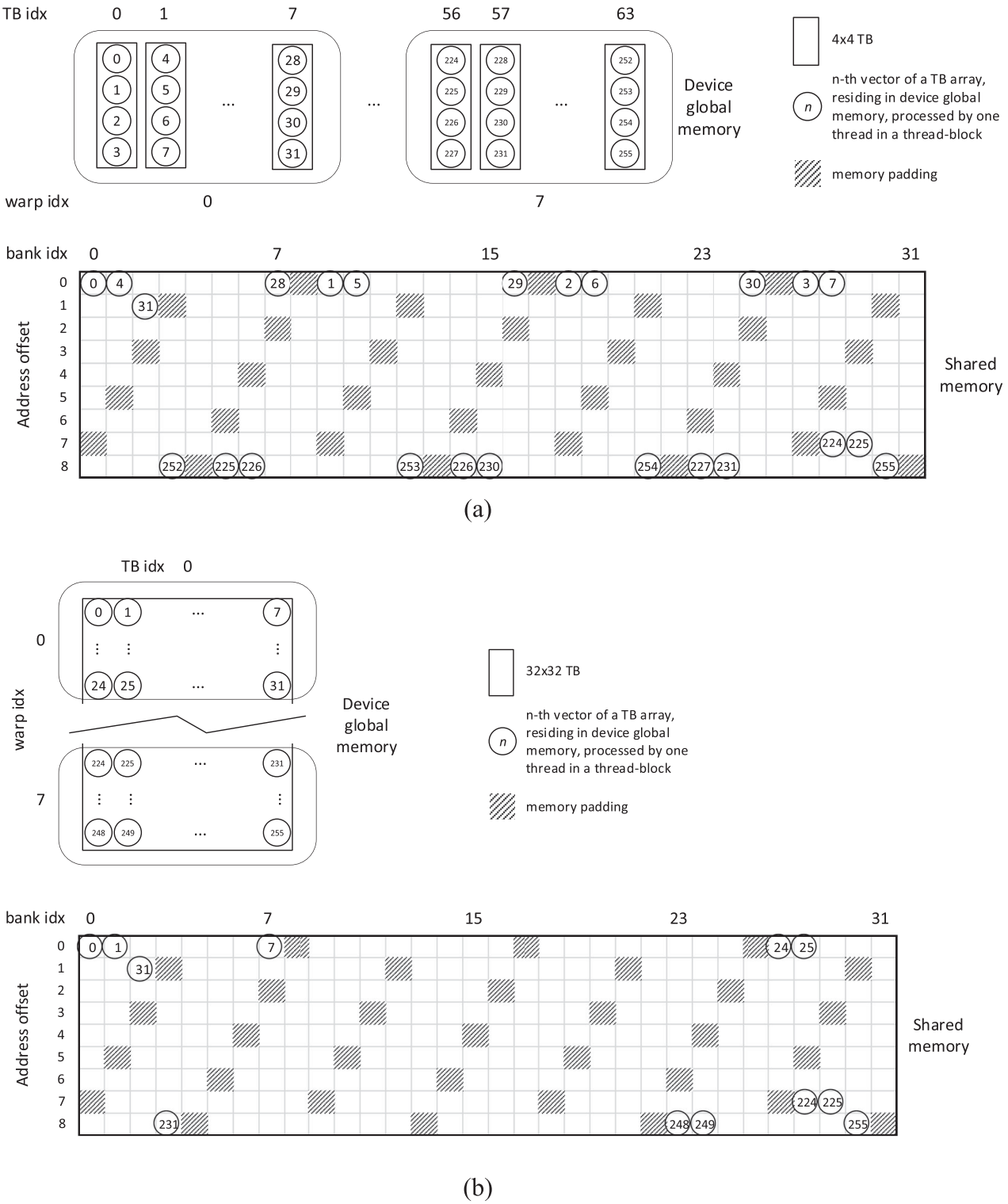

Figure 3. Padded shared memory with the access pattern for (a) 4 × 4 and (b) 32 × 32 TBs.

of requested shared memory throughput to required shared memory throughput, is used as the decision criteria.

Core transform matrix is predefined for each transform size. It is loaded from global memory and stored to the shared memory once for a whole thread-block. Read accesses from threads in a warp, belonging to different TBs, to a single core transform array result in the performance efficient broadcast mechanism.

Implementation and evaluation

In this section, the performance of the proposed parallel implementation named GHTQ is evaluated in terms of overall and kernel processing time per frame. TBs with random values were generated at the host side, transferred to the device using page-locked memory and processed there. Quantized transform coefficients and AZB identification are sent back as process outputs

Table 2. Execution times and shared memory efficiency with or without data padding.

TB size	No. of blocks (DCI 4K)	Shared memory array padding	Overall execution time [ms]	Kernel execution time [ms]	Shared memory efficiency [%]
4 × 4	829,440	No	9.69	4.99	81
		Yes	10.01	5.15	67.5
8 × 8	207,360	No	12.59	7.89	90.2
		Yes	12.67	8.07	83.4
16 × 16	51,840	No	17.92	13.26	69.9
		Yes	17.49	12.87	91
32 × 32	12,960	No	30.04	25.37	26.7
		Yes	27.69	22.93	95.3

Table 3. Experiment environment.

Environment name	Desktop GPU	Workstation GPU
CPU	Intel Core i5-4570 (3.20 GHz)	Intel Core i5-3570K (3.40 GHz)
GPU	NVIDIA GeForce GT 640	NVIDIA Tesla K40c
Bus	PCI Express x16 Gen3	

Table 4. GPU comparison.

GPU model	NVIDIA GeForce GT 640	NVIDIA Tesla K40c
SM count	2	15
Core count per SM	192	192
Core clock (MHz)	902	745
Memory bandwidth (GB/s)	28.51	288.4

to the host. The number of blocks matches to two video resolutions DCI 4K and 8K Full Format. To place results in the context of other computing platforms and evaluate the efficiency of performance optimization strategies, processing times were compared with CPU, AVX2 and GPU implementation based on NVIDIA's library cuBLAS. The proposed and cuBLAS implementation are made using CUDA Toolkit 10.1. The cuBLAS functions don't support integer data types larger than eight bits so cuBLAS function for strided batched matrix multiplication for a 32-bit floating-point was used to carry out 1D transforms with scaling. Custom high-performance rounding and rounding with quantization kernels were designed, following principles for proposed TQ kernel, to get HEVC compliant intermediate and output values.

Since computational complexity differs for different transform sizes, the three block distributions were evaluated, frame split to only 4 × 4 and 32 × 32 TBs respectively and simulated real-valued block distribution with block shares for transform sizes 4 × 4–32 × 32 as 58%, 32%, 8% and 2% according to [22]. QP value was set to 22 for all measurements.

The proposed parallel HEVC transform and quantization kernel with AZB identification was implemented using the environments given in Table 3.

GPUs from both environments have the same Kepler architecture but Tesla K40c exhibits higher performance due to more SMs and faster memory as shown in Table 4. For measurements obtained for benchmarking and comparison with competing implementation in this section, the Workstation GPU was used with exception of the AVX2 implementation. The CPU in the Workstation GPU environment doesn't support the AVX2 instruction set. The AVX2 implementation was therefore made on the Desktop GPU with the most computational complex block distribution and DCI 4K resolution.

The average frame processing times for different HEVC TQ distributions and frame resolutions are presented in Tables 5 and 6.

The results show that the GHTQ implementation outperforms others. Compared to the CPU implementation speed-ups range from 40 to 80 times depending on block distribution. The cuBLAS implementation unexpectedly exhibits lower performance for the lower complexity 4-point transform and quantization. This is the reason why that implementation has a different performance trend compared to the proposed implementation. Using the manufacturer's profiling tool, the NVIDIA Visual Profiler, it can be seen that kernel functions which implement the 1D transform are characterized by inefficient access to global and shared memory respectively and low thread utilization. Moreover, kernel function with a limitation of a grid in x-dimension, which was valid for earlier GPU versions, was invoked. With many kernel function calls, the performance is significantly degraded. For the 32-point transform memory, accesses are not an issue but low occupancy indicates low resource utilization. If block distribution with real-valued distribution is observed, the achieved speed-up compared to cuBLAS is about 5.8 times for the DCI 4K resolution and 6.7 times for the 8K Full Format. For the block distribution 32 × 32 for DCI 4K on the Desktop GPU environment, where it is supported, AVX2 implementation is slower about 4 times.

Performance comparison is made with a parallel implementation on a GPU from [14] for the 3840 × 2160 frame resolution. Though that proposal brings parallel, fully HEVC compliant, implementation of DQIT, the share of control logic for bypassing TQ, handling CBF and transform skip in processing time can be assessed as neglectable. Moreover, the proposed implementation contains AZB identification as an additional processing stage. As written before, the computational complexity for TQ and its inverse is

Table 5. Frame processing time for different implementations of the HEVC TQ with AZB identification on the Workstation GPU environment.

		Processing time [ms]				
Resolution	Block distribution	CPU	GHTQ overall	GHTQ kernel	cuBLAS overall	cuBLAS kernel
DCI 4K	4×4	246.26	6.03	0.79	113.84	101.75
	real-valued	421.32	8.51	2.30	48.95	38.55
	32×32	708.11	8.88	3.66	21.48	8.96
8K Full Format	4×4	9.79	23.94	2.96	447.65	407.12
	real-valued	1693.29	29.12	8.49	193.71	153.12
	32×32	2832.67	36.19	14.00	76.38	35.84

Table 6. Frame processing time for different implementations of the HEVC TQ with AZB identification on the Desktop GPU environment.

		Processing time [ms]					
Resolution	Block distribution	CPU	AVX2	GHTQ overall	GHTQ kernel	cuBLAS overall	cuBLAS kernel
DCI 4K	32×32	818.7	109.48	27.53	22.95	37.62	28.80

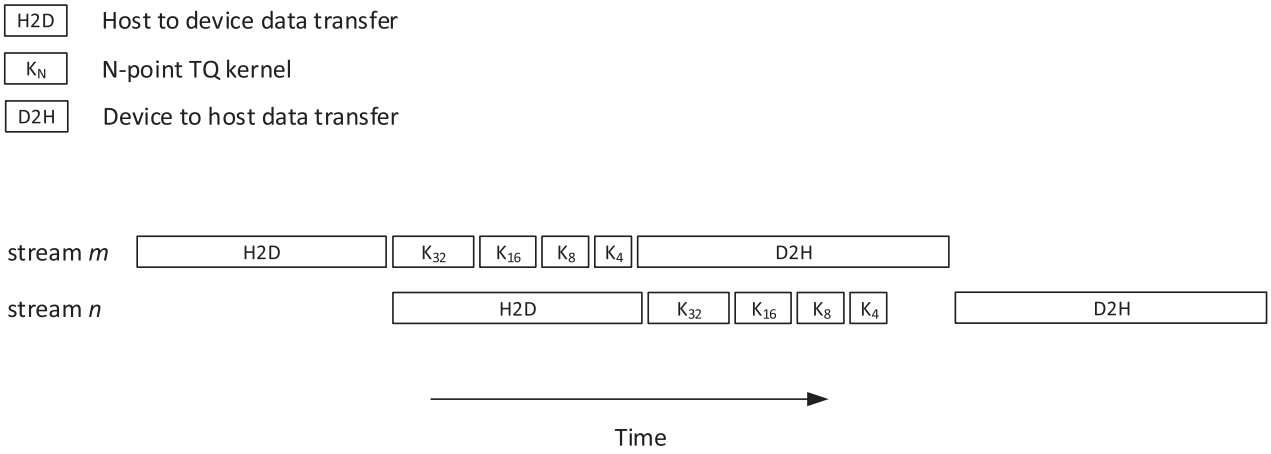

Figure 4. Overlapping TQ kernel execution and data transfers by using two CUDA streams.

Table 7. Comparison with related implementation.

Implementation	Block distribution	Processing time [ms]
De Souza	"DucksTakeOff" B Frame, Random Access	6.56
Proposed	32×32	6.17
	real-valued	5.38

the equivalent or lower. OpenCL implementation used in the related work doesn't obtain worse performance than CUDA [23]. To be able to have a fair comparison, kernel execution is overlapped with data transfers in the proposed implementation by using CUDA streams. The frame is broken up into two segments with the execution flow shown in Figure 4.

Frame processing times for parallel implementations are shown in Table 7. In the case of the proposed implementation, two block distributions are considered, real-valued and 32×32, as worst-case distribution related to performance. Time comparison is made against the best time for a given resolution in the competing proposal. As can be observed, the proposed implementation achieved a speedup as high as 1.22 times.

The performance gain achieved through the overlapping generally depends on the ratio between the data transfer duration and the duration of kernel execution. Closer this value is to one, the higher the speedup is. Highest speedup for the worst-case distribution is obtained with nine segments where the frame processing time equals 4.82 ms. Resulting processing time exposes the capability of this implementation to be used in real-time applications. For a frame rate of 50 fps, each frame has to be processed in less than 20 ms. Considering the decoding time distribution for DQIT functions in all intra configuration [3], the remaining time would be sufficient for other decoding stages to be done.

Conclusion

This paper presents the performance-engineered, transparent HEVC TQ kernel to be executed on GPU accelerators. Proposed methods combine efficient transform block to thread-block mapping and vectorized access patterns to shared memory. Using page-locked memory, residual data blocks were transferred to the device, processed to transformed quantized coefficient blocks and accompanied by the AZB identification array. Result data are transferred back to the host. Experiments were conducted using mid and high-end GPUs to confirm performance optimizations. They showed

increases in speed up to 80, 19 and 4 times compared to the CPU, cuBLAS and AVX2 implementations respectively. Comparison with the related highly parallel implementation exhibits speedups up to 1.22 times. The proposed GPU implementation can support the real-time decoding process.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

The work presented in this paper has been partially funded by the European Processor Initiative project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826647.

ORCID

Mate Čobrnić http://orcid.org/0000-0002-1422-0392
Alen Duspara http://orcid.org/0000-0002-9660-2959
Leon Dragić http://orcid.org/0000-0002-4558-7269
Igor Piljić http://orcid.org/0000-0003-2345-0322
Mario Kovač http://orcid.org/0000-0002-8365-7002

References

[1] Cisco. Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper. February 27 2019. [Online]. Available from: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html.
[2] Sullivan GJ, Ohm JR, Han WJ, et al. Overview of the high efficiency video coding (HEVC) standard. IEEE Trans Circuits Syst Video Technol 2012;22(12):1649–1668.
[3] Bossen F, Bross B, Sühring K, et al. HEVC complexity and implementation Analysis. IEEE Trans Circuits Syst Video Technol. 2012;22(12):1685–1696.
[4] Budagavi M, Fuldseth A, Bjontegaard G, et al. Core transform design in the high efficiency video coding (HEVC) standard. IEEE J Sel Topics in Signal Process. 2013;7(6):1029–1041.
[5] F. Liu, Y. Liang, L. Wang, A survey of the heterogeneous computing platform and related technologies. 2016 International Conference on Informatics, Management Engineering and Industrial Application (IMEIA), Phuket; 2016.
[6] NVIDIA Corp. CUDA Compute Unified Device Architecture, Programming Guide, Version 10.1.243, 19 August 2019. [Online]. Available from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[7] Xiao W, Li B, Xu J, et al. HEVC encoding optimization using multicore CPUs and GPUs. IEEE Trans Circuits Syst Video Technol. Nov. 2015;25(11):1830–1843.
[8] Wang F, Zhou D, Goto S. OpenCL based high-quality HEVC motion estimation on GPU. 2014 IEEE International Conference on Image Processing (ICIP), Paris; 2014, pp. 1263–1267.
[9] de Souza DF, Ilic A, Roma N, et al. GHEVC: An efficient HEVC decoder for graphics processing units. IEEE Trans Multimedia. March 2017;19(3):459–474.
[10] Mohamed B, et al. High-level synthesis hardware implementation and verification of HEVC DCT on SoC-FPGA. 2017 13th International Computer Engineering Conference (ICENCO), Cairo; 2017, pp. 361–365.
[11] He L, Goto S. A high parallel way for processing IQ/IT part of HEVC decoder based on GPU. 2014 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), Kuching; 2014, pp. 211–215.
[12] Igarashi H, Takano F, Moriyoshi T. Highly parallel transformation and quantization for HEVC encoder on GPUs. 2016 Visual Communications and Image Processing (VCIP), Chengdu; 2016, pp. 1–4.
[13] Masoumi M, Ahmadifar H. Performance of HEVC discrete cosine and sine transforms on GPU using CUDA. 2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI), Tehran; 2017, pp. 0857–0861.
[14] Souza DFD, Roma N, Sousa L. Opencl parallelization of the HEVC de-quantization and inverse transform for heterogeneous platforms. 2014 22nd European Signal Processing Conference (EUSIPCO), Lisbon; 2014, pp. 755–759.
[15] Ryoo S, Rodrigues CI, Baghsorkhi SS, et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP), ACM, New York, USA; 2008 pp. 73–82.
[16] Cui X, Chen Y, Mei H. Improving performance of matrix multiplication and FFT on GPU. 2009 15th International Conference on Parallel and Distributed Systems, Shenzhen; 2009, pp. 42–48.
[17] Volkov V, Demmel JW. Benchmarking GPUs to tune dense linear algebra. SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, TX; 2008, pp. 1–11.
[18] Harris M. How to optimize data transfers in CUDA C/C++. NVIDIA Developer Blog, 19 August 2019. [Online]. Available from: https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/.
[19] Harris M. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. NVIDIA Developer Blog, 19 August 2019. [Online]. Available from: https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.
[20] Piljić I, Dragić L, Duspara A, et al. Bolt65 – performance-optimized HEVC HW/SW suite for Just-in-Time video processing. 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia; 2019, pp. 966–970.
[21] NVIDIA Corp. CUDA Occupancy Calculator, Version 10.1.243, August 19 2019. [Online]. Available from: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.
[22] Hong L, He W, Zhu H, et al. A cost effective 2-D adaptive block size IDCT architecture for HEVC standard. 2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS), Columbus, OH; 2013, pp. 1290–1293.
[23] Fang J, Varbanescu AL, Sips H. A comprehensive performance comparison of CUDA and OpenCL. 2011 International Conference on Parallel Processing, Taipei City; 2011, pp. 216–225.