QuickETC2: Fast ETC2 Texture Compression using Luma Differences

JAE-HO NAH, LG Electronics, South Korea

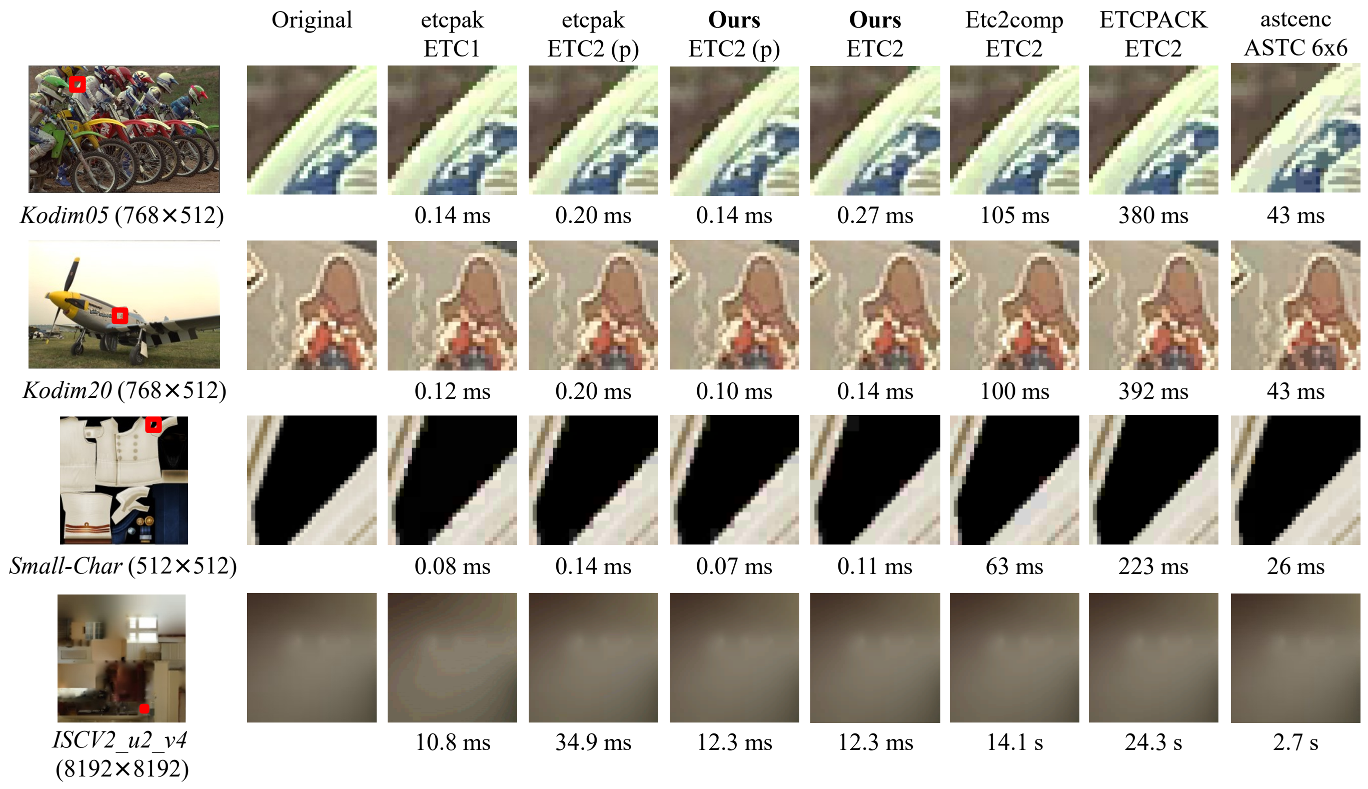


Fig. 1. **Quality and performance comparison of our approach and other compressors (with their fastest settings)**. Performance of *QuickETC2* in the partial ETC2 mode (planar only - ETC2 (p)) is comparable to that of *etcpak* [Taudul and Jungmann 2020] in the ETC1 mode, but its quality is similar to that of *etcpak* in the ETC2 (p) mode (see no banding artifacts on *ISCV2_u2_v4*). Ours in the full ETC2 mode provides much better edge handling and less color distortion than *etcpak* in the ETC2 (p) mode. Compared to *Etc2comp* [Google Inc. and Blue Shift Inc. 2017] and *ETCPACK* [Arm Limited 2016; Ericsson 2018], ours is two to three orders of magnitude faster. Compared to *astcenc* [Arm Limited. 2020] in the ASTC 6x6 mode, ours is two orders of magnitude faster and shows better color preservation (*Kodim05*) and less ringing artifacts (*Kodim20*). We obtained the timings on a desktop with an AMD Ryzen 7 3700X@3.6GHz 8-core (with hyper-threading) CPU. ©Kodak, UNC GAMMA Lab, and *fhernand.*

Compressed textures are indispensable in most 3D graphics applications to reduce memory traffic and increase performance. For higher-quality graphics, the number and size of textures in an application have continuously increased. Additionally, the ETC2 texture format, which is mandatory in OpenGL ES 3.0, OpenGL 4.3, and Android 4.3 (and later versions), requires more complex texture compression than the traditional ETC1 format. As a result, texture compression becomes more and more time-consuming.

To accelerate ETC2 compression, we introduce two new compression techniques, named QuickETC2. The first technique is an early compression-mode decision scheme. Instead of testing all ETC1/2 modes to compress a texel block, we select proper modes for each block by exploiting the luma difference of the block to reduce unnecessary compression overhead. The second technique is a fast luma-based T- and H-mode compression method. When clustering each texel into two groups, we replace the 3D RGB space with the 1D luma space and quickly find the two groups that have the minimum luma differences. We also selectively perform the T- or H-mode and reduce its distance candidates, according to the luma differences of each group. We have implemented both techniques with AVX2 intrinsics to exploit SIMD parallelism. According to our experiments, QuickETC2 can compress more than 2000 1K×1K-sized images per second on an octa-core CPU.

CCS Concepts: • **Computing methodologies → Image compression**.

Additional Key Words and Phrases: texture compression, ETC2

Author's address: Jae-Ho Nah, LG Electronics, 19, Yangjae-daero 11-gil, Seocho-gu, Seoul, 06772, South Korea, nahjaeho@gmail.com.

0730-0301/2020/12-ART270 $15.00
https://doi.org/10.1145/3414685.3417787

ACM Reference Format:
Jae-Ho Nah. 2020. QuickETC2: Fast ETC2 Texture Compression using Luma Differences. *ACM Trans. Graph.* 39, 6, Article 270 (December 2020), 10 pages. https://doi.org/10.1145/3414685.3417787

1 INTRODUCTION

Texture mapping is one of the fundamental concepts in computer graphics. Textures are versatile for various types of data, such as diffuse maps, normal maps, reflection maps, and lightmaps. Usually, textures are stored in a compressed format, such as ETC1 [Ström and Akenine-Möller 2005], ETC2 [Ström and Pettersson 2007], BC [Microsoft 2018], ASTC [Nystad et al. 2012], or PVRTC [Fenney 2003], to reduce the required memory bandwidth, memory size, and storage space. A compressed texture should be decoded in real-time on a GPU, so texture decompression is relatively simple, while texture compression takes longer than texture decompression.

With increasing graphics quality and screen resolution, the total size of textures in an app (e.g., a game) has reached up to dozens of gigabytes. Additionally, some online GIS (geographic information systems) software (e.g., Google Maps and Google Earth) include an extremely large amount of textures. In these cases, texture compression may be a bottleneck in the SW development process. For example, if we need to compress 5000 4K-resolution textures by using a texture encoder with a throughput of 1M pixels/s, the total encoding time will be almost one day.

There are also a few scenarios with limited time budgets. Real-time 3D reconstruction is one of the representative examples; high-resolution textures, captured from the real world (e.g., *ISCV2_u2_v4* in Figure 1), need to be compressed before mapping them into reconstructed geometry [Easterbrook et al. 2010]. Thus, low-speed encoding can make users feel stuck. If textures are resized during app loading [Nah et al. 2018] for reducing a GPU power consumption, the texture compression time needs to be minimized for a similar reason. In in-home streaming [Pohl et al. 2017], in-game video capturing [Kemen 2012], and web browsing [Oom 2016], low latency is commonly critical. In the above types of software, the speed of texture compression can be more important than its quality.

Since Krajcevski and Manocha [2013] presented a fast BC7 encoding algorithm, there have been several active attempts to accelerate texture compression. Among them, *etcpak* [Taudul and Jungmann 2020] has been known as the fastest ETC compressor. This software supports both the ETC1 and ETC2 codecs, and its ETC2 compression with the planar mode provides better quality with around 1.5-2× increased compression time than the ETC1-only compression. Because ETC1 and ETC2 are the standard formats on the Android platforms, *etcpak* has been widely used by mobile app developers. The current version of *etcpak* does not support the T- and H-modes because these ETC2 modes can rapidly increase compression time, as implemented in *ETCPACK* [Ericsson 2018], the reference ETC2 compressor.

1.1 Main results

Our goal is to increase the speed or quality of the existing compressors. To achieve this, we present the following novel approaches using luma (linear luminance) differences.

- To reduce the ETC2 overhead, we present an early compression-mode decision scheme that prevents unnecessary duplicated tests.
- To support the full ETC2 mode (T, H, and planar) without significantly increased costs, we present a new fast T-/H-mode compression algorithm.

Our experiments show that the first scheme in the partial ETC2 mode (planar only) increases the compression speed by 30%-202% (67% on average) compared to *etcpak* [Taudul and Jungmann 2020], and the full ETC2 compression with a combination of the two schemes achieves two to three orders of magnitude speed-up than *Etc2Comp* [Google Inc. and Blue Shift Inc. 2017] and *ETCPACK* [Arm Limited 2016; Ericsson 2018].

Note that this paper is the full version of the extended abstract [Nah 2020]. Compared to the earlier version, we introduce additional related work, describe the two novel approaches in detail, and analyze the experimental results with limitations. The new results in this paper also show further performance and quality improvements, thanks to additional optimizations.

2 RELATED WORK

In this section, we will first summarize the ETC1/2 codecs. After that, we will introduce state-of-the-art ETC compressors. Finally, we will briefly describe the other standard texture-compression formats.

2.1 ETC1/2

The basic idea of ETC1 [Ström and Akenine-Möller 2005] is a combination of two base colors and per-pixel luminance modulations in a block. The size of a block is 4×4, and an ETC1 encoder codes either two neighboring 4×2 (horizontal) or 2×4 (vertical) sub-blocks together. If the two sub-blocks share similar color values, then the encoder would select the differential mode with two different base colors in the RGB555 and dRdGdB333 formats. Otherwise, the encoder would separately calculate the base colors in the RGB444 format in the individual mode. For luminance modulations, the encoder would try to select proper modifiers for a sub-block from the predefined codebook to fill 16 2-bit per-pixel indices with the modifier values for each pixel. This ETC1 compression converts 16 24-bit RGB pixels into a 64-bit block (a 6:1 compression ratio).

ETC2 [Ström and Pettersson 2007] improves the quality of ETC1 by adding the three new modes: T, H, and planar (Figure 2). If color differences in a block are high and the horizontal or vertical mode cannot properly handle the block, block artifacts can appear (see *Kodim05* and *Small-char* in Figure 1). To solve these artifacts, the T- and H-modes do not limit the partition pattern to 2×4 or 4×2 subblocks and instead limit the number of colors per block to four. After clustering the colors of a block into two groups, the T-mode modulates the base color of the larger cluster for unevenly distributed colors. In contrast, the H-mode modulates both base colors of the two clusters. On the other hand, the planar mode improves gradients of slowly varying colors (see *ISCV2_u2_v4* in Figure 1) by interpolating three RGB676 base colors at the corners of each block. An ETC2 encoder usually selects a block with the lowest error among the blocks calculated in each mode. By the use of invalid bit combinations, ETC2 keeps backward compatibility on ETC1 and maintains the compression ratio.

The ETC2 codecs are mandatory in OpenGL ES 3.0 [Leech and Lipcha 2019] and OpenGL 4.3 [Segal and Akeley 2013] and consist of the ETC2 RGB codec, the additional EAC codecs, and the ETC2/EAC RGBA codec. EAC remedies ETC1's another shortcoming: lack of alpha support. The EAC codecs use table-based alpha compression executed independently from the ETC1/2 modes. In the codecs, EAC compression is either solely used for compressing a one- or two-channel texture or used with ETC2 compression for encoding the alpha channel in an RGBA texture.

ETC1S [Khronos Group 2019] is a subset of ETC1 and one of the base formats of Basis Universal [Binomial LLC 2020], a supercompressed GPU texture compression system. Basis Universal provides higher compression ratios than the original ETC1/2, and a supercompressed basis texture can be transcoded into one of the standard GPU texture compression formats (BC1-5, BC7, ETC1/2, PVRTC, and ASTC) before being uploaded to a GPU memory. Therefore, this system is useful for reducing the network bandwidth required for texture streaming or app downloading.

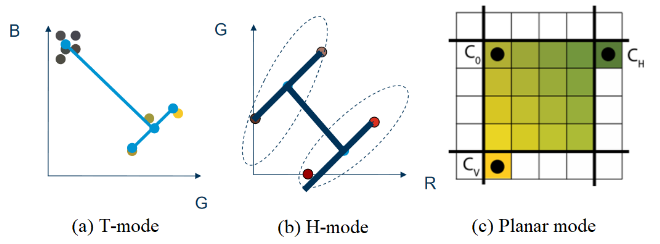


Fig. 2. The additional ETC2 modes [Ström and Pettersson 2007]: (a) the T-mode for an uneven color distribution in a block, (b) the H-mode for an even color distribution in a block, and (c) the planar mode for smooth gradients. ©Strőm and Pettersson.

2.2 ETC compressors

The first author of ETC1/2 has opened the encoding and decoding source code, named *ETCPACK* [Ericsson 2018], to the public. The ETC1/2 compression logic in texture compression tools made by mobile GPU companies, such as the Mali Texture Compression Tool and PVRTexTool, is based on *ETCPACK*.

ETCPACK does not support multi-threading and SIMD vectorization, and its slow mode takes a long compression time due to its exhaustive search. Faster ETC compressors have been announced as alternatives to address the performance issue. *Etc2Comp* [Google Inc. and Blue Shift Inc. 2017; McAnlis 2016] accelerates ETC1/2 compression by a more targeted search, fine control of search-space exploration, and multi-threading. As a result, *Etc2Comp* provides better trade-offs between quality and speed than *ETCPACK*. On the other hand, *etcpak* [Taudul and Jungmann 2020] aims at extremely fast ETC1/2 compression; this compressor consists of highly scalable, AVX2-optimized code and limits search spaces and the ETC2 compression mode (planar only). The Intel ISPC texture compressor [Dufresne 2015; Intel Corp. 2019] accelerates BC, ETC1, and ASTC compression by exploiting data- and thread-level parallelism through the Intel SPMD Program Compiler (ISPC). Among the compressors, Unity Technologies [2017] have integrated *etcpak*, *ETCPACK*, and *Etc2Comp* into their game engine (since Unity 2017.3) for the fast, normal, and best compression options, respectively.

2.3 Other texture compression formats

BC [Microsoft 2018] is a set of block compression formats included in DirectX for desktop platforms. BC1 compresses a 4×4 RGB block using two endpoints, four colors derived from the endpoints, and a per-pixel index table. BC1 also supports a punch-through alpha channel. BC2 consists of a 64 bit BC1 RGB data and a 64bit uncompressed 4-bit alpha data in a 128-bit block. BC4 and 5 are BC1-style alpha-compression formats for a single-channel and two-channel alpha values, respectively. BC3 is a combination of BC1 and BC4. BC6H is a dedicated format for HDR textures. BC7 provides very high quality for both RGB and RGBA textures, through multiple partition sets and endpoint formats. In OpenGL, S3TC [Iourcha et al. 1999], RGTC, and BPTC correspond to BC1-3, BC4-5, and BC6H/BC7, respectively.

PVRTC [Fenney 2003], one of the iOS standard formats, adopts a different strategy compared to other block-based compression formats: bilinear upscaling of two lower resolution images. Due to that, this can alleviate block artifacts but can lose some details. PVRTC2 [Voica 2013] makes up for a lot of weakness of PVRTC regarding compression quality and features, but it is only supported on a limited set of Android devices with PowerVR GPUs.

ASTC [Nystad et al. 2012] is mandatory in OpenGL ES 3.2, so most of the recent Android/iOS devices support that format. Its the most important feature is flexibility and scalability; it allows multiple block sizes (4×4 to 12×12), color channels (1-4), dimensions (2D and 3D), and color spaces (SDR and HDR). This format also provides comparable or better compression quality than BC7 and ETC2, thanks to the support of the multiple compression modes and partition sets.

If you are interested in a detailed overview of the above formats, we recommend Paltashev and Perminov [2014]'s review article.

3 EARLY COMPRESSION-MODE DECISION

We observe that the three ETC2 modes assist ETC1 in different ways. The planar mode achieves smooth gradients in low-contrast regions, while the T- and H-modes reduce block artifacts in high-contrast regions. Therefore, if we can determine proper compression mode(s) in advance according to the luma difference of a block, we can avoid duplicated compression (Figure 3).

The procedure is described as follows. We first calculate the luma values of each pixel in a block and find the min/max luma values in the block. Using the values, we classify the block into one of four types: very-low contrast (luma_diff$\leq T_1$), low-contrast ($T_1<$luma_diff$\leq T_2$), mid-contrast ($T_2<$luma_diff$<T_3$), and high-contrast (luma_diff$\geq T_3$). We set the thresholds T_n, according to our experimental results illustrated in Figure 8: T_1=0.03, T_2=0.09, and T_3=0.38.

For very-low contrast blocks, we use the planar mode. The reason is that the planar mode is the most suitable mode to the type of blocks to remove banding artifacts.

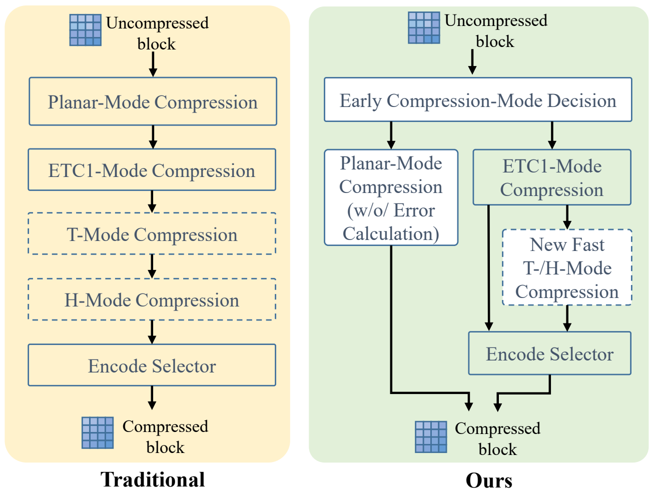


Fig. 3. ETC2-compression flow charts of the previous work (left) and ours (right). The dotted blocks can be excluded if speed is preferable to quality. The white blocks are newly introduced or modified in this paper. Note that the compression order in the left chart does not affect the compression results and can be changed.

In the case of low-contrast blocks, we check whether a block can be smoothly expressed by the base colors at the corners of the block. If a pair of two corresponding corner pixels (top-left and bottom-right, or bottom-left and top-right) has the min and max luma values, we exploit a high possibility that the other pixels can be properly interpolated in the planar mode. Otherwise, we perform traditional ETC1 compression. Because we do not pass a block compressed in the planar mode to the encode selector stage, we omit the error calculation part in the planar mode for a speedup.

We compress mid-contrast blocks in the ETC1 mode. The reason is that this mode can usually compress the type of blocks well without visible banding or block artifacts.

High-contrast blocks may create block artifacts, so we perform both the ETC1 and T-/H-mode compression. The T-/H-mode compression may not find an ideal compression solution. Thus, at the late encode selector stage, we compare the errors in the ETC1 and T/H modes and select the block with a lower error value.

We execute this early compression-mode decision for all blocks, so we need to minimize its overhead. To achieve this, we implement most of the parts described above using SSE/AVX2 operations. The key point is to avoid loop iterations when accessing 16 pixels in a block. The detailed procedure is described as follows. First, we calculate 16 8-bit luma values from the RGB colors of the 16 pixels; we store them together in a single 128-bit integer (__m128i) variable. After that, we obtain the minimum and maximum values and their positions over the 16 pixels by exploiting the _mm_minpos_epu16() function; because this function returns only the position of the minimum value among eight 16-bit values, we tweak the use of this function, similar to Kluev [2014]. Finally, we perform a corner pixel check by using a single SSE comparison of the corner index pairs and the pixel indices corresponding to the min/max values. If necessary, please see the source code in the supplemental material.

4 LUMA-BASED T-/H-MODE COMPRESSION ALGORITHM

4.1 Analysis of traditional T-/H-mode compression

Algorithm 1 describes a brief procedure of traditional T-/H-mode compression. First, an encoder calculates a pair of base colors by classifying all pixels in a block into two clusters. After that, the encoder consecutively compresses the block in the T- and H-modes. Finally, the encoder returns a compressed block with the lower error value between them.

ALGORITHM 1: Traditional T-/H-mode compression procedure

```
1: procedure ETC2_TH($pix$)
2:     $\{c1, c2\} \leftarrow FindBaseColors(pix)$
3:     $\{blockT, errorT\} \leftarrow CompressBlockT(pix, c1, c2)$
4:     $\{blockH, errorH\} \leftarrow CompressBlockH(pix, c1, c2)$
5:     return $(errorT < errorH)?$
           $\{blockT, errorT\}$:$\{blockH, errorH\}$
6: end procedure
```

We can express the computational cost of the T-/H-mode compression (C_{TH}) with the following equation.

$$C_{TH} = N_{BC}(F_{BC} \cdot C_{BC} + N_{Mode} \cdot N_{Dist} \cdot C_{EC}), \quad (1)$$

where C_{BC} and C_{EC} are the costs of base-color calculation and error calculation, respectively. Additionally, N_{BC}, N_{Mode}, and N_{Dist} are the numbers of base-color pairs, compression modes, and distance candidates, respectively. Note that the distance means an 8-bit color difference between a base color and two paint colors in the direction (1, 1, 1). This value controls the luminance ranges of a cluster, as depicted in Stage 6 in Figure 4. Finally, F_{BC} is a function that outputs either one or two. If the T- and H-modes share the same base-color pairs, the value is one; otherwise, the value is two.

N_{BC} outside the parenthesis means that an encoder can iteratively execute the ETC2_TH() function in Algorithm 1 to test different base colors; for example, *ETCPACK* [Ericsson 2018] with the fast option injects different clustering parameters to the T- and H-modes and tests three color pairs per mode calculated using the LBG algorithm [Linde et al. 1980]. Its slow option tests many more colors by using a exhaustive search. N_{Mode} in the previous work [Ericsson 2018; Google Inc. and Blue Shift Inc. 2017] is three: T-mode with swapping, T-mode without swapping, and H-mode. The reason for the necessity of this swapping is to test which base color is suitable for the upper horizontal line of a “T” character with three paint colors (Figure 2-(a)). N_{Dist} is eight, as defined in the T-/H-distance look-up table [Ström and Pettersson 2007]. Thus, C_{TH} of *ETCPACK* with the fast option is $6C_{BC} + 72C_{EC}$.

4.2 Our approach

To remedy the performance issue, we design a new fast T-/H-mode compression algorithm. We describe our strategy to reduce the computational cost in Equation 1 as follows. First, we replace the 3D RGB space on pixel clustering with the 1D luma space, for minimizing C_{BC}. Second, we select a proper mode in advance, for decreasing N_{Mode} from three to one. Third, we reduce the number of

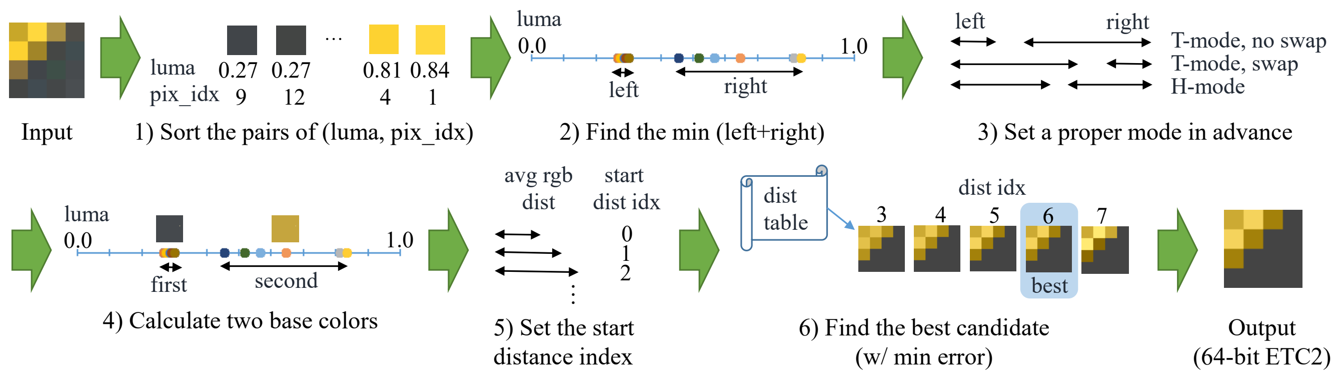


Fig. 4. An overview of the new fast T-/H-mode compression algorithm. The above six steps convert an RGB block into a 64-bit ETC2 block. Steps 1, 2, and 4 correspond to FindBaseColors() in Algorithm 1. Step 6 corresponds to CompressBlockT/H() in Algorithm 1. Steps 3 and 5 are additional steps for a speedup.

distance candidates of a block (N_{Dist}). Fourth, we implement the error calculation part using SSE/AVX2 intrinsics to exploit SIMD parallelism, and it results in a reduction in C_{EC}. Finally, we only use one base-color pair and do not allow additional iterations with multiple base-color pairs, thereby reducing N_{BC} to one.

Let us describe the detailed procedure of our approach. Note that the order of the following description will be the same as the step numbers in Figure 4.

First, we sort the pairs of a luma value and a pixel index in an input block, in ascending order of luma values. This sorting results in a single 1D line.

Second, to find two proper clusters, we calculate the minimum value of the 15 summed luma differences in the line. A summed luma difference means the luma difference in the left region plus that in the right region after dividing the pairs into the left and right regions. For the calculation, an iterator sweeps the line from left to right and finds the minimum value. We add small bonus factors to both ends of the line: 8, 4, 2, 2, 4, and 8 in the 8-bit fixed format to the first, second, third, 13rd, 14th, and 15th elements. Adding these bonus factors aims at preventing the two unwanted situations. First, the longer cluster covers too large color spaces, thereby increasing errors. Second, the iterator has a high possibility of selecting the left-most or right-most candidate as the optimal point because the shorter cluster with one element has always a “zero” difference. This zero difference can be incorrect after converting RGB888 to RGB444 on the ETC2 format.

Third, based on the two clusters generated from the above luma-based clustering, we set a proper compression mode of the block before error calculations. If the luma difference of the longer cluster is higher than that of the shorter one by a factor of two or more, we set the compression mode to the T-mode. Otherwise, we set the mode to the H-mode. If the left cluster is longer than the right one in the T-mode, we swap the two colors for further processing because the second base color is located on the upper horizontal line of a “T” character.

Fourth, we calculate two base colors from the two clusters. For the calculation, we apply different strategies for the two following cases. The first case is ranged paint colors, which corresponds to the second base color in the T-mode and both base colors in the H-mode. In this case, we first pick the midpoint RGB color of both ends of each cluster because it is suitable for representing symmetric ranges from the midpoint. We then clamp its RGB444 color to [1, 14] instead of [0, 15] because the midpoint on the minimum (0) or maximum (15) value halves its ranges. The second case is that a base color is equal to its paint color, which corresponds to the first base color in the T-mode. In this case, we average all the RGB colors in the cluster to get a base color. We then convert this color to RGB444 with a clamping range of [0, 15].

Fifth, to reduce the number of error-calculation iterations using the distance table in the ETC2 format (Table C.8 in the OpenGL ES 3.0 specification [Leech and Lipcha 2019]), we set the start distance index in advance, according to the average RGB distances of the two clusters. We set the start index to ‘the distance index corresponding to distance *d*’ minus ‘three’; the higher contrast, the higher start index. For example, if the average distance from a base color ranges from 24 to 32, its corresponding index in the table is five, so the start index selected by us is two. According to our experiments, our three-level earlier start is conservative enough to maintain compression quality.

Sixth, we find the best candidate with the minimum error by changing the distance value. For error calculation, we use the perceptual error metric [Ström and Akenine-Möller 2005] similar to *ETCPACK*. The differences between *QuickETC2* and *ETCPACK* in this step are the number of distance candidates and SIMD optimizations as follows. First, we stop further iterations if the current iteration does not decrease the error. This end-distance-index optimization is based on that the pattern of error values is usually V-curves (Figure 5). By combining with the start-distance-index optimization in the fifth step, we can effectively decrease the number of distance candidates without sacrificing image quality. Second, we deal with 16 pixels together using SSE/AVX2 intrinsics. We first calculate differences between the pixel colors and the four paint colors and obtain their luma errors. We then select the best paint color with the minimum error per pixel. For efficient SIMD processing, we convert 16 RGB888 colors into three __m256i variables (16 pixels × 16 bits per channel) and use the halved scaling factors (38, 76, and 14 for each RGB channel) for the error calculation to prevent overflows

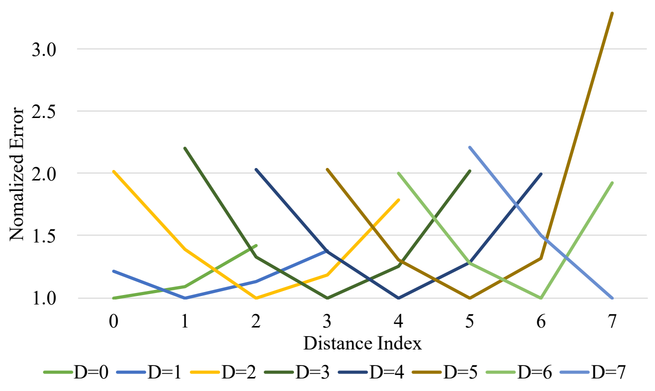


Fig. 5. Relationships between errors normalized to the lowest errors of each case and distance indices during the T-/H-mode compression on *Kodim05*. In this graph, we divide T/H blocks into eight cases according to their best distance index D and clamp each result to D±2 for better visibility. The V-curve patterns are the basis of our end-distance-index optimization.

of signed int16, similar to the ETC1 compression logic in *etcpak*. At the end of an iteration, we calculate the total block error value and compare it with the best error value from the previous iterations. If the best error value is changed, we update the best base colors and the corresponding pixel indices.

We finally pack the best candidate into a 64-bit T- or H-block. The reduced total cost of our approach ($C^{\prime}_{TH}$) described above is $C^{\prime}_{BC} + N^{\prime}_{Dist} \cdot C^{\prime}_{EC}$ ($2 \le N^{\prime}_{Dist} \le 8$), where $C^{\prime}$ and $N^{\prime}$ indicate a reduced cost and number.

5 EXPERIMENTS AND RESULTS

5.1 Test setup

For the comparison, we used the 64 textures which represent different types, sizes, and formats. The actual images of the entire set are available in the supplemental document. The sizes of the images range from 256×256 to 8192×8192. Nine images contain an additional alpha channel (No. 27, 38, 41, 42, 51, 52, 55, 56, and 57), and the others only contain three RGB channels.

We experimented on a desktop with an AMD Ryzen 7 3700X@3.6GHz 8-core (with hyper-threading) CPU and 32 GB of RAM. The operating system for our experiment is Ubuntu 20.04. For quality analysis on the RGB and RGBA textures, we used ImageMagick 7.0.10-28 and Icy 2.0.3.0 [De Chaumont et al. 2012], respectively.

In the next subsection, we will compare our compressor with *etcpak* 0.7, *Etc2Comp*, *ETCPACK* 4.0.1, and *astcenc* 1.7 [Arm Limited. 2020]. *ETCPACK* 4.0.1 is the latest version included in the Mali Texture Compression Tool 4.3.0 [Arm Limited 2016] and provides more features and higher speed than Ericsson's original version (v2.74) [Ericsson 2018]. *astcenc* is the reference ASTC encoder. We selected the fastest options for each compressor because our goal is real-time rates. We describe the detailed setting values of each compressor as follows.

- *etcpak*: ETC1, partial ETC2 (planar only)
- *QuickETC2* (ours): partial ETC2 (planar only), full ETC2
- *Etc2Comp*: effort = 0 & error metric = rgba
- *ETCPACK*: fast perceptual
- *astcenc*: very fast & block size = 4x4 (for RGBA) or 6x6 (for RGB)

The reason for choosing different ASTC block sizes for RGB and RGBA textures is fair comparisons between ASTC and ETC2. In terms of compression ratios, ETC2 RGB at 4 bits per pixel (BPP) is comparable to ASTC 6x6 at 3.56 BPP, and ETC1 and ETC2 RGBA at 8 BPP has the same bit rate as ASTC 4x4. When we compress an alpha texture in the ETC1 mode in *etcpak*, we compress the alpha channel in the same way as the RGB channels and store the result into a separated RGB texture. Thus, its compression ratio (4:1) is the same as that of ETC2 RGBA.

We evaluate the performance of each encoder with the two settings: highest performance and baseline control. In the former setting, we set the number of threads to 16 except for single-threaded *ETCPACK* and enabled the SIMD vectorization in *etcpak* and *QuickETC2*. In the last setting, we commonly used a single thread and disabled the vectorization for an algorithmic comparison.

5.2 Quantitative analysis of the results

Table 1 summarizes the experimental results. We separate the results of the RGB and RGBA textures because ETC1, ETC2 with EAC, and ASTC differently handle the alpha channel; ETC1 does not have any special mode for alpha compression, and ASTC provides higher compression ratios than EAC.

Let us first focus on the compression performance of the ETC compressors. Compared to *etcpak*'s ETC2 mode, *QuickETC2* in the partial ETC2 mode (planar only) achieves a 67% speedup on average and up to a 202% speedup (No. 51, *Vokselia Spawn*), as shown in Figure 7. On the RGB textures, QuickETC2 in the partial ETC2 mode shows almost similar performance to *etcpak* in the ETC1 mode, in contrast, the average performance drop of *etcpak*'s ETC2 mode is 37.8% compared to its ETC1 mode. These results indicate that our early compression-mode decision scheme absorbs most ETC2-planar overheads.

The addition of the T- and H-modes in *QuickETC2* increases the compression time by 33% on average. The overheads vary according to image characteristics, as illustrated in Figure 7, because a high proportion of high-contrast blocks in a texture leads to more T-/H-compression tests. Compared to the other tools that fully support the ETC2 modes, *QuickETC2* is two to three orders of magnitude faster, as shown in Table 1 and Figure 1. When we commonly disable multi-threading and SIMD optimizations as a baseline, ours is still one order of magnitude faster than them; these results prove that QuickETC2 is a lightweight method.

Next, we compare the quality of our compressor with other ETC compressors. For this comparison, we use the peak signal to noise ratio (PSNR) and the structural similarity (SSIM) index [Wang et al. 2004] (Table 1 and Figure 6). *QuickETC2* in the partial ETC2 mode shows similar or slightly higher levels of PSNR/SSIM values as *etcpak*. The full ETC2 support increases PSNR values up to 1.04 dB (No. 52, *Vector-Streets*) and 0.15 dB on average.

Compared to *Etc2Comp* and *ETCPACK*, *QuickETC2* in the full ETC2 mode shows relatively worse quality: 0.72-1.47 dB lower average PSNR values and 0.06-0.08 lower SSIM values than the two

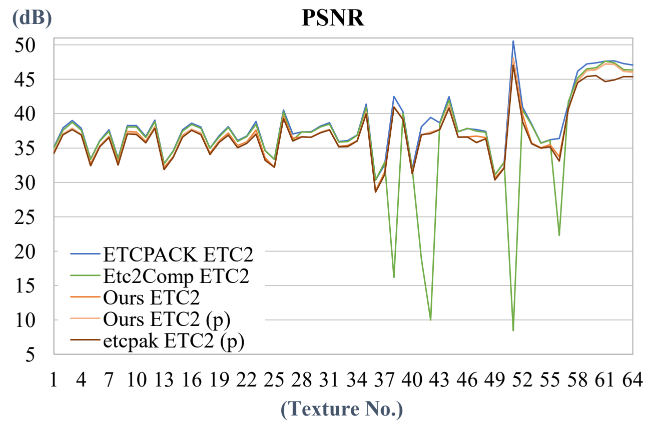
Table 1. Experimental results with the 64 test textures. PSNR and SSIM values are related to quality, and units of megapixels per second (Mpixels/s) are related to speed; the higher, the better. Note that ETC2 (p) denotes partial ETC2 support - planar only. HP and BC are the abbreviations of the highest-performance and baseline-control settings mentioned in Section 5.1, respectively.

Compressor	Codec	PSNR (dB)	SSIM	Mpixels/s HP	Mpixels/s BC
55 RGB Textures (ETC1/2: 4 BPP, ASTC 6x6: 3.56 BPP)					
etcpak	ETC1	35.87	0.949	3250	44.6
	ETC2 (p)	36.82	0.957	1931	28.9
QuickETC2 (ours)	ETC2 (p)	37.02	0.958	3196	45.8
	ETC2	37.17	0.958	2577	40.6
Etc2Comp	ETC2	37.89	0.965	3.8	3.1
ETCPACK	ETC2	38.09	0.966	1.4	1.4
astcenc	ASTC 6x6	38.28	0.966	11.5	1.5
9 RGBA Textures (ETC1/2 & ASTC 4x4: 8 BPP)					
etcpak	ETC1	36.19	0.914	3742	158
	ETC2 (p)	38.19	0.968	1784	24.2
QuickETC2 (ours)	ETC2 (p)	38.30	0.968	3147	45.1
	ETC2	38.47	0.969	2742	40.8
Etc2Comp	ETC2	25.31	0.787	5.0	4.3
ETCPACK	ETC2	39.94	0.975	1.1	1.1
astcenc	ASTC 4x4	45.01	0.986	22.5	4.1

encoders, except for the results of *Etc2Comp* on the RGBA textures. The main reason for the differences is the high-speed ETC1 compression logic in *etcpak*; because we do not alter the logic, ETC1 blocks compressed by *QuickETC2* and *etcpak* are the same. *Etc2Comp*'s low PSNR/SSIM values on the RGBA textures are caused by its compression policy; when the alpha value of a pixel on an RGBA texture is zero, *Etc2Comp* writes a black RGB color to the pixel regardless of the original RGB color. This issue has been fixed in Unity 2018.2 but still exists in the Google's original open-source version.

We also compare *QuickETC2* with the reference ASTC encoder *astcenc* because recent mobile devices support both ETC2 and ASTC. *astcenc* shows higher average PSNR and SSIM values than *QuickETC2*, as shown in Table 1. Especially, ASTC provides much higher alpha compression ratios (which are equal to higher quality at the same BPP) than ETC2. However, our qualitative comparisons on each image, which will be described in Section 5.3, show that each compressor has advantages and disadvantages. If compression speed is considered as another important factor, our compressor is competitive (Table 1 and Figure 1). To sum up, if compatibility or speed is a high priority, ETC2 compression using ours will be a better choice, and if scalability or effective alpha compression is important, ASTC compression using *astcenc* will be a good choice.

To find optimal threshold values (T_1, T_2, and T_3) in our early compression-mode decision, we had executed batch processing with the following ranges: $0.01 \leq T_1 \leq 0.05$, $0.03 \leq T_2 \leq 0.12$, and $0.26 \leq T_3 \leq 0.44$. As a result, we obtained the three graphs in Figure 8 and set the T_2 and T_3 to 0.09 and 0.38, respectively, because they are the best trade-offs between performance and quality. In contrast, we set the value



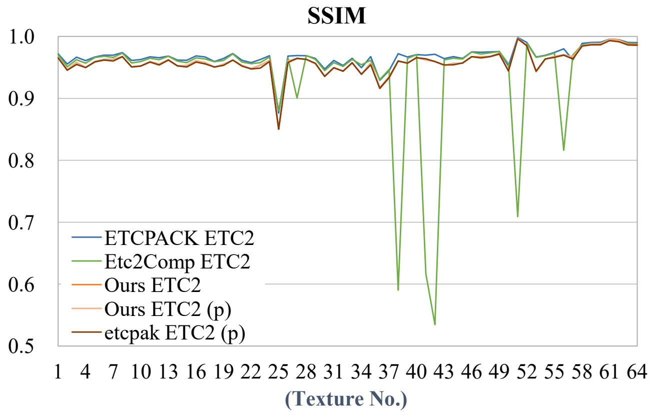


Fig. 6. Quality comparison of the ETC2 compressors by using PSNR and SSIM values. Several PSNR/SSIM drops of *Etc2Comp* are caused by its RGBA compression policy that ignores the original RGB colors at zero-alpha pixels. It may or may not affect texture quality on actual applications.

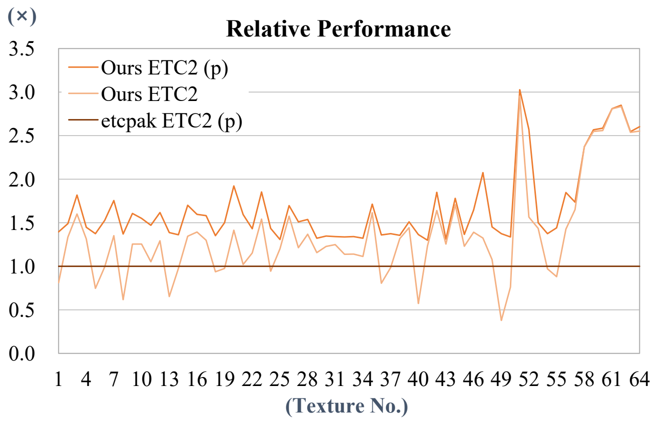


Fig. 7. Performance comparison between *etcpak* and *QuickETC2*. The results are normalized to *etcpak*. The similar patterns between the relative performance and PSNR/SSIM values imply that our early compression-mode decision adaptively controls performance and quality.

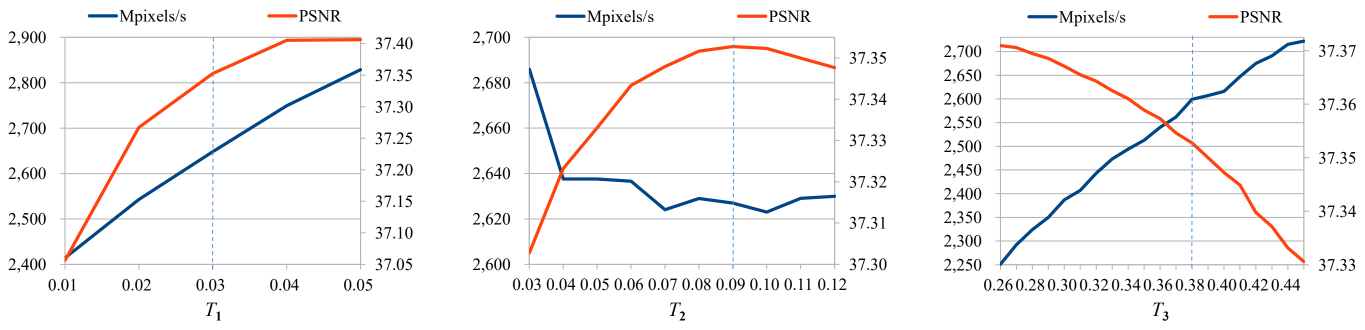


Fig. 8. Effects of three thresholds for the early compression-mode decision scheme: T_1, T_2, and T_3 mentioned in Section 3. The three dashed lines indicate our choice: T_1=0.03, T_2=0.09, and T_3=0.38. The reason for choosing the first threshold (T$_1$=0.03) is to prevent excessive blurring.

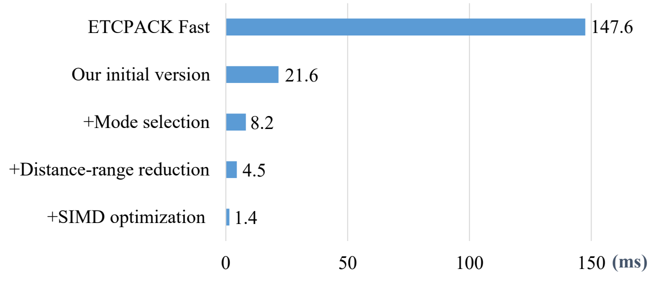


Fig. 9. T-/H-mode compression timings on *Kodim05*. With the early compression-mode decision scheme, only high-contrast blocks (luma_diff≥T_3=0.38) were injected into the T-/H-mode compression stage. We collected the results on a single thread, so the result in the last row (1.4 ms) is approximately 10 × slower than that with multi-threading in Figure 1 (0.13 ms).

of T_1 in another way. Even though the leftmost graph in Figure 8 indicates that 0.05 seems to be optimal, our choice is 0.03 because the higher values sometimes wipe details on some of the test images.

Finally, we compare the T-/H-mode compression functions in *QuickETC2* and *ETCPACK* v2.74. For this experiment, we had added *ETCPACK*'s T-/H-mode compression functions to our code and conducted respective tests with our function and *ETCPACK*'s functions. We also broke our function down to investigate its speedup factors (Figure 9). On average, our algorithm achieves 95 × faster speed with 0.005 dB higher PSNR values than *ETCPACK*'s algorithm. These results support that our cost estimation in Section 4.2 is reasonable, and the quality of our T-/H-mode compression on high-contrast blocks is comparable to that of *ETCPACK*.

5.3 Qualitative analysis of the results

According to Griffin and Olano [2015], texture quality judgments need to consider perceptual sensitivity to compression artifacts and geometric or texture-set masking effects; in other words, less conservative compression algorithms can be enough for actual rendering in many cases. Chait [2015] uses subjective visual image-quality (IQ) ratings based on visual examinations.

In that respect, we classify compression artifacts that appear in ETC2 into five categories: block artifacts, blurring, banding, color shifts, and loss of smooth anti-aliasing (AA) or gradients. Figure 10 illustrates representative examples of the five artifact types using magnified images, block maps, and ꟻLIP error maps [Andersson et al. 2020]. Table 1 in the supplemental document tabulates our detailed qualitative analysis of the results on each image. According to the results, our *QuickETC2* provides comparable quality to *ETCPACK* except for increased block artifacts on a few images and minor blurring in GIS maps.

Let us look at each case in Figure 10. The block artifacts in Figure 10-(a) occur when all pixels in a block have similar luma values (luma_diff<T_3=0.38), while their actual colors are quite different. In this case, *QuickETC2* does not pass the block into the T-/H-compression path and always compresses that in the ETC1 mode. Even if we insert the block to the T-/H-compression path, our luma-based algorithm may not find proper base colors in this case, and the final encode selector may eventually select the ETC1 block resulting in 4×2 or 2×4 block patterns.

In contrast to the above case, the block artifacts in Figure 10-(b) are caused by T- or H-blocks. Because *etcpak*'s high-speed ETC1 compression results in relatively lower-quality ETC1 blocks, the final encode selector in Figure 3-(right) sometimes selects a T- or H-mode block instead of an ETC1 block, in contrast to *ETCPACK*. As a result, larger-sized block artifacts occur on the image. This issue can be alleviated by high-quality ETC1 compression.

Blurring in Figure 10-(c) is caused by the false-positive of our early compression-mode decision scheme. If the scheme misjudges that the inside in a planar-mode block is gradually changed, this misjudgment can blur some thin texts or geometry with pale colors.

As shown in Figure 1, the planar mode can usually handle banding problems that appear in ETC1. However, if there are block-wise gradients (a solid color per block) in sparse regions on a high-resolution captured image (Figure 10-(d)), the ETC2 codec cannot make smooth gradients. However, the banding in ETC2 is much less than that in ETC1.

Color shifts (Figure 10-(e)) and loss of smoothness (Figure 10-(f)) are common problems in ETC1/2. The reason for these types of artifacts is the quantization in the ETC format. A color shift often occurs in grey or yellow colors which can be expressed as slightly purple or greenish after compression.

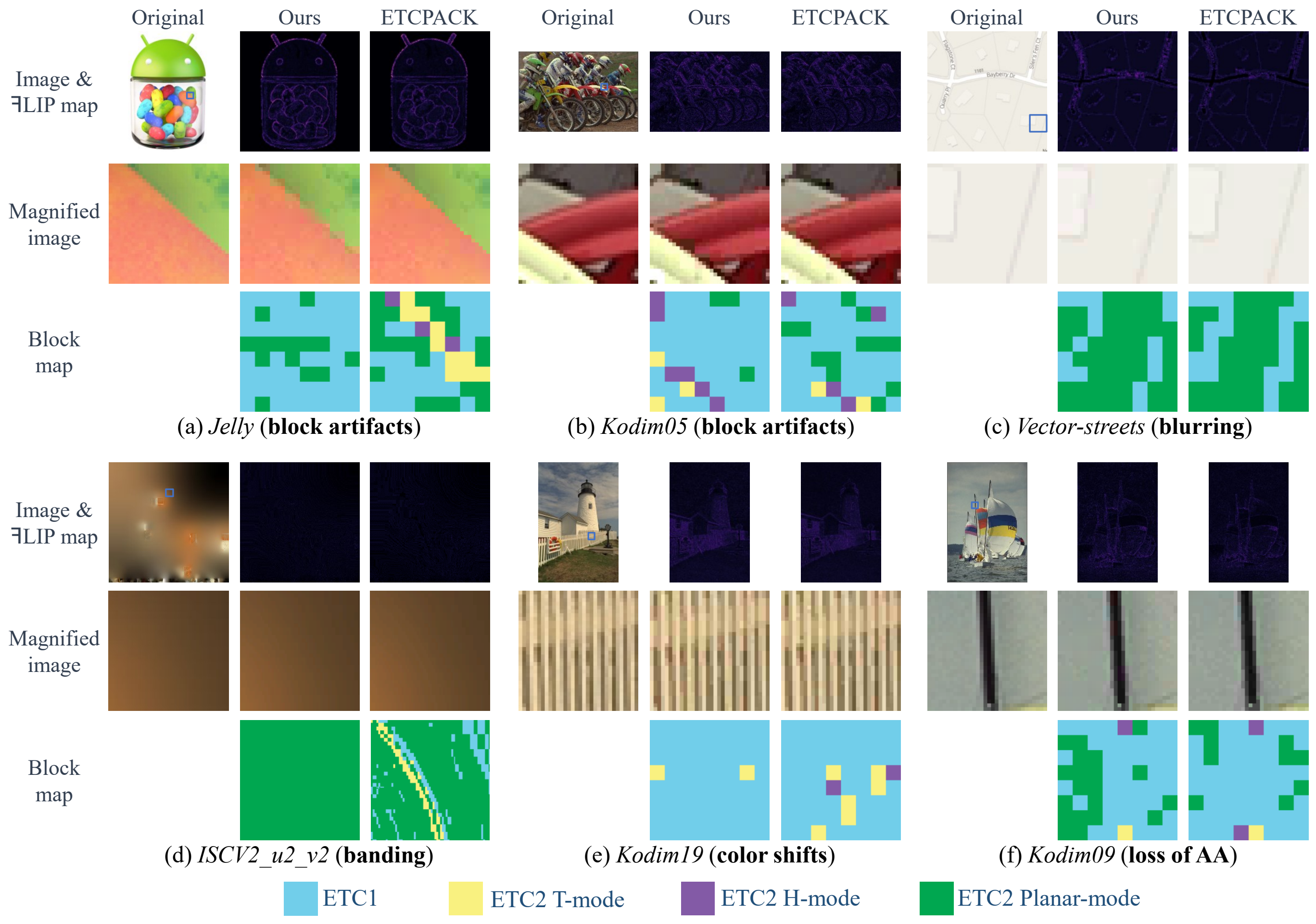

Original
Ours
ETCPACK
Image & ꟻLIP map
Magnified image
Block map
(a) *Jelly* (**block artifacts**)
(b) *Kodim05* (**block artifacts**)
(c) *Vector-streets* (**blurring**)
(d) *ISCV2_u2_v2* (**banding**)
(e) *Kodim19* (**color shifts**)
(f) *Kodim09* (**loss of AA**)
ETC1
ETC2 T-mode
ETC2 H-mode
ETC2 Planar-mode

Fig. 10. Examples of compression artifacts: the upper cases (a)-(c) only occur in ours, and the lower cases (d)-(f) occur in both ours and *ETCPACK*. Block artifacts are usually distinguishable, but the others are relatively minor. The ꟻLIP maps [Andersson et al. 2020] were generated at $p = 33.5$ pixels per degree. ©Kodak, UNC GAMMA Lab, Google, and *fhernand*.

To sum up, compared to block artifacts, the other types of artifacts are minor in ETC2 encoding. The ꟻLIP maps in Figure 10 support that claim. Our T-/H-mode compression reduces visible block artifacts in many high-contrast regions (Figure 1), thereby making up for *etcpak*'s compression quality.

Finally, our compressor shows comparable quality to *astcenc* with the fastest setting (Figure 1). Ours produces neither faded colors (*Kodim05*) nor ringing artifacts (*Kodim20*) shown in *astcenc*. On the other hand, *astcenc* achieves almost perfect edge handling and no color shifts (*Small-Char*), and its gradients are also slightly better than those in ETC2 (*ISCV2_u2_v4*). PSNR/SSIM values do not reveal these characteristics over different codecs well.

6 CONCLUSIONS AND FUTURE WORK

We have presented two approaches for fast ETC2 compression: an early compression-mode decision scheme and a new T-/H-mode compression algorithm. Both approaches exploit the luma difference in a block for faster processing. With the AVX2 implementation, we have achieved two to three orders magnitude faster performance than the existing high-quality ETC2 compressors that fully support the ETC2 mode.

One of the limitations of our approach is that it does not affect the ETC1 and EAC compression logic in *etcpak*. As a result, when compressing one- or two-channel textures, such as normal maps or lightmaps, our approach improves neither the compression quality nor speed of the EAC compression. The high-speed ETC1 logic in *etcpak* is the main reason of quality gaps between ours and other high-quality compressors. Thus, additional improvements over the ETC1 and EAC compression functions would help to achieve higher-quality compression.

Our current implementation only targets at x86 platforms. We are interested in extending that to other platforms. First, porting of our code to ARM Neon will increase its utility in mobile apps that require real-time texture compression. Second, efficient ETC2 compression on GPUs is an interesting topic for future work. There are a few CUDA-accelerated texture encoders, such as the NVIDIA texture tools exporter [NVIDIA 2020], but all of them do not support the ETC2 format. We believe that our algorithm would also be easily parallelized using CUDA or OpenCL because each block is independently compressed. If the GPU implementation provides enough performance for real-time rendering, we will be able to consider using the ETC2 codec for streaming G-buffer compression [Kerzner and Salvi 2014], which can reduce the amount of memory traffic caused by deferred shading. Finally, if we exploit libraries or languages for advanced automatic vectorization, such as Enoki [Jakob 2020] or Halide [Ragan-Kelley et al. 2013], we expect not only to greatly increase the number of supported platforms but also to further increase compression performance by utilizing heterogeneous computing.

REFERENCES

Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, and Kalle Åströmand Mark D. Fairchild. 2020. ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques (HPG 2020)* 3, 2, Article 15 (2020), 23 pages.

Arm Limited. 2016. *Mali Texture Compression Tool.* https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-texture-compression-tool/downloads

Arm Limited. 2020. *astc-encoder.* https://github.com/ARM-software/astc-encoder/tree/1.x

Binomial LLC. 2020. *Basis Universal Supercompressed GPU Texture Codec.* https://github.com/BinomialLLC/basis_universal

David Chait. 2015. *Using ASTC Texture Compression for Game Assets.* https://developer.nvidia.com/astc-texture-compression-for-game-assets

Fabrice De Chaumont, Stéphane Dallongeville, Nicolas Chenouard, Nicolas Hervé, Sorin Pop, Thomas Provoost, Vannary Meas-Yedid, Praveen Pankajakshan, Timothée Lecomte, Yoann Le Montagner, Thibault Lagache, Alexandre Dufour, and Jean-Christophe Olivo-Marin. 2012. Icy: an open bioimage informatics platform for extended reproducible research. *Nature methods* 9, 7 (2012), 690–696.

Marc Fauconneau Dufresne. 2015. How to create a high quality, fast texture compressor using ISPC. In *Game Developer Conference 2015.* https://software.intel.com/sites/default/files/managed/4a/38/High-Quality_Fast-DX11-Texture-Compression.pdf

Jim Easterbrook, Oliver Grau, and Peter Schubel. 2010. A system for distributed multi-camera capture and processing. In *Proceedings of the 2010 Conference on Visual Media Production.* IEEE, 107–113.

Ericsson. 2018. *ETCPACK.* https://github.com/Ericsson/ETCPACK

Simon Fenney. 2003. Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware.* 84–91.

Google Inc. and Blue Shift Inc. 2017. *Etc2Comp - Texture to ETC2 compressor.* https://github.com/google/etc2comp

Wesley Griffin and Marc Olano. 2015. Evaluating texture compression masking effects using objective image quality assessment metrics. *IEEE Transactions on Visualization and Computer Graphics* 21, 8 (2015), 970–979.

Intel Corp. 2019. *Fast ISPC Texture Compressor.* https://github.com/GameTechDev/ISPCTextureCompressor

Konstantine I Iourcha, Krishna S Nayak, and Zhou Hong. 1999. System and method for fixed-rate block-based image compression with inferred pixel values. US Patent 5,956,431.

Wenzel Jakob. 2020. Enoki: structured vectorization and differentiation on modern processor architectures. https://github.com/mitsuba-renderer/enoki.

Brano Kemen. 2012. In-Game Video Capture with Real-Time Texture Compression. In *OpenGL Insights*, Patrick Cozzi and Christophe Riccio (Eds.). CRC Press, 455–466. http://www.openglinsights.com/.

Ethan Kerzner and Marco Salvi. 2014. Streaming G-buffer compression for multi-sample anti-aliasing. In *Proceedings of High Performance Graphics 2014.* 1–7.

Khronos Group. 2019. *Khronos Data Format Specification, Version 1.3, Revision 1.* https://www.khronos.org/registry/DataFormat/specs/1.3/dataformat.1.3.pdf

Evgeny Kluev. 2014. *Stack Overflow - Horizontal minimum and maximum using SSE.* https://stackoverflow.com/questions/22256525/horizontal-minimum-and-maximum-using-sse

Pavel Krajcevski, Adam Lake, and Dinesh Manocha. 2013. FasTC: accelerated fixed-rate texture encoding. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.* 137–144.

Jon Leech and Benj Lipcha. 2019. *OpenGL© ES Version 3.0.6 (November 1, 2019).* https://www.khronos.org/registry/OpenGL/specs/es/3.0/es_spec_3.0.pdf

Yoseph Linde, Andres Buzo, and Robert Gray. 1980. An algorithm for vector quantizer design. *IEEE Transactions on Communications* 28, 1 (1980), 84–95.

Colt McAnlis. 2016. *Building a blazing fast ETC2 compressor.* https://medium.com/@duhroach/building-a-blazing-fast-etc2-compressor-307f3e9aad99#.acqks0pzct

Microsoft. 2018. *Texture Block Compression in Direct3D 11.* https://docs.microsoft.com/en-us/windows/win32/direct3d11/texture-block-compression-in-direct3d-11

Jae-Ho Nah. 2020. QuickETC2: How to finish ETC2 compression within 1 ms. In *ACM SIGGRAPH 2020 Talks.* Article 4.

Jae-Ho Nah, Byeongjun Choi, and Yeongkyu Lim. 2018. Classified texture resizing for mobile devices. In *ACM SIGGRAPH 2018 Talks.* Article 73.

NVIDIA. 2020. *NVIDIA Texture Tools Exporter.* https://developer.nvidia.com/nvidia-texture-tools-exporter

Jorn Nystad, Anders Lassen, Andy Pomianowski, Sean Ellis, and Tom Olson. 2012. Adaptive scalable texture compression. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on High-Performance Graphics.* 105–114.

Daniel Oom. 2016. *Real-Time Adaptive Scalable Texture Compression for the Web.* Master's thesis. Chalmers University of Technology.

T. Paltashev and I. Perminov. 2014. Texture Compression Techniques. *Scientific Visualization* 6, 1 (2014), 106–146.

Daniel Pohl, Daniel Jungmann, Bartosz Taudul, Richard Membarth, Harini Hariharan, Thorsten Herfet, and Oliver Grau. 2017. The next generation of in-home streaming: Light fields, 5K, 10 GbE, and foveated compression. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS).* IEEE, 663–667.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

Mark Segal and Kurt Akeley. 2013. *The OpenGL© Graphics System: A Specification (Version 4.3 (Core Profile) - February 14, 2013).* https://www.khronos.org/registry/OpenGL/specs/gl/glspec43.core.pdf

Jacob Ström and Tomas Akenine-Möller. 2005. iPACKMAN: High-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware.* 63–70.

Jacob Ström and Martin Pettersson. 2007. ETC 2: texture compression using invalid combinations. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware.* 49–54.

Bartosz Taudul and Daniel Jungmann. 2020. *etcpak.* https://bitbucket.org/wolfpld/etcpak/src/master

Unity Technologies. 2017. *Unity User Manual (2017.3).* https://docs.unity3d.com/2017.3/Documentation/Manual/class-EditorManager.html

Alex Voica. 2013. *Taking texture compression to a new dimension with PVRTC2.* https://www.imgtec.com/blog/pvrtc2-taking-texture-compression-to-a-new-dimension

Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612.

A SUPPLEMENTAL MATERIALS

We attach a supplemental document including the actual images of the entire set and a detailed qualitative analysis table. We also attach a diff patch file, which can be directly applied to *etcpak* 0.7 (the parent commit is 5c1021b), and its executables compiled for Linux and Windows. We hope that these attachments could be helpful to software developers and graphics researchers.

ACKNOWLEDGMENTS

We appreciate the reviewers who gave us a lot of constructive suggestions. We also thank the authors of *etcpak* and *ETCPACK* who have opened their source code to the public. Finally, we are grateful to Jiwoon Hwang, Yongbong Choi, and Seongeun Soh for their support.