Pacific Graphics 2023
R. Chaine, Z. Deng, and M. H. Kim
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 7

H-ETC2: Design of a CPU-GPU Hybrid ETC2 Encoder

H. Lee[1] and J.-H. Nah[†1]

[1]Sangmyung University, South Korea

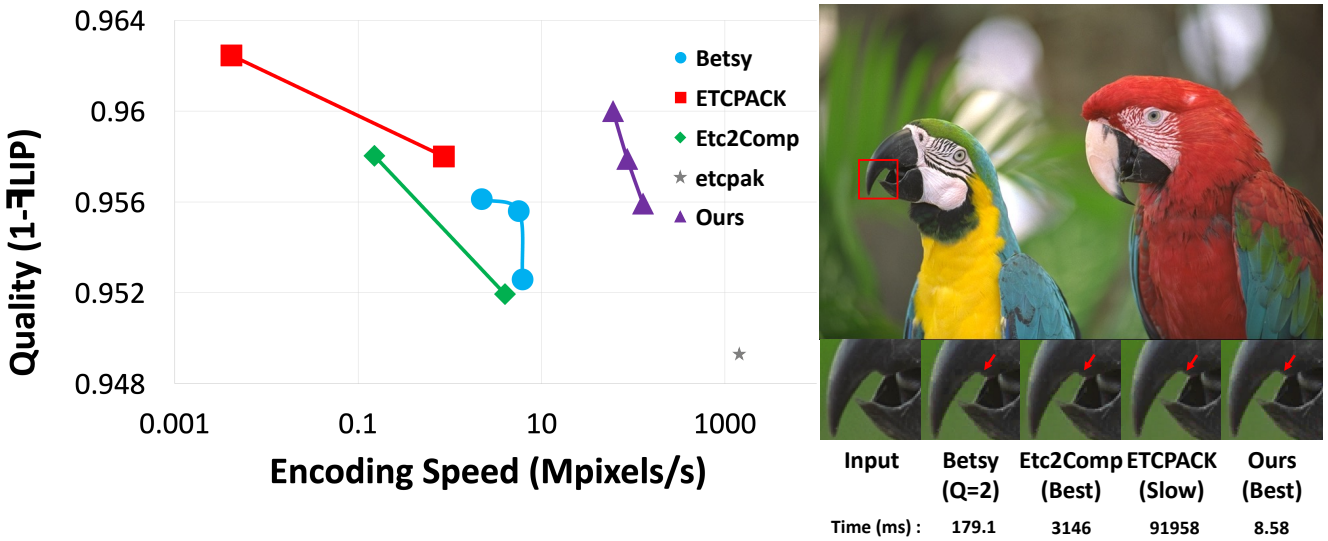

Figure 1: *Our H-ETC2 encoder is one to four orders of magnitude faster than previous high-quality ETC2 encoders, such as ETCPACK [Eri18], Etc2Comp [GB17], and Betsy [Gol22], while delivering a level of quality comparable to that of ETCPACK's slow mode. The right image is taken from kodim23, which is part of the Kodak Lossless True Color Image Suite. Please zoom in the image to distinguish the difference in quality.*

Abstract

This paper proposes a novel CPU-GPU hybrid encoding method based on the ETC2 format, commonly used on mobile platforms. Traditional texture compression techniques often face a trade-off between encoding speed and quality. For a better trade-off, our approach utilizes both the CPU and GPU. In a pipeline we designed, the CPU encoder identifies problematic pixel blocks during the encoding process, and the GPU encoder re-encodes them. Additionally, we carefully improve the base CPU and GPU encoders regarding encoding speed and quality. As a result, our encoder minimizes compression artifacts, increases encoding speed, or achieves both of these goals compared to previous high-quality offline ETC2 encoders.

CCS Concepts

• ***Computing methodologies*** → ***Image compression;***

1. Introduction

High-quality computer graphics are increasingly important in various fields, such as games, movies, and virtual/augmented reality. Texture mapping plays a crucial role in achieving this quality. Advancements in hardware and algorithms have made it possible to employ complex rendering algorithms with high-resolution textures in real-time applications. However, the utilization of multiple high-resolution textures necessitates greater memory capacity and bandwidth, which can result in decreased performance or increased power consumption. As a solution of the problems, textures are typically stored in memory as a compressed format [PP14].

† Corresponding author

BC [Mic18], ETC1/2 [SAM05, SP07], and ASTC [NLP*12] are standard texture compression formats, all of which employ block compression schemes. The BC series is commonly used on desktop platforms that rely on DirectX and OpenGL, while ETC1/2 and ASTC are preferred for mobile platforms based on OpenGL ES and Vulkan. GPUs typically include several decoders for these formats, enabling real-time texture decoding. Conversely, texture encoding is primarily performed offline, and compressed textures included in an application are uploaded to a GPU when the application is running. In the case of recent AAA games where the total texture sizes can range from tens to hundreds of gigabytes, insufficient encoding speed can become a bottleneck during software development. Furthermore, if a texture compression format is used for low-latency video coding, the encoding process must be completed in real-time as indicated by Žádník, et al. [vMVJ22].

There exists a trade-off between encoding speed and quality in texture encoding. Achieving the closest color approximation to the original color within a compression format is an optimization problem. Widening the search range to minimize error values inherently results in increased encoding time. Conversely, prematurely terminating the encoding process for improved encoding speed can result in noticeable artifacts after compression.

In order to enhance encoding speed without compromising compression quality,we propose a CPU-GPU Hybrid ETC2 (H-ETC2) encoding method in this paper. Our objective is to partition the workload in texture encoding tasks between CPUs and GPUs, maximizing encoding speed on the hybrid encoder. This can be achieved by leveraging the distinct parallelism capabilities of modern CPU and GPU architectures. Modern CPUs can efficiently handle specific operations with fine-grained data-dependent branching in parallel by utilizing single-instruction multiple-data (SIMD) intrinsics [WWB*14]. On the other hand, high-performance GPUs generally offer superior computing power compared to CPUs, but they can experience performance degradation in scenarios involving high single-instruction multiple-threads (SIMT) execution divergence [PBD*10].

To leverage the strengths of both CPUs and GPUs, we have developed an encoding pipeline that combines two ETC2 encoders—one CPU-based and one GPU-based. In the initial stage on the CPU side, we utilize the etcpak encoder [Tau22] to swiftly compress a texture. While this encoder now supports all ETC2 modes through the integration of QuickETC2 [Nah20a], its heuristic mode selection and limited search ranges can occasionally introduce artifacts that are not present in other high-quality encoders. To mitigate this issue, we identify a set of problematic blocks with error values exceeding a threshold and transfer them to the GPU for further processing. On the GPU side, we employ the OpenGL-based Betsy encoder [Gol22] to recompress these blocks. The Betsy encoder utilizes parallel processing by creating multiple threads of potential encoding combinations and concurrently processing them on the GPU at high speed. This makes it suitable for enhancing the compression quality and resolving the artifacts introduced by the CPU-based encoder.

To validate the effectiveness of our hybrid method, we conducted experiments on the test set used in QuickETC2 [Nah20a] and compared ours to other high-quality encoders such as ETCPACK [Eri18], Etc2Comp [GB17], and Betsy [Gol22]. According to the results (shown in Figure 1), our encoder outperforms them in terms of speed by several orders of magnitude while achieving similar or better quality. Although our encoder is slower than the currently fastest encoder, etcpak [Tau22], it effectively mitigates many artifacts that appear in etcpak.

2. Related Work

2.1. ETC Format and Encoders

Ericsson Texture Compression (ETC) is a mobile-standard format that originated from PACKMAN [SAM04]. PACKMAN is a lossy texture compression technique designed for mobile platforms. It effectively compresses 2×4 pixel blocks into 32 bits to cater to mobile platforms with limited memory bandwidth. *i*PACKMAN [SAM05], also known as ETC1, introduces a more complex bit arrangement compared to PACKMAN. It compresses two 8-pixel subblocks together in 64 bits, and each subblock can be configured as 2×4 or 4×2 by using the flip bit. The base colors of the subblocks can be stored individually in RGB444 or differentially (RGB555 and dRdGdB333) by utilizing the diff bit. While maintaining the same compression ratio as PACKMAN, *i*PACKMAN enhances the quality of compressed textures.

ETC2 [SP07] introduces three additional modes that utilize invalid bit sequences to address problematic blocks caused by the horizontal/vertical subblock pattern in ETC1. The T- and H-modes [PS05] employ pixel clustering to derive two base colors and calculates the distance vector based on the small look-up table, resulting in a total of four palette colors. Although there are fewer palette colors per block compared to ETC1, these modes effectively reduce block artifacts caused by subblock patterns. On the other hand, the Planar mode utilizes interpolation to find smoothly varying chrominances and quantizes them in RGB676, thereby mitigating the banding artifacts present in ETC1. To compress the alpha channel in an RGBA texture, the EAC [SA13] codec is used with the ETC2 RGB codec. Operating independently from ETC1/2, the EAC codec employs table-based alpha compression. EAC compression can also be applied for compressing textures with one- or two-channels.

Ericsson has released a reference encoder called ETCPACK [Eri18] for compressing ETC1/2 textures. This encoder performs compression on a block-by-block basis for each ETC mode and selects the mode with the smallest error value to encode the block. When calculating the error value, the encoder offers an option to apply the same weight to RGB channels or to use perceptual weights based on luma (linear luminance) calculation. The latter option may result in a decrease in PSNR values using the same RGB weights, but it can enhance the clarity of edges in different color areas. Furthermore, the ETCPACK encoder provides a speed option that allows users to choose between fast and slow modes. Opting for the slow option yields higher quality results with a broader range of search settings, although it comes at the expense of significantly increased encoding time.

Google's Etc2Comp [GB17] encoder offers users an effort parameter that enables precise control over the trade-off between encoding speed and quality. Image blocks are scored and sorted based

on their mean squared error (MSE) values, after which block refinement takes place using the percentage specified by the effort parameter. Notably, Etc2Comp also supports the utilization of multiple CPU threads during compression, distinguishing it from ETCPACK in terms of parallel processing capabilities.

In contrast to the previous encoders, etcpak [Tau22] was developed with a primary focus on encoding speed. It minimizes the search scope for palette colors, extensively optimizes the code using SIMD intrinsics, and provides a scalable multi-threaded implementation. By doing so, etcpak achieves several orders of magnitude faster encoding speed than ETCPACK and Etc2Comp.

QuickETC2 [Nah20b, Nah20a] enhances the compression quality by introducing T-/H-mode compression logic, which was not present in etcpak 0.7. It also improves compression speed by employing a heuristic that selects one or two modes in advance based on the block's luma difference, rather than performing compression for all ETC1/2 modes. This approach leverages the suitability of the Planar mode for blocks with low contrast and the T- or H-mode for blocks with high contrast. The QuickETC2 patch was integrated into etcpak 1.0, which is the foundation of the CPU component in our hybrid system.

In contrast to the CPU encoders mentioned earlier, the Betsy GPU compressor [Gol22] takes advantage of the compute shader in OpenGL for encoding. This encoder aims to deliver high-quality compression for BC and ETC series at fast speeds by leveraging the computational power of GPUs. The ETC1 mode code in Betsy is based on rg_etc1 by Rich Geldreich [Ric12], and it assigns all potential subblock candidates to individual GPU threads to perform refinement operations in parallel. For the ETC2 compressor, Betsy utilizes an open-source implementation by Jean-Philippe ANDRE [AND14]. In the T-/H-mode, Betsy performs k-means clustering on 16 pixels, considering 120 possible combinations. It then conducts compression in the T-mode with and without swapping, as well as the H-mode, to find the results with the lowest errors. In the Planar mode, the internal pixel operations are parallelized, and an optimized gradient is obtained through single linear regression. Similar to ETCPACK and Etc2Comp, Betsy selects the block with the smallest error from each mode for the final compression. When developing the GPU encoder for our hybrid system, we referenced the Betsy GPU compressor due to its relatively fast compression speed compared to other high-quality CPU encoders, while still delivering acceptable compression quality.

2.2. CPU-GPU Hybrid Techniques

The use of both CPUs and GPUs together for computation has been extensively researched in the field of high-performance computing [MV15]. However, effectively harnessing the potential of heterogeneous architectures to improve the performance of a specific task is not straightforward. It requires implementing algorithms that take into account the unique characteristics of each processing unit (PU), appropriately distributing the workload across the PUs, and carefully designing interfaces between the units. Aiming to simplify the development process for heterogeneous computing architectures, Intel has developed the OneAPI programming model [Int22].

Our focus will be on graphics-related approaches that utilize both CPUs and GPUs, rather than high-performance computing. This focus allows us to gain valuable insights into optimizing performance in graphics-intensive applications.

Real-time ray tracing, historically constrained by intensive computational requirements, is a prominent research domain. Nah et al. [NKL*10] introduced an interactive mobile ray tracer using OpenGL ES. The CPU manages tasks like kd-tree construction and ray generation, while the GPU takes on ray traversal and shading. In contrast, the HART architecture [NKP*15] employs a hardware-accelerated strategy. The CPU handles BVH rebuilding, whereas BVH refitting and ray traversal occur asynchronously through the GPU's dedicated fixed-function hardware logic. Additionally, Barringer et al. [BAAM17] presented a ray accelerator that efficiently leverages combined CPU and integrated GPU processing capabilities via shared memory utilization.

The simultaneous utilization of CPUs and GPUs for generating high-quality ray-traced images has been a subject of study. Pajot et al. [PBPP11] proposed an algorithm that facilitates cooperative bidirectional path tracing between the CPU and GPU. Additionally, researchers have explored techniques to accelerate collision detection using heterogeneous resources. The HPCCD method [KHH*09] achieves real-time frame rates by utilizing the CPU for BVH updates and culling, while offloading elementary tests to the GPU.

3. System Overview

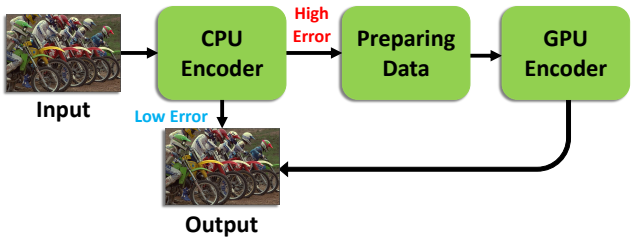

Figure 2: *A flow chart of our encoding pipeline. The green blocks and the black arrows represent the steps being performed and the data movement, respectively.*

In this section, we will discuss the design of our texture encoding pipeline. Our pipeline aims to preserve as much visual detail from the input image as possible while ensuring a fast encoding process. To achieve this, we focused on developing a hybrid pipeline that utilizes the strengths of both CPUs and GPUs. Figure 2 illustrates the flow chart of our encoding pipeline. When an input image is received for compression, it undergoes a series of stages.

In the first step, the CPU encoder utilizes luma (linear luminance) values to perform the initial encoding operation. While this luma-based approximation enables fast encoding, it can result in high errors in certain pixel blocks, leading to various compression artifacts. To address this drawback, pixel blocks with high errors are stored in the thread-local buffers to be used as input for the GPU encoder. Conversely, pixel blocks with low errors are stored

in the CPU memory and contribute to the final encoding results (Section 3.1).

In the second step, we repurpose pixel blocks with high errors that have accumulated in the thread-local buffers. Firstly, these blocks are consolidated into a single array. Then, sorting and cutting operations are performed, considering the selected quality mode. Finally, the resulting data, in the form of an image format, is prepared as input for the GPU encoder (Section 3.2).

The final step entails re-encoding the pixel blocks generated in the previous step. The GPU encoder, implemented using the compute shader of the OpenGL API, enhances the quality of the pixel blocks by leveraging the parallel capabilities of the GPU for extensive RGB space searching and iterations (Section 3.3). Once the GPU encoding process is finished, we can obtain the final encoding results by merging the results from the CPU and the GPU. Each of these steps will be further elaborated in the subsequent subsections.

3.1. Design of the CPU Encoder

We implemented our CPU encoder based on etcpak 1.0 [Tau22] with QuickETC2 [Nah20a], as mentioned in Section 2. It is important to note that the flow chart depicted in Figure 3 is executed independently on each CPU thread. The main difference between QuickETC2 and ours is the final block re-calculation stage and an additional buffer for further GPU encoding.

The original QuickETC2 method utilizes luma values to achieve fast encoding speed. Its early compression-mode decision scheme selects the appropriate compression mode in advance based on the luma difference for each block. Additionally, it performs clustering based on per-pixel luma values within each block for T-/H-mode compression. However, since this approach involves dimension reduction from the 3D RGB space into the 1D luma space, it can lead to inappropriate encoding results, resulting in a loss of quality.

Let's consider the two points $D_1(255,0,0)$ and $D_2(0,128,9)$ in the RGB space, as depicted in Figure 4. We can utilize Equation 1 to convert the 3D RGB-space data as 1D luma-space data.

$$luma = 0.3 \times R + 0.59 \times G + 0.11 \times B \tag{1}$$

The equation employed to reduce the dimension from RGB space to luma space remains the same as the one used in QuickETC2. While the two points in the RGB space exhibit significant differences, they become quite similar in the luma space. However, due to the QuickETC2 method perceiving the difference between the two points as similar, this discrepancy can result in incorrect mode or cluster selection. As a consequence, artifacts may manifest in the encoding results. To identify these problematic pixel blocks, we recalculate the error between the original pixel block and the compressed pixel block using Equation 2.

$$error = \sum_{i=0}^{N-1} \max\left(\left|\bar{x}_{i,r} - x_{i,r}\right|, \left|\bar{x}_{i,g} - x_{i,g}\right|, \left|\bar{x}_{i,b} - x_{i,b}\right|\right)^2 \tag{2}$$

In the equation, N represents the number of pixels in the pixel block, x denotes the pixel in the original block, and $\bar{x}$ represents the pixel in the compressed block. We determine the pixel block error by selecting and accumulating the largest difference value for each channel in the RGB space. This approach is especially effective in detecting errors in the red or blue channel since the green channel in Equation 1 carries the highest weight for the luma conversion.

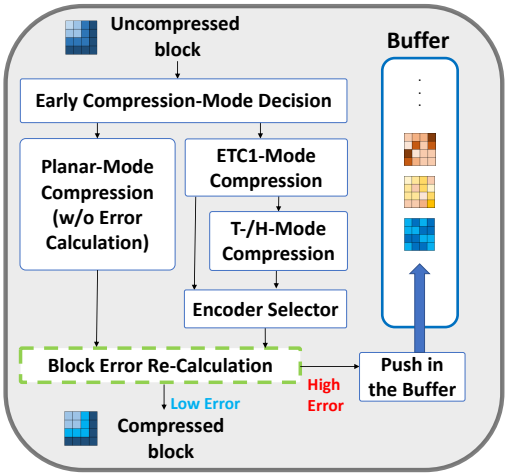

Figure 3: *A flow chart of our CPU encoder, which includes an additional error re-calculation process integrated into the QuickETC2 implementation. This process determines whether to retain the encoding result based on the calculated block error during the encoding process (indicated by the green dot box). We have also incorporated a local buffer for each thread to accumulate problematic pixel blocks.*

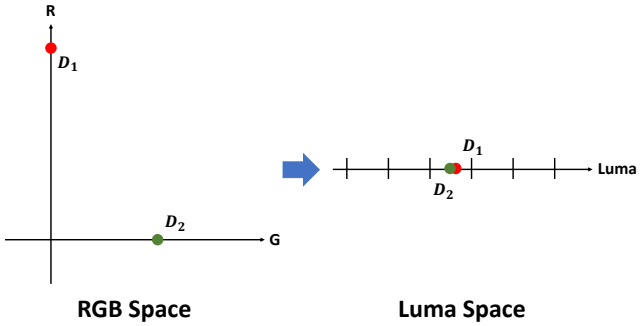

Figure 4: *An example of incorrect encoding results after dimension reduction. The left side of the figure displays the data represented along the R and G axes in the RGB space, and the data points appear widely scattered from each other in this space. However, when the same data is represented in the luma space, shown on the right side, the data points appear much closer to each other. This indicates that the luma space representation can fail to capture the significant differences present in the original RGB space.*

To determine the problematic pixel blocks based on the recalculated error values, we employ the following strategy. Initially, we establish a threshold value denoted as T, which assists in deciding whether a pixel block should be utilized as a GPU input. If the error of the specific block exceeds T, it is accumulated in the local buffer designated for each thread. Conversely, if the error is below T, the encoding result is stored in the CPU memory. We set the threshold

value T corresponding to the *dblimit* value for the medium preset (PSNR 35.68 dB) in the reference ASTC encoder [Smi18].

We also note that we have disabled the solid-color check function in etcpak 1.0 (CheckSolid_AVX2()). The role of this function is to quickly compress a block with a single solid color using the ETC1 logic before the early compression-mode decision stage. This is effective in reducing encoding time for backgrounds, but it can cause banding artifacts due to RGB555 quantization. To minimize the number of problematic blocks transferred to the GPU, we have decided to disable this function during ETC2 compression, as implemented in etcpak 0.7.

3.2. Preparing Data for the GPU Encoder

As mentioned earlier, we need to process problematic pixel blocks on the CPU encoder so that the GPU encoder can re-encode them. There are several factors to consider in this process. First, the amount of work processed on the GPU should be properly determined to strike a balance between encoding quality and performance. We also need to determine how to design the interface between the CPU and GPU to minimize communication overhead between them. Additionally, since the number of problematic pixel blocks varies for each image to be compressed, dynamic memory allocation is required.

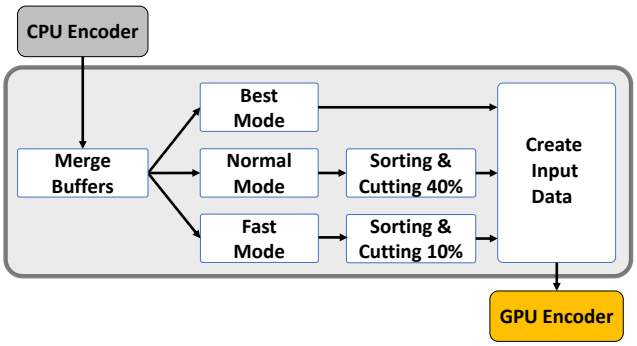

Figure 5: *A flow chart of the process of preparing input data for the GPU encoder with problematic pixel blocks determined by the threshold T. The three quality modes determine how many pixel blocks will be fed to the GPU for re-encoding them.*

Taking into consideration the factors described above, we have developed a process between the CPU and GPU that operates differently in three modes (Figure 5), drawing inspiration from Etc2Comp [GB17]. In Etc2Comp, a perceptual score is calculated based on the mean square error (MSE) for each block at the end of each encoding iteration. Subsequently, the blocks are sorted based on their perceptual score. The compression level is then determined by the value of the effort parameter, which represents the percentage of blocks that will undergo further refinement in the next iteration. As a result, developers are able to control the trade-off between quality and encoding speed by adjusting this parameter.

Similar to Etc2Comp's approach, we merge the local buffers in each thread that contain problematic pixel blocks into a single buffer. The handling of this buffer varies depending on the three quality modes. In the fast and normal modes, we sort the blocks in the merged buffer in descending order based on their error values. By arranging the blocks in this manner, with higher error values towards the front, we can extract only the top 10% or 40% of blocks for the fast and normal modes, respectively. We have determined the percentages according to our experiments will be introduced in Section 4.2. Conversely, in the best mode, we transmit all of the problematic blocks listed in the buffer to the GPU, eliminating the need for error-based sorting.

For further use in the GPU encoder, we convert the buffer data into an image. We define the width and height of the image to be the same, using $4 \times ceil(\sqrt{N})$, where N is the total number of problematic pixel blocks, as each block's size is 4×4. Subsequently, we apply zero-padding to the image to maintain the rectangular shape (Figure 6).

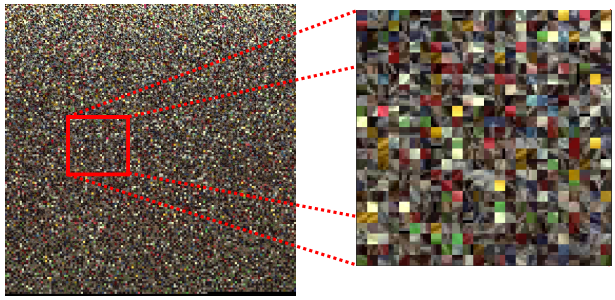

Figure 6: *The left image consists of problematic pixel blocks on kodim05 in the test set. As shown in the right image, which displays a zoomed-in view of the red box, the left image was created by accumulating 4×4 pixel blocks with high error values during CPU encoding. A rectangle image like this example is sent to the GPU.*

3.3. Design of the GPU Encoder

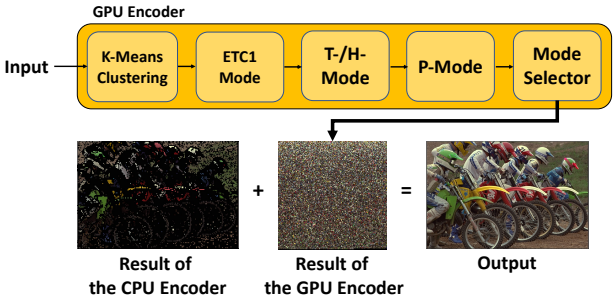

Figure 7: *A flow chart for the GPU Encoder consisting of five steps. When this GPU Encoder finishes all the encoding process, we combine the encoding results from the CPU and GPU to generate the final encoding output.*

The GPU encoding component of our encoder is based on the Betsy GPU compressor [Gol22]. Figure 7 illustrates the entire GPU encoding process. Similar to traditional CPU-based ETC2 compressors, Betsy employs multiple modes to encode each block and

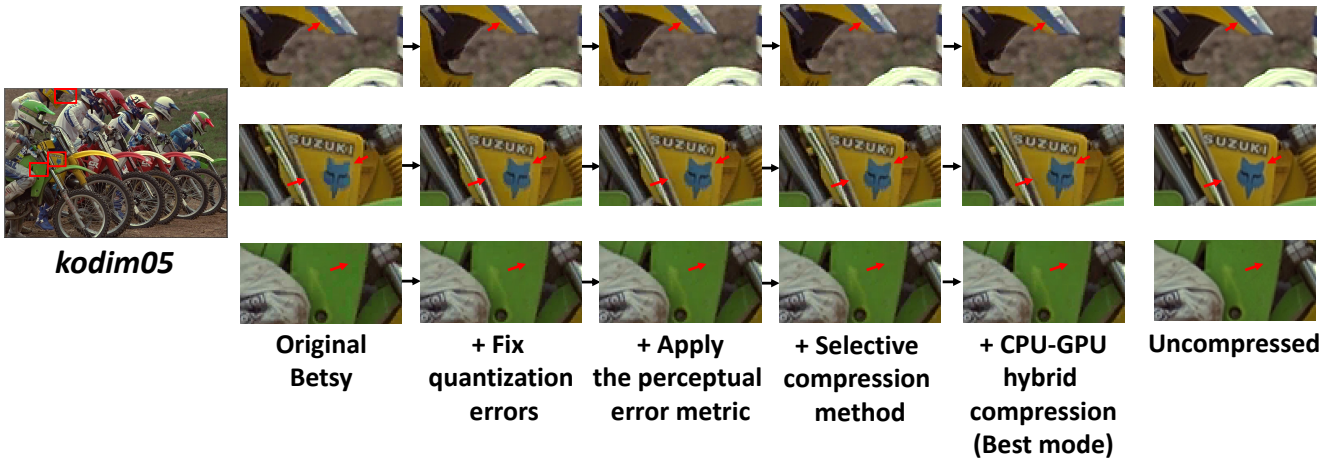

Figure 8: *How to solve issues of BetsyGPU observed in kodim05. Please focus on the areas indicated by the red arrows. Our first two approaches, fixing quantization errors in Betsy and applying the perceptual error metric, effectively eliminate compression artifacts, including block artifacts, color bleeding, and blurring. Our last two approaches, selective compression and CPU-GPU hybrid compression schemes, have a minimal impact on compression quality while significantly reducing encoding time. To better discern the quality difference, please zoom in on the image.*

selects the mode that yields the smallest error. It utilizes massive parallelism on the GPU in each mode, enabling it to achieve higher speeds than other CPU-based high-quality encoders. Betsy provides a quality parameter ranging from 0 to 2, and the number of iterations and the range of search spaces are determined based on the value of this parameter. Among the three available quality parameter values, we have chosen a value of 1 for our pipeline due to its optimal encoding speed-quality trade-off, as demonstrated in Table 1 (Section 4.4).

While Betsy aims to achieve high-quality and high-encoding speed by leveraging GPU parallelism in a brute-force manner, we encountered several issues regarding compression quality and efficiency. Figure 8 illustrates how we addressed these problems in the original Betsy GPU compressor.

We first addressed the issue in Betsy's code that generated quantization errors. In the T-/H-mode compression shader, there was a bug when converting an RGB444 quantized color to RGB888. The maximum value of an 8-bit color should be 255, but the calculated values in the shader code sometimes exceeded this limit and reached 285. This bug resulted in block artifacts because excessively bright base colors, far from the original colors in the block, were often generated after clipping the exceeded values to 255. After fixing this bug, we observed a reduction in block artifacts.

However, some block artifacts remained, as shown in the second column of Figure 8. Upon analyzing these artifacts, we discovered that most of them were caused by the equal weights assigned to each RGB channel in the luma calculation. While this weighting scheme may be suitable for increasing PSNR values, as the PSNR of an RGB image is calculated from the MSE value of each channel with the same weight, it fails to account for the fact that the human eye is more sensitive to green. To address this, we adjusted the weights for luma calculations using the perceptual error metric

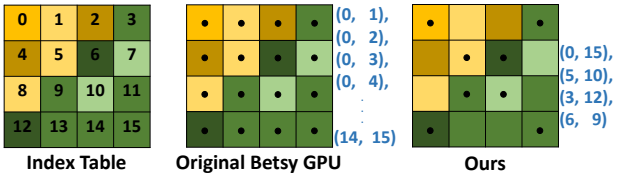

Figure 9: *The difference in K-means clustering between the original Betsy encoder and our encoder during the T-/H-mode process. The number of index pairs for two initial points has been reduced from 120 to 4.*

introduced in *i*PACKMAN [SAM05]. The weight values for each channel are the same as those given in Equation 1. Similar to Ström and Akenine-Möller's experiments [SAM05], we observed significantly improved block artifacts compared to the original Betsy GPU Encoder after modifying the weights (shown in the third column of Figure 8).

While the quality has been improved to an acceptable level, the low encoding speed of Betsy remains a problem. One of the main reasons for this low encoding speed is that its brute-force searching is excessive. In the original Betsy implementation, two initial points are selected from 16 different pixels in a block in any order for K-means clustering (${}_{16}C_2 = 120$). Subsequently, K-Means clustering is performed with all pairs in parallel, and the best base-color pair is chosen based on the results.

Our solution to increase encoding speed involves reducing the number of initial pairs of points, inspired by the selective compression method in THUMB [PS05]. We designate four special indices located in diagonal regions, as shown in Figure 9. Each point in our reduced set can still represent different partitions for two reasons. Firstly, the T- or H-mode handles diagonally divided clusters better

than the ETC1 mode, and secondly, pixels within each separate partition exhibit spatial coherence. Therefore, our approach prevents redundant clustering by avoiding the selection of adjacent pixels with similar data as the initial cluster points. By reducing the number of initial paired points from 120 to 4, our approach significantly improves encoding speed. However, there are no visual differences observed before and after our selective compression (shown in the fourth column of Figure 8).

The approaches described above are specifically related to the GPU shader computations, and our hybrid CPU-GPU approach can further enhance the synergy. The CPU encoding part efficiently handles the non-problematic blocks, reducing the input GPU data. Consequently, by leveraging both the CPU and GPU, we can significantly expedite the encoding process without compromising quality. Please compare the fifth and sixth columns of Figure 8.

4. Experiments & Results

4.1. Test Setup

For our experiment, we used the dataset utilized in QuickETC2 [Nah20a] for comparison with existing encoders. This texture dataset consists of a total of 64 textures with a variety of sizes (ranging from 256×256 to 8192×8192), channels (RGB and RGBA), and types (such as photos, game textures, GIS data, synthesized textures, and captured images). When compressing an image with an alpha channel, our encoder compresses the alpha channel by the CPU encoder only.

The desktop used for our experiments was equipped with an Intel Core i5-12400 CPU, 32GB of RAM, NVIDIA GeForce RTX 3060, a 1TB SSD, and running Windows 11. To measure the encoding quality and speed, we wrote test scripts using Python version 3.10 and OpenCV version 4.6.0.

For encoding speed comparison, we initially measured the encoding time for each texture and then converted the results into Mpixels/s to account for the varying sizes of the textures. This metric indicates the number of pixels that can be processed within a given time frame.

For quality comparison, we utilized the ꟻLIP metric [ANAM*20]. The reason for choosing this metric is that other commonly-used evaluation metrics in the image-processing community, such as PSNR and SSIM [WBSS04], tend to underestimate block artifacts that are noticeable in local areas or overestimate less visible artifacts. ETCPACK's slow mode, which employs exhaustive searches, actually delivers the highest quality in the ETC2 format. Nevertheless, PSNR or SSIM values of Etc2Comp and Betsy are similar or higher than those of the reference ETCPACK encoder with the slow mode (Appendix includes the detailed results). This observation aligns with the findings reported by Nilsson and Akenine-Möller [NAM20]. In contrast, the ꟻLIP metric more accurately measures the compression quality in this context as shown in Figure 12. Note that we used ꟻLIP version 1.2, mean values, and 67 pixels per degree (default) for our experiments.

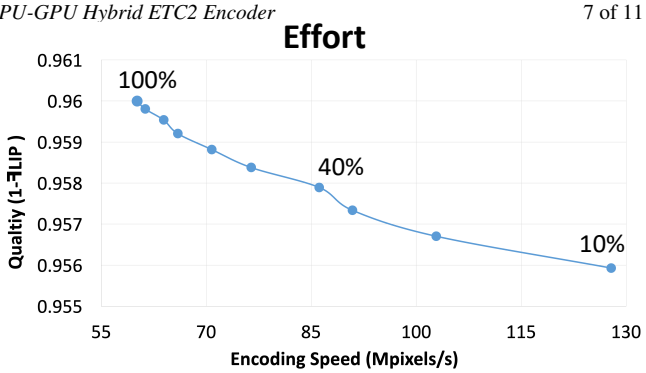

Figure 10: *The presented results showcase the experimental findings regarding the extent of utilization of the problematic pixel-block buffers. The visual representation begins from the left, illustrating a scenario of 100% usage, and gradually progresses towards the right, demonstrating a successive 10% reduction in usage at each subsequent interval.*

4.2. Effort Parameter Assignment

We conducted experiments to determine the values of the effort parameters, which represent the utilization ratios of the problematic pixel-block buffers, for each quality mode. As a result, we obtained the average of ꟻLIP values and encoding speed for each effort value, as illustrated in Figure 10. We observed that a value of 40% yielded results near the middle in terms of both encoding speed and quality. Therefore, we designated 10% as the fast mode for high encoding speed, 40% as the normal mode, and 100% as the best mode to achieve the highest quality.

4.3. Ablation Study

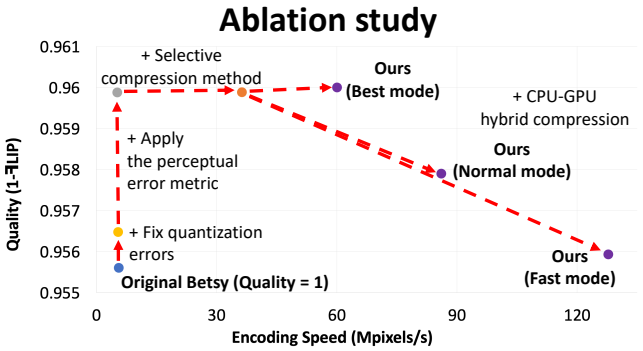

Figure 11: *Starting with the original Betsy GPU compressor, we present the evaluations of quality and encoding speed for each of our successive improvement steps. After conducting a comprehensive analysis, the figures depict a gradual and consistent enhancement in both quality and encoding speed throughout the entire process.*

We evaluated the encoding quality and speed of the steps outlined in Section 3.3, as depicted in Figure 11. Initially, by addressing the quantization error and implementing the perceptual error metric, we observed an improvement in quality and a 10% reduction in the ꟻLIP value. When applying our selective compression

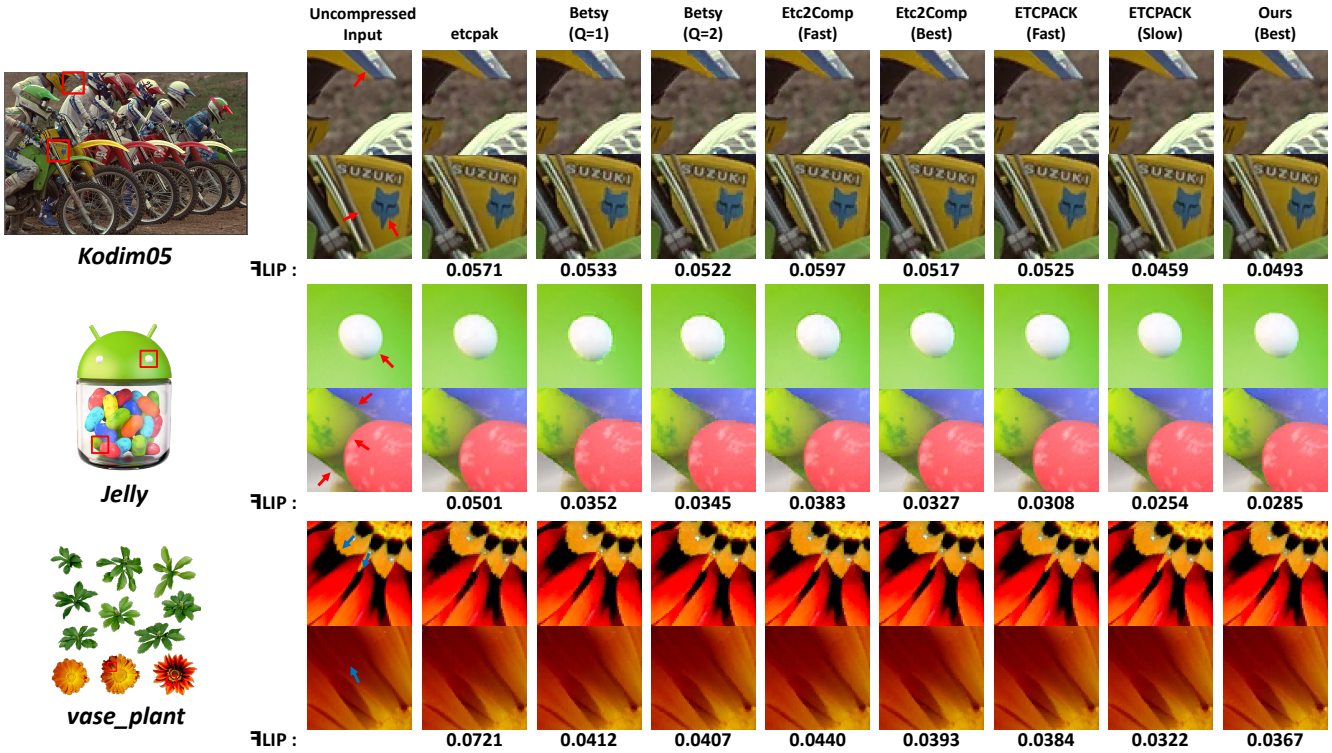

Figure 12: *Zoomed-in examples of the results from various encoders. Please focus on the areas indicated by the red and blue arrows. As depicted in the figure, our encoder (best) delivers comparable quality to ETCPACK (slow), while the other encoders (or modes) exhibit some compression artifacts, such as block artifacts, blurring, ringing, shape distortion, or color distortion.*

method, the increase in the ꟻLIP value was only around 0.00006 (0.015%), indicating minimal impact on quality. However, the encoding speed saw a significant increase by a factor of 6.5×. Finally, by merging the CPU and GPU encoders, the hybrid encoder in the best mode achieved 1.6× higher encoding speed with similar quality compared to the GPU-only encoder. In comparison to the original Betsy, our encoder in the fast mode is 22× faster while maintaining similar ꟻLIP values.

4.4. Results in Terms of Quality & Encoding speed

In this subsection, we compare the encoding speed and quality of several different encoders: etcpak 1.0, Betsy, Etc2Comp, and ETCPACK 2.74. As mentioned in the QuickETC2 paper [Nah20a], the open-source version of Etc2Comp ignores color values in transparent regions during compression, resulting in increased errors in RGBA textures. However, this issue has been resolved in etccompress.exe included in Unity version 2020.3.47f1 LTS. Hence, we utilized this executable from Unity for quality measurement. Since this executable does not have a time measurement feature, we measured the encoding time using the original open-source version of Etc2Comp. We discovered that the fast and best modes in etccompress.exe correspond to the original Etc2Comp's effort parameters of 0 and 70, respectively. The detailed configuration values for each encoder are described below.

- Betsy with 0, 1, and 2 as the quality parameters
- Etc2Comp with the fast (effort = 0) and best (effort = 70) modes
- ETCPACK with the fast and slow modes
- H-ETC2 with the fast, normal, and best modes

Table 1: *Average ꟻLIP values and Mpixels/s for each encoder across the dataset of 64 textures. Lower ꟻLIP values and higher Mpixels/s indicate higher compression quality and faster encoding time, respectively.*

Compressor	Mode	ꟻLIP	Mpixels/s
etcpak		0.0506	1350.82
Betsy	Q=0	0.0474	6.20
	Q=1	0.0444	5.63
	Q=2	0.0438	2.22
Etc2Comp	Fast	0.0480	3.97
	Best	0.0419	0.15
ETCPACK	Fast	0.0419	0.85
	Slow	0.0375	0.0041
H-ETC2 (ours)	Fast	0.0440	127.87
	Normal	0.0421	86.15
	Best	0.0400	60.14

Table 1 provides a summary of our experimental results. The

quantitative analysis based on our encoder in the best mode is described as follows. Firstly, when comparing the average Mpixels/s with Betsy for each quality parameter, we observe that our encoder outperforms Betsy with a quality parameter of 0 by 9.7×. However, the average ꟻLIP value represented by our encoder is approximately 8.6% lower than that achieved by Betsy with a quality parameter of 2. This indicates that our encoder is capable of encoding textures with better quality and encoding speed compared to Betsy.

Comparison to Etc2Comp is similar to the above comparison to Betsy. Our encoder in the best mode is 15× faster than Etc2Comp in the fast mode, but ours has a 4.5% lower average ꟻLIP value than Etc2Comp in the best mode.

Compared to ETCPACK, a reference encoder, our encoder achieves significantly faster encoding times, often tens or thousands of times faster. In terms of quality, our encoder surpasses ETCPACK in the fast mode, with an average ꟻLIP difference of about 0.0019 (4.5%). However, our encoder in the best mode is behind ETCPACK in the slow mode, with an average ꟻLIP difference of 0.0025 (6.3%). Nevertheless, upon a thorough examination of the entire encoding results in the test set, no noticeable visual differences have been observed between them. This demonstrates that our encoder can achieve similar quality encoding results to the reference encoder in significantly less time. A encoding speed-quality graph for each encoder in each mode is depicted in Figure 1.

Although our hybrid encoder is slower than etcpak due to the additional GPU encoding process, it effectively reduces most compression artifacts observed in etcpak. Consequently, our encoder in the best mode achieves a 21% lower average ꟻLIP value. Additionally, the speed gap between the fast and best modes of H-ETC2 is narrower than that of Etc2comp and ETCPACK. In contrast to the approach of expanding the search space in their best/slow mode using exhaustive search, we focus on adjusting only the amount of input texels to recompress. This results in relatively smaller speed differences between the fast and best modes (approximately 2×).

We qualitatively analyze the results shown in Figure 12 as follows. In the helmet and the fox picture in *kodim05*, we can observe that Betsy, in all modes, produces block artifacts due to quantization errors. While Etc2Comp and etcpak improve upon this, some block artifacts remain. ETCPACK in the fast mode exhibits better results than Etc2Comp but slightly rougher results compared to that in the slow mode. The results from our encoder in the best mode provide comparable quality to ETCPACK in the slow mode.

Next, let's analyze the results in *jelly*. Betsy exhibits ringing and block artifacts in the eye, and these artifacts are only completely removed in ETCPACK (slow) and our encoder (best), but not in the others. The block artifacts at the boundaries between the jellies are most severe in etcpak due to similar luma values between the pink and green jellies. ETCPACK (slow) and our encoder (best) also demonstrate the highest quality in this particular test.

The final image, *vase_plant*, showcases the mixed results of our encoder. As depicted in the first row, most of the block artifacts present in etcpak are effectively handled by the other encoders, including ours. However, in the second row, our encoder still exhibits block patterns in the upper-left side of the orange floral leaf. This occurred because the region has low contrast, leading the CPU encoder to compress it using the Planar mode. The error values of these blocks were low, preventing them from being accumulated into problematic pixel-block buffers. While this represents the worst-case scenario in the test set, these block patterns are not visible unless a user zooms in on the image. Therefore, we believe that this case does not cause any visual inconvenience to users after mapping the texture to an object.

5. Discussion

5.1. Limitations

By implementing selective compression in the GPU encoder, we successfully improved encoding speed. However, it is important to note that in rare and extreme cases, such as when deviating from the index pattern we have established, our encoder may not accurately calculate proper base colors. Figure 13 showcases several examples of such extreme cases. Nevertheless, during our tests, we did not encounter any instances of quality degradation in these scenarios. In fact, if the hue difference within a block is not significant and the two partitions exhibit distinct levels of saturation or brightness, the ETC1 mode can effectively handle the compression, even if our encoder's T- or H-mode struggles to compress the block optimally.

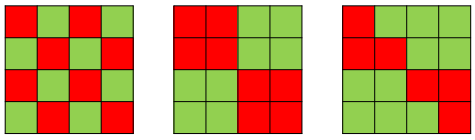

Figure 13: *Failure cases that our selective compression mishandles. The four index pairs we use (Figure 9) fail to properly divide the partitions in the above cases.*

Another limitation is that GPU encoding accounts for a significant portion of the total encoding time in our hybrid implementation. While the CPU encoding component, derived from etcpak, exhibits fast encoding speed, the GPU encoding component remains comparatively slower, even after our optimizations. This is primarily due to the fact that the GPU encoder explores a much broader search space and tests all possible compression modes. As a result, our encoder currently performs ETC2 encoding sequentially in a pipeline format. However, if there were a method to offload more processing burden onto the CPU encoding part in exchange for higher-quality compression [Nah23], it could potentially balance the encoding time with the GPU encoder.

5.2. Conclusions and Future Work

In this paper we have introduced a hybrid ETC2 encoding pipeline that combines CPU and GPU processing. Our pipeline uses the GPU encoding component to enhance compression quality following CPU encoding. As a result, our encoder achieves a better balance between compression quality and encoding speed, surpassing other encoders in the experiment. Specifically, our encoder achieves comparable quality to ETCPACK in the slow mode and outperforms other encoders in terms of encoding speed, except for etcpak.

Currently, the default ETC2 configuration in Unity utilizes etcpak, ETCPACK (fast), and Etc2Comp (best) for the fast, normal, and best settings, respectively [Uni22]. However, given the significant speed and quality enhancements offered by our encoder, we believe it can be considered as an alternative to the encoders used in the last two settings. By incorporating our encoder, the ETC2 encoding process will be considerably faster, while simultaneously delivering superior quality.

In terms of future work, we aim to explore the applicability of our CPU-GPU hybrid approach to other texture formats, including ASTC and BC7. Additionally, we are keen on enhancing performance by refining the balance between CPU and GPU processing times, as mentioned earlier.

Acknowledgments

We appreciate the reviewers for their error corrections and constructive suggestions, and we also thank the authors of etcpak and Betsy who have opened their source code to the public. We plan to release the H-ETC2 source code introduced in this paper soon at the following link: https://github.com/gusrlLee/HETC2. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1G1A1093404). Image courtesy of Kodak, Simon Fenney, Crytek, UNC GAMMA Lab, Spiral Graphics, Vokselia Spawn, Cesium, Google, fhernand of Sketchfab.

References

[ANAM*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖMAND MARK D. FAIRCHILD K.: ꟻLIP: A difference evaluator for alternating images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques (HPG 2020) 3*, 2 (2020). doi:10.1145/3406183. 7

[AND14] ANDRE J.-P.: etc2_encoder, 2014. URL: https://github.com/titilambert/packaging-efl/blob/master/src/static_libs/rg_etc/etc2_encoder.c. 3

[BAAM17] BARRINGER R., ANDERSSON M., AKENINE-MÖLLER T.: Ray accelerator: Efficient and flexible ray tracing on a heterogeneous architecture. *Computer Graphics Forum 36*, 8 (2017), 166–177. doi:10.1111/cgf.13071. 3

[Eri18] ERICSSON: ETCPACK, 2018. URL: https://github.com/Ericsson/ETCPACK. 1, 2

[GB17] GOOGLE INC., BLUE SHIFT INC.: Etc2Comp - texture to ETC2 compressor, 2017. URL: https://github.com/google/etc2comp. 1, 2, 5

[Gol22] GOLDBERG M. N.: Betsy GPU compressor, 2022. URL: https://github.com/darksylinc/betsy. 1, 2, 3, 5

[Int22] INTEL: oneAPI specification release 1.2-rev-1, 2022. URL: https://spec.oneapi.io/versions/latest/oneAPI-spec.pdf. 3

[KHH*09] KIM D., HEO J.-P., HUH J., KIM J., YOON S.-E.: HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs. *Computer Graphics Forum 28*, 7 (2009), 1791–1800. doi:10.1111/j.1467-8659.2009.01556.x. 3

[Mic18] MICROSOFT: Texture block compression in Direct3D 11, 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/texture-block-compression-in-direct3d-11. 1

[MV15] MITTAL S., VETTER J. S.: A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Survey 47*, 4 (jul 2015). doi:10.1145/2788396. 3

[Nah20a] NAH J.-H.: QuickETC2: Fast ETC2 texture compression using luma differences. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2020) 39*, 6 (2020). doi:10.1145/3414685.3417787. 2, 3, 4, 7, 8, 11

[Nah20b] NAH J.-H.: QuickETC2: How to finish ETC2 compression within 1 ms. In *ACM SIGGRAPH 2020 Talks* (2020). doi:10.1145/3388767.3407373. 3

[Nah23] NAH J.-H.: QuickETC2-HQ: Improved ETC2 encoding techniques for real-time, high-quality texture compression. *Computers & Graphics* (2023). doi:10.1016/j.cag.2023.08.032. 9

[NAM20] NILSSON J., AKENINE-MÖLLER T.: Understanding SSIM. *ArXiv e-prints* (2020). doi:10.48550/arXiv.2006.13846. 7

[NKL*10] NAH J.-H., KANG Y.-S., LEE K.-J., LEE S.-J., HAN T.-D., YANG S.-B.: MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices. In *ACM SIGGRAPH ASIA 2010 Sketches* (2010), pp. 50:1–50:2. doi:10.1145/1899950.1900000. 3

[NKP*15] NAH J.-H., KIM J.-W., PARK J., LEE W.-J., PARK J.-S., JUNG S.-Y., MANOCHA D., HAN T.-D.: HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics 21*, 3 (2015), 389–401. doi:10.1109/TVCG.2014.2371855. 3

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive scalable texture compression. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on High-Performance Graphics* (2012), pp. 105–114. doi:10.2312/EGGH/HPG12/105-114. 1

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., ET AL.: OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG) 29*, 4 (2010), 1–13. doi:10.1145/1778765.1778803. 2

[PBPP11] PAJOT A., BARTHE L., PAULIN M., POULIN P.: Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering. *Computer Graphics Forum 30*, 2 (2011), 315–324. doi:10.1111/j.1467-8659.2011.01863.x. 3

[PP14] PALTASHEV T., PERMINOV I.: Texture compression techniques. *Scientific Visualization 6*, 1 (2014), 106–146. URL: http://sv-journal.org/2014-1/06/en/index.php?lang=en. 1

[PS05] PETTERSSON M., STRÖM J.: Texture compression: THUMB: Two hues using modified brightness. In *SIGRAD 2005 The Annual SIGRAD Conference Special Theme-Mobile Graphics* (2005), no. 016, Linköping University Electronic Press, pp. 7–12. URL: https://ep.liu.se/ecp/016/002/ecp01602.pdf. 2, 6

[Ric12] RICH GELDREICH: rg_etc1: Fast, high quality ETC1 (Ericsson Texture Compression) block packer/unpacker, 2012. URL: https://code.google.com/archive/p/rg-etc1/. 3

[SA13] SEGAL M., AKELEY K.: The OpenGL© Graphics System: A Specification (Version 4.3 (Core Profile) - February 14, 2013), 2013. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec43.core.pdf. 2

[SAM04] STRÖM J., AKENINE-MÖLLER T.: PACKMAN: Texture compression for mobile phones. In *ACM SIGGRAPH 2004 Sketches* (2004), p. 66. doi:10.1145/1186223.1186306. 2

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: High-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2005), pp. 63–70. doi:10.1145/1071866.1071877. 1, 2, 6

[Smi18] SMITH S.: Adaptive scalable texture compression. In *GPU Pro 360 Guide to Mobile Devices*. AK Peters/CRC Press, 2018, pp. 153–166. 4

Table 2: *Experimental results for each category in terms of quality (PSNR, SSIM, and ꟻLIP) and encoding speed (Mpixels/s).*

	Betsy GPU			Etc2Comp		ETCPACK		etcpak 1.0	Ours		
	Q=0	Q=1	Q=2	Fast	Best	Fast	Slow	-	Fast	Normal	Best
PSNR											
Photo	36.36	37.20	37.31	36.40	37.49	36.56	37.42	35.70	36.05	36.38	36.62
Game	37.43	38.15	38.24	37.54	38.56	37.71	38.46	36.22	36.96	37.29	37.55
GIS Map	37.30	38.11	38.25	37.68	39.04	37.83	38.96	34.64	37.09	37.50	37.81
Synth	38.81	39.41	39.51	38.80	39.72	38.87	39.58	37.04	37.65	37.95	38.10
Captured	46.54	46.82	46.91	46.60	48.53	47.20	48.63	41.62	46.37	46.49	46.67
SSIM											
Photo	0.9597	0.9646	0.9649	0.9594	0.9663	0.9618	0.9654	0.9525	0.9533	0.9549	0.9583
Game	0.9641	0.9686	0.9689	0.9637	0.9697	0.9649	0.9684	0.9487	0.9576	0.9598	0.9629
GIS map	0.9724	0.9763	0.9767	0.9743	0.9791	0.9752	0.9783	0.9555	0.9672	0.9690	0.9730
Synth	0.9727	0.9794	0.9798	0.9745	0.9817	0.9709	0.9745	0.9031	0.9702	0.9724	0.9734
Captured	0.9909	0.9913	0.9914	0.9903	0.9914	0.9915	0.9915	0.9811	0.9897	0.9899	0.9903
ꟻLIP											
Photo	0.0506	0.0475	0.0469	0.0523	0.0450	0.0450	0.0402	0.0479	0.0467	0.0447	0.0424
Game	0.0502	0.0466	0.0460	0.0506	0.0447	0.0445	0.0401	0.0533	0.0476	0.0452	0.0426
GIS Map	0.0643	0.0615	0.0610	0.0621	0.0553	0.0561	0.0499	0.0807	0.0584	0.0564	0.0543
Synth	0.0384	0.0349	0.0345	0.0398	0.0339	0.0327	0.0299	0.0431	0.0332	0.0316	0.0297
Captured	0.0184	0.0183	0.0183	0.0176	0.0156	0.0165	0.0135	0.0360	0.0165	0.0165	0.0163
Mpixels/s											
Photo	6.04	5.54	2.09	3.58	0.12	0.84	0.003	708.85	55.70	43.72	34.18
Game	5.94	5.38	2.23	4.14	0.17	0.80	0.004	1279.41	99.32	78.72	62.32
GIS Map	5.67	4.80	2.05	3.63	0.11	0.85	0.004	453.17	33.79	24.86	20.58
Synth	3.40	3.18	1.59	4.66	0.23	0.89	0.003	663.90	32.66	28.29	24.90
Captured	8.82	8.08	2.92	4.76	0.13	1.08	0.004	4618.02	575.59	316.98	177.40

[SP07] STRÖM J., PETTERSSON M.: ETC 2: texture compression using invalid combinations. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2007), pp. 49–54. doi:10.2312/EGGH/EGGH07/049-054. 1, 2

[Tau22] TAUDUL B.: etcpak:the fastest ETC compressor on the planet, 2022. URL: https://github.com/wolfpld/etcpak. 2, 3, 4

[Uni22] UNITY TECHNOLOGIES: Unity user manual (2022.2), 2022. URL: https://docs.unity3d.com/2022.2/Documentation/Manual/class-EditorManager.html. 9

[vMVJ22] ŽÁDNÍK J., MÄKITALO M., VANNE J., JÄÄSKELÄINEN P.: Image and video coding techniques for ultra-low latency. *ACM Computing Surveys 54*, 11s (sep 2022). doi:10.1145/3512342. 2

[WBSS04] WANG Z., BOVIK A. C., SHEIKH H. R., SIMONCELLI E. P.: Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612. doi:10.1109/TIP.2003.819861. 7

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 1–8. doi:10.1145/2601097.2601199. 2

Appendix

Figure 14 illustrates the texture set employed in our experiments, which can be categorized into five types: Photos (1st to 25th), game textures (26th to 51st), GIS map data (52nd to 55th), synthesized images (56th to 57th), and captured images from a camera for 3D reconstruction (58th to 64th). In Table 2, the "Captured" dataset exhibits a resolution of 8192 × 8192, resulting in higher PSNR values than other input textures. Consequently, the proportion of problematic pixel blocks to be reprocessed on the GPU is significantly lower than in the other lower-resolution textures, leading to a higher encoding speedup factor. Detailed experimental results for each category are provided in Table 2.

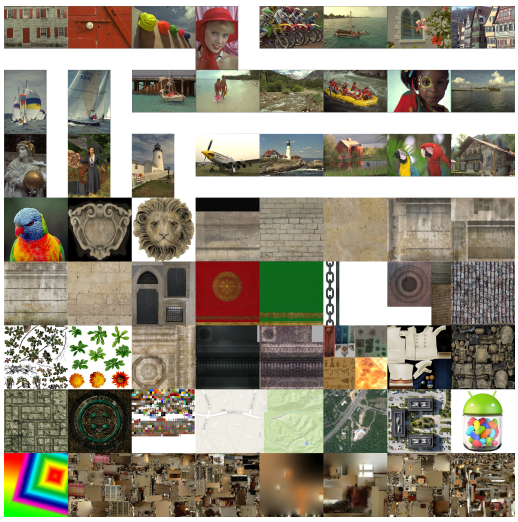

Figure 14: *64 textures in the QuickETC2 test set [Nah20a].*