Article Type: Theme Article

An Analytical Survey of ASTC Encoding Techniques

In-hyuk Jeong, *Sangmyung University, Seoul, 03016, South Korea*

Jae-Ho Nah, *Sangmyung University, Seoul, 03016, South Korea*

Abstract—In 3D graphics, texture mapping is an essential technique. Over the past three decades, various texture codecs have been developed to efficiently compress data, balancing compression quality, ratio, speed, and flexibility. Among them, ASTC (Adaptive Scalable Texture Compression) is widely adopted on mobile devices. Its high flexibility brings structural complexity, requiring careful consideration during encoding. This study analyzes the main stages, characteristics, and design methodology of ASTC encoding. We discuss key techniques, including partitioning algorithms, color encoding, and block size trade-offs, and how to balance encoding quality and speed. Furthermore, we experimentally evaluate quality and encoding speed by varying the encoder, block size, and modes. Finally, we provide a structured overview of representative studies on ASTC. This work aims to enhance understanding of ASTC among app developers and graphics researchers to facilitate its more effective use.

In real-time 3D graphics, texture mapping is an indispensable component for rendering. There are many different types of texture maps, including diffuse, specular, normal, and opacity maps, and even non-graphic data is stored in texture maps. To effectively manage texture maps in a compressed format, several texture codecs, such as BC [1], ETC [2], [3], PVRTC [4], and ASTC [5], have been developed and standardized. As a result, most apps across various platforms that support real-time 3D graphics now include compressed texture maps, including those for PCs, mobile/embedded devices, and consoles.

As 3D graphics has advanced, texture codecs have evolved to provide better compression efficiency, support multiple formats, and offer flexible compression ratios. ASTC (Adaptive Scalable Texture Compression) [5], jointly developed by Arm and AMD, aims to support all the goals mentioned above in a single codec. Currently, most modern mobile and embedded GPUs include hardware ASTC decoders across major mobile platforms (e.g., Android and iOS). As a result, ASTC has become a widely adopted texture compression standard.

XXXX-XXX © 2026 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

Like other texture codecs, ASTC exhibits a significant difference in computational complexity between encoding and decoding. Decoding is performed on demand by dedicated on-chip GPU hardware during rendering, and the format inherently supports random access. In contrast, encoding is done offline, taking much longer as it applies various compression methods. Furthermore, unlike other block-based texture codecs, ASTC supports various block sizes to adjust compression ratios, which introduces more factors to consider during encoding.

While newer neural texture codecs [36], [37] have emerged to achieve higher compression efficiency on PBR (physically based rendering) textures, they shift a critical computational burden to the runtime stage by requiring software-based MLP (multilayer perceptron) inference for decoding. This lack of dedicated hardware acceleration demands heavy runtime overhead, making neural approaches impractical for real-time use on mobile devices. Consequently, ASTC remains the most viable and widely adopted standard for mobile deployment, making a comprehensive understanding of its encoding behaviors crucial for optimal performance.

Driven by this need, this overview paper summarizes recent research on ASTC encoding, with a focus on *astcenc* [6], Arm's reference encoder continuously updated for over a decade. To the best of our

knowledge, no existing study provides a comprehensive analysis of the entire *astcenc* encoding pipeline. Therefore, this paper serves as a valuable resource for researchers in texture compression.

The structure of this paper is as follows: An overview of the ASTC format is provided in the next section. A detailed analysis of the *astcenc* source code is then offered. Experimental results benchmarking encoder speed and quality across various configuration options are presented subsequently. Following this, a literature review summarizes major studies related to ASTC. Finally, concluding remarks and potential future research directions are explored.

ASTC OVERVIEW

ASTC introduces several novel features, including BISE (Bounded Integer Sequence Encoding) [7], a partitioning table generated by a hash function, variable block sizes, and other distinctive encoding approaches. Since these features are well documented in the original ASTC paper [5], we will provide a brief overview of each in this section.

BISE

In ASTC, color data is compressed using BISE to store it more efficiently. BISE efficiently packs bounded integer values when the number of possible values is not a power of two. When the color and weight values stored in texels—which are simply pixels within a texture—are floating-point numbers, there aren't enough bits to directly store the actual values in an ASTC block. Therefore, to reduce the required storage space, the values must be quantized.

Let us see an example described in *astcenc* [6]. When texel values ranging from 0.0 to 1.0 are quantized into five values, 0.0, 0.25, 0.5, 0.75, and 1.0, they can be represented in 3 bits. Because 2.32 ($\approx \log_2 5$) bits per value are required to represent them, the binary representation wastes 22% of the available bits. To solve this problem, BISE combines bits, trits, and quints (base-3 and base-5 digits, respectively) (Table 1). As shown in Figure 1, a combination with the smallest storage is selected for BISE, resulting in a bit-wastage reduction.

Variable Block Sizes

One of the key features of ASTC, compared to other formats, is its support for variable block sizes. Each block occupies 128 bits, but the number of texels per block can vary depending on 14 different presets in 2D (ranging from 4x4 to 12x12, as shown in Table 2) and 10 presets in 3D (ranging from 3x3x3 to 6x6x6). As the block size increases, the compression quality tends to decrease, but the compression ratio improves. This flexibility allows developers to strike a balance between compression ratio and image quality.

TABLE 1: BISE encoding ranges [8]

Value range	MSB encoding	LSB encoding	Value	Block	Packed block size
$0..2^n-1$	-	n bit value m ($n \leq 8$)	m	1	n
$0..(3{\cdot}2^n)-1$	Base-3 "trit" value t	n bit value m ($n \leq 6$)	$t{\cdot}2^n+m$	5	$8+5n$
$0..(5{\cdot}2^n)-1$	Base-5 "quint" value q	n bit value m ($n \leq 5$)	$q{\cdot}2^n+m$	3	$7+3n$

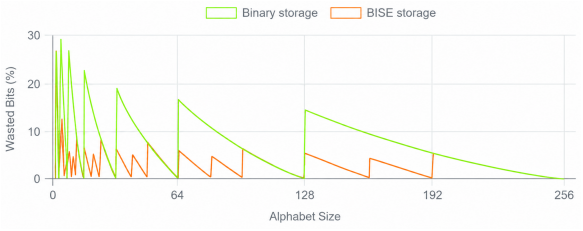

FIGURE 1: Storage efficiency of BISE and binary encoding [6], [9]

TABLE 2: ASTC block sizes and compression ratios [5], [8] (e.g., 3.00 represents 3× reduction)

ASTC block size	Bits per pixel (bpp)	Compression ratio (RGB / RGBA)
4 × 4	8.00	3.00 / 4.00
5 × 4	6.40	3.75 / 5.00
5 × 5	5.12	4.69 / 6.25
6 × 5	4.27	5.63 / 7.50
6 × 6	3.56	6.75 / 9.00
8 × 5	3.20	7.50 / 10.00
8 × 6	2.67	9.00 / 12.00
10 × 5	2.56	9.38 / 12.50
10 × 6	2.13	11.25 / 15.00
8 × 8	2.00	12.00 / 16.00
10 × 8	1.60	15.00 / 20.00
10 × 10	1.28	18.75 / 25.00
12 × 10	1.07	22.50 / 30.00
12 × 12	0.89	27.00 / 36.00

Partitioning

A partition divides a compression block into separate texel regions, each evaluated with its own color gradient. Standard DirectX formats—such as BC6H for HDR and BC7 for high-quality LDR textures—operate strictly on a fixed 4 × 4 block size with a maximum of two and three partitions, yielding only 32 and 64 total patterns, respectively. This small pattern count allows decoders to easily store all partition layouts in

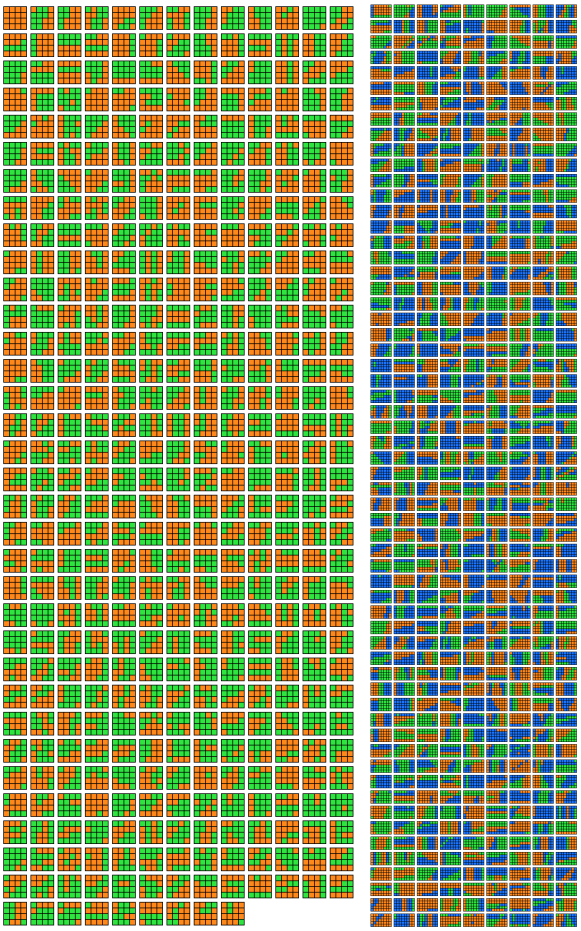

FIGURE 2: Example partitioning cases: 2-partition (4×4) and 3-partition (8×5) patterns with duplicates removed [10].

compact hardware lookup tables (LUTs). Conversely, ASTC supports up to four partitions across 24 different block sizes, creating a combinatorial space far too vast for static tables. Consequently, ASTC dispenses with lookup tables, relying instead on a procedural generator to compute partition patterns on the fly.

The number of partition patterns for a single block size supported by ASTC is 3,072 ($1{,}024 \times 3$). To represent these patterns, the number of partitions (ranging from one to four) is stored in 2 bits, while the partition ID is stored in 10 bits if the block is partitioned. To generate 1,024 patterns, a PRNG (pseudo-random number generator) with a 10-bit seed value is used. Since the hash functions in both the encoder and decoder are identical, the intended partition pattern can be immediately obtained through a separate calculation in the decoder. According to Nystad et al. [5], however, about half of the patterns generated by the PRNG are either duplicated, return constants, or are too fragmented to be useful. As a result, while this method offers flexibility in supporting various block sizes and partition counts as illustrated in Figure 2, some patterns can be underutilized during encoding and be wasted.

Encoding modes

ASTC encoding primarily utilizes endpoint interpolation. Endpoint interpolation reconstructs texel colors by blending two representative endpoint colors stored for a block. First, the color values of the two endpoints in a block are determined. These two colors can be stored in three ways: independently (direct), as a combination of a base color and its offset values (differential) [2], or as a combination of a base color and its scale factors (multiplicative). Once the endpoints are defined, colors along a straight line connecting the two colors in color space can be obtained through interpolation. Each texel in a block has a weight to select the interpolated color. For larger blocks that do not have enough space to store per-texel weights, a low-resolution weight grid is stored instead, and per-texel weights are calculated during the decoding process through interpolation.

ASTC supports both low and high dynamic ranges (LDR and HDR), as well as one to four color channels. There are sixteen encoding modes in the ASTC format (Table 3) to handle the different cases, with appropriate encoding modes used to determine endpoint colors. Additionally, ASTC allows for different interpolation weights for two separate color channels (e.g., RGB and A), a feature known as dual-plane weights.

TABLE 3: 16 encoding modes in ASTC [8]. CEM stands for color endpoint mode.

CEM	Dynamic range	Color space	Encoding method
0	LDR	L	direct
1	LDR	L	base+offset
2	HDR	L	large range
3	HDR	L	small range
4	LDR	LA	direct
5	LDR	LA	base+offset
6	LDR	RGB	base+scale
7	HDR	RGB	base+scale
8	LDR	RGB	direct
9	LDR	RGB	base+offset
10	LDR	RGB	base+scale plus two A
11	HDR	RGB	direct
12	LDR	RGBA	direct
13	LDR	RGBA	base+offset
14	LDR	RGBA	direct + LDR alpha
15	HDR	RGBA	direct + HDR alpha

ANALYSIS OF THE *ASTCENC* SOURCE CODE

In this section, we analyze the implementation of astcenc to show how this reference encoder optimizes the ASTC encoding pipeline.

Encoding parameters

The performance of *astcenc* can be finely tuned through the various parameters listed in Table 4, which determine the trade-off between compression quality and encoding speed. In this section, we analyze the key tuning parameters of *astcenc* and evaluate the impact of different mode settings on its performance.

Parameters related to partitions The Partition Count Limit is a key parameter that restricts the maximum number of partitions used within a block. ASTC allows each block to be divided into up to four partitions, enabling independent handling of different color regions. As shown in the table, the fastest mode limits the number of partitions to a maximum of two to minimize computation, whereas the exhaustive mode utilizes up to four partitions to achieve the highest quality.

2-, 3-, and 4-Partition Index Limit for each partition count determine the range of partitioning patterns to be explored. Although ASTC defines a total of 3,072 partition patterns per block, *astcenc* controls computational efficiency by limiting the number of patterns explored. For example, in the fastest mode, only ten indices are searched for 2-partition blocks, six for 3-partition blocks, and four for 4-partition blocks. Even in the highest-quality exhaustive mode, 512 patterns are considered per partition. Due to the existence of redundant patterns [10] and the similarity among patterns, exploring only half of the total patterns results in negligible quality loss.

Once the partition indices to be searched are determined, candidate partition patterns are selected based on the Hamming distances (the numbers of differently assigned texels) between the K-means result and the partition patterns. The number of candidate patterns per partition is then determined by 2-, 3-, and 4-Partitioning Candidate Limit. Through this two-step process, the computation is reduced by evaluating the error for only a subset of partition patterns rather than all possible patterns for the current block.

Parameters related to block modes and candidates Block Mode Limit restricts the percentage of block modes that are allowed to be evaluated. ASTC provides various modes, including decimation mode, weight quantization mode, weight bits, and dual plane options. In *astcenc*, combinations of these modes are constructed into block modes. These block modes are examined in an empirically determined order based on their frequency of use. Therefore, the block mode limit parameter enables control over the speed-quality trade-off. Specifically, the setting increases progressively from 43% in the fastest mode to 100% in the exhaustive mode, representing the range of block modes considered by the encoder.

Candidate Limit restricts the number of partition candidates evaluated when compressing each block. Evaluating more candidates increases the likelihood of achieving optimal compression quality, but at the cost of higher computational complexity.

Parameters related to quality control and early termination Refinement Limit sets the number of iterations for refining colors and weights to improve the quality of the initial compression result. In the fastest mode, the refinement process is minimized (two iterations) to prioritize speed, whereas in the exhaustive mode, four iterations are performed to maximize quality.

dB Limit is a value that allows early termination of encoding when the quality metric, measured in PSNR, reaches the specified threshold. Naturally, higher values result in continued encoding until higher quality is achieved. This value is set according to the following formula, which depends on two base values and the number of texels in the block:

$$\max\left(A\_Base - 35\log_{10} N_{\text{texel}},\ B\_Base - 19\log_{10} N_{\text{texel}}\right)$$

MSE Overshoot is an option related to both dB Limit and Refinement Limit. It acts as a safety margin that prevents refinement from stopping too early. After converting the dB limit to MSE (mean squared error), refinement stops if the current error is smaller than the calculated error threshold. The overshoot setting is used as a divisor when computing this threshold. Larger overshoot values result in tighter thresholds, which in turn require more computations and can achieve higher compression quality.

2- and 3-Partition Early Out Limit Factor are another early termination mechanism that determines whether evaluation should continue for a larger number of partitions. If the lowest error for N+1 partitions exceeds the product of this factor and the lowest error for N partitions, evaluation for the additional partition is skipped. Therefore, higher values of this factor increase the likelihood that 3- or 4-partition configurations will be examined.

2-Plane Early Out Limit Correlation provides a criterion for whether the dual-plane mode should be used. *astcenc* calculates the correlation between channels

TABLE 4: Preset parameters for 4 × 4 blocks from astcenc (v5.7), derived from the source code [6]

Attribute	fastest	fast	medium	thorough	verythorough	exhaustive
Quality	0.0	10.0	60.0	98.0	99.0	100.0
Partition Count Limit	2	3	4	4	4	4
2-Partition Index Limit	10	18	34	82	256	512
3-Partition Index Limit	6	12	28	60	128	512
4-Partition Index Limit	4	10	16	30	64	512
Block Mode Limit	43	55	77	94	98	100
Refinement Limit	2	3	3	4	4	4
Candidate Limit	2	3	3	4	6	8
2-Partition Candidate Limit	2	2	2	3	8	8
3-Partition Candidate Limit	2	2	2	2	6	8
4-Partition Candidate Limit	2	2	2	2	4	8
dB Limit A Base	85.2	85.2	95.0	105.0	200.0	200.0
dB Limit B Base	63.2	63.2	70.0	77.0	200.0	200.0
MSE Overshoot	3.5	3.5	2.5	10.0	10.0	10.0
2-Partition Early Out Factor	1.0	1.0	1.1	1.35	1.6	2.0
3-Partition Early Out Factor	1.0	1.0	1.05	1.15	1.4	2.0
2-Plane Early Out Correlation	0.85	0.90	0.95	0.97	0.98	0.99
Search Mode0 Enable	0.0	0.0	0.0	0.0	0.0	0.0

within a block. If the correlation is below the specified threshold, the encoder attempts the second plane; otherwise, it uses only a single plane to save time. This setting plays an important role in RGBA texture compression performance. If you are interested in the two-plane mode, please refer to the heading "Comparison of 1- and 2-plane modes."

Finally, Search Mode0 Enable determines whether only the Mode0 fast path for "always" block modes should be executed. A value of 1 restricts encoding to the essential decimation modes, allowing fast encoding, whereas a value of 0 enables encoding of additional decimation modes to improve quality. In Table 4, all values are shown as 0 for 4x4 blocks, but depending on block size and compression settings, this value can be set to 1.

Parameter interactions and performance impact A combination of the parameters determines the overall performance of the ASTC encoder. The fastest mode sets all parameters to their minimum values to prioritize speed, whereas the exhaustive mode uses maximum values to maximize quality. The fast, medium, thorough, and verythorough modes employ intermediate values between these two extremes, providing a range of options to accommodate different usage scenarios.

ANALYSIS OF THE *ASTCENC* ENCODING PROCEDURE

Algorithm 1 Compress block

```
1:  Initialize block and configuration
2:  Identify block type
3:  if block is constant color then
4:      Encode as FP16 or UNORM16 and return
5:  end if
6:  Calculate error threshold
7:  Compress 1-partition 1-plane symbolic block
8:  Compress 1-partition 2-plane symbolic block
9:  for 2 to 4 partitions do
10:     for each candidate partition do
11:         Compress multi-partition symbolic block
12:         if no significant improvement then
13:             break
14:         end if
15:     end for
16: end for
17: Handle error block
```

Overall flow Algorithm 1 presents an abstraction of the compress_block() function of the ASTC encoder. This function encodes an input image block and first generates a symbolic block representation that encapsulates all compression-related fields, which is eventually packed into a 128-bit physical block.

The first operation performed by this function is the initialization of the block configuration (Lines 1-5). If the block is a constant-color block, it is encoded in either FP16 or UNORM16 format depending on whether the dynamic range is HDR or LDR, and the

function terminates early. Otherwise, an error threshold for early termination is computed on a per-block basis (Line 6). If the color space of the block is L (luminance) or LA (luminance and alpha), this threshold is adjusted accordingly.

The subsequent encoding process proceeds progressively from simpler and more commonly used configurations to more complex and less frequently used ones. First, encoding is performed using a single partition with one plane (Line 7). If the resulting quality does not satisfy the required criterion, the number of planes is increased to two and encoding is repeated (Line 8). When determining which channel should be separated into an additional plane, the channels are tested in the order A-B-G-R. This ordering reflects the fact that the alpha channel often behaves differently from the RGB channels.

If the result from the single-partition configuration fails to meet the quality criterion, the number of partitions is increased incrementally, and encoding is performed for each candidate configuration (Lines 9-16). For each partition candidate, the one-plane compression described above is applied, and the total block error is computed by summing the errors of all partitions. If the resulting error is not lower than the best error obtained in the previous stage multiplied by a predefined scaling factor, further exploration with a larger number of partitions is terminated. For example, if the error of the two-partition configuration exceeds that of the one-partition configuration, the three-partition configuration is not evaluated.

Finally, if the current block still fails to satisfy the quality criterion, it is converted into a constant block using the color of the pixel at index 0, after which the compression process terminates. (Line 17)

Partitioning As outlined in Algorithm 2, find_best_partition_candidates() determines optimal block partitioning within the compress_block() loop. First, quantization error is adaptively adjusted by block size to balance weight accuracy and compression ratio (Line 1). Next, texels are clustered using K-means for N partitions (Line 2). As described in the section “Parameters related to partitions,” these clustered partitions are compared against ASTC candidates. Candidates are ranked by mismatch scores, and only those within Partition Index Limit are retained.

After determining the candidate partition patterns for error evaluation, the algorithm iterates over each candidate to compute its error (Lines 3-7). For each candidate, principal component vectors are computed for all partitions, and errors are evaluated separately

Algorithm 2 Find best partition candidates

```
1: Set weight error constant based on texels per block
2: Compute k-means partition ordering and get partition sequence
3: for each partition up to search limit do
4:     Compute avgs and dirs
5:     Compute metrics and errors
6:     Estimate weight quantization error per partition
7: end for
8: Interleave uncorrelated partitions and same chrominance partitions
9: Remove duplicate partitions
10: return Number of best partitions
```

under two endpoint models: uncorrelated-chrominance endpoints and same-chrominance endpoints. For each model, the sum of squared distances between texels and the lines defined by the endpoints is calculated, and the previously assigned weighted quantization error is incorporated into the final error metric.

The resulting partition candidates are stored in an interleaved array (Line 8), alternating between cases where the color components are uncorrelated and correlated, thereby preventing bias toward a specific endpoint model under a limited candidate budget. Since duplicate partitions may exist among these candidates, the array is traversed and duplicates are removed using a bitmask (Line 9). Only unique partitions are selected as optimal partitions, and the function terminates (Line 10). The number of selectable partitions is limited by Partitioning Candidate Limit.

In summary, the algorithm first performs K-means clustering to obtain partitions. In the first filtering stage, Hamming distance is used to retain only ASTC partition candidates that are similar to the clustered result. In the second selection stage, a small number of promising partition patterns are selected based on error evaluation under two endpoint models with additional quantization error compensation.

Compression of a symbolic block in 1-plane mode Algorithm 3 provides the pseudo-code for compressing a symbolic block in 1-plane mode. A similar procedure is used for 2-plane mode, but endpoint and weight computations are performed separately for each plane.

The first step is to select a function that computes the difference between the symbolic block and the original block, according to the number of partitions (Lines 1-2). Then, the ideal endpoint colors and weights are calculated. At this stage, the colors are used as-is, without any compression applied.

A loop iterates over each mode, performing deci-

Algorithm 3 Compress 1-plane symbolic block

```
 1: Compute symbolic block difference
 2: Compute ideal colors and weights
 3: for each decimation mode do
 4:     if mode is invalid for current quantization then
 5:         Skip this decimation mode
 6:     end if
 7:     Compute ideal weights for decimation
 8: end for
 9: Calculate minimum endpoint clamp values
10: Compute angular endpoints
11: for each block mode do
12:     if quantization exceeds limits or insufficient bits
        available then
13:         Set high error and continue
14:     end if
15:     Compute quantized weights for decimation
16:     Compute error of weight set
17: end for
18: Compute ideal endpoint formats
19: for each candidate block mode do
20:     for each refinement iteration do
21:         Recompute ideal colors
22:         Pack colors and endpoints
23:         Pre-realign test
24:         Adjust and realign weights if necessary
25:         Post-realign test
26:     end for
27: end for
28: return  Best error value from evaluated modes
```

mation of the ideal (as-is) data, which is a data reduction step within the overall compression process (Lines 3-8). If a mode is invalid, it is skipped. The decimation function (compute_ideal_colors_and_weights_1plane) estimates the principal component axis for each partition, projects each texel onto this axis to obtain scalar parameters, and computes the ideal endpoint colors and per-texel weights in the [0,1] range. The quantization error of the weights is recorded for later use in weight grid selection, precision choice, and endpoint quantization baseline searches. If the number of partitions exceeds one, this procedure is applied identically to each partition.

To further improve compression quality, the angular algorithm adjusts the endpoints based on the distribution of weights, using the range between the highest and lowest weight values as guidance (Line 10).

A loop then generates a set of quantized, decimated weights and calculates the quantization errors (Lines 11-17). Using these weights and endpoints, the most suitable combinations of endpoint color encodings and weight candidates are selected. If weights exist for only some texels, the remaining weights are interpolated and optimized using a steepest-descent grid solver to minimize the overall block error.

The endpoint format and quantization level are then determined to optimally compress the block (Line 18). Multiple compression candidates are evaluated, and the combinations yielding the lowest error are selected.

The next loop calculates the error for each candidate and identifies the optimal block that minimizes error, determining the endpoint colors and weight quantization (Lines 19-27). This refinement process is repeated for a specified number of iterations (Line 20). Since weight decimation and quantization may slightly modify the weights, the ideal endpoint colors and weights are recalculated before further refinement (Line 21). If the color quantization level has changed, the newly computed colors and endpoints are re-quantized and packed. If all endpoints have the same color, additional bits may be allocated for storage and utilized (Line 22).

The pre-realign test, weight realignment, and post-realign test are as follows. Before and after the realignment step, which adjusts decimated weights upward or downward, the differences between the original and compressed blocks are recalculated (Lines 23-25). If the newly computed error in the current iteration is lower than the previous best and below a threshold, further candidate evaluation can be terminated early. Finally, the function returns the best error value computed (Line 28).

Comparison of 1- and 2-plane modes If the color channels are highly correlated, all channels can share a single weight plane (1-plane mode). However, this is not always the case. For example, in an RGBA texture, the alpha channel represents transparency and varies independently from the RGB color information, resulting in low correlation between the alpha and RGB channels. This makes 1-plane encoding less accurate for blocks with transparency. Similarly, in normal maps, the X and Y components of the normal vector often vary independently, necessitating separate weight planes for accurate encoding.

To address these situations, ASTC provides dual-plane encoding, offering a more suitable encoding method for each texture. In practice, the implementation calculates certain channels (such as alpha or normals) and the color textures (RGB) independently, and then combines them later. However, in 2-plane mode, an additional weight plane must be stored. This reduces the available bits for weights and partitions, potentially limiting the effectiveness of 2-plane mode

TABLE 5: Comparison between 1- and 2-plane modes

Feature	1-plane	2-plane
Weight handling	All channels share a single plane	Specific channels are handled separately
Channel correlation	Suitable when all the color channels are highly correlated	Suitable when certain channels vary independently from others
4-partition support	Yes	No
Use cases	RGB textures and single-channel textures	RGBA textures and 2-channel normal maps
Advantages	Higher data efficiency; supports more color partitions	Accurate representation of independently varying channels

for blocks with complex color variations. Table 5 summarizes the comparison of 1- and 2-plane modes.

EXPERIMENTS AND RESULTS

In this section, we analyze and compare the compression quality and speed of *astcenc* introduced in the previous section. The results include comparisons by mode and block size, as well as comparisons with other representative ASTC/ETC2 encoders.

Experimental setup

Experiments were conducted on a Windows 10 desktop equipped with an AMD Ryzen 5 5600X CPU, 16 GB RAM, 1 TB M.2 SSD, and an AMD Radeon RX 6600 XT GPU. For quality evaluation, PSNR and FLIP (v1.6) [12] metrics were used; lower raw FLIP values indicate better perceptual quality. All tests used *astcenc* v5.7 on 64 RGB/RGBA images from the QuickETC2 dataset [17], [18] across three block sizes: 4x4 (small), 6x6 (medium), and 8x8 (large). Plotting was performed using Python 3.

Other encoders

In addition to Arm's reference encoder *astcenc*, several other ASTC encoders exist. The characteristics of the following ASTC encoders are summarized in Table 6.

ISPC Texture Compressor [14] was designed for high-speed texture encoding using ISPC (Intel Implicit SPMD Program Compiler) [13] for exploiting both multi-threading and SIMD vectorization. It supports ASTC as well as BCn [1] and ETC1 [2] formats. Its encoding process can be summarized as follows: the input texture is first divided into blocks. Each block is then ranked according to suitable modes, and blocks using the same mode are grouped together. Finally, all blocks in a group are encoded collectively. Similar to *astcenc*, this tool provides two optimization levels (fast and slow) to allow a speed-quality trade-off depending on the number of mode candidates and endpoint refinement level. Upon its initial release, it was reported to be tens of times faster than *astcenc*; however, its development was discontinued in 2023.

TABLE 6: Comparison of ASTC Encoders

Encoder (& version)	Availability	Development period	Supported features
astcenc [6] 5.7	Source code	2013–Present	Full support
ISPC Texture Compressor [14]	Source code	2017–2023	LDR only, block sizes up to 8×8
astcrt [11]	Source code	2016–2017	LDR 4×4 block only
TexNN [15]	Not publicly available	2018	N/A
NVTT 3 [16]	SDK	2020–Present	LDR only

astcrt [11] was designed for real-time texture compression in web environments. It prioritizes low memory usage and high execution speed, through simplified endpoint and weight selection, partitioning, and heuristics for solid and grey blocks. While *astcrt* achieved tens of times higher speed than the fastest mode of *astcenc* 1.3, it also introduced block artifacts and less smooth gradients, resulting in significantly lower compression quality.

TexNN [15] reformulates complex search-stage problems as classification tasks, which are predicted using pre-trained neural networks. TexNN employs two separate neural networks: P-NN (Partition Neural Network) predicts partition counts and indices, while DEM-NN (Decimation and Endpoint Mode Neural Network) predicts endpoint and decimation modes independently. At the publication time, TexNN achieved approximately 7× speed improvement compared to *astcenc*'s exhaustive mode. Unlike advanced neural texture codecs (e.g., NTC [36] and TSNC [37]) that utilize learned compressed representations, thereby requiring a neural decoder, TexNN retains the standard ASTC bitstream and uses neural networks only to accelerate encoder decisions.

NVIDIA Texture Tools 3 (NVTT 3) [16] is a CUDA-based GPU encoder supporting both BC and ASTC.

Unlike the previously described encoders, only an installer and SDK are provided, not the full source code.

In this study, among the four encoders other than *astcenc*, only ISPC Texture Compressor was selected for comparison. This encoder is currently integrated into Unreal Engine alongside *astcenc* and is widely used by developers. For integration, developers can use a fast preset during iterative Unreal mobile cooks, apply block-size overrides by texture group, and switch to a higher-quality mode (slow for ISPC_texcomp or thorough for *astcenc*) for the shipping cook; in custom SDK pipelines, a proper encoder can be called as an offline command-line step or linked as a library [9].

In contrast, *astcrt* has long been discontinued and supports only the 4x4 block size, which significantly limits its applicability. NVIDIA Texture Tools 3 is implemented based on CUDA and therefore cannot be executed in our experimental environment, which uses an AMD GPU. Finally, TexNN is not publicly available in the form of source code or binary files, making experimental evaluation infeasible.

Along with ASTC, a widely used texture codec in mobile environments is ETC2 [3]. ETC2 was adopted as part of the OpenGL ES 3.0 standard, and most currently available Android and iOS devices support this codec. Therefore, developers can choose ETC2 instead of ASTC for reasons such as compatibility or encoding speed. From this perspective, the latest ETC2 encoding methods, QuickETC2-HQ [19] and H-ETC2 [20], were compared with the ASTC encoders in terms of compression quality and encoding speed.

Quantitative and Qualitative Comparison Results

A graph comparing the perceptual quality (FLIP) and encoding speed (MPixels/s) of each encoder is shown in Figure 3. The six data points on each *astcenc* line correspond to the six columns in Table 4, while the two ISPC_texcomp data points correspond to the slow and fast modes of the ISPC encoder.

ISPC Texture Compressor [14] currently exhibits a less favorable speed-quality trade-off compared to *astcenc* 5.7. This is likely due to continuous optimizations applied to *astcenc*, whereas development of ISPC Texture Compressor has been discontinued. In particular, the slow mode shows considerably lower encoding speed compared to the fast mode, despite minimal quality gains.

The ETC2 encoders (4bpp for RGB and 8bpp for RGBA) generally achieve slightly lower quality than *astcenc* (6x6 block size) but demonstrate markedly higher encoding speed. Specifically, QuickETC2-HQ [19] is approximately ten times faster than *astcenc*'s fastest mode, suggesting that ETC2 can be useful for latency-constrained encoding when its format limitations are acceptable.

Smaller block sizes show less variation in speed and quality between modes, whereas larger block sizes exhibit greater differences. Accordingly, the medium mode is sufficient for larger blocks, while smaller blocks benefit from higher-quality modes. For all block sizes, the verythorough and exhaustive modes provide only marginal quality improvements while significantly reducing encoding speed, indicating that thorough is generally sufficient for production builds except in special cases. Optimal block sizes are content-dependent; detailed guidelines are provided in Section "Block size configuration."

To support this analysis, PSNR delta values relative to the thorough mode were calculated for each mode at the 6x6 block size using the *kodim* images in the QuickETC2 dataset (Figure 4). Consistent with the FLIP results, the two modes slower than thorough did not yield significant quality improvements. Conversely, *astcenc*'s fast and fastest modes, as well as the ISPC encoder's modes, showed quality reductions of roughly 1 dB, confirming a significant speed-quality difference between modes.

A qualitative comparison was also performed using the *Jelly* texture from the QuickETC2 dataset, which exhibits the most prominent compression artifacts due to diagonal edges and alpha transitions (Figure 5). Encoder results were evaluated for each mode and block size. The ISPC encoder displayed some block-level artifacts, manifesting as abrupt color discontinuities and color bleeding near saturated region boundaries, even at the highest-quality 4x4 block size when compared to *astcenc*. As block size increased, these artifacts became more pronounced, manifesting as stair-stepping and haloing along diagonal boundaries. At the 8x8 block size, all the encoders used in the experiment suffer from noticeable compression artifacts regardless of the compression mode.

The quality differences between *astcenc*'s fastest and exhaustive modes were negligible for small block sizes but became more pronounced with increasing block size. These observations are consistent with the quantitative analysis. To address this, *astcenc* provides intermediate modes–fast, medium, thorough, and verythorough–offering a controllable trade-off between encoding speed and output quality. This highlights the importance of selecting the appropriate mode based on specific application requirements.

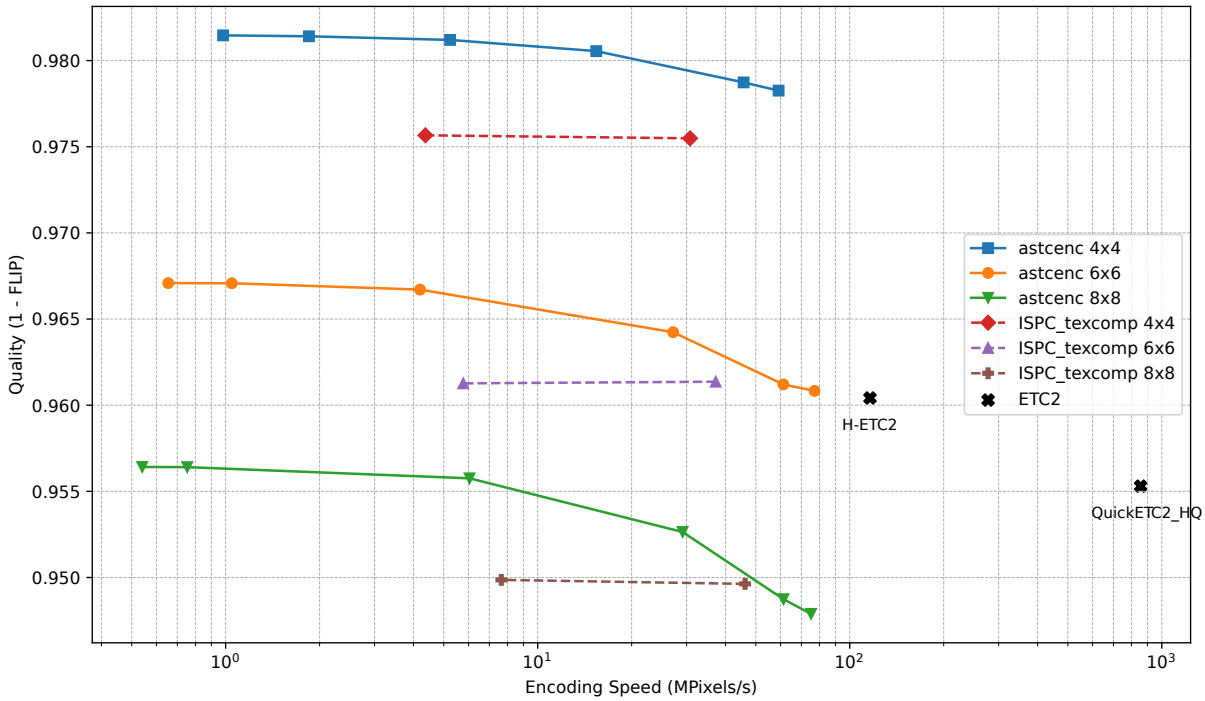

FIGURE 3: Quantitative comparison of *astcenc* and other encoders. The astcenc data points follow the six presets in Table 4, ranging from exhaustive to fastest, while the ISPC_texcomp data points represent the slow and fast modes. Higher 1-FLIP indicates better quality. The supplemental material includes a table presenting the numerical figures plotted in the graph.

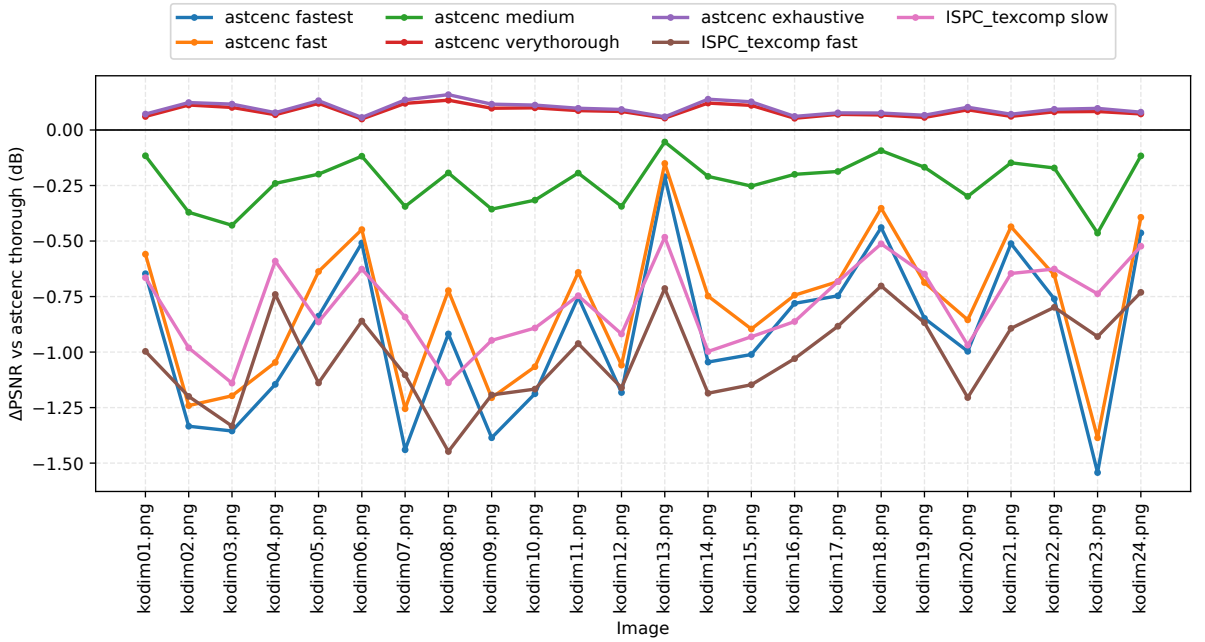

FIGURE 4: Relative PSNR changes compared to *astcenc*'s thorough mode at the 6x6 block size for the *kodim* dataset.

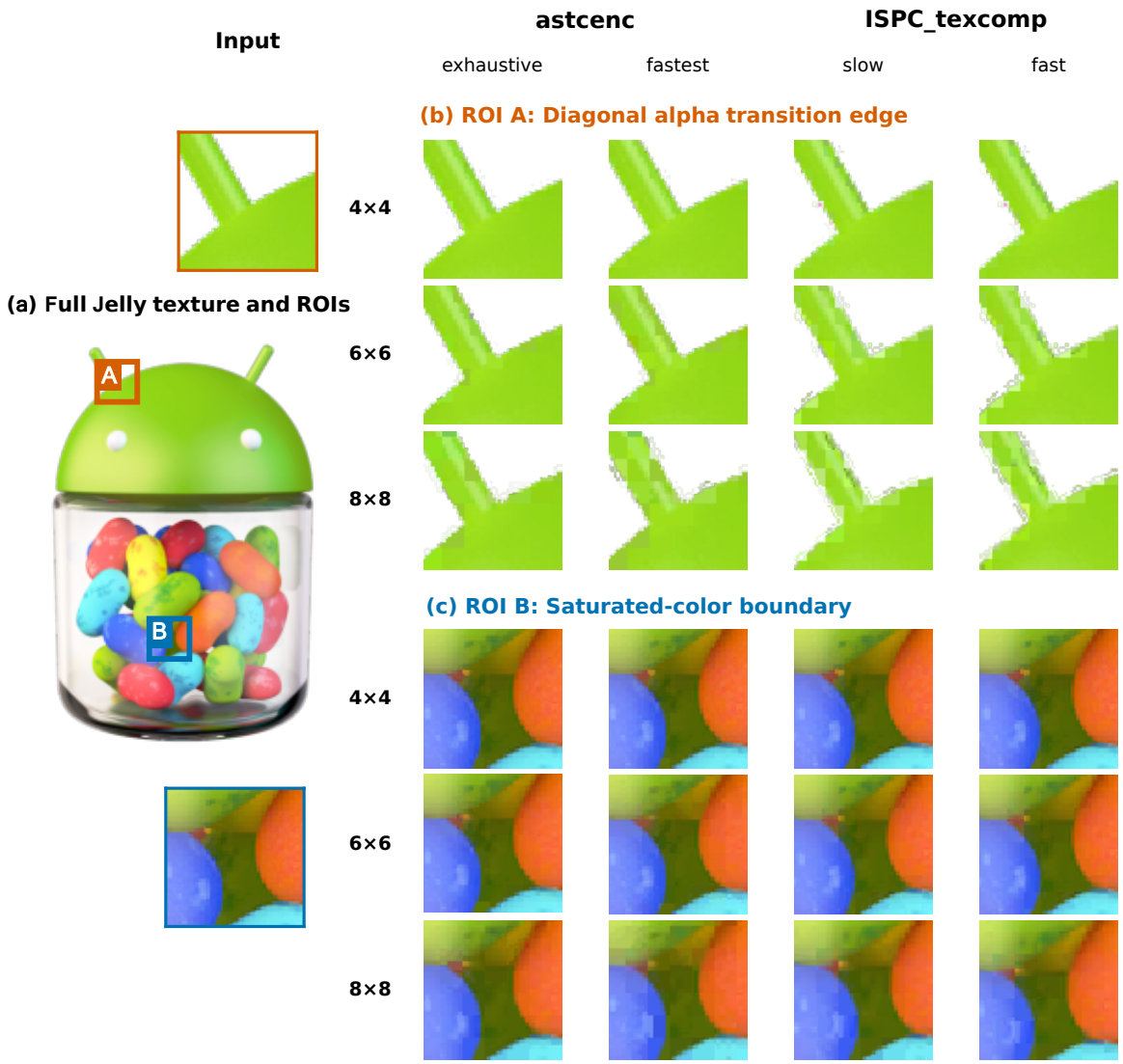

FIGURE 5: Qualitative comparison on the *Jelly* texture. The overview indicates two regions of interest (ROIs), and the enlarged crops compare *astcenc* and *ISPC_texcomp* at 4×4, 6×6, and 8×8 block footprints. The supplemental material includes original and compressed image files. (Original image courtesy of Google)

LITERATURE REVIEW ON ASTC

Since the introduction of ASTC, numerous follow-up studies have been conducted. In this section, we review representative studies and classify them into six categories, as outlined below.

Enhanced codec

ASTC allows different block sizes to be set for each texture map; however, each texture map can only be compressed using a single block size. When a texture map exhibits significant density variation, using a large block size can result in noticeable quality differences between blocks, whereas using a small block size may unnecessarily reduce compression efficiency in uniform regions. To address this limitation, Krajcevski et al. [21] proposed an improved ASTC codec, named VBTC, supporting variable block sizes. This approach uses larger blocks for sparse regions to improve compression efficiency and smaller blocks for dense regions to preserve quality, achieving better visual quality and energy efficiency compared to standard ASTC. However, this method requires an additional indirect memory access to retrieve block size information, which necessitates modifications to both software encoders and hardware decoders. Consequently, no devices currently support this approach.

Block size configuration

ASTC provides a wide range of block sizes, making it challenging to select the optimal size for each texture map in large datasets. Accordingly, Arm's developer guide [22] recommends 6x6 or 8x8 for color textures, 4x4 or 5x5 for 2D UI elements, and 6x6 for normal maps. Nevertheless, textures within the same category vary significantly; excessive compression introduces visible artifacts, whereas overly conservative settings increase memory and storage overhead.

To alleviate the manual burden of selecting optimal block sizes, Nah [23], [24] proposed methods that automatically adjust compression to meet a target PSNR. These methods iteratively adjust block sizes until the target PSNR is achieved, which naturally increases compression time. In his first work [23], a batch program used the fastest mode for block size selection and the thorough mode for final compression. In a subsequent study [24], this process was integrated into *astcenc*, using only a subset of pixels during block size evaluation to remove mode restrictions and reduced variance in PSNR and output quality. Popov [25] developed another batch program that selects block sizes based on SSIMULACRA2 [26] instead of PSNR and supports mipmap generation, which *astcenc* lacks.

Griffin and Olano [27] emphasized that block size selection should consider masking effects during rendering, rather than focusing solely on the quality of individual texture maps. For PBR texture sets (e.g., diffuse, normal, and roughness), the key metric is artifact visibility in the final rendered image across multiple viewing angles. Masking effects and perceptual sensitivity depend on texture type, larger blocks can often be used to increase compression aggressiveness without perceptible quality loss. Through psychophysical experiments, Lavoué et al. [28] confirmed that artifact detectability varies significantly with masking effects and found that perceptual metrics like SSIM correlate better with human visual perception than PSNR.

RGBM encoding

RGBM encodes HDR RGB colors into LDR textures by storing a multiplier (*M* channel) to reconstruct brightness. Compressing RGBM with ASTC presents unique challenges [29]: low correlation between RGB and *M* channels often necessitates two-plane encoding, *M*-channel errors severely impact quality, and pixels can have multiple valid RGBM representations. To address this, Harris [29] proposed a pre-processing method that unifies *M* values within a block and adjusts RGB values accordingly. This eliminates the need for a separate weight plane, achieving higher compression efficiency while preserving perceptual quality.

Hardware decoding

GPUs supporting ASTC include a hardware decoder designed to deliver real-time performance while minimizing silicon area. While most GPU vendors do not publicly disclose decoder architectures, Imagination Technologies [30] introduced a lightweight architecture that decodes 2×2 texel blocks simultaneously. However, if the four texels belong to different ASTC blocks, additional decoding steps are required, which makes the architecture particularly efficient with larger block sizes. Experiments showed that this design improves performance per unit area by approximately 25% compared to using individual decoders for each texel.

Use in computer vision and neural networks

Žádník et al. [31] evaluated texture compression formats for computer vision tasks as alternatives to JPEG or HEVC. ASTC in the exhaustive mode outperformed JPEG at low bitrates for object detection and provided superior results for semantic segmentation across all bitrates. Even the fastest ASTC mode performed comparably to or better than other codecs, such as BC1 and YCoCg-BC3, in certain bitrate scenarios. However, overall ASTC encoding and decoding are 1–2 orders of magnitude slower than these alternatives. In follow-up work, Žádník et al. [32] reduced encoding complexity for latency-sensitive computer vision systems using simplified JPEG XS and ASTC. For image classification and semantic segmentation tasks, block size selection initially degraded performance by 6.4–16.8 percentage points (pp). After retraining the networks on compressed data, the performance loss was reduced to 1.8–5.0 pp. Additionally, this simplified approach achieved over twice the encoding speed of JPEG.

ASTC compression applies not only to neural network inputs but also to the network weights themselves. Since GPUs with native ASTC support enable on-device decoding without additional latency, weight compression using ASTC is particularly attractive for edge deployment. Building on this advantage, Sharma et al. [33] demonstrated that the weights of LeNet and AlexNet could be compressed by up to $23.2\times$ and $10.8\times$, respectively, with a 5% allowable accuracy loss. Later, Apple Intelligence [34] compressed pre-trained server-model weights using ASTC HDR-ch mode with 6×6 blocks, reducing storage from 16 to 3.56 bits per weight with minimal impact on MMLU (Massive Multitask Language Understanding) performance ($80.0 \rightarrow 79.2$).

Supercompression

Basis Universal by Binomial LLC [35] is a GPU-friendly transcoding format for LDR and HDR textures and has been adopted in the KTX 2.0 standard. It compresses textures into ETC1S or UASTC, where UASTC is compatible with a subset of ASTC 4x4 block modes. Basis Universal supports rate–distortion optimization (RDO), enabling lossy compression to further reduce storage requirements. UASTC-compressed textures can be transcoded to ASTC or other standard formats (BCn, ETC1/2, and PVRTC1), making the format suitable for a wide range of platforms, from mobile devices to desktop and console systems.

CONCLUSIONS

This paper presented a system-level analysis of the internal mechanisms and performance characteristics of ASTC encoding technology, with a focus on the reference encoder *astcenc*. Through an experimental evaluation, we examined improvements in the latest version of *astcenc* by comparing it with the legacy ISPC Texture Compressor. We further provided quantitative and qualitative analyses of how block sizes and encoding modes affect the quality–speed trade-off. In addition, we reviewed research related to ASTC.

Based on these findings, we propose several directions for future ASTC research. First, encoding performance could be further improved by leveraging statistical modeling and deep learning techniques. Achieving this goal would require new training methodologies, large-scale datasets, and architectural innovations beyond existing approaches such as TexNN [15].

Second, improvements may be explored not only at the encoder level but also at the codec level. The current ASTC codec compresses each image independently. Approaches similar to NTC [36] and TSNC [37], which jointly compress multi-channel textures, may lead to improved compression efficiency. Furthermore, if intra-image variable block-size support proposed in VBTC [21] were implemented in hardware decoders, it could further enhance texture mapping quality in real-time rendering applications.

ACKNOWLEDGMENTS

The authors thank the developers of *astcenc* for publicly releasing their code, as well as the reviewers for providing constructive feedback on our manuscript. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2025-00521436).

REFERENCES

1. Microsoft, “Texture block compression in Direct3D 11,” 2020. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d11/texture-block-compression-in-direct3d-11
2. Jacob Ström and Tomas Akenine-Möller, “iPACKMAN: high-quality, low-complexity texture compression for mobile phones,” in *Proc. of Graphics Hardware*, 2005, pp. 63–70.
3. Jacob Ström and Martin Pettersson, “ETC 2: texture compression using invalid combinations,” in *Proc. of Graphics Hardware*, 2007, pp. 49–54.
4. Simon Fenney, “Texture compression using low-frequency signal modulation”, in *Proc. of Graphics Hardware*, 2003, pp. 84–91.
5. Jørn Nystad, Anders Lassen, Andy Pomianowski, Sean Ellis, and Tom Olson, “Adaptive scalable texture compression,” in *Proc. of High-Performance Graphics*, 2012, pp. 105–114.
6. Arm Limited. “astc-encoder,” 2026. Available: https://github.com/ARM-software/astc-encoder
7. Jørn Nystad, Anders Lassen, and Thomas J. Olson, “Flexible texture compression using bounded integer sequence encoding,” in *SIGGRAPH Asia 2011 Sketches*, 2011, Article 32.
8. Andrew Garrard, “Khronos Data Format Specification v1.4.0,” The Khronos Group Inc., 2025. Available: https://registry.khronos.org/DataFormat/specs/1.4/dataformat.1.4.html
9. Arm Limited, “Adaptive Scalable Texture Compression User Guide,” Issue 01, 2020. Available: https://developer.arm.com/-/media/ArmDeveloperCommunity/PDF/ASTC_User_Guide_102162_0001_01.pdf
10. Jon Rrocatis, “ASTC Partitions,” 2018. Available: https://rockets2000.wordpress.com/2018/01/09/astc-partitions
11. Daniel Oom, “Real-Time Adaptive Scalable Texture Compression for the Web,” Master’s thesis, Chalmers University of Technology, 2016.
12. Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild, “ꟻLIP: A Difference Evaluator for Alternating Images,” *Proc. of the ACM on Computer Graphics and Interactive Techniques*, 2020, Article 15.
13. Matt Pharr and William R. Mark. “ispc: A SPMD compiler for high-performance CPU programming,” in *2012 Innovative Parallel Computing*, 2012, pp. 1–13.
14. Intel, “Fast ISPC Texture Compressor,” 2023. Available: https://github.com/GameTechDev/ISPCTextureCompressor
15. Srihari Pratapa, Tom Olson, Alex Chalfin, and Di-

nesh Manocha, “TexNN: Fast Texture Encoding Using Neural Networks,” *Computer Graphics Forum*, vol. 38, 2019, pp. 328–339.
16. NVIDIA, “NVIDIA Texture Tools 3,” 2026. Available: https://developer.nvidia.com/gpu-accelerated-texture-compression
17. Jae-Ho Nah, “QuickETC2: How to Finish ETC2 Compression within 1 ms,” in *ACM SIGGRAPH 2020 Talks*, 2020, Article 4.
18. Jae-Ho Nah, “QuickETC2: Fast ETC2 Texture Compression using Luma Differences,” *ACM Transactions on Graphics*, vol. 39, no. 6, 2020, Article 270.
19. Jae-Ho Nah, “QuickETC2-HQ: Improved ETC2 encoding techniques for real-time, high-quality texture compression,” *Computers & Graphics*, vol. 116, 2023, pp. 308–316.
20. Hyeon-ki Lee and Jae-Ho Nah, “H-ETC2: Design of a CPU-GPU Hybrid ETC2 Encoder,” *Computer Graphics Forum*, vol. 42, iss. 7, 2023, Article e14969.
21. Pavel Krajcevski, Abhinav Golas, Karthik Ramani, Michael C. Shebanow, and Dinesh Manocha, “VBTC: GPU-Friendly Variable Block Size Texture Encoding,” *Computer Graphics Forum*, vol. 35, iss. 2, 2016, pp. 409–418.
22. Arm Limited, “Adaptive Scalable Texture Compression Developer Guide. Version 4.3,” 2023. Available: https://developer.arm.com/documentation/102162/latest
23. Jae-Ho Nah, “ASTC Block-Size Determination Method based on PSNR Values,” *Journal of the Korea Computer Graphics Society*, vol. 28, no. 2, 2022, pp. 21–28.
24. Jae-Ho Nah, “Addition of an adaptive block-size determination feature to *astcenc*, the reference ASTC encoder,” *Software Impacts*, vol. 17, 2023, Article 100569.
25. Oleksandr Popov, “Automated ASTC compression,” 2025. Available: https://github.com/keaukraine/astc-compression
26. Jon Sneyers, “SSIMULACRA 2 - Structural SIMilarity Unveiling Local And Compression Related Artifacts,” 2023. Available: https://github.com/cloudinary/ssimulacra2
27. Wesley Griffin and Marc Olano, “Objective image quality assessment of texture compression,” in *Proc. of Symp. on Interactive 3D Graphics and Games*, 2014, pp. 119—126.
28. Guillaume Lavoué, Michael Langer, Adrien Peytavie, and Pierre Poulin, “A Psychophysical Evaluation of Texture Compression Masking Effects,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, iss. 2, 2019, pp. 1336–1346.
29. Peter Harris, “High quality RGBM texture compression with ASTC,” Arm Community, 2019. Available: https://developer.arm.com/community/arm-community-blogs/b/mobile-graphics-and-gaming-blog/posts/high-quality-rgbm-texture-compression-with-astc
30. Kenneth C. Rovers and Sam Elliott, “On Improving the Performance Per Area of ASTC with a Multi-output Decoder,” in *Proc. of IEEE Symp. on Computer Arithmetic*, 2017, pp. 58–59.
31. Jakub Žádník, Markku Mäkitalo, Jussi Iho, and Pekka Jääskeläinen, “Performance of Texture Compression Algorithms in Low-Latency Computer Vision Tasks,” in *European Workshop on Visual Information Processing*, 2021, pp. 1–6.
32. Jakub Žádník, Markku Mäkitalo, and Pekka Jääskeläinen, “Pruned Lightweight Encoders for Computer Vision,” in *IEEE Workshop on Multimedia Signal Processing*, 2022, pp. 1–6.
33. Hardik Sharma, Tom Olson, and Alex Chalfin, “Using Texture Compression Hardware for Neural Network Inference,” in *Hot Chips: A Symp. on High Performance Chips (Posters)*, 2017.
34. Ethan Li et al., “Apple Intelligence Foundation Language Models: Tech Report 2025,” arXiv preprint arXiv:2507.13575, 2025.
35. BinomialLLC. “basis_universal v2.5,” 2026. Available: https://github.com/BinomialLLC/basis_universal
36. Karthik Vaidyanathan, Marco Salvi, Bartlomiej Wronski, Tomas Akenine-Möller, Pontus Ebelin, and Aaron Lefohn, “Random-Access Neural Compression of Material Textures,” *ACM Transactions on Graphics*, vol. 42, no. 4, 2023, Article 88.
37. Laurent Belcour and Anis Benyoub, “Hardware accelerated neural block texture compression with cooperative vectors,” in *Proc. of High-Performance Graphics*, 2025.

In-hyuk Jeong is an independent researcher based in Seoul, South Korea. His research interests include computer graphics and robotics. Jeong received his B.S. degree in computer science from Sangmyung University, Seoul, South Korea, in 2025. Contact him at inhyukjeong10@gmail.com.

Jae-Ho Nah is an assistant professor at Sangmyung University, Seoul, South Korea. His research interests include ray tracing, rendering algorithms, and graphics hardware. Nah received his Ph.D. degree in computer science from Yonsei University, Seoul, South Korea, in 2012. He is a member at IEEE, ACM, and KCGS. He is the corresponding author of this paper. Contact him at jaeho.nah@smu.ac.kr.