GST: GPU-decodable Supercompressed Textures

Pavel Krajcevski*[1], Srihari Pratapa†[1] and Dinesh Manocha‡[1]

[1]The University of North Carolina at Chapel Hill

http://gamma.cs.unc.edu/GST/

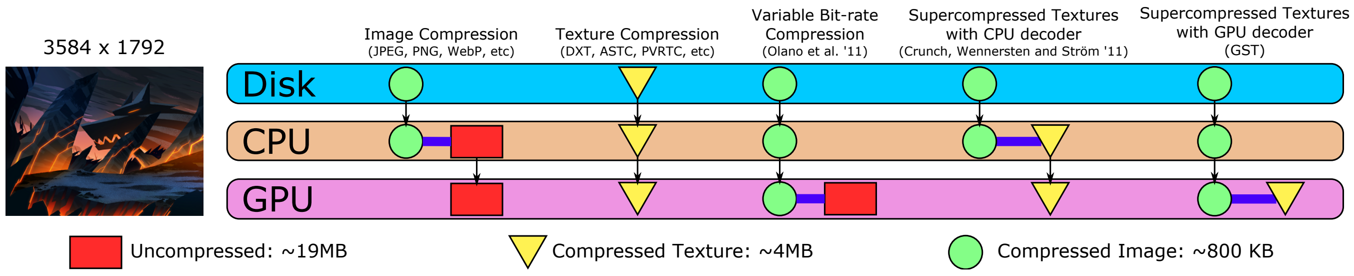
3584 x 1792
Image Compression
(JPEG, PNG, WebP, etc)
Texture Compression
(DXT, ASTC, PVRTC, etc)
Variable Bit-rate Compression
(Olano et al. '11)
Supercompressed Textures with CPU decoder
(Crunch, Wennersten and Ström '11)
Supercompressed Textures with GPU decoder
(GST)
Disk
CPU
GPU
Uncompressed: ~19MB
Compressed Texture: ~4MB
Compressed Image: ~800 KB

Figure 1: *An overview of the methods for sending texture data from the disk to the GPU. Each vertical arrow indicates data sent over a bandwidth-limited channel. Each horizontal bar represents a decoding procedure. Green circles represent entropy-encoded data used for storage and streaming. Yellow triangles represent compressed texture data for use with hardware decoders in commodity GPUs. Red boxes represent the full uncompressed image data. We present a new method (GST) for maintaining a compressed format across all bandwidth-limited channels that decodes directly into a compressed texture on the GPU. Compared to prior techniques, our approach has the lowest CPU-GPU bandwidth requirements while maintaining compressed textures in GPU memory. Texture courtesy of Trinket Studios.*

Abstract

Modern GPUs supporting compressed textures allow interactive application developers to save scarce GPU resources such as VRAM and bandwidth. Compressed textures use fixed compression ratios whose lossy representations are significantly poorer quality than traditional image compression formats such as JPEG. We present a new method in the class of *supercompressed* textures that provides an additional layer of compression to already compressed textures. Our texture representation is designed for endpoint compressed formats such as DXT and PVRTC and decoding on commodity GPUs. We apply our algorithm to commonly used formats by separating their representation into two parts that are processed independently and then entropy encoded. Our method preserves the CPU-GPU bandwidth during the decoding phase and exploits the parallelism of GPUs to provide up to 3X faster decode compared to prior texture supercompression algorithms. Along with the gains in decoding speed, our method maintains both the compression size and quality of current state of the art texture representations.

Keywords: texture compression, image compression, GPGPU

Concepts: •**Computing methodologies** → **Image compression;** *Graphics processors; Graphics file formats;*

*pavel@cs.unc.edu
†psrihariv@cs.unc.edu
‡dm@cs.unc.edu

SA '16 Technical Papers,, December 05 - 08, 2016, Macao
ISBN: 978-1-4503-4514-9/16/12
DOI: http://dx.doi.org/10.1145/2980179.2982439

1 Introduction

For over a decade, commodity graphics hardware has shipped with dedicated compressed texture decoding units. Classically, these units decode a fixed number of bits into a block of pixels of predetermined dimension to use with the texture sampling pipeline. Storing compressed textures with respect to these hardware capabilities reduces the amount of bandwidth needed to transfer a texture into dedicated video memory, and the compressed representation allows for significantly more texture data to reside on the GPU.

Hardware texture compression formats map nicely to GPU architectures by allowing random-access to texture data. However, random access requires the texture to be encoded using fixed compression rates. In contrast to image compression formats such as PNG and JPEG [1992], which provide up to 50:1 compression, GPU texture formats commonly provide lower quality for file sizes at 6:1 compression. As an example, a 4K video frame (19MB uncompressed) requires 345KB of storage as a JPEG, whereas the same video frame requires 3.21MB as a pure DXT compressed texture. This discrepancy is largely due to random access hardware requirements preventing the use of variable-length encoding techniques that are used in image compression. The result is that application developers must choose between optimizing their data for streaming and optimizing for run-time efficiency, as image compression formats such as JPEG decode into fully uncompressed textures in memory. This trade-off becomes even more troublesome for applications that stream their texture assets over a low-bandwidth channel such as network-enabled GIS applications (e.g., Google Maps) and video game streaming services [Pohl et al. 2014]. Additionally, large virtual environments that cannot store all of their rendering data in memory would significantly benefit from lower disk-to-VRAM latency in order to avoid noticable loading artifacts.

In order to tackle the limitations of fixed-rate compression, recent work has focused on *supercompressing* the textures [Geldreich 2012; Ström and Wennersten 2011]. In other words, an additional layer of compression is used in order to encode the already compressed representation in preparation for storage on disk. These methods typically process the compressed texture representations in preparation for an entropy encoding step, such as Huffman or arithmetic encoding providing an additional 2-3X compression to regain the advantage of compressed image sizes on disk. However, decoding the texture on the CPU eschews the main benefits of compressing textures: the gained bandwidth across the CPU-GPU bus. This bandwidth is even more important in mobile devices that have power constrained GPUs [Leskela et al. 2009].

In this paper, we present a new supercompression algorithm GST, pronounced *jist*, for decompressing textures on the GPU into hardware-compressed formats. Our three main contributions include:

1. A new supercompressed texture representation for endpoint-compressed formats;

2. A method for encoding textures into this format;

3. A parallel decoding algorithm suitable for SIMD architectures such as GPUs.

The basis of our algorithm is a state-of-the-art entropy encoding technique known as ANS that allows multiple compression streams to be interleaved and decoded in parallel on the GPU [Giesen 2014]. We exploit the underlying structure of commonly used endpoint compression formats in a way that increases the internal redundancy of the texture data, allowing for efficient static context modeling for the entropy encoder. Our approach saves both streaming and CPU-GPU bandwidth by providing compressed texture data to be decompressed by the device that will use it. Furthermore, one of our main benefits is the increased decoding speed realized by using massively parallel architectures.

We target the class of endpoint compression formats as described in Section 2.3. We first re-encode the per-pixel palette indices from a compressed representation into per-block dictionary entries. To improve redundancy between successive index blocks, we store the differences in sequential dictionary entries, similar to differential pulse-code modulation (DPCM). Next, we treat the separate palette-generating endpoints of each block as two low-resolution images for which we use a wavelet transform. Finally, both of these parts are written to disk using entropy encoding. Our current implementation rivals the state-of-the-art CPU codecs in compression size and quality with up to 3X improvement in decompression speed. This translates to about a 2-3X improvement in compression size over the original hardware-compressed formats, which is realized as additional gains in CPU-GPU bandwidth when the supercompressed texture data is sent to the GPU to be decoded. Our algorithm is designed from the ground up to target current desktop and mobile GPU architectures, and we show benefits to loading 4K video frames and large numbers of textures.

The rest of this paper is organized as follows. Section 2 discusses prior work and provides context for the dichotomy between image and texture compression algorithms. Section 3 gives an overview of our compression pipeline. Section 4 describes details on parallel encoding while Section 5 discusses implementation details for specific compressed formats. Results are described in Section 6, and a discussion of limitations and future work is covered in Section 7.

2 Background

Texture data bandwidth has been a major, well-studied issue for interactive graphics applications for decades. In the rest of this section, we give a brief overview of texture and image compression techniques.

2.1 Image Compression

Traditional image compression algorithms such as PNG and JPEG use a variable-rate entropy encoding step to allocate bits to pixels with respect to their amount of detail. In order to prepare an image for entropy encoding, the raw RGB data usually undergoes one or more transforms in order to increase the amount of redundancy in the data. For example, JPEG uses the discrete cosine transform to condense the information content of blocks of pixels [Wallace 1992], and its successor, JPEG-2000, uses an optionally lossless wavelet transform [Skodras et al. 2001]. Similarly, colors are usually transformed into color spaces that collect psychovisual detail into a single channel in order to increase overall compression efficiency. One such example is the lossless conversion of 8-bit RGB to a colorspace such as YCoCg [Malvar et al. 2008].

The entropy encoding stage of image compression algorithms is usually an inherently serial procedure that is difficult to parallelize. Olano et al. [2011] address this problem by proposing a new variable-rate compression scheme in which a GPU range codec is used to decompress the images by decoding differences between mip-levels. The resulting full-resolution textures are stored uncompressed in GPU memory.

2.2 Entropy Encoding

Techniques such as Huffman [1952], arithmetic [1979], and ANS [2013] encoding are the basis for many image compression formats [Buccigrossi and Simoncelli 1999]. Although entropy decoding algorithms are inherently serial due to their variable length output, we may use multiple encoders and decoders in parallel. Duda [2013] presented asymmetric numeral systems (ANS) that maintains an internal state consistent between the encoder and decoder. This property allows multiple encoding streams to be interleaved into a single data stream. Giesen [2014] uses this property to demonstrate how to create data streams that can be decoded in parallel using single-instruction multiple-data (SIMD) architectures. We give an overview of the range variant of ANS encoding and its use in our method in Section 3.3.

2.3 Texture Compression

The random access properties of compressed textures imply a fixed-rate compression ratio regardless of texture detail. This requirement allows texture mapping hardware to quickly compute an address to the underlying texture data. Typically, fixed-rate compression formats represent $N \times M$ blocks of pixels in some fixed number of bits. One of the earliest such representations was introduced by Delp and Mitchell [1979] to provide a two bit-per-pixel (bpp) lossy interpretation of eight-bit grayscale images. Later, many graphics architectures were proposed using similar compressed texture representations [Torborg and Kajiya 1996; Knittel et al. 1996; Beers et al. 1996].

Modern texture compression formats belong to one of two classes. The first class, known as *endpoint* compression formats, uses two low-precision RGB endpoints per block to generate a palette of colors by linear interpolation. Along with these two low-precision colors, a per-pixel palette index is stored to recreate the final pixel color [Delp and Mitchell 1979; Fenney 2003; OpenGL 2010; Nystad et al. 2012]. Among the first endpoint compressed texture formats available on commodity graphics hardware was DXT, introduced by Iourcha et al. [1999]. In this format, 4×4 blocks of pixels are represented using two 16-bit values and 16 two-bit values. The two 16-bit values are each interpreted as 565 RGB endpoints that generate a four-color palette for the block via linear interpolation. The following 16 two-bit values index into this palette to recreate the final pixel values. Fenney [2003] targeted the worst-case filtering step of DXT by providing a compression format that bilinearly interpolates palette data across block boundaries. Later, the BPTC specification and ASTC introduced many improvements upon the original DXT format that provide multiple palettes per block, variable precision palette indices, a variable number of indices per block, and even variable global block sizes [OpenGL 2010; Nystad et al. 2012].

The second class of texture compression formats is known as *tabled* compression formats. These formats store a single color per block of $N \times M$ pixels and per-pixel offsets. Ström and Akenine-Möller [2004; 2005] introduced the first tabled formats as PACKMAN and iPACKMAN. Later, Ström and Pettersson [2007] improved upon iPACKMAN by exploiting unused encoded representations to provide additional detail.

Although most texture hardware requires random-access to the pixel data, many architectures have been proposed to alleviate this constraint. Inada and McCool [2006] proposed texturing hardware that stores textures in a b-tree to allow for efficient pixel access. Similarly, Krajcevski et al. [2016] presented a scheme that provides variable bit-rate texture compression by adding a layer of indirection to dynamically select ASTC block sizes for regions of an image.

2.4 Supercompressed Textures

In order to reduce the streaming latency of textures, most applications store these compressed texture formats (e.g., DXT, ASTC, ETC) on disk without additional processing. Recently, however, there has been progress into targeting both on-disk compression and in-memory compression. Ström and Wennersten [2011] proposed a scheme for further compressing ETC2 textures. In their formulation, they predict the final pixel colors in order to predict the per-pixel indices for the given block. They observe a gain of up to 3X in some cases over existing ETC2 textures. A different approach, known as *Crunch*, developed by Geldreich [2012], uses a Huffman-encoded dictionary of endpoints and index blocks to further compress a DXT-encoded texture. Blocks are stored in order using the differences between successive dictionary entries. Both of these methods decode the compressed texture on the CPU before sending them to the GPU. In this paper, we present an approach for supercompression that preserves bandwidth using GPU decompression similar to Olano et al. [2011], while maintaining the decompression benefits of DXT.

3 Compression Pipeline

In this section we present our encoding scheme and discuss the techniques used to prepare our data for entropy encoding. Our supercompression algorithm is designed to be compatible with many widely used texture formats and to map well to current GPU architectures. Our approach is based on fulfilling the following design goals:

- The supercompressed texture representation should be directly decodable on SIMD architectures, such as GPUs, without additional processing.
- The final decoded result should be a compressed texture in GPU memory.
- The supercompressed texture representation should be agnostic to the underlying endpoint compression formats.

Given an endpoint compressed texture representation, our compression pipeline is organized in three stages, one for each of the con-

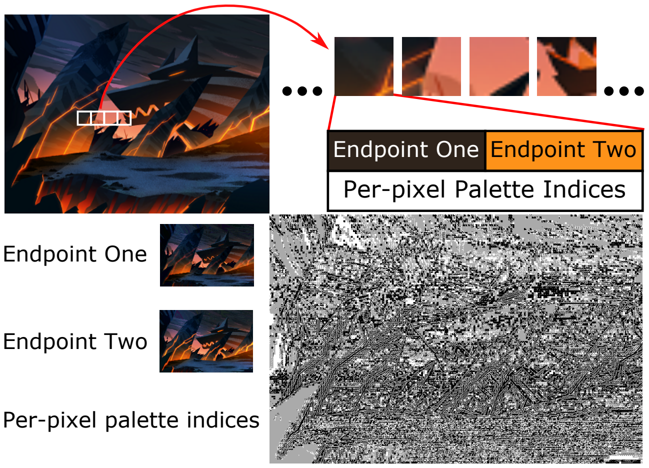

Figure 2: *The constituent parts of a compressed texture. Each endpoint compressed texture represents a sequence of equally sized blocks. Each block contains a fixed number of bits containing two endpoint colors that generate a palette and per-pixel index data. Here we show the endpoints separated into individual images and visualize the per-pixel indices. We re-encode the indices using VQ-style dictionary compression and transform the endpoint images using a wavelet transform prior to encoding the final texture using an entropy encoder.*

stituent parts of a compressed texture as described in Figure 2, and one for the final entropy encoding. Only the first stage introduces a minimal amount of error while the last two stages are lossless. In the first stage, starting with the original texture, we generate an initial target endpoint compressed representation. We then re-encode each compressed block in an attempt to reuse indices from successive blocks of pixels in preparation for dictionary encoding similar to vector quantization (VQ). In doing so, we also generate a new set of endpoints per-block. In the second stage, we independently process these endpoints as separate low-resolution images in preparation for entropy encoding. Finally, we combine similar data streams and encode each using the range variant of ANS before storing to disk [Duda 2013]. The final output of our compression pipeline is four ANS streams to be decoded as described in Section 4.

3.1 Index Block Dictionary Generation

The per-pixel palette indices are classically the most difficult piece of information to compress [Ström and Wennersten 2011][Waveren 2006]. The first step in our compression pipeline is a re-encoding stage as described in Figure 3. The purpose of this step is to recompute palette indices for blocks of pixels in a way that is conducive to dictionary construction, similar to VQ. The endpoints for each block are then optimized to fit these newly assigned indices. Our goal is to build a dictionary of *index blocks* representing $N \times M$ blocks of indices. As an example, a 4×4 block of pixels that uses two-bit palette indices will have a dictionary of thirty-two bit index blocks. The goal of the dictionary is to have many redundancies in order to map well to the final entropy encoding step of pipeline as described in Section 3.3.

We begin by using an existing codec such as DXT or PVRTC as a black box for providing the original compressed representation of a given texture. This codec is assumed to process each $N \times M$ block of pixels to produce two RGB endpoints defining a palette, and $N \times M$ per-pixel palette indices as described in Section 2.3. We define an error threshold $\mathcal{E}$ that determines the amount of addi-

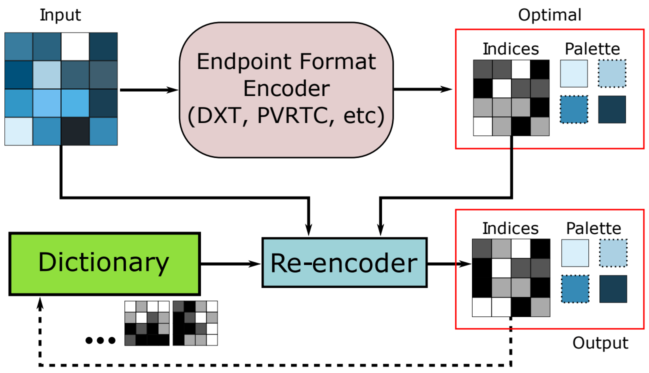

Figure 3: *The first stage of our encoding pipeline. We process each block in raster-scan order while maintaining a dictionary of recently added index blocks. For each index block, if we find an existing index block in the dictionary that closely matches the original, we reuse that index block. If significant error is introduced, then we add this index block to the dictionary.*

tional mean-squared error that can be introduced for a given block. Comparing against the original compressed representation, we proceed by searching for recently added index blocks to the dictionary. If no such dictionary entry is found, then we add the index block corresponding to the original indices for that block to the dictionary. The amount of overall error introduced in the compressed representation is directly related to the choice of $\mathcal{E}$. This error threshold is a simple way to affect the rate versus distortion properties of our compression method. By choosing a higher value for $\mathcal{E}$, we get more redundancy in our dictionary, as more recently used index blocks become acceptable, but introduce additional error resulting in more noticeable compression artifacts.

Similarly to VQ, we replace each index block in the texture with a dictionary entry. In order to decrease the number of bits required to store the entries, we only consider the last k dictionary entries. This allows us to represent a given block's dictionary entry as a delta in the range $[-k, k]$ from the previous block's entry. The entries can then be reconstructed performing a prefix-sum. By increasing the size of k, we have a smaller dictionary to store but suffer from the increased size of each dictionary entry. Although we process blocks in raster-scan order, different images may provide better delta compression using different orderings, such as a Z-curve. In our experiments, the compression performance of each ordering is highly dependent on the texture, and can be specified in a small per-file header. In our implementation, however, we choose raster-scan order and $k = 127$ in order to represent each entry delta using one byte.

In order to determine the amount of error introduced by deviating from the optimal index block, we use an optimization technique for the endpoints found in many existing endpoint texture encoders [Fenney 2003; Brown 2006; Castaño 2007; Krajcevski et al. 2013]. For endpoint-based texture compression each reconstructed pixel comes from a palette generated by two endpoints $\mathbf{p}_A$ and $\mathbf{p}_B$. The number of palette entries $\mathbf{p}_i$ is determined by the number of bits b allotted to each pixel index in an index block,

$$\mathbf{p}_i = \frac{(2^b - 1 - i)\mathbf{p}_A + i\mathbf{p}_B}{2^b - 1},$$

with $i \in [0, 2^b - 1]$. Using this formulation, for a given index block we can construct a $NM \times 2$ matrix $\mathbf{B}$ such that the optimal endpoints $\mathbf{p}_A$ and $\mathbf{p}_B$ are found by minimizing the least-squares error of the equation

$$\left\| \mathbf{B} \begin{bmatrix} \mathbf{p}_A \\ \mathbf{p}_B \end{bmatrix} - [\mathbf{p}_x] \right\|,$$

where each $\mathbf{p}_x$ corresponds to the RGB value of the x-th pixel in the original block.

3.2 Endpoint Processing

The second stage of our compression pipeline handles the endpoints themselves. Once the index block dictionary is generated, each block in the texture contains two RGB endpoints that define the palette for that block. Similarly to the PVRTC algorithm, we consider these endpoints independently as two separate low-resolution images that approximate the final image [Fenney 2003]. Each of these images can be treated independently as a separate image using traditional compression techniques.

Our endpoint encoding step processes the images in two steps prior to entropy encoding, similar to JPEG2000 [Skodras et al. 2001]. The first step is a decorrelation step in order to improve the redundancy of neighboring values and to collect the visual information into a single channel. We chose the lossless YCoCg transform in order to avoid additional loss in the final texture and for its simplicity of implementation [Malvar et al. 2008]. The lossless property of this color transform is important because any additional error is magnified by the block dimensions in the final reconstructed image. The second step applies a wavelet transform to each color plane after the YCoCg transform. This step alters the total distribution of values in order to skew their probability distribution in preparation for entropy encoding. In our experiments, the choice of wavelet basis does not significantly affect the resulting compression size. However, in order to preserve lossless compression of the endpoints, we use the CDF 5/3 wavelet as in JPEG2000 [Cohen et al. 1992].

3.3 ANS Entropy Encoding

The final stage of our compression pipeline combines the output of the two previous stages into a single data stream. Each previous stage produces two symbol streams with different probability distributions requiring a separate context model for each. The index block dictionary and entries comprise the two streams from the first stage, and the second two are the separate Y and CoCg streams for the combined endpoints (Figure 4). Each of the four streams are entropy encoded separately and the results are concatenated and saved on disk along with the associated probability distributions. In the rest of this subsection we describe the entropy encoding technique used, known as Asymmetric Numeral Systems (ANS), first introduced by Duda [2013].

3.3.1 Background

Entropy encoding is a general term used for any method that converts a sequence of values, or *symbols*, chosen from an *alphabet*, into a sequence of bits based solely on the probability of each value appearing in the input stream. The earliest such method, known as Huffman coding, directly assigns a pattern of bits to each possible input symbol [Huffman 1952]. The length of each bit pattern corresponds to the probability of that symbol appearing in the input stream. Compression occurs when the probability of a few symbols is far larger than the probability of others.

In Huffman encoding, since each symbol is represented by $b \in \mathbb{Z}$ bits, the corresponding probability of seeing the symbol in the input stream becomes $\frac{1}{2^b}$. As a result, we are not able to represent non-power-of-two probability distributions (or *models*) of symbols used with the input sequence. To rectify this limitation, a technique known as arithmetic coding takes a different approach [Rissanen and Langdon 1979]. Both encoder and decoder take as input an alphabet of symbols $\mathcal{A} = \{0, ..., n-1\}$ with probabilities $p_0, ..., p_{n-1}$ such that $1 = \sum_{s=0}^{n-1} p_s$. The encoder maintains two values, or *states*, L and H, that describe the range of numbers that encode all previously seen symbols. Initializing $L = 0$ and $H = 1$, for each symbol s received as input, the encoder alters the states by the following formula:

$$L_{new} = L + (H - L) \sum_{i=0}^{s-1} p_i$$
$$H_{new} = L_{new} + p_s (H - L)$$

The final result written to disk can be any number in the range $[L, H)$. This number uniquely determines the input sequence for the given probability distribution of symbols in $\mathcal{A}$. Compression occurs when the numbers L and H are relatively far apart, and we can choose a number that requires few bits to represent within that range. In particular, we can see that for a sequence of symbols $s_0, s_1, ...$, the range $H - L$ gets smaller at the rate of $p_{s_0} p_{s_1} ...$. This implies that the number of bits needed to represent a number in this range grows at the rate $O(\log \frac{1}{p_i})$, which matches the optimal theoretical limit established by Shannon [1948]. By knowing the final result, the decoder can follow the same procedure as the encoder and stop upon reaching the end of the bit stream.

3.3.2 Asymmetric Numeral Systems

ANS is similar to arithmetic encoding in that it approximates the theoretical limit to compression size, but has certain properties that make it amenable for implementation on SIMD architectures. The input to the compression algorithm is an alphabet $\mathcal{A} = \{0, ..., n-1\}$, a stream of input symbols $s \in \mathcal{A}$, and a probability distribution $\{p_s\}$, $\sum_s p_s = 1$. Commonly, this probability distribution is discretized with the approximation $F : \mathcal{A} \rightarrow \mathbb{N}$ such that $F(s)/M \approx p_s$, where $M = \sum_s F(s)$. We use the common approach of using the notation F_s to represent $F_s = F(s)$. Given a symbol s and a *state* $x \in \mathbb{N}$ that encodes all of the previously seen symbols of a given stream, ANS provides an encoder C and a decoder D such that

$$C(s, x) = x' \qquad = M \left\lfloor \frac{x}{F_s} \right\rfloor + B_s + (x \bmod F_s)$$
$$D(x') = (s, x) \qquad = \left(\mathbf{L}(R), F_s \left\lfloor \frac{x'}{M} \right\rfloor + R - B_s \right),$$

where

$$R = x' \bmod M \qquad \text{and} \qquad B_s = \sum_{i=0}^{s-1} F_i.$$

The function $\mathbf{L}$ is a lookup function that determines the symbol s such that

$$\mathbf{L}(z) = \max_{B_s < z} s.$$

It follows that C and D are direct inverses of each other, a property that arithmetic coding does not satisfy. Furthermore, C and D are monotonically increasing and decreasing with respect to the state x, respectively. The more interesting property is that our state grows at a rate similar to that of arithmetic encoding, requiring $O(\log \frac{M}{F_s}) \approx O(\log \frac{1}{p_s})$ bits per symbol.

In order to stream data into and out of bits, as is required for use with a physical machine, each intermediate state x must be *normalized*. In other words, we must write data to disk and decrease x in order to prevent it from growing out of physical memory bounds, typically a 32-bit register. Duda [2013] claims that this is possible by defining a normalized interval $[L, bL)$ such that $L = kM$ for some $k, b \in \mathbb{N}$. In this interval, b represents the divisor required to normalize the interval. In the encoder, C, whenever x grows larger than bL, then x is repeatedly normalized by x/b until $x \in [L, bL)$. For each required normalization, the compressor C first writes the corresponding $x \pmod b$ to disk. Common choices for b are powers of two in order to map nicely to integer shift and bitwise masking operations during encoding and decoding. Just as the encoder may exceed the normalized interval from above, a decoder may require normalization when $x < L$. In this case, the decoder will read a value of size b from disk and increase x by $x' = bx$ until x is in the normalized interval. In order to maintain reciprocity with the encoder, a decoder is required to read from disk at the same point in the data stream that the corresponding encoder wrote to disk. However, it is not required that the data for the data stream remain contiguous in memory, which is a separate property from arithmetic coding.

3.3.3 Preparing ANS for Parallel Decoding

As with all entropy encoding, the variability of the length of the resulting data stream makes it difficult for decoders to start working in parallel, or for a single decoder to be parallelized. In general, parallel decoding approaches have required additional metadata to to the start positions in the datastream, increasing the overhead of the compressed data. ANS provides new properties for being amenable to GPU decoding. The biggest advantage that ANS gives is that C and D are inverses of each other, as defined in Section 3.3.2, and always read or write a value of size b to disk. As Giesen [2014] shows, if we have N compressors C_i working in parallel, then all N can operate in lockstep and share a common data stream as long as they write to disk in a deterministic ordering. To maintain the reciprocity between the corresponding set of N decoders D_i, we must make sure that all decoders read from the shared stream in the same order that the encoders wrote to. This will ensure that each of the inputs to a given decoder D_i will be exactly the output from the corresponding encoder C_i.

Furthermore, if we choose b such that $b \geq M$, then we know that each encoding step will at most write one value in the range $[0, b)$ to disk during state normalization. This property implies that each encoder will write at most one value $x \bmod b$ to disk per encoded symbol and hence each decoding thread (e.g. on a GPU) will read at most one such value per decoded symbol. Each state normalization therefore requires the evaluation of a conditional rather than a loop. A lock-step SIMD implementation such as those found in GPUs will always require each decoder to read at the same machine instruction, and by masking out the threads that must read from disk, we can maintain the order of reads and decode in parallel.

Using a value of $b = 2^n$ we ensure an integer number of bits being written and read from disk. Once all encoders C_i finish writing to the shared stream, N final encoding states x_i will be used to seed the decoders during decompression. We discuss these trade-offs in more detail in Section 4. To map well to current GPU architectures, the rest of this paper will assume the settings $k = 2^2$, $b = 2^{16}$, $M = 2^{11}$, and $|\mathcal{A}| = 2^8$.

4 Parallel Decoding

The design outlined in Section 3 facilitates the decoding of textures on SIMD architectures. We outline the overall structure of a decompressor in Figure 4. A key feature is that each step in the decoding process is very well suited for implementation on a SIMD processor. We have structured our encoders and decoders for use with

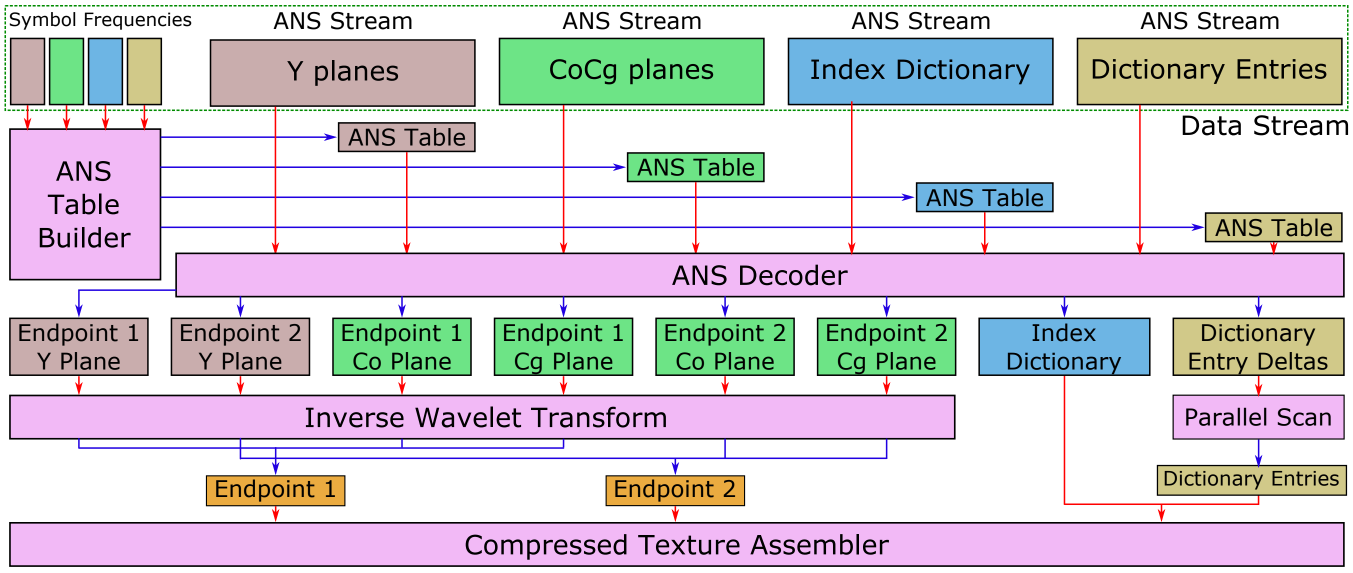

Figure 4: *Our decompression pipeline. Pink boxes represent separate GPGPU executions for which red arrows are inputs and the blue arrows are the outputs. Per our design, the supercompressed texture data can be uploaded directly to the GPU for decoding. The input data and intermediate results all remain resident in GPU memory during decoding. Multiple texture streams can be interleaved between the symbol frequencies and the ANS streams to provide additional decoding parallelism.*

commodity GPUs, but our algorithm can be appropriated to any SIMD architecture. In practice, the main trade-off of our method is between decompression speed and compression size. Our compressed representation contains a small header in order to properly construct the data pointers needed to begin the decoding process on the GPU. For a full implementation of both CPU encoder with GPU decoder, please refer to the source code included as supplementary material.

ANS is able to take advantage of SIMD hardware by interleaving many compression streams and decoding them in parallel. One of the biggest constraints is the number of streams that can be interleaved at a time. In order to determine the order in which the interleaved compressors read from the shared compression stream, each decoder thread must notify all the others that it is reading in order to advance their shared offset [Giesen 2014]. Although arbitrarily many encoders can be interleaved, we suggest using 32 or 64 in order to use the available 32-bit or 64-bit shared registers that map well to the warp size of certain GPU vendors.

Additionally, the number of symbols encoded per thread has significant implications on the decoding performance and compression size. First, each decoding thread must be initialized with the ANS state of the corresponding encoding stream. The fewer symbols encoded per thread will increase the total number of threads and hence will also increase the storage overhead of the encoder states. However, decreasing the total number of encoded symbols per thread increases overall parallelism by giving the GPU additional opportunity for scheduling work while waiting on reads and writes to global memory. Finally, the total number of symbols encoded per thread limits the resolution of the final texture. In our method, we use a fixed number of symbols per encoding stream in order to keep all threads busy during decoding. Due to each endpoint belonging to a block of pixels, the number of symbols per set of encoders must divide the total number of pixel blocks in the texture. In our approach, we choose to use 256 symbols per thread requiring the total number of pixel blocks to be a multiple of $256 \times 32 = 8192$. The ramifications of these trade-offs are shown in Figure 5.

Each ANS decoder relies on a context model given by the frequencies F_s described in Section 3.3. The decoder needs to know the values of F_s and B_s along with a fast implementation of $\mathbf{L}(z)$. Hence, each ANS encoded stream contains an additional set of frequencies F_s on disk. Because we know $z \in [0, M)$, we can construct a table of size M containing triplets (s, F_s, B_s) for every possible z. This table can be constructed from the set of F_s as a parallel prefix-sum to construct the B_s and a parallel binary search to find s for each value of z. Constructing this table is the first step of our parallel decoding process as outlined in Figure 4. Using this table across all decoders requires us to use static histograms as our probability distribution. Adaptive models are difficult to implement because of race conditions while updating the model from different threads.

5 Implementation

Many of the limits of SIMD architectures require careful consideration of implementation details. Our method mainly focuses on further encoding *endpoint* compression formats as described in Section 2.3. We present an investigation of the compression pipeline presented in Section 3 with respect to the DXT and PVRTC compression formats. We have chosen these formats due to their simplicity and widespread usage [Iourcha et al. 1999][Fenney 2003]. PVRTC and DXT differ only in the amount of bits allotted to store the endpoints and the manner in which their corresponding hardware reconstructs the compressed block. The compression quality of the texture is largely determined by the target hardware compressed format, although DXT and PVRTC usually provide similar quality encodings.

5.1 DXT

DXT (a.k.a. S3TC) has a number of variations in order to deal with textures containing alpha, single-channel, and two-channel textures. Here we will address the most common variation, DXT1, and note that additional variations involve adding or removing a pair of channels (DXT3/4) and possibly a separate re-encoding of additional index blocks (DXT5) [Iourcha et al. 1999].

DXT1 has two palette generation modes depending on which order the RGB endpoints are placed. If the 16-bit integer representation of the first endpoint yields a smaller value than the second endpoint, then only three palette colors are generated and the fourth corresponds to a solid black or transparent pixel. The optimal endpoint values described in Section 3.1 for a given DXT index block may be generated in either order. In the case in which these endpoints do not generate the expected four-color palette, we discard the index block as invalid.

During the color transform step as described in Section 3.2, we attempt to maintain the low bit depth of the pixel channels. Maintaining a low bit depth allows us to limit the number of symbols needed for entropy encoding following the wavelet transform. In the case of DXT1, endpoints are stored using five, six, and five bits for the red, green, and blue channels, respectively. Each 565 RGB value can be losslessly converted to 667 YCoCg [Malvar et al. 2008]. After performing the wavelet transform on each channel of the YCoCg data separately, we need less than eight bits to represent the coefficients. In order to increase parallelism, we use 32×32 blocks of endpoints. As this wavelet transform is operating on endpoints per 4×4 pixel blocks, this implies that we currently limit the dimensions of each texture to be a multiple of 128. This block size was chosen to map well to the common limit of 256 threads per work-group on modern AMD GPUs.

5.2 PVRTC

PVRTC is similar to DXT in that it has the ability to choose between opaque and transparent textures in the compressed block representation. However, in the original PVRTC, the opacity of the color is determined on a per-endpoint basis rather than on a per-block basis. As a result, each color contains an extra bit to determine whether or not the color contains opaque RGB values or transparent RGBA values. A further bit is provided to alter the generated palette to provide a similar punch-through alpha as in DXT1. In contrast to DXT, these features of PVRTC are agnostic to the ordering of the endpoints, so we can choose opaque endpoints every time to match the generated endpoints from the re-encoding described in Section 3.1 [Fenney 2003].

Additionally, PVRTC uses lower-precision endpoints than DXT1. Where DXT1 stores three-channel endpoints with 565 precision, PVRTC stores endpoints using 555 and 554 precision. With 565 endpoints, the additional bit in the green channel requires additional bits in the resulting YCoCg transform. However, using 555 endpoints restricts the additional bits needed for YCoCg to an additional bit in the Co and Cg channels, meaning these endpoints can be represented using 566 YCoCg, i.e. two fewer bits per endpoint than DXT1.

6 Results

In order to compare our data against prior state of the art methods, we have restricted our testing to DXT1 based textures. We have used Barett's port of Giesen's DXT encoder as our ground truth for optimal DXT encoding due to its overall quality of compressed output and encoding speed [Barett and Giesen 2009]. We measure against raw images, stored as BMP, standard JPEG compression, raw DXT1 compression, and the Crunch library [Geldreich 2012]. For any given set of images, our method produces similar quality results as leading DXT1 compressors, as shown in the detailed analysis of Figure 9. Additional close-up comparisons can be found in the supplementary material.

Format	CPU Load	CPU Decode	GPU Load	GPU Decode	Total
JPG	0.1	51.9	2.8	0	54.8
DXT	3.0	0	0.4	0	3.4
BMP	116.3	0	2.2	0	118.5
CRN	0.4	7.7	0.4	0	8.5
GST	0.5	0	0.3	2.5	3.3

Table 1: *Comparison of various timings in milliseconds for different compression schemes. We test our method against various formats rendering a set of frames from a* 360° *video at 4K resolution* (3584×1792) *similar to motion JPEG video [1992].*

For fair comparison, we have chosen the maximum quality settings for Crunch and have chosen settings for our method to be $\mathcal{E} = 30$, as described in Section 3.1. For these settings, on a typical 512×512 texture, our method takes about 0.76s to compress compared to the 6.32s for Crunch. However, as can be seen in Figure 7, our compression method has slightly larger variability than Crunch in terms of optimizing for rate-distortion. One of the main sources of this variability is the discrepancy in compressed index data, which is the least amenable to entropy encoding due to its incoherency (Figure 2). The only parameter that we can use to control the rate-distortion properties is the error threshold $\mathcal{E}$ that is fairly coarse grained, as shown in Figure 6. The global dictionary of Crunch also gives it an advantage on hard-to-compress images since neighboring redundancies are generally hard to find. To our benefit, however, using a truncated dictionary does improve the compression size for many textures in the Kodak data set, as shown in Figure 8. We present a breakdown of the size of each of the parts of a GST texture in Figure 10. This benefit arises due to the assumption that neighboring blocks produce similar palette indices during compression and as a result our method is dependent on the details of the image, similar to JPEG.

We use two main benchmarks for testing the performance benefits of our implementation. The first benchmark measures the average load time for all 600 4K resolution frames in a 360° video using a motion JPEG application similar to Pohl et al. [2014]. The second benchmark measures the time required to load all 128 Pixar textures, each with dimensions 512×512, into GPU memory on a single CPU thread [Pixar 2015]. One of the main benefits of our method is the reduction in load times. We observe this benefit in both the batch load times for the Pixar dataset described in Table 2 and our 360° video benchmark in Figure 1. All results are measured on desktop PC running Windows 7 on an Intel Xeon 8 core CPU and AMD R9 Fury GPU. As demonstrated in these results, the raw DXT1 load times are significantly faster than the supercompressed textures. Disk seek times and actual disk reads provide a significant amount of inconsistency in disk load timings. In our measurements we make sure that each of the files are fully loaded in the operating system's page cache prior to doing any measurements in order to improve consistency. We observed significantly longer load times of high resolution DXT and BMP frames due to the full saturation of this cache. Our method, on the other hand, is particularly well-suited to high resolution textures due to the increased parallelism offered in the GPU decoder.

We also investigate some of the tradeoffs presented in Section 4. In particular, we show the impact of varying the number of symbols encoded per thread for a single large resolution texture. The performance implications are twofold. Fewer symbols per thread leads to increased parallelism from having more decoders in flight. On the other hand, more symbols per thread reduces the amount of on-disk overhead per group of interleaved decoders. The speed of using more or fewer symbols per thread also depends on whether or not we copy the ANS decoding table into local memory for each group of decoders (Figure 4). These tradeoffs are shown in Figure 5.

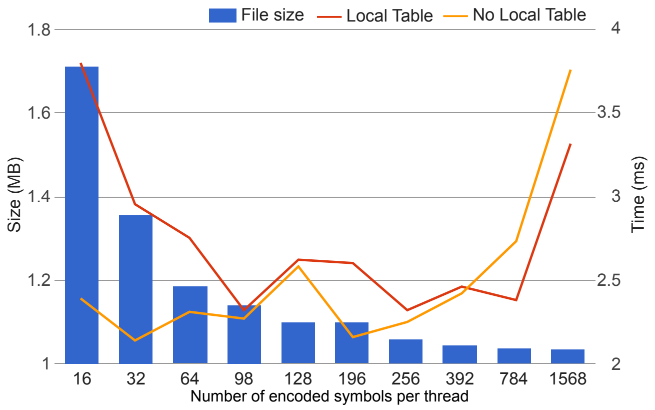

Figure 5: *The affect of varying the number of symbols per decoding thread, and hence number of parallel decoders, as a comparison between average file size and decoding time of 600 frames of a 4K 360° video. When decoding few symbols per thread, size is dominated by storing many encoder states, although the increased parallelism helps decoding speed. Copying the ANS decoding table (Section 4) into local memory only benefits decoding speed when there is enough work per thread to benefit from fewer global reads.*

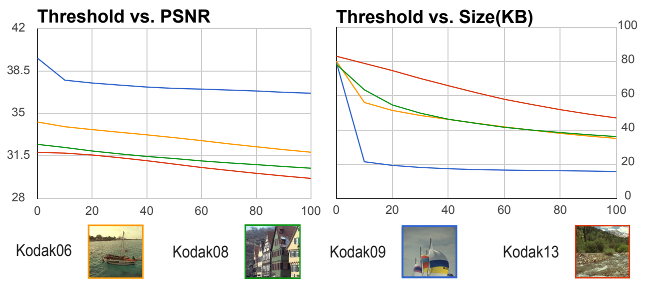

Figure 6: *The difference in PSNR and compression size as a function of our error threshold for a few images from the Kodak test suite. As we increase our error threshold, we see a decrease in the size of our index data and a drop in our PSNR. Both of these metrics are sensitive based on the features of the encoded image. The PSNR stabilization after an increase in the error threshold supports the assumption of block-level coherency between indices.*

7 Discussion

Based on the results in Section 6, we observe significant benefits from using a GPU-based decoding algorithm in the general case. In particular, very large textures are the most susceptible to the serial decoding requirements of traditional entropy encoders. In other applications, a multicore machine may parallelize the decoding of multiple low-resolution textures, which may be beneficial in terms of reducing the overhead of interfacing with the GPU. We observed that a set of per-core Crunch decoders processed all 128 textures of the Pixar dataset at similar load times to our method.

Limitations: Although our method presents many advantages, there are a few limitations in practice. First, our current implementation requires additional scratch memory for intermediate results during decompression. Although the additional memory is minimal, about 3X the size of the final texture, it may be a limiting factor for streaming texture applications that try to exhaust the available GPU resources. However, this limitation may be overcome by using the final compressed texture as scratch memory, but this reduces efficiency by requiring unaligned memory reads and writes along with additional synchronization requirements. This sort of 'in-place' decoding presents additional performance concerns by retrieving the values for individual channels in each of the endpoints as described in Section 5.

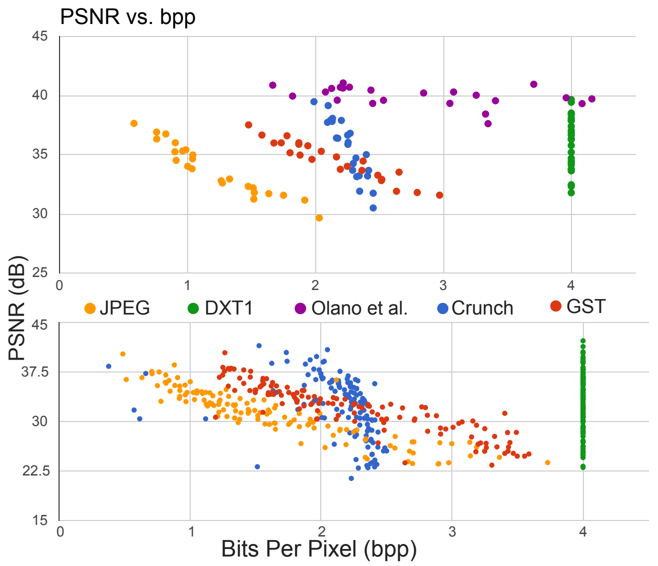

Figure 7: *We show PSNR versus bitrate values for our method against other compression schemes. The data shows that our method provides bit rates and quality comparable to the state of the art supercompressed textures. Each data point is an image in the (top) Kodak [1999] and (bottom) Pixar [2015] datasets with dimensions* 512×512.

Format	JPEG	PNG	DXT1	Crunch	GST
Time (ms)	848.6	1190.2	85.8	242.3	93.4
Disk size (MB)	6.46	58.7	16.8	8.50	8.91
CPU size (MB)	100	100	16.8	16.8	8.91

Table 2: *Quantitative results of single-threaded loading of the 128 textures in the Pixar dataset [2015]. The CPU size represents the size of all textures in memory after any decoding procedure and prior to uploading to the GPU. The disk bandwidth is sufficiently fast to make decoding textures the bottleneck.*

Additionally, the results in Section 6 are presented using our reference implementation written in OpenCL for portability. However, during our experiments, we noticed significant stalls on the GPU that were unaccounted for. Our implementation would benefit from additional fine-grained control over the GPU programming interface, such as those presented in the Vulkan API, and further optimization could go a long way to realizing additional performance gains in our application.

Future Work: Although our implementation focuses on PVRTC and DXT1, more recent endpoint methods such as BPTC and ASTC have introduced increased complexity in the compressed representation of textures [OpenGL 2010; Nystad et al. 2012]. They allow multiple palettes per block and variable bit depths for their palette indices. Additionally, ASTC provides variable index block dimensions that presents increased complexity to our re-encoding scheme. Although our method works with simple endpoint compressed formats, extensions to more complicated formats that preserve the additional quality may be possible.

HDR compression formats present another natural extension of our

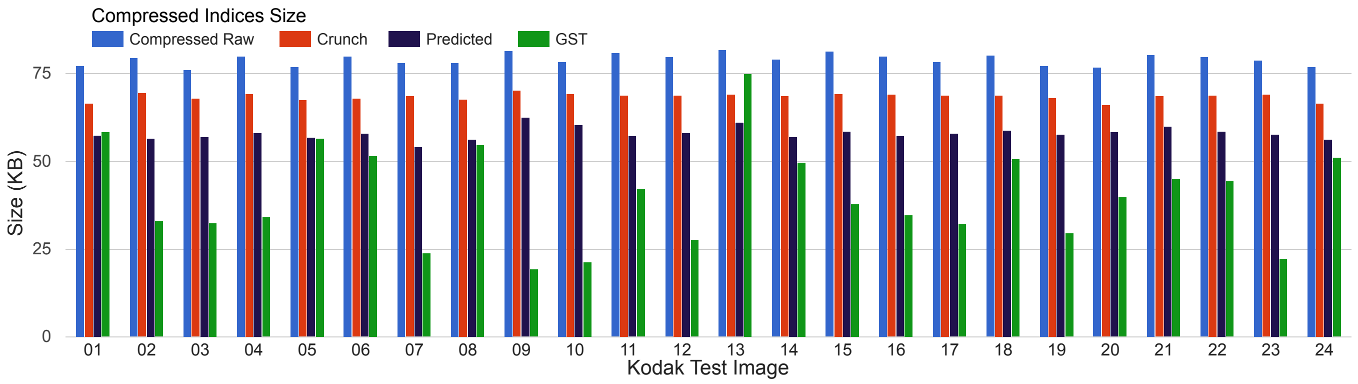

Figure 8: *We demonstrate a comparison of the compressibility of various approaches to preprocessing index data. This graph demonstrates the size of the compressed index data for various algorithms against the Kodak data set [1999] using the same Huffman encoder used in the Crunch textures. Palette indices are classically the most incoherent data in compressed textures. We show that by limiting the dictionary lookup to the k most recently added dictionary entries, we increase the entropy encoding capabilities of the index data significantly over Crunch at maximum quality settings. Compressed raw is Huffman encoding applied to the unprocessed index data while the predicted method is the same method used by Ström and Wennersten [2011] applied to DXT textures.*

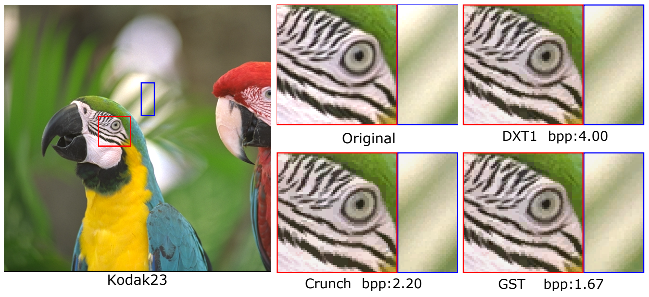

Figure 9: *A zoomed-in view of the visual quality of various compressed formats. The only stage in our compression pipeline that may introduce additional error is the re-encoding stage. Here we show that the amount of error introduced is imperceptible with respect to other DXT compression formats.*

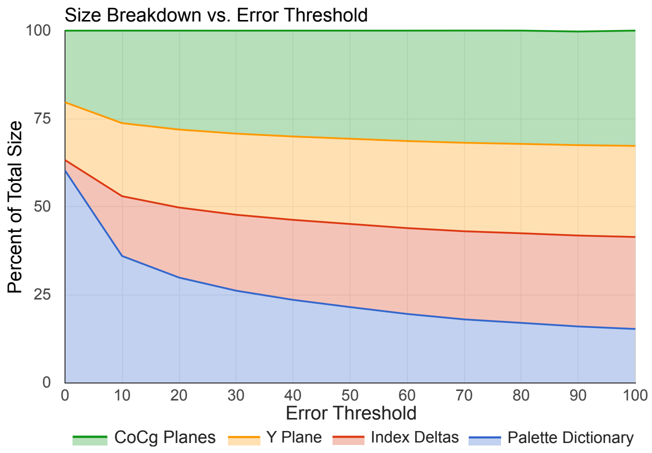

Figure 10: *An average of the percentage-wise breakdown of each of the constituent parts of a GST encoded texture using various error thresholds. As we allow more error, the size of the dictionary decreases as a percentage of overall space consumed. We used both the Pixar and the Kodak data sets.*

work. We also believe that the tabled compression formats such as ETC1 and ETC2 can be tackled using a variation of our algorithm. We do not expect our algorithm to emit compression rates as low as those presented by Ström and Wennersten [2011]. However, interpreting the parameters of tabled compressed textures as separate low-resolution images may significantly reduce the overhead of this class of compressed textures.

Although our benchmark uses 360° video, our method is inherently designed for single-texture representations. A method similar to Olano et al [2011] is possible where an entire mip-map chain can be used to encode each of the individual endpoint images rather than an explicit wavelet transform. Additionally, the endpoint images need not be compressed independently, as they have a lot of coherence that can be exploited to see additional gains in compression performance.

Conclusions: We have presented a new algorithm for storing compressed textures on disk. The benefits of our algorithm show a significant improvement in decoding speed over state of the art CPU techniques. Furthermore, our algorithm provides a way to upload texture data directly to the GPU for decoding in order to maintain the CPU-GPU benefits. In particular, we believe that our method is best suited for streaming high-resolution textures from disk and network. In certain image-heavy applications, such as Google Street View, that require fast access to remote image data, and then manipulate it using the GPU, we believe our method will provide the most benefit. Additionally, as more web page renderers move to the GPU, we believe that the improved decoding speed will be a boon for quickly introducing image data to mobile devices where decoding speed and responsiveness become increasingly important. As GPUs become more widespread in general, effective streaming solutions present a growing need.

8 Acknowledgements

This research is supported in part by ARO Contract W911NF-14-1-0437 and Google.

References

BARETT, S., AND GIESEN, F., 2009. DXTC encoder. https://github.com/nothings/stb/blob/master/stb_dxt.h.

BEERS, A. C., AGRAWALA, M., AND CHADDHA, N. 1996. Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, SIGGRAPH '96, 373–378.

BROWN, S., 2006. libsquish. http://code.google.com/p/libsquish/.

BUCCIGROSSI, R. W., AND SIMONCELLI, E. P. 1999. Image compression via joint statistical characterization in the wavelet domain. *IEEE Transactions on Image Processing 8*, 12 (Dec), 1688–1701.

CASTAÑO, I. 2007. High Quality DXT Compression using CUDA. *NVIDIA Developer Network*.

COHEN, A., DAUBECHIES, I., AND FEAUVEAU, J.-C. 1992. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics 45*, 5, 485–560.

DELP, E., AND MITCHELL, O. 1979. Image compression using block truncation coding. *Communications, IEEE Transactions on 27*, 9 (sep), 1335–1342.

DUDA, J. 2013. Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR abs/1311.2540*.

FENNEY, S. 2003. Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, HWWS '03, 84–91.

GELDREICH, R., 2012. Crunch. http://code.google.com/p/crunch/.

GIESEN, F. 2014. Interleaved entropy coders. *CoRR abs/1402.3392*.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (Sept), 1098–1101.

INADA, T., AND MCCOOL, M. D. 2006. Compressed lossless texture representation and caching. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '06, 111–120.

IOURCHA, K. I., NAYAK, K. S., AND HONG, Z., 1999. System and method for fixed-rate block-based image compression with inferred pixel values. U. S. Patent 5956431.

KNITTEL, G., SCHILLING, A., KUGLER, A., AND STRAßER, W. 1996. Hardware for superior texture performance. *Computers & Graphics 20*, 4, 475–481.

KODAK, 1999. Kodak lossless true color image suite. available at http://r0k.us/graphics/kodak.

KRAJCEVSKI, P., LAKE, A., AND MANOCHA, D. 2013. FasTC: accelerated fixed-rate texture encoding. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, I3D '13, 137–144.

KRAJCEVSKI, P., GOLAS, A., RAMANI, K., SHEBANOW, M., AND MANOCHA, D. 2016. VBTC: GPU-friendly variable block size texture encoding. *Computer Graphics Forum 35*, 2, 409–418.

LESKELA, J., NIKULA, J., AND SALMELA, M. 2009. Opencl embedded profile prototype in mobile device. In *2009 IEEE Workshop on Signal Processing Systems*, 279–284.

MALVAR, H. S., SULLIVAN, G. J., AND SRINIVASAN, S. 2008. Lifting-based reversible color transformations for image compression. In *SPIE Applications of Digital Image Processing*, International Society for Optical Engineering.

NYSTAD, J., LASSEN, A., POMIANOWSKI, A., ELLIS, S., AND OLSON, T. 2012. Adaptive scalable texture compression. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on High Performance Graphics*, Eurographics Association, HPG '12, 105–114.

OLANO, M., BAKER, D., GRIFFIN, W., AND BARCZAK, J. 2011. Variable bit rate GPU texture decompression. In *Proceedings of the Twenty-second Eurographics Conference on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR '11, 1299–1308.

OPENGL, A. R. B., 2010. ARB_texture_compression_bptc. http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt.

PIXAR, 2015. Pixar one twenty eight. https://community.renderman.pixar.com/article/114/library-pixar-one-twenty-eight.html.

POHL, D., NICKELS, S., NALLA, R., AND GRAU, O. 2014. High quality, low latency in-home streaming of multimedia applications for mobile devices. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, 687–694.

RISSANEN, J., AND LANGDON, G. G. 1979. Arithmetic coding. *IBM J. Res. Dev. 23*, 2 (Mar.), 149–162.

SHANNON, C. E. 1948. A mathematical theory of communication. *The Bell System Technical Journal 27* (July, October), 379–423, 623–656.

SKODRAS, A., CHRISTOPOULOS, C., AND EBRAHIMI, T. 2001. The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine 18*, 5 (Sep), 36–58.

STRÖM, J., AND AKENINE-MÖLLER, T. 2004. PACKMAN: texture compression for mobile phones. In *ACM SIGGRAPH 2004 Sketches*, ACM, SIGGRAPH '04, 66–.

STRÖM, J., AND AKENINE-MÖLLER, T. 2005. iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, HWWS '05, 63–70.

STRÖM, J., AND PETTERSSON, M. 2007. ETC2: texture compression using invalid combinations. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, GH '07, 49–54.

STRÖM, J., AND WENNERSTEN, P. 2011. Lossless compression of already compressed textures. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, HPG '11, 177–182.

TORBORG, J., AND KAJIYA, J. T. 1996. Talisman: Commodity realtime 3d graphics for the PC. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '96, 353–363.

WALLACE, G. K. 1992. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics 38*, 1 (Feb), xviii–xxxiv.

WAVEREN, J. M. P. V. 2006. Real-time texture streaming and decompression. *Id Software Technical Report*.