FasTC: Accelerated Fixed-Rate Texture Encoding

Pavel Krajcevski*
University of North Carolina at Chapel Hill

Adam Lake†
Intel Corporation

Dinesh Manocha‡
University of North Carolina at Chapel Hill

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

Abstract

We present a new algorithm for encoding low dynamic range images into fixed-rate texture compression formats. Our approach provides orders of magnitude improvements in speed over existing publicly-available compressors, while generating high quality results. The algorithm is applicable to any fixed-rate texture encoding scheme based on Block Truncation Coding and we use it to compress images into the OpenGL BPTC format. The underlying technique uses an axis-aligned bounding box to estimate the proper partitioning of a texel block and performs a generalized cluster fit to compute the endpoint approximation. This approximation can be further refined using simulated annealing. The algorithm is inherently parallel and scales with the number of processor cores. We highlight its performance on low-frequency game textures and the high frequency Kodak Test Image Suite.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations I.3.4 [Computer Graphics]: Graphics Utilities—Application Packages;

Keywords: Texture compression, Content pipeline, GPU hardware

1 Introduction

In modern day real-time graphics applications, textures are heavily used to increase the fidelity of synthesized images. To achieve this fidelity, a large amount of video memory must be devoted to storing these textures. Furthermore, accessing texture memory at runtime comes at a high performance cost. The need for ever increasing realism in today's games drives a constant requirement for more and higher resolution art assets. As a result, rendering from a compressed representation of the image has become a key technique for both reducing memory footprint and increasing texture access bandwidth [Beers et al. 1996] [Fenney 2003].

One key aspect of using compressed textures for rendering is the notion that fast encoding speed is useful but not necessary [Beers et al. 1996] [Fenney 2003]. However, modern-day games use hundreds, if not thousands, of textures for everything from lightmaps to character albedo maps. Additionally, the game development process relies heavily on constant iteration over all included assets. These requirements make development of fast, high-quality texture compression methods important.

The state of the art in texture compression formats apply a variety of techniques to compress data. In order to provide random-access pixel lookup and cache coherency, each format encodes a fixed size block of texels separately. The size of this block combined with the size of its encoding determines the compression ratio of the format. For low dynamic range images, the texels in each block are treated as lattice points in three or four dimensional color space [Iourcha et al. 1999] [OpenGL 2010] [Nystad et al. 2012]. For example, sixteen points on a 256^3 lattice represent a 4×4 texel block of RGB data at eight bits per channel. These points are encoded by a line segment whose endpoints reside on a much sparser lattice and are recovered using linear interpolation. Furthermore, recent encoding formats allow data points to be partitioned into separate subsets that each have their own interpolating line segment. Additionally, methods for high dynamic range images (HDR) have also been proposed [Munkberg et al. 2008] [Roimela et al. 2006].

Figure 1: *The Bootcamp demo from Unity3D[1] using uncompressed textures (top) and using textures compressed with FasTC-64 (bottom). The visual quality of the scene is only slightly altered and no visible artifacts appear. The scene uses 156 textures which were compressed in a total of 8.75 minutes by our method. The same textures are compressed by the BC7 Compressor in the NVIDIA Texture Tools in a total of 13.27 hours.*

Encoding for texture compression formats is usually performed offline. Algorithms for encoding textures vary on the spectrum of quality versus performance. Some publicly available and widely used compressors for older formats use cluster analysis techniques to achieve good compression quality [Brown 2006] [Donovan 2010]. Others attempt to compress the image even further using a lossless codec in addition to the one supported by hardware [Geldreich 2012] [Strom and Wennersten 2011]. However, the simplicity of these formats restricts the quality of the compressed image.

*e-mail:pavel@cs.unc.edu
†e-mail:adam.t.lake@intel.com
‡e-mail:dm@cs.unc.edu

[1] Unity3D engine vailable at *http://www.unity3d.com/*. Bootcamp demo available through the Unity Asset Store.

Newer formats use the same kind of analysis to achieve the quality available by the format but result in long compression times for the best quality [Nystad et al. 2012]. It is not uncommon for current high quality texture encoders to take hours to compress hundreds of textures for a single scene (see Figure 1).

Main Results: We present a new algorithm, FasTC, for generating texture encodings that provide orders of magnitude improvements in speed over existing compression methods while maintaining comparable visual quality. Our approach is applicable to low dynamic range formats that support partitioning by using a coarse approximation scheme to estimate the best partition. Due to the quantization artifacts in the compressed data, this coarse approximation gives a substantial increase in speed while avoiding a severe penalty in quality. Next, we use a generalized *cluster-fit* [Brown 2006] to find the best encoding for the partition. The benefit of this approach is that we perform linear regression in the quantized space rather than in the continuous or even discrete RGB space. Optionally, we allow the user to refine the data using simulated annealing which provides the ability to set the speed versus quality ratio within a single algorithm. We test our algorithm on low-frequency game textures and the canonical high-frequency Kodak Test Image Suite [1999] against popular compressors provided by both NVIDIA [Donovan 2010] and Microsoft [2010]. Our method performs orders of magnitude faster than these methods while maintaining visually similar results. Furthermore, we demonstrate parallel scalability with the number of processor cores.

The rest of the paper is organized as follows: In Section 2 we review relevant texture compression formats and various methods to encode them. Section 3 gives an overview of the problem that we are trying to solve and presents the details of FasTC. Section 4 compares compression results against known methods in terms of speed and quality.

2 Related Work

The texture encoding schemes most used in today's formats follow the Block Truncation Coding (BTC) scheme introduced by Delp and Michell [1979] for compressing eight-bit grayscale images. In this format, 4×4 grayscale blocks are encoded using two eight-bit grayscale values and a bit for each texel denoting which value to use resulting in two *bits per pixel* (bpp). Various generalizations of this idea have been proposed [Nasrabadi et al. 1990] [Fränti et al. 1994].

Campbell et al [1986] extended the idea of BTC to include color by introducing a 256-value color palette instead of grayscale values. This compresses textures up to 2 bpp but requires an additional memory lookup making it inferior to formats that provide hardware-supported pixel decompression with a single texture access. Knittel et al [1996] proposed improvements to this method by adding hardware support and a texturing system, but ultimately was too slow for real-time graphics applications.

The most popular rendition of the BTC idea was introduced by Iourcha et al [1999] as S3TC/DXT1. In S3TC, 4×4 RGB blocks are encoded with two RGB565 endpoints and a two-bit index for each texel. Based on its index, each texel is reconstructed by mapping to one of four points along the line segment in RGB color space defined by the endpoints. Although many popular offline encoders exist for S3TC [Bloom 2009] [Donovan 2010] [AMD 2008] [Brown 2006], real-time compression algorithms have also been developed [Waveren 2006] [Castaño 2007]. Our approach differs by focusing on formats with partitioning and variable encoding precision. Recently algorithms have been developed that extend the efficiency of S3TC by either cleverly using the compression format [Mavridis and Papaioannou 2012] or by weighing the importance of endpoints based on the input [Krause 2010].

A popular format used in mobile devices is the PACKMAN format introduced by Ström and Akenine-Möller [2004]. In this format, 2×4 texel blocks are encoded by storing a single chrominance value and then indexing into a table of luminance offsets to produce the final color. The format has been expanded upon by introducing iPACKMAN, or ETC1, in which two PACKMAN blocks are stored per 4×4 texel block by choosing either vertical or horizontal partitioning [Ström and Akenine-Möller 2005]. ETC2 further expands upon this format by introducing new chrominance options for blocks with large variation [Ström and Pettersson 2007].

Recently, many formats that support both low dynamic range (LDR) and high dynamic range (HDR) images have emerged [OpenGL 2010] [Nystad et al. 2012]. BPTC [OpenGL 2010] compresses 8-bit per channel LDR images by encoding 4×4 texels of data down to 128 bits (8 bpp). A separate mode that also operates on 4×4 texel blocks is used to compress HDR images. ASTC [2012] also compresses both HDR and LDR blocks of texels down to 128 bits, but provides different modes that vary the size of the compressed block anywhere from 4×4 texels (8 bpp) to 12×12 (0.89 bpp). This technique was presented alongside an encoder that has various parameters that provide different tradeoffs for speed versus quality. Both BPTC- and ASTC-encoded blocks have control information that specify the partitioning, index precision, and endpoint precision of the compressed block. In ASTC, these options are specified independently on a per-block basis, while BPTC supports eight different modes that have predetermined values for each of these options.

Of the eight modes that are supported by BPTC, numbered from zero to seven, five support partitioning the block. Modes zero and two partition the block into three subsets, and modes one, three, and seven partition the block into two subsets where each partition is specified by a four or six bit partition index [OpenGL 2010]. In ASTC, partitions are specified using a ten bit partition ID, and are determined by evaluating a function that takes this ID, the number of partitions, and the texel location as arguments [Nystad et al. 2012]. In this paper, we introduce ways to improve upon the speed of compression algorithms that encode textures into these newer formats.

3 Texture Encoding

The main consideration for a texture compression algorithm is to efficiently produce a stream of bits, or encoding, that when decompressed recreates the original image as accurately as possible. In this section, we formally define the problem of compressing low-dynamic range data for popular fixed-rate texture compression formats. We split the problem into three parts: *partition selection*, *endpoint estimation*, and *endpoint refinement*. We present an algorithm that selects partitions using the real-time technique described by Waveren [2006] and uses a generalized cluster fit along with simulated annealing to determine the optimal endpoint selection [Brown 2006] [Kirkpatrick et al. 1983]. In the following sections, we assume that all textures are encoded using eight-bit RGB channels, although the precision of the input may vary.

3.1 Background

Currently, the most popular texture compression formats, including S3TC, BPTC, and ASTC, encode RGB data by defining a line segment in RGB space and assigning an index value to each texel. This index is used to interpolate between the endpoints of the line segment in order to reproduce the original color value. The formats require the compressor to accurately select values for the endpoints given a block of texels, as shown in Figure 3. Since low dynamic

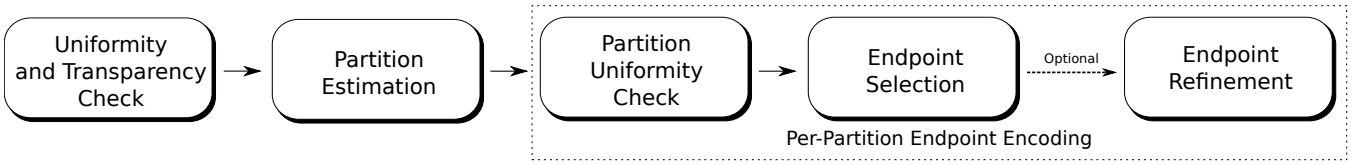

Figure 2: *Overview of FasTC. It is applicable to all fixed-rate compression formats that support partitioning.*

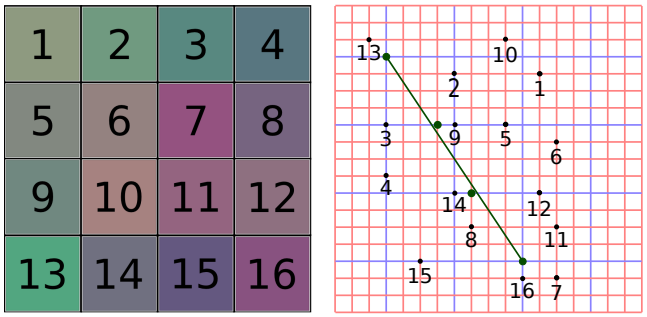

Figure 3: *(right) A 4×4 block of texels. (left) The texels approximated with two-bit indices. The texels are interpreted as points on a lattice defined by the precision of the source texture (red). The endpoints approximating the texels are on a sparse lattice (blue) and the interpolation points are in green. For two bits per index we have $2^2 = 4$ interpolation points. Note: The internal interpolation points do not lie on the line segment due to quantization.*

range color values are usually described with eight-bit color channels, they can be interpreted as points along a 256^3 lattice. The endpoints for each set of texels are usually encoded with a lower per-channel precision, meaning that they exist on a sparse lattice overlaid on the original. The goal of a compression algorithm is to compute the best points that lie on this lattice and per-texel indices that together reconstruct the original texel values.

Since S3TC is the most popular compression format, it has also been extensively investigated in terms of endpoint compression. S3TC operates on blocks of size 4×4, and stores compressed data with a single line segment whose endpoints use 5, 6, and 5 bits for red, green and blue respectively with two bits per index. Simon Brown's *libsquish* [2006] uses a method known as a cluster-fit. In this method, the 16 texel values $\mathbf{p}_i$ are first ordered along their principal axis; then each 4-clustering that preserves this ordering is used to solve the following least-squares problem

$$\min_{\mathbf{a},\mathbf{b}} |(\alpha_i \mathbf{a} + \beta_i \mathbf{b}) - \mathbf{p}_i|$$

where (α_i, β_i) are determined by the cluster that $\mathbf{p}_i$ belongs to:

$$(\alpha_i, \beta_i) \in \left\{(1,0), (\tfrac{1}{3}, \tfrac{2}{3}), (\tfrac{2}{3}, \tfrac{1}{3}), (0,1)\right\}$$

The endpoints $\mathbf{a}$ and $\mathbf{b}$ are then snapped to the lattice induced by the endpoint precision, and the result with the smallest error, as described in Section 3.2, is chosen. This algorithm is also the basis of Castaño's real-time GPU implementation [Castaño 2007].

A CPU-based real-time algorithm was first introduced by J.M.P. Van Waveren that computes the diagonal of the axis-aligned bounding box (AABB) of the texels in color space, and uses the resulting diagonal as endpoints [Waveren 2006]. Although this algorithm does not produce results as high in quality as the NVIDIA Texture Tools [2010], it is fast enough to support compression of textures generated at run-time, such as the frame buffer. Our approach expands upon these ideas in order to allow content pipeline designers to compress textures at a higher quality.

3.2 Problem Formulation

Formally, a compression method for endpoint-based texture compression takes as input n texels $\mathbf{p}_i = \left(p_i^r, p_i^g, p_i^b\right)$, a triplet $\zeta = (\zeta_r, \zeta_b, \zeta_g)$ that denotes the number of bits per channel in the compressed endpoint data, and an integer $\mathbf{I}$ specifying the number of bits per index. The output of the compression method is a pair of endpoints $(\mathbf{p}_a, \mathbf{p}_b)$ and n index values d_i, with $0 \leq d_i < 2^{\mathbf{I}}$. Here p_a^k and p_b^k, with $k \in \{r, g, b\}$, are specified with ζ_k bits. The goal of a compression method is to minimize the total error of the compressed texels, i.e.

$$\min_{\mathbf{p}_a, \mathbf{p}_b} \Phi(\mathbf{p}_a, \mathbf{p}_b),$$

where $\Phi(\mathbf{p}_a, \mathbf{p}_b)$ is the *endpoint compression error* defined as

$$\Phi(\mathbf{p}_a, \mathbf{p}_b) = \left[\sum_{i=0}^{n} \sum_{k \in \{r,g,b\}} \left| \frac{p_a^k \left(2^{\mathbf{I}} - d_i - 1\right) + p_b^k d_i}{2^{\mathbf{I}} - 1} - p_i^k \right| \right].$$

The values of d_i are inferred from the given endpoints $\mathbf{p}_a$ and $\mathbf{p}_b$ and the input data $\{\mathbf{p}_i\}$ by assigning each input value to the closest interpolation point between $\mathbf{p}_a$ and $\mathbf{p}_b$. This minimization problem is a special case of the quadratic integer programming problem, and it can be reduced to computing the shortest vector on a lattice (SVP). SVP is known to be NP-Hard [van Emde Boas 1981] [Ajtai 1998] and most techniques approximate the optimal solution. We refer to this specific instance of the problem outlined above as the *endpoint optimization problem*. The triplet ζ is known as the *endpoint precision*, and $\mathbf{p}_a$, $\mathbf{p}_b$, and $\{d_i\}$ are collectively known as a *palette*. The number of bits per index $\mathbf{I}$ is known as the *index precision*.

3.3 Overview

FasTC operates on the BPTC format. We have chosen this format because of the availability of compressors and hardware support for decompression in current GPUs. We expect all of the methods described in this paper to be equally applicable to other block-based compression formats such as ASTC. Figure 2 gives an overview of our algorithm.

We consider the input to be a square block of texels. The first step of the algorithm is to check whether or not all of the texels are uniform or transparent. If not, we estimate the best partitioning for the block as described in Section 3.4. Next, for each subset in the partition, we perform another uniformity check and approximate the best endpoints to use (Section 3.5). We discuss an optional refinement step that searches for better solutions using simulated annealing (Section 3.6).

For a given index and endpoint precision, most eight-bit values can be compressed with an index value of one. For example, with six-bit endpoint precision and two-bit index precision, the value 73 is

exactly encoded as the first index between endpoints (64, 92) after integer truncation. For a uniform block, if we assume that each texel will have an index value of one, we can find the optimal endpoints for this block by saving the optimal endpoints per channel in a lookup table with 256 entries. In this manner, we can effectively discard uniform blocks and partitions. This procedure corresponds to the uniform and transparency checks in Figure 2

3.4 Choosing a Partitioning

As mentioned in Section 2, some modern texture compression formats support partitioning the texels into subsets. Since saving partitioning information for each individual texel would incur too much overhead, formats that support partitioning supply a predetermined list of common partitionings called *shapes*. Each shape partitions the block into *subsets* as illustrated in Figure 4.

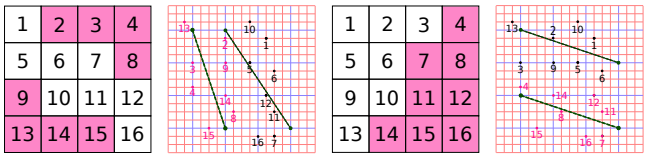

Figure 4: *A 4×4 block partitioned by different shapes into two subsets from the BPTC format. Shape partitioning is determined based on a lookup into a table of common partitionings. The texels marked with a pink background belong to one subset and the unmarked texels belong to another. Each subset is approximated with its own line segment (in green). (left) Shape #31 (right) Shape #4*

The quality of a shape with respect to the block's texels is determined by the interrelationships of the texels in each subset and the input parameters to the compression algorithm. If we ignore endpoint precision, the best shapes are those whose subsets provide clusterings of the data that line up along interpolation points on a line segment. Mathematically, the collinearity of the texels in RGB space would be a good measurement of their ability to be approximated in this way. Collinearity of a point cloud is measured by taking the eigenvalues $\{\lambda_i\}$ of the covariance matrix and comparing the first and second maximal eigenvalues. If the second eigenvalue is small in comparison to the first, then the variation from the principal axis of the point cloud is minimal. Based on this observation, we can posit that the best shape estimation would be one whose average ratio λ_1/λ_0 is small for each subset, where λ_0 and λ_1 are the first and second largest eigenvalues, respectively. However, this method for shape estimation has the disadvantage that we must compute eigenvalues for each of the hundreds of possible shapes in our compression format for each block. As we demonstrate in Section 4, this eigenvalue method incurs significant performance penalties and may not provide good estimates due to quantization artifacts.

A modification of this approach is to calculate the Euclidean distance from points to the principal axis. Instead of computing the first and second eigenvalues of the covariance matrix, one only needs to compute the principal eigenvector, define a line segment with the extremes of the points projected onto this axis, and use this segment as an estimated solution to the endpoint optimization problem for each subset. In fact, some encoders, such as NVIDIA's compressor, perform this step in order to estimate the quality of a given shape [Donovan 2010].

Due to the discrete nature of endpoint optimization, any approximation that relies on the continuity of the domain may introduce inaccuracies. Instead of performing principal component analysis, we propose an approximation to each subset to be the amount of error it would create if it were encoded using the real-time CPU based method introduced by Waveren [2006], which we will refer to as *bounding-box estimation*. Bounding box estimation uses the diagonal of the axis-aligned bounding box (AABB) as an estimate for the line segment that solves the endpoint optimization problem. It also provides an efficient check for uniformity by measuring the length of the diagonal. We present the following metric for approximating the quality of a given shape:

$$e\left(\{\mathbf{p}_i\}\right) = \Phi\left(\psi_+\left(\{\mathbf{p}_i\}\right), \psi_-\left(\{\mathbf{p}_i\}\right)\right)$$

where

$$\psi_+\left(\{\mathbf{p}_i\}\right) = \left(\max_i(p_i^r), \max_i(p_i^g), \max_i(p_i^b)\right),$$

$$\psi_-\left(\{\mathbf{p}_i\}\right) = \left(\min_i(p_i^r), \min_i(p_i^g), \min_i(p_i^b)\right)$$

The index precision $\mathbf{I}$ and endpoint precision ζ which influence the value of Φ should be chosen based on the encoding format. In our implementation, we use $\zeta = (8, 8, 8)$ and $\mathbf{I} = 2$ for three-subset shapes and $\mathbf{I} = 3$ for two-subset shapes. This approximation provides the main speed-up for our algorithm. For each block it must be performed twice per each of the 64 shapes in BPTC (once for two-subset shapes, and once for three-subset shapes). The ramifications of this approximation, with respect to compression quality and speed, are given in Section 4.

3.5 Endpoint Estimation

There are currently two main compression algorithms that are widely used to solve the endpoint optimization problem with respect to S3TC. One is AMD's Compressonator [2008], which does not have any released source code. The other has been incorporated from Simon Brown's *libsquish* [2006] into NVIDIA's texture tools [2010]. As described in Section 3.1, Brown's cluster-fit algorithm searches over all 4-point clusterings that preserve the points' ordering along the principal axis. The core of Brown's algorithm is to assign each texel of a cluster to an interpolation point along the line segment. In this way, we present the generalized cluster fit: Given n texels $\mathbf{p}_i$ and index precision $\mathbf{I}$, compute indices d_i and $2^{\mathbf{I}}$ clusters with centers at $\mathbf{c}_k$ such that

$$\|\mathbf{c}_{d_i} - \mathbf{p}_i\| \leq \|\mathbf{c}_k - \mathbf{p}_i\|$$

for all $0 \leq k < 2^{\mathbf{I}}$ and $0 \leq i < n$. We approximate the solution to the endpoint optimization problem as the pair $(\mathbf{p}_a, \mathbf{p}_b)$ that best approximates the overdetermined system of n equations $\alpha_i \mathbf{p}_a + \beta_i \mathbf{p}_b = \mathbf{p}_i$ where

$$(\alpha_i, \beta_i) = \left(\frac{2^{\mathbf{I}} - d_i - 1}{2^{\mathbf{I}} - 1}, \frac{d_i}{2^{\mathbf{I}} - 1}\right)$$

In S3TC, there are only two bits per index giving $\binom{16+2^2-1}{2^2-1} = \binom{19}{3} = 969$ possible clusterings. In BPTC, the texels can be encoded with up to four bits per index. This produces a total of $\binom{16+2^4-1}{2^4-1} = \binom{31}{15} \approx 3 \times 10^9$ possible 16-clusters making an exhaustive search infeasible. Charles Bloom presents an alternative method for S3TC that only uses a 2-means clustering along the principal axis instead of iterating through each possible clustering [Bloom 2009]. Similarly, we assume the best clustering to be along the direction of the principal axis. In FasTC, instead of iterating through all possible clusterings, we compute an appropriate k-means clustering where $k = 2^{\mathbf{I}}$. As shown in Algorithm 1, we initially project all points onto the principal axis. Then we generate $2^{\mathbf{I}} - 2$ equally spaced points along the axis between the extremes

of the projection. We use these points as a seed to Lloyd's [1982] algorithm to compute a final clustering. Once we solve the least squares problem in $\mathbf{R}^3$, we choose the closest lattice endpoints as our approximation. Instead of projecting points onto the principal axis, we could take the bounding box diagonal as we did during shape estimation in Section 3.4. However, both the difference in quality and speed are negligible using this method.

Algorithm 1 FasTC-0: Generalized cluster fit for estimating a solution to the endpoint optimization problem

Input:
Texels $\mathbf{p}_i \in \mathbf{Z}^3, 0 \leq i < n$
Index Precision $\mathbf{I}$
Output:
Endpoints $\mathbf{p}_a$ and $\mathbf{p}_b$, and indices d_i s.t. $0 \leq d_i < 2^{\mathbf{I}}$

$\mathbf{v} \leftarrow$ ComputePrincipalDirection($\mathbf{p}_i$)
$\mathbf{m} \leftarrow \frac{1}{n}\sum \mathbf{p}_i$

// Initialize the endpoints as extremes along the principal axis
$(q_a, q_b) \leftarrow (\max[(\mathbf{p}_i - \mathbf{m}) \cdot \mathbf{v}], \min[(\mathbf{p}_i - \mathbf{m}) \cdot \mathbf{v}])$
$(\mathbf{p}_a, \mathbf{p}_b) \leftarrow (\mathbf{m} + q_a\mathbf{v}, \mathbf{m} + q_b\mathbf{v})$

// Initialize cluster centers as the points along the segment
$\mathbf{C} \leftarrow \{\mathbf{c}_0, ..., \mathbf{c}_{2^{\mathbf{I}}-1}\}, \mathbf{c}_k = \frac{(2^{\mathbf{I}}-k-1)\mathbf{p}_a + k\mathbf{p}_b}{2^{\mathbf{I}}-1}$

// Compute an initial clustering
$d_i \leftarrow k$ s.t. $\|\mathbf{c}_k - \mathbf{p}_i\| \leq \|\mathbf{c}_j - \mathbf{p}_i\| \forall\, j \neq k \forall\, i$

// Perform k-means clustering to achieve a final clustering $\{d_i\}$
$(\{d_i\}, \mathbf{C}) \leftarrow Lloyd(\{d_i\}, \mathbf{C})$

// Setup and solve overdetermined system of linear equations
$(\alpha_i, \beta_i) \leftarrow \left(\frac{2^{\mathbf{I}}-d_i-1}{2^{\mathbf{I}}-1}, \frac{d_i}{2^{\mathbf{I}}-1}\right) \forall\, i$
$(\mathbf{p}_a, \mathbf{p}_b) \leftarrow (\mathbf{a}, \mathbf{b})$ s.t. $\forall \mathbf{a}_0, \mathbf{b}_0 \in \mathbf{R}^3$ and $0 \leq i < n$

$$\|\alpha_i\mathbf{a} + \beta_i\mathbf{b} - \mathbf{p}_{d_i}\| < \|\alpha_i\mathbf{a}_0 + \beta_i\mathbf{b}_0 - \mathbf{p}_{d_i}\|$$

3.6 Endpoint Refinement

After estimating the endpoints for a given block of texels, we have introduced errors due to quantization. The endpoint estimation algorithm mentioned in Section 3.5 must clamp the final endpoints to the sparse lattice defined by the endpoint precision ζ. Nearby lattice points to these endpoints can provide better approximations to the endpoint optimization problem. However, the initial approximation usually lands within a local minimum of Φ rendering methods that involve gradient descent ineffective. Furthermore, due to the high frequency nature of Φ, it is difficult to locate global minima for a given subset. Most compressors include a localized search around the estimated endpoints to improve their approximation. The size of this search space proves to be another limiting factor in the speed of compression algorithms.

Instead of using a local exhaustive search as in NVIDIA's tool [2010], we use simulated annealing to do a local search for the best endpoints in each subset [Kirkpatrick et al. 1983]. Simulated annealing is particularly well suited for this problem due to the size and discrete nature of the search space. Furthermore, the lack of known structure in the search space gives an advantage to a stochastically-based process. For each step of simulated annealing, we choose neighboring lattice points $\mathbf{q}'_a, \mathbf{q}'_b$ for each of the two endpoints. If the approximation to the endpoint optimization problem $\Phi(\mathbf{q}'_a, \mathbf{q}'_b)$ is better than the initial approximation, then we restart the annealing process with the improved approximation. Otherwise, if the approximation is close enough to the one we calculated in the previous annealing step, we continue the annealing with this approximation. In this situation, *close enough* is measured by some function $Accept$ whose value depends on how far along we are in the annealing process.

Special consideration should be taken when implementing the $Accept$ function. In general, if this function is too restrictive, then there is a high chance that the annealing process will get stuck in a local minimum. Conversely, if the function is too permissive, the annealing might go in directions that have a catastrophically large amount of error. In our studies, we have selected a function with exponential decay:

$$Accept(\epsilon, \epsilon', \tau) = e^{\frac{\epsilon - \epsilon'}{10\tau}}$$

For the remainder of the paper, we will refer to our compressor as FasTC-k, where k is the number of steps used for the simulated annealing portion of our endpoint refinement. We implement at least k steps of annealing for each subset for which we solve the endpoint optimization problem. Since we run the optimization on each block, this incurs a fairly large cost on the run-time efficiency of the total algorithm. However, if we assume that the initial approximation is close to the optimal solution, low values of k should be sufficient for generating high quality images.

3.7 Multi-Core Parallelization

Since fixed-rate formats specify texture encoding on a block by block basis, they are inherently parallel. The most common approach is to spawn as many threads as the host machine has cores and divide the number of blocks evenly across all threads. However, the algorithm that we have presented uses uniformity checks and other tests to attempt to resolve the encoding of a block as quickly as possible. An application parallelized in this way will thus not have every thread finish its work at the same time. For example, one such scenario is for a texture atlas that contains subtextures for a character or scene. In this case, all of the clothing, face, arm and leg textures for a character may contain areas of uniform texture values to separate the components.

In order to fully utilize the multiple CPU cores, we use a worker queue for processing textures. In this scheme, we spawn as many threads as we have cores, but for each thread we only process as many blocks as will fit in L1 cache. In practice, this approach results in faster performance on processors with a large number of cores. Using this method, we obtain parallel scaling proportional to the number of CPU cores.

4 Results

In order to test all acceleration features, we use the BPTC format due to its availability in both hardware and existing toolsets. For this format the two compressors that were tested were the one developed by Walt Donovan provided with NVIDIA's Texture Tools [Donovan 2010], which we will refer to as NVTC, and the one distributed with the June 2010 release of Microsoft's DirectX SDK [2010], which we will refer to as DX-CPU. The tests were performed using both game textures and the standard Kodak Test Image suite [1999] where each texture is 512×768 pixels. Most game textures, such as character maps, are not as high frequency as the images in this test suite and generally are more amenable to compression. We use the Kodak Test Image Suite in order to provide a worst-case scenario in terms of compression quality.

In our comparisons, we assume NVTC to be the baseline for BPTC. We make this assumption because this tool exhaustively explores

kodim13	atlas	small-char	big-char
512×768	512×512	512×512	1024×1024

Peak Signal to Noise Ratio				
Image	FasTC-0	FasTC-256	DX CPU	NVTC
kodim13	41.53	41.68	40.27	42.27
atlas	45.16	45.34	43.77	46.32
small-char	47.84	48.03	46.20	49.38
big-char	47.10	47.37	45.02	48.05

Compression Speed in Seconds				
Image	FasTC-0	FasTC-256	DX CPU	NVTC
kodim13	5.2	48.8	264.4	783.0
atlas	2.7	25.2	118.5	381.7
small-char	3.2	29.4	145.6	376.5
big-char	13.4	125.6	544.1	1760.8

Table 1: *Average compression speed for various compression algorithms. We use a selection of both low and high frequency textures. FasTC easily outperforms all of the other algorithms in terms of speed while maintaining comparible quality. Tests were performed using a 3.0 GHz quad-core Intel Core i7 workstation.*

an extremely large portion of the solution space, and in general produces very high quality results. Although a good metric for overall perceptible compression quality is the structural similarity metric introduced by Wang et al [2004], the fixed-rate nature of compression formats will always introduce block artifacts in compressed images which artificially skew this metric. Instead, we use the canonical metric of *Peak Signal to Noise Ratio* (PSNR), using the following formula given by Ström and Petterson [2007]

$$PSNR = 10 \log_{10} \left(\frac{3 \times 255^3 \times w \times h}{\sum_{x,y} \left(\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2 \right)} \right)$$

As we can see in Figure 5 our algorithm provides competitive results in terms of quality even without simulated annealing. Moreover, it provides better quality than DX-CPU. As a commonly accepted rule of thumb, 0.25 dB of difference in PSNR is noticeable to the human eye. Although our results average less than one decibel worse in quality over NVTC, the relatively high values of PSNR make even this difficult to notice. The performance gains achieved with our method are deomonstrated in Table 1. We observe order-of-magnitude increases in speed over previous implementations without simulated annealing. Figure 6 displays a closeup of the areas that produce the highest error in our algorithm for various images.

4.1 Shape Selection

In Section 3.4 we discussed various different methods for estimating the proper shape to choose when encoding a block. One of those methods was to compare the magnitude of the first and second eigenvalues of the covariance matrix and use their ratio as a measurement of linearity. Figure 7 compares the difference in PSNR between using this method and using bounding box estimation. From this figure, we can see the effect quantization has on shape estimation. Furthermore, the speed of image compression using this method without simulated annealing was 10.7 seconds slower on average.

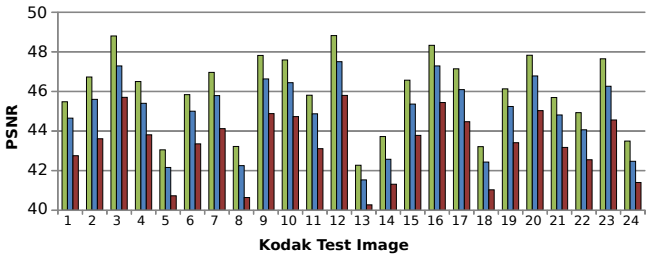

Figure 5: *Peak Signal to Noise Ratio for various compression algorithms. NVTC, the tool provided with NVIDIA's Texture Tools (green). FasTC-0, our algorithm without simulated annealing (blue). DX CPU, the tool provided with Microsoft's DirectX SDK (red). Our algorithm (FasTC-0) provides similar quality to existing implementations.*

Context Original NVTC FasTC-50

Figure 6: *Detailed investigation of areas with high noise in the Kodak Test Images that produce lowest PSNR. We notice that the visual quality of FasTC is comparable to NVTC and close to the original texture.*

In order to further demonstrate the effects of quantization on shape selection, Figure 8 shows the difference in NVTC when we use bounding box estimation versus measuring distance from the principal axis. The average compression time of a single texture from the Kodak Test Image Suite reduces to 181.3 seconds from 1384.6 seconds. Moreover, the minimal difference in compression quality suggests that the largest gain in quality comes from an exhaustive search around the coarse approximation to the endpoint selection. This reinforces the assumption that taking a better approximation to the endpoints as outlined in Section 3.5 will accelerate texture encoding by reducing the need for this kind of search.

4.2 Simulated Annealing

We discussed simulated annealing in Section 3.6 as an alternative to an exhaustive search of the neighboring area. As expected, we observe a linear increase in compression time with respect to the number of annealing steps. Figure 9 demonstrates the increase in quality from using simulated annealing. In general, we see a sublinear increase in compression quality as more steps of simulated annealing are applied. This means that small amounts of simulated annealing are beneficial, but because of the nature of the search space, excessive application of the annealing process does not produce better

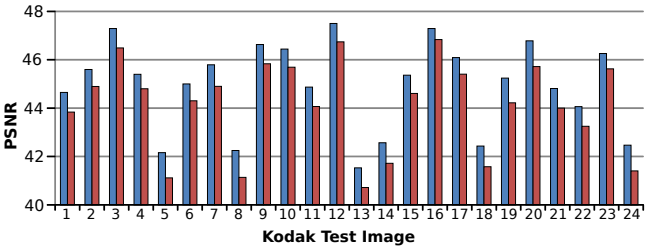

Figure 7: *Peak Signal to Noise Ratio for FasTC-0 using bounding box estimation (blue) and eigenvalue comparison (red). In this experiment, we replaced the shape estimation technique from FasTC-0 with one that uses the ratio of the first and second eigenvalues of the covariance matrix. We can see that due to quantization errors, using bounding box estimation produces better results.*

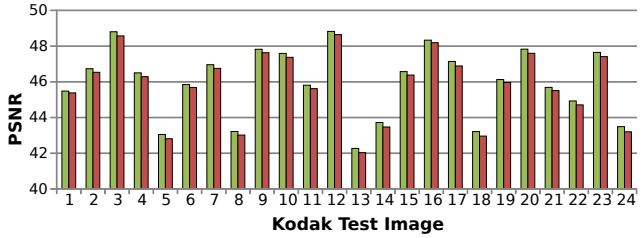

Figure 8: *We compare the compression quality of the original NVTC (green) with a modified version that measures shape quality using bounding box estimation (red). In general, the difference in PSNR is very small, but we avoid a costly eigenvector computation during shape estimation giving us up to 10x in performance gains.*

results. The cause of such tapering is twofold. First, if the simulated annealing takes place over a long period of time, it allows the procedure's endpoint approximations to wander away from the optimum early on due to the loose restrictions of the *Accept* function. Second, once the annealing process passes a certain point, the *Accept* function will only accept errors that are very close to the best error, causing the algorithm to loop in a local minimum. We recommend no more than 64 steps of annealing to achieve the best ratio between performance and quality.

4.3 Parallelization

Each compression format that supports fixed-rate encoding operates on separate blocks of data independently making compressors inherently parallelizable. Our compression method is no different and observes speedups that scale proportionally to the number of cores in a machine. This massive amount of parallelization lends itself to high end work-stations in content pipelines, and likely GPU optimization. Moreover, the relatively small amount of data that must be read for each block means that the entirety of the computation is cache resident and hence scales with computing power. Figure 10 represents the gains in compression speed for various configurations.

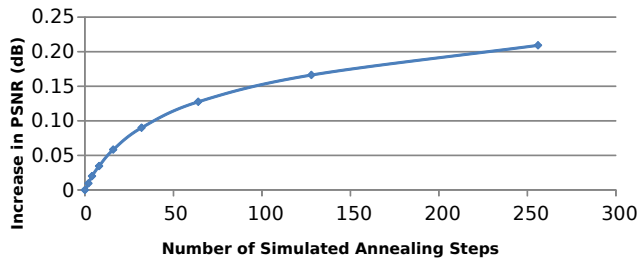

Figure 9: *Average increase in Peak Signal to Noise ratio for the images in the Kodak Test Image suite for various different amounts of simulated annealing. The increase in quality is sublinear due to the nature of the solution space.*

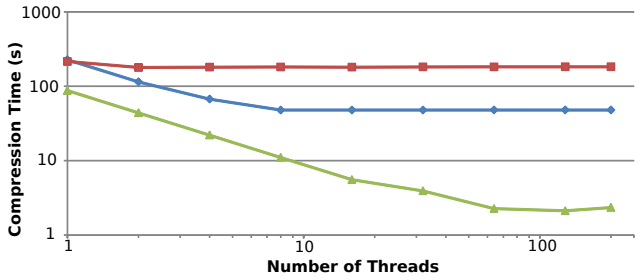

Figure 10: *Compression time in seconds for FasTC-0 of a 2048^2 sized texture on different multi-core configurations using different numbers of threads. Tests were run on a Single-Core 3.00 GHz Pentium 4 running 32-bit Ubuntu Linux 11.10 (red), Quad-Core 3.50 GHz Intel Core i7 running 64-bit Windows 7 (blue) and 40-Core 2.40 GHz Intel Xeon running 64-bit Ubuntu Linux 11.04 (green). We observe a linear speedup with the number of cores.*

5 Closing Remarks

Limitations: The approximations we have presented use heuristics that have not been fully explored. In the simulated annealing step of the optimization, instead of choosing neighbors randomly, it would be better to weigh neighbors that are likely to produce better endpoints. Also, if we are in a sufficiently severe local minimum then we will cycle through all of the nearby endpoints without proceeding. Furthermore, the generalized cluster fit is based off of a continuous representation. There may be methods that operate in the discrete solution space that could avoid quantization errors and be leveraged to require fewer annealing steps.

Future Work: We believe that there is room to build upon the methods introduced in this paper. First and foremost, the massively parallelizable aspect of block truncation coding lends itself to both SIMD and GPU implementation. We believe that with these enhancements such implementations would be able to achieve rates that are viable for real-time encoding. Second, the only methods that were introduced for shape estimation were localized to the block that they were operating on. Due to the fact that shapes are known prior to encoding, the optimal shape for a neighboring block may indicate what a likely shape is for a block adjacent to it. In this way we could split up the block estimation step to be a subset of all total blocks and subsequently only search a much smaller subset for the remaining blocks. Finally, we believe that the methods presented here are applicable to compressing HDR textures as well.

Conclusion: We have presented a new method, FasTC, for the acceleration of encoding textures for formats that employ variations of Block Truncation Coding. FasTC offers up to orders of magnitude increases in compression speed while maintaining high compression quality in terms of PSNR. Moreover, we present a flexible paradigm that gives the content pipeline designer a mechanism to choose between encoding time and compression quality.

Acknowledgements: This work was supported in part by ARO Contract W911NF-04-1-0088, NSF awards 0917040, 0904990, and 100057 and Intel. The authors would also like to thank Ignacio Castaño for his feedback and David Houlton for providing guidance during the early phase project.

References

AJTAI, M. 1998. The shortest vector problem in l2 is np-hard for randomized reductions (extended abstract). In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, ACM, New York, NY, USA, STOC '98, 10–19.

AMD, 2008. The compressonator. available at http://developer.amd.com/archive/gpu/compressonator.

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, New York, NY, USA, SIGGRAPH '96, 373–378.

BLOOM, C., 2009. Oodle, distributed with the granny3d package from rad game tools. available at http://www.radgametools.com/granny.html.

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

CAMPBELL, G., DEFANTI, T. A., FREDERIKSEN, J., JOYCE, S. A., AND LESKE, L. A. 1986. Two bit/pixel full color encoding. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '86, 215–223.

CASTAÑO, I. 2007. High quality dxt compression using cuda. *NVIDIA Developer Network*.

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

DONOVAN, W., 2010. Bc7 export, distrubuted with nvidia texture tools. available at http://code.google.com/p/nvidia-texture-tools/.

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

FRÄNTI, P., NEVALAINEN, O., AND KAUKORANTA, T. 1994. Compression of digital images by block truncation coding: A survey. *The Computer Journal 37*, 4, 308–332.

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

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.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598, 671–680.

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

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

KRAUSE, P. K. 2010. ftc–floating precision texture compression. *Computers Graphics 34*, 5, 594–601. CAD/GRAPHICS 2009, Extended papers from the 2009 Sketch-Based Interfaces and Modeling Conference, Vision, Modeling & Visualization.

LLOYD, S. 1982. Least squares quantization in pcm. *Information Theory, IEEE Transactions on 28*, 2 (mar), 129–137.

MAVRIDIS, P., AND PAPAIOANNOU, G. 2012. Texture compression using wavelet decomposition. *Computer Graphics Forum 31*, 7(1), 2107–2116.

MICROSOFT, 2010. Directx software development kit. http://www.microsoft.com/en-us/download/details.aspx?id=6812.

MUNKBERG, J., CLARBERG, P., HASSELGREN, J., AND AKENINE-MLLER, T. 2008. Practical hdr texture compression. *Computer Graphics Forum 27*, 6, 1664–1676.

NASRABADI, N. M., CHOO, C. Y., HARRIES, T., AND SMALLCOMB, J. 1990. Hierarchical block truncation coding of digital hdtv images. *IEEE Trans. on Consum. Electron. 36*, 3 (Aug.), 254–261.

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, Aire-la-Ville, Switzerland, Switzerland, HPG '12, 105–114.

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

ROIMELA, K., AARNIO, T., AND ITÄRANTA, J. 2006. High dynamic range texture compression. In *ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, SIGGRAPH '06, 707–712.

STRÖM, J., AND AKENINE-MÖLLER, T. 2004. Packman: texture compression for mobile phones. In *ACM SIGGRAPH 2004 Sketches*, ACM, New York, NY, USA, 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, New York, NY, USA, 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, Aire-la-Ville, Switzerland, Switzerland, GH '07, 49–54.

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

VAN EMDE BOAS, P. 1981. Another np-complete partition problem and the complexity of computing short vectors in a lattice. Tech. Rep. MI-UvA-81-04, Department of Mathematics, University of Amsterdam, April.

WANG, Z., BOVIK, A., SHEIKH, H., AND SIMONCELLI, E. 2004. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on 13*, 4 (april), 600–612.

WAVEREN, J. M. P. V. 2006. Real-time dxt compression. *Intel Software Network*.