VBTC: GPU-Friendly Variable Block Size Texture Encoding

P. Krajcevski  A. Golas  K. Ramani  M. Shebanow  and D. Manocha

1 The University of North Carolina at Chapel Hill, USA
2 Samsung Research America, USA

Abstract
Recent advances in computer graphics have relied on high-quality textures in order to generate photorealistic real-time images. Texture compression standards meet these growing demands for data, but current texture compression schemes use fixed-rate methods where statically sized blocks of pixels are represented using the same numbers of bits irrespective of their data content. In order to account for the natural variation in detail, we present an alternative format that allows variable bit-rate texture compression with minimal changes to texturing hardware. Our proposed scheme uses one additional level of indirection to allow the variation of the block size across the same texture. This single change is exploited to both vary the amount of bits allocated to certain parts of the texture and to duplicate redundant texture information across multiple pixels. To minimize hardware changes, the method picks combinations of block sizes and compression methods from existing fixed-rate standards. With this approach, our method is able to demonstrate energy savings of up to 50%, as well as higher quality compressed textures over current state of the art techniques.

1. Introduction
Over the last few decades, texture mapping has become a standard feature for consumer-level desktop and mobile GPUs [max, nVi15, img, arm, R9g]. The cost of adding texture mapping capabilities on graphics hardware includes additional memory for storing textures and the data bandwidth therein for transferring and accessing these textures. These additional features result in a significant increase in data access energy leading to reduced battery life. Fetching a byte of data from a modern DRAM memory incurs 74-200 picojoules (pJ) [KDK∗11, Ros12] while performing a floating point operation incurs only 5-10 pJ [KDK∗11]. As a result, reducing memory bandwidth for texture accesses has been an important design metric, especially for mobile GPUs.

Over the last two decades, a number of standards have emerged to allow lossy compressed textures to be used efficiently on GPUs with dedicated hardware decompression circuitry [INH99, Fen03, SAM05, SP07, Ope10, NLP∗12]. To improve hardware decompression performance and reduce energy usage, these formats work with blocks of pixels, 4 × 4 being the historically popular choice, commonly compressed down to eight or sixteen bytes. These formats follow both fixed-rate compression, and fixed-rate addressing. Fixed-rate compression requires that each block of pixels be compressed to the same number of bytes, while fixed-rate addressing implies that the physical memory location of a block can be computed purely from the coordinates of the block within the texture.
These two features allow GPU hardware to fetch and decompress compressed texture blocks with high throughput and low latency, both of which are essential to obtain high GPU performance and energy efficiency.

One artifact of using compressed textures is that fixed-rate compression introduces an inherent quality versus size tradeoff. Since a fixed number of bytes are used to represent varying amounts of local image detail, higher compression rates may be obtained at the cost of lost texture detail. In cases where these details are localized to small portions of the texture, this requires the user to reduce compression ratios for the entire texture to maintain the necessary detail. With increasing photorealism in computer graphics, high-detail textures occur with increasing frequency [GO14]. In the context of mobile graphics, penalizing the compression ratio for an entire texture due to sparse, localized regions of high detail is less acceptable than on desktops. Recent industrial approaches that take steps towards addressing this problem include the Nvidia Maxwell GPU [nV15, img], which implements framebuffer compression although the actual details are not public.

**Main Results:** We present a new and practical approach to support variable bit-rate texture compression on mobile GPUs. This includes a new variable bit-rate compression algorithm as well as modified hardware architecture that can support real-time decompression. Our approach is designed to reduce the impact of the quality versus size tradeoff for textures with sparse high-detail regions. We present two adaptive block size techniques with varying levels of granularity. The crux of our approach lies in varying the block size dynamically throughout the texture, as opposed to a static choice. To illustrate the practicality of our method, we utilize the block types proposed by the new Adaptive Scalable Texture Compression (ASTC) standard available on all modern mobile devices compatible with the Android Extension Pack [Ope14]. This existing hardware choice results in incremental changes to the current texture architecture, small modifications to the address computation block, while using the same texture block decomposition that is also used for decompressing ASTC textures.

Variable bit-rate compression in our approach is achieved by adding one level of indirection in the decompression path - adding a metadata dictionary defining the location of the desired block. The form of this dictionary differs among our two proposed variations, but in both it allows the compressor to perform de-duplication of texture blocks, allowing only one copy of a unique compressed block to be stored. This helps in terms of the memory capacity and any caches used in the memory subsystem.

Furthermore, we show that variable bit-rate compression with hardware decompression and random-access for textures on GPUs are not mutually exclusive. In this scenario we are able to get the best of both worlds: our textures are small so we load faster and reduce the memory bus congestion. On the other hand, they are still stored as compressed textures in GPU memory so the number of cache lines fetched during rendering remains low. We gain the benefits of using more color information bits where the textures are detailed and eliminate duplication by using a dictionary style reuse of compressed data. The key contributions of this paper include:

- A novel variable bit-rate texture compression method with efficient hardware decompression for current GPUs. Compared to ASTC, our new adaptive schemes can provide a significant reduction in texture access energy. Results on standard texture benchmarks show that data access energy can be reduced by as much as 33-50%
- A flexible dictionary-based block addressing scheme taking advantage of redundant compressed blocks and maintaining low latency overheads
- A compression algorithm to perform variable bit-rate texture compression with a local image quality threshold

For a wide variety of textures, our method provides comparable compression ratios, some with higher quality, as compared to the reference ASTC compressor [ARM12]. We also provide a reference decompressor design. Our codec generates textures compressed into our proposed format that can be decompressed using modern GPU architectures providing lower memory bandwidth usage with low hardware cost.

The remainder of the paper is organized as follows. In Section 2, we give an overview of prior work in texture compression and texture mapping hardware. We describe our proposed variable bit-rate texture compression format in Section 3, including a reference design for a hardware decompressor capable of decompressing both techniques in Section 3.4. In Section 4, we describe an offline algorithm for compressing textures into our proposed format. We conclude by discussing the results of our algorithm in Section 5 on a dataset of reference textures, and discussing avenues for future work in Section 6.

### 2. Related Work

The first fixed-rate of images was the implementation of Block Truncation Coding by Delp and Mitchell later augmented by Knittel et al. [DM79] [KS96]. A compression and Mitchell were able to define necessary requirements of texture decomposition hardware, namely the need for fixed-rate addressing and an algorithm that can be implemented in hardware. Beers et al. [BAC96] formalized this notion by defining the features that a hardware compression scheme would support: decoding speed, random access, a compression rate versus compression quality tradeoff, and less emphasis on encoding speed.

Texture compression has evolved significantly to meet the demands of modern GPUs. The first commercially available texture compression format was S3TC (also known as DXT1 and BC1) [IN99]. In this format, a single 4 x 4 pixel block is encoded using two sixteen-bit colors and sixteen two-bit interpolation values. Simon Fennelly introduced hardware that took advantage of the worst case of S3TC during texture filtering [Fen03]. Ström and Akenine-Möller introduced PACKMAN and iPACKMAN that decouple the chrominance from luminance during compression [SAM05] [SP07]. Table-based alpha compression by Wennersten and Ström shows a one bit-per-pixel scheme for eight-bit textures that achieves higher quality compression than previous formats [WS09]. However, most recently, there has been considerable work in developing significantly higher quality compression and more aggressive compression rates [Ope10]. With the introduction of ASTC, Nystad et al. [NLP12] demonstrate a single hardware unit that can compress 32-bit RGBA pixels from eight bits per pixel all the way down to 0.89 bits per pixel.
Additionally, there is extensive work on designing texture mapping hardware to provide high bandwidth to texture memory and low latency access [KSKS96] [HG97] [Wei04]. Other methods include a block-oriented lossless texture compression algorithm based on a simple variable bit-rate differencing scheme [IM06] and real-time decompression of textures that are compressed with a variable bit-rate compression scheme such as JPEG [IB14]. However, these methods require expensive addressing schemes and do not leverage existing compressed texture hardware. More recently, commercial mobile GPU manufacturers have support for surface or render target compression [nVi15, max, img]. Our approach is similar to these in terms of improved energy efficiency. As few hardware and algorithmic details are public, it is nonetheless difficult to compare against commodity GPUs.

Variable bit-rate compressed texturing solutions have been proposed by Olano et al. using a range coder and decompressing them in a shader program on the GPU [OBGB11]. This compression scheme uses texture mipmap levels to predict the higher resolution colors and encodes the prediction error. Although this scheme generates small textures providing savings in bandwidth over the shared memory bus, the textures are still ultimately stored at full resolution in GPU memory. For embedded devices, such as mobile phones, the number of cache lines required to fetch texel data has a direct correlation with power consumption.

3. Variable bit-rate Texture Compression

Much of prevailing literature in the field of texture compression assumes a single memory lookup operation [NLP12, Ope10] per texel, utilizing fixed-rate compression schemes where each texel uses exactly the same number of bits. This approach has certain downsides, the primary shortcoming being that it is agnostic to the natural variation of detail within a texture. Most textures demonstrate a variation of detail within the image by possessing regions of high and low detail.

Consider Figure 2 for example, where a fixed-rate compression scheme would utilize the same quality representation for the entire image. To represent these regions in a compressed format while preserving salient features requires a varying number of bits-per-pixel – large numbers for high-detail regions like tiles with the edges of the character ‘S’, and smaller for low-detail ones like the constant color white or blue regions. An end-user using a fixed-rate compression format must make a tradeoff between image quality and compression – either choose a large texture to preserve high-detail regions, or compromise on quality in these regions and get higher compression ratios. The second shortcoming of these formats is that regions within a texture which are duplicates of each other cannot be mapped to the same region, which leads to missed compression opportunities for the texture.

3.1. Two-level Texture Layout

To remedy the aforementioned shortcomings, we propose a two-level compressed texture layout to enable variable bit-rate texture compression. This approach is a dictionary-based scheme using a metadata dictionary for:

- Addressing and fetching a particular block of texels
- Describing the method of compression for the given block

To minimize the amount of required hardware changes for supporting this scheme, we utilize existing hardware decompressors for decoding blocks of texels. In particular, we use block types proposed by Nystad et al. [NLP12] as the underlying block storage formats – changing only the data fetch portion of the pipeline.

The metadata used in this paper is fixed-rate, simplifying address calculations for metadata fetch. In addition, hardware designs can be optimized to ensure that a small metadata cache will reduce the number of memory accesses per texel from the theoretical maximum of two to an effective rate very close to one. The size of metadata presents a tradeoff between the overhead of storing additional metadata, and the compression ratios it enables. To illustrate this tradeoff, we present two possible metadata definitions along with the flexibility they provide - one minimizing metadata overhead, and the other providing increased flexibility to enable higher compression ratios.

3.2. Adaptive Compression with Metadata per 12 × 12 block

The first proposal maintains metadata at a 12 × 12 block granularity, representing it as one of the following choices:

- a 12 × 12 block
- a combination of one 8 × 8 sub-block, and five 4 × 4 sub-blocks
- four 6 × 6 sub-blocks
- nine 4 × 4 sub-blocks

These 7 configurations, described in Figure 3, can be encoded using a 3-bit code, augmented with a 21-bit block offset from the texture base address. For decompressing any 12 × 12 block, the decompressor reads 3 bytes of metadata, followed by at least 16 bytes of data. The metadata will be followed by at most 144 bytes if the 12 × 12 block is comprised of nine 4 × 4 ASTC blocks.

Modern computing systems interface with the memory subsystem in chunks of data - cache lines - commonly sized to 32 or 64 bytes - dictating that all accesses will fetch that amount of data irrespective of the amount of data actually needed. Since metadata for a 12 × 12 block consists of 3 bytes, a 64-byte cache line will contain data for at least 21 blocks, i.e. 21 · 144 = 3024 texels. Matching the metadata memory layout to the expected access patterns, such as using a 2-dimensional Z-order curve, one cache line of metadata can service metadata requests for a significant percentage of texture.
Our proposed metadata layout of the previous section constrains the possible configurations of ASTC sub-blocks to minimize metadata overhead, at the cost of possibly higher compression ratios. An alternative approach is to store metadata for every 4 × 4 block - the finest possible granularity in ASTC. In this paradigm, each 4 × 4 may belong to one of the following:

- flat/constant block: all pixels within the block have the same color value
- 4 × 4 block
- one of 4 sub-blocks of an 8 × 8 block
- one of 9 sub-blocks of a 12 × 12 block

These 15 configurations can be expressed using a 4-bit code, augmented with a 20-bit block offset to maintain byte aligned data. Note that each metadata entry is 3 bytes long, now corresponding to a 4 × 4 block, implying that fixed overhead of the metadata size is nine times larger than the formulation described in Section 3.2. However, this layout lends more flexibility for larger blocks (8 × 8 or 12 × 12) to be placed within the texture, making higher compression ratios more likely depending on the texture data.

This layout also allows for more flexible data packing, particularly for flat blocks, in which all texels have the same color value. Two of the 4-bit code values (using the 16th available configuration) indicate flat blocks, the first indicating storage in the first half of a 16-byte compressed block, the second indicating the latter half. An offline compressor can utilize this data layout to make similar blocks point to the same memory location, improving the hit rate of any caching mechanism for texture data.

### 3.4. Unified Adaptive Compression and Decompression

The two manifestations of metadata shown above can be unified into a common metadata decoder akin to multiple kinds of texture formats commonly implemented in a texture pipeline. The input to such a block can be an offset within the uncompressed texture or texel coordinates, the output being the resulting compressed data to be decompressed by an existing ASTC decoder, or in the case of constant blocks, the color data itself.

Figure 5 highlights the process of translating a texture request into a compressed block which can be processed by the decompressor. In a traditional pipeline, this begins by mapping the requested texel to a block of texels within the texture – 4 × 4 in most cases. Since the traditional pipeline has fixed-rate addressing, this block coordinate can be translated into a memory address for the compressed block using the fixed size of each compressed block, and the layout of blocks in memory.

In our proposed method, we replace the block generation logic with a metadata table lookup. Once the (s,t) coordinate is translated to a block – 4 × 4 or 12 × 12 in our case – the address generation logic can be used to address into the metadata table instead of compressed texture data. Once fetched, the metadata provides a map into compressed texture space to fetch the appropriate texture block. Though at first glance this seems to imply two memory accesses per texture access, this is not the case in practice. Given that data is accessed at cache-line granularity, one metadata request fetches data for multiple blocks, which can be cached in a small metadata cache within the texture fetch hardware.

As described in Section 3.2, we can exploit the fact that most texture accesses are spatially coherent, meaning that a request for a specific texel will be followed and preceded by requests for its neighbors. A 64-byte cache-line holds metadata for 21 blocks, and ordering metadata blocks in a Z-order can help this cache achieve high hit rates, reducing the penalty of metadata access. In cases where an additional cache exists for uncompressed texture blocks, the metadata parser can also be used to improve its hit rates by remapping multiple compressed blocks to the same data. For constant blocks, the parser can directly return uncompressed block data, or pass the color value to the decompressor with a flag set to denote constant block data. These two optimizations provide energy savings in addition to those already provided by higher compression ratios.

With block-based addressing, each offset in both the 4 × 4 and
12 × 12 metadata refers to an ASTC-compressed block. The worst-case subdivision criteria for both schemes is to have every block use 4 × 4 ASTC blocks. It follows that the block addressing for both schemes requires offsets at this granularity. In each compression scheme we either use 20 or 21 bits for block offsets, meaning 220 or 221 total 4 × 4 blocks. This implies a texture dimension limitation of 4096 × 4096 for 4 × 4 metadata and a limitation of 8192 × 4096 for 12 × 12 metadata.

4. Offline Compression

Following the requirements of texture compression formats from Beers et al. [BAC96], we use an offline compressor to encode our images. The compression problem can be posed as a constrained optimization problem to minimize compressed texture size, while satisfying the following constraints:

- A user-specified error threshold ε
- Block sizes and configurations belong to an allowed subset of possible options

The second constraint differs for the two metadata layouts proposed in Section 3.2 and Section 3.3, since allowed block sizes and configurations differ between them.

4.1. Compressor Structure

In order to perform offline compression we rely on the ASTC reference codec implementation [ARM12] as a black-box for compressing a block of texels. The codec exposes a variety of settings that control the quality versus speed of the compressor. The settings used to perform the compression for each block are chosen to match those used to generate the comparisons demonstrated in Section 5.

The compression process then iterates through the possible block configurations from most to least efficient with respect to the compression size. The available block configurations are dependent on the block-size granularity of the metadata. For each iteration, if a certain configuration provides adequate compression quality with respect to the user-specified error threshold ε, then that configuration is chosen. After tests with the L1, L2, and L∞ norms, we selected the L-infinity norm to determine the feasibility of a compressed ASTC block. We chose this norm based on a subjective analysis of the results provided. More formally, a block of size N × M pixels can be treated as a value of an NM dimensional Euclidean vector space. From this definition, the L∞ norm provides a very useful metric given a block x and a decompressed block y

$$|x - y|_{\infty} = \max_i |x_i - y_i|,$$

where x_i and y_i are individual texels within the block. Other candidates included the L2 norm, which has the downside of rejecting blocks that have low absolute error but high aggregate error. This property of the L2 norm proved to be more difficult to tune due to the nonuniformity of blocks across a texture.

One of the main benefits of variable rate encoding is the ability to reuse compressed blocks by duplicating the offset stored in the metadata. In order to effectively search for matching blocks within the user-specified error threshold, we use Vantage-Point Trees [Yia93] to effectively store decompressed block representations. Prior to compressing any new block, we first search for an existing block representation in the VP-tree, as in [KM14].

4.2. Compression for 4 × 4 metadata

For any compression scheme based on ASTC, 12 × 12 blocks are the largest possible sized block that can serve as a basis. In our compressor, we begin by dividing the entire texture into 12 × 12 blocks. We keep the blocks that fall within ε of the original texture based on the metric described in Section 4.1. For each 12 × 12 block that is kept, we insert the appropriate decompressed blocks into various VP-Trees. One 12 × 12 block creates nine new 4 × 4 entries, four 8 × 8 entries, and one 12 × 12 entry into three separate VP-trees. When considering subsequent 12 × 12 blocks, we first check these VP-tress for any already compressed block accurately approximating the current block.

After processing the 12 × 12 blocks that provide adequate compression quality, we investigate the blocks that need to be subdivided. We proceed by looking at each of the possible 8 × 8 configurations of the uncompressed 12 × 12 blocks (Figure 3). For each
We take full advantage of the offset in the metadata by reusing the existing compressed blocks and looking them up in a VP-Tree. However, the greedy strategy described is by no means an optimal solution. Indeed, the problem of determining the best compressed representation for a given texture is a constrained optimization problem and is at best a special case of the set-cover problem, which is known to be NP-complete [Kar72]. For example, there is no reason that two $4 \times 4$ blocks in separate areas of the texture cannot belong to the same $12 \times 12$ block. Furthermore, it is not necessary that the entire $12 \times 12$ block be used in order to provide compression benefits, such as when there is only details in the corners. In this case, it would be useful for most of the blocks to be included in the $12 \times 12$ block while using $4 \times 4$ blocks for the corners. Another limitation of this approach is that it only considers $12 \times 12$ blocks on boundaries that are a multiple of 12. The hardware decoder does not have this restriction and introducing it only hinders the possible results.

### 4.3. Compression for $12 \times 12$ metadata

For textures using $12 \times 12$ metadata entries, the problem simplifies considerably. While we still have the ability to reuse compressed data, we only need to remember decompressed $12 \times 12$ blocks instead of all available dimensions. For each $12 \times 12$ block, the compressor must choose one of the seven configurations in Figure 3. The most straightforward way to do this is also the optimal with respect to the metric used to determine block error as described in Section 4.1. The compressor goes in order from least to most expensive configuration in terms of bitrate, and the first one to provide an adequate error threshold is the one chosen to represent the block.

The choice between using a $12 \times 12$ granularity metadata versus a $4 \times 4$ granularity, as described in Section 4.2, depends on the content of the texture. For certain textures with very high repetition of details, such as animated sprites in a game, the repetitions can be hidden in the metadata using the $4 \times 4$ metadata entries. However, with textures that have sharp contrasts in the amount of detail of a given area, such as coverage masks, the $12 \times 12$ metadata compression scheme will likely produce better results. Most importantly, however, is that due to the metadata overhead, there are textures that still perform better with simple ASTC compression because of the lack of coherence between different areas and their uniform distribution of texture details, such as natural images.

### 5. Results

We test our method against a few representative images, the 128 textures recently distributed by Pixar [Pix15], the Kodak Test Image Suite [KOD99], and the Retargetme Image suite [RGSS10]. Using our scheme, application developers can choose a baseline quality level for their textures rather than a bitrate. We have run evaluations using two main metrics for compression quality: 

- **Peak Signal to Noise Ratio (PSNR)**
- **Structural Similarity Image Metric (SSIM)** [WBSS04].

For grayscale images, we use $||ΔB||_{\infty} < 4$ and for the color results, we use $||ΔB||_{\infty} < 8$.

We present images with which our method is both suited for (android and alto from Figure 11) and incompatible (Figure 7). Figure 9 compares a few of these images compressed with various algorithms.

The efficiency of compression schemes can be measured by two metrics:

- Memory bandwidth reduction, computed as a ratio of the total adaptive texture size (including metadata) compared to the traditional ASTC compressed texture size in kilobytes (KB). The total adaptive texture size is the sum of the variable rate compressed texture size plus the total size of the metadata. We investigate this size by measuring the number of bits used per pixel.
- Energy reduction, computed as a ratio of the energy cost incurred for fetching the adaptive texture compared to the energy cost of fetching the traditional ASTC texture. The energy cost of the adaptive texture comprises of the following components: energy cost of fetching the compressed block data, and the energy cost of fetching and decoding the metadata. We investigate this value by measuring the number of times we miss a 1 KB L1 cache with 64 byte cache lines.

In mobile systems, the energy cost of fetching a byte of data from DRAM (LPDDR3 or LPDDR4) is in the range of 75 pJ [Ros12] to 155 pJ. For the purpose of this study, we will employ a median energy cost of 115 pJ/byte, inline with LPDDR4 memories that are expected to be used in current and future systems. There is a metadata entry per $4 \times 4$ or $12 \times 12$ block and each block is 3 bytes wide. In addition to the cost of fetching metadata, additional hardware is required to decode and compute addresses (3-5 multiply-add operations) for the various blocks. We will employ an average energy cost of 30 pJ for decoding and computing compressed addresses for one block of metadata.

The majority of energy costs are incurred by fetching the data for a given texture. In order to measure the energy efficiency of our algorithm, we simulated a direct-mapped cache with a least recently used replacement scheme to determine how many times we would incur a cache miss for various cache sizes. We tested using various cache sizes each assuming a cache line size of 64 bytes. We measured the number of cache misses incurred by reading out an entire texture by three different access patterns: morton ($z$-curve), raster, and random access. The results for general texture images are displayed in Figure 9 and Figure 10. We use this analysis to show the benefits of our algorithm on mixed-detail images in Figure 11.

The key avenues by which our proposed method may provide increased compression are the use of coarser block sizes, and removal of redundancy within the image data. In the following two sections, we analyze the compression benefits shown in Figure 11 to demonstrate their efficacy.
Figure 6: Block distribution with (left) 4x4 metadata and (right) 12x12 metadata. Comparing these results to Figure 11, we observe that higher percentages of 12 × 12 blocks leads to higher compression rates.

Figure 7: Images with which our algorithm performs relatively poorly. In these images tuning the local subdivision criteria proves difficult. We observe this in images that have uniform low detail with noise, such as bump maps. For comparison with ASTC, we observed (left) 43.8 PNSR (8 bpp) against 40.8 PSNR (9 bpp) with adaptive 4x4 metadata, (middle) 55.27 PSNR (8 bpp) against 40.4 PSNR (7.58 bpp) with adaptive 12x12 metadata, and (right) 39.43 PSNR (5.12 bpp) against 32.94 PSNR (6.32 bpp) with adaptive 4x4 metadata

5.1. Compression vs. Image Complexity

Figure 6 shows the distribution of block sizes used in compressing the test images shown in Figure 11. Correlating the bitrates of these images with the block distribution provides some useful insights.

Firstly, as expected, higher percentages of 12 × 12 leads to lower bitrates and higher compression. Due to the quality threshold, the occurrence of such blocks varies with the content of the image — the android image shows higher percentages of coarser blocks like 12 × 12 due to its large expanse of white, while the galaxys6 image has the lowest occurrence of the same owing to its natural gradient of colors.

Secondly, the usage of finer metadata (4 × 4 vs. 12 × 12) leads to a higher percentage of coarser 12 × 12 blocks used in the image – an increase of nearly 2x. However, as can be seen by the quality per bit per pixel in Figure 11, this does not translate into more compression due to the metadata overhead. This observation also suggests that a metadata representation that combines the flexibility of 4 × 4 metadata with the size of 12 × 12 metadata would be ideal and thus a promising direction of future research.

5.2. Compression vs. Redundancy

The second avenue for compression is exploiting the redundancy of blocks within the image. Figure 8 shows these statistics for the color images in Figure 11. Again, a higher redundancy in pixel blocks has a positive correlation with compression. The key fact to note is that most images – barring photographs of natural scenes with lighting variations – demonstrate a significant redundancy (as high as 90%) which can be exploited by our proposed method. On the other end of the spectrum, images demonstrating a wide spread of detail expectedly present the worst case results. This is expected due to the high information content in these images which cannot be compressed without lowering the specified quality threshold.

The android image represents the ideal case for our algorithm, with concentrated details surrounded by redundant simple blocks. The compression obtained in such a case is high enough that the metadata size begins to dominate, as noted in the nearly 3x increase in bitrate when moving from 12 × 12 metadata to 4 × 4.

5.3. Energy Efficiency

To understand the data fetch energy improvements, the grandcanyon image is used for explaining the energy results below. grandcanyon is a 512 × 512 texture compressed using ASTC 4 × 4 blocks. There are 128 × 128 ASTC blocks in the texture, thus requiring 256 KB of total storage with ASTC 4 × 4 block based compressed. Fetching this texture from memory incurs a total of 30 microJoules or 0.03 milliJoules (mJ) (115 pJ × 256 × 1024) of energy. We assume that our adaptive 4 × 4 scheme delivers an additional compression rate of 2.3X over 4 × 4 ASTC. The texture compressed using our scheme comprises of approximately 111 KB of compressed data and 48 KB of metadata. Using these numbers, total compressed data fetch energy is 0.013 mJ. Total metadata fetch energy is 0.0056 mJ. Total metadata fetch and decode cost is approximately 0.495 µJ or 0.0005 mJ (128*128 blocks * 30 pJ). Total cost of fetching adaptive ASTC 4 × 4 data is a total energy of 0.019 nJ. Compared to

© 2016 The Author(s)
Computer Graphics Forum © 2016 The Eurographics Association and John Wiley & Sons Ltd.
Figure 9: A comparison of different compression algorithms. (top) We select a few images to compare PSNR against compression size measured in bits per pixel. We compare against ASTC, JPEG2000 (J2K) and Olano et. al. [OBGB11]. The designations v1, v2 and v3 are used to match those presented in the paper [OBGB11]. (middle) The same comparison across all images from the Kodak Test Image Suite [KOD99]. (bottom-left) Comparison of MSSIM [WBSS04] across the Kodak images. (bottom-right) Comparison of cache coherency measured in total cache misses for hardware formats. As we can see from these results, our algorithm performs favorably on images with non-uniform distribution of details. We can contrast the natural images from the Kodak Image Suite against the android image from Figure 11, where our adaptive 12 × 12 variant performs significantly better than ASTC approaching bitrates similar to J2K. Furthermore, we can observe the effect of the metadata overhead in our approach with some images with a larger bitrate than 4 × 4 ASTC. From the cache coherency graph, we can see that for certain images, we are within the same number of cache misses as ASTC.

Figure 10: Similar to Figure 9, we compare a large suite of images with bitrate versus PSNR. The images used were from the Kodak Image Suite [KOD99], the 128 Pixar Textures [Pix15], the Retargetme Image Suite [RGSS10], and the images from Figure 11. In this plot, we notice many of the images here are high detail bump and normal maps, for which our algorithm performs poorly. These images usually contain very uniform detail and require accurate compression. Our method subdivides these images to full 4 × 4 ASTC, creating clusters around at 9.5 bits per pixel for 4 × 4 metadata and 8.1 for 12 × 12 metadata. Similarly, for some images, such as those in the lower-left portion of the plot, their high repetitive nature or large areas of low detail make them ideal candidates for our method. Additionally, we observe a significant difference between non-GPU based variable bitrate algorithms. In particular, J2K has a much more tunable quality threshold that is apparent in the bitrate distributions of images.
Figure 11: An analysis of our method for compressing textures against Adaptive Scalable Texture Compression [NLP12]. We observe that sparse textures, such as alto and android, take advantage of the redundancy inherent in dictionary encoding and produce significant gains. A variety of metrics using the reference ASTC encoder [ARM12] are included. In our adaptive compression schemes we require the error for each grayscale block to be within $||\Delta B||_\infty < 4$ and each color block to be within $||\Delta B||_\infty < 8$. On the left we provide a comparison of the compression artifacts generated by each algorithm. On the right, we compare the compression quality per bit per pixel and the number of times we would miss an 1KB L1 cache with 64B cache lines for raster and morton access patterns.

ASTC 4 × 4 compression, the adaptive ASTC 4 × 4 reduces energy by 36% for the grand canyon texture.

It can be observed that the adaptive scheme reduces overall data fetch energy by up to 24-55% depending on the settings. However, as we can see in Figure 10, the general case observes a net efficiency decrease in energy consumption. This decline is due to the expansion that occurs when compressing the image. The uniform high-complexity of the details of the image allow little room to exploit both redundancy and our adaptive technique. As a result, the image is larger than all ASTC variants and requires more energy to decode.

6. Conclusion and Future Work

In this paper we have proposed a novel variable-rate texture compression format, which provides reductions in memory usage and memory bandwidth usage. For a certain class of images, our technique generates compressed textures with higher quality with smaller bitrate compared to current fixed-rate formats. In addition, the changes to the hardware architecture and decompression logic have a low impact on overall hardware complexity as well as texture fetch latency.

Our proposed approach has certain limitations. For textures with an even distribution of details, which are ideally suited for fixed-rate compression schemes, our method performs poorly, as is evident from certain results in Figure 7. This is expected as the metadata overhead increases the size of a compressed texture with min-
imal benefit. Further investigation is needed to determine the optimal metadata configuration - for all textures in general, as well as optimizations for specific classes of textures. Further progress can be made by improving the algorithm for compressing textures using the 4 × 4 metadata formulation.

One interesting avenue of future work that this approach enables is user-controlled compression of art assets during game production. A painting tool could be easily designed that allows artists to mark specific regions of the texture in which compression should maintain quality, while not prioritizing compression in other regions. Such a tool could reduce the number of iterations used in compressing art assets for games.

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

References


