Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

Compressed Lossless Texture Representation and Caching

T. Inada[1] and M. D. McCool[2],

[1]University of Waterloo and Sony Corporation
[2]University of Waterloo and RapidMind Inc.

Abstract

A number of texture compression algorithms have been proposed to reduce texture storage size and bandwidth requirements. To deal with the requirement for random access, these algorithms usually divide the texture into tiles and apply a fixed rate compression scheme to each tile. Fixed rate schemes are by nature lossy, and cannot adapt to local changes in image complexity. Multiresolution schemes, a form of variable-rate coding, can adapt to varying image complexity but suffer from fragmentation and can only compress a limited class of images. On the other hand, several lossless image compression standards have been established. Lossless compression requires variable-rate coding, and more efficient lossy algorithms also use variable-rate coding. Unfortunately, these standards cannot be used directly as texture compression schemes since they do not allow random access.

We present a block-oriented lossless texture compression algorithm based on a simple variable-bitrate differencing scheme. A B-tree index enables both random access and efficient $O(1)$ memory allocation without external fragmentation. Textures in our test suite compressed to between 6% and 95% of their original sizes. We propose a cache architecture designed to support our compression scheme. Cycle-accurate simulation shows that this cache architecture consistently reduces the external bandwidth requirements as well as the storage size without significantly affecting latency.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture-Graphics processors

1. Introduction

Video games and interactive applications use large and growing quantities of texture data for rendering. This leads to two problems: space and bandwidth. Although the cost of RAM is decreasing, the memory space available for the storage of textures is limited. The overall bandwidth of the memory bus can also limit performance. High depth complexity scenes and complex shaders can result in several texture reads per pixel, so texture bandwidth is not limited by display resolution.

Complex multipass algorithms also often require repeated allocation and deallocation of buffers. Allocating textures as large contiguous regions can cause memory fragmentation and limits the flexible use of GPUs for general-purpose computation. Instead, we would like to allocate memory using a simple free list, which requires organizing memory into fixed-size blocks. Such a memory organization also simplifies the implementation of advanced memory management mechanisms such as virtual memory paging.

The most critical problem is bandwidth, although our approach addresses all of the above issues. Memory bandwidth is not growing as fast as on-chip computational performance. This suggests a strategy that reduces bandwidth and space consumption at the cost of some additional processing on-chip. Compression is one such strategy. Compression algorithms transform data to a more compact representation, but some computation is required to recover the original data.

Most techniques proposed for hardware texture compression are based on independently accessed tiles because textures must be randomly accessed. Commonly, textures are divided into small tiles and each tile is compressed using a fixed rate lossy mechanism, such as vector quantization. These schemes offer good compression rates but often have poor image quality. Image compression standards exist that support both good compression rates and high image quality. The industry standards, PNG [Wor96] and JPEG2000 [CSE01], also support lossless compression modes. In order to be transparent to applications, we would

also like to support a lossless compression scheme for texture data. Unfortunately, standard lossless image compression schemes usually do not support efficient random access to fine-grained pixel data directly from the compressed representation. JPEG2000 supports indexing and tiling, but this is meant for coarse-grained access. Fine-grained random access is complicated by the fact that the compression rate *must* vary for lossless schemes. In addition to a good compression rate, an ideal scheme should have good cache performance (and bandwidth reduction) under typical random access patterns.

Cache compression systems have been developed to alleviate the gap between processor and memory performance in the area of CPU design. In order to support general-purpose applications, CPU cache compression schemes must be lossless. There are two types of CPU cache compression: data cache compression and instruction cache compression. Data cache compression schemes for CPUs must support both read and write operations. In contrast, instruction cache compression is simpler since the instruction cache is read only, although they must still support random access. Since texture caches are also read-only, our work builds upon ideas in CPU instruction cache compression.

We present a block-oriented lossless texture compression algorithm based on a variable-bitrate block-based difference scheme combined with an index. This enables both lossless compression and random access.

Our compression algorithm uses a B-tree index for fast random access of variable-length compressed data. Each leaf of the B-tree index consists of pointers to fixed-size blocks that include a variable number of compressed data chunks and a table of address offsets for random access to these chunks. The average of the memory overhead for the indices was relatively small: 1.7% of the original texture sizes in our test suite. Index nodes are handled as fixed-size blocks of the same size as the compressed data blocks, enabling a simple $O(1)$ free-list allocation scheme with low overhead and no external fragmentation. The block-oriented nature of this data structure also permits efficient access to DRAM. Our compression algorithm achieves significant compression, reducing the sizes of the textures in our test suite to between 6% and 95% of their original sizes.

In addition to developing a compression algorithm, we also designed a cache architecture to realize it in hardware. Our cycle-accurate simulator shows that this cache architecture can reduce external bandwidth requirements substantially while not significantly affecting latency.

Our main contributions are the following:

- a texture representation that enables both lossless compression and random access;
- a block-oriented indexing scheme that permits simplified memory allocation;
- a cache architecture for implementing this scheme in hardware, based on a tile cache, an index cache, and a cache for compressed tiles; and
- an analysis of not only the static compression ratio but also the bandwidth consumption and latency using a cycle-accurate simulation.

2. Related Work

Most texture compression schemes are based on tiles because textures must handle random access. In such a scheme, textures are divided into small tiles and each tile is compressed at a fixed rate using some lossy approximation. S3TC [INH99] is one of the most popular tile based texture compression schemes. It is based on a BTC/CCC scheme proposed by Knittel et al [KSKS96]. In S3TC, textures are divided into 4×4 pixel tiles and each tile is compressed to 64 bits. Two base colors are stored in 16 bits each. Each pixel then stores a two-bit index into a local color set that includes the two base colors and two linearly interpolated colors based on the base colors. One of S3TC's shortcomings is its color limitation: only four colors can be used for each tile.

Ström et al. present a tile base scheme called iPACKMAN [SAM05]. The texture is split into 2×4 or 4×2 tiles. Each tile has a base color and 8 luminance values. The final color is the sum of the base color and luminance values. In order to increase the resolution of base colors, they combined two 2×4 or 4×2 tiles into 4×4 tiles. The compressed size of the 4×4 tile is 64 bits, the same as S3TC. Beers et al. use a vector quantization (VQ) approach to compress textures [BAC96]. They can compress to a rate as low as 1 or 2 bpp. A VQ approach accesses a codebook and uses an index map lookup to determine which color to use for each pixel. The Talisman architecture [TK96] used a fixed-rate lossy scheme similar to JPEG that could achieve compression rates of up to 15:1 on 8×8 texel blocks, but had a decompression latency of 100 cycles.

Although these texture compression schemes offer good compression ratios, they cannot support lossless compression because of their fixed rate compression constraint. In contrast, the compression ratio of lossless compression depends on the entropy of the data. There are several methods that can save space by discarding unused parts of the texture and scaling down other parts [KE02, CH02, LDN04] that can be implemented using existing shader and dependent texture read functionality. These can be considered simple variable-bitrate compression schemes but only work for a limited class of textures and suffer from fragmentation. Goris et al. patented an indirection-based retrieval method for variable-rate compressed texture data [GA99], but the details of how to implement decompression were not presented.

The Talisman architecture supported rendering to and from compressed texture tiles [TK96]. They used an architecture that was similar to ours in some respects, including a cache for both compressed and uncompressed blocks.

Barkans and Lengyel [Bar97, LS97] discuss how this architecture can be used for high-quality rendering. However, their compression scheme was lossy, took significant chip area, and had high latency. Our major contribution over this prior work is the introduction of an indexing scheme that can transparently deal with *variable-rate* compressed data. We focus on lossless compression but our architecture could be easily extended to support variable-rate lossy compression.

Yee proposed a B-tree indexing method [Yee04] for lossless texture compression based on a wavelet transform and a mechanism to exploit the sparse nature of the non-zero coefficients. Unfortunately, like other sparse texture methods, this approach tends to suffer from internal fragmentation, and so compresses well only in special circumstances. However, the B-tree approach, while it requires the support of special hardware, also permits flexible and efficient memory allocation. We describe the details of B-tree indexing in Section 3 because our representation uses it. Our representation can compress images more generally than Yee's approach and the above sparse texture approaches, but also happens to compress sparse textures efficiently.

Several image compression standards have been established to compress images with both high compression ratios and high image quality, and with both lossy and lossless compression. Their basic strategies are first a transformation that decorrellates the values that need to be stored, followed by variable-length coding. Lossy schemes also include a quantization step. JPEG [PM93] uses the Discrete Cosine Transform (DCT) followed by quantization and Huffman coding. JPEG2000 uses a wavelet transformation and arithmetic coding. For lossless compression, a special wavelet transform is used that can be exactly inverted. For lossy compression, the wavelet coefficients can be quantized. Unfortunately, these image compression standards were not designed to support fine-grained random access to pixel data directly from the compressed format.

Several CPU cache compression systems have been developed for both data and instruction caches, based on modifications of standard sequential compression algorithms. Ziv-Lempel proposed a simple memory compression algorithm [ZL77] which builds a dictionary from an LRU stack model. If a current byte is found in the dictionary, the byte is encoded by the index of the dictionary entry. If a current byte is not found in the dictionary, the byte is added to the index. In order to improve performance when applying this idea to cache compression, X-Match [KGJ96] and WK [Kap99] process memory data units of four bytes, and use a partial hit where the current word can partly match a word in the dictionary. These data cache compression schemes support both read and write operations. However, the compressed size of modified data can be different from that of the original data. A data cache mechanism that supports both writes and compression must handle this problem and this leads to increased hardware complexity.

Instruction cache compression is simpler because it needs to support only read operations. Lekatsas et al. proposed a random access compression method for the instruction cache [LW99]. In this scheme, instruction codes are compressed by arithmetic coding and multiple compressed data chunks are packed into fixed-size storage blocks. A table of indices permits random access. Our scheme is similar to this approach.

3. B-Tree Indexing

We will now describe Yee's sparse texture approach [Yee04], because our representation is an elaboration of it. Yee's approach is based on B-tree indexing, a block-oriented data structure with interesting properties in this context.

We assume a 2-D texture with default value d as the target texture. The texture is divided into small uniform tiles. Tiles that have at least one texel with a non-d value are considered to be *occupied*. Unoccupied, or *void* tiles, are marked as "discarded". Each tile has a key k associated with it that can be computed from the (x,y) coordinates of its origin. We call the method of computing k the *index scheme*. Index schemes that conserve spatial coherence are preferred; for instance, the index can be the distance along the Hilbert curve or the Morton (Z order) curve. The Morton curve can be computed by bit interleaving and is especially suitable for hardware, and so is what we use here. This index scheme can also easily be applied to different dimensionalities.

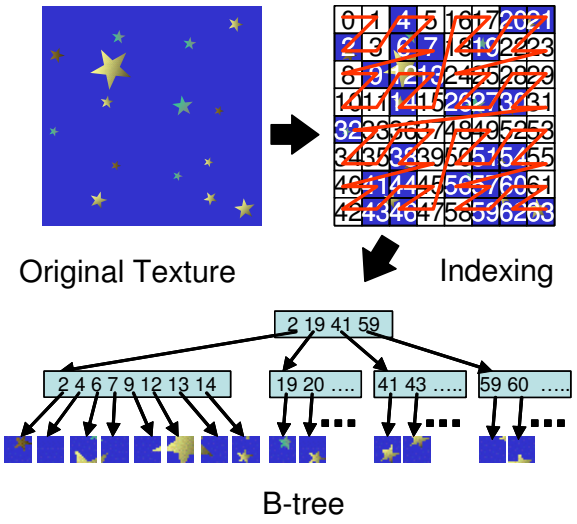

Figure 1: B-tree indexing

The top right of Figure 1 gives an example. Occupied tiles retain their original content and void tiles are represented as white quads. The numbers on the tiles are the keys generated by the Morton curve index scheme, while the red line indicates the order of the tiles.

The B-tree is a standard block-oriented data structure. In our implementation, data and index blocks are the same size.

This greatly simplifies memory allocation and eliminates external fragmentation since blocks can be allocated from anywhere in memory. Also, DRAM is most efficient when used to access large blocks using burst transfers.

Blocks are referred to by pointers p_i, each of which gives the high-order bits of their starting address in memory. Several (k_i, p_i) pairs are stored in each internal node of a B-tree, with keys in sorted order. To find a data block, its (x, y) location is first converted to a key value k. We then recursively search the tree from the top, using the intervals between keys at each level to establish the range of keys indexed by the next level down. If a data block is not found for a certain key, the background value d is returned.

Every internal node stores a maximum of N key-pointer pairs. Non-root internal nodes will never have fewer than $N/2$ key-pointer pairs. To insert a new data block, its key-pointer pair is inserted starting at the bottom of the tree, after first searching recursively for the insertion point. If there is space in the lowest level index block, the new key-pointer pair is simply inserted in sorted order. If there is no free space, the index node is split in two. Half the key-pointer pairs are then moved to a new index node and a pointer to the new index node inserted into the parent of the original index node. This process proceeds recursively, if necessary, to the root of the tree. If the root must be split, the depth of the tree increases by 1. Note that splits are rare, on average occurring only every $N/2$ insertions, and splits to higher levels are progressively rarer.

Both lookup and insertion are simple enough to implement in hardware. The insertion algorithm described results in a balanced tree, so it is possible to render directly into this representation while maintaining efficient lookup. Deletion is harder if we wish to maintain a balanced tree. The simplest solution to this is to not support deletion, except of the entire tree. Also, after random insertions the B-tree may only use, in the worst case, half of the space in its internal nodes. After rendering, an optional compaction pass can be used reencode the tree in linear time with all but one index node completely full [MM02].

The bottom of Figure 1 gives an example B-tree. Every node has up to eight key-pointer pairs in this example. In practice, a larger fan-out and a larger block size would be used, leading to a shallow tree depth and a low overhead for storing the index. The size of the B-tree is based on the number of key-pointer pairs stored in it, not the texture resolution. The size of the index is based only on the memory needed for non-background parts of the texture.

4. Proposed Method

Although Yee's B-tree indexing achieves lossless compression for a certain limited class of images, her representation cannot compress textures that have no void tiles, and suffers from internal fragmentation: tiles with even a single non-background pixel are considered occupied. We solve this problem by compressing occupied tiles using a variable-bitrate differencing scheme. We pack multiple compressed tiles into a larger fixed-size leaf block along with a table of offsets to enable random access.

4.1. Variable Bitrate Difference Compression

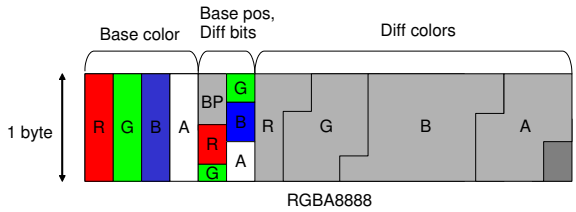

Figure 2: Differencing scheme and bit packing.

The texture image is divided into 4×4 tiles. The top of Figure 2 shows what is included in a compressed occupied tile. The *base color* is the color at the position specified by the 2+2 bit *base pos* value. *Diff colors* represent differences between the base color and the 15 other texels of the tile. The *diff bits* field, 3 bits per component, defines the size of the diff colors, from 1 to 8 bits. For example, if all of the red diff colors are within a range from -8 to 7, the required size to express the red diff colors is 4 bits and the value of the diff bits field is 3. If the differences require 8 or 9 bits to be represented, we store the original 8-bit values instead of the difference. The compressed tile is stored as byte data. Therefore, at most 7 bits are unused (internal fragmentation). If we interpret the diff colors as indices to a codebook, we can also extend this scheme to lossy compression with minor changes.

There are two special cases in this compression scheme. The first one is when all colors are the same. In this case, the compressed tile is completely defined by its base color. The second special case is when size of the "compressed" representation exceeds the original size. In this case, the tile is stored in uncompressed form. The decompression unit detects these two special cases by the size of the tile data as given by the offset table, described later.

The most significant advantage of our scheme is its simplicity. All of the diff bits for one component have the same bit length. All 16 final colors in a tile can be calculated in parallel which results in a very low decompression latency. Standard variable-length coding schemes, such as Huffman or arithmetic codes, can also be used for tile compression, and would be able to adapt better to the entropy distribution of non-image data, but would have higher decompression latency. We will show later that performance is in fact not especially sensitive to decompression latency, so more complex tile compression schemes are suitable targets for future research.

4.2. Data structure

We will now describe the details of our data structure, and in particular, how our indexing scheme deals with the variable size of compressed tiles.

4.2.1. Leaf block

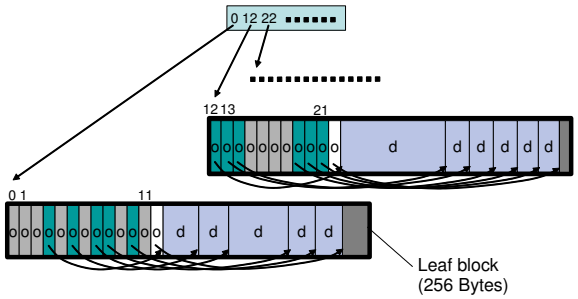

Figure 3: Leaf block. Leaf blocks include address offsets and multiple compressed data blocks.

Since we use lossless compression, the size of each compressed block depends on the block's content.

After compressing occupied tiles, we pack compressed tiles and address offsets into fixed-size *leaf blocks*, as in Lekatsas's compressed instruction cache design. Figure 3 shows an example based on the texture in Figure 1. The Z order (Morton curve) index scheme is used for this example. While Yee's B-tree includes only the keys of occupied tiles, our B-tree index includes the keys of both occupied tiles and void tiles. Also, leaf blocks conceptually include all data tiles in sequence, whether void or not. Each potential tile data in a leaf block is indexed from a table that gives, for each key, an offset to the start address of the associated compressed tile data (shown as o values in Figure 3). The difference between one offset and the next is used to determine the length of each compressed tile's data. If two adjacent offsets are the same, it means that the associated data is zero length, and is interpreted as a void tile. The light grey offsets in Figure 3 have the same value as the offsets to their right. For example, the tiles for index 0 and 1 are void tiles and the tile for index 2 is a occupied tile, whose size is the difference between index 2 and 3. We need one additional offset to act as a sentinel, which is illustrated in white. We use the same size blocks for both the index blocks of the B-tree and leaf blocks. We have chosen a block size of 256 bytes, which means each offset is one byte.

Since we need index offsets, the total size of leaf blocks can exceed the original size of the raw image data if tiles are not compressed well. We use an uncompressed leaf block when necessary to avoid this situation. Uncompressed leaf blocks include only uncompressed tile data, without any offsets. For example, a 256 byte leaf block can hold 4 uncompressed 4×4 RGBA pixel tiles. If all 4 tiles are not compressed well, the sum of address offsets and tile data might be more than 256 bytes.

4.2.2. Index block

Yee's B-tree index block includes 32 key-pointer pairs in 256 bytes; the size of keys and pointers are 32 bits each. We use 20-bit keys and 24-bit pointers instead. Since the tile size is 4×4, keys need to be 20 bits long in order to support 4k×4k textures. Pointers only need to be 24 bits long for a 4GB memory address space, since they only need to refer to 256-byte block boundaries. Pointers could of course be smaller for smaller address spaces.

We also pack into an index block a single-bit flag for each child. The *compressed block flag* indicates whether the child is a compressed leaf block or not. If the child is not a leaf block, this flag is ignored. The tree is always balanced, so we store the depth as a global value associated with each texture. We also store a default color with each texture.

An index block includes 45 key-pointer pairs and flags. The total size is 253 bytes and one bit: $253.125 = (45*(20+24+1))/8$. Of the remaining 23 bits, 20 can be used for a sentinel key to make searching more efficient.

4.3. Cache Architecture

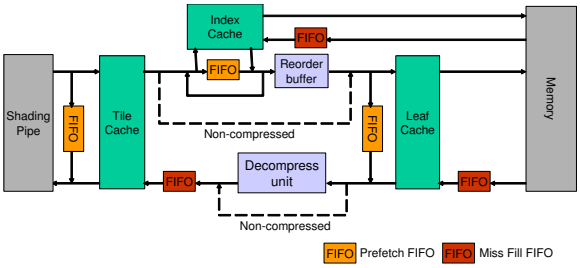

Figure 4: Cache architecture.

We will now describe a cache architecture to support the proposed representation. The block diagram of our architecture is shown in Figure 4. It consists of three cache units: the *tile cache*, the *index cache*, and the *leaf cache*. All three cache units use the LRU replacement policy. Since our data structure is based on blocks of the same size, all data transactions between the caches and the main memory are standardized burst transfers. FIFOs and pipelined memory accesses are used to support multiple in-flight requests.

If texture data is uncompressed, the decompression unit can be bypassed. It is also possible to bypass the index unit if uncompressed texture data is allocated from a contiguous address space. The latency of the decompression unit (when used) is 2 cycles: one cycle for unpacking the bits for a tile and another for parallel addition.

The rasterizer outputs one fragment per cycle with up to 8 texel reads for tri-linear filtering. The number of misses that can occur in each cache is limited to one miss per cycle. Also, the number of cache blocks that can be committed to the cache from the FIFO is similarly limited to one block per

cycle. We chose DDR2 SDRAM [JED05] to model our main memory: On the assumption that a fragment clock cycle is 400 MHz, the memory model's setup latency is 20 cycles and it requires 32 cycles to transfer each 256 byte block.

4.3.1. Tile Cache

The tile cache stores decompressed texture tiles. There is no difference in the tile cache between void tiles and occupied tiles. The tile cache receives requests from shader/filtering units and returns texel data.

The tile cache block size is the same as tile data's size, which in our implementation is 64 bytes (4×4 pixels). If a cache miss occurs, the tile cache sends a miss request to the index cache, which caches the index blocks that are used to determine the address of the leaf block holding the desired tile data. The requested leaf blocks are provided by the leaf cache. The required tile data is decompressed before it arrives at the tile cache.

4.3.2. Index Cache

The task of the index cache is to support efficient B-tree index traversal. The index cache stores recently accessed B-tree index blocks. It receives requests from the tile cache and searches for the requested key by descending the texture's B-tree. If required B-tree index blocks are missing from the index cache during the descent, the index cache must request the blocks from main memory. In our implementation the index cache block size is 256 bytes.

The B-tree search provides a pointer to the leaf block containing the target tile data. The index cache also calculates the offset index of the leaf block for the tile data. The pointer and the offset index are sent to the leaf cache.

The index cache requires a reorder buffer to support multiple outstanding recursive references to index blocks. When a request comes to the index cache, it must reserve a slot for the request in the reorder buffer. If the reorder buffer is full, the index cache waits until the buffer can reserve a slot. After the request is granted, the cache recursively accesses index blocks, doing a parallel search on each block for the requested key. If the target leaf blocks' address is not found the request is sent to the prefetch FIFO and evaluated again. Once target leaf block's address has been found, the address and the calculated offset index are sent to the reorder buffer. The reorder buffer releases the oldest slot's data to the leaf cache when it is completed. The index cache only has to handle requests that miss in the tile cache and so does not need to process an index request on every cycle.

4.3.3. Leaf Cache

The leaf cache holds compressed tile data in the form of leaf blocks. If a cache miss occurs, a request is sent to main memory. The leaf cache receives from the index cache an address and an offset index and extracts the appropriate compressed tile data, which is decompressed and sent to the tile cache. If the requested tile is a void tile, the decompressor generates a tile initialized with the default color d recorded for the current texture. The default color is loaded into the decompressor as part of binding the texture.

4.3.4. FIFOs and Memory Units

The three caches have a standard prefetch mechanism similar to what Igehy et al. described [IEP98]. Since our architecture guarantees in-order requests, each cache has a FIFO for requested data that provides support for a limited number of outstanding requests and pipelined access to memory. We show the effects of varying FIFO sizes in Section 5.2.2. Memory units also return requested data in-order. Therefore, all of the requested data returns in-order and the cache overall does not need reorder buffers for the data.

5. Results

We have computed both the static compression rates for a suite of images and the latency and the external bandwidth requirements for a suite of scenes. Test textures in our suite are shown in Figure 5. The Water, Stars1, Stars2, and Building2 textures are typical of tileable image textures. The Building1, Car1, and Car2 textures are texture atlases for models of our test scene suite. The Water, Stars1, Stars2, and Car2 are 512×512 and The Building1, Building2, and Car1 are 1024×1024. The pixel format for each texture is RGB, 24 bits per pixel.

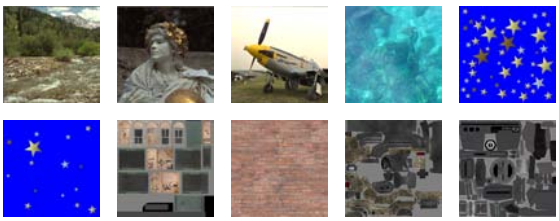

Figure 5: Test suite. From top left to right: Kodak13, Kodak17, Kodak20, Water, Stars1, Stars2, Building1, Building2, Car1, and Car2.

We also applied our scheme to a photographic image suite by Kodak. These images are non-square, and therefore the top left 512×512 part of the images were used.

5.1. Compression Ratio

We present compression ratio results in Figure 6. The Kodak13, Kodak17, and Kodak20 are the worst, median, and best of the results of Kodak images respectively. We also tested the use of arithmetic coding to compress color differences instead of bit packing. Although bit packing is much simpler to implement and has lower latency than arithmetic coding, the compression rates were almost the same. Both

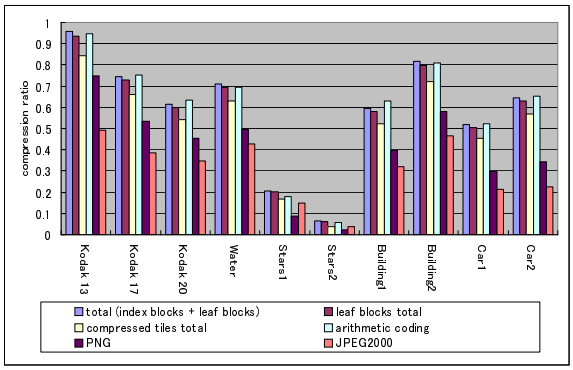

Figure 6: Compression ratio.

JPEG2000 and PNG results were better than our compression scheme, and JPEG2000 was generally better than PNG. The average of the compression ratio differences between PNG and our scheme was 0.209. Comparing to only the compressed tile total, the average is reduced to 0.125. The overhead for enabling random access is mainly caused by leaf blocks' unused area (internal fragmentation). In terms of compression ratio, the average overhead due to index blocks is only 1.7%, and indexing also provides the substantial advantage of flexible memory allocation.

The Stars1 and Stars2 textures are compressed very well because of their sparseness: 22% and 6% of the original data's size. These are also the only two textures that PNG compressed better than JPEG2000. Texture atlases, represented by the Building1, Car1, and Car2 examples, compressed to 51% to 64% of their original sizes. Like sparse textures, texture atlases can have large void areas.

5.2. Simulation

We implemented a cycle-accurate simulator of our architecture inside Mesa. The test scenes used for simulation are shown in Figures 7 and 8. The Quad scene consists of a single quad and is used for theoretical bandwidth testing. The Building and Car scenes represent typical models using texture atlases. The Multi4 scene simulates a typical scene with multiple objects and shaders using multitexturing. In this scene, four textures are read for each fragment. The same texture coordinates are used for each texture, but the image data itself is separate. In order force cache reloading, the buildings and cars are drawn in alternating order.

Our system supports MIP-mapping. Each MIP-map level has a corresponding compressed texture. Our tests take 8 samples for every trilinearly interpolated filtered texture sample required by a shader (4 from each level). Our rasterizer scans triangles in a spatially coherent 'Z' (Morton curve) order. We used Kodak17, Water, and Stars2 textures for the Quad and Teapot scenes. The Building1 and Building2 textures are used for the Building scene and the Car1 and Car2 textures are used for the Car scene. We simulated these scenes with both compressed and uncompressed textures. All uncompressed texture data was stored in tiled format but over a single contiguous address space. To replicate the behaviour of a conventional cache architecture, the tile cache can send miss requests for uncompressed textures directly to the leaf cache, bypassing the index cache. Also, uncompressed tiles bypass the decompression unit. In this case, the leaf cache acts as an L2 cache for the tile cache. For a comparison base case, we also ran the simulation using a conventional architecture built using a larger unified cache using the same total area as our proposed system.

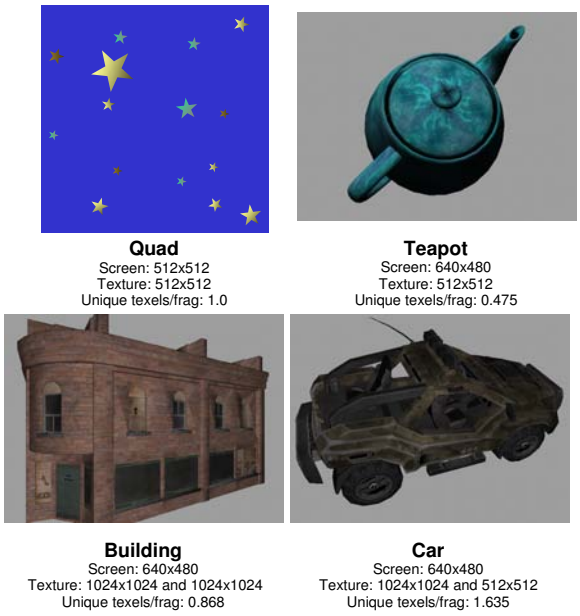

Figure 7: Basic test scenes.

Figure 8: The multiobject multitexturing test scene.

5.2.1. Cache Sizing

There are three caches that are sequentially accessed in order from the shader unit side to the memory side: (1) tile cache, (2) index cache, and (3) leaf cache. The sizes of the index cache and leaf cache do not affect the tile cache's miss rate. Likewise, the size of the leaf cache does not affect the

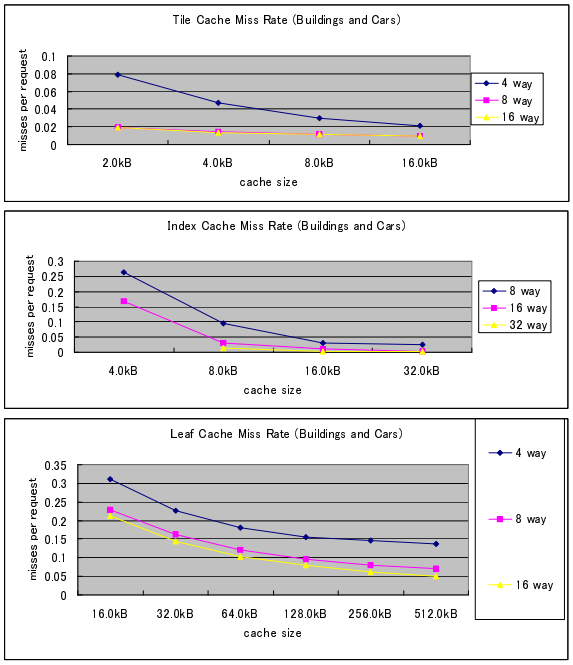

Figure 9: Effects of varying cache sizes and types. Cache miss rate of the tile cache, the index cache (tile cache: 2way-2kB), and the leaf cache (tile cache: 2way-2kB, index cache: 4way-4kB)

index cache's miss rate. We therefore tuned the sizes of the three cache units in order from the shader side. As we obtained each result, we fixed the cache size, and tuned the next level. We used the Car scene as our sample scene because its unique texel per fragment ratio was the largest.

The results of this process are shown in Figure 9. We have chosen 2-way/2kB, 4-way/4kB, and 2-way/16kB configurations for the tile cache, index cache, and leaf cache respectively. These values will be used for all of the following simulation results.

We can gauge the efficiency of our texture compression scheme from the leaf cache's miss rate. In this scene, the compression ratios of the Car1 and Car2 textures are 48% and 56% respectively. If the textures are compressed, the miss rates are reduced almost the same as the compression ratios.

5.2.2. FIFO size

	Tile Cache	Index Cache	Leaf Cache
Prefetch FIFO	128	1	32
Miss Fill FIFO	2	1	2

Table 1: FIFO size parameters.

Figure 10 shows the effects of varying prefetch FIFO depths on the Building and Car scenes with texture compression enabled. For each graph, one FIFO size is varied while the others are held fixed (at the values specified in Table 1). The results are reported in fragments per cycle. The main reason why the fragments per cycle do not approach 1.0 is bandwidth. As long as the bandwidth is not large enough to transfer consecutive requested blocks, prefetch FIFOs can not hide pipeline stall. As the bandwidth gets larger, the fragments per cycle tends to 1.0 as in Figure 13.

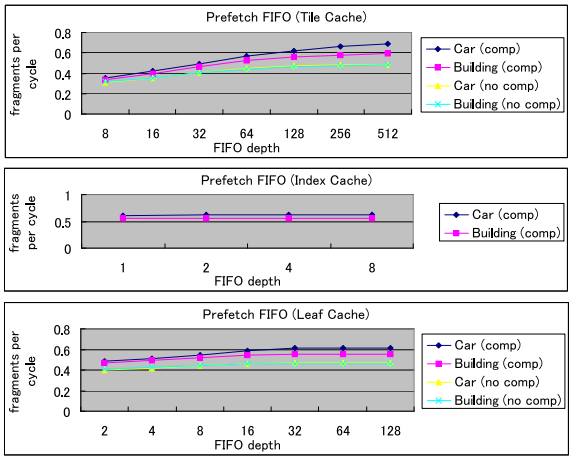

Figure 10: The effects of varying prefetch FIFO depth.

The index cache's prefetch FIFO size does not effect the performance because the miss rate of this cache is relatively low, about 0.01 per request. A depth of 1 for the index cache FIFO gives a very simple implementation, since only one request needs to be handled at a time. Varying the miss fill FIFO depth hardly affects the performance. Therefore, we set the depth to 2 as in Table 1.

5.2.3. Latency

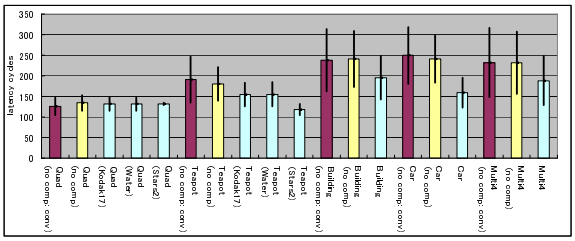

Figure 11: Latencies of the test scenes.

Figure 11 shows the latency results for each fragment from generating the fragment to finishing texture filtering. Bars and lines represent average latency and standard deviation. To test Multi4 with its larger working set (four texture accesses per shader) the cache sizes and associativities were set to be four times as large as for the single texture scenes.

The dark bars report results for a conventional architecture with a single unified cache. For the conventional architecture, we set the cache size to 32kB and the tile size to 256 bytes. This conventional architecture actually has 10kB more memory than our architecture. We estimated 10kB SRAM to be equivalent in area to the additional control logic, FIFOs, and adders required for our architecture.

In case of Quad, the latencies are almost the same and texture compression actually reduced the standard deviations. In the other three scenes, compression reduced both the average latency and its variance. Our low latency decompression scheme does not make the average latency significantly longer, and in fact seems to have improved it (although our test suite is small). There also was not a significant difference between the results of two types of uncompressed textures. This means our architecture can be used for uncompressed textures as well as compressed textures.

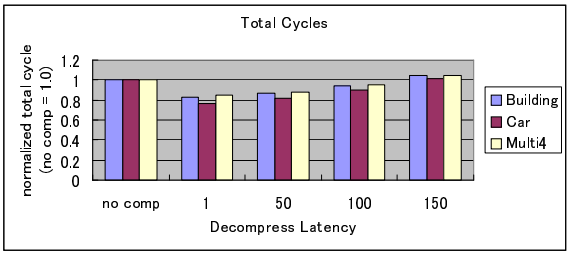

Figure 12: Effects of varying the decompression latency.

Although our decompression scheme has very low latency, we were interesting in testing the impact of decompression latency since our architecture can easily be extended to support other approaches to compression. Figure 12 shows the effects of artificially increasing decompression latency while rendering the Building and Car scenes. Unless the decompression latency is 150 cycles or more, compression still led to an overall improvement in latency.

5.2.4. External Bandwidth Consumption

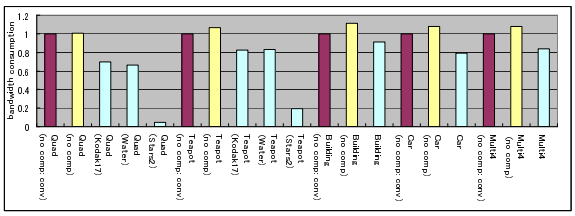

Figure 13: External bandwidth consumption for all scenes.

External bandwidth consumption results are shown in Figure 13. The parameters for the conventional architecture are the same as in Section 5.2.3.

The numbers are normalized by the conventional architecture's result. Our architecture demonstrates significantly better bandwidth behaviour than the conventional architecture on all of the test scenes and textures. Since our texture compression is lossless, our architecture provides exactly equivalent quality at a lower bandwidth requirement, in some cases, over an order of magnitude lower.

The Quad scene results show almost the same bandwidth reduction as its static compression ratio (Figure 6). Due to the alignment between the rasterization order and the texel storage order in this scene, most texels of each cache block are used if they are loaded into the compressed cache, and so this scene gives the best possible performance. In contrast, the bandwidth reductions in the Teapot, Building, Car, and Multi4 scenes are more typical, because each compressed leaf block is not always fully utilized.

5.2.5. Index Cache Working Set

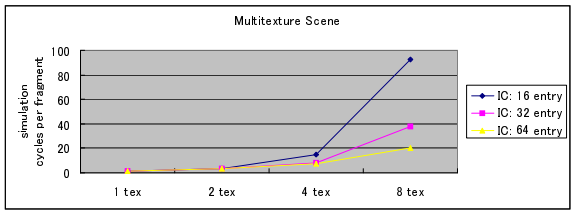

Figure 14: The effects of varying the number of textures and index cache entries.

According to Section 5.2.2, the index cache does not need a prefetch mechanism because of its relatively low miss rate. However, since it must be accessed recursively during an index lookup, once its capacity is exceeded, the overall system performance will get significantly worse. In a multitexturing environment in particular, each texture will require a certain number of blocks of its index to be resident in the index cache.

Figure 14 shows the effects of varying the number of textures and index cache entries on the Multi4 scene. The same texture coordinates are used for each texture. The heights of the index for this scene's 1024×1024 textures is four, including three index levels and one leaf block level. Only the index blocks are stored in the index cache. Based on the empirical results, the working set in the index cache for these textures appears to be between 3 and 4 index blocks per texture, or 6 to 8 taking MIP-mapping into account. For example, performance with a 16-block index cache is excellent for two MIP-mapped textures, but degrades when we attempt to use it with four. Note, however, that such performance degradation is also a problem with conventional architectures when multitexturing increases the working set size. We can also expect the working set of a texture in the index cache to only increase logarithmically with the entropy of the texture, and so can trade off texture entropy for number of textures.

6. Conclusions

We have presented a block-oriented lossless compressed texture representation based on a variable-bitrate differencing scheme. Our main contribution is a B-tree indexing scheme that enables random access and efficient memory management and allocation. Textures in our test suite compressed to between 6% and 95% of their original sizes. We also proposed and analyzed a cache architecture designed to support our representation. Our cycle-accurate simulation of this cache architecture showed that our compression scheme reduces the external bandwidth requirements as well as the data sizes.

In future work, exploring better compression schemes would be useful. Our tile compression scheme is based on the assumption that adjacent pixels have similar values. Though this assumption often works for image data, it is not good assumption for general data, as in GPGPU applications, where compression and flexible memory management would be especially useful. Application to variable-bitrate *lossy* compression schemes would also be interesting.

References

[ASMW97] ANDERSON B., STEWART A., MACAULAY R., WHITTED T.: Accommodating memory latency in a low-cost rasterizer. In *Proc. Graphics Hardware* (1997), pp. 97–101.

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. *Proc. SIGGRAPH* (1996), 373–378.

[Bar97] BARKANS A. C.: High quality rendering using the Talisman architecture. In *Proc. Graphics Hardware* (1997), pp. 79–88.

[CH02] CARR N. A., HART J. C.: Meshed atlases for real-time procedural solid texturing. *ACM Trans. on Graphics 21*, 2 (Apr. 2002), 106–131.

[CSE01] CHRISTOPOULOS C., SKODRAS A., EBRAHIMI T.: The JPEG 2000 still image coding system. *IEEE Signal Processing Mag. 18* (Sept. 2001), 336–58.

[GA99] GORIS A. C., ALCORN B. A.: Data structure for efficient retrieval of compressed texture data from a memory system. US Patent 6,243,081, 1999.

[HG97] HAKURA Z. S., GUPTA A.: The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th annual international symposium on Computer architecture* (1997), ACM Press, pp. 108–120.

[IEH99] IGEHY H., ELDRIDGE M., HANRAHAN P.: Parallel texture caching. In *Proc. Graphics Hardware* (1999), pp. 95–106.

[IEP98] IGEHY H., ELDRIDGE M., PROUDFOOT K.: Prefetching in a texture cache architecture. In *Proc. Graphics Hardware* (1998), pp. 133–142.

[INH99] IOURCHA K., NAYAK K., HONG Z.: System and method for fixed-rate block-based image compression with inferred pixels values. US Patent 5,956,431, 1999.

[JED05] JEDEC: *DDR2 SDRAM Specification*, 2005. Specification document, available from http://www.jedec.org/.

[Kap99] KAPLAN S.: *Compressed Cacheing and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, 1999.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proc. Graphics Hardware* (2002), pp. 7–15.

[KGJ96] KJELSO M., GOOCH M., JONES S.: Design and performance of a main memory hardware data compressor. In *Proc. the Euromicro Conference* (1996), pp. 422–430.

[KSKS96] KNITTEL G., SCHILLING A. G., KUGLER A., STRASSER W.: Hardware for superior texture performance. *Computers & Graphics 20*, 4 (July 1996), 475–481.

[LDN04] LEFEBVRE S., DARBON J., NEYRET F.: *Unified Texture Management for Arbitrary Meshes*. Tech. Rep. RR-5210, INRIA, May 2004.

[LS97] LENGYEL J., SNYDER J.: Rendering with coherent layers. In *Proc. SIGGRAPH* (1997), pp. 233–242.

[LW99] LEKATSAS H., WOLF W.: Random access decompression using binary arithmetic coding. In *Data Compression Conference* (1999), pp. 306–315.

[MM02] MA V. C. H., MCCOOL M. D.: Low latency photon mapping via block hashing. In *Proc. Graphics Hardware* (2002), pp. 89–98.

[PM93] PENNEBAKER W. B., MITCHELL J. L.: JPEG still image data compression standard. Van Nostrand Reinhold, 1993.

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *Proc. Graphics Hardware* (2005), pp. 63–70.

[TK96] TORBORG J., KAJIYA J. T.: Talisman: Commodity realtime 3D graphics for the PC. In *Proc. SIGGRAPH* (1996), pp. 353–363.

[Wor96] WORLD WIDE WEB CONSORTIUM: *PNG (Portable Network Graphics) Specification Version 1.0*, 1996. Specification document, available from http://www.w3.org/TR/REC-png.

[Yee04] YEE W. M.: *Cache Design for a Hardware Accelerated Sparse Texture Storage System*. Master's thesis, University of Waterloo, 2004.

[ZL77] ZIV J., LEMPEL A.: A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337–343.