Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

Floating-Point Buffer Compression in a Unified Codec Architecture

Jacob Ström[1], Per Wennersten[1], Jim Rasmusson[1,2], Jon Hasselgren[2], Jacob Munkberg[2], Petrik Clarberg[2], and Tomas Akenine-Möller[2]

[1]Ericsson Research [2]Lund University

Abstract

This paper presents what we believe are the first (public) algorithms for floating-point (fp) color and fp depth buffer compression. The depth codec is also available in an integer version. The codecs are ***harmonized****, meaning that they share basic technology, making it easier to use the same hardware unit for both types of compression. We further suggest to use these codecs in a* ***unified codec*** *architecture, meaning that compression/decompression units previously only used for color- and depth buffer compression can be used also during texture accesses. Finally, we investigate the bandwidth implication of using this in a* ***unified cache architecture****. The proposed fp16 color buffer codec compresses data down to 40% of the original, and the fp16 depth codec allows compression down to 4.5 bpp, compared to 5.3 for the state-of-the-art int24 depth compression method. If used in a unified codec and cache architecture, bandwidth reductions of about 50% are possible, which is significant.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture - Graphics processors; E.4 [Data]: Coding and Information Theory - Data compaction and compression;

1. Introduction

To increase performance for graphics processing units (GPUs), bandwidth reduction techniques continue to be important [AMN03]. This is especially so, since the yearly performance growth rate for computing capability is much larger than that for bandwidth and latency for DRAM [Owe05]. The effect of this can already be seen in current GPUs, such as the GeForce 8800 architecture from NVIDIA, where there are about 14 scalar operations per texel fetch [NVI06]. Algorithms can often be transformed into using more computations instead of memory fetches (e.g., instead of evaluating, say, $\sin(x^2+y^2)$ as a lookup in a precomputed texture, this expression could simply be evaluated by executing the relevant instructions). However, at some point, the computation needs are likely to be satisfied, and then the GPU will be idle waiting for memory access requests. Also, the brute force approach of duplicating the number of memory banks and accessing these by using more pins on the chip may not be possible in the long run. Hence, we argue that memory bandwidth reduction algorithms is a field of research worthy of more attention.

There are many types of bandwidth reduction techniques, but here we will focus mostly on algorithms related to some form of compression on the pixel or fragment level. Other techniques are mentioned in Section 3. The two major compression techniques for GPUs are *buffer compression* [HAM06, RHAM07] and *texture compression*, introduced in 1996 [BAC96,KSKS96,TK96]. Since buffers, such as color, depth, and stencil, are created during rendering using both reads and writes, these algorithms must be rather symmetric in that compression and decompression are performed in similar amounts of clock cycles. Furthermore, the majority of these algorithms are exact, i.e., lossless, even if there are exceptions for error-bounded color buffer compression [RHAM07]. For lossless buffer compression, there must always be a fallback to sending uncompressed data so that all blocks can be processed in the system. Since random access is required, these algorithms do not reduce storage, only bandwidth usage is reduced when transmitting buffer data. One or a few bits are stored on-chip or in a cache to indicate whether the block is compressed or not, and this memory is called the tile table. In recent papers, there are new buffer compression algorithms and surveys for depth buffer compression [HAM06] and for color buffer compression [RHAM07]. As a special mode in buffer compression techniques, it is also possible to implement a fast clear operation [Mor00], which is a type of compression. When a buffer clear is requested, each block of pixels can flag (in the tile table) that the block has been cleared. When such a block needs to be accessed, the flag is first checked, and it is discovered that the block is cleared, and hence there is no need to read the block from external memory. Instead, all the pixels in the block are simply set to a “clear”-value. This makes buffer clears inexpensive.

The two most widely adopted texture compression schemes are S3TC, also known as DXTC [MB98], which has seen widespread adoption both for OpenGL and DirectX, and ETC [SAM05], which has been adopted in the OpenGL ES API, targeting mobile devices. In contrast to buffer compression techniques, texture compression algorithms can be asymmetric so that the compression phase can take much longer than the decompression. The reason for this is that textures usually are read-only data, and hence, compression can be done in a preprocess. It is only when the GPU accesses the texels that decompression needs to be done, and hardware for this must be fast and of relatively low complexity. To simplify random access, most texture compression schemes compress to a fixed rate, e.g., 4 bits per pixel. Note that texture compression actually reduces the storage needs as well as the bandwidth usage. An exception to the fixed rate rule is presented by Inada and McCool [IM06] who use B-tree indices to implement a lossless variable bit rate texture compression system. Rendering to such textures would be possible, which should mean that the algorithm could be used for buffer compression as well.

Overview In this paper, we present two new codecs for fp color buffer compression and fp depth buffer compression. We have not seen any previous work in either of these two fields before. Whereas these two new algorithms give rise to bandwidth savings in their own right, the paper also investigates how different types of hardware architectures influence these savings.

OpenEXR [Ope08] is a format for fp image compression, and while it cannot be used for buffer compression as is, we have created a modified version of one of its codecs to benchmark our fp color buffer compression algorithm.

In OpenGL, the framebuffer object extension `EXT_framebuffer_object` allows render-to-texture to be performed without data being copied. For instance, it is possible to create a shadow map by rendering to a depth buffer, and later render from that depth buffer without having to first copy the data to a texture. If depth buffer compression is used, and the texturing unit cannot decode the compressed shadow map, it is necessary to first decompress the data before texturing takes place, which may be as costly as copying the data. Alternatively, compression of the shadow map can be turned off, which is approximately equally bad. Therefore, the natural next step is to allow rendering from textures irrespectively of what compression format the data is stored in. This paper refers to this as a *unified codec* architecture.

If any buffer/texture can use any compression format, it helps if the different formats are based on the same basic techniques, so that a hardware unit can decompress several formats with little extra logic. Such compression formats are denoted *harmonized codecs* in this paper.

In CPU architectures, there are systems which have separate data and instruction caches, while other systems use unified caches. Sometimes, the L1 caches are separate and the L2 cache unified. Separate caches have the advantage that instruction caches, being read-only, can be made simpler than data caches, which are read-modify-write. Unified caches, on the other hand, have the advantage that a large part of the cache can be used for instructions if instruction traffic is high and data traffic low, and vice versa. In this paper, we investigate what happens to memory bandwidth usage if the texture caches (read-only) and the buffer caches (read-modify-write) in a graphics architecture are unified. This paper denotes this a *unified cache* graphics architecture.

Next, we will describe our new floating point codecs, followed by a description on what type of architecture they are tested in.

2. Compressing RGBA half Data

To the best of our knowledge, no (published) attempts have been made at color buffer compression for floating-point RGBA data, even though there are works in nearby fields such as texture compression of HDR textures [MCHAM06, RAI06], floating point data compression (not for buffers) [LI06], integer RGBA color buffer compression [RHAM07] and lossless fp image compression [Ope08].

Our method builds on the work of Rasmusson et al. [RHAM07]. We first divide the image into 8×8 blocks, and each block is given two bits in the tile table to indicate whether it is fast-color-cleared, uncompressed, compressed to 50% (2048 bits) of original size or compressed to 25% (1024 bits) of original size. Often, the destination alpha is not used, and therefore we assume it is 1.0 during compression. We also assume that the color components are positive, storing only the last 15 bits of the half. If a block violates either of these two assumptions, the uncompressed mode is used instead. The 8×8 block is further subdivided into four 4×4 sub-blocks. The idea is to first encode the red component separately, and then encode the difference between the green and red, and finally the difference between blue and green. This is a simple but efficient way of exploiting the correlation between the color channels, and hence an expensive YUV transformation is avoided, which lowers complexity.

The first pixel of the red component R_{11} (see left part of Figure 1) is stored directly using 15 bits. The rest of the pixels in the first row and column are predicted from the previous pixel as indicated by the arrows. For instance, R_{12} is predicted using $\hat{R}_{12} = R_{11}$, and instead of storing R_{12} directly, the prediction error $\tilde{R}_{12} = R_{12} - \hat{R}_{12} = R_{12} - R_{11}$ is calculated and stored. Hopefully, $\tilde{R}_{12}$ should be smaller and

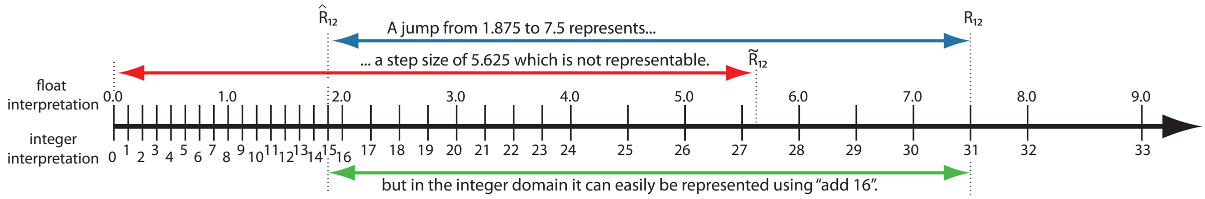


Figure 2: *Lossless prediction in the floating-point domain can be tricky, since differences between two representable floating-point numbers may not be representable. In the example, both the prediction $\hat{R}_{12} = 1.875$ and the target $R_{12} = 7.5$ are representable, but the difference $\tilde{R}_{12} = 5.625$ is not. Doing the prediction in the integer domain instead solves this problem.*

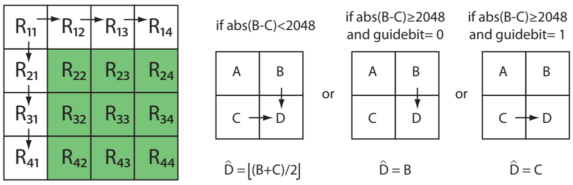

R_{11} R_{12} R_{13} R_{14}
R_{21} R_{22} R_{23} R_{24}
R_{31} R_{32} R_{33} R_{34}
R_{41} R_{42} R_{43} R_{44}
if abs(B-C)<2048
if abs(B-C)≥2048 and guidebit= 0
if abs(B-C)≥2048 and guidebit= 1
A B C D
or
A B C D
or
A B C D
$\hat{D} = \lfloor(B+C)/2\rfloor$
$\hat{D} = B$
$\hat{D} = C$

Figure 1: *Left: Arrows indicate prediction. For instance, R_{12} is used to predict R_{13}. Pixels marked in green can be predicted from one or two pixels, as shown to the right.*

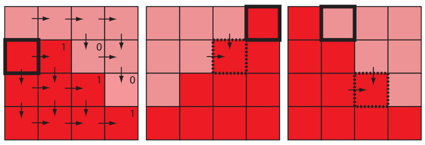


Figure 3: *Left: The restart value is marked with a box. Note how the guide bits make sure no prediction is made across the discontinuity. Middle: Sometimes a pixel (marked with dots) can only choose from two erroneous predictors. This is alleviated by rotating the block 90 degrees counter-clockwise, as shown to the right.*

simpler to code than R_{12}. However, the floating-point nature of halfs leads to problems. Since the density of floating-point numbers is not uniform (see Figure 2), the difference, $\tilde{R}_{12}$, between two floats may not be representable (red line). This problem is solved in the seemingly counter-intuitive but well documented way [LI06] of treating the halfs as integers. Due to the clever way floating-point numbers are defined, neighboring halfs will be interpreted as neigbhoring integers, as can be seen below the axis in the figure. Note that this trick works because we only compress positive halfs as described above. Following the green arrow, adding the difference 16 to the prediction 15 gives the correct result 31, or 7.5 if interpreted as a float. Doing the arithmetics in this "integer domain" also avoids costly floating-point operations, and since the compression is lossless, we will also get a correct handling of NaNs, Infs and denorms.

For the values marked with green in Figure 1, we can predict using the values that we already have encoded above and to the left of the pixel. As seen to the right in Figure 1, we can predict the value D using A, B and C. Color buffer compression differs from other image compression in that there may be an unnaturally sharp color discontinuity between a rendered triangle and pixels belonging to the background or to a previously rendered triangle of a very different color. In order to avoid doing prediction across such discontinuity edges, we propose the use of *guide bits*. If the difference between B and C is larger than a threshold, one bit is used to indicate whether we will predict from B or C. If the difference is smaller than the threshold, the prediction will be $\lfloor(B+C)/2\rfloor$ (where $\lfloor\cdot\rfloor$ denotes rounding to the nearest lower integer), and no guide bits will be used. We found that this simple predictor works better than the standard Weinberger predictor [WSS96], even when accounting for the cost of the guide bits. Our predictor uses $\text{abs}(B-C) < t$, where we found that $t = 2048$ works well for half data. The predictor is also illustrated in Figure 1,

The leftmost block in Figure 3 shows an example of a block with such a discontinuity edge. Traversing the pixels in scan-line order, the boxed pixel will be the first different value, and we call this the *restart value*. Its position in the block (4 bits) and its value (15 bits) will be stored explicitly. We have used exhaustive search among all 15 positions to find the best restart value, but faster heuristic approaches can also be used. We describe one such heuristic for *depth* values in Figure 4 . Note how the five guide bits in Figure 3 make sure that prediction is never done across the discontinuity. In some cases, as shown in the dotted pixel in the middle illustration, neither the top nor the left pixel will give a good prediction. Therefore, we have introduced a bit to indicate whether the block is rotated or not. After rotation, the same pixel has a better chance to choose a good predictor, although there are still degenerate cases when this is not possible.

Next, the prediction errors $\tilde{R}_{xy}$ are stored. Since these are differences between 15-bit numbers, they are 16-bits signed integers. The first step is to make them positive so that a Golomb-Rice coder can be used. By applying the function $n(x) = -2x$ to negative numbers and $p(x) = 2x - 1$ to positive ones, the new arrangement will be $\{0, 1, -1, 2, -2, 3, -3\ldots\}$, which means that numbers of small magnitudes will have a small value. Each value will then be Golomb-Rice encoded: First, it is divided by 2^k, creating a quotient and a remainder. The quotient is encoded using unary encoding according to the table to the right. Values larger than 31 are encoded using `0xffff` followed by the 16 bits of the value. The k bits of the remainder are then stored. Four pixels in a 2×2 group share the same k-value, which is stored before the quotient and remainder data. An exhaustive search between 0 and 15 is used to find the best k for the group. In practice, though, the search can be made smaller by looking at the bit position p of the most significant bit of the largest value. The best k is almost always found in the interval $[p-4, p]$.

Code	Value
0_b	0
10_b	1
110_b	2
1110_b	3
11110_b	4
...	

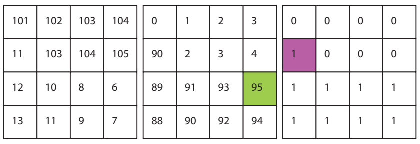
Whereas the red component is encoded independently of the other data, the green component is encoded relative to the red one. First, the difference between the green and the red component is calculated for each pixel, again treating the fp data as 15-bit integers. Then this difference is encoded in the same way as the red component was above, with two changes. First, the restart value and the top left value are not treated separately, but are fed into the Golomb-Rice encoding just as the other values are, but without prediction. Second, the prediction pattern from the red component is used again, meaning that no extra guide bits need to be sent. After this, the difference between the blue and the green component is encoded in the same way as the difference between green and red.

Many blocks have uniform data, and for them it may be unnecessary to encode a restart value. Therefore we reserve one bit to indicate whether we use the restart value or not. The total data structure is as follows: restart bit (1 bit), restart pos (4/0 bits), restart value (15/0 bits), rotate bit (1 bit), start value (15 bits), k-values (16 bits), guide bits (variable), and Golomb-Rice bits (variable).

2.1. Compressing Depth Data

The fp depth buffer compression algorithm, which is the first method of its kind to the best of our knowledge, is built using the same techniques as the color buffer compression system. In this way, a single hardware unit can handle both depth and color with little extra hardware. Hence, the data is also predicted, and the prediction error is encoded using Golomb-Rice. However, the way the prediction is carried out is different.

We use the complementary-Z representation of Lapidous and Jiao [LJ99] with no sign bit, three bits of exponent and 13 bits of mantissa, which is sometimes an alternative to 24 bit integer depth buffers. Since this move from 24 bits to 16 bits is in itself a data reduction by 2/3, it gives us a head start against 24 bit integer based depth buffer compression techniques. Note that most integer depth compression algorithm rely on the fact that the depths of a plane (e.g., from a triangle) are coplanar (up to rounding accuracy) in the depth buffer. However, for floating-point depths, this may not be true, especially when the exponents differ. However, for values of the same exponent, they are still coplanar, so a planar prediction still produces good results, and the remaining deviations can be handled by the Golomb-Rice encoding. However, planar prediction requires more pixels to predict from compared to the RGBA case.



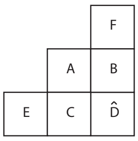
101	102	103	104
11	103	104	105
12	10	8	6
13	11	9	7

0	1	2	3
90	2	3	4
89	91	93	95
88	90	92	94

0	0	0	0
1	0	0	0
1	1	1	1
1	1	1	1

Figure 4: *The position of the restart value Z_R is calculated as follows: First the absolute difference to element Z_{11} is calculated (middle). The element most different from Z_{11}, called Z_{diff}, is found (marked with green). Then each pixel that is closer to Z_{11} than to Z_{diff} is allotted a restart pixel of value of 0, else 1 (right). The restart position is the first pixel in traversal order of value 1 (purple).*

The algorithm is intended to be able to handle two different planes per 4×4 sub-block. The top left pixel belongs per definition to plane 0, and its value Z_{11} will be stored. For cleared blocks, Z_{11} will be equal to the Z_{far}-value, hence one bit is used to signal whether this is the case or if it should be stored explicitly using 16 bits. The value of the first encountered pixel of plane 1, which we call the restart value or Z_R for short, will also be stored explicitly. A bit mask telling which plane each pixel belongs to is also stored. The figure to the right shows all the pixels that can be used to predict the value D. However, it can only predict from values that belong to the same plane as D. If A, B and C belongs to the same plane as D, it will use the prediction $\hat{D} = B + C - A$. Else, if B and F belong to the same plane as D, it will use $\hat{D} = 2B - F$. Else, it will try $\hat{D} = 2C - E$. If this also fails, but B and C both belong to the same plane as D, one *extra guide bit* will be sent to choose between the predictions $\hat{D} = B$ and $\hat{D} = C$. If only one of them share planes with D, that one will be used and no extra guide bit will be spent. Finally, if none of these pixels are of the same plane as D, it will use the value of the first encountered pixel of that plane. For plane 0, this means $\hat{D} = Z_{11}$, and for plane 1, this means $\hat{D} = Z_R$. If any of the pixels A, B, C, E and F are outside the block, they are treated as not belonging to the same plane as D.



The position for the restart value, Z_R, is determined as shown in Figure 4. First, the pixel is found that differs the most from Z_{11}. This is done by calculating the absolute value

$|Z_{xy} - Z_{11}|$ for each pixel position (x,y), as shown in the middle diagram. This pixel, which is green in the figure, is called Z_{diff}. Then a *restart bit* is given to each pixel: if $|Z_{xy} - Z_{11}| < |Z_{xy} - Z_{\text{diff}}|$ it will be 0, otherwise 1. The restart bits are then traversed in the prediction order, and the position of the first '1' will define the position of the restart value, marked with purple. The restart bits are only used for determining the restart value during compression, and are not stored. One could use the restart bits for the guide bits, but it turns out to be better to select the guide bits during compression; both '0' and '1' are tried for each guide bit, and the choice giving the best prediction will be selected. If this method of encoding two separate planes is not beneficial, the entire block can be compressed as a single plane instead. One bit is used to indicate which method was used.

The differences between the predicted and the actual values are encoded using Golomb-Rice encoding just as in the color buffer compression scheme, with one important difference. Only values predicted using two or more values, such as $\hat{D} = B + C - A$, $\hat{D} = 2B - F$ and $\hat{D} = 2C - E$, use the normal k value indicated for the 2×2 block. Values predicted from just one other pixel, i.e., using the predictions $\hat{D} = B$ and $\hat{D} = C$, will use $k_2 = \lfloor k/2 \rfloor + 10$. The reason for this is that the errors are much larger in this case, and would force too big a k value to be used if not compensated for in this way. These values can be seen as the encoding of the slope of the plane, whereas the values predicted from two values are merely the deviations from the plane, which should be small. We have tried encoding the k-values independently from each other, but this gave very little coding gain over the simple $k_2 = \lfloor k/2 \rfloor + 10$ formula. This is fortunate, since the search space for the best k becomes much smaller. In addition, due to very small errors within a plane, $k = 0$ is a very common case, so using only one bit to indicate wether $k = 0$ leads to further gains.

The depth scheme uses two modes, where the first is 192 bits and the second 768 bits. Many 8×8 blocks consist of pixels from only one plane, and for those it is unnecessary to store several starting values for the 4×4 sub-blocks. The guide bits will also be unnecessary, since the entire block is only one plane. Therefore, the 192-bit mode uses just one 8×8 block instead of four 4×4 blocks. Also, in the 8×8 case, no guide bits or restart value are stored, and k-values are selected for 4×4 blocks instead of 2×2 blocks. In summary, the bits are as follows, for each 4×4 sub-block of the the 768-bit mode: $Z_{11} = Z_{\text{far}}$ (1 bit), Z_{11} (16/0 bits), two-plane mode (1 bit), guide bits (15/0 bits), restart value Z_R (16/0 bits, k ($4 \times 6/1$ bits), extra guide bits (variable), Golomb-Rice bits (variable). For the 192-bit mode, $Z_{11} = Z_{\text{far}}$ (1 bit), Z_{11} 16/0 bits), k ($4 \times 6/1$ bits), extra guide bits (variable), Golomb-Rice bits (variable). As a reference, we have also created a 24-bit integer version of the algorithm. It is exactly the same, except for the fact that the starting and restart values are stored with 24 bits instead of 16 bits.

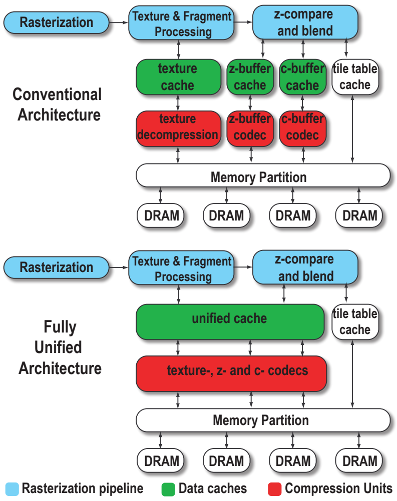


Figure 5: *Conventional vs unified cache architecture. In the bottom part, both the cache and the codecs are unified.*

Codecs can be put together in many ways, especially when it comes to details. However, we hope to have showed that some overall strategies are useful, such as not predicting across discontinuities.

3. Implementation

In order to evaluate our compression algorithms, and their impact on different architectures, we have built a software-based simulation framework, which can be configured to emulate a host of different architectures. All configurations use Zmax-culling [GKM93, Mor00] and Zmin-culling [AMS03] to avoid unnecessary fragment accesses. Pixels are rendered tile-by-tile in order to maximize data locality, and the tile size is 8×8 pixels. Fast Z-clears [Mor00] are also used. In all our tests, we use HDR texture compression (8 bits per texel) [MCHAM06], and normal map compression (8 bits per normal) when possible. The caches for all configurations store decompressed data, are fully associative and use a least recently used replacement policy.

The first configuration, A, is a conventional architecture [KF05] (shown on the top of Figure 5), with separate caches for textures and buffers (a non-unified cache), and where it is not possible to render from textures compressed with buffer compression schemes (a non-unified codec structure). Furthermore, no fp color buffer compression is used, however 24-bit integer depth buffer compression is employed. We have used a texture cache [HG97, IEH99] of 13 kB, a color buffer cache of 1 kB and a depth buffer cache of 512 bits. The texture cache is rather large, and the reason for this is that we use floating-point data throughout, which increases storage needs. We also use 256 bits for a tile table cache to store the tile table bits for the depth buffer.

Configuration B is equal to A except that it includes fp color buffer compression. It thus shows the benefit of adding the proposed fp color buffer compression scheme to a traditional architecture.

Configuration C equals B except that it also implements unified codecs, meaning that compressed color and depth buffers can be recast as textures without any need for decompression. However, since different caches are used, the cache must be flushed before the recast takes place.

Finally, configuration D utilizes both fp color buffer compression, unified coders and unified caches, as shown in the bottom part of Figure 5. The unified cache is of 14 kB for texture, color buffer and depth buffer, and a common tile table cache of 768 bits. Thus the configuration D is given equal amounts of cache memory as compared to A, B and C.

Example To highlight the differences between configuration A and D, consider the case of shadow mapping. The shadow map is created by rendering a depth buffer, which is used as a texture (the "shadow map"). When rendering the final image, the shadow map is used as lookup data to determine if a fragment is in shadow or not. In configuration A, depth buffer compression can be used in the first shadow map generation pass. However, when it is time to cast the depth buffer to a texture it has to be decompressed and when it is subsequently used as a texture in the later passes it can only be used in uncompressed form. In configuration D, due to the unified cache architecture, the depth buffer can use the entire cache during the creation of the shadow map, increasing hit rate and lowering bandwidth usage. Furthermore, the cache does not need to be flushed when recasting the shadow map from a buffer to a texture. Due to the unified codec architecture, no bandwidth-consuming decompression is needed during recasting. In the final pass, the shadow map is accessed in compressed form, which further reduces bandwidth.

We use three test scenes; *Water*, *Shadows*, and *Reflections*, where all render to fp16 color buffers. HDR textures in cube maps and object textures make sure that the dynamic range of the floating-point color buffer is used. Note that the final fp color buffers are tone mapped before display, so even though a screenshot may seem simple to compress (e.g., large bright area), this is very seldom true since tone mapping is a non-linear operator with clamping at the end, and this hides details in the fp colors. *Water* is a rather regular scene, without any render-to-texture, which means that the unified codec architectures cannot exploit the important feature of reusing codecs for render-to-textures, when accessing these as textures. Still, we included this simple example in order to show that bandwidth can still be saved under those conditions. The second scene is called *Shadows*, and since it contains a shadow map, it is designed to highlight the benefits of the proposed system. *Reflections* renders to a dynamic floating-point cube map every frame, and a sphere reflects the surrounding objects using the cube map.

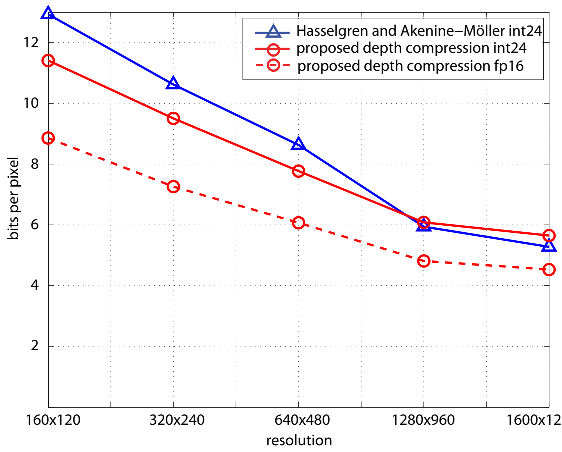


Figure 6: *Depth compression results for the "Shadows" scene for different resolutions. Note how our proposed fp16 depth buffer (dashed red circles) has a significantly lower rate than the state-of-the-art int24 system (blue triangles). Our proposed int24 system (solid red circles) outperforms the state-of-the-art for high-complexity (low-resolution) scenes.*

4. Results

In this section, we present the results from our simulations. Performance numbers for the fp16/int24 depth buffer compression and the fp16 color buffer compression are presented first, followed by overall performance results.

The diagram in Figure 6 shows the performance of the proposed depth buffer compression schemes (Section 2). We have rendered the *Shadows* scene in resolutions ranging from 160 × 120 to 1600 × 1200, and plotted the rate for our fp16 depth buffer compression algorithm (dashed red circles). Since there is no previous fp16 depth buffer algorithm to compare to, we plot it against the int24 algorithm of Hasselgren and Akenine-Möller [HAM06] (blue triangles), which we believe is state-of-the-art. As can be seen, our new technique has a substantially lower rate. However, the results are not directly comparable, since the original data is fp16 and not int24. Therefore, we also show the result of our proposed int24 technique, which outperforms the state-of-the-art at lower resolutions. The reason the state-of-the-art encoder does well for large resolutions is that it can compress an 8 × 8 block down to 128 bits if it is a single triangle covering the block, whereas the proposed algorithms will need 192 bits. For very high resolutions, this will happen more and more frequently. Note however that this means very big triangles with respect to the pixel size. Already at the cutover resolution of 1280 × 960, we have an average size of 507 pixels per triangle. We argue that the lower resolutions, which are equivalent to a higher scene complexity/smaller triangles, are more important, and here both proposed methods are better than the state-of-the-art.

The proposed fp16 color buffer codec algorithm is shown in Figure 7 with green crosses. There is no previous fp color buffer compression method to compare to. As a comparison, both the HDR texture compression algorithms from Munkberg et al. [MCHAM06] and Roimela et al. [RAI06] use 8 bpp (blue squares in Figure 7), whereas our buffer compression method goes down to about 25 bpp. Note however, that the methods used for HDR texture compression are lossy, and therefore not amenable to color buffer compression. In fact, being lossy is a huge gain; as a comparison, lossy JPEG codecs typically operate at around 2 bpp whereas lossless PNG codecs manage around 12 bpp. Nor is it a good idea to use our fp color buffer compression method for texture compression; too many simultaneous textures could thrash the tile table cache, resulting in abysmal performance. The extra latency needed in order to first fetch the tile table data before knowing how many bytes to read from memory is manageable if the number of such textures is small so that the tile table data is mostly in the cache. While a scene would typically contain just a few render-to-textures (no thrashing), it would most likely have several ordinary textures even per pixel (risk of thrashing). In its current form, we would therefore recommend not using our fp color buffer compression for all regular fp16 textures, but definitely use it for fp16 render-to-textures. Note also that the proposed method reduces bandwidth down to 40% of the original size—a significant compression. Results for the *Water* and *Reflections* scenes are similar at about 43%. Admittedly, much of this compression comes from demanding that destination alpha equals 1.0. However, even if it was assumed that the original color buffer was RGB and not RGBA, the compression would still reduce the size to 53%–60%.

We also compare our method to the fp image compression scheme in OpenEXR [Ope08] (black triangles in Figure 7). Since it is not intended for buffer compression, we have made a number of changes to make the comparison. OpenEXR uses several codecs, of which the PIZ-codec is the one best suited for RGBA data. It performs a Haar wavelet transform and then stores the resulting symbols using run length encoded Huffman symbols. A PIZ-encoded block includes a header of 20 bytes which can most likely be made much smaller—we have therefore removed 20 bytes from the calculated byte-size of each PIZ-encoded block. Moreover, the PIZ-codec is created for RGBA textures, whereas our algorithm assumes that alpha is 1.0 everywhere. This gives the PIZ-codec an unfair disadvantage. In order to compensate for that, we compressed an RGBA image of 16 × 16 pixels with the PIZ-encoder where R=2.0, G=3.0, B=4.0 and A=1.0 everywhere, which resulted in 33 bytes, of which 20 bytes was header. We have therefore removed another 13 bytes from the output of the PIZ-encoder to give it a fair comparison (in total we have removed 33 bytes). Note that the PIZ-codec is designed for large images where the cost of the Huffman table can be amortized over many pixels. It is also designed for natural images without discontinuities. However, we have included it here for completeness.

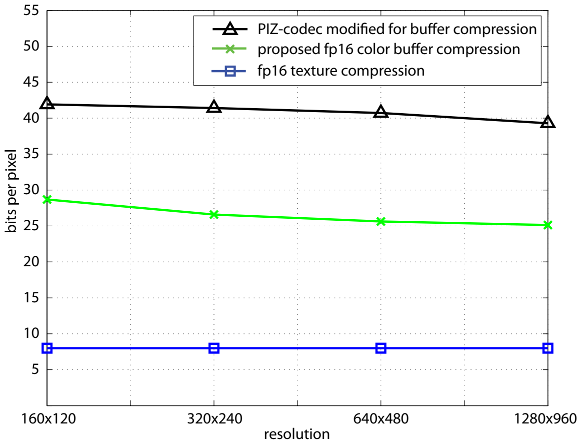


Figure 7: *Fp16 color buffer compression results for the "Shadows" scene for different resolution. Black triangles: The PIZ-codec from OpenEXR, modified to be used for buffer compression, as a benchmark. Note that the PIZ-codec is designed for larger images without discontinuities, but it is included here for completeness. Green crosses: the proposed fp16 compression scheme. Blue squares: The HDR texture compression methods of Munkberg et al. and Roimela et al., which cannot be used for buffer compression.*

Using the PIZ-codec on 8 × 8 data gives no data reduction; compressed blocks are almost always bigger than 100% of the original data. Therefore, 16 × 16 blocks have been used for the PIZ-codec. We have also doubled the size of the color buffer cache in the PIZ-codec case so that a full block can fit. For the proposed method, we have instead halved the cache size, so that both methods can store exactly one block in the cache, to avoid that any possible cache thrashing influences the comparison.

All scenes have been rendered twice in the 640 × 480 resolution, first with the proposed 8 × 8 algorithm and then with the 16 × 16 algorithm based on the PIZ-codec. Bandwidth usage is reported in the table below:

	Proposed	PIZ	factor
Ocean	2.82 MB	5.56 MB	2.0×
Shadow	5.38 MB	13.66 MB	2.5×
Reflections	8.28 MB	28.18 MB	3.4×

This looks favorable for the proposed algorithm, with more than a factor of two difference on average. However, it should be noted that moving from 8 × 8 to 16 × 16 tiles in itself increases bandwidth usage even if no compression is used, since many pixels that are never used will be read/written. Therefore, we have also compared the bandwidth usage against uncompressed 8 × 8 and 16 × 16 rendering:

	Proposed as % of original 8 × 8	PIZ as % of original 16 × 16	factor
Ocean	46%	84%	1.8×
Shadow	40%	68%	1.7×
Reflect.	43%	71%	1.7×

Thus, the improvement ratio is reduced down to around 1.7×. Hence, even without the bandwidth gains of an 8 × 8 system, there is still a substantial compression advantage of the proposed algorithm compared to PIZ.

At first it may seem a bit counter-intuitive that the difference between the two algorithms is so large. One important factor here is that many blocks generated during rendering are not completely filled by a single triangle, but have pieces of the background or pieces of another triangle in the block. This yields significantly better compression for the proposed method as the following simple test will show:

When compressing blocks that are 100% full of image data (i.e., no background), the two algorithms are very similar to each other in terms of compression ratio. However, when turning several pixels to the background color (e.g., black), we see a clear difference in the behavior of the two systems. If the majority of the pixels are black, it is easier to code the block for both systems. However, for the 25% mode to kick in for the PIZ-encoder, only about 8 out of 256 pixels can be non-black, whereas for the proposed compressor, about 20 out of 64 pixels can be non-black. This means that the 25% mode almost never kicks in for the PIZ-encoder, which we have also seen in the simulation results. At the same time, it is no surprise that the proposed method is better at these discontinuities, since it was designed to handle exactly this common case.

For this reason, the PIZ-codec works better with block sizes of 50% and 75% instead of 25% and 50%. (Block sizes of 75% and 50% are just as burst-friendly as 25% and 50%.) As an example, this lowers bandwidth for the *Shadow* scene from 68% to 63% of the original. Hence we use 50% and 75% blocks for the PIZ-codec in Figure 7. It is likely that performance would increase further if these block sizes were optimized, but we have not performed this optimization neither for the proposed algorithm nor for the PIZ-encoder. It is also clear that introducing more possible block sizes would benefit both algorithms. Therefore, to make sure that the relative superiority of the proposed algorithm is not only due to bad choices of block sizes, we have made a final test where all block sizes are allowed, which gives the rate equivalent to allowing full variable bit lengths. In this comparison we have also used 16 × 16 tiles for both methods (by storing four 8 × 8-blocks together for the proposed encoder) to remove any possible dependencies on block size. This yields the following results:

	proposed 16 × 16 as % of original BW (any block size)	PIZ 16 × 16 as % of original BW (any block size)
Ocean	33%	57%
Shadow	28%	39%
Reflec.	28%	46%

We see that our algorithm is better than the PIZ-codec by an average factor of 1.6×, which is substantial. Note further that 8 × 8 blocks are much to prefer over 16 × 16 blocks, not only because of the extra bandwidth usage associated with 16 × 16 blocks as described above, but also since a larger number of pixels with data dependencies take more clock cycles to compress/decompress. This is the reason why we use four 4 × 4 blocks inside the 8 × 8 block—these can be compressed in parallel given enough hardware resources. We have also tried an 8 × 8 version of our algorithm, which reduces the bandwidth by another four percentage units, but this does not give the same support for parallel hardware compression. So for fp color *buffer* compression, we have shown that our proposed algorithm performs better than OpenEXR's PIZ algorithm. In addition, we want to mention that PIZ also needs to create a unique Huffman table per tile when compressing a tile, and this is expected to be rather expensive.

Next, we evaluate our entire architecture with the new compression algorithms, unified cache, and unified codecs. Table 1 shows the bandwidth figures for the the three scenes rendered at 1024 × 768 for the different configurations. For the *Water* scene, configuration *A* through *D* are as described above. Since there is no render-to-texture taking place, *B* and *C* are in fact equivalent.

For the *Shadows* scene, configuration *A* and *B* are slightly different in that we do not to compress the depth buffer during the creation of the shadow map. This is due to the fact that we want to be able to render from it later, and the texturing unit in a traditional architecture cannot read data compressed with the depth compression method. An alternative would be to create the shadow map with compression turned on, and then uncompress it when it is moved to a texture, but we found this to use slightly more bandwidth for this scene. The final depth buffer is compressed, however. In the *Reflections* scene, an HDR environment map is rendered as background, and a dynamic HDR cube map is created every frame, so that a reflection can be rendered in the sphere in the center. The numbers for depth buffer bandwidth include rendering from the camera as well as to the six faces of the cube for the environment.

Comparing columns *A* and *B* in the BW ratio row, we see that substantial gains are made over a traditional architecture just by adding the floating point color buffer compression. To ensure pixel exactness, we have used the int24 version of the depth buffer compression. Depth buffer bandwidth would be decreased by about 14% further if fp16 were used. Column C uses the proposed unified codec architecture, and we see that bandwidth is further lowered to about 60% of the orig-

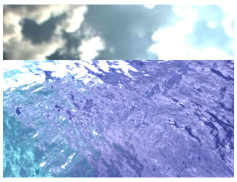
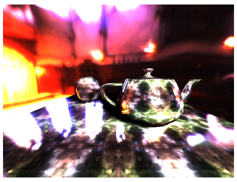
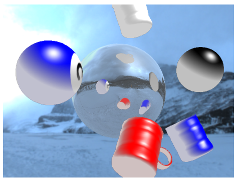
Scene	**Water**				**Shadows**				**Reflections**			
												
# triangles	44				6468				60336			
	1024 × 768											
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
Color BW	15.0	6.4	6.4	6.5	30.0	11.8	11.8	11.1	32.6	14.1	14.1	10.4
Depth BW	1.1	1.1	1.1	1.1	3.4	3.4	3.4	2.7	14.0	14.0	14.0	9.6
Shadow/Cube	n/a	n/a	n/a	n/a	5.9	5.9	2.0	1.5	52.3	52.3	24.4	21.9
Texture BW	17.0	17.0	17.0	16.1	12.5	12.5	12.5	11.6	7.3	7.3	7.3	7.6
Total BW	33.1	24.5	24.5	23.7	51.8	33.6	29.6	27.0	106.2	87.7	59.7	49.4
BW ratio	**100 %**	**74.1%**	**74.1%**	**71.5%**	**100 %**	**64.8%**	**57.2%**	**52.0%**	**100 %**	**82.5%**	**56.2%**	**46.5%**
	320 × 240											
BW ratio	**100 %**	**83.9%**	**83.9%**	**77.4%**	**100 %**	**78.9%**	**56.5%**	**48.8%**	**100 %**	**95.2%**	**59.8%**	**50.1%**
	1600 × 1200											
BW ratio	**100 %**	**69.6%**	**69.6%**	**68.1%**	**100 %**	**59.8%**	**54.6%**	**50.9%**	**100 %**	**75.5%**	**54.4%**	**47.2%**

Table 1: *Performance figures in MB/frame for our three test scenes. Configuration A is a traditional architecture, with depth compression. In the* Shadows *scene, depth compression is turned off during creation of the shadow map. Configuration B is equal to A, but with fp color buffer compression turned on. Configuration C uses unified codecs, i.e., render-to-texture textures can be created and rendered from in compressed form. Configuration D shows results for a unified cache architecture. Note how color BW goes down substantially between A and B, and how overall bandwidth is reduced significantly for C and D.*

inal. The reason for this is that the shadow map (*Shadows* scene) and cube map (*Reflections* scene) textures can be created and rendered from with compression turned on. Finally, on a system with a unified cache for all the buffers, bandwidth can be reduced down to below 50% for the *Reflections* scene. Detailed figures are provided for the 1024 × 768 resolution. For 320 × 240 and 1600 × 1200, we have presented BW ratio figures showing similar results.

5. Discussion

A unified codec architecture is certainly simpler if most codecs are harmonized. We have presented two such algorithms but it would be interesting to have more. Fp32 color and depth buffers could be compressed with the proposed methods, although we have not tried. The 8-bit color buffer by Rasmusson [RHAM07] could be made even more similar to our work. We have even tried naive adaptations of our fp16 color for vertex data and (lossy) texture compression, with premature but promising results. That said, full codec harmonization remains a hard goal. For instance, S3TC/DXTC is hard to replace for RGBA8 textures.

Although a unified cache architecture seems competitive in this evaluation, it should be noted that there may be potential implementation problems that are not discovered at this level of simulation. There is no way of finding out short of actually implementing a unified cache, which is out of the scope of this paper.

Even though some measures have been taken to lower compression/decompression latency in this paper (such as using four 4 × 4 tiles instead of one 8 × 8), a proper latency analysis would require more detailed work.

6. Conclusion

We have proposed two new algorithms for fp buffer compression, one for color and one for depth data, which are the first published algorithms on these topics. An int24 version of the depth codec has been shown to be competitive against state-of-the-art. Finally, we have demonstrated how these algorithms behave in architectures with varying degrees of unification, reaching bandwidth reductions down to 50%.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vinnova.

References

[AMN03] AILA T., MIETTINEN V., NORDLUND P.: Delay Streams for Graphics Hardware. *ACM Transactions on Graphics, 22*, 3 (2003), 792–800.

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics 22*, 3 (2003), 801–808.

[BAC96] BEERS A., AGRAWALA M., CHADDA N.: Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96* (1996), pp. 373–378.

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93* (August 1993), pp. 231–238.

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: Efficient Depth Buffer Compression. In *Graphics Hardware* (2006), pp. 103–110.

[HG97] HAKURA Z. S., GUPTA A.: The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture* (1997), pp. 108–120.

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

[IM06] INADA T., MCCOOL M. D.: Compressed Lossless Texture Representation and Caching. In *Graphics Hardware* (2006), pp. 111–120.

[KF05] KILGARIFF E., FERNANDO R.: The GeForce 6 Series GPU Architecture. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 471–491.

[KSKS96] KNITTEL G., SCHILLING A. G., KUGLER A., STRASSER W.: Hardware for Superior Texture Performance. *Computers & Graphics, 20*, 4 (1996), 475–481.

[LI06] LINDSTROM P., ISENBURG M.: Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics, 12*, 5 (2006), 1245–1250.

[LJ99] LAPIDOUS E., JIAO G.: Optimal Depth Buffer for Low-Cost Graphics Hardware. In *Graphics Hardware* (1999), pp. 67–73.

[MB98] MCCABE D., BROTHERS J.: DirectX 6 Texture Map Compression. *Game Developer Magazine 5*, 8 (1998), 42–46.

[MCHAM06] MUNKBERG J., CLARBERG P., HASSELGREN J., AKENINE-MÖLLER T.: High Dynamic Range Texture Compression for Graphics Hardware. *ACM Transactions on Graphics, 25*, 3 (2006), 698–706.

[Mor00] MOREIN S.: ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings* (August 2000), ACM SIGGRAPH/Eurographics.

[NVI06] NVIDIA: *GeForce 8800 GPU Architecture Overview*. Tech. rep., TB-02787-001_v01, 2006.

[Ope08] OPENEXR: www.openexr.com/about.html. web site, 2008.

[Owe05] OWENS J. D.: Streaming Architectures and Technology Trends. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 457–470.

[RAI06] ROIMELA K., AARNIO T., ITÄRANTA J.: High Dynamic Range Texture Compression. *ACM Transactions on Graphics, 25*, 3 (2006), 707–712.

[RHAM07] RASMUSSON J., HASSELGREN J., AKENINE-MÖLLER T.: Exact and Error-bounded Approximate Color Buffer Compression and Decompression. In *Graphics Hardware* (2007), pp. 41–48.

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Graphics Hardware* (2005), pp. 63–70.

[TK96] TORBORG J., KAJIYA J.: Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of SIGGRAPH* (1996), pp. 353–364.

[WSS96] WEINBERGER M. J., SEROUSSI G., SAPIRO G.: LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm. In *In Data Compression Conference* (1996), pp. 140–149.