V 2

Real-Time BC6H Compression on GPU

Krzysztof Narkowicz

2.1 Introduction

BC6H texture compression is a lossy block compression designed for compressing high-dynamic range (HDR) images; it is also widely supported by modern GPUs. It drastically decreases memory usage and improves runtime performance, as it also decreases required GPU bandwidth.

Real-time HDR compression is needed in certain applications—e.g., when HDR data is generated at runtime or when offline compression is too slow. It is usually desirable for real-time compression to entirely bypass the CPU and run the compression algorithm on the GPU. This way, resource expensive CPU-GPU synchronization is avoided and data transfer between CPU and GPU is not required at all.

This chapter describes a simple real-time BC6H compression algorithm, one which can be implemented on GPU entirely.

2.1.1 Real-Time Environment Map Compression

A typical BC6H compression application is HDR environment maps, commonly used in physically-based lighting. Open-world games often require separate environment maps for every location, time of day, or weather condition. The resulting combinatorial explosion of possibilities makes it impractical to generate and store them offline. In those cases games have to generate environment maps dynamically. Some games use simplified scene representation to render a single global environment map every frame [Courrèges 15]. This environment map is attached to the camera and used for all reflections. Another possible option is to store G-buffer data for every environment map and dynamically relight it when lighting conditions change [McAuley 15].

Generating dynamic environment maps is not limited to open world games only. It also allows dynamic scenes to be lit—procedurally generated scenes, scenes containing geometry destruction, or ones containing dynamic object movement.

Furthermore, when using dynamic environment map generation, only the environment maps in the viewer's proximity have to be kept in memory. This allows a greater density of environment maps and better lighting quality. Environment maps are pre-convolved for a given single point—usually the center of capture. Lighting quality degrades further from that center because of the changing filter shape, occlusion, and limited resolution [Pesce 15]. The simplest way to alleviate those artifacts is to increase the number of environment maps by generating them at runtime.

2.1.2 BC6H Alternatives

Before BC6H was introduced, HDR images were usually stored in BC3 compressed textures with special encoding. These encodings could also be used to compress in real time as suitable BC3 compression algorithms exist [Waveren 06]. Such encodings either separate chrominance and luminance and store luminance in two channels (LogLuv, YCoCg) or store normalized color and some kind of multiplier (RGBE, RGBM, RGBD, RGBK) [Guertault 13]. The compression ratio is on a par with BC6H and runtime performance is usually better for LDR formats. However, those approaches will result in inferior image quality, as the mentioned methods result in various encoding and compression artifacts. Encoding only individual channels causes a hue shift, while encoding all the channels together increases luminance errors. BC3 compression was designed for LDR images, where small value changes resulting from compression are acceptable. In the case of encoded HDR data, the consequence of such small changes can be magnitude differences in the decoded results. Finally, current hardware does not support native texture filtering for these kinds of encodings, so either additional filtering artifacts will appear or manual texture filtering is required.

2.2 BC6H Details

BC6H block compression was introduced together with Direct3D 11. It is designed for compressing signed and unsigned HDR images with 16-bit half-precision float for each color channel. The alpha channel is not supported, and sampling alpha always returns 1. BC6H has an 6:1 compression ratio and stores the texture data in separate 4×4 texel blocks. Every block is compressed separately. It is convenient for native GPU decompression, as the required blocks can be located and decompressed without the need to process other data. The basic idea is the same as for BC1, BC2, or BC3. Two endpoints and 16 indices are stored per block. A line segment in RGB space is defined by endpoints, and indices define

Bits	Value
[0;4]	Header – 0x03
[5;14]	First endpoint red channel
[15;24]	First endpoint green channel
[25;34]	First endpoint blue channel
[35;44]	Second endpoint red channel
[45;54]	Second endpoint blue channel
[55;64]	Second endpoint green channel
[65;67]	First index without MSB
[68;71]	Second index
[72;75]	Third index
...	...
[124;127]	Last index

Table 2.1. Mode 11 block details [MSDN n.d.].

a location of every texel on this segment. The entire format features 14 different compression modes with different tradeoffs between endpoint, index precision, and palette size. Additionally, some modes use endpoint delta encoding and partitioning. Delta encoding stores the first endpoint more precisely, and instead of storing the second endpoint, it stores the delta between endpoints. Partitioning allows defining two line segments per block and storing four endpoints in total. Using one of 32 predefined partitions, texels are assigned to one of the two segments.

BC6H was designed to alleviate compression quality issues of BC1, BC2, and BC3: "green shift" and limited color palette per block. "Green shift" is caused by different endpoint channel precision. BC1 uses 5:6:5 precision—this is why many grayscale colors cannot be represented and are shifted toward green (e.g., 5:6:5 encodes the grayscale color RGB 15:15:15 as RGB 8:12:8). In order to prevent a hue shift, BC6H encodes every channel with the same precision. Another prominent compression issue occurs when a block contains a large color variation—a variation that cannot be well approximated by a single segment in RGB space. In order to fix this, BC6H has introduced compression modes with two independent endpoint pairs.

2.2.1 Mode 11

The proposed real-time compression algorithm uses only mode 11. This specific mode was chosen because it is simple, universal, and, in most cases, has the best quality-to-performance ratio. Mode 11 does not use partitioning or delta encoding. It uses two endpoints and 16 indices per 4×4 block (see Table 2.1). Endpoints are stored as half-precision floats, which are quantized to 10-bit integers by dropping last 6 bits and rescaling. Indices are stored as 4-bit integers indexed into a palette endpoint interpolation weight table (Table 2.2).

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Weight	0	4	9	13	17	21	26	30	34	38	43	47	51	55	60	64

Table 2.2. Index interpolation weight table.

Weight 0 corresponds to the first endpoint and weight 64 to the second endpoint. The values in between are calculated by interpolating quantized integers and converting the results to half-precision floats. They are interpolated as 16-bit integers instead of as floats, thus allowing efficient hardware implementation. Due to IEEE floating point specification, this method actually works reasonably well, as the integer representation of a float is a piecewise linear approximation of its base-2 logarithm [Dawson 12]. In this case, interpolation does not have to handle special cases because BC6H does not support NAN or infinities.

There is one final twist. The MSB (most significant bit) of the first index (the upper-left texel in the current block) is not stored at all. It is implicitly assumed to be zero, and the compressor has to ensure this property by swapping endpoints if the first index is too large.

2.3 Compression Algorithm

The compression algorithm consists of two steps:

1. Compute two endpoints per block.

2. Compute indices for every texel located in this block.

2.3.1 Endpoints

The classic endpoint algorithm for real-time BC1 compression computes a color-space bounding box of the block's texels and uses its minimum and maximum values as endpoints [Waveren 06]. This algorithm is very fast, as it requires only 16 minimum and 16 maximum instructions. J. M. P. van Waveren additionally decreases the size of the calculated bounding box by 1/16th. In most cases, this lowers the encoding error because most colors will be located inside the new bounding box, but it also tends to cause visible blocky artifacts when colors in a block are clustered near the edges of the bounding box. To solve this, it is better to refine the bounding box by rebuilding it without the outliers—minimum and maximum RGB values. The resulting bounding box decreases encoding error and removes mentioned artifacts (Figure 2.1).

The final step of the endpoint calculation algorithm is to convert the resulting endpoints to half-precision floats and quantize them to 10-bit integers. The presented algorithm does not form an optimal palette, but it is a good and fast approximation.

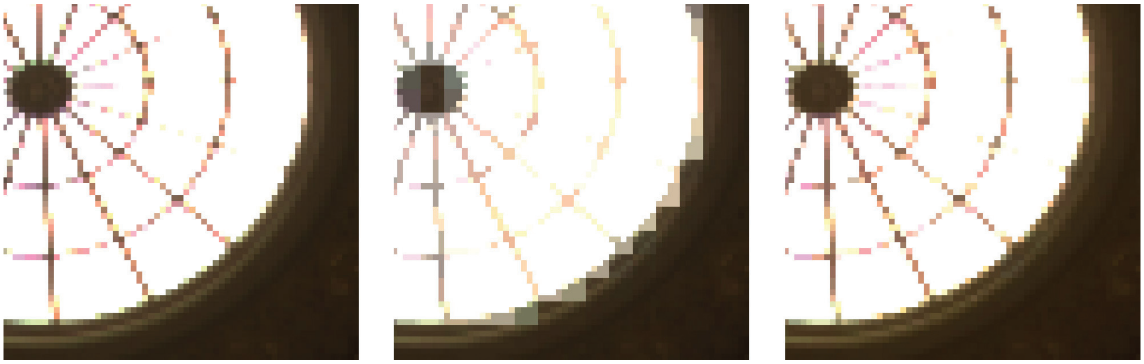

Figure 2.1. Endpoint algorithm: reference (left), compressed using bounding box inset (center), and compressed using bounding box refine (right).

2.3.2 Indices

Index computation requires picking the closest color from a palette that consists of 16 interpolated colors located on the segment between endpoints. A straightforward approach is to compute the squared distance between the texel's color and each of the palette entries and to choose the one that is closest to the texel's color. Due to a relatively large palette, this approach is not practical for real-time compression. A faster approach is to project the texel's color on a segment between the endpoints and pick the nearest palette entry. Unfortunately, endpoint interpolation weights are not evenly distributed (see Table 2.2). The first and last indices have the smallest range of best weights, and the remaining indices have similar ranges. A simple approximation is to fit the equation for smallest error—which is the same as solving the equation for the first and last bucket:

$$\text{index}_i = \text{Clamp}\left[\left(\frac{\text{texelPos}_i - \text{endpointPos}_0}{\text{endpointPos}_1 - \text{endpointPos}_0}\right) \times \frac{14}{15} + \frac{1}{30}, 0, 15\right].$$

The above equation wrongly assumes that the distance between middle indices is equal, but in practice this error is negligible. The final step is to swap endpoints if the MSB of the first index is set, as it is assumed to be zero and, thus, is not stored.

2.3.3 Implementation

The algorithm can be entirely implemented using a pixel shader, but if required, it could also be implemented using a compute shader or CPU code.

In the case of pixel shader implementation in DirectX 11, two resources are required: temporary intermediate `R32G32B32A32_UInt` render target and the destination BC6H texture. The first step is to bind that temporary render target and output compressed blocks from the pixel shader. The render target should be 16 times smaller than the source texture, so one texel corresponds to one BC6H block (16 source texels). The final step is to copy the results from the temporary render target to the destination BC6H texture using the `CopyResource` function.

To achieve optimal performance, the algorithm requires a native float to half conversion instructions, which are available in Shader Model 5. Additionally, it is preferable to avoid integer operations altogether (especially 32-bit divides and multiplies), as they are very costly on modern GPUs. For example, the popular Graphics Core Next (GCN) architecture does not natively support integer division, and it has to be emulated with multiple instructions [Persson 14]. The floating point number consists of a 24-bit integer (1-bit sign and 23-bit mantissa) and an 8-bit exponent. The algorithm uses only 16-bit integers, so all of the calculations can be done using floating point numbers without any loss of precision. Fetching a source 4×4 texel block can be done efficiently using 12 gather instructions instead of sampling a texture 16 times, as the alpha channel is ignored. Finally, `CopyResource` can be skipped entirely, when using low-level APIs that support resource aliasing of different formats.

The HLSL code in Listing 2.1 shows an implementation of the presented algorithm.

```
float Quantize( float x )
{
    return ( f32tof16( x ) << 10 ) / ( 0x7bff + 1.0f );
}

float3 Quantize( float3 x )
{
    return ( f32tof16( x ) << 10 ) / ( 0x7bff + 1.0f );
}

uint ComputeIndex( float texelPos, float endpoint0Pos,
                   float endpoint1Pos )
{
    float endpointDelta = endpoint1Pos - endpoint0Pos;
    float r = ( texelPos - endpoint0Pos ) / endpointDelta;
    return clamp( r * 14.933f + 0.0333f + 0.5f, 0.0f, 15.0f );
}

// Compute endpoints (min/max RGB bbox).
float3 blockMin = texels[0];
float3 blockMax = texels[0];
for ( uint i = 1; i < 16; ++i )
{
    blockMin = min( blockMin, texels[i] );
    blockMax = max( blockMax, texels[i] );
}

// Refine endpoints.
float3 refinedBlockMin = blockMax;
float3 refinedBlockMax = blockMin;
for ( uint i = 0; i < 16; ++i )
{
        refinedBlockMin = min( refinedBlockMin,
            texels[i] == blockMin ? refinedBlockMin : texels[i] );
        refinedBlockMax = max( refinedBlockMax,
            texels[i] == blockMax ? refinedBlockMax : texels[i] );
    }

    float3 deltaMax = ( blockMax - blockMin ) * ( 1.0f / 16.0f );
    blockMin += min( refinedBlockMin - blockMin, deltaMax );
    blockMax -= min( blockMax - refinedBlockMax, deltaMax );

    float3 blockDir = blockMax - blockMin;
    blockDir = blockDir / ( blockDir.x + blockDir.y + blockDir.z );

    float3 endpoint0    = Quantize( blockMin );
    float3 endpoint1    = Quantize( blockMax );
    float endpoint0Pos = f32tof16( dot( blockMin, blockDir ) );
    float endpoint1Pos = f32tof16( dot( blockMax, blockDir ) );

    // Check if endpoint swap is required.
    float texelPos = f32tof16( dot( texels[0], blockDir ) );
    indices[0] = ComputeIndex( texelPos, endpoint0Pos,
    endpoint1Pos );
    if ( indices[0] > 7 )
    {
        Swap( endpoint0Pos, endpoint1Pos );
        Swap( endpoint0, endpoint1 );
        indices[0] = 15 - indices[0];
    }

    // Compute indices.
    for ( uint j = 1; j < 16; ++j )
    {
        float texelPos = f32tof16( dot( texels[j], blockDir ) );
        indices[j] = ComputeIndex( texelPos, endpoint0Pos,
                                   endpoint1Pos );
    }
```

Listing 2.1. BCH6 compression algorithm.

2.4 Results

The proposed algorithm was compared with two offline BC6H compressors: Intel's BCH6 CPU-based compressor and the DirectXTex BC6H GPU-based compressor.

2.4.1 Quality

Root mean square error (RMSE) was used for measuring quality. RMSE is a generic measure of signal distortion, where lower values are better:

$$RMSE = \sqrt{\frac{1}{3n}\sum_{i=1}^{n}\left[(\hat{r}_i - r_i)^2 + (\hat{g}_i - g)^2 + (\hat{b}_i - b_i)^2\right]}.$$

Based on the results shown in Table 2.3, the proposed algorithm has similar quality to Intel's "veryfast" preset. Intel's "veryfast" has a smaller error

	Proposed Algorithm	**Intel BC6H** "veryfast"	**Intel BC6H** "basic"	**Intel BC6H** "veryslow"	**DirectXTex BC6H**
Atrium	0.122	0.112	0.1	0.099	0.084
Backyard	0.032	0.027	0.024	0.024	0.025
Desk	0.992	1.198	0.984	0.975	0.829
Memorial	0.25	0.278	0.241	0.237	0.216
Yucca	0.086	0.083	0.065	0.064	0.063
Average	0.296	0.340	0.283	0.280	0.243
Average of relative errors	22%	22%	10%	7%	0%

Table 2.3. Quality comparison (RMSE).

for images with a low RMSE, where 10-bit mode 11 quantization becomes a limiting factor. This is due to the delta encoding that is implemented even in the "veryfast" preset. For harder-to-compress images ("desk" and "memorial"), however, the proposed algorithm has error similar to Intel's "basic" preset. (See Figure 2.2.)

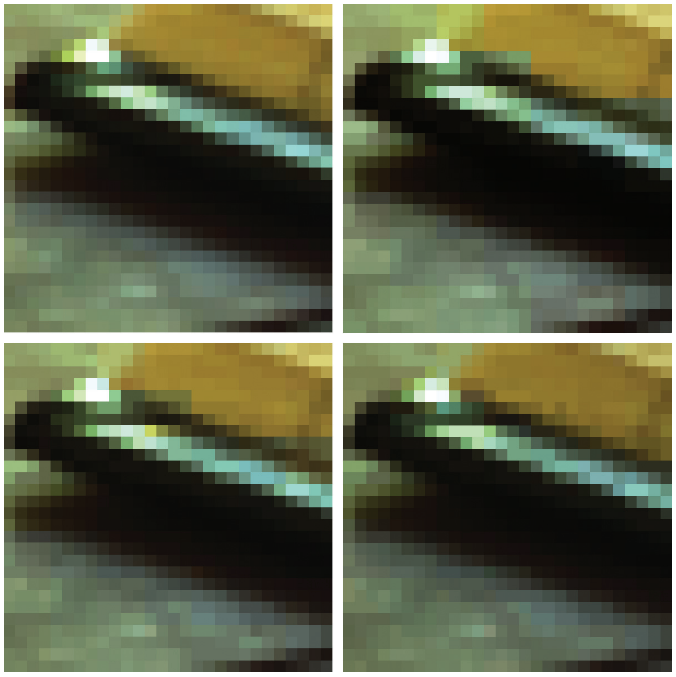

Figure 2.2. Quality comparison of "desk": original (top left), proposed algorithm (top right), Intel "veryfast" (bottom left), and Intel "basic" (bottom right).

	Image Size	**Proposed Algorithm**	**DirectXTex BC6H**
Atrium	760 × 1016	0.094 ms	1230 ms
Backyard	768 × 1024	0.095 ms	1240 ms
Desk	644 × 872	0.074 ms	860 ms
Memorial	512 × 768	0.057 ms	840 ms
Yucca	1296 × 972	0.143 ms	1730 ms
Average		7957.8 MP/s	0.6 MP/s

Table 2.4. Performance comparison.

2.4.2 Performance

Performance was measured using AMD PerfStudio. Results were verified by comparing timings obtained from DirectX performance queries. Tests were run on AMD Radeon R9 270 (mid-range GPU). The timings presented in Table 2.4 do not include `CopyResource` time, so the measured times should be increased by ~15% for APIs that require a redundant copy in order to modify the destination BC6H texture.

A standard 256 × 256 × 6 environment map with a full mipmap chain has almost the same number of texels as the "Desk" image, which can be compressed in about 0.07 ms on the mentioned GPU. This performance level is fast enough to compress dynamically generated environment maps or other content without a noticeable impact on performance.

2.5 Possible Extensions

A straightforward way to enhance quality is to use other compression modes. There are two possibilities:

1. to add delta encoding,
2. to use partitioning.

2.5.1 Delta Encoding

The first possible approach is to add modes 12, 13, and 14. These modes extend mode 11 with delta encoding by storing the first endpoint in higher precision (respectively 11, 12, and 16 bits) and storing the delta instead of a second endpoint in the unused bits (respectively 9, 8, and 4 bits). Delta encoding implementation is not too resource expensive. Its impact on quality is also limited, as in the best case delta encoding cancels the 10-bit quantization artifacts.

The delta encoding algorithm starts by computing the endpoints using the proposed endpoint algorithm. Endpoints are encoded with various formats, and the encoding precision is compared in order to select the best mode. Next, the indices are computed with the same algorithm as for mode 11. Finally, the appropriate block is encoded in a rather non-obvious way. First, 10 bits of the first endpoint are stored just as in mode 11. Next, the bits are stored together with the delta instead of the second endpoint. Both have their bits reversed.

2.5.2 Partitioning

The second approach is to add mode 10 (supporting partitioning). Partitioning adds a second palette (endpoint pair). This greatly improves the worst-case result—when colors in a block cannot be well approximated by a single segment in RGB space. Unfortunately, it is slow due to a large search space.

The partitioning algorithm starts with the selection of the best partition. This requires calculating all possible endpoints for partitions. There are 32 possible partition sets, so it means computing 32 combinations of endpoints. The partition with the smallest sum of bounding-box volumes is selected. Indices are computed just as for mode 11. Finally, the block is encoded using the computed endpoints, indices, and selected partition index.

2.6 Conclusion

The presented algorithm allows real-time BC6H compression on GPU with quality similar to fast offline solution presets. The quality can be improved further by using other compression modes—at the cost of lower performance. The algorithm proved to be efficient enough in the terms of both quality and performance for compressing runtime generated environment maps in *Shadow Warrior 2*—a game by Flying Wild Hog. The full source code for a simple application that implements the presented algorithm and the HDR images used for the tests can be found in the book's supplemental materials or on GitHub (https://github.com/knarkowicz/GPURealTimeBC6H).

2.7 Acknowledgements

Big thanks for proofreading to Przemysław Witkowski, Bartłomiej Wronski, Michał Iwanicki, and Artur Maksara.

Bibliography

[Courrèges 15] Adrian Courrèges. "GTA V—Graphics Study." http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study, November 2, 2015.

[Dawson 12] Bruce Dawson. "Stupid Float Tricks." https://randomascii.wordpress.com/2012/01/23/stupid-float-tricks-2/, January 23, 2012.

[Guertault 13] Julien Guertault. "Gamma Correct and HDR Rendering in a 32 Bits Buffer." http://lousodrome.net/blog/light/2013/05/26/gamma-correct-and-hdr-rendering-in-a-32-bits-buffer/, May 26, 2013.

[McAuley 15] Stephen McAuley. "Rendering the World of Far Cry 4." Presented at Game Developers Conference, San Francisco, CA, March 2–6, 2015.

[MSDN n.d.] MSDN. "BC6H Format." https://msdn.microsoft.com/en-us/library/windows/desktop/hh308952, no date.

[Persson 14] Emil Persson. "Low-level Shader Optimization for Next-Gen and DX11." Presented at Game Developers Conference, San Francisco, CA, March 17–21, 2014.

[Pesce 15] Angelo Pesce. "Being More Wrong: Parallax Corrected Environment Maps." http://c0de517e.blogspot.com/2015/03/being-more-wrong-parallax-corrected.html, March 28, 2015.

[Waveren 06] J. M. P. van Waveren. "Real-Time DXT Compression." http://mrelusive.com/publications/papers/Real-Time-Dxt-Compression.pdf, 2006.