Adaptive Scalable Texture Compression

Stacy Smith, ARM

1.1 Introduction

Adaptative Scalable Texture Compression (ASTC) is a new texture compression format which is set to take the world by storm. Having been accepted as a new Khronos standard, this compression format is already available in some hardware platforms. This article shows how it works, how to use it, and how to get the most out of it. For more in-depth information, there is a full specification provided with the encoder [Eva].

1.2 Background

ASTC was developed by ARM Limited as the flexible solution to the sparsely populated list of texture compression formats previously available. In the past, texture compression methods were tuned for one or more specific sweet spot combinations of data channels and related bit rates. Worsening the situation was the proprietary nature of many of these formats, limiting availability to specific vendors, and leading to the current situation where applications have to fetch an additional asset archive over the internet after installation, based on the detected available formats. The central foundation of ASTC is that it can compress an input image in every commonly used format (Table 1.1) and output that image in any user selected bit rate, from 8bpp to 0.89bpp, or 0.59bpp for 3D textures (Table 1.2).

Bitrates below 1bpp are achieved by a clever system of variable block sizes. Whereas most block based texture compression methods have a single fixed block size, ASTC can store an image with a regular grid of blocks of any size from 4x4 to 12x12 (including non square block sizes). ASTC can also store 3D textures, with block sizes ranging from 3x3x3 to 6x6x6.

Regardless of the blocks' dimensions, they are always stored in 128 bits, hence the sliding scale of bit rates.

Raw input format	Bits per pixel
HDR RGB+A	64
HDR RGBA	64
HDR RGB	48
HDR XY+Z	48
HDR X+Y	32
RGB+A	32
RGBA	32
XY+Z	24
RGB	24
HDR L	16
X+Y	16
LA	16
L	8

Table 1.1. Bitrates of raw image formats

Output Block Size [1]	Bits per pixel
4x4	8.000
5x5	5.120
6x6	3.556
8x8	2.000
10x10	1.280
12x12	0.889
3x3x3	4.741
4x4x4	2.000
5x5x5	1.024
6x6x6	0.593
4x6	5.333
8x10	1.600
12x10	1.067

Table 1.2. Bitrates of ASTC output

1.3 Algorithm

Each pixel in these blocks is defined as a quantised point on a linear gradient between a pair of boundary colours. This allows for fairly smooth areas of shading. For blocks containing boundaries between areas of completely different colours, the block can use one of 2048 colour partitioning patterns, which split the block into different designs of 1-4 colour gradients.

These blocks are algorithmically generated and selecting the right one is where the majority of the compression time goes. This technique allows a block to contain areas of completely different hues with arbitrary shading or multiple intersecting hard-edged patches of different tones. Each block defines up to four pairs of colours and a distribution pattern ID, so that each pixel knows which of those pairs it uses to define its own colour. The individual pixels then have a quan-

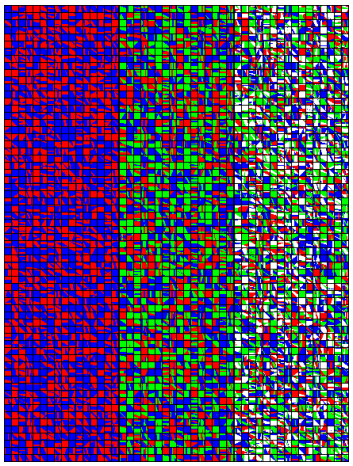

Figure 1.1. Different partition patterns

[1] This is by no means an exhaustive list of available block sizes, merely the square/cube block sizes to show data rates, with a few non square examples.

tised value from 0 to 1 to state where they are on the gradient between the given pair of colours.[1] Due to the variable number of bounding colours and individual pixels in each 128 bit block, the precision of each pixel within the block is quantised to fit in the available remaining data size.

During compression, the algorithm must select the correct distribution pattern and boundary colour pairs, then generate the quantised value for each pixel. There is a certain degree of trial and error involved in the selection of patterns and boundary colours, and when compressing, there is a trade off between compression time and final image quality. The higher the quality, the more alternatives the algorithm will try before deciding which is best. However long the compression takes, the decompression time is fixed, as the image data can always be re-extrapolated from the pattern and boundary colours in a single pass.

The compression algorithm can also use different metrics to judge the quality of different attempts, from pure value ratios of signal to noise, to a perceptual judgement weighted towards human visual acuity. The algorithm can also judge the channels individually rather than as a whole, to preserve detail for textures where the individual channels may be used as a data source for a shader program, or to reduce angular noise, which is important for tangent space normal maps.

Overall, correct usage of these options can give a marked improvement over existing compression algorithms, as shown in Figure 1.2.

[1] Indices are encoded at variable resolution using a scheme developed by researchers at Advanced Micro Devices, Inc.

Figure 1.2. From top to bottom on the right we see a close up of the original image, the 2bpp PVRTC [PVR] compressed image then the 2bpp ASTC image at the bottom.

1.4 Getting started

After downloading the evaluation compression program [Eva] the command line interface can be used to compress textures. This program supports input images in PNG, Targa, JPEG, GIF(non-animated only), BMP, Radiance HDR , Khronos Texture KTX , DirectDraw Surface DDS and Half-Float-TGA. There is also limited support for OpenEXR.

The `astcenc` application provides a full list of available command line arguments, the most basic commands are:

```
astcenc -c <input.file> <output.file> <rate> [options]
```

The `-c` tells the program to compress the first file, and save the compressed form into the second file. The rate is used to decide a block size. A block size can either be directly chosen as block size, such as 5x4 or 3x3x3, or the algorithm can be given the bpp to aim for and it chooses automatically. The bit rate must always have one decimal place, in the range 8.0 to 0.8 (or as low as 0.6 for 3D textures).

When wishing to decompress a texture to view, use the following command:

```
astcenc -d <input.file> <output.file> [options]
```

In this case, the `-d` denotes a decompression, the input file is a texture which has already been compressed, and the output file is one of the uncompressed formats.

To see what a texture would look like compressed with a given set of options, use the command:

```
astcenc -t <input.file> <output.file> <rate> [options]
```

The `-t` option compresses the file with the given options then immediately decompresses it into the output file. The interim compressed image is not saved, and the input and output files are both in a decompressed file format.

The options can be left blank but to get a good result there are a few useful ones to remember.

The most useful arguments are the quality presets:

```
-veryfast
-fast
-medium
-thorough
-exhaustive
```

There are many available options to set various compression quality factors including:

- The number and scope of block partition patterns to attempt.
- The various signal to noise cut-off levels to early out of the individual decision making stages.
- The maximum iterations of different bounding colour tests.

Most users won't explore all of these to find the best mix for their own needs. Therefore, the quality presets can be used to give a high level hint to the compressor, from which individual quality factors are derived.

It should be noted that *veryfast*, whilst almost instantaneous, gives good results only for a small subset of input images. Conversely, the *exhaustive* quality level (which does exactly what it says and attempts every possible bounding pattern combination for every block) takes a very much longer time, but will often have very little visible difference to a file compressed in thorough mode.

1.5 Using ASTC Textures

ASTC capability is a new hardware feature available from late 2013. To get started with ASTC right away the ARM® Mali™ OpenGL® ES 3.0 Emulator [Ope] is available from ARM's website and this is compatible with ASTC texture formats and as such can be used to test ASTC based programs on a standard desktop GPU.

Loading a texture in ASTC format is no different to loading other compressed texture formats, but the correct internal format must be used. Files output by the compressor have a data header containing everything needed to load the compressed texture.

```
struct astc_header
{
  uint8_t magic[4];
  uint8_t blockdim_x;
  uint8_t blockdim_y;
  uint8_t blockdim_z;
  uint8_t xsize[3];
  uint8_t ysize[3];
  uint8_t zsize[3];
};
```

Listing 1.1. ASTC Header Structure

Using the data-structure in Listing 1.1 the application can detect the important information needed to load an ASTC texture. Please see the Mali Developer Center web site for source code examples.

1.6 Quality settings

This article has already mentioned some of the high level quality settings, but there are far more precisely-targeted ways to tweak the quality of the compressor's output. The command line compressor has two main categories of argument, search parameters and quality metrics.

The algorithm for compressing a texture relies heavily upon trial and error. Many combinations of block partition and boundary colours are compared and the best one is used for that block. Widening search parameters will compare more combinations to find the right one, enabling the algorithm to find a better match, but also lengthening search time (and therefore compression time).

plimit is the maximum number of partitions tested for each block before it takes the best one found so far.

preset	plimit	dblimit [2]	oplimit	Mincorrel	bmc	maxiters
veryfast	2	18.68	1.0	0.5	25	1
fast	4	28.68	1.0	0.5	50	1
medium	25	35.68	1.2	0.75	75	2
thorough	100	42.68	2.5	0.95	95	4
exhaustive	1024	999	1000.0	0.99	100	4

Table 1.3. Preset quality factors

dblimit is the Perceptual Signal to Noise Ratio (PSNR) cut off for a block in dB. If the PSNR of a block attempt exceeds this, the algorithm considers it good enough and uses that combination. This PSNR may not be reached, since the algorithm may hit the other limits first.

oplimit implements a cut off based on comparing errors between single and dual partitioning. That is, if the dual partition errors are much worse than the single partition errors, it's probably not worth trying three or four partitions. The *oplimit* defines how much worse this error must be to give up at this point.

mincorrel defines the similarity of colour coefficients which the algorithm will try to fit in a single colour plane. The smaller this number, the more varied colours on the same plane can be, and therefore the block will not be tested with higher numbers of partitions.

bmc is a cut off for the count of block modes to attempt. The block mode defines how the individual colour values are precision weighted using different binary modes for each partition.

maxiters is the maximum cut off for the number of refining iterations to colours and weights for any given partition attempt.

These values can be set individually to extend searching in specific directions; for example:

- A texture which has lots of subtle detail should probably have a high *oplimit* to ensure subtle colour variations don't get bundled into the same partition plane.
- A texture which has very busy randomised patterns should probably search more partition types to find the right one.

Usually however these are set to default levels based on the general quality setting, as seen in Table 1.3 Improving quality with these factors is a trade off between compression time and quality. The greater the search limits and the less willing the algorithm is to accept a block as 'good enough' the

[2] dblimit defaults for levels other than exhaustive are defined by an equation based on the number of texels per block.

more time is spent looking for a better match. There is another way to get a better match though, and that is to adjust the quality metrics, altering the factors by which the compressor judges the quality of a block.

1.6.1 Channel weighting

The simplest quality factors are channel weighting, using the command line argument:

```
-ch <red-weight> <green-weight> <blue-weight> <alpha-weight>
```

This defines weighting values for the noise calculations. For example the argument `-ch 1 4 1 1` makes error on the green channel four times more important than noise on any other given channel. The argument `-ch 0.25 1 0.25 0.25` would appear to the same effect but that assumption is only partly correct. This would still make errors in the green channel four times more prevalent, but the total error would be lower, and therefore more likely to be accepted by a 'good enough' early out.

Channel weighting works well when combined with swizzling, using the `-esw` argument. For textures without alpha, for example, the swizzle `-esw rgb1` saturates the alpha channel and subsequently doesn't count it in noise calculations.

1.6.2 Block weighting

Though human eyesight is more sensitive to variations in green and less sensitive to variations in red, channel weighting has limited usefulness. Other weights can also improve a compressed texture in a number of use cases. One of these is block error checking, particularly on textures with compound gradients over large areas. By default there is no error weight based on inter-block errors. The texels at the boundary of two adjacent blocks may be within error bounds for their own texels, but with noise in opposing directions, meaning that the step between the blocks is noticeably large. This can be countered with the command line argument:

```
b <weight>
```

The equation to judge block suitability takes into account the edges of any adjacent blocks already processed. Figure 1.3 and 1.4 show this in action. However this simply makes the search algorithm accept blocks with better matching edges more readily than others, so it may increase noise in other ways. Awareness of adjacent smooth blocks can be particularly helpful for normal maps.

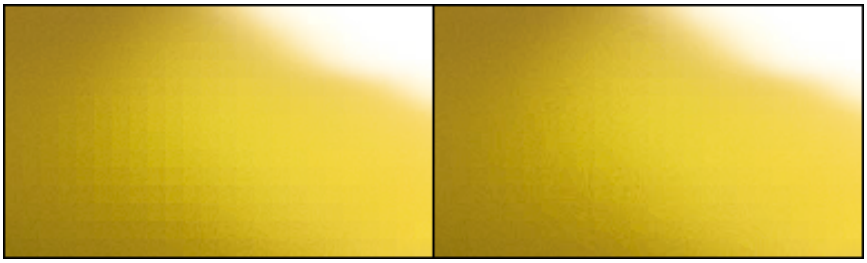

Figure 1.3. An example of block weighting. The left image shows block errors, the right image is recompressed with b 10.

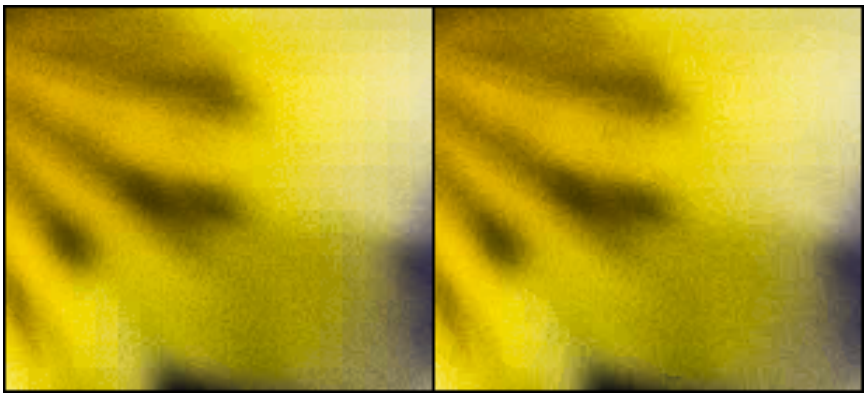

Figure 1.4. A second example of improvements from block weighting, using the same settings as Figure 1.3.

1.6.3 Normal weighting

When compressing normal maps or maps used as a data source rather than colour information, there are arguments which implement a number of additional settings all in one. These are `-normal_psnr`, `-normal_percep` and `-mask`. Only one of these should be used at a time, as they override each other.

The first two of those are geared towards the compression of 2 channel normal maps, swizzling the X and Y into luminance and alpha, overriding the default oplimit and mincorrel, and adding weight on angular error, which is far more important in normal maps. `-normal_percep` is similar but has subtly different weighting for better perceptual results. These can be seen in Figure 1.5.

Both of these functions swizzle the X and Y into luminance and alpha, with an implied `esw rrrg` argument, and also have an internal decode

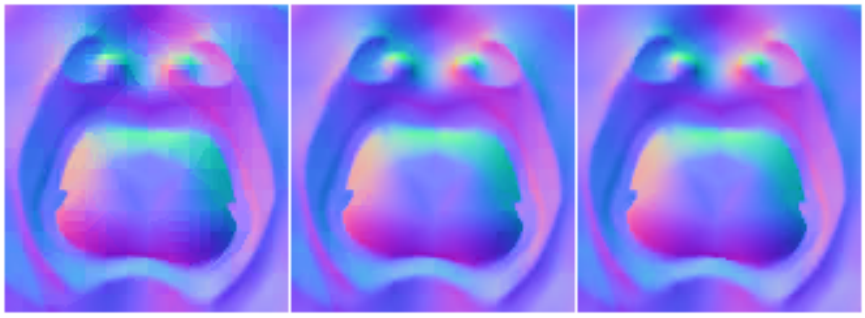

Figure 1.5. The leftmost normal map is compressed with default settings, the second uses `normal_psnr` and the normal map on the right uses `-normal_percep`

Figure 1.6. The left image is the uncompressed data, the second is compressed with default settings, the right image uses the `mask` argument.

swizzle of `dsw raz1` placing the luminance into X, the alpha into Y and reconstruct Z using:

$$z = \sqrt{1 - r^2 - a^2}$$

The argument `rn`, adds an error metric for angular noise. Other texture compression methodologies have lacked this option. Normal maps traditionally have been a problem to compress, as the minor variations in the angular component implied by the X and Y value can get ignored in pure signal to noise calculations.

1.6.4 Masking channel errors

The argument `-mask` tells the compressor to assume that the input texture has entirely unrelated content in each channel, and as such it is undesirable for errors in one channel to affect other channels.

This is shown in Figure 1.6, an example of a bitmap font where the red channel represents the characters, the blue is a rear glow and the green is a drop shadow.

The perceptual and mask filters are based upon combinations of the -v and -va arguments. These two arguments give low level access to the way an error value for a block is collected from its individual texel differences. The full syntax is:

```
-v <radius> <power> <baseweight> <avgscale> <stdevscale> <mixing-factor>
```

The radius is the area of neighbouring texels for which the average and standard deviations in error are combined using the equation:

$$weight = \frac{1}{baseweight + avgscale * average^2 + stdevscale * stdev^2}$$

The individual error values are raised to power before average and standard deviation values are calculated. The mixing-factor is used to decide if the individual channel errors are combined before or after averaging. If the mixing-factor is 0, each colour channel has its own average and standard deviation calculated, which are then combined in the equation above. If the mixing-factor is 1, the errors of each channel are combined before a single average is calculated. A value between 1 and 0 provides a combination of these two values.

The result is an erratic error value which, if just averaged over a block, can lead to a fairly noisy output being accepted. Using the standard deviation over a given radius gives the error calculations visibility of any added noise between texels, in much the same way that step changes between blocks can be checked with the block weighting (see Section 1.6.2) argument.

This equation of average noise works on the colour channels, the alpha channel is controlled separately, with a similar set of values in the command line arguments:

```
-va <baseweight> <power> <avgscale> <stdevscale>
-a <radius>
```

The alpha channel is controlled separately as, particularly with punch through textures, a little bit of quantisation noise may be preferable to softening of edges[2].

1.7 Other color formats

ASTC supports images with 1 to 4 channels, from luminance only all the way to RGBA. Additionally, the algorithm has support for different colour

[2]It's worth reassuring the reader that yes, that is quite a lot of powers and coefficients to throw at a somewhat chaotic system, so manual adjustments can often feel like a stab in the dark - hence the presets.

space encodings:

- Linear RGBA[3]
- sRGB + linear A
- HDR RGB + A[4]
- HDR RGBA

1.7.1 sRGB

ASTC supports non-linear sRGB colour space conversion both at compression and decompression time.

To keep images in sRGB colour space until the point that they are used, simply compress them in the usual way. Then when they're loaded, instead of the regular texture formats, use the sRGB texture formats. These are the ones that contain `SRGB8_ALPHA8` in the name. There's an sRGB equivalent of every RGBA format.

Helpfully the constants for the sRGB formats are always `0x0020` greater than the RGBA constants, allowing an easy switch in code between the two.

As an alternative to using sRGB texture types at run time, there is also a command line argument for the compressor to transform them to linear RGBA prior to compression. The `-srgb` argument will convert the colour space and compress the texture in linear space, to be loaded with the usual RGBA texture formats.

1.7.2 HDR

ASTC also supports HDR image formats. Using these requires no additional effort in code and the same loading function detailed above can be used. When encoding an image in a HDR format the encoder doesn't use HDR encoding by default. For this one of two arguments must be used:

```
-forcehdr_rgb
-forcehdr_rgba
```

In this mode, the encoder will use a HDR or LDR as appropriate on a per-block basis. In `-forcehdr_rgb` mode, the alpha channel (if present) is always encoded LDR. There are also the the simpler arguments:

```
-hdr
-hdra
```

[3] the most commonly understood and supported colour space
[4] HDR RGB channels with an LDR alpha

Which are equivalent to `-forcehdr_rgb` and `-forcehdr_rgba` but with additional alterations to the evaluation of block suitability (a preset `-v` and `-va`) better suited for HDR images. Also available are:

```
-hdr_log
-hdra_log
```

These are similar but base their suitability on logarithmic error. Images encoded with this setting typically give better results from a mathematical perspective but don't hold up as well in terms of perceptual artefacts.

1.8 3D Textures

The compression algorithm can also handle 3D textures at a block level. Although other image compression algorithms can be used to store and access 3D textures, they are compressed in 2D space as a series of single texel layers, whereas ASTC compresses 3 dimensional blocks of texture data, improving the cache hit rate of serial texture reads along the Z axis. Currently there are no suitably prolific 3D image formats to accept as inputs, as such encoding a 3D texture has a special syntax:

```
-array <size>
```

With this command line argument the input file is assumed to be a prefix pattern for the actual inputs, and decorate the file name with _0, _1, and so on all the way up to _<size-1>. So for example if the input file was slice.png with the argument `array 4` the compression algorithm would attempt to load files named slice_0.png, slice_1.png, slice_2.png, and slice_3.png. The presence of multiple texture layers would then be taken as a signal to use a 3D block encoding for the requested rate (see Table 1.2).

1.9 Summary

This article shows the advantages of ASTC over other currently available texture compression methodologies, and provides code to easily use ASTC texture files in an arbitrary graphics project, as well as a detailed explanation of the command line arguments to get the most out of the the evaluation codec.

ARM provides a free Texture Compression Tool [Tex] that automates the ASTC command line arguments explained in this paper via a Graphical User Interface (GUI) which simplifies the compression process and provides visual feedback on compression quality.

Bibliography

[et. al. 12] Jorn Nystad et. al. "Adaptive Scalable Texture Compression." pp. 105–114.

[Eva] "ASTC Codec and Source." Available online (http://malideveloper.arm.com/develop-for-mali/tools/astc-evaluation-codec/).

[Ope] "OpenGL ES SDK Emulator." Available online (http://malideveloper.arm.com/develop-for-mali/sdks/opengl-es-sdk-for-linux/).

[PVR] "Texture Compression using Low-Frequency SignalModulation." Available online (http://web.onetel.net.uk/~simonnihal/assorted3d/fenney03texcomp.pdf/).

[Tex] "Texture Compression Tool." Available online (http://malideveloper.arm.com/develop-for-mali/mali-gpu-texture-compression-tool/).