Area-delay Trade-offs of Texture Decompressors for a Graphics Processing Unit

Emilio Novoa Súñer, Pablo Ituero and Marisa López-Vallejo

ETSI Telecomunicación
Universidad Politécnica de Madrid, Spain

ABSTRACT

Graphics Processing Units have become a booster for the microelectronics industry. However, due to intellectual property issues, there is a serious lack of information on implementation details of the hardware architecture that is behind GPUs. For instance, the way texture is handled and decompressed in a GPU to reduce bandwidth usage has never been dealt with in depth from a hardware point of view. This work addresses a comparative study on the hardware implementation of different texture decompression algorithms for both conventional (PCs and video game consoles) and mobile platforms.

Circuit synthesis is performed targeting both a reconfigurable hardware platform and a 90nm standard cell library. Area-delay trade-offs have been extensively analyzed, which allows us to compare the complexity of decompressors and thus determine suitability of algorithms for systems with limited hardware resources.

Keywords: Texture Decompressor, Graphics Processing Unit, GPU, DXT1, DXT5, ETC

1. INTRODUCTION

The continuous growth of the video game market during the last two decades has indirectly fueled the parallel industry that provides the graphics hardware required to play these games. Modern graphics hardware systems both for PCs and video game consoles are built around a Graphics Processing Unit or GPU. The task of the GPU is to draw a scene from the raw data that describes it: a polygonal representation of the objects present, characterization of their surfaces, presence of light sources, and the position of the view camera. For the sake of a brief explanation, we could coarsely divide the work that the GPU performs in three parts. First, a *vertex processing* stage, in which the vertices that form the objects in the scene undergo a series of geometric transformations that places them in their correct position, and have their lighting calculated. Then, the *rasterizer* stage translates the geometric information of the vertices into a stream of *fragments*, that determine the activation of the individual pixels of the viewing display. Finally, a *fragment processing* stage enables to add detail and more precise lighting calculations at this individual pixel level.

Real-time applications require the generation of a number of frames per second in the order of 20 to 60, which gives a few tens of milliseconds of delay to render each single frame. GPUs capitalize on this available time by using deeply pipelined structures. Historically, this graphics pipeline was implemented as a long succession of fixed-function stages. However, the latest generations of GPUs have experienced a paradigm shift by moving to a model in which much of the work is performed by a pool of *shaders*. These shaders are simple processors that are dynamically loaded with specific programs and assigned streams of vertices or fragments to process. This architecture is referred to as the *unified shader model*.[1]

Another fundamental aspect that GPUs exploit in order to speed up the rendering process is parallelization. The graphics pipeline is devised in such a way that interdependencies are avoided as much as possible, so performance can be easily increased by replicating hardware. For example, each new model of GPU boasts a higher number of shaders in order to be able to process more vertices and fragments simultaneously.

Further author information: (Send correspondence to Pablo Ituero)
Pablo Ituero: E-mail: pituero@die.upm.es, Telephone: +34915495700 ext. 4207
This work was funded by the CICYT project DELTA TEC2009-08589 of the Spanish Ministry of Science and Innovation

A well-known bottleneck of the graphics pipeline is the access to graphics memory. More specifically, reading the data contained in textures is routinely the major consumer of available memory bandwidth.[2] Consequently, the storage of textures in compressed formats is a common way of reducing bandwidth usage. Our work concentrates in this part of the graphics processing. We have performed hardware synthesis of several texture decompression algorithms to analyze the area-speed tradeoffs offered by each algorithm. Furthermore, we have also chosen an algorithm targeting mobile platforms where we can find hard area constraints. Circuit synthesis is performed targeting both a reconfigurable hardware platform and a 90nm standard cell library. We measure the timing of the modules, which is critical for texture decompression. We also provide numeric values for FPGA resource usage and required silicon area in the standard cell implementation, which allows us to compare the complexity of decompressors and thus determine suitability of algorithms for systems with limited hardware resources. To the best of our knowledge this is the first time this comparative study is carried out, since most work around GPUs is kept secret because of high competitiveness in the hardware segment of this industry.

The structure of the paper is the following. Next, we introduce the role of the texture unit in the graphics pipeline. In section 3 we make a description of the implemented decompression algorithms and outline our proposed hardware solutions. We then describe our methodology for the FPGA and standard cell synthesis and provide the numerical results in section 4.

2. TEXTURES IN THE GRAPHICS PIPELINE

One of the key aspects when rendering convincing 3D graphics through a GPU is the use of textures. In most cases, texture data takes the form of a two dimensional array, and each element of this array is referred to as texel. Textures contain information that may be used to determine the aspect of the surfaces present on the scene, allowing to display detail at the level of the individual pixel. Therefore, it is the shaders operating on fragments which request sampling of textures. These requests are handled by dedicated texture units, that manage the access to raw texture data from graphics memory, and perform the necessary processing of this data. The availability of high quality textures and the capability to efficiently process them can have massive impact on the quality of a rendered scene.

Compressing textures for use in real-time graphics has certain peculiarities that make it different from general data compression and even image compression. Timing requirements dictate that there should be no indirections when accessing texture data, nor need to perform decompression of a substantial part of the image in order to get a specific texel. This is a must because graphics memory is not on the same chip as the GPU, so there is a high latency penalty when texture data is required. This forces texture compression systems to work at fixed compression ratios, and therefore texture compression is always lossy. Also, decompression has to be performed on the fly, possibly for many texels at the same time, and should add as little delay as possible to prevent further stalling of the pipeline. Thus, reasonably fast and small hardware decompressors must be used. Complexity of performing compression, on the other hand, does not bear so much importance, as in most cases it can be performed by software without any strict timing limitations before real-time graphics are engaged.

S3 Texture Compression (S3TC) is a set of texture compression algorithms developed (and patented[3] in 1999) by S3 Graphics Ltd., one of the companies that pioneered in the area of 3D graphics hardware. They were incorporated into Microsoft's 3D API Direct3D version 6.0 in 1998. Since then this compression system is also known as DXTC and has established itself as the standard in texture compression for both the PC and game console platforms.

In 2005, a new texture compression algorithm presently named ETC was published[4] (it was referred to as iPackman at the time of publication). ETC stands for Ericsson Texture Compression, and this algorithm aims to compete against S3TC/DXTC. Particularly, ETC is targeted at mobile architectures in which hardware resources are more limited. Thus, the objective of ETC is to provide a level of quality similar to that of S3TC/DXTC while having a smaller cost when the decompressor has to be implemented in hardware. While the quality aspect is thoroughly analyzed in,[4] when it comes to hardware implementation details it is stated in that publication that no data is readily available regarding the cost of implementing an S3TC decompressor. This makes it difficult to compare the algorithms in terms of hardware cost.

The contribution of this article is the filling of this gap by providing implementation data for ETC and two S3TC/DXTC texture decompressors.

3. DESCRIPTION AND IMPLEMENTATION OF THE ALGORITHMS

In this section we provide a brief description of our algorithms of interest, focusing on decompression mechanics and their hardware implementations.

3.1 S3TC/DXTC

S3TC/DXTC was conceived to store color and transparency information (the *alpha channel*) in textures. Five distinct algorithms are part of S3TC/DXTC, named DXT1 through DXT5. All of them are rather similar, sharing the same basic idea. Textures are subdivided in blocks of 4x4 texels for compression. For each of these blocks, a pair of reference colors (and transparency values) are stored. The possible values of color and transparency that the individual texels in the block can have are interpolations of the stored reference values. For this purpose, each texel is assigned a code that indicates which interpolation exactly corresponds to it.

We have chosen two of the five DXT variants for implementation in this paper. First, the DXT1 algorithm is selected because we have deemed it to be closest match for ETC. This is so because it is possible to use a DXT1 compressed texture that only contains color information, following the specification defined in the OpenGL extension number 309[5] (corresponding to the texture type COMPRESSED_RGB_S3TC_DXT1_EXT). The reason why we are interested in comparing ETC against a color-only texture decompressor is that, even though in[4] a couple of candidate transparency compression schemes are outlined, they are said to be still work in progress. Furthermore, the specification of ETC as in OpenGL ES extension number 5[6] only allows compression of color data. In spite of this, and for the sake of completeness we have also implemented the DXT5 algorithm. DXT5 handles both color and transparency information, so we can test the complexity of ETC versus that of a more complete codec.

3.1.1 DXT1

DXT1 encodes independently each 4x4 texel block with 64 bits of data. The two reference colors $color0$ and $color1$ are assigned 8 bits each, with an RGB565 format, that is, 5 bits for the red component, 6 for the green one, and 5 for the blue one. The remaining 48 bits are used to store a 3-bit code for each texel. These codes, along with the relative values of $color0$ and $color1$ indicate the color of the individual texels in the following way:

If $color0 > color1$:

$code = 0:$ $color0$

$code = 1:$ $color1$

$code = 2:$ $(2 \cdot color0 + color1)/3$

$code = 3:$ $(color0 + 2 \cdot color1)/3$

If $color0 \leq color1$:

$code = 0:$ $color0$

$code = 1:$ $color1$

$code = 2:$ $(color0 + color1)/2$

$code = 3:$ $black$

Figure 1 shows an outline of our proposed implementation for a combinational DXT1 decompressor. It should be noted that in order to perform interpolation of the color components it is necessary to carry out divisions, which determine the critical path of the design. Given that the divisors are always fixed, our approach to speed up the critical path has been to avoid costly divider circuits by substituting them with a multiplication by a constant.

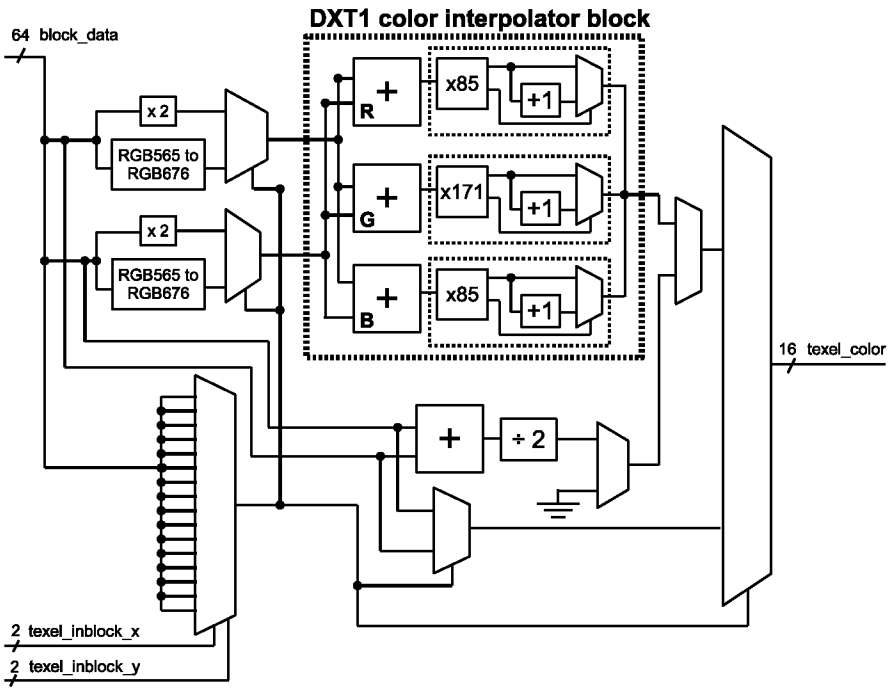

Figure 1. Diagram of the DXT1 decompressor.

3.1.2 DXT5

DXT5 requires 128 bits to store each 4x4 compressed block. Half of these bits are dedicated to color, while the other half is for transparency.

Regarding color, DXT5 works like DXT1, but always acting as if $color0 > color1$.

For transparency, two 8-bit reference values $alpha0$ and $alpha1$ are stored. The remaining 48 bits are used to store a 3-bit code for each texel. Depending on the value of this code, and the relative values of $alpha0$ and $alpha1$, the transparency value for a texel can be any of the following:

$code = 0: \quad alpha0$

$code = 1: \quad alpha1$

If $alpha0 > alpha1$:

$code = 2: \quad (6 \cdot color0 + 1 \cdot color1)/7$

$code = 3: \quad (5 \cdot color0 + 2 \cdot color1)/7$

$code = 4: \quad (4 \cdot color0 + 3 \cdot color1)/7$

$code = 5: \quad (3 \cdot color0 + 4 \cdot color1)/7$

$code = 6: \quad (2 \cdot color0 + 5 \cdot color1)/7$

$code = 7: \quad (1 \cdot color0 + 6 \cdot color1)/7$

If $alpha0 \leq alpha1$:

$code = 2: \quad (4 \cdot color0 + 1 \cdot color1)/5$

$code = 3: \quad (3 \cdot color0 + 2 \cdot color1)/5$

$code = 4: \quad (2 \cdot color0 + 3 \cdot color1)/5$

$code = 5: \quad (1 \cdot color0 + 4 \cdot color1)/5$

$code = 6: \quad 0$

$code = 7: \quad 255$

In figure 2 we show our implementation of a DXT5 decompressor. Again, dividers are substituted with constant multipliers here, both in the part dedicated to color (which is partially shared with the DXT1 module) and the part that handles transparency. Although the color part of DXT5 is simpler than that of DXT1, the addition of the transparency circuitry results in a bigger design.

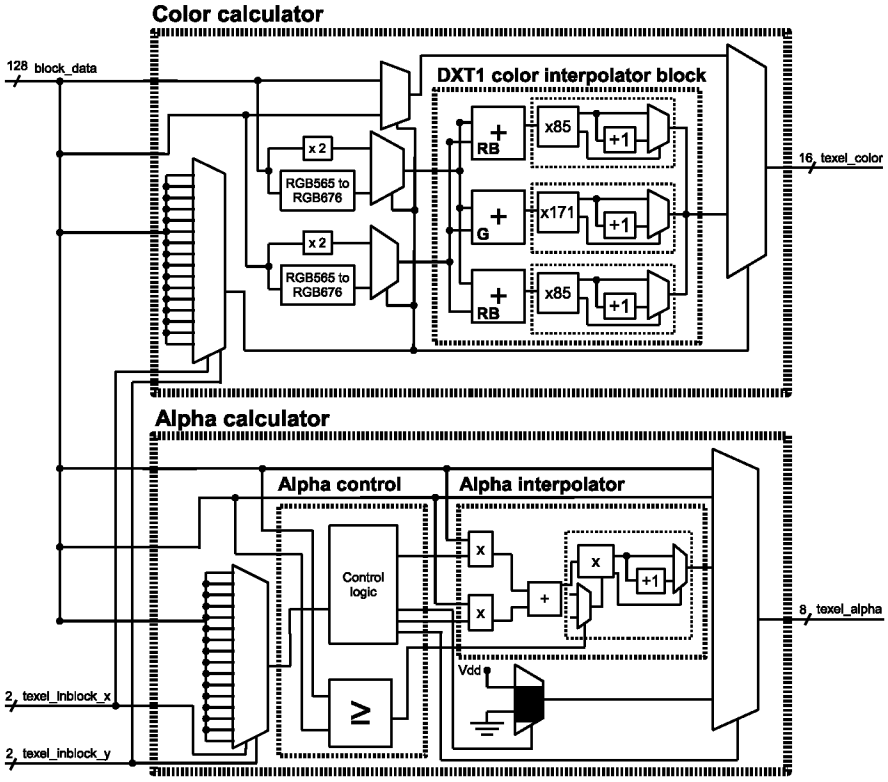

Figure 2. Diagram the DXT5 decompressor.

3.2 ETC

An ETC stored texture produces RGB888 texel values as output. The algorithm is also block based with a block size of 4 by 4 texels. Each block is subdivided in two halves or sub-blocks, either horizontally or vertically.

To produce the final color for a specific texel, ETC needs two pieces of data: the *base color* of the sub-block in which the texel lies, and a luminance modifier, which is individually encoded for each texel. The color of the texel is reconstructed by adding the luminance modifier to the base color.

There are two modes in which the base colors of the sub-blocks can be encoded in the ETC compressed block: differential and non-differential. In non-differential mode, each sub-block is assigned an RGB444 base color. In differential mode, the first sub-block is assigned an RGB555 color and the second sub-block has three 3-bit values (stored in 2's complement) that indicate how much the base color (that is, its R, G and B components) of the second sub-block deviates from that of the first one.

For the luminance, each sub-block is assigned one of eight existing predefined luminance tables ($2 \cdot 3 = 6$ bits required). Then, for each texel we have a 2-bit index that assigns a luminance modifier value from its corresponding table (totaling $16 \cdot 2 = 32$ bits).

The layout of the ETC decompressor can be seen in figure 3. Note how no multiplications are required for the implementation of this algorithm. This absence of any arithmetical operations other than simple addition is the key point by which this algorithm bests S3TC/DXTC in simplicity and speed.

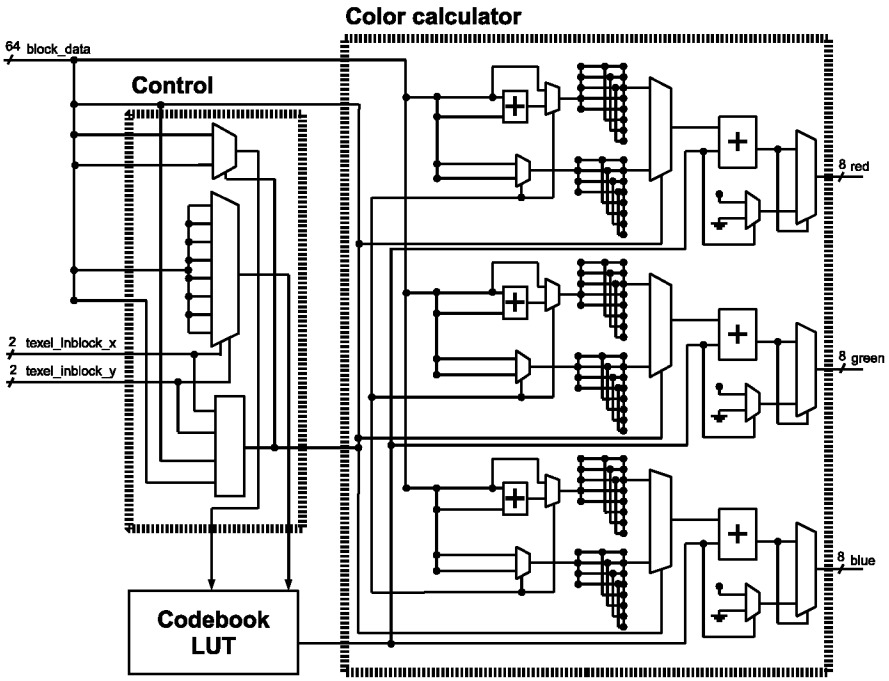

Figure 3. Diagram of the ETC decompressor.

4. EXPERIMENTAL RESULTS

This section presents the results from synthesizing the previously depicted decompressor modules. Their VHDL hardware descriptions have been sythesized to two different platforms: an FPGA, and a 90nm standard cell library.

4.1 FPGA

The ETC, DXT1 and DXT5 decompressors have been synthesized targeting the Xilinx Virtex2P XC2VC30 FPGA. It must be said, though, that the natural target for a GPU is direct implementation on silicon, so metrics taken from an FPGA synthesis should be taken as a coarse indication of the comparison in a more realistic situation. However, some research groups show interest in implementing GPU related modules on them.[7]

Table 4.1 summarizes the results of the FPGA synthesis regarding the use of slices and embedded multipliers and their speed.

	FPGA RESOURCES		**DELAY**
	Slices	Multipliers	
ETC	75	0	10 ns
DXT1	96	3	14.8 ns
DXT5	82	6	16.5 ns

Table 1. FPGA Synthesis Results.

As expected, ETC manages to be faster and take up less slices than both DXT1 and DXT5. Note that this occurs despite the fact that DXT1 and DXT5 happen to take advantage of the embedded multipliers in the FPGA, whose absence would result in the need of a substantial amount of additional slices in order to perform the required multiplications.

4.2 Standard cell

We have synthesized the decompressors to a 90nm standard cell library from Faraday Technology Corporation operating at a nominal voltage of 1V. The area-delay design space for each module has been thoroughly explored by making a number of synthesis passes with varying design constraints. The results are given in figure 4. This plot presents the design space that a GPU hardware designer has to explore when evaluating the most suitable texture compression algorithm for a given set of area and delay constraints.

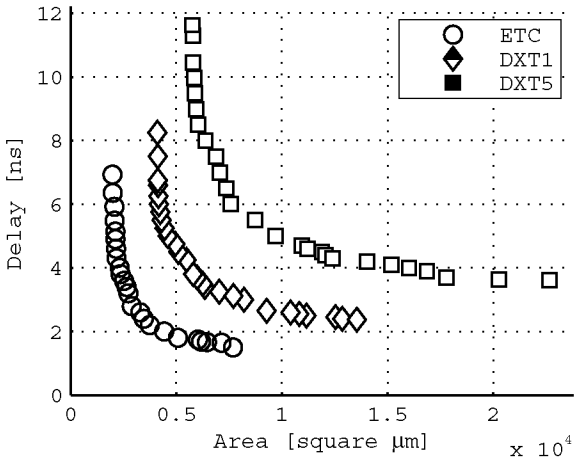

Figure 4. Area-delay design space for the standard cell synthesis.

Analyzing figure 4, we have verified that ETC is indeed less costly in terms of hardware resources compared with S3TC/DXTC. Comparing with DXT1, savings in area are in the order of 50% for a given delay constraint in the 2 to 7 nanosecond range. This makes ETC a suitable candidate for texture compression/decompression in systems with limited hardware resources as is the case of mobile platforms.

5. CONCLUSION

At present time, there is little information about low-level details of GPU related hardware. In this work we have shed some light over texture decompressors. This block belonging to the texture unit is relatively small, although significant because of the impact of compression in bandwidth usage. In particular, ETC and two S3TC/DXTC decompressors have been designed and synthesized to FPGA and 90nm standard cell technologies. The implementation of these algorithms has allowed us to prove that ETC is currently the best alternative for compressing color textures in systems with strict hardware constraints, such as mobile platforms.

REFERENCES

[1] Moya, V., González, C., Roca, J., Fernández, A., and Espasa, R., "Shader Performance Analysis on a Modern GPU Architecture," *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (November 2005).

[2] Aila, T., Miettinen, V., and Nordlund, P., "Delay Streams for Graphics Hardware," *ACM Transactions on Graphics* **22**(3), 792–800 (2003).

[3] Iourcha, K., Nayak, K., and Hong, Z., "System and method for fixed-rate block-based image compression with inferred pixel values," *US Patent 5,956,431* .

[4] Akenine-Möller, T. and Ström, J., "iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones," *Graphics Hardware* , 63–70 (2005).

[5] "OpenGL extension #309 at the OpenGL Registry.." `http://www.opengl.org/registry/specs/EXT/texture_compression_dxt1.txt`.

[6] "OpenGL ES extension #5 at the OpenGL ES Registry.." `http://www.khronos.org/registry/gles/extensions/OES/OES_compressed_ETC1_RGB8_texture.txt`.

[7] Thomas, D. B. and Luk, W., [*Implementing Graphics Shaders Using FPGAs*], Lecture Notes in Computer Science. Springer Berlin (2004).