Fast CPU DXT Compression

# Introduction

DXT compression is a lossy texture compression algorithm that is supported by Direct3D. DXT compression reduces storage requirements and bus traffic. DXT textures are usually compressed offline and consumed at runtime. However, it is sometimes necessary to perform DXT compression during runtime. For example, a more efficient texture compression algorithm can be used to store textures on disk. When the textures are needed they can be decoded from disk and encoded to the DXT format during runtime. This type of scenario can be useful for virtual texturing (Id Software, 2009).

Fast DXT compressors can be implemented on the CPU (Waveren, 2006) and the GPU (Castano, 2007). While GPU implementations are usually faster, CPU implementations are sufficiently fast for real-time applications and consume less bus traffic. Which implementation should be chosen is workload dependent. If GPU resources are scarce, then a CPU implementation should be chosen and vice-versa.

This paper discusses a CPU implementation of DXT compression based on (Waveren, 2006). Two improvements have been made to the code. First, the code has been converted from assembly to SIMD intrinsics. This allows the compiler to perform scheduling and register allocation, which improves performance by as much as 15% in the 64-bit build. Second, a task manager (Minadakis, 2011) and Intel Threading Building Blocks are used to improve performance by approximately 150%.

# DXT Compression

A brief summary of the DXT compression algorithm is presented next. For a more thorough explanation of the algorithm, the reader is encouraged to read (Waveren, 2006).

DXT compression works on a block of 4 by 4 colors. For each block in the texture, the DXT compression algorithm first finds a line through color space by calculating a bounding box. Next, 4 colors are chosen on the line. The maximum and minimum colors represent the endpoints of the line and two additional equally-spaced intermediate points are calculated. Finally, each color in the color block is flattened by choosing the closest color on the line.

The maximum and minimum colors are stored in RGB565 format and each color in the block is stored as a 2-bit index to one of the 4 colors on the line through color space. For DXT1 compression (without alpha), 64 bytes are compressed to 8 bytes (8:1 compression ratio). To compress the alpha channel, a separate line through alpha space is found and the alpha values in the block are flattened to one of 8 alpha values along the line. Two additional bytes are needed to store the maximum and minimum alpha value, and 3 bits per color are required to store the alpha index to one of the 8 alpha values. For DXT5 compression (with alpha), 64 bytes are compressed to 16 bytes (4:1 compression ratio).

There are several ways to implement DXT compression. These variations typically trade performance for quality. The DXT compression algorithm used in this paper is chosen for speed. The quality of the algorithm is a matter of taste. My own view is that the quality is acceptable with respect to the performance trade-offs.

# Optimizations

The DXT compression implementation used in this sample has several optimizations. SSE2 instructions are used to vectorize the code. The SSE2 implementation operates on 4 colors at a time and completely avoids conditional branches. The only difference between the original implementation presented by (Waveren, 2006) and the implementation in this sample is that the assembly has been converted to compiler intrinsics. This allows the extra registers to be utilized in 64-bit builds and results in up to a 15% boost in speed.

Intel Threading Building Blocks (TBB) is used to multi-thread the DXT compression algorithm. TBB is a library that allows developers to distribute and load balance a set of tasks across all available physical cores of a CPU. Tasks are high-level mechanisms that abstracts work from low-level physical threads. The DXT compression task functions (e.g. CompressImageDXT1Task) each compress a set of spatially continuous blocks in the texture. The number of blocks in the set can affect the performance of TBB. A very small number of blocks can negatively affect the memory access pattern and result in slow performance.

This sample also uses the task manager created by (Minadakis, 2011). The task manager is used for convenience in this sample because it simplifies TBB task creation and synchronization. The CompressImageDXTTBB function uses the task manager to create, execute, and wait for the DXT compression tasks.

# Sample Usage

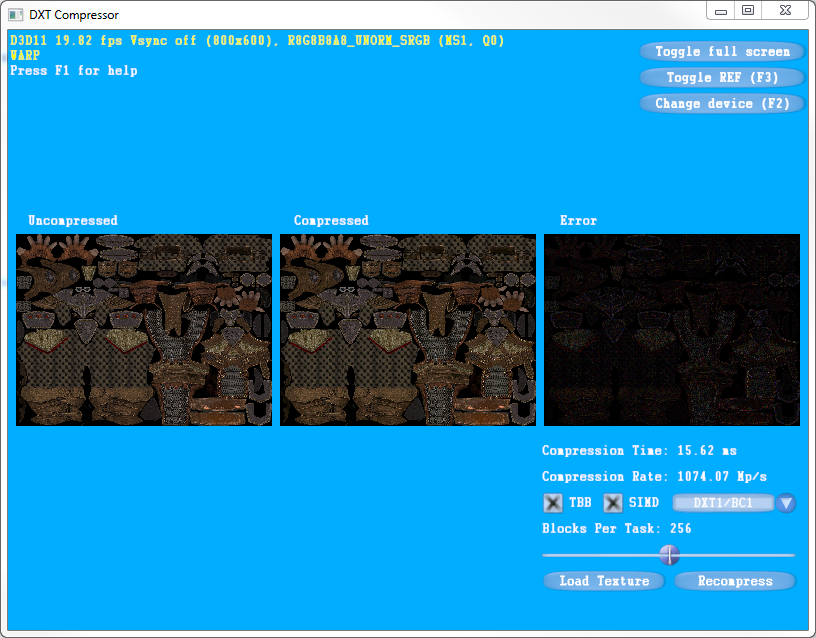


Figure . Screenshot of the DXT Compressor sample.

Figure 1 shows a screenshot of the DXT Compressor sample. The sample displays 3 textured quads. The leftmost quad displays an uncompressed texture. The middle quad displays a compressed texture using the DXT compression algorithm discussed in this paper. The rightmost quad displays the absolute error in each texel.

The “Compression Time” field displays the amount of time, in milliseconds, that it took to compress the texture. The “Compression Rate” field displays the rate of compression in megapixels per second. When the sample is started, it loads and compresses the default texture shown in Figure 1. However, because the compression tasks are often competing with the initialization of the sample, the texture is recompressed again after the sample has been running for several frames. This improves the accuracy of the compression time and compression rate for the default texture.

The TBB and SIMD checkboxes enable TBB optimizations and SIMD optimizations, respectively. The combo box can be used to switch between DXT1 compression (without alpha) and DXT5 compression (with alpha). The “Blocks Per Task” slider changes the number of blocks each task compresses.

The “Load Texture” button can be used to select a DDS texture from disk. The sample will load and compress the texture and compute the absolute error. The “Recompress” button can be used to force the compression algorithm to be rerun. Only the compression algorithm and error computation will run when the “Recompress” button is pressed. The texture is not reloaded from disk. Note that if any of the compression options are changed, the sample will automatically recompress the texture and update the performance meters.

# Performance Results

Table . DXT compression rates.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Compressor | Scalar | TBB | SIMD | TBB + SIMD |
| DXT1 | 55 Mp/s | 186 Mp/s | 331 Mp/s | 831 Mp/s |
| DXT5 | 34 Mp/s | 130 Mp/s | 234 Mp/s | 576 Mp/s |

Table 1 shows the DXT compression rates, in megapixels per second, for a 2.2GHz Sandy Bridge mobile processor with 4 cores/8 threads. The average rate over 10 trials is shown. The scalar column represents non-vectorized single-threaded code. The TBB column represents non-vectorized multi-threaded code. The SIMD column represents vectorized single-threaded code. The TBB + SIMD column represents vectorized multi-threaded code.

Adding TBB to the scalar code improves performance by over 200% while adding TBB to SIMD code improves performance by approximately 150%. A TBB + SIMD implementation achieves very fast compression rates. A 4k by 4k texture can be compressed in fewer than 20 milliseconds on the test machine.

# Future Work

The current SSE2 implementation of the DXT compressor relies on integer instructions. Consequently, AVX instructions cannot be used to improve performance because AVX does not support integer operations yet. Once supported, it should be trivial to update the DXT compressor to use AVX instructions.

# Works Cited

Castano, I. (2007, February). High Quality DXT Compression using CUDA.

Id Software. (2009, August). From Texture Virtualization to Massive Parallelization.

*Intel Threading Building Blocks*. (n.d.). Retrieved from http://threadingbuildingblocks.org/

Minadakis, Y. (2011, March). Using Tasking to Scale Game Engine Systems.

Waveren, J. v. (2006). Real-Time DXT Compression.