Fast CPU DXT Compression

Introduction

DXT compression is a lossy texture compression algorithm that can reduce texture storage requirements and decrease texture bandwidth. DXT decompression is typically hardware accelerated, which makes it very fast and efficient. 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).

Real-time DXT compression algorithms can be implemented on both the CPU (Waveren, 2006) and the GPU (Castano, 2007). While GPU implementations are usually faster, CPU versions are sufficiently fast for real-time applications and result in less bus traffic. Which implementation should be chosen is workload dependent and should be based on where the application is bottlenecked.

This paper discusses a CPU implementation of DXT compression based on J.P. Van Waveren's article "Real-Time DXT Compression" (Waveren, 2006). Two improvements have been made to the code from that article. 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 (http://threadingbuildingblocks.org/) are used to improve performance by approximately 150% (see Table 1). A brief summary of the DXT compression algorithm is presented next.

DXT Compression

DXT compression works on a block of 4 by 4 texels. For each block in the texture, the DXT compression algorithm quantizes the color of each texel to a set of 4 colors. These 4 colors represent equidistant points on a line through color space. The line through color space is calculated by computing the bounding box of the colors in the block, which simply involves computing the maximum and minimum color. The maximum and minimum colors represent the endpoints of the line and two additional equidistant points that lie between the endpoints are calculated. The color of each texel in the block is flattened by choosing the closest color on the line. For each block, DXT compression stores the maximum and minimum color and 16 indices that reference one of the 4 principal colors.

The maximum and minimum colors are stored in RGB565 format and 2 bits are used to store each index. 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 to emphasize speed.

For a more thorough explanation of the DXT compression algorithm, the reader is encouraged to read J.P. Van Waveren's article "Real-Time DXT Compression" (Waveren, 2006).

Optimizations

SSE2 instructions are used to vectorize the code. The SSE2 implementation operates on 4 colors at a time and avoids conditional branches. This optimization results in a 500% performance increase (see Table 1).

There are two main differences between the original implementation presented by (Waveren, 2006) and the implementation in this sample. First, 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. Second, the DXT compressor has been multi-threaded.

Intel® Threading Building Blocks (Intel® TBB) is used to multi-thread the DXT compression algorithm. Intel TBB is a library that allows developers to distribute and load balance a set of tasks across all available physical cores of a CPU. The DXT compression task functions (e.g. `CompressImageDXT1Task`) each compress a set of spatially continuous blocks in the texture. Individual task size can be set to balance CPU utilization and cache efficiency. Many small tasks can be spread be spread evenly to achieve good utilization of hardware threads, but excessively small tasks 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 Intel TBB task creation and synchronization. The `CompressImageDXTTBB` function uses the task manager to create, execute, and wait for the DXT compression tasks.

Performance Results

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. *DXT compression rates.*

Error! Reference source not found. shows the DXT compression rates, in megapixels per second, for a 2.2GHz Core i7™ 2675QM processor with 4 cores/8 hardware threads running the 64-bit Windows 7* OS. The average rate over 10 trials is shown. The test texture, which is loaded by default in the sample, is 4096 by 4096 in size. The scalar column represents non-vectorized single-threaded code. The Intel TBB

column represents non-vectorized multi-threaded code. The SIMD column represents vectorized single-threaded code. The Intel TBB + SIMD column represents vectorized multi-threaded code.

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

Sample Usage

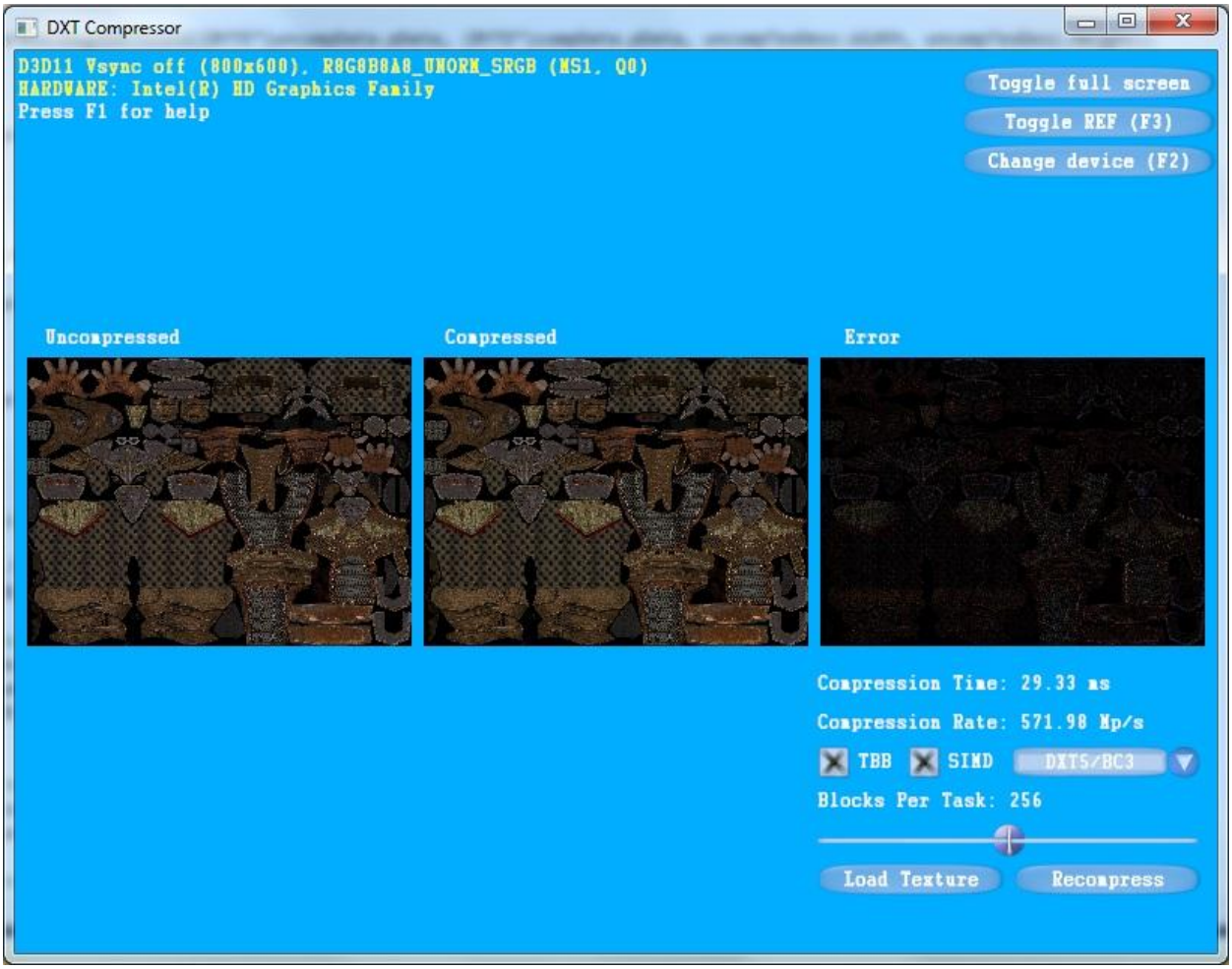

Figure 1. *Screenshot of the DXT Compressor sample*.

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 Intel TBB and SIMD checkboxes enable Intel 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.

Future Work

The current SSE2 implementation of the DXT compressor relies heavily on integer instructions. Intel® AVX support is planned for this sample when AVX2 is released, which expands integer instructions to 256 bits.

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.