High Quality DXT Compression using OpenCL for CUDA

Ignacio Castaño
icastano@nvidia.com

March 2009

Document Change History

Version	Date	Responsible	Reason for Change
0.1	02/01/2007	Ignacio Castaño	First draft
0.2	19/03/2009	Timo Stich	OpenCL version

Abstract

DXT is a fixed ratio compression format designed for real-time hardware decompression of textures. While it's also possible to encode DXT textures in real-time, the quality of the resulting images is far from the optimal. In this white paper we will overview a more expensive compression algorithm that produces high quality results and we will see how to implement it using CUDA to obtain much higher performance than the equivalent CPU implementation.

Motivation

With the increasing number of assets and texture size in recent games, the time required to process those assets is growing dramatically. DXT texture compression takes a large portion of this time. High quality DXT compression algorithms are very expensive and while there are faster alternatives [1][9], the resulting quality of those simplified methods is not very high.

The brute force nature of these compression algorithms makes them suitable to be parallelized and adapted to the GPU. Cheaper compression algorithms have already been implemented [2] on the GPU using traditional GPGPU approaches. However, with the traditional GPGPU programming model it's not possible to implement more complex algorithms where threads need to share data and synchronize.

How Does It Work?

In this paper we will see how to use CUDA to implement a high quality DXT1 texture compression algorithm in parallel. The algorithm that we have chosen is the cluster fit algorithm as described by Simon Brown [3]. We will first provide a brief overview of the algorithm and then we will describe how did we parallelize and implement it in CUDA.

DXT1 Format

DXT1 is a fixed ratio compression scheme that partitions the image into 4x4 blocks. Each block is encoded with two 16 bit colors in RGB 5-6-5 format and a 4x4 bitmap with 2 bits per pixel. Figure 1 shows the layout of the block.

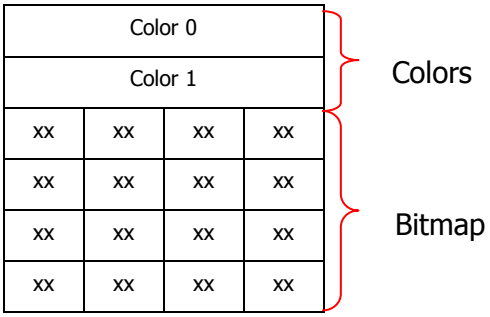

Figure 1. DXT1 block layout

The block colors are reconstructed by interpolating one or two additional colors between the given ones and indexing these and the original colors with the bitmap bits. The number of interpolated colors is chosen depending on whether the value of 'Color 0' is lower or greater than 'Color 1'.

The total size of the block is 64 bits. That means that this scheme achieves a 6:1 compression ratio. For more details on the DXT1 format see the specification of the OpenGL S3TC extension [4].

Cluster Fit

In general, finding the best two points that minimize the error of a DXT encoded block is a highly discontinuous optimization problem. However, if we assume that the indices of the block are known the problem becomes a linear optimization problem instead: minimize the distance from each color of the block to the corresponding color of the palette.

Unfortunately, the indices are not known in advance. We would have to test them all to find the best solution. Simon Brown [3] suggested pruning the search space by considering only the indices that preserve the order of the points along the least squares line.

Doing that allows us to reduce the number of indices for which we have to optimize the endpoints. Simon Brown provided a library [5] that implements this algorithm. We use this library as a reference to compare the correctness and performance of our CUDA implementation.

The next section goes over the implementation details.

OpenCL Implementation

Partitioning the Problem

We have chosen to use a single work group to compress each 4x4 color block. Work items that process a single block need to cooperate with each other, but DXT blocks are independent and do not need synchronization or communication. For this reason the number of work groups is equal to the number of blocks in the image.

We also parameterize the problem so that we can change the number of work items per block to determine what configuration provides better performance. For now, we will just say that the number of work items is N and later we will discuss what the best configuration is.

During the first part of the algorithm, only 16 work items out of N are active. These work items start reading the input colors and loading them to local memory.

Finding the best fit line

To find a line that best approximates a set of points is a well known regression problem. The colors of the block form a cloud of points in 3D space. This can be solved by computing the largest eigenvector of the covariance matrix. This vector gives us the direction of the line.

Each element of the covariance matrix is just the sum of the products of different color components. We implement these sums using parallel reductions.

Once we have the covariance matrix we just need to compute its first eigenvector. We haven't found an efficient way of doing this step in parallel. Instead, we use a very cheap sequential method that doesn't add much to the overall execution time of the group.

Since we only need the dominant eigenvector, we can compute it directly using the Power Method [6]. This method is an iterative method that returns the largest eigenvector and only requires a single matrix vector product per iteration. Our tests indicate that in most cases 8 iterations are more than enough to obtain an accurate result.

Once we have the direction of the best fit line we project the colors onto it and sort them along the line using brute force parallel sort. This is achieved by comparing all the elements against each other as follows:

```
cmp[tid] = (values[0] < values[tid]);
cmp[tid] += (values[1] < values[tid]);
cmp[tid] += (values[2] < values[tid]);
cmp[tid] += (values[3] < values[tid]);
cmp[tid] += (values[4] < values[tid]);
cmp[tid] += (values[5] < values[tid]);
cmp[tid] += (values[6] < values[tid]);
cmp[tid] += (values[7] < values[tid]);
cmp[tid] += (values[8] < values[tid]);
cmp[tid] += (values[9] < values[tid]);
cmp[tid] += (values[10] < values[tid]);
cmp[tid] += (values[11] < values[tid]);
cmp[tid] += (values[12] < values[tid]);
cmp[tid] += (values[13] < values[tid]);
cmp[tid] += (values[14] < values[tid]);
cmp[tid] += (values[15] < values[tid]);
```

The result of this search is an index array that references the sorted values. However, this algorithm has a flaw, if two colors are equal or are projected to the same location of the line, the indices of these two colors will end up with the same value. We solve this problem comparing all the indices against each other and incrementing one of them if they are equal:

```
if (tid > 0 && cmp[tid] == cmp[0]) ++cmp[tid];
if (tid > 1 && cmp[tid] == cmp[1]) ++cmp[tid];
if (tid > 2 && cmp[tid] == cmp[2]) ++cmp[tid];
if (tid > 3 && cmp[tid] == cmp[3]) ++cmp[tid];
if (tid > 4 && cmp[tid] == cmp[4]) ++cmp[tid];
if (tid > 5 && cmp[tid] == cmp[5]) ++cmp[tid];
if (tid > 6 && cmp[tid] == cmp[6]) ++cmp[tid];
if (tid > 7 && cmp[tid] == cmp[7]) ++cmp[tid];
if (tid > 8 && cmp[tid] == cmp[8]) ++cmp[tid];
if (tid > 9 && cmp[tid] == cmp[9]) ++cmp[tid];
if (tid > 10 && cmp[tid] == cmp[10]) ++cmp[tid];
if (tid > 11 && cmp[tid] == cmp[11]) ++cmp[tid];
if (tid > 12 && cmp[tid] == cmp[12]) ++cmp[tid];
if (tid > 13 && cmp[tid] == cmp[13]) ++cmp[tid];
if (tid > 14 && cmp[tid] == cmp[14]) ++cmp[tid];
```

During all these steps only 16 work items are being used. For this reason, it's not necessary to synchronize them. All computations are done in parallel and at the same time step, because 16 is less than the warp size on NVIDIA GPUs.

Index evaluation

All the possible ways in which colors can be clustered while preserving the order on the line are known in advance and for each clustering there's a corresponding index. For 4 clusters there are 975 indices that need to be tested, while for 3 clusters there are only 151. We pre-compute these indices and store them in global memory.

We have to test all these indices and determine which one produces the lowest error. In general there are indices than work items. So, we partition the total number of indices by the number of work items and each work item loops over the set of indices assigned to it.

It's tempting to store the indices in constant memory, but since indices are used only once for each work group, and since each work item accesses a different element, coalesced global memory loads perform better than constant loads.

Solving the Least Squares Problem

For each index we have to solve an optimization problem. We have to find the two end points that produce the lowest error. For each input color we know what index it's assigned to it, so we have 16 equations like this:

$$\alpha_i a + \beta_i b = x_i$$

Where $\{\alpha_i, \beta_i\}$ are $\{1,0\}$, $\{1/3, 2/3\}$, $\{1/2, 1/2\}$, $\{2/3, 1/3\}$ or $\{0,1\}$ depending on the index and the interpolation mode. We look for the colors a and b that minimize the least square error of these equations. The solution of that least squares problem is the following:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum \alpha_i^2 & \sum \alpha_i \beta_i \\ \sum \alpha_i \beta_i & \sum \beta_i^2 \end{pmatrix}^{-1} \cdot \begin{pmatrix} \sum \alpha_i x_i \\ \sum \beta_i x_i \end{pmatrix}$$

Note: The matrix inverse is constant for each index set, but it's cheaper to compute it every time on the kernel than to load it from global memory. That's not the case of the CPU implementation.

Computing the Error

Once we have a potential solution we have to compute its error. However, colors aren't stored with full precision in the DXT block, so we have to quantize them to 5-6-5 to estimate the error accurately. In addition to that, we also have to take in mind that the hardware expands the quantized color components to 8 bits replicating the highest bits on the lower part of the byte as follows:

```
R = (R << 3) | (R >> 2);
G = (G << 2) | (G >> 4);
B = (B << 3) | (B >> 2);
```

Converting the floating point colors to integers, clamping, bit expanding and converting them back to float can be time consuming. Instead of that, we clamp the color components, round the floats to integers and approximate the bit expansion using a multiplication. We found the factors that produce the lowest error using an offline optimization that minimized the average error.

```
r = round(clamp(r,0.0f,1.0f) * 31.0f);
g = round(clamp(g,0.0f,1.0f) * 63.0f);
b = round(clamp(b,0.0f,1.0f) * 31.0f);
r *= 0.03227752766457f;
g *= 0.01583151765563f;
b *= 0.03227752766457f;
```

Our experiment show that in most cases the approximation produces the same solution as the accurate solution.

Selecting the Best Solution

Finally, each work item has evaluated the error of a few indices and has a candidate solution. To determine which work item has the solution that produces the lowest error, we store the errors in local memory and use a parallel reduction to find the minimum. The winning work item writes the endpoints and indices of the DXT block back to global memory.

Implementation Details

The source code is divided into the following files:

- `DXTCompression.cl`: This file contains OpenCL implementation of the algorithm described here.
- `permutations.h`: This file contains the code used to precompute the indices.
  `dds.h`: This file contains the DDS file header definition.

Performance

We have measured the performance of the algorithm on different GPUs and CPUs compressing the standard Lena. The design of the algorithm makes it insensitive to the actual content of the image. So, the performance depends only on the size of the image.

Figure 2. Standard picture used for our tests.

As shown in Table 1, the GPU compressor is at least 10x faster than our best CPU implementation. The version of the compressor that runs on the CPU uses a SSE2 optimized implementation of the cluster fit algorithm. This implementation pre-computes the factors that are necessary to solve the least squares problem, while the GPU implementation computes them on the fly. Without this CPU optimization the difference between the CPU and GPU version is even larger.

Table 1. Performance Results

Image	Tesla C1060	Geforce 8800 GTX	Intel Core 2 X6800	AMD Athlon64 Dual Core 4400
Lena 512x512	83.35 ms	208.69 ms	563.0 ms	1,251.0 ms

We also experimented with different number of work-items, and as indicated in Table 2 we found out that it performed better with the minimum number.

Table 2. Number of Work Items

64	128	256
54.66 ms	86.39 ms	96.13 ms

The reason why the algorithm runs faster with a low number of work items is because during the first and last sections of the code only a small subset of work items is active.

A future improvement would be to reorganize the code to eliminate or minimize these stages of the algorithm. This could be achieved by loading multiple color blocks and processing them in parallel inside of the same work group.

Conclusion

We have shown how it is possible to use OpenCL to implement an existing CPU algorithm in parallel to run on the GPU, and obtain an order of magnitude performance improvement. We hope this will encourage developers to attempt to accelerate other computationally-intensive offline processing using the GPU.

References

[1] "Real-Time DXT Compression", J.M.P. van Waveren.
www.intel.com/cd/ids/developer/asmo-na/eng/324337.htm

[2] "Compressing Dynamically Generated Textures on the GPU", Oskar Alexandersson, Christoffer Gurell, Tomas Akenine-Möller.
http://graphics.cs.lth.se/research/papers/gputc2006/

[3] "DXT Compression Techniques", Simon Brown.
http://www.sjbrown.co.uk/?article=dxt

[4] "OpenGL S3TC extension spec", Pat Brown.
http://www.opengl.org/registry/specs/EXT/texture_compression_s3tc.txt

[5] "Squish – DXT Compression Library", Simon Brown.
http://www.sjbrown.co.uk/?code=squish

[6] "Eigenvalues and Eigenvectors", Dr. E. Garcia.
http://www.miislita.com/information-retrieval-tutorial/matrix-tutorial-3-eigenvalues-eigenvectors.html

[7] "An Experimental Analysis of Parallel Sorting Algorithms", Guy E. Blelloch, C. Greg Plaxton, Charles E. Leiserson, Stephen J. Smith
http://citeseer.ist.psu.edu/blelloch98experimental.html

[8] "NVIDIA CUDA Compute Unified Device Architecture Programming Guide".

[9] NVIDIA OpenGL SDK 10 "Compress DXT" sample
http://developer.download.nvidia.com/SDK/10/opengl/samples.html#compress_DXT

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com