ISPC Texture Compressor

1 Introduction

This document presents an overview of the techniques used for BC7 compression in the ISPC Texture Compressor software. We make references to functions in `kernel.ispc`

The BC7 texture format[3] has 8 'modes', but they all share a similar approach, which is also the core of the BC1 (aka DXT1/S3TC) texture format: Representing pixels with a quantized interpolation of two colors. This is thus the core problem we have to solve, and we'll discuss our optimization approach in section 2.

For modes that can divide a block into multiple partitions, there still remains to determine which partition to use. We could simply encode all possible partitions (up to 64) and keep the best result. However this can be very time consuming, thus we use a fast pruning scheme, explained in section 3, to reduce the set of candidate partitions.

Some modes can encode a single channel in isolation from the others. This is a simpler problem, and the three other channels are encoded as usual. The decision that remains is which channel to isolate. Exhaustive search is affordable here (4 candidates), and in some cases (for RGBA textures) isolation of the alpha channel is by far the strongest candidate.

Experimentation shows that some modes are far more likely to be effective than others, thus we can shorten encoding time significantly if we ignore less promising possibilities. On the other hand, if time permits, quality can be slightly improved if we conduct a more exhaustive search. Those tradeoffs can be achieved through encoder settings, and we made a few preset available.

Our implementation is highly optimized and exploits SIMD instruction sets. We use the Intel SPMD Program Compiler[2] (ISPC) for this purpose, which allows the codebase to be fairly simple. However, this still introduces some challenges that we'll discuss in section 4.

2 Partition optimization

The main problem we seek to solve during encoding is to find a pair of endpoints such that projection and quantization will produce as little distortion as possible. To simplify notation we'll consider here a partition over all 16 pixels in a block, and only three channels. This is similar to BC1 compression, and our approach is indeed close to some existing BC1 encoders[1]. Extension to smaller partitions and four channels is trivial.

For our purposes, we will try to reduce unweighted mean square error:

$$\min_{\substack{\mathbf{a},\mathbf{b}\in\mathcal{P}\\ \mathbf{q}\in\mathcal{Q}^{16}}} \sum_{i}^{16} \|\mathbf{x}_i - (\mathbf{a} + q_i(\mathbf{b}-\mathbf{a}))\|_2^2 = \|\mathbf{X} - (\mathbf{1}\mathbf{a}^T + \mathbf{q}(\mathbf{b}-\mathbf{a})^T)\|_2^2 \tag{1}$$

Here, $\mathcal{P}$ is the subset of $\mathbb{R}^3$ in which the endpoints are represented (16-bit RGB for BC1) and $\mathcal{Q}$ is the quantization levels for each pixel ($\mathcal{Q} = \{0, 1/3, 2/3, 1\}$ for BC1). $\mathbf{X} \in \mathbb{R}^{3\times 16}$ represents the source pixels.

This is a integer bilinear program, and finding optimal solutions is hard enough to be impractical. We thus consider a few approximations. First, we iteratively optimize $\mathbf{a}, \mathbf{b}$ and $\mathbf{q}$ to break the bilinear program into two least square problems. We also relax endpoints $\mathbf{a}, \mathbf{b}$ as $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$. For a given quantization $\mathbf{q}$ there is no interaction between channels, thus can we solve a_c, b_c separately for each channel c:

$$\min_{a_c, b_c \in \mathcal{R}} \|\mathbf{x}_c - (\mathbf{1}a_c + \mathbf{q}(b_c - a_c))\|_2^2 = \| \begin{bmatrix}(\mathbf{1} - \mathbf{q}) & \mathbf{q}\end{bmatrix} \begin{bmatrix}a_c \\ b_c\end{bmatrix} - \mathbf{x}_c\|_2^2 \tag{2}$$

We can efficiently solve this least square problem (see `opt_endpoints`). We then project $\mathbf{a}, \mathbf{b}$ back to $\mathcal{P}$ (see `ep_quant` functions). Closing the iteration loop, for a given $\mathbf{a}, \mathbf{b}$, there is no dependency over pixels, thus we can solve our other least square problem for each q_i in isolation:

$$\min_{q_i \in \mathcal{Q}} \|\mathbf{x}_i - (\mathbf{a} + q_i(\mathbf{b} - \mathbf{a}))\|_2^2 \propto \| \frac{(\mathbf{b} - \mathbf{a})^T}{\|\mathbf{b} - \mathbf{a}\|^2}(\mathbf{x}_i - \mathbf{a}) - q_i\|_2^2 \tag{3}$$

$$q_i = \text{proj}_{\mathcal{Q}} \frac{(\mathbf{b} - \mathbf{a})^T}{\|\mathbf{b} - \mathbf{a}\|^2}(\mathbf{x}_i - \mathbf{a}) \tag{4}$$

We see that each value q_i can be obtained by scalar quantization (see `block_quant`).

In order to start the iterative refinement of $\mathbf{a}, \mathbf{b}$ and $\mathbf{q}$, we need an initial value. For this purpose, we relax constraints, and allow $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$ and $\mathbf{q} \in \mathbb{R}^{16}$.

Setting $\mathbf{d} = \mathbf{b} - \mathbf{a}$ and $\hat{\mathbf{x}} = \sum_i^{16} \mathbf{x}_i$, we can show that

$$\min_{\substack{\mathbf{a}, \mathbf{b} \in \mathbb{R}^3 \\ \mathbf{q} \in \mathbb{R}^{16}}} \|\mathbf{X} - (\mathbf{1}\mathbf{a}^T + \mathbf{q}(\mathbf{b} - \mathbf{a})^T)\|_2^2 = \|(\mathbf{X} - \mathbf{1}\hat{\mathbf{x}}^T) - \mathbf{q}\mathbf{d}^T)\|_2^2 \tag{5}$$

We recognize rank-1 PCA here, which is easy to compute (in our implementation, we use power iteration). We adjust $\mathbf{a}, \mathbf{b}$ such that $q_i \in [0, 1]$ (see `block_segment`). Then we discard $\mathbf{q}$ and start iterative refinement. A few iterations (one or two) are typically enough to converge.

3 Fast partition pruning

Iterative refinement requires a significant amount of computation. This becomes quite expensive when we evaluate all 64 possible partitions. Thus we first rank partitions by applying crude endpoint selection and quantization (see `bc7_enc_mode01237_part_fast`). We then use iterative refinement on the most promising partition.

Even with this optimization, we still have to apply per-pixel quantization 64 times. Ideally, we would like to be able to reduce the number of candidate partitions even further. As we have seen in the previous section, using PCA, we can obtain an exact solution to the partition optimization problem when the unknowns are relaxed to be reals.

Relaxation gives us a lower bound, but it's not sharp enough to eliminate a significant amount of candidates. Using that bound directly to rank candidate partitions, however, yields effective results: In practice, testing a few partitions with the best PCA bounds is almost as good as exhaustive search.

We also use the fact that the sum of the covariance matrices from each partition is equal to the covariance matrix for the whole block. We thus precompute the covariance matrix for the whole block and use the linear relation to infer the covariance matrix for the last partition (see `pca_bound_split`). We also reduce the number of power iterations in this context, as it has little impact on the quality of the ranking.

In practice, we have only implemented this optimization for two partitions, as our testing shows three partitions to bring only small improvements, despite their higher complexity. Exhaustive search for three partitions is still available, and enabled in slower profiles.

4 ISPC implementation

In order to take advantage of the SIMD capability of modern processors, we use the Intel SPMD Program Compiler[2]. We map each program instance to a 4x4 texture block, and thus typically compress 4 (SSE2-SSE4) or 8 (AVX-AVX2) blocks at once. SIMD execution makes it inefficient to execute different code paths for each block. This creates a challenge when we process per-block candidate lists, or encode blocks into their bit-packed representation, where the format depends on the blocks' mode. To address this issue, we use various techniques:

After partition pruning, we are left with different pattern lists to handle in each block. We sort the lists using branchless insertion sort (see `partial_sort_list`), which is effective at partial sorting, and only requires one scatter operation per sorted item.

For the initial PCA and endpoint optimization, we proceed through all pixels for each pattern, but filter results using the pattern's bitmask. For quantization, endpoints from the current pixel's partition are loaded for each pixel with gather.

For bit-packing, we keep the best representation found so far. For each mode, we always encode the final candidate, and if it outperforms previous modes, we overwrite the saved representation. One difficulty here is that each pattern has a pixel with an implicit most significant bit (to remove the degree of freedom in swapping endpoints). This introduces variable bitfield positions in the representation. We resolve this by using an altered fixed encoding, producing up to 131 bits, and then using variable shifts to remove the implicit bits (see `bc7_code_adjust_skip_mode01237`).

References

[1] F. Giesen. DXT1/DXT5 compressor `http://nothings.org/stb/stb_dxt.h`

[2] Intel. Intel SPMD Program Compiler `http://ispc.github.io/`

[3] Microsoft. BC7 Format. `http://msdn.microsoft.com/library/hh308953.aspx`