High-Quality, Fast DX11 Texture Compression with ISPC

Marc Fauconneau
March 04, 2015

Legal

Outline

- Introduction to DX11 Texture Compression
- The Fast ISPC Texture Compressor
- Overview of the DX11 Formats
- Fast Texture compression: Algorithms
- Effective SIMD acceleration with ISPC
- Future Work

Introduction to DX11 Texture Compression

DX11 Texture Formats

- 4x4 Blocks, 128 bits per block

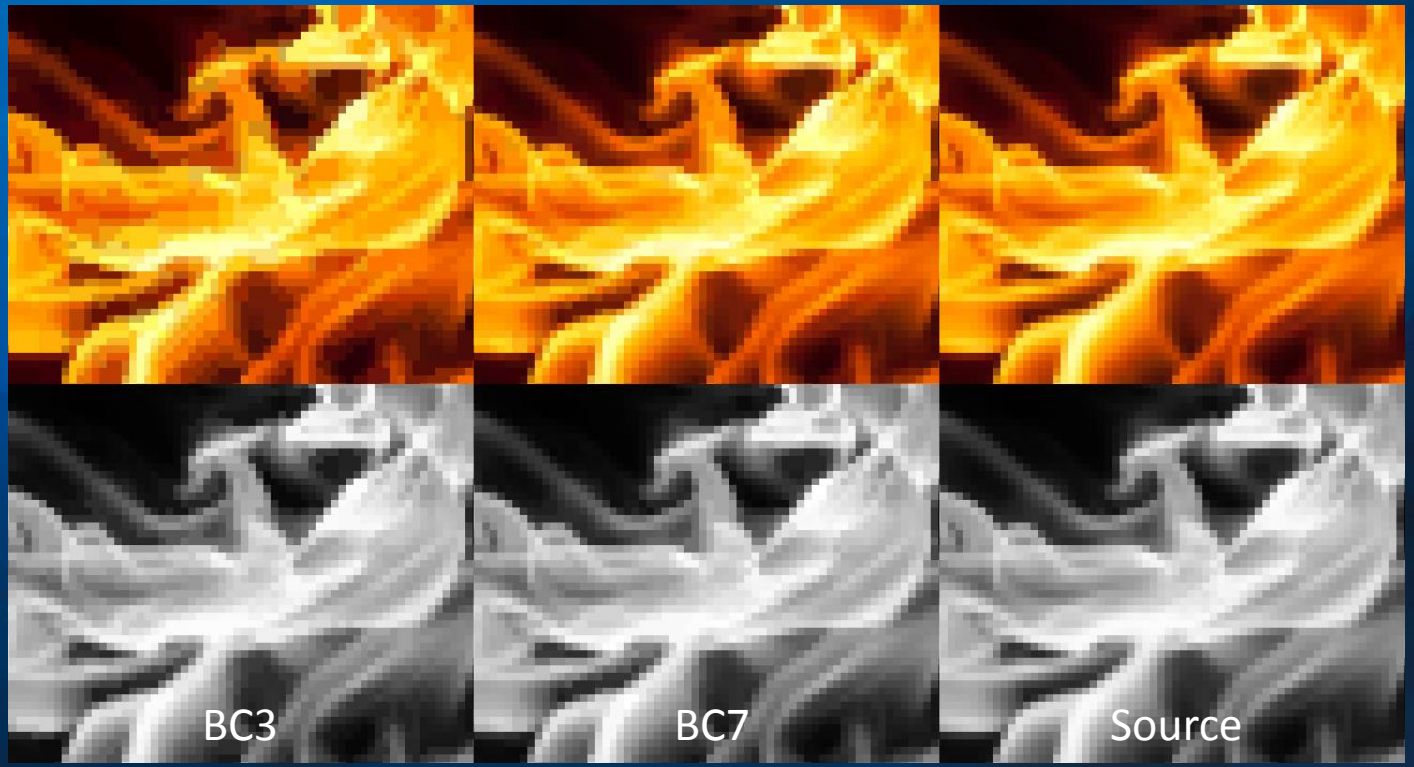
- BC7
  - RGB or RGBA, LDR (8-bit)
  - 4:1 compression for R8G8B8A8
  - Very high quality RGB (far better than BC1/S3TC, due to more bits)
  - Great replacement for BC3 (higher quality, same size)

- BC6H
  - RGB, HDR (16-bit)
  - 4:1 compression for R9G9B9E5_SHAREDEXP (8:1 for 4ch FP16)

BC7 in action

RGB

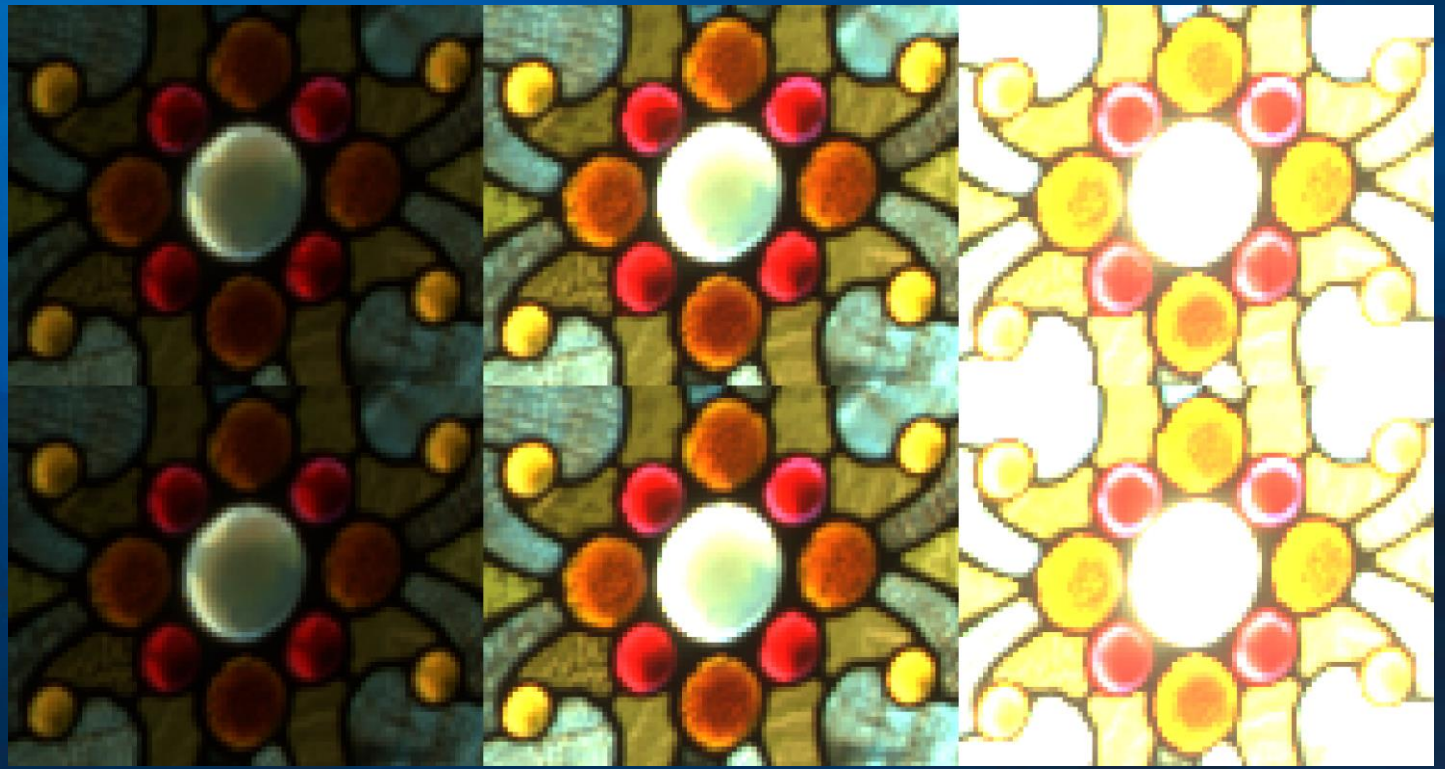
Alpha



BC6H in action

BC6H

Source



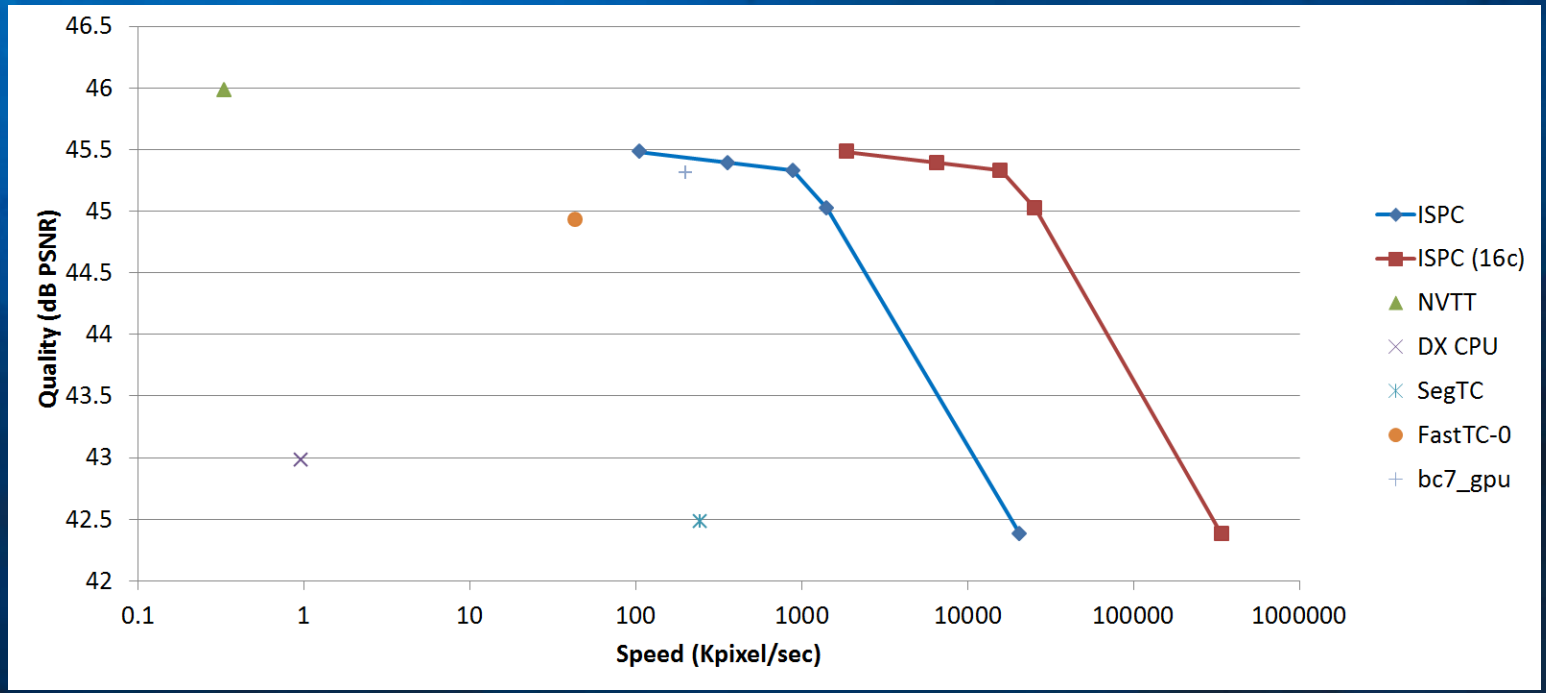
Fast ISPC Texture Compressor

ISPC Texture Compressor

- Driven by ISV demand for a good, fast compressor
- State of the art compressor
  - Efficient pruning scheme
  - SIMD implementation using ISPC (Intel SPMD Compiler)
- Permissive license, easy to use
  - Has been integrated into several content pipelines

State of the art

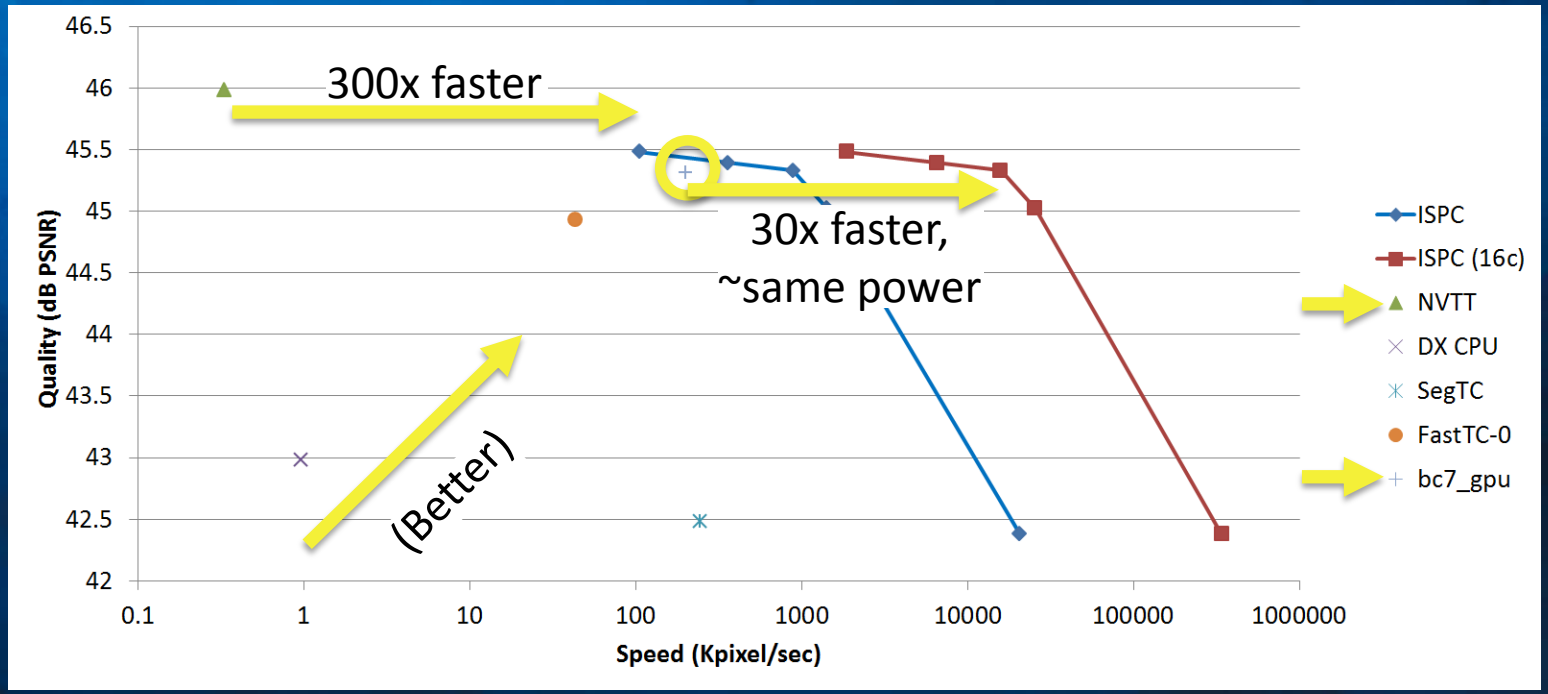
- Performance of available BC7 compressors on Kodak dataset (RGB only)



- Independent testing confirms those results: http://gamma.cs.unc.edu/FasTC/

State of the art

- Performance of available BC7 compressors on Kodak dataset (RGB only)



- Independent testing confirms those results: http://gamma.cs.unc.edu/FasTC/

Easy to integrate

- Builds as a DLL with a C interface
  - Encoder settings are highly configurable
  - A set of profiles spanning a wide curve of performance/quality trade-offs is included
- No external dependencies (other than ISPC)
- No hidden multithreading: Responsibility of the calling application (everyone has their preferences)

Mandatory sample code

```
bc6h_enc_settings settings;
GetProfile_bc6h_basic(&settings);
CompressBlocksBC6H(input, output, &settings);
```

Mandatory sample code

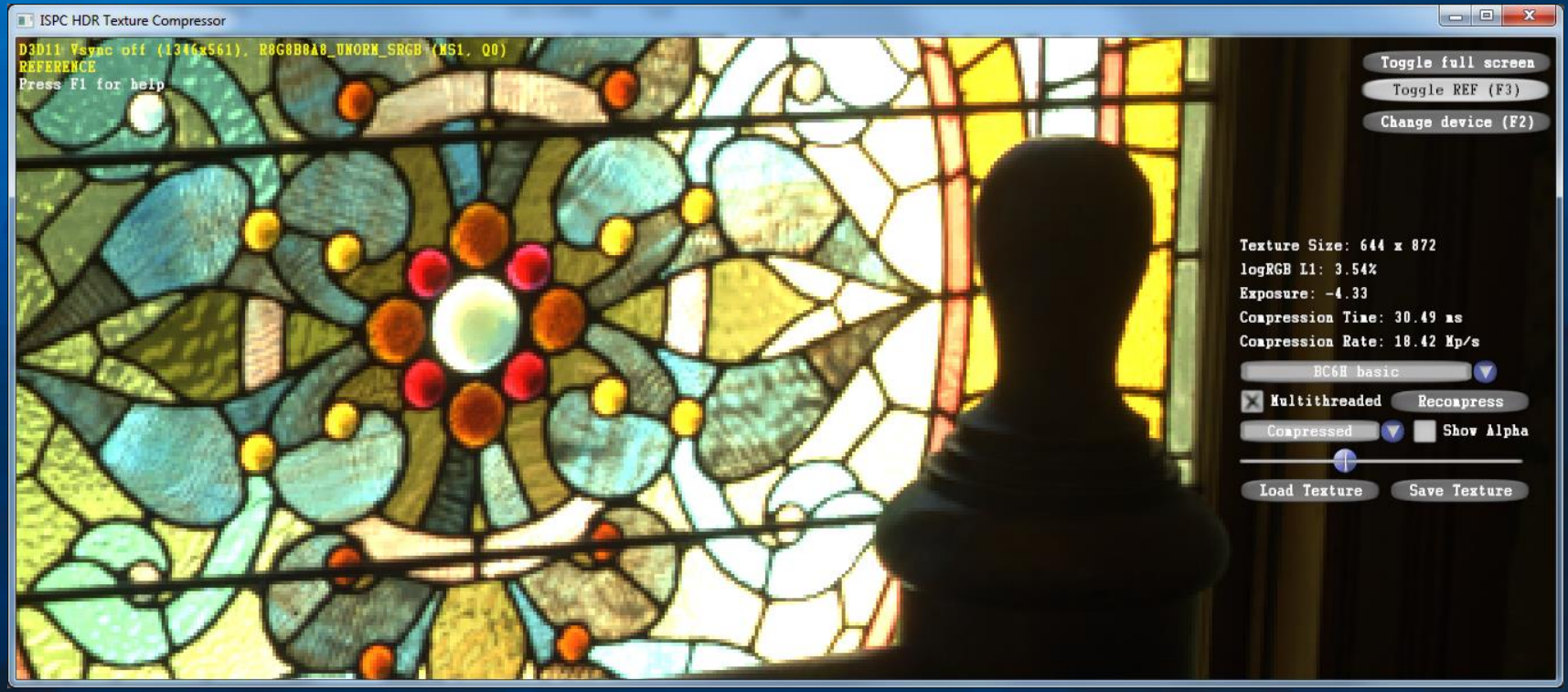
```
D3D11_MAPPED_SUBRESOURCE uncompData, compData;
deviceContext->Map(uncompTex, 0, D3D11_MAP_READ_WRITE, 0, &uncompData);
deviceContext->Map(compTex, 0, D3D11_MAP_READ_WRITE, 0, &compData);

rgba_surface input;
input.ptr = (BYTE*)uncompData.pData;
input.stride = uncompData.RowPitch;
input.width = uncompTexDesc.Width;
input.height = uncompTexDesc.Height;

BYTE* output = (BYTE*)compData.pData;

bc6h_enc_settings settings;
GetProfile_bc6h_basic(&settings);
CompressBlocksBC6H(input, output, &settings);
```

Sample application



- https://software.intel.com/en-us/articles/fast-ispc-texture-compressor-update

Overview of the DX11 Formats

Overview of the DX11 Formats

- This is not a reference
  - See MSDN for that:
    https://msdn.microsoft.com/en-us/library/hh308955(v=vs.85).aspx
- I will not talk about specifics like bit packing…
- Those formats build upon BC1 aka DXT1 aka S3TC
  - Per-pixel interpolation between two endpoints

Coding tools

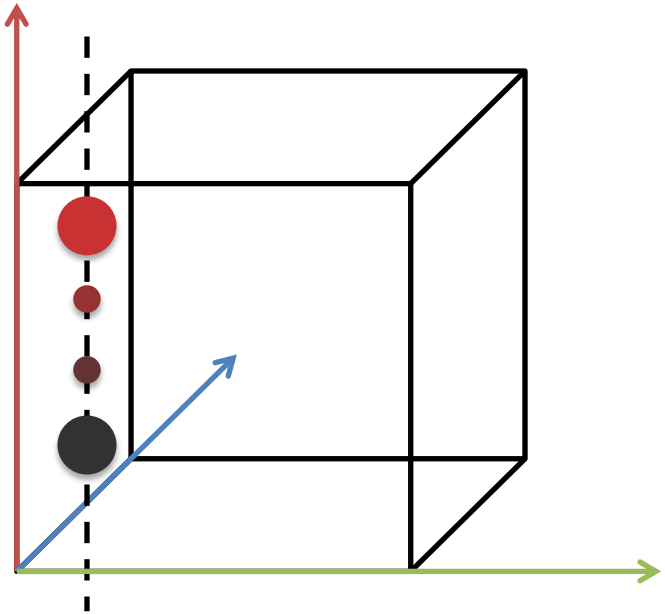
- Each block is set to one mode (14 in BC6H, 8 in BC7)
  - It uniquely defines how all values in the block are packed
- Some modes enable specific features:
  - Multiple partitions
  - RGBA endpoints (BC7)
  - Encoding two sets of weights (BC7)
  - Differential endpoint encoding (BC6H)

Endpoint interpolation

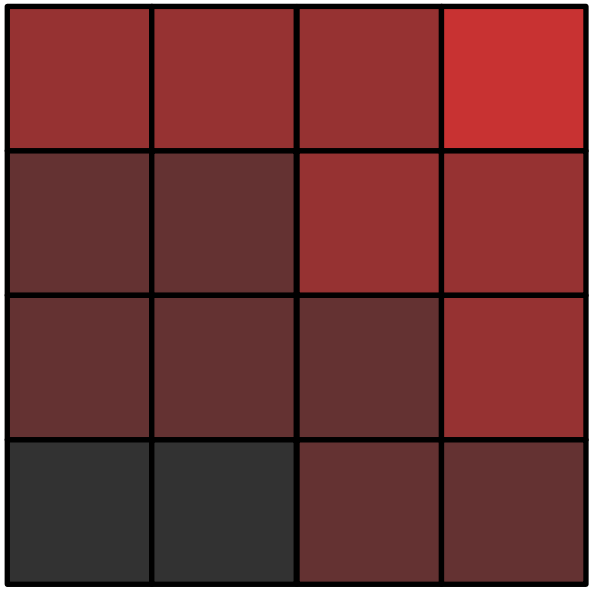
- Like in BC1 aka DXT1 aka S3TC
  - More variation in endpoint/weight sizes
  - Avoids the unequal channel quantization problem (RGB565)
- This technique exploits local color correlation/contrast
  - It fails if pixels are not clustered along a line in RGB(A) space
  - It introduces more error in high-contrast blocks

Endpoint interpolation

RGB Cube

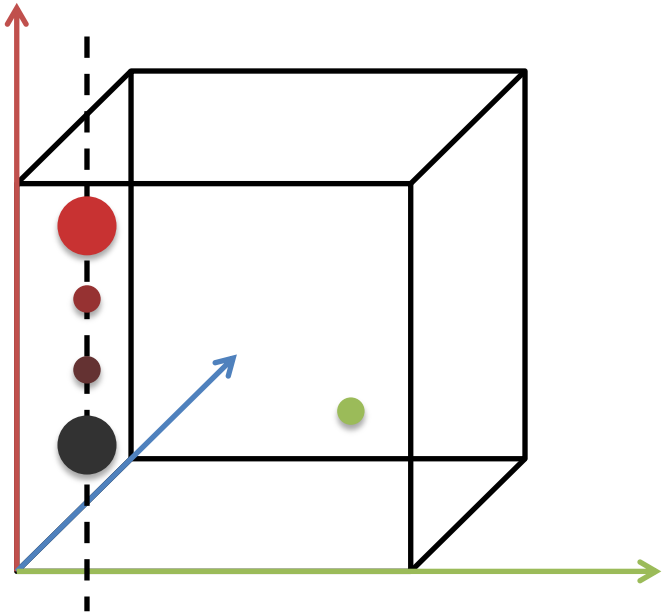


Block

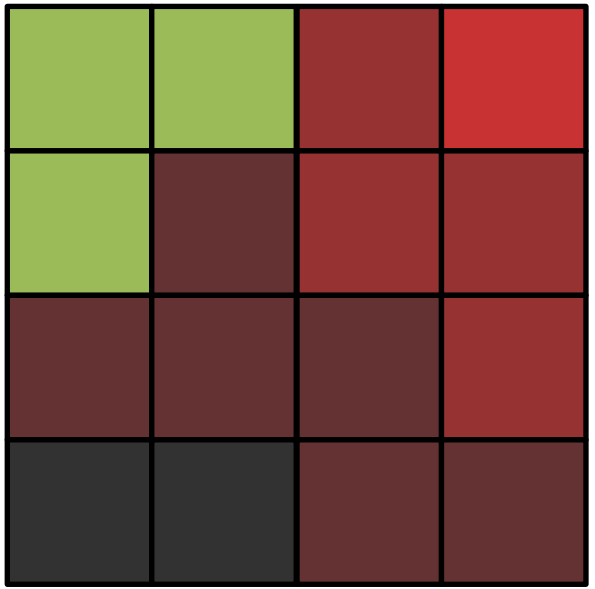


Endpoint interpolation

RGB Cube



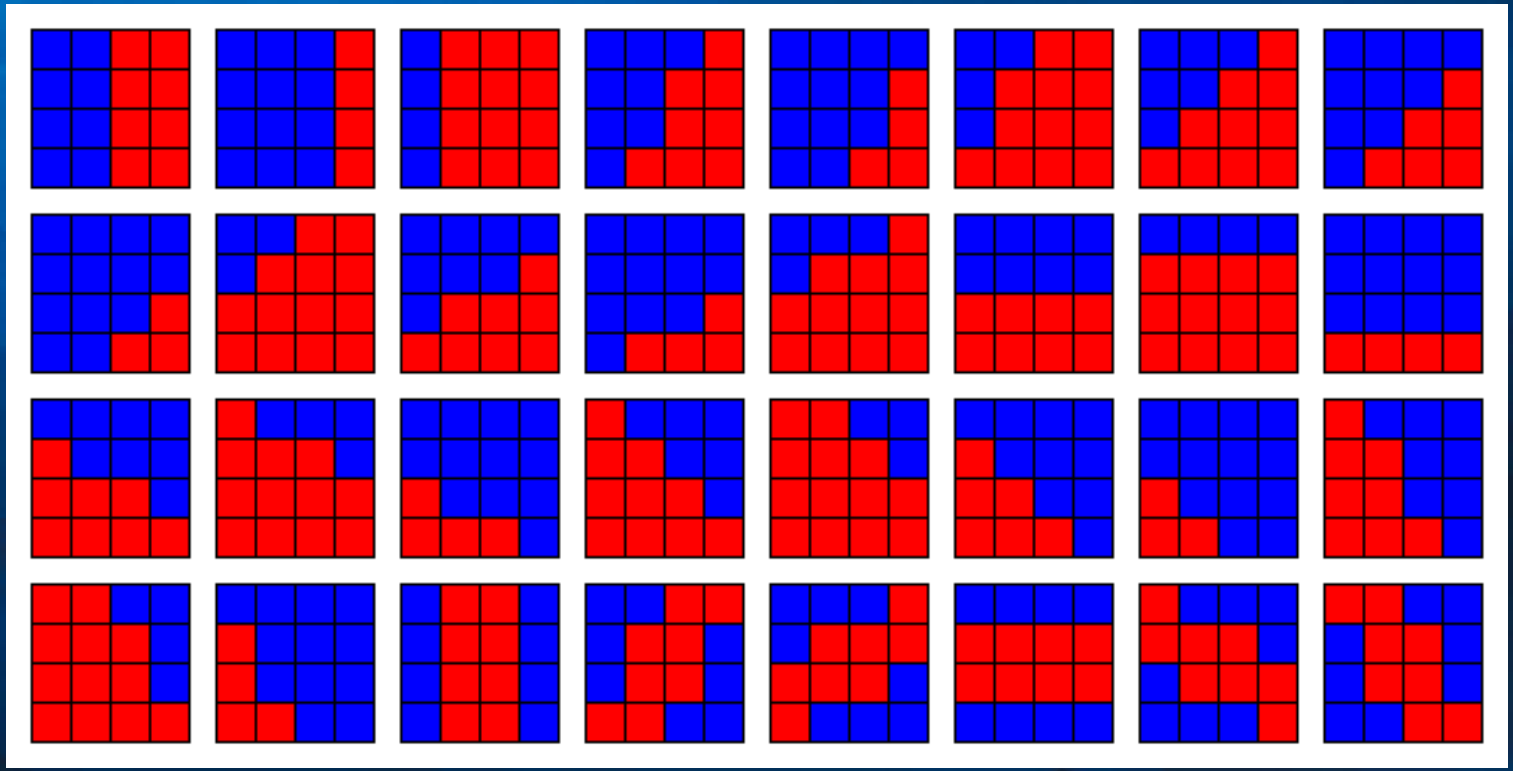
Block



Multiple partitions

- Some modes split the 4x4 block into 2 or 3 partitions
  - Each partition has it's own pair of endpoints
  - Pre-defined table with 64 ways to split
- This drastically reduces error in blocks presenting difficult combinations of colors or contrast

First 32 partitions



BC7 specifics

- RGB or RGBA endpoints
  - Many modes only encode RGB, assume Alpha=1.0
  - Very effective for transparent textures with opaque regions
- Some modes can encode two sets of weights
  - A “vector” channel and “scalar” channel (any of R,G,B,A)
  - Bounds error for uncorrelated alpha (behaves like BC3)
  - Also very effective at representing certain RGB blocks

BC7 modes

- Multiple partitions (modes 0, 1, 2, 3, 7)
  - 3 partitions, RGB, 3 or 2 bits per weight (mode 0, 2)
  - 2 partitions, RGB, 3 or 2 bits per weight (mode 1, 3)
  - 2 partition, RGBA, 2 bits per weight (mode 7)
- Two sets of weights (modes 4, 5)
  - 1 partition, RGBA, 2+3 or 3+2 bits per weight (mode 4)
  - 1 partition, RGBA, 2+2 bits per weight (mode 5)
- Vanilla (mode 6)
  - 1 partition, RGBA, 4 bits per weight (mode 6)

BC6H specifics

- Differential endpoint encoding
  - The first endpoint is subtracted from all other endpoints
  - Deltas can fit in less bits (amount defined by each mode)
  - Some modes allow more bits in a given channel
- Effective on blocks with low dynamic range
  - Absolute encoding almost never used in practice

BC6H modes

- Differential endpoint encoding (mode 1-9, 12-14)
  - 2 partition, 3 bits per weight (mode 1-9)
  - 1 partition, 4 bits per weight (mode 12-14)
- Absolute endpoint encoding (mode 10, 11)
  - 2 partition, 3 bits per weight (mode 10)
  - 1 partition, 4 bits per weight (mode 11)

Fast Texture compression: Algorithms

Fast Texture compression: Algorithms

- Local optimization
  - The encoder processes each 4x4 block in isolation
- Parameter search
  - Mode, Partition, Endpoints, Weights…
- Encoding
  - Endpoint swaps, swizzles, bit packing…

Parameter search

- Essentially a nested loop: Mode>Partition>Endpoints
- Mode
  - BC7: Exhaustive search
  - BC6H: First-fit (with tweaks)
- Partition
  - Fast partition pruning
- Endpoints, Weights
  - Iterative refinement with PCA initialization

Iterative refinement

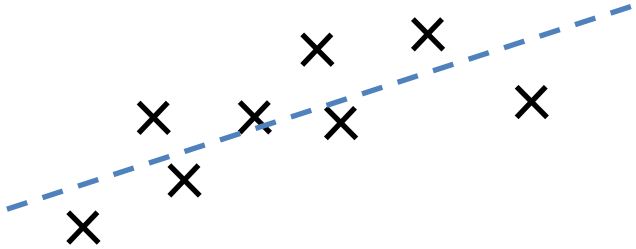
- For each partition, find the best endpoints and weights
  - Integer bilinear programming (NP-hard)
- But we can find the “best” endpoints given weights
  - Least squares (trivial)
- And the best weights given endpoints
  - Scalar quantization (trivial)

- This is iterative refinement (this is what stb_dxt.h does)

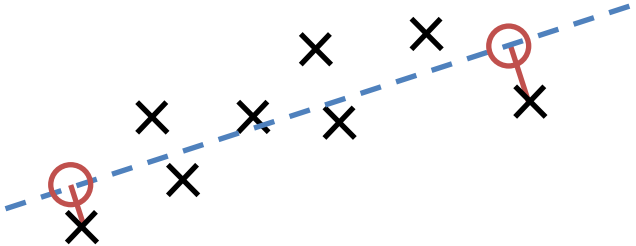
PCA initialization

- We need something to start the iterations
  - What if we didn't have to quantize weights?
  - This would reduce to projecting all points to a line
- Finding the "best line" is easy (first PCA component)
  - We use power iteration
- Use the two farthest projections as initial endpoints

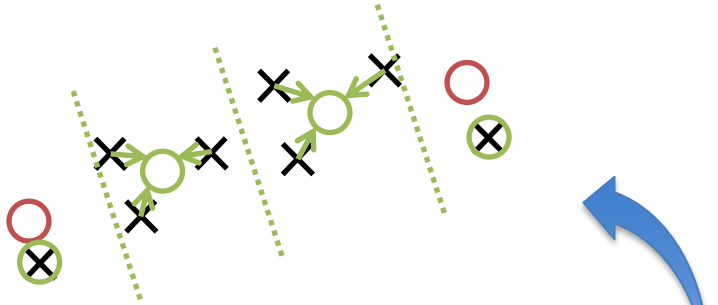
2D Toy example



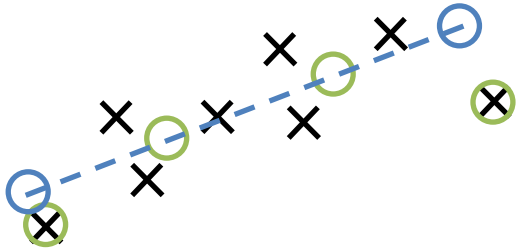
PCA



Initial Endpoints



Scalar Quantization



Least squares

Fast partition pruning

- PCA is a lower bound for endpoint/weight search
  - Could use it to skip partitions, but the pruning is too shallow
  - However, it's a decent proxy for a partition's optimum
- We rank partitions by bound, and only try the best
  - Aggressive pruning has minimal impact on quality
- Small trick: The sum of covariance matrices is constant

BC6H specifics

- Most of the modes vary endpoint bit allocation
  - We can take a guess based on local dynamic range
  - Use the narrowest mode that works (first-fit)
- Do this for both 1 partition and 2 partitions modes
- Only UF16 implemented so far
  - Is there a big usage case for signed HDR values (SF16) ?

Encoder parameters

- The encoder is fairly configurable
  - Iterative refinement iterations
  - Fast pruning thresholds
  - Alpha channel can be ignored
  - Many of the shortcuts can be turned off
  - Some modes can be completely skipped

- A range of presets is available (profiles)

Effective SIMD acceleration with ISPC

Intel SPMD Program Compiler

- Effective way to use vector ISAs of modern processors
  - Scales from SSE2 to AVX2
  - Not Intel-specific!
  - Supports many platforms, including Jaguar
- Code is easy to maintain, quite portable

SIMD-friendly implementation

- Each program instance is mapped to a 4x4 block
  - Typically 4 or 8 blocks processed in parallel
- It's inefficient to run different code paths per block
  - Gather/scatter is more efficient, but still avoid if possible
- The encoder was designed under those constraints
  - There is no scalar code path

SIMD-friendly search

- Fast partition pruning requires partial sorting
  - We use selection sort, 1 scatter per sorted element
- Blocks will typically have divergent partitions
  - PCA initialization and endpoint optimization (LS)
    - For each partition, process all pixels but mask out inactive ones
  - Weight optimization (scalar quantization)
    - For each pixel, use gather to get the right endpoints

SIMD-friendly encoding

- Each mode requires very different bit-packing
  - By using exhaustive search we avoid divergent paths
  - Encode every time, but only keep the best 128-bit result
- But implicit bits positions depend on the partition
  - This makes the bit positions variable
  - Instead we use a fixed encoding in 130 bits
  - The difference is resolved using 160-bit shifts (one or two)

Future Work

What now?

- New supported formats in the next release:
  - ETC1, ETC2
  - ASTC LDR
- Don’t hesitate to ask if you‘re looking for a specific feature: We’ll implement it if there’s enough demand.
- Try it out for yourself:
  - Google: “fast ISPC texture compressor”
  - https://software.intel.com/en-us/articles/fast-ispc-texture-compressor-update

Questions
(and maybe answers)