GPU-friendly Data Compression
Motivation

• Compression is generally perceived as positive, to
  – Reduce the memory footprint of the data
  – Reduce bandwidth requirements
  – Increase cache use
  – ...

• It can also be thoroughly necessary!
  – Data infeasibly large
  – Background storage „not actually that fast“
  – ...

King Abdullah University of Science and Technology
Motivation

• There are tons of methods out there, why re-invent the wheel?
• Some methods are „seriously serial“
  – Huffman Code, LZ* family, arithmetic coding
• Some methods do not have easy random-access
  – Wavelets, DCT,...
• Some methods do not go well with HW interpolation
  – Vector quantization
• Some methods do not offer enough rate-distortion trade-off
  – DXT, PVRTC,...
Motivation

• Do not re-invent, but re-combine methods to fit your needs!

• Compression schemes can be classified as
  – Lossy / Lossless
  – Symmetric / Asymmetric
  – Serial / Parallel

• In this talk we’re concerned mostly with
  – Assymetric – Encoding may take a lot longer than decoding
  – Parallel – The decoder is data-parallel and offers random access
Contemporary Data Compression Pipelines

Input Signal → Decorrelation or Prediction → „Backend“ (Entropy Coder) → Compressed Output
S3TC, a.k.a. DXT

- Find linear component per 4x4 block, 2 R5G6B5 colors
- Per pixel, select one of 4 linear interpolations, 32Bits
- Pros:
  - Fully hardware accelerated, including filtering
- Cons:
  - Fixed bitrate (4bpp, DXT1 compression 6:1 for RGB)
  - Artifacts in non-linear regions of the image
  - Somewhat expensive to encode, use CUDA implementation!
YCoCg DXT

• 2 neat contributions to DXT standard
  – Fast encoding on both CPU & GPU (source available)
  – Better quality than DXT5 by using 4bpp alpha-channel for color

• Gist:
  – Transform RGB to YCoCg color space
  – Store Y (luma) in alpha
  – Store CoCg (chroma) in RG channels
  – Store a per-block constant scaling factor in B channel
  – DXT5 on resulting RGBA image, decoding in shader

van Waveren, Castano, „Real-Time YCoCg-DXT Compression“, NVIDIA Whitepaper, 2007
• Hardware-accelerated on PVR chips (all iPads & iPhones)

• Pros:
  – Generally better than DXT

• Cons:
  – Emulation on Desktop
  – Very slow encoding

Parallel Huffman & Transform Coding
• Compresses scalar & vector data at very high fidelity
• Uses on-the-fly GPU encoding, decompression & rendering

Treib et al, „Turbulence Visualization at the Terascale on Desktop PCs“, IEEE Vis 2012
Basic outline for color images

• Color space transform
  – RGB to YCoCg, no chroma subsampling

• Discrete Wavelet Transform
  – CUDA-based CDF 9/7 lifting wavelet transform
  – Can also use lossless (integer) DWT

• Quantization & Push-Pull
  – Top-down quantization of floating point detail coefficients to integers
  – Push-Pull error compensation to avoid error accumulation
After the error compensated quantization step
- Traverse image in scan-line
- Use a run-length encoder on the resulting stream (prefix scans)
- Finally, use a parallel Huffman coder on the RLE stream

EBCOT is generally JPEG2000‘s most expensive module

Replacing EBCOT results in a great performance gain, at the cost of no longer having a generic image format
Results on GeForce GTX 580

- Throughput encoding: >270Mp/s
- Throughput decoding: >730Mp/s decoding
- Comparison: libjpeg-turbo: up to 95M pixels/sec
- Comparison: Kakadu JPEG2000: up to 35M pixels/sec

Implementational Details & Code
Terrain Rendering
Efficient Out-of-Core Terrain Rendering

Orthophoto

Digital Elevation Map

GPU
Efficient Out-of-Core Terrain Rendering

Usually TBs of data, high geometric fidelity $\rightarrow$ Google Earth inadequate
Methodology – multi-resolution, tile-based representation, Tiles arranged in a quad-tree
Efficient Out-of-Core Terrain Rendering

- Tiles are pre-meshed using restricted quad-tree triangulation
- Geometric error increases by 2 between levels
- Tiles selected and rendered depending on projective error
- View frustum culling to avoid rendering of “invisible” parts
• So far, more or less standard terrain rendering engine
• Novel: Geometry Compression of restricted quadtree meshes
  – Find Hamiltonian path on dual of mesh
    • Easy, essentially term-replacement system
  – Classify triangles
    – Encode class (A,B,C) in 2 bits per triangle
    – Winding (L,R) can be inferred
      • Proof: Analysis of first, follow sets of the grammar
• So far, topology has been encoded.
• For the geometry (height values)
  – Split admitted error for this level into a meshing and a quantization part
  – Use a scalar quantizer with appropriate bins for height values
• Ensures total maximum error is below a given threshold
• Renderer selects error that projects to less than 1 pixel

• Texture compression: standard GPU-accelerated format (S3TC)
Term replacement system used during meshing (recursive splits)
A meshing example
Visual repairing of T-vertices across tile boundaries

Cracks are visually closed using skirts with a closed form solution for minimum height necessary.
• Rendering of ~3TB+ raw data requires data management:
  – Concurrent loading thread to stream data (CPU)
  – Prioritized memory management using FIFO queues (CPU)
  – Circular pre-fetching region around camera position
  – Data is uploaded to GPU in compressed form and rendered
    • Uses either geometry shader (single pass) or pixel shader (multi-pass)
  – Some tricks employed to fully utilize vertex caches on GPU
    • “NULL” vertex buffer, Index buffer stores actual compressed vertex
    • Minimizes data transfers, allows caching of transformed vertices
  – More details available in offline discussion.
Efficient Out-of-Core Terrain Rendering

Performance
- Limited by triangle throughput
- Achieves ~20fps on Intel 4000
- Recent NVIDIA GPUs 200+ fps
- Degrades “graceful”
  - Coarse levels fetched first
- Due to compression, can stream across “narrow” bandwidths
  - e.g., 30MB/sec USB drives

Vector data overlay (Utah data set)
(Hierarchical) Vector Quantization
Hierarchical Vector Quantization

- One of the first methods to be decoded by a GPU (2003)
- Vector Quantization in a nutshell:
  - Similar to paletted images, but may use pixel-blocks
  - Have a set of indices per pixel & a codebook (LUT) of color-blocks
• Initial Codebook: Principal Component Analysis (PCA)
• Refinement: Linde-Buzo-Gray / k-means / GLA etc.
• Implementational Detail: LBG can use partial & fast searches
Vector Quantization – Decoder (GPU)

- Store coded image in Buffer (DX) / TextureBuffer (GL)
  - Advantage over texture: no padding to proper format
  - No 2D address computation
- Compute address of LUT entry (DX, GL, CUDA)
  - Uses GPU bit arithmetic
- Fetch LUT entry (DX, GL, CUDA)

- Bilinear / Trilinear filtering needs to be done “by-hand”

Deferred Filtering, GPU Gems 2, Chapter 41
Pyramid Filters

- Reduce ~ build a mipmap
- Expand: hardware accelerated bilinear fetches
- Use normalized texcoords
- Multiscale predictive coding, essentially.
Hierarchical Vector Quantization – Pyramid Filters

Pyramid

Bilinear

Reconstruction from 8x8 pixels

Depth of Field
Example – confocal microscopy data

- Tile into 512x512xN stacks, N~ 8..16
- N slices in the stack are very similar
- Encode front and back image jointly
  - Predictive Coding, pyramidal filters & VQ
- Linearly interpolate intermediate slices
- Add detail from another VQ stage
- Up to 20:1 compression ratio
- Good fidelity
Decoding at a glance

- Fetch Level 0: 2x2 pixels for front and back slice
- Interpolate to desired slice
- Expand to Level 1: 32x32, fetch detail.
  - Can be done on Codebook entries, not on images
- Interpolate detail to desired slice
- Repeat for Levels 2 & 3: 128x128 & 512x512
- Finally, add detail for desired slice
Round-up

- Vector Quantization used here as lossy backend
- 2D hierarchical prediction
- Linear prediction between slices
- Decodes slices into a GPU cache before rendering
  - To ensure correct filtering between VQ blocks
- Decoding speed (GTX 480 using shared memory)
  - 0.58ms per slice
  - Entire stack can recycle data and is significantly faster
  - Close to 50% for expand and VQ decoding each.
Take-Away
Confocal Microscopy
• Staining typically results in almost planar colorspace
• Adjacent slices (focal lengths) are very similar
• Suggests:
  – Decorrelation of color space
  – Assign bits where needed (chroma subsampling)
  – Joint-Encoding of adjacent slices
• Constraint: Must run on Desktop and Apple devices
  – Uses PVRTC, emulated on Desktop in a shader
• **The Tools:**
  – Karhunen Loéve Transform (KLT), PCA of colorspace
  – PowerVR Texture Compression (PVRTC)
  – $L_2$-optimal downsampling of lesser components, „chroma subsampling“

• **KLT:**
  – Compute covariance of colors
  – Compute Eigenbasis $R$ of covariance matrix
  – Forward: $(\alpha \beta \gamma)^T := R(r \ b \ g)^T$
  – Backward: $(r \ b \ g)^T := R^T (\alpha \beta \gamma)^T$

Custom Compression – The idea for a close-to-planar colorspace

KLT group

$\begin{bmatrix}
0.358 & -1.133 & 0.954 \\
1.057 & 0.028 & -0.169 \\
1.131 & 0.333 & -0.144
\end{bmatrix}$

Binary output

PVRTC
Custom Compression – The idea for close-to-planar colorspace

- KLT decorrelates image into 3 components, $\alpha$, $\beta$, $\gamma$.
  - Store $\alpha$ at full resolution, $\beta$ at half resolution, $\gamma$ as constant
  - Exploits peculiarities of color space, akin chroma subsampling

- For $\alpha$, $\beta$, group 3 adjacent focal planes into RGB image
- Encode each RGB image using PVRTC
  - 2 bits per RGB-triple, exploits intra-slice coherence
- Store PVRTC output and KLT matrix as binary output
Custom Compression – The idea for a close-to-planar colorspace

\[
\begin{bmatrix}
0.358 & -1.131 & 0.954 \\
1.057 & 0.026 & -0.169 \\
1.131 & 0.333 & -0.144
\end{bmatrix}
\]

KLT

\[\alpha, \beta\]

group

Binary output
• Decoding Mobile Client
  – Fetch proper $\alpha, \beta$ samples using bilinear texture filtering
  – Select components corresponding to slice
  – $(r \ g \ b)^T = K(\alpha \ \beta \ 1)^T$, where $K$: stored KLT matrix

• Decoding Desktop
  – PVRTC not supported by Desktop GPUs
  – But Desktop GPUs powerful enough to emulate in shader

• Extremely light-weight on mobile GPU
  – 2 bilinear samples, 2 select, 1 matrix-vector multiplication per pixel
• $\beta$ is stored at half resolution...
• Novel $L_2$ optimal downscale filter
  – Reconstruction is known: HW bilinear upscale $\tilde{\mu}^2$
  – Write reconstruction as convolution $\tilde{\mu}^2 \ast \beta$
  – $\tilde{\mu}^2 \ast \beta$ corresponds to multiplication $A\beta$, where $A$ is a circular matrix of dimension $(4N) \times N$, $N = \#\text{pixels in } \beta$
  – Pseudo-inverse $(A^TA)^{-1}A^T$ yields desired $L_2$ optimal filter $\tilde{\mu}_2$ (modulo boundary effects)
Custom compression – the missing link

• What does it look like?
  – \( \downarrow_2 \ast 2[...0,3^{-n},0,...,0,3^{-3},0,3^{-2},0,3^{-1},0,3^{-2},0,3^{-3},0,...,0,3^{-n},0,...] \)
  – Infinite support, but exponential decay

• What does it cost?
  – Average-of-two costs \( \mathcal{O}(N) \), where \( N \) = #pixels in image
  – \( \widetilde{\downarrow}_2 \) still \( \mathcal{O}(N) \), 2x slower in practical implementation (IIR)

• What does it give us?
  – 1.4dB—2.3dB on our data – good & virtually for free!
In an user study, our compression method performed slightly better than JPEG at the same bitrate, but results in significantly better response times (especially on mobile devices)

|       | rms       |         |         |         |         |         |         |         |         |         |         |         |
|-------|-----------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
|       | 0.0502    | 0.0258  | 0.0442  | 0.0322  | 0.0434  | 0.0274  | 0.0494  | 0.0684  | 0.0567  | 0.0508  | 0.0464  | 0.0324  |
|       | ± 0.0096  | ± 0.0078| ± 0.0124| ± 0.0071| ± 0.0095| ± 0.0061| ± 0.0089| ± 0.0096| ± 0.0119| ± 0.0082| ± 0.0091| ± 0.0072|
|       | PSNR      |         |         |         |         |         |         |         |         |         |         |
|       | Corr.     |         |         |         |         |         |         |         |         |         |         |
|       | 0.8647    | 0.9577  | 0.9516  | 0.9818  | 0.9700  | 0.9895  | 0.9881  | 0.9726  | 0.9854  | 0.9880  | 0.9861  | 0.9764  |
|       | ± 0.0289  | ± 0.0107| ± 0.0085| ± 0.0013| ± 0.0029| ± 0.0009| ± 0.0007| ± 0.0023| ± 0.0009| ± 0.0007| ± 0.0010| ± 0.0016|
Wrap Up – Rendering Performance

- Desktop (GeForce GTX 480, PVRTC emulation)
  - Rendering 512x512 tile, w/ CPU-to-GPU transfer: **0.6ms**
  - 1920 x 1080 viewport, 30 tiles visible, full user interation >**55 fps**

- Apple iPad (PVRTC in hardware)
  - Rendering 512x512 tile, w/ CPU-to-GPU transfer: **0.87ms**
  - 1024 x 768 view, 24 tiles visible, full user interaction **47 fps**

- These timings include actual streaming & user interaction!
- Compression: 14.4:1 (4bpp PVRTC) or 28.8:1 (2bpp PVRTC)
The Future

NEXT EXIT
Future Work

• GPU-friendly lossy compression is well established
  – VQ, DCT, Wavelets, block truncation codes, etc.
  – Typically custom-designed for data at hand, sophisticated error control

• Parallel lossless compression is still in its infancy!
  – Parallel Huffman coding, first parallel Golomb-Rice codes

• Convergence towards general image formats
  – E.g. Poster VI06 by Jiri Matela (JPEG2000 coding) in this GTC
  – Until this takes off, combine existing methods in creative ways!
Thank You