Real-Time Buffer Compression

Michael Doggett

Department of Computer Science

Lund University

Assignments are over...
Say welcome to THE PROJECT

- Everyone needs to do this (2 people/proj)
- Every group must write a report, 2-4 pages
  - Use the LaTeX-style on course webpage
  - Or similar style
- Two paths (take only one):
  1. **iPhone application**
  2. **Graphics hardware optimization**
     - bandwidth reducing algorithms in our SW framework
- Formulated as a non-compulsory competition
  - With grand prizes: **honor and glory**
- Should correspond to 3 points (i.e., 2 weeks of work x 2 persons)
- Finish by 2009-12-04, 12:00, or you're welcome to try again when the make-up exam takes place (Jan 2007)
  - Can continue to work on it until competition

iPhone Project

- Come up with your own iPhone application that is either:
  - Cool
  - Beautiful
  - Interesting
  - Useful
  - Or combination of all of the above
- Examples:
  - Game
  - Screensaver (remember though: 2x2 weeks of work)
  - New GUI for mobile platforms (might give you a job at TAT ☺)
  - A useful tool
- The performance is not the most important thing right now
  - But if you can optimize, that's good
- Formulate project, and clear with me (Mike)!
  - A paragraph description of what your app will do
- Who wins the iPhone competition?
  - A jury (with participants from industry, I hope) decides ...

iPhone project

- Get latest RenderChimp framework on Friday
  - Mac version soon after
- Running on the iPhone is optional
  - Can participate in competition using a PC
  - You need to port it to the mac, vc++ vs g++
- iPhone Developer Program
  - Need an intel mac
  - iPod Touch for overnight use
  - Send me an email

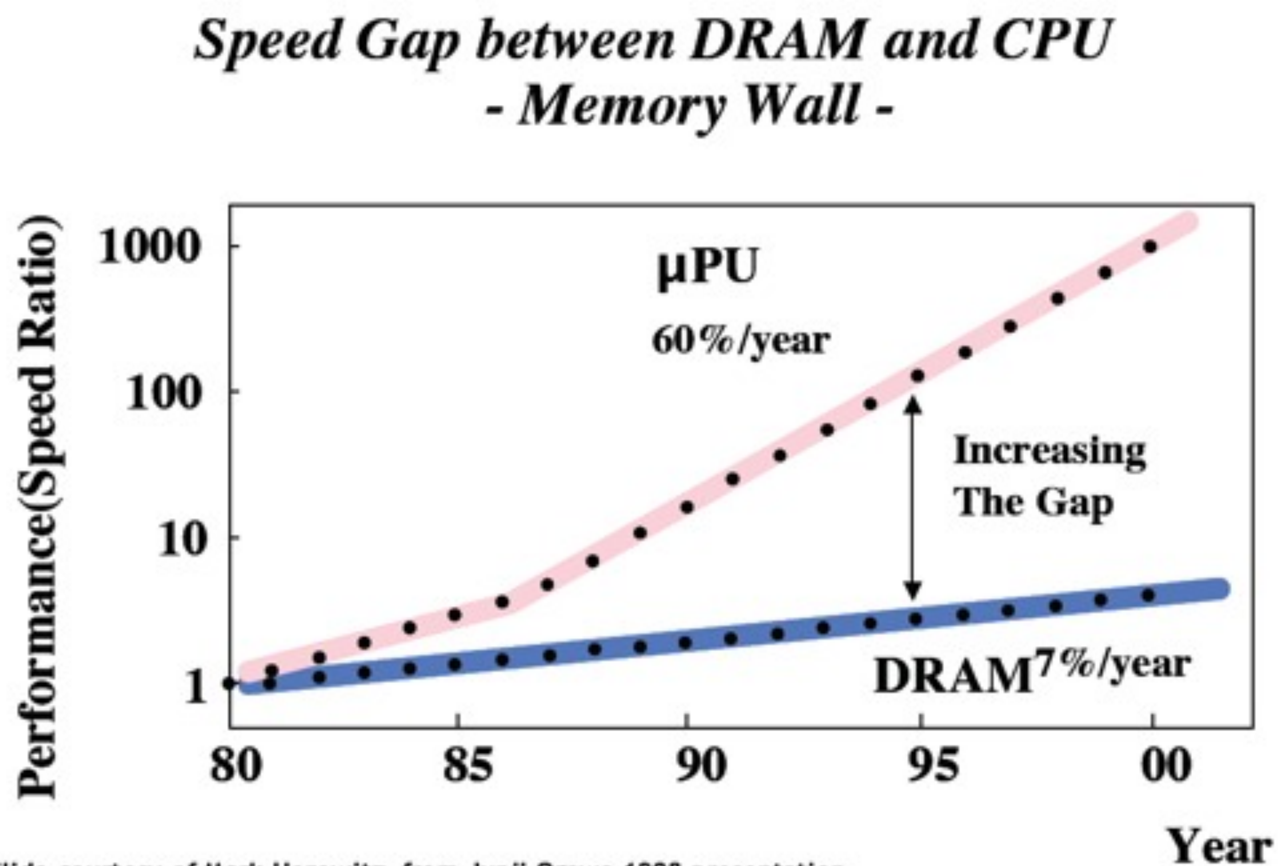
iPhone Particles

Graphics Hardware project

- We provide:
  - A simple animated scene (in the code at the project web page)
  - Basic rasterizer (the framework you've used)
  - **3.0 kB** of onchip memory that you can use for BW-reducing algorithm
  - @ 320x240 (QVGA)
- You give us:
  - A rasterizer that uses as little BW to external memory as possible... (lands you a job at ATI/NVIDIA/INTEL? ☺)
- Who wins?
  - The group that uses least BW to external memory

Back to today's lecture

Memory bandwidth problem



Slide courtesy of Mark Horowitz, from Junji Ogawa 1998 presentation

Mini overview: DRAM

- Dynamic Random Access Memory
  - Looses its content, unless the charge is refreshed periodically, i.e., if power is removed, content is lost
  - Smaller and less expensive than SRAM (which do not need periodic refresh)
  - Thefore, DRAM used as graphics memory in desktop GPUs

  - Many many different types of DRAM:
    - SDRAM, VRAM, SGRAM, PSRAM, etc.
  - There are also double data rate (DDR) versions
    - Works on both low-to-high clock transitions, and high-to-low transitions

Memory architectures in a

- GeForce 6800 DRAM DDR memory:
  - 550 MHz x 256 bits/clock x 2 transfers/clock=~33-35 Gbytes/s
    - Kilgariff and Fernando's article said 35Gb, I get 33Gb

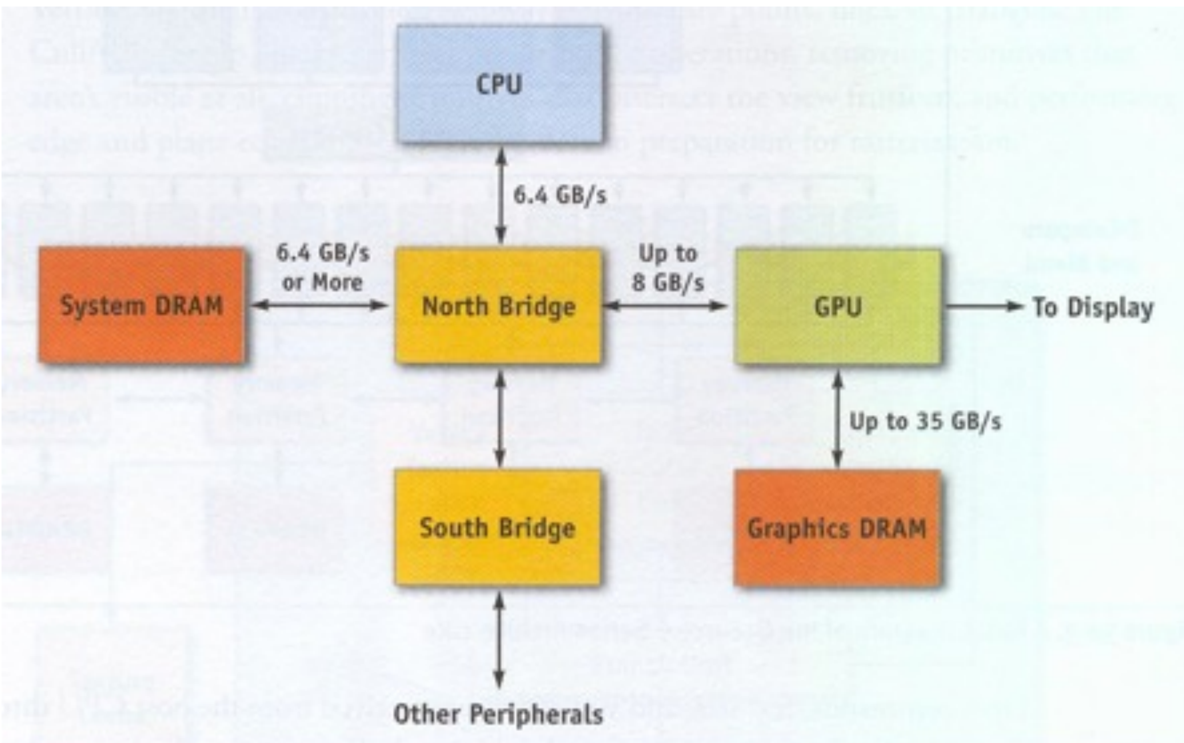
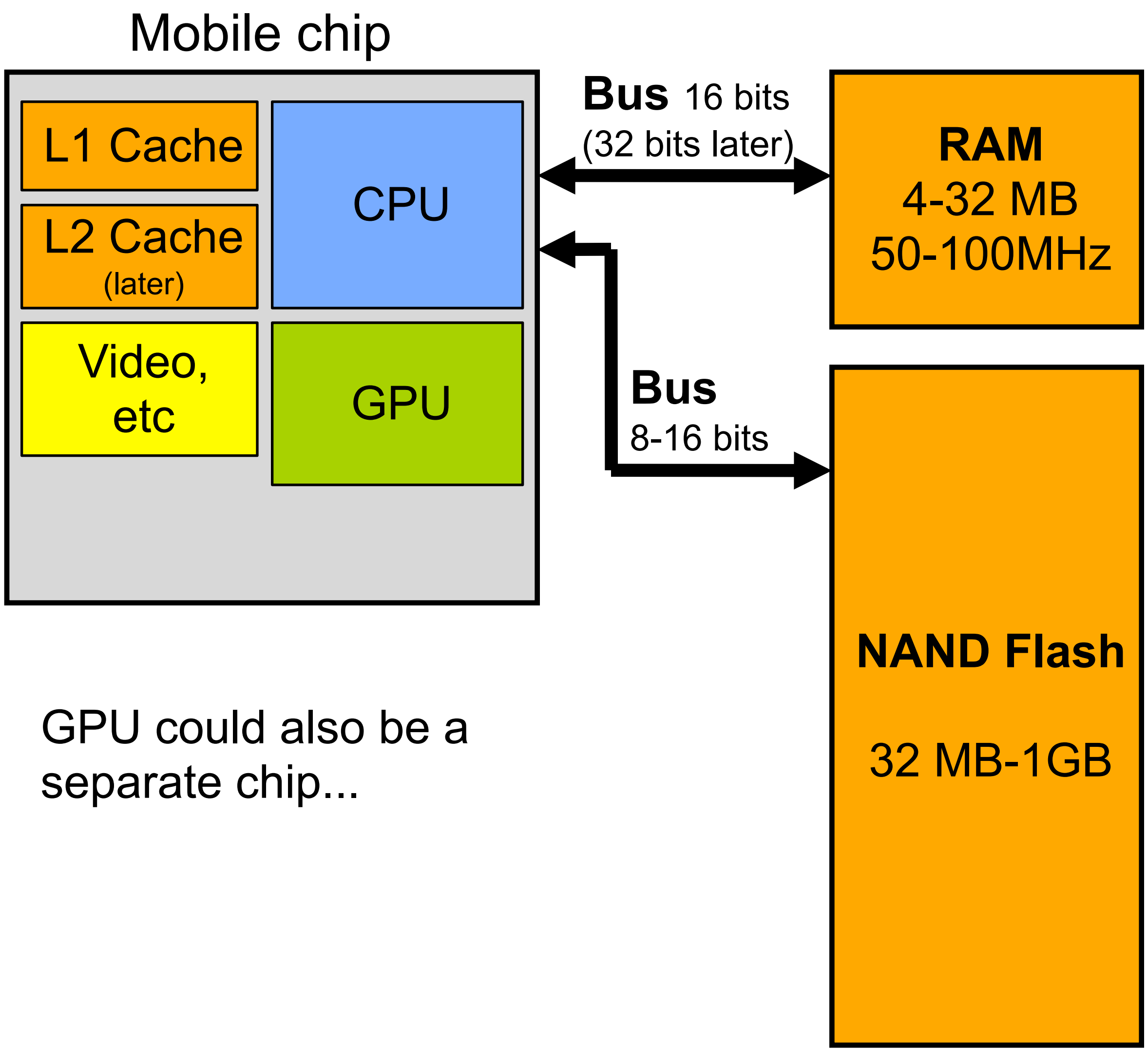


Figure 30-2. The Overall System Architecture of a PC

This is not what we have in mobile systems!

Memory system in mobile



i.e., maybe 200 MB/s (peak performance) Can be SRAM or DRAM.

~10-30 MB/s

GPU could also be a separate chip...

NAND flash is used as a ”hard drive”, i.e., data is move into RAM, and executed from there.

Flash memory

- iPod nano has 2 or 4 GB of NAND-flash

- Is static: no need for power...
- Used in digital cameras, USB memories, MP3 players, mobile phones: i.e., mobiles...
- Two types: NAND and NOR
  - NAND are cheaper to produce, more silicon-effective (i.e., smaller), and consume less power, and lasts longer...

More about flash

- Can read at random places
- Cannot write randomly
  - Must erase a block of memory, and can then update the content of that block
- NOR: much faster read (can ”execute in place” (XIP) without reading into RAM first)

- One NAND part can hold 16 Gbit (2 GB)
  - From Samsung, September 2005
  - This is in the iPod nano
  - Even bigger now (2007)?

Back to
Graphics Hardware algorithms

Why depth buffer?

hardware Thus the only variation of interest here is Newell et al, an order of magnitude less "costly" and the brute-force approach which is already ridiculously expensive.

"A Characterization of Ten Hidden-Surface Algorithms", Ivan Sutherland, Robert Sproul, and Robert Schumacker (ACM Computing Surveys, March 1974)

[Slide courtesy of John Owens]

The ”brute-force apprach” is depth buffering (aka Z-buffering):
It won over sorting-polygons-methods because memory became ridiculously inexpensive...

Depth buffer bandwidth

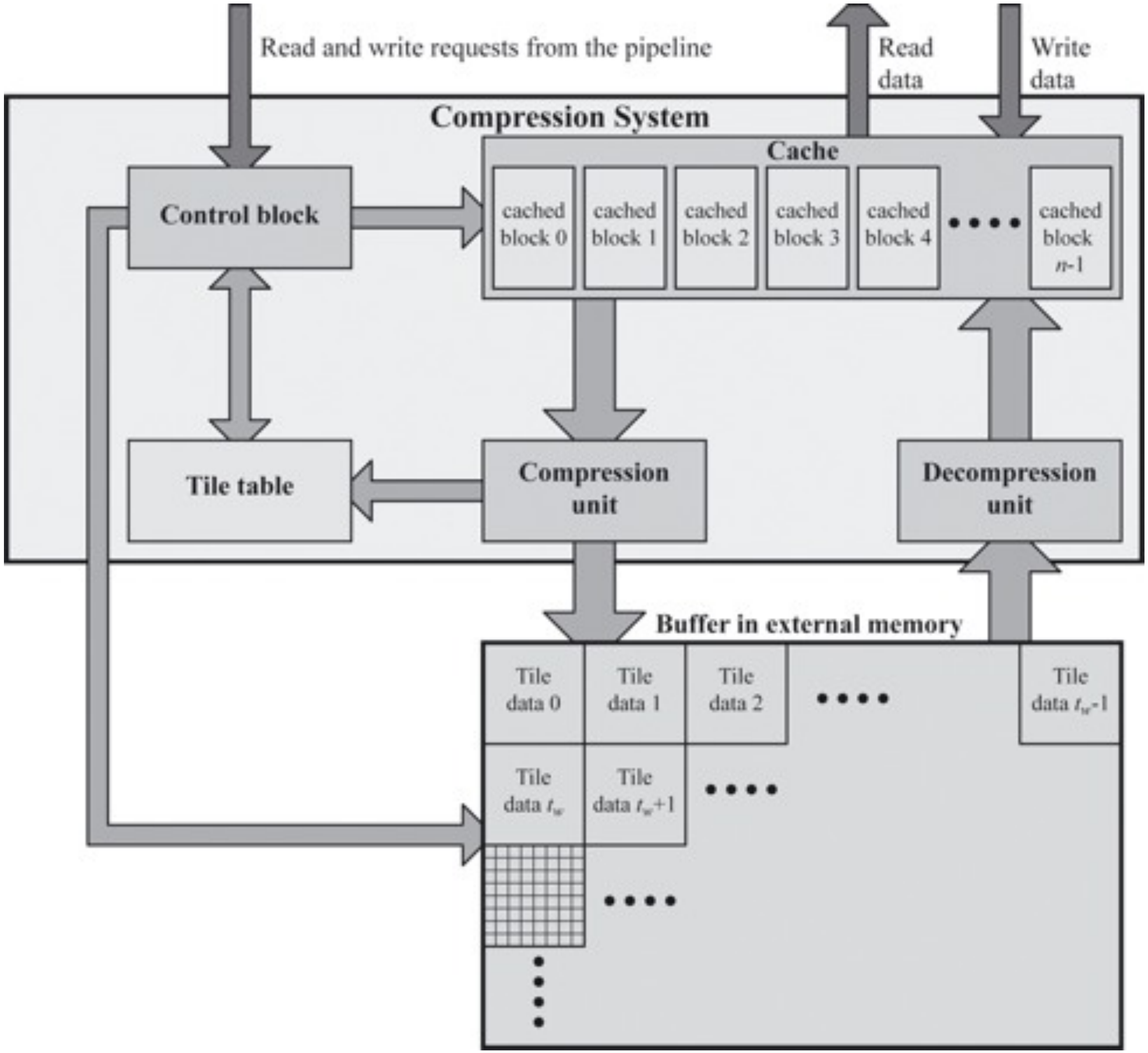
- Still could be quite expensive!
- Zmin/Zmax-culling helped (previous lecture)

- Real-Time Buffer Compression can help reduce
  - Depth buffer bandwidth
  - Color buffer bandwidth
  - Other buffers...

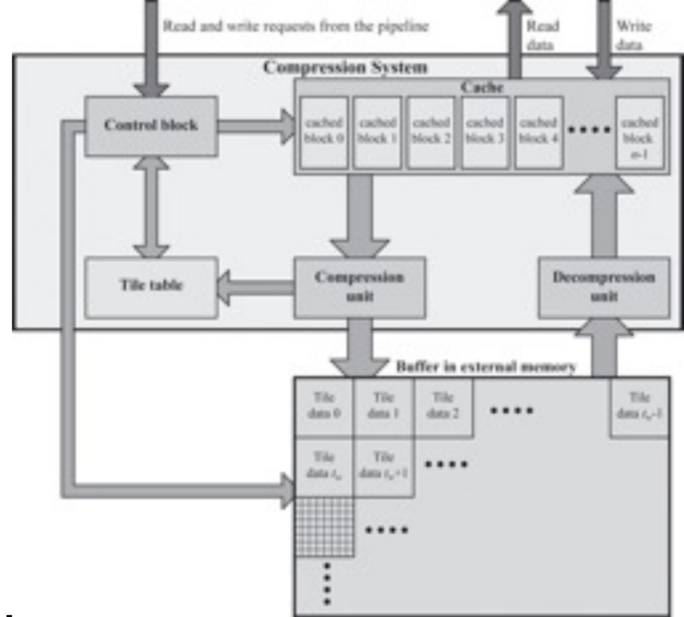
Real-Time Buffer Compression

- Techniques that are or ***may be*** used in mobiles...
- Basic idea:
  - Lots of coherency (correlation) between pixels
  - Use that to compress buffer info
  - Send compressed buffer info over the bus
  - Special hardware handles compression and decompression on-the-fly
  - Must be lossless!!

General Compression System

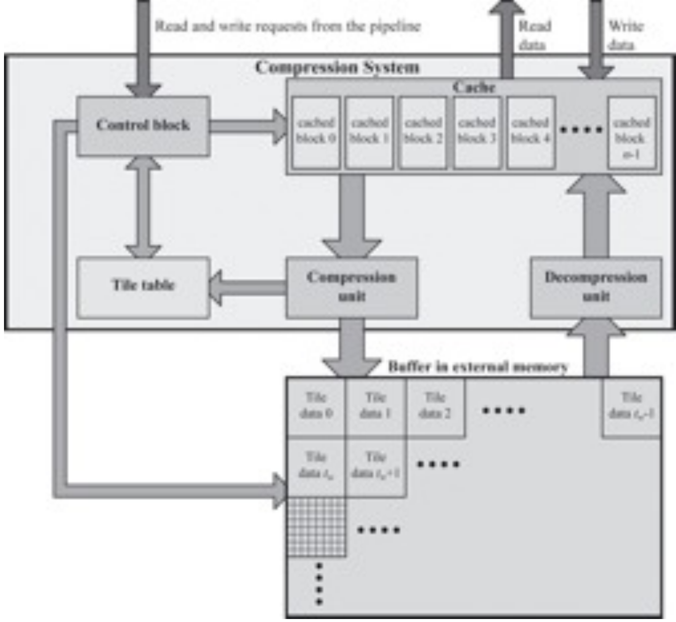


Compression System



- Works on a tile basis
  - Eg 8x8 pixels at a time
- Cache is important!
  - Do not want to decompress tile for every fragment that need access values in that tile
- Tile table store ”per-tile info”:
  - E.g., which compression mode is used
  - Example: 00 is uncompressed, 01 is compressed at 25%, 10 is at 50%, 11 is cleared
  - Always needs one uncompressed mode as a fallback

Example



- Read request → ctrl block
- Checks cache
  - If there, deliver immediately
  - If not
    - Evict one tile from cache by attempting to compress info, and sending resulting representation, update tile table for that tile
    - Check tile table for requested tile, and
    - Read appropriate amount of bytes
    - Decompress (or send cleared info without reading, or in case of data being uncompressed, no decompression needed)
- Done

Dirty bit

- Each tile in cache has one bit for this
- When new info has been written to a tile in cache, set dirty bit=1
- When tile in cache need to be evicted, check dirty bit
  - If =0, information in external memory is up to date → no need to write back!
  - If =1, attempt to compress, and send to external memory
- Saves a lot when no updates
  - Example: particle systems – do not write depth!

Depth buffer compression

- Hard to get accurate information about this
- Looking at patents we can extract some ideas
- Three techniques:
  - Depth offset compression
  - Layered plane equation compression
  - DPCM compression

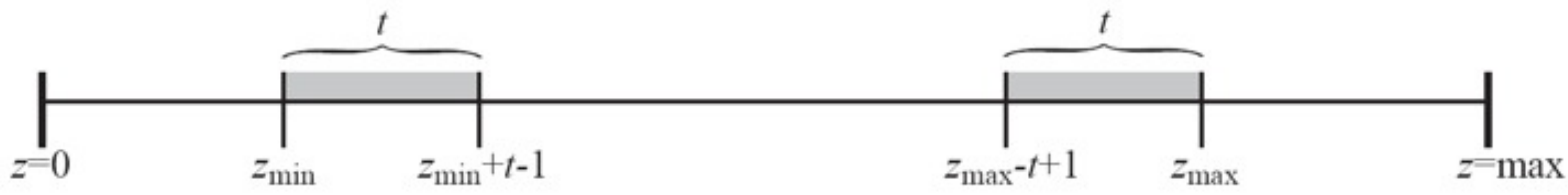
Depth buffer compression

- Simplest buffer to compress
  - Highly coherent info (big triangles wrt tile size)
  - Depth is linear in screen space
- Depth cache and depth compression needed for Zmax-culling
  - Zmin culling is more difficult to combine with depth compression (can avoid depth reads for tiles that are entirely overlapped by a triangle)
- Depths, $d(i,j)$ per tile,
  - i is in $[0, w\text{-}1]$, j is in $[0, h\text{-}1]$
  - Min depth value is $00...00_b$ (all zeroes, eg 24 bits)
  - Max depth value is $11...11_b$ (all ones)
    - i.e., we can use integer math

Depth offset compression (1)

- Identify a set of reference values, r_k,
  - and compress each depth as an offset with respect to one reference value
- Easiest to only use two reference values
  - Use Zmin and Zmax of tile!
  - Rationale: we have two layers
    - One with depths close to Zmin and
    - one with depths close to Zmax

Visually, this means...



- Can encode if all z-values are in the gray regions

Depth offset compression (2)

- Use an offset range of $t=2^p$
- Can use offset, $o(i,j)$, per pixel as:

$$o(i,j) = \begin{cases} d(i,j) - z_{\min}, & \text{if } d(i,j) - z_{\min} < t, \\ z_{\max} - d(i,j), & \text{if } z_{\max} - d(i,j) < t. \end{cases}$$

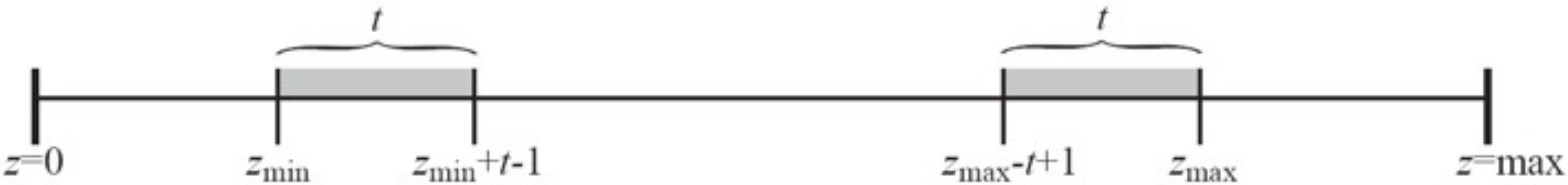
- If at least one pixel, (i,j), cannot fulfil the above, the tile cannot be compressed!
- Info to store (if compression possible):
  - Zmin and Zmax
  - Plus wxh p-bit values

Depth offset compression (3)

- Example with following assumptions:
  - 8x8 pixels per tile
  - $t=2^8$ means 8 bits per offset, o
  - 24 bits depth → Zmin and Zmax has 24 bits each

- Storage (uncompressed: 8x8x3=192 bytes):
  - 1 bit per pixel to indicate whether offset to Zmin or Zmax → 8x8x1 bits= 8 bytes
  - Offsets: 8x8x8 bits= 64 bytes
  - Zmin & Zmax: 6 bytes (might be onchip though)
  - Total: 8+64+6=78 bytes → 100*78/192=41% compression

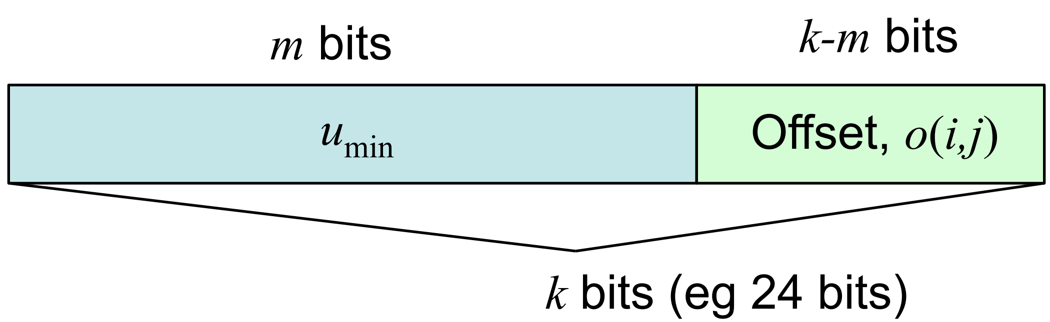
Less expensive implementation



- Only possible to compress if all depths in a tile are gray regions
- There are some extensions to this that makes the hardware simpler!
- Make the offset computation inexpensive!
  - Currently costs an adder in HW

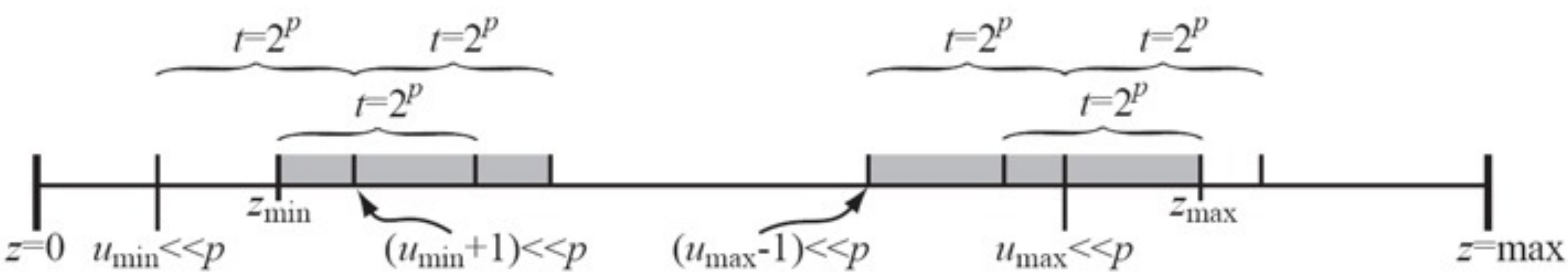
Inexpensive offsets...

- Instead of storing exactly Zmin and Zmax, store only m most significant bits (MSBs)
  - Call these truncated values, $u_{\min}$ and $u_{\max}$
- Offset is now simply the k-m least significant bits of depth (no add needed)



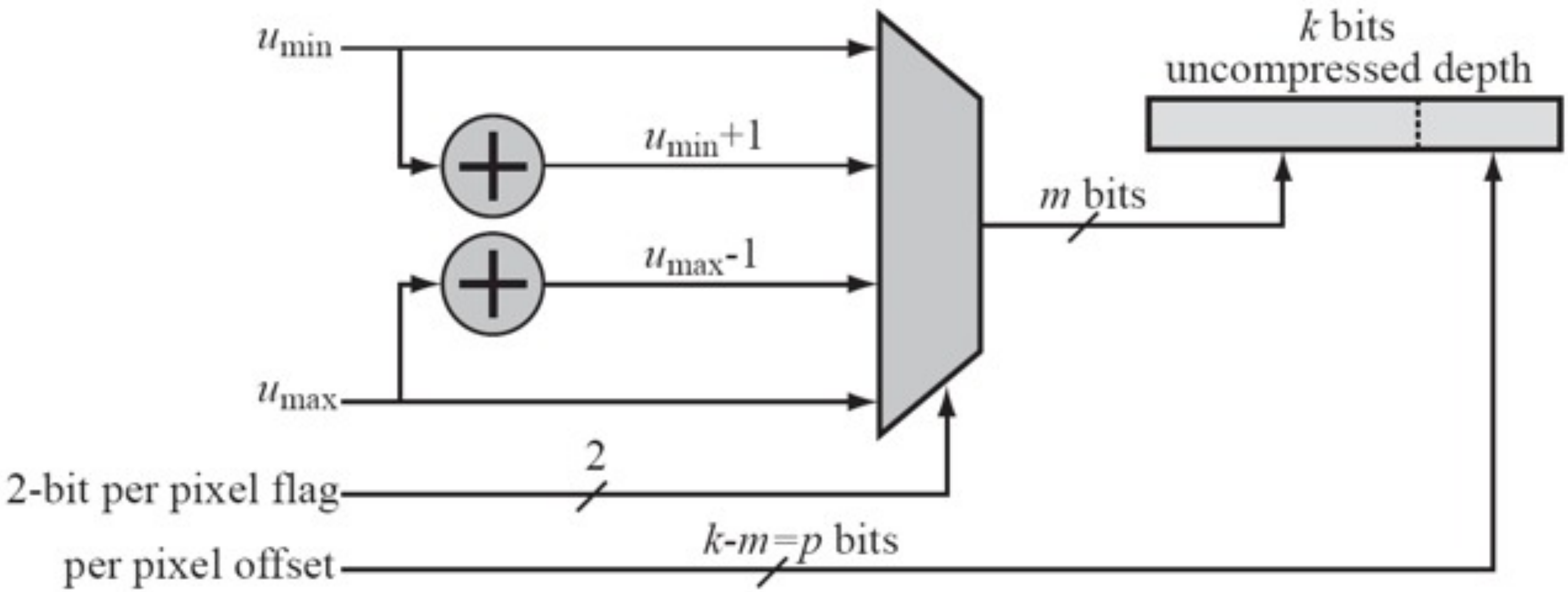
Disadvantage of cheap offsets, and a solution

- Only values in dark gray area can be coded → loss of compressibility!
- Simple solution: use one more bit per pixel → four reference values:
  - u_{min}, $u_{min}+1$, $u_{max}-1$ and u_{max}



Decompression hardware

- Very simple



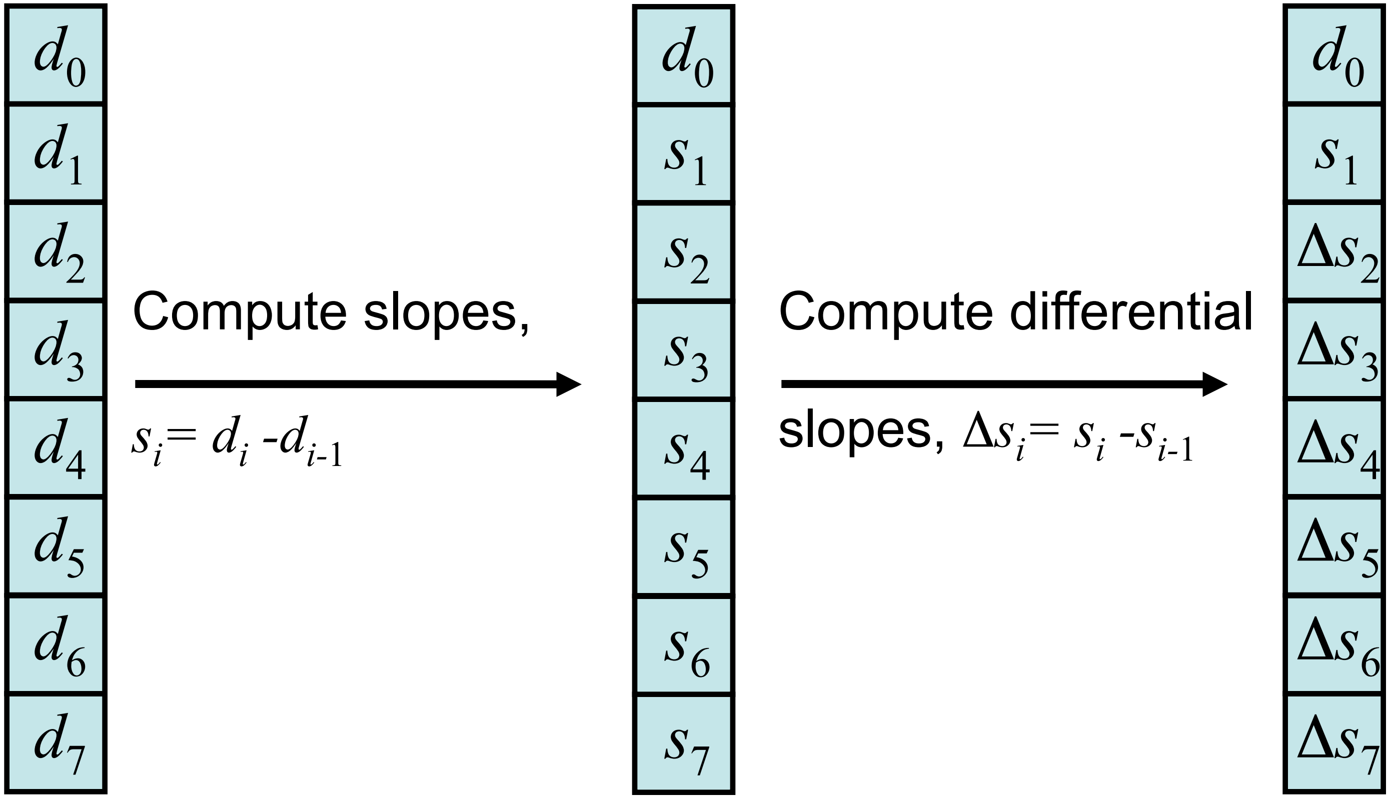
Compression ratio with inexpensive variant

- Slightly worse → 44% instead of 41%
- But, range of offsets is larger!
  - Best case: range is twice as large
  - Worst case: range is only one depth value larger
  - Average case: range is about 50% larger!
  - So more tiles can be compressed, but still costs more

DPCM Compression

- ATI had this in their hardware
  - DPCM=differential pulse code modulation
- Basic idea: we usually have linearly varying values in tile
  - Second derivative of linear function is zero!
  - However, we have discretized function, so need discretized ”second derivatives”

DPCM: Focus on one column of depths



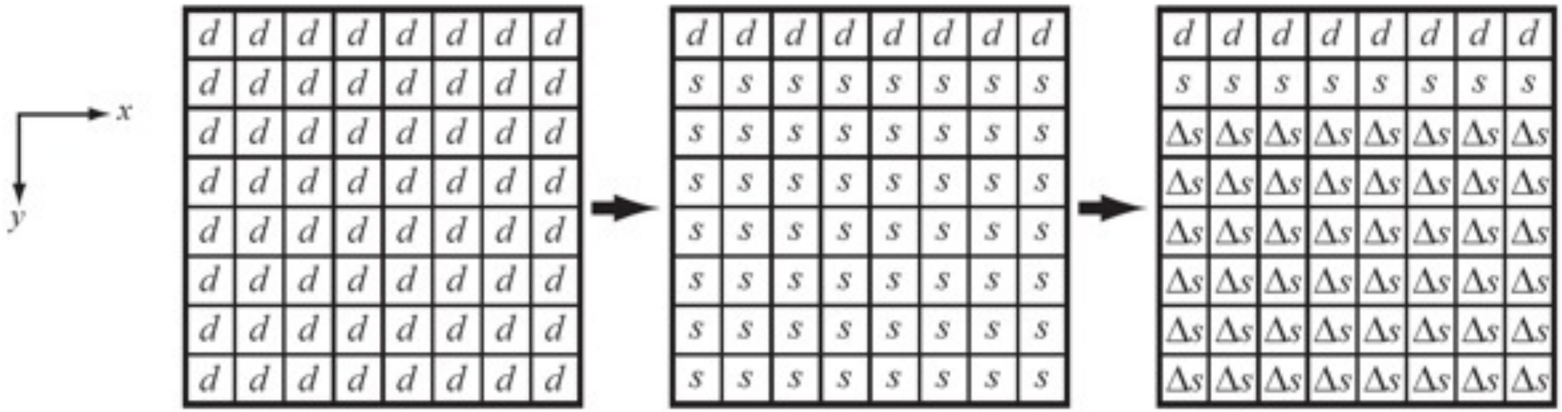
- For linear functions, the Δs's will be close to 0
- Reconstruction is simple (next slide)

DPCM reconstruction

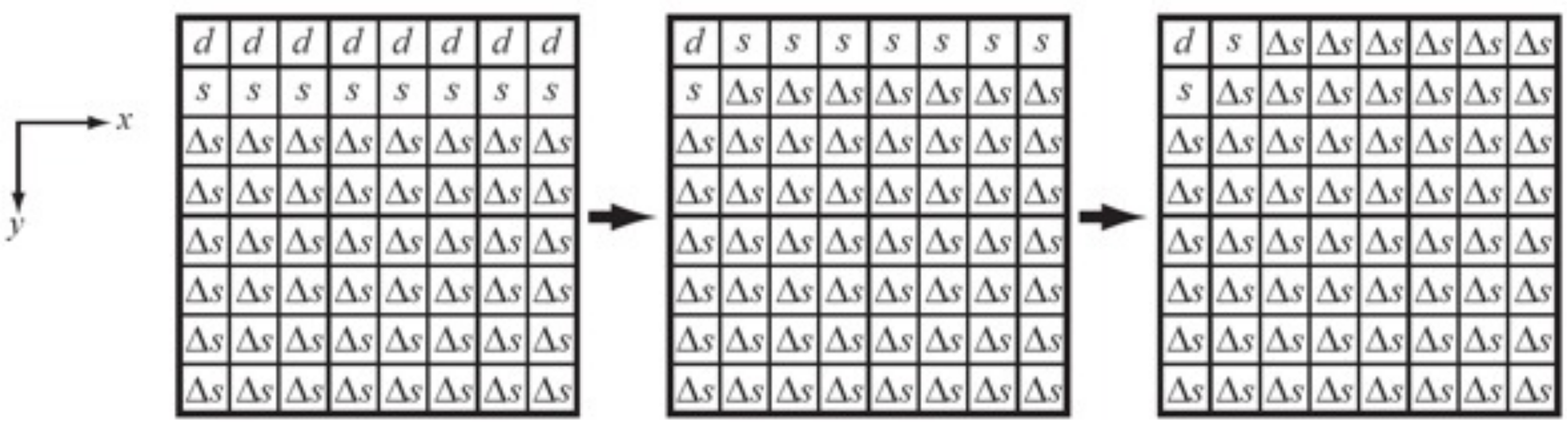
- From definition, we get: $d_i = d_{i-1} + s_i, \quad i \geq 1$
- Only s_1 is known, but: $s_i = s_{i-1} + \Delta s_i, \quad i \geq 2$

$$\begin{array}{ll}
d_0, & \text{already known} \\
d_1 = d_0 + s_1, & s_1 \text{ already known} \\
d_2 = d_1 + s_2, & \text{where } s_2 = s_1 + \Delta s_2 \\
d_3 = d_2 + s_3, & \text{where } s_3 = s_2 + \Delta s_3 \\
\ldots \text{and so on} &
\end{array}$$

Process each column independently



- Not ideal: still many d's and s's
- Process first two rows similarly →



DPCM: How to store this?

- One depth, d, two slopes, s, and 61 Δs
- The Δs are small, in [-1,+1] inside triangle
- Use two extra bits per pixel:
  - 00: add 0
  - 01: add +1
  - 10: add -1
  - 11: use as escape code to handle extraordinary cases...
- Best case compression (no escapes at all):
  - 24 bits + 25 +25 +8x8x2 ~= 25 bytes (13% ratio)
  - If a single triangle covers entire tile
    - Do not need the 11-escape case then though...

DPCM: common case

- Single column:
  - Depths:1,2,3,4,8,10,12,14
  - Slopes: 1,1,1,4,2,2,2
  - Diff slopes: 0,0,3,-2,0,0
- Two escape codes needed per column to change from one plane eq (tri) to another
  - Becomes expensive! 40% compression ratio
- Solution: encode from the top & down and from bottom & up
  - Store also where transition happens
  - Gives about 20% compression ratio!
  - Might be possible to use fewer bits per slope
  - Can only handle two plane equations per tile
  - Still does not use escape (trinary encoding would be good!)

DPCM: more complex cases

- Do the common case
  - Add real escapes as well, and store, say 16 extra escape codes
  - Or change from 2 extra bits per pixel to, perhaps, 8 bits per pixel

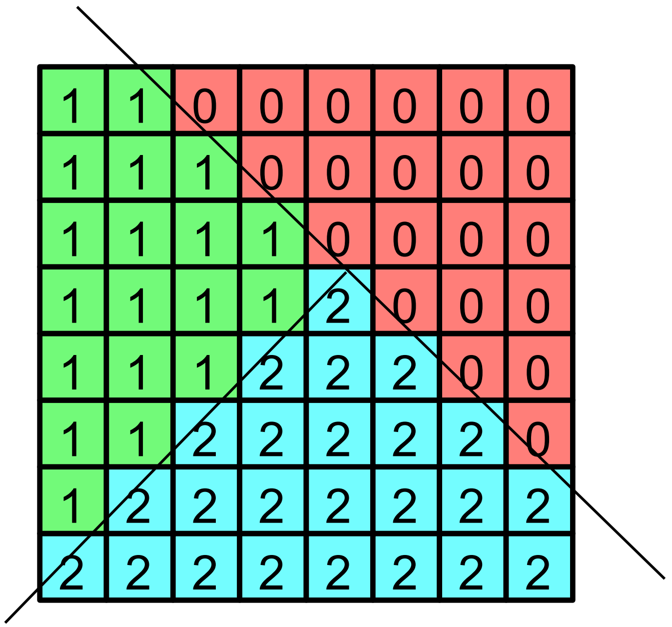
Plane Eq. Compression

- Each triangle is a plane
- For every triangle in a tile store that plane equation
  - Store one depth in center of tile, dz/dx, dz/dy
- For every pixel in the tile store an index to find the matching plane equation
- Works great for multisample!
- [VanHook07] US Patent 7,242,400

Plane Eq. Compression

- 0 : Zc, dz/dx, dz/dy
- 1 : Zc, dz/dx, dz/dy
- 2 : Zc, dz/dx, dz/dy

- Plane Equations
  - 3 x (3 + 2 + 2)B = 21B
- Indexes
  - 64 x 2bits = 16B
- Compressed
  - 37B
- Uncompressed
  - 64 x 3B = 192B
- Compression ratio
  - 19%



Depth Buffer Compression

- A little is known about this topic:
  - To pass the course, read this paper:
    - Jon Hasselgren and Tomas Akenine-Möller, “Efficient Depth Buffer Compression,” *in Graphics Hardware 2006*
- You can invent your own superior algorithms!
- ATI has reported about 50% compression on a wide range of benchmarks

Color Buffer Compression

- Could use offset compression for R, G, and B separately (perhaps)
- Could use JPG's non-lossy algorithms
- Can do simple color compression for multi-sample anti-aliasing
- Can compress clear color
- Is generally very difficult due to restrictions
  - Cannot be lossy
  - Must decode very fast for alpha blending

Conclusion

- Reduces bandwidth further
  - Quite simple for depth
  - Harder for color
  - Needs cache
  - Needs fallback for non-compressed mode

The end