

# OpenGL Performance Tuning

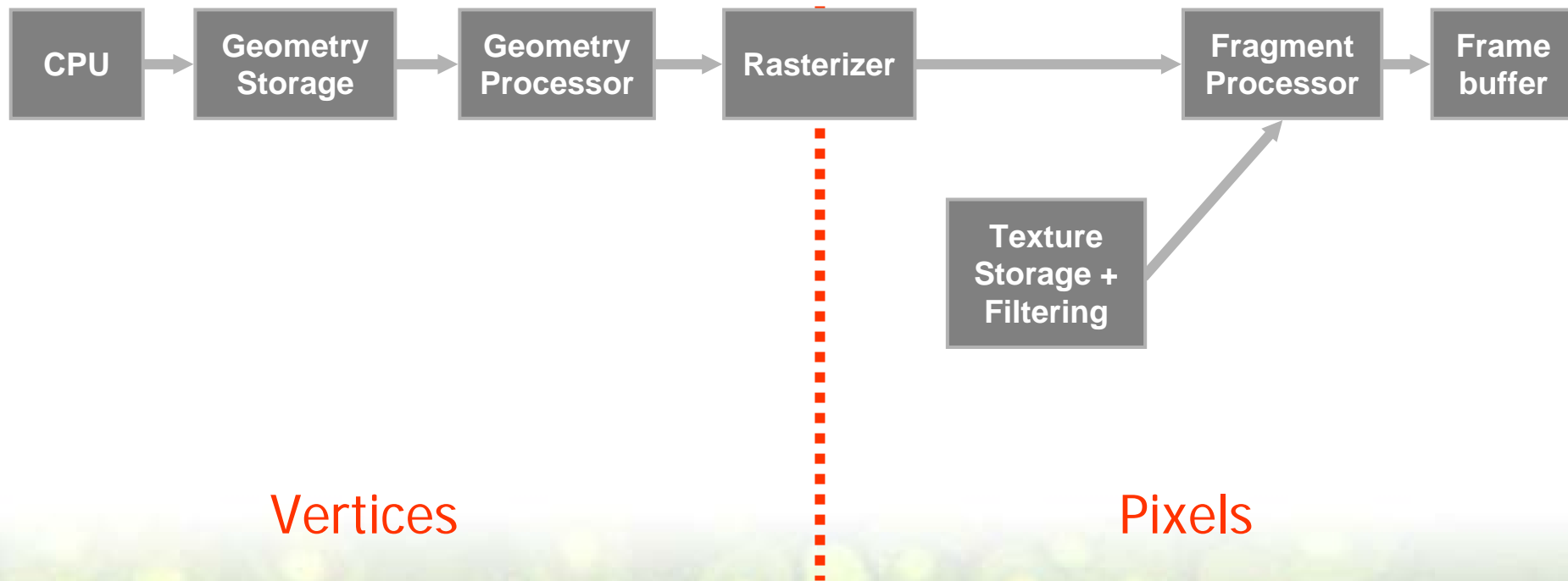
Evan Hart – ATI

Pipeline slides courtesy John Spitzer -  
NVIDIA

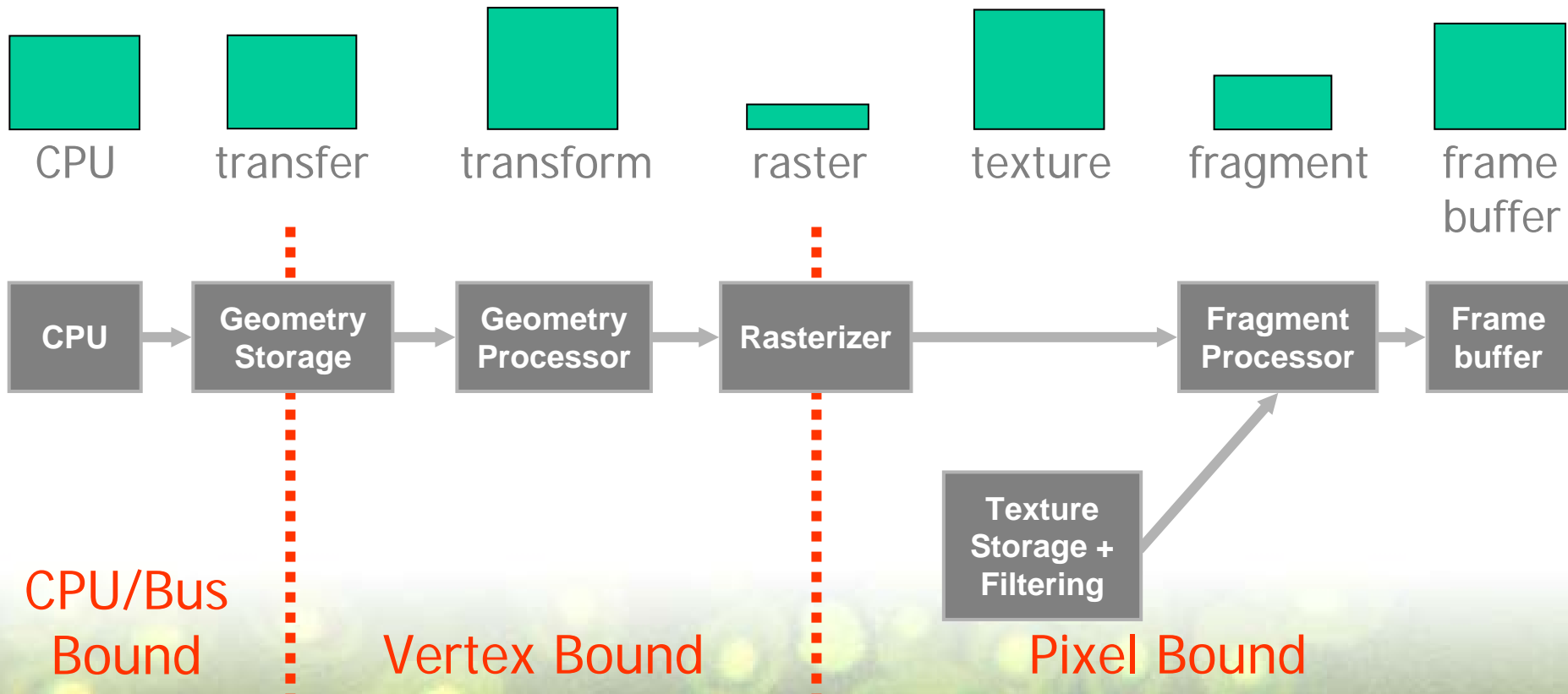
# Overview

- What to look for in tuning
- How it relates to the graphics pipeline
- Modern areas of interest
  - Vertex Buffer Objects
  - Shader performance

# Simplified Graphics Pipeline

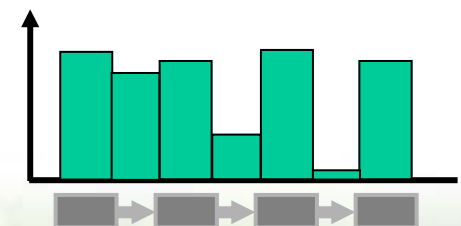
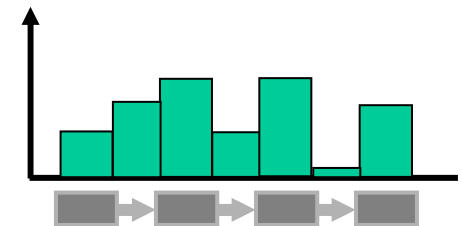
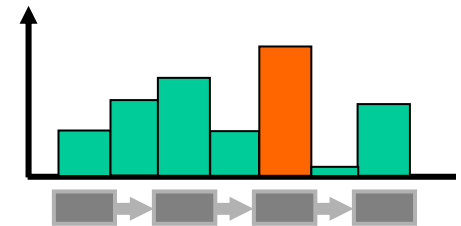


# Possible Pipeline Bottlenecks

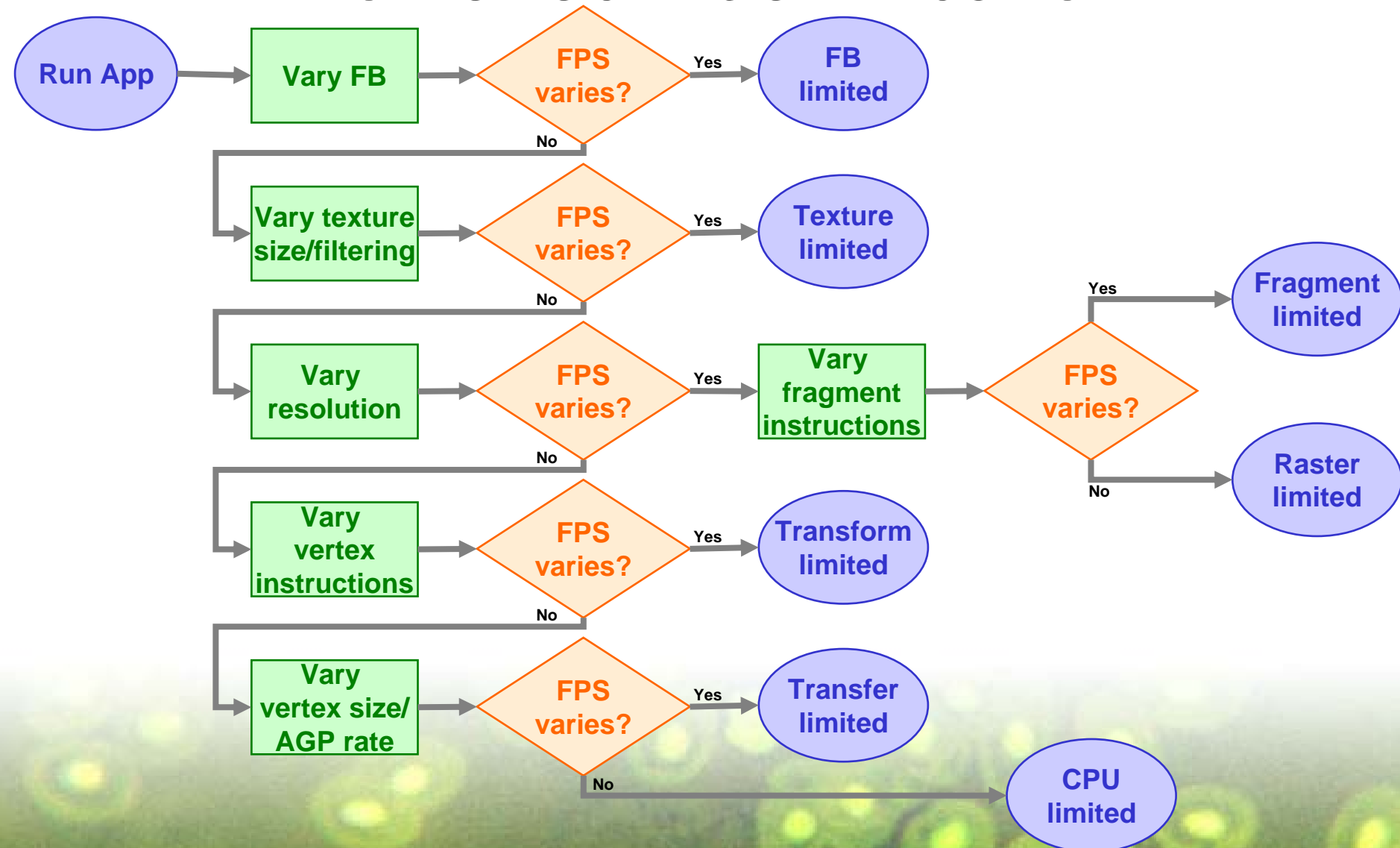


# Battle Plan for Better Performance

- Locate the bottleneck(s)
- Eliminate the bottleneck (if possible)
  - Decrease workload of the bottlenecked stage
- Otherwise, balance the pipeline
  - Increase workload of the non-bottlenecked stages:

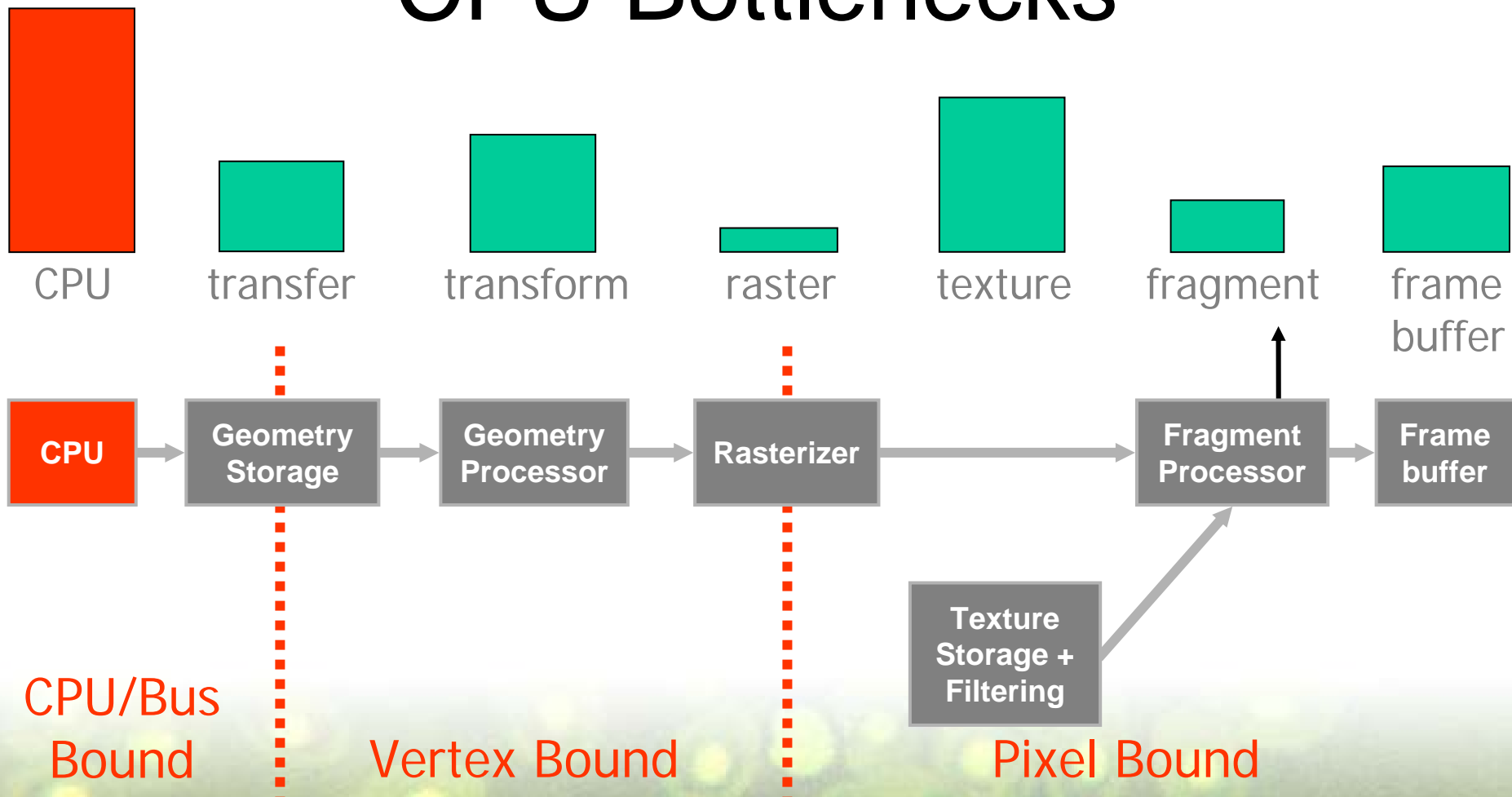


# Bottleneck Identification





## CPU Bottlenecks

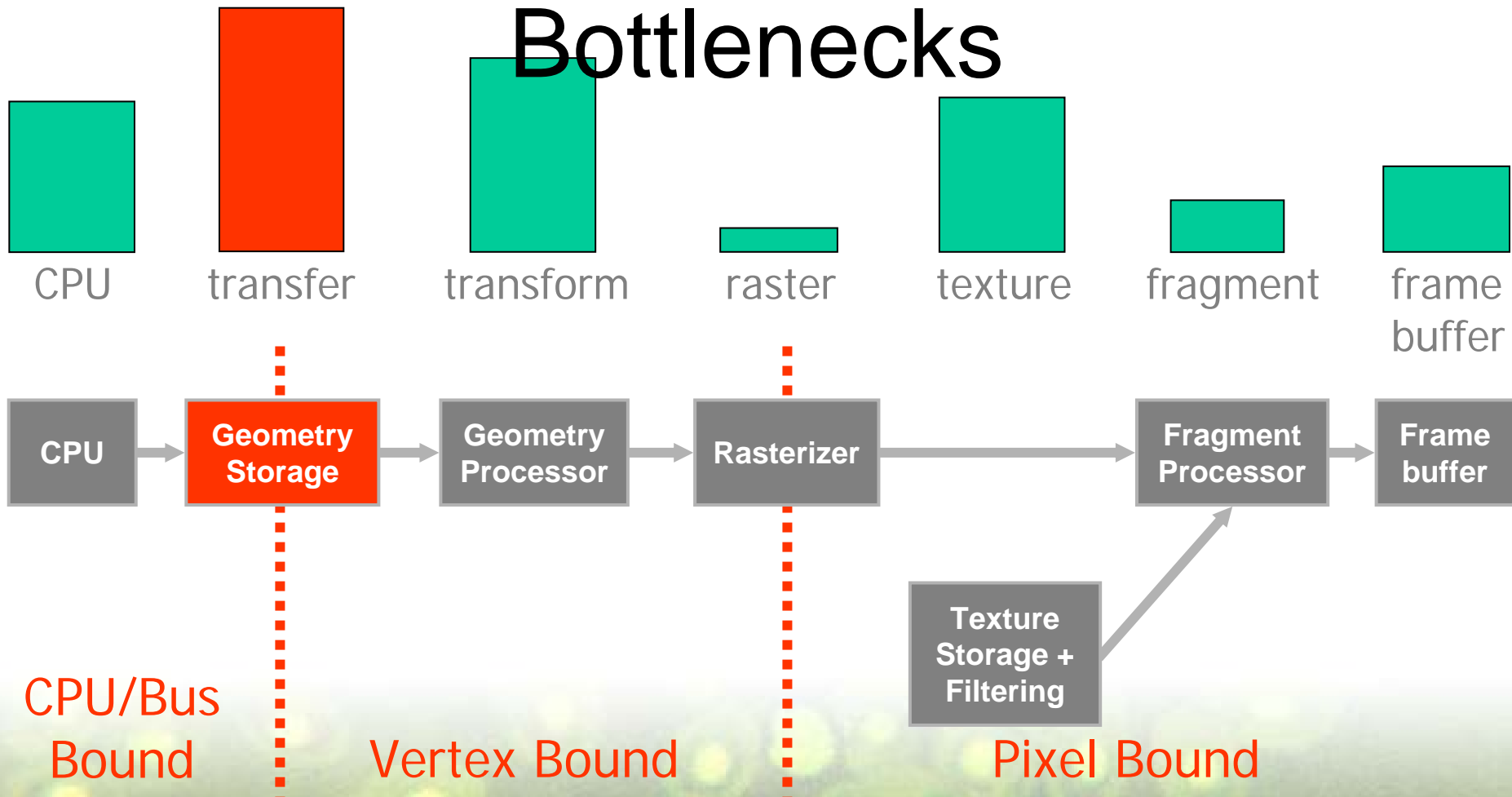


# CPU Bottlenecks

- Application limited (most games are in some way)
- Driver or API limited
  - Too many state changes per draw
  - Non-optimal paths
- Use VTune (Intel performance analyzer)
  - caveat: truly GPU-limited games hard to distinguish from pathological use of API



# Geometry Transfer Bottlenecks



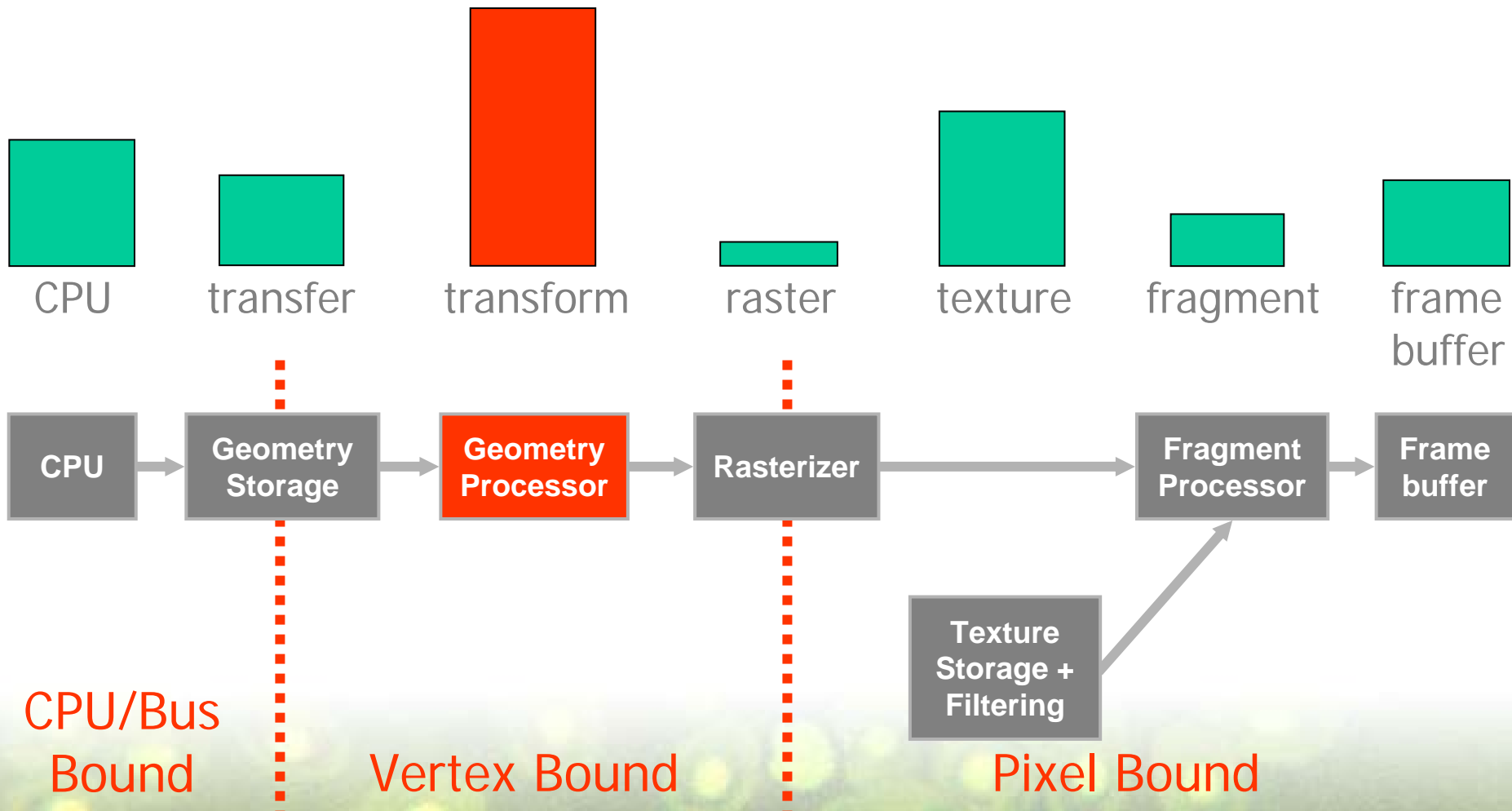
# Geometry Transfer Bottlenecks

- Vertex data problems
  - size issues (just under or over 32 bytes)
  - non-native types (e.g. double, packed byte normals)
- Using the wrong API calls
  - Immediate mode, non-accelerated vertex arrays
  - Non-indexed primitives (e.g. `glDrawArrays`)
  - Prefer `glDrawRangeElements` over `glDrawElements`
- AGP misconfigured or aperture set too small

# Optimizing Geometry Transfer

- Dynamic geometry - use `ARB_vertex_buffer_object`
  - vertex size ideally multiples of 32 bytes
  - access vertices in sequential pattern
  - always use indexed primitives (i.e. `glDrawElements`)
  - 16 bit indices can be faster than 32 bit
  - try to batch at least 100 tris/call
- Static geometry - can use display lists

# Geometry Transform Bottlenecks



# Geometry Transform Bottlenecks

- Too many vertices
- Too much computation per vertex
- Vertex cache inefficiency



# Too Many Vertices

- Maximize vertex cache usage with locality of reference
- Use levels of detail (but beware of CPU overhead)
- Use bump maps to fake geometric details



# Too Much Vertex Computation: Fixed Function

- Avoid superfluous work
  - >3 lights (saturation occurs quickly)
  - local lights/viewer, unless really necessary
  - unused texgen or non-identity texture matrices
- Consider commuting to vertex program if a good shortcut exists
  - example: texture matrix only needs to be 2x2
  - Fixed function already tuned for the HW

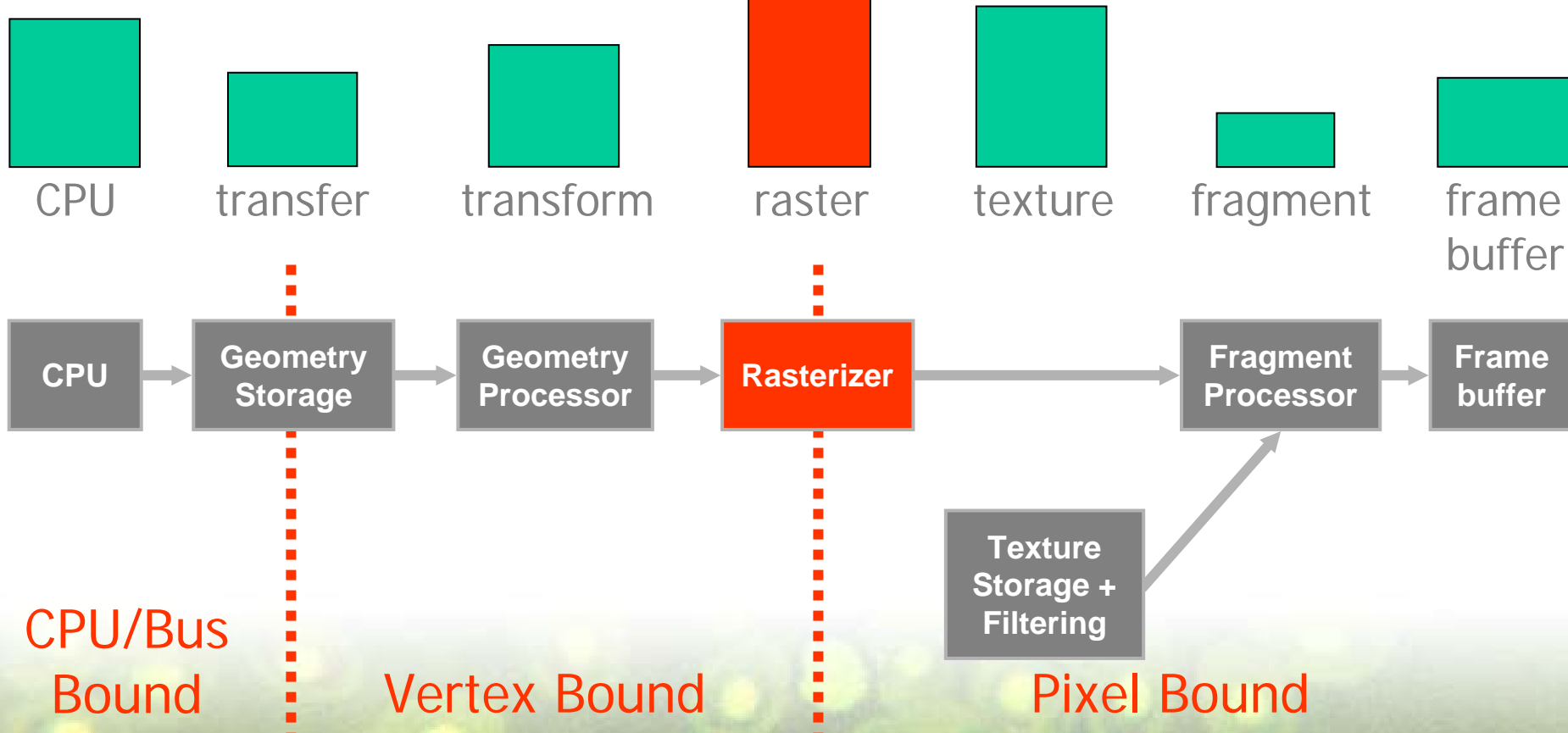
# Too Much Vertex Computation: Vertex Programs

- Move per-object calculations to CPU, save results as constants
- Leverage full spectrum of instruction set (LIT, DST, SIN,...)
- Leverage swizzle and mask operators to minimize MOVs
- Consider using shader levels of detail

# Vertex Cache Inefficiency

- Always use indexed primitives on high-poly models
- Re-order vertices to be **sequential in use** (e.g. NVTriStrip)
- Favor strip order over random order for triangle lists

# Rasterization Bottlenecks



# Rasterization

- Rarely the bottleneck (exception: stencil shadow volumes)
- Speed influenced primarily by size of triangles
- Also, by number of vertex attributes to be interpolated
- Be sure to maximize depth culling efficiency

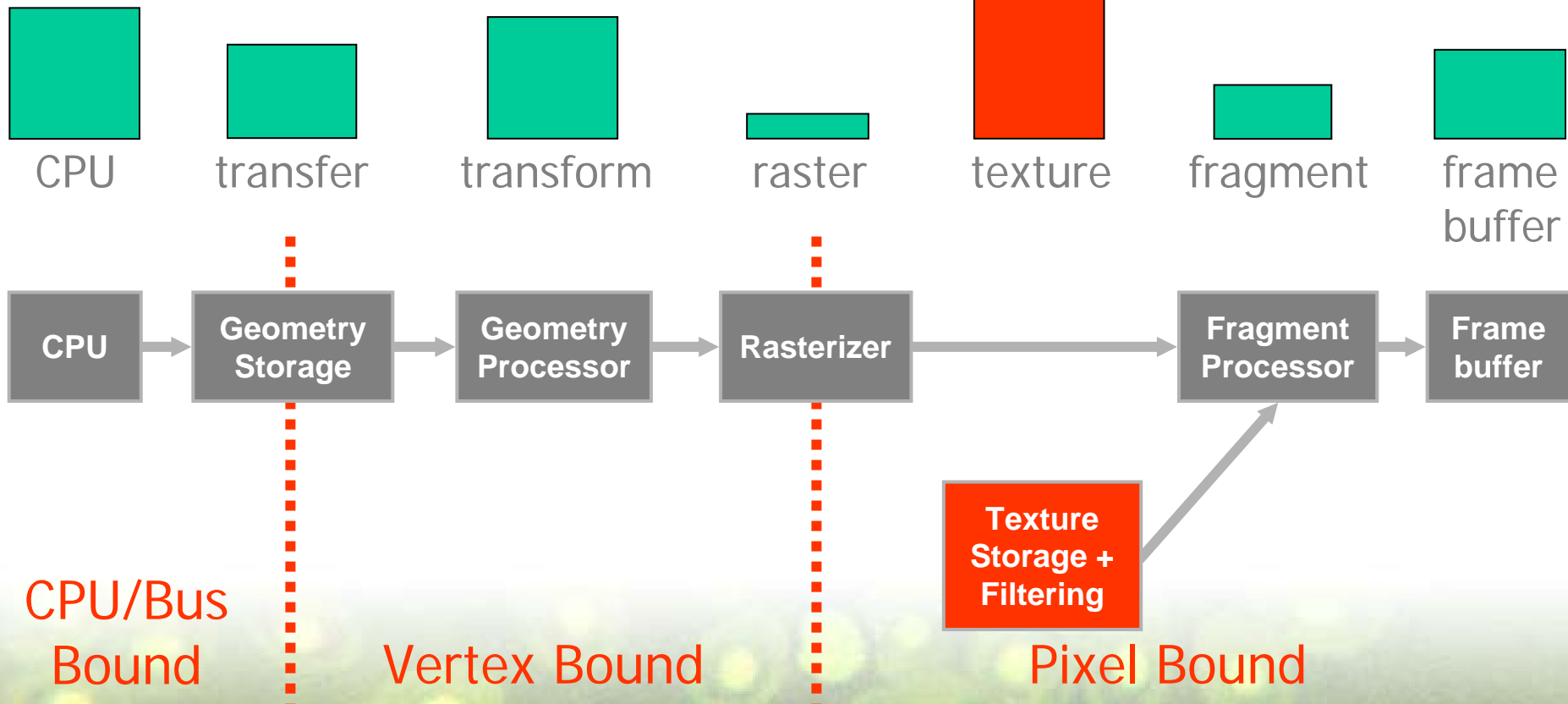


# Maximize Depth Culling Efficiency

- Always clear depth at the beginning of each frame
  - clear with stencil, if stencil buffer exists
  - feel free to combine with color clear, if applicable
- Coarsely sort objects front to back
- Don't switch the direction of the depth test mid-frame
- Constrain near and far planes to geometry visible in frame
- Avoid polygon offset unless you really need it
- NVIDIA advice
  - use scissor and depth bounds test to minimize superfluous fragment generation for stencil shadow volumes
- ATI advice
  - avoid EQUAL and NOTEQUAL depth tests



# Texture Bottlenecks



# Texture Bottlenecks

- Running out of texture memory
- Poor texture cache utilization
- Excessive texture filtering

# Conserving Texture Memory

- Texture resolutions should be only as big as needed
- Avoid expensive internal formats
  - Modern GPUs allow floating point formats
- Compress textures:
  - Collapse monochrome channels into alpha
  - Use 16-bit color depth when possible (environment maps and shadow maps)
  - Use DXT compression
    - Be smart and use tools to selectively compress (ATI's Compressorator)

# Poor Texture Cache Utilization

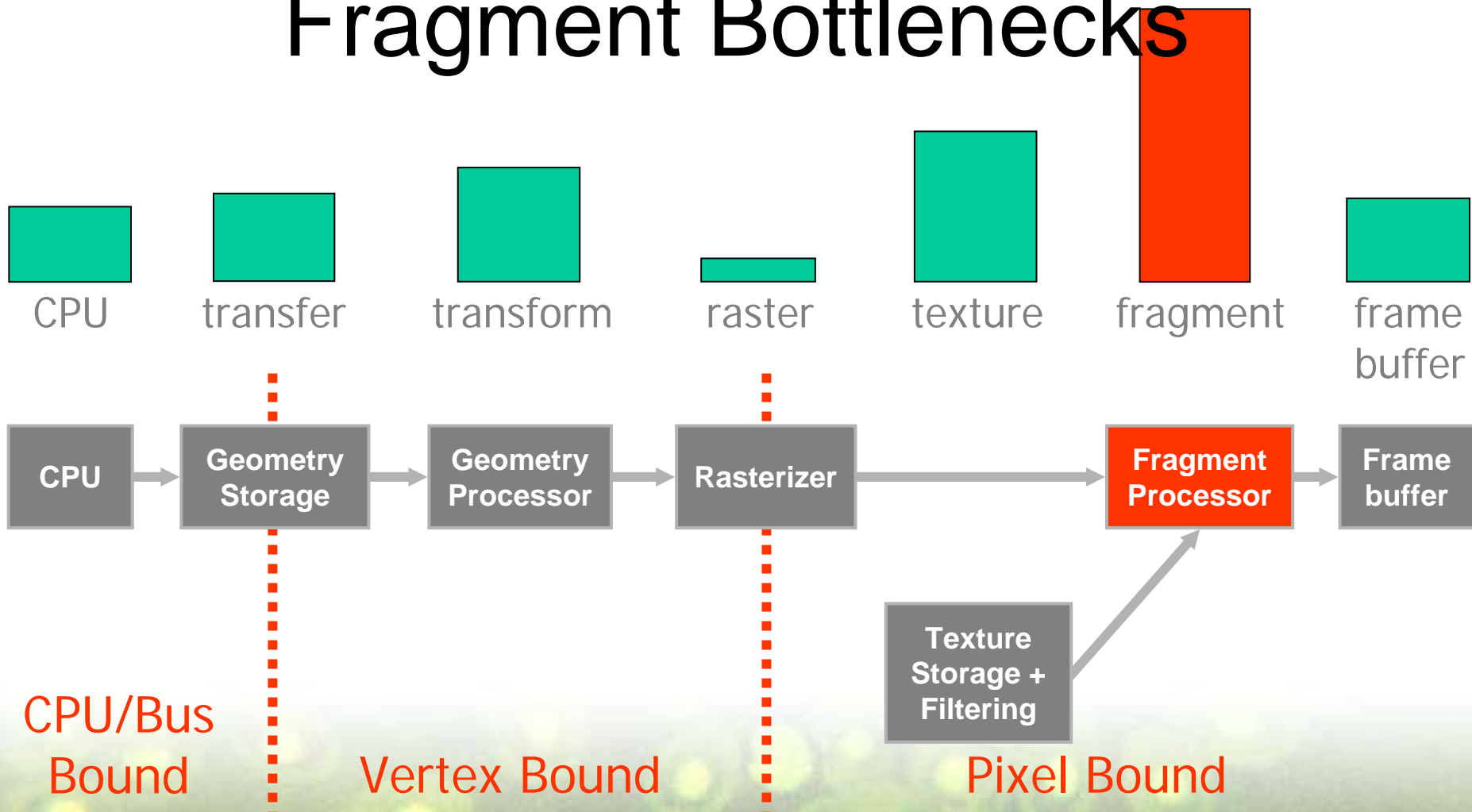
- Localize texture accesses
  - beware of dependent texturing
  - ALWAYS use mipmapping
- Avoid negative LOD bias to sharpen
  - texture caches are tuned for standard LODs
  - sharpening usually causes aliasing in the distance
  - opt for anisotropic filtering over sharpening

# Excessive Texture Filtering

- Use trilinear filtering only when needed
  - trilinear filtering can cut fillrate in half
  - typically, only diffuse maps truly benefit
  - light maps are too low resolution to benefit
  - environment maps are distorted anyway
- Use anisotropic filtering judiciously
  - often more expensive than trilinear
  - not useful for environment maps



## Fragment Bottlenecks





# Fragment Bottlenecks

- Too many fragments
- Too much computation per fragment
- Unnecessary fragment operations

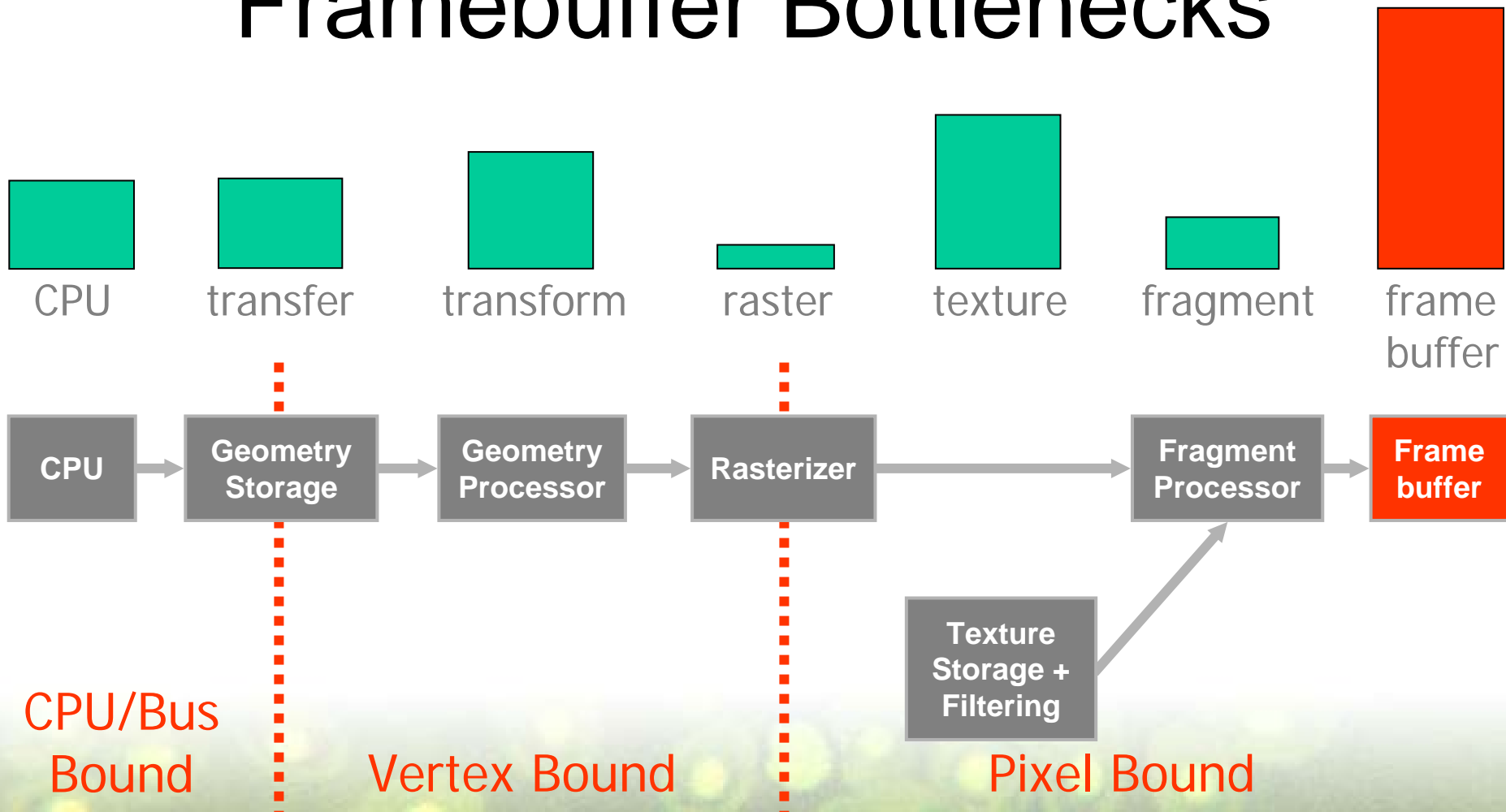
# Too Many Fragments

- Follow prior advice for maximizing depth culling efficiency
- Consider using a depth-only first pass
  - shade only the visible fragments in subsequent pass(es)
  - improve fragment throughput at the expense of additional vertex burden (only use for frames employing complex shaders)
  - Less helpful if opaque geometry is rendered front to back

# Too Much Fragment Computation

- Use a mix of texture and math instructions (they often run in parallel)
- Move constant per-triangle calculations to vertex program, send data as texture coordinates
- Do similar with values that can be linear interpolated (e.g. fresnel)
- Consider using shader levels of detail

# Framebuffer Bottlenecks



# Minimizing Framebuffer Traffic

- Collapse multiple passes with longer shaders
- Turn off Z writes for transparent objects and multipass
- Question the use of floating point frame buffers
- Reduce number and size of render-to-texture targets
  - Cube maps and shadow maps can be of small resolution and at 16-bit color depth and still look good
  - Try turning cube-maps into hemisphere maps for reflections instead
    - Can be smaller than an equivalent cube map
    - Fewer render target switches
  - Reuse render target textures to reduce memory footprint



# NVIDIA specific FB Optimizations

- Do not mask off only some color channels unless really necessary
- Use 16-bit Z depth if you can get away with it



# Use Occlusion Query

- Use occlusion query to minimize useless rendering
- It's cheap *and* easy!
- Examples:
  - multi-pass rendering
  - rough visibility determination (lens flare, portals)
- Better than glReadPixels
- Caveat: need time for query to process

# Vertex Buffer Objects

- Improve geometry throughput and CPU usage
- Provides extra information to the driver
- Can be 3-4x faster than regular vertex arrays
- Can hurt performance in non-optimal cases

# VBO Quick Tips

- Do use static VBO's for maximum performance
- Do invalidate buffer contents on streaming buffers to prevent synchronization
- Do update contiguous blocks of data in a single operation
- Do set the usage flags properly
- Do use Element Array

# VBO Don'ts

- Do not use `glArrayElement`
- Do not use non-native types
  - Double, int, RGB color in ubyte format
- Do not make really small VBO's
- Do not read VBO's unnecessarily
- Do not use one massive VBO for everything

# Vendor Specific VBO Info

- NVIDIA
  - Avoid Redundant calls to `gl*Pointer`
  - Use the 'first' parameter of `glDrawArrays`
  - Use `BufferData` instead of `Map` to replace an entire buffer
- ATI
  - Prefer `BufferData` and `BufferSubData` over `Map` to reduce synchronization overhead



# Shader Performance

- Generic category that covers many areas of the pipeline
- Compilers do much of the work

# General Shader Tips

- Lift constant or linear expressions to higher levels
- Avoid excessive control flow
- Utilize built-in functions/operations
  - Switch to specialized versions when you know certain problem domain limits
- Avoid unnecessary complexity
  - Compiler is likely better with the straight-forward code

# ATI Specific Shader Tips

- Be careful with unnecessary swizzles in fragment shaders
  - Not all swizzles are native
- Avoid unnecessary use of Rectangle Textures
- Don't use the derivative operator
- Check native instruction counts for ops
  - Available in the ATI OpenGL SDK
- Try to use ALU normalization

# GeForceFX-specific Optimizations

- Use even numbers of texture instructions
- Use even numbers of blending (math) instructions
- Use normalization cubemaps to efficiently normalize vectors
- Leverage full spectrum of instruction set (LIT, DST, SIN,...)
- Leverage swizzle and mask operators to minimize MOVs
- Minimize temporary storage
  - Use 16-bit registers where applicable (most cases)
  - Use all components in each (swizzling is free)

## Conclusion

- Complex, programmable GPUs have many potential bottlenecks
- Rarely is there but one bottleneck in a game
- Understand what you are bound by in various sections of the scene
  - The skybox is probably texture limited
  - The skinned, dot3 characters are probably transfer or transform limited
- Exploit imbalances to get things for free