OpenGL Performance Tuning

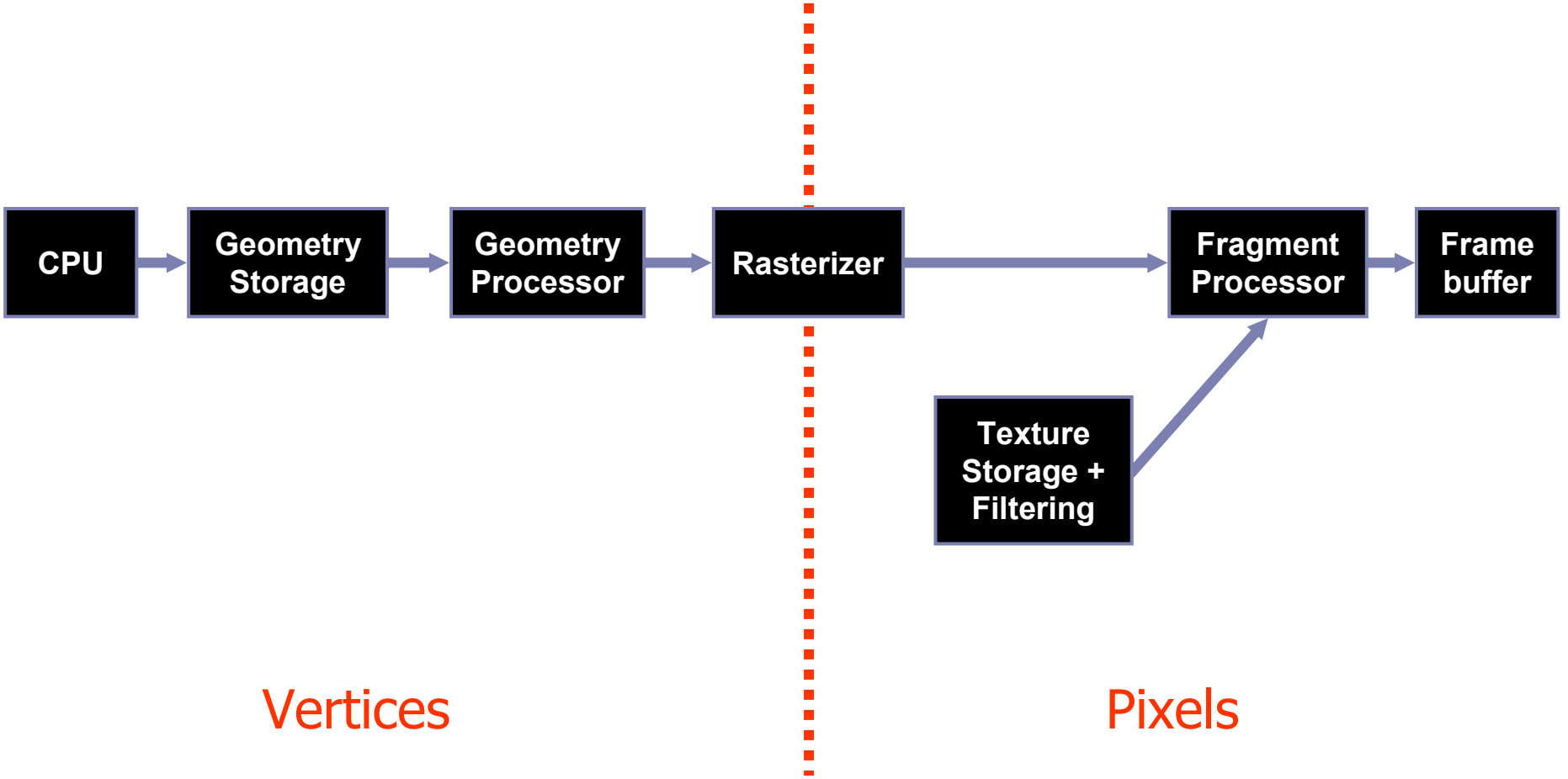
John Spitzer

NVIDIA Corporation

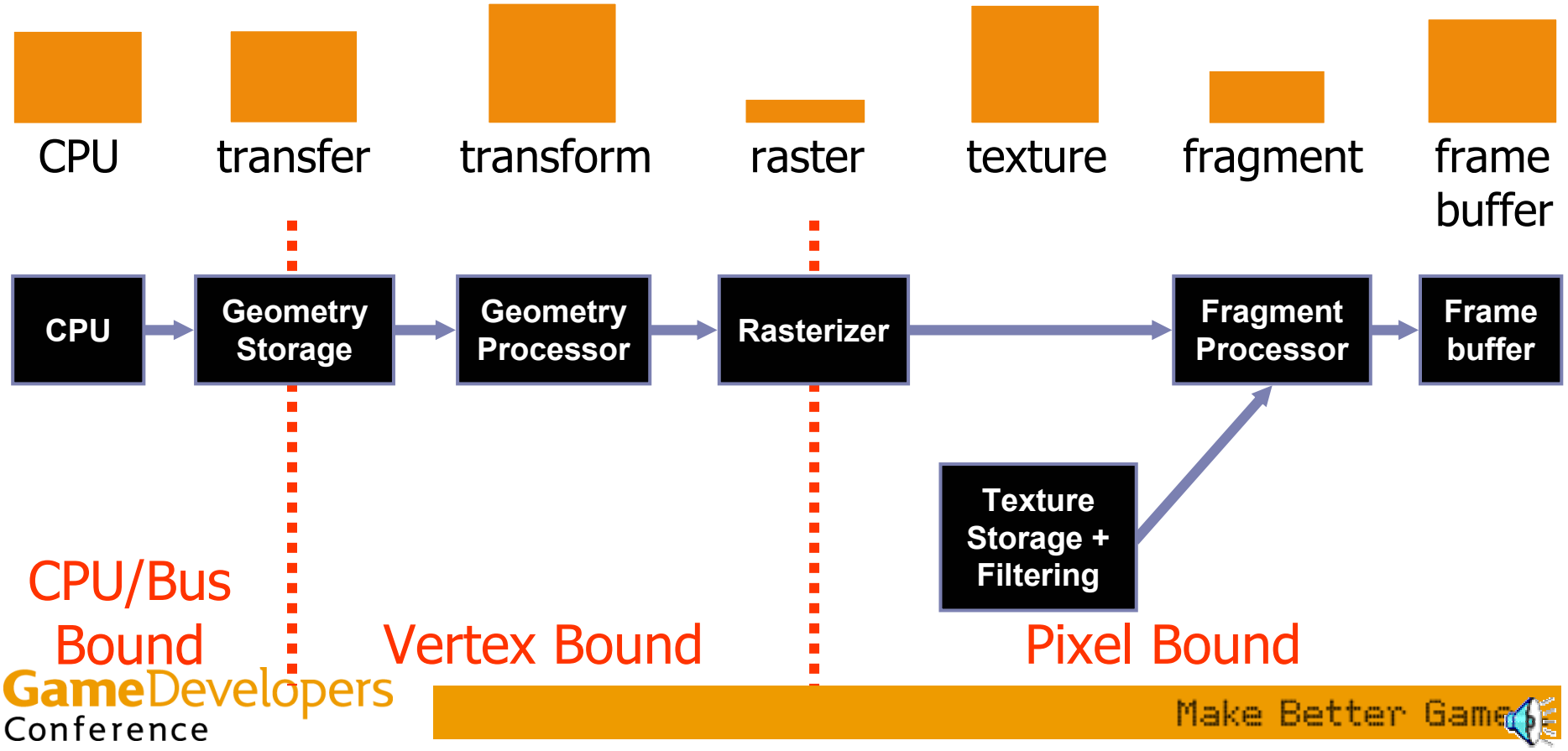
Overview

- Understand the stages of the graphics pipeline
- *Cherchez la bottleneck*
- Once found, either eliminate or balance

Simplified Graphics Pipeline

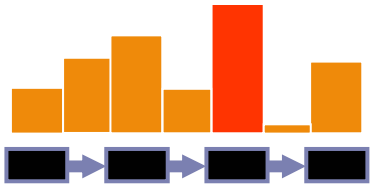


Possible Pipeline Bottlenecks



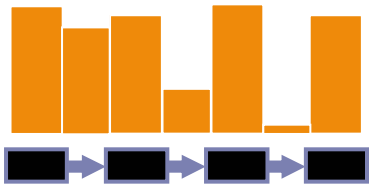
Battle Plan for Better Performance

- Locate the bottleneck(s)

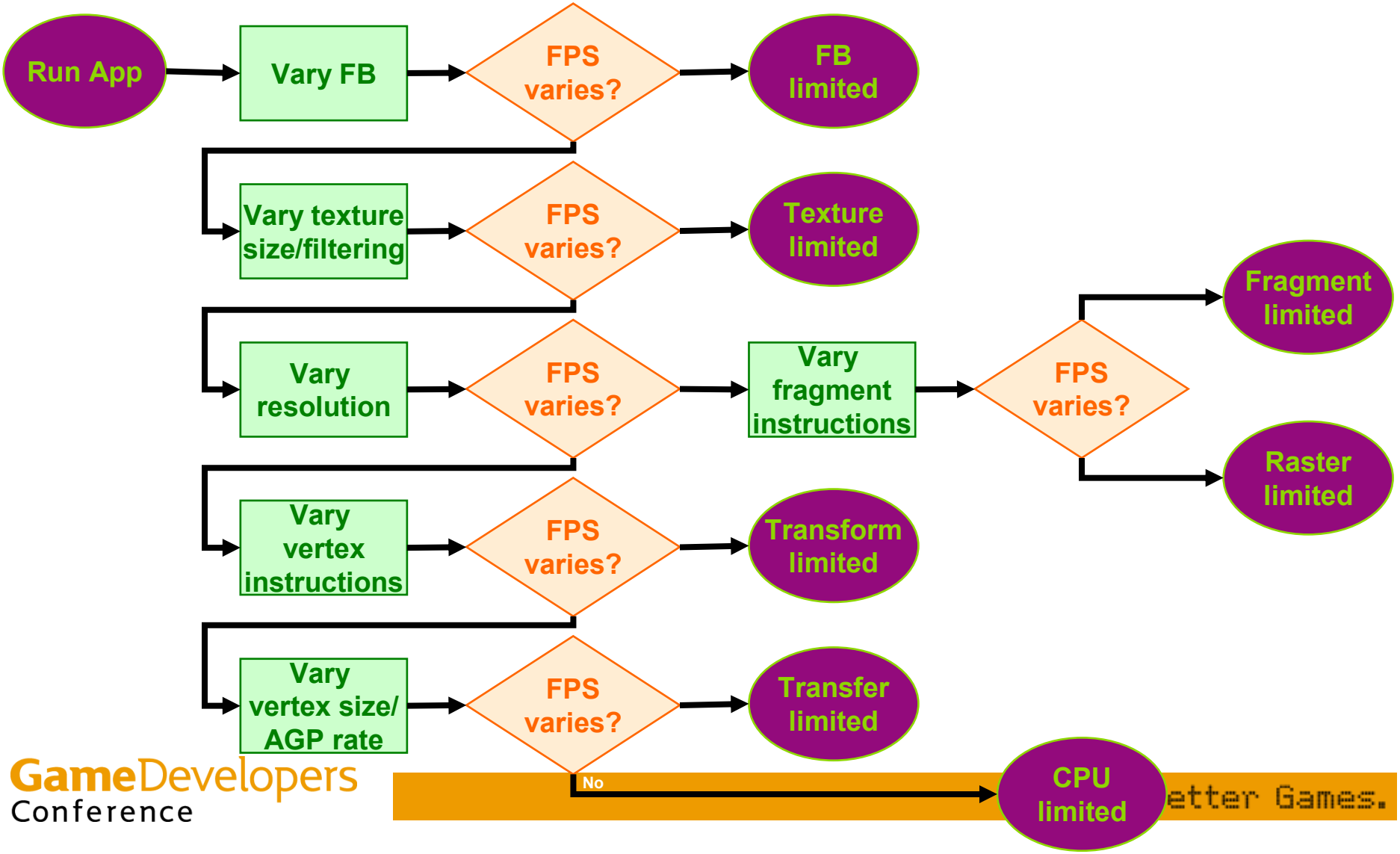


- Eliminate the bottleneck (if possible)
  - Decrease workload of the bottlenecked stage

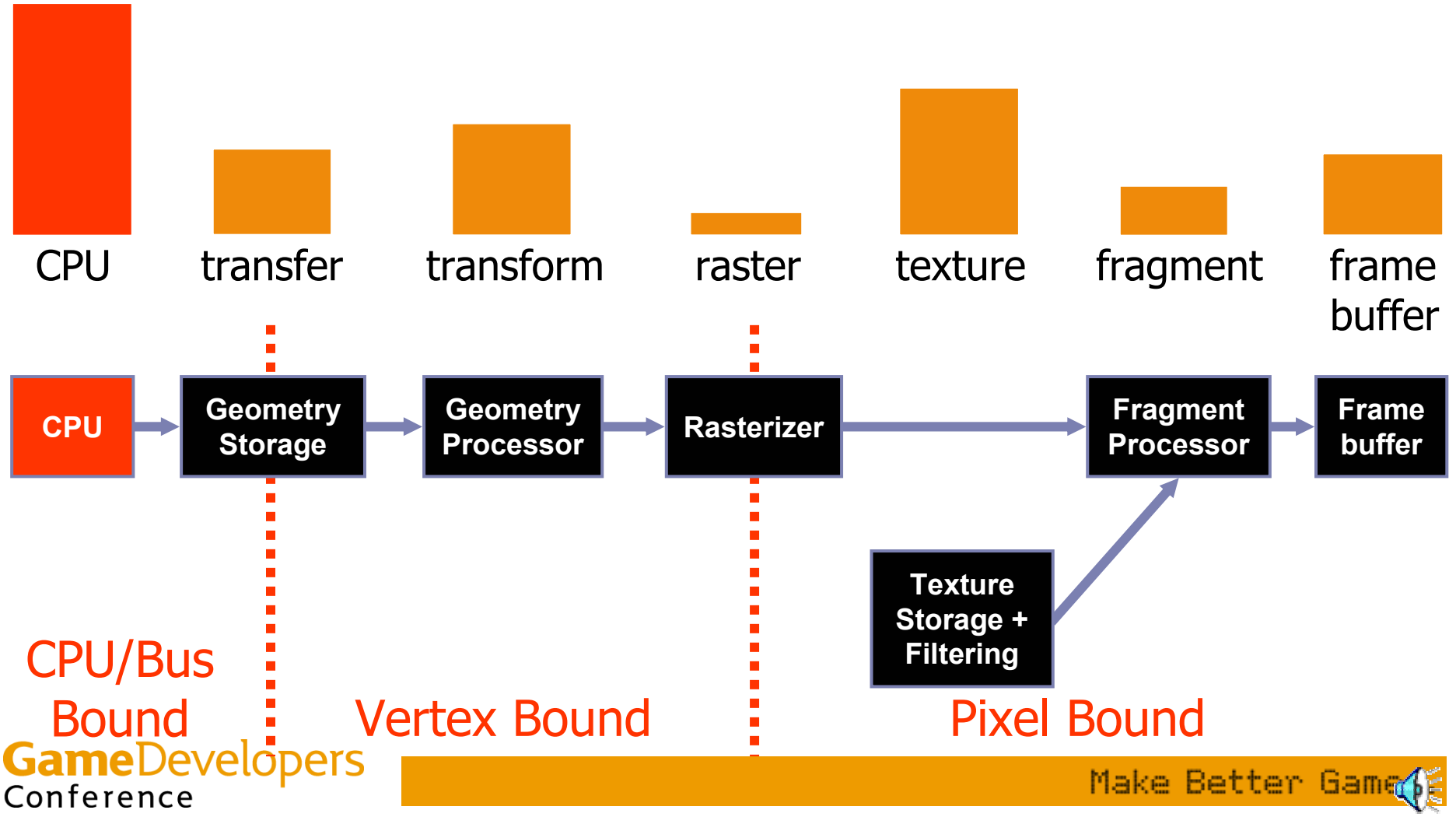
- Otherwise, balance the pipeline
  - Increase workload of the non-bottlenecked stages:



Bottleneck Identification



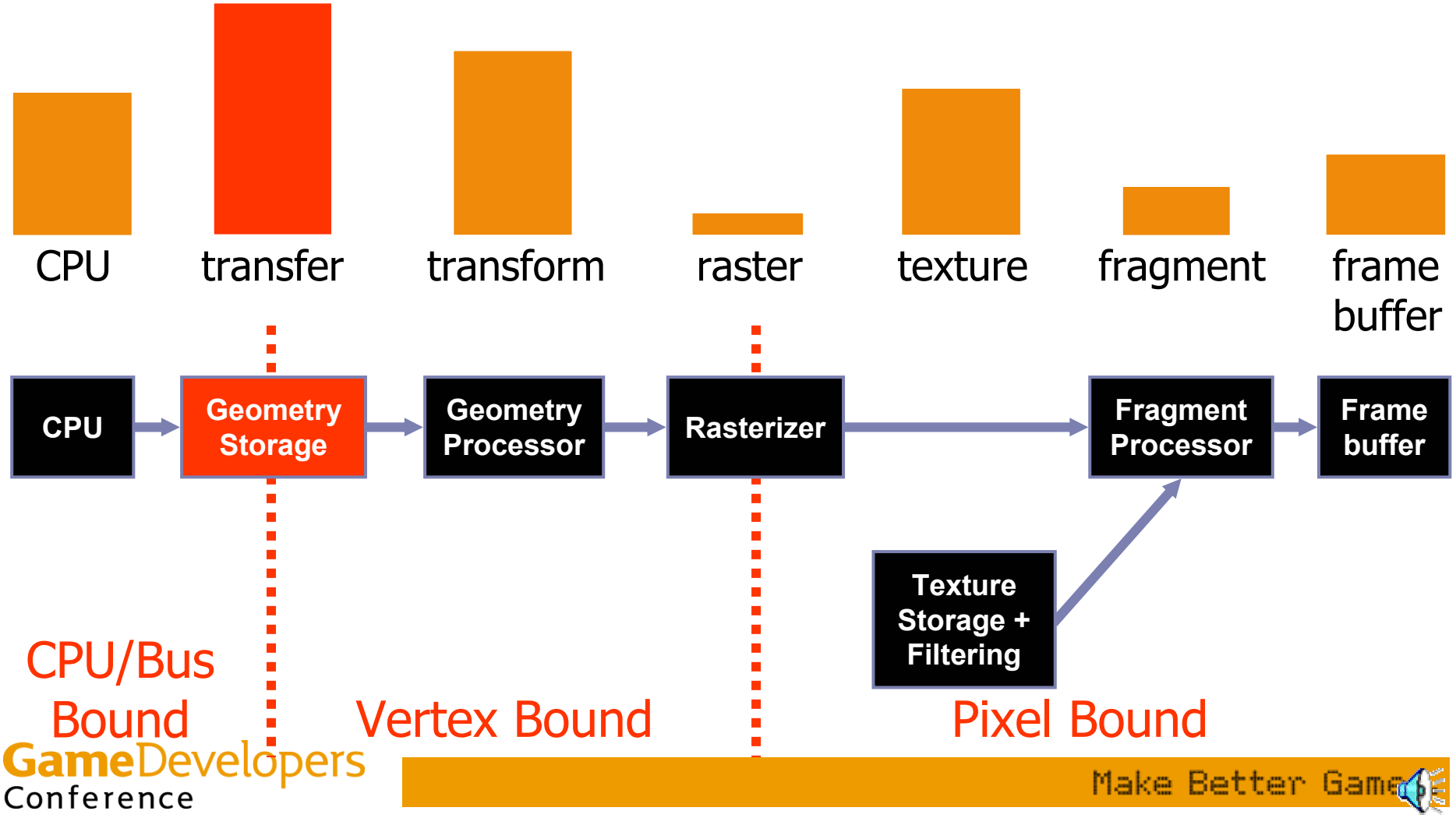
CPU Bottlenecks



CPU Bottlenecks

- Application limited (most games are in some way)
- Driver or API limited
  - too many state changes (bad batching)
  - using non-accelerated 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

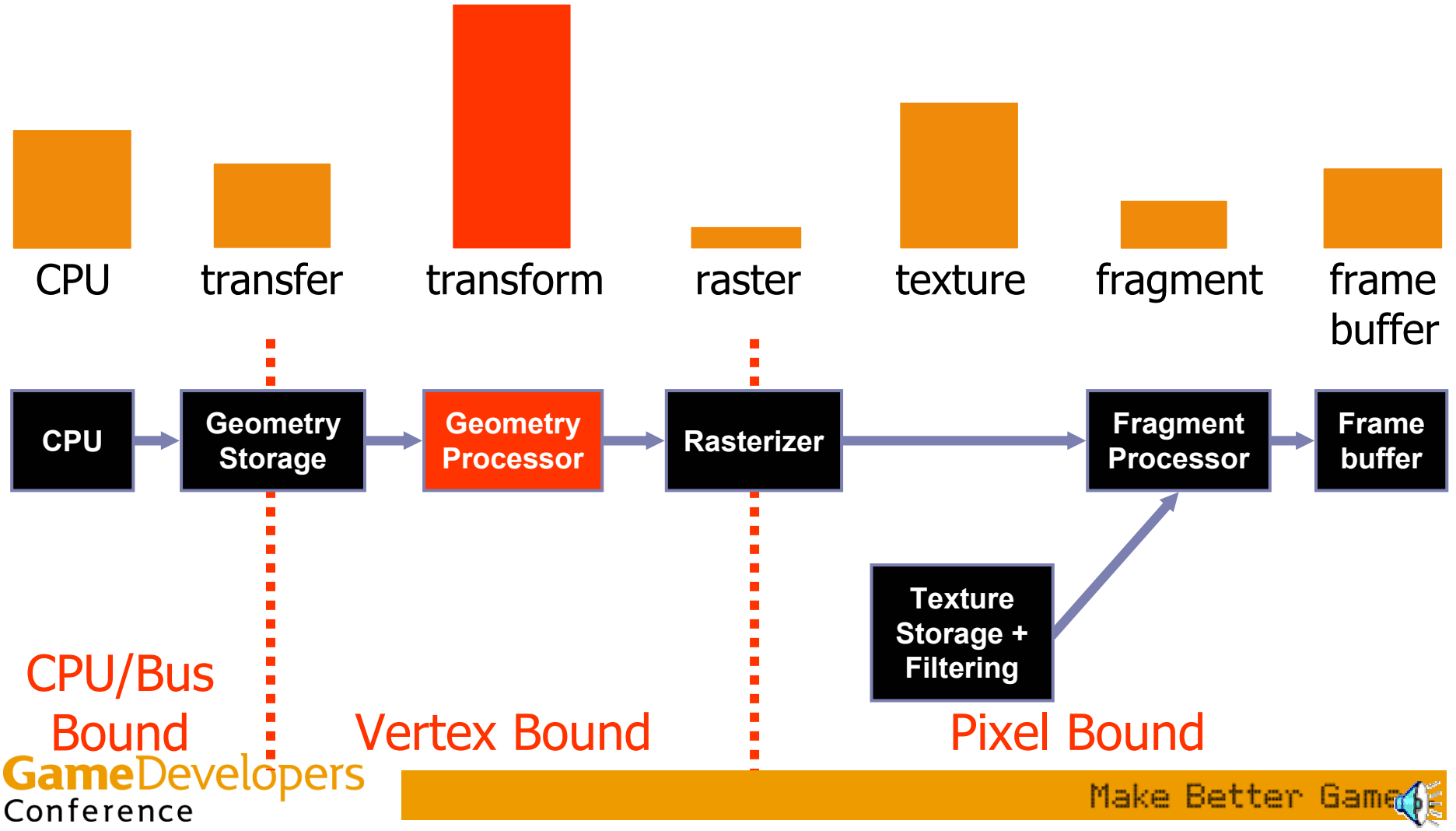
- Vertex data problems
  - size issues (just under or over 32 bytes)
  - non-native types (e.g. double, packed byte normals)
- Using the wrong API calls
  - Immediate mode, non-accelerated vertex arrays
  - Non-indexed primitives (e.g. glDrawArrays)
- AGP misconfigured or aperture set too small

Optimizing Geometry Transfer

- Static geometry – display lists okay, but ARB_vertex_buffer_object will be better

- Dynamic geometry - use ARB_vertex_buffer_object
  - vertex size ideally multiples of 32 bytes (compress or pad)
  - access vertices in sequential (cache friendly) 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

Geometry Transform Bottlenecks



Geometry Transform Bottlenecks

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

Too Many Vertices

- Favor triangle strips/fans over lists (fewer vertices)
- Use levels of detail (but beware of CPU overhead)
- Use bump maps to fake geometric detail

Too Much Vertex Computation: Fixed Function

- Avoid superflous 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 (and only if) good shortcut exists
  - example: texture matrix only needs to be 2x2
  - not recommended for optimizing fixed function lighting

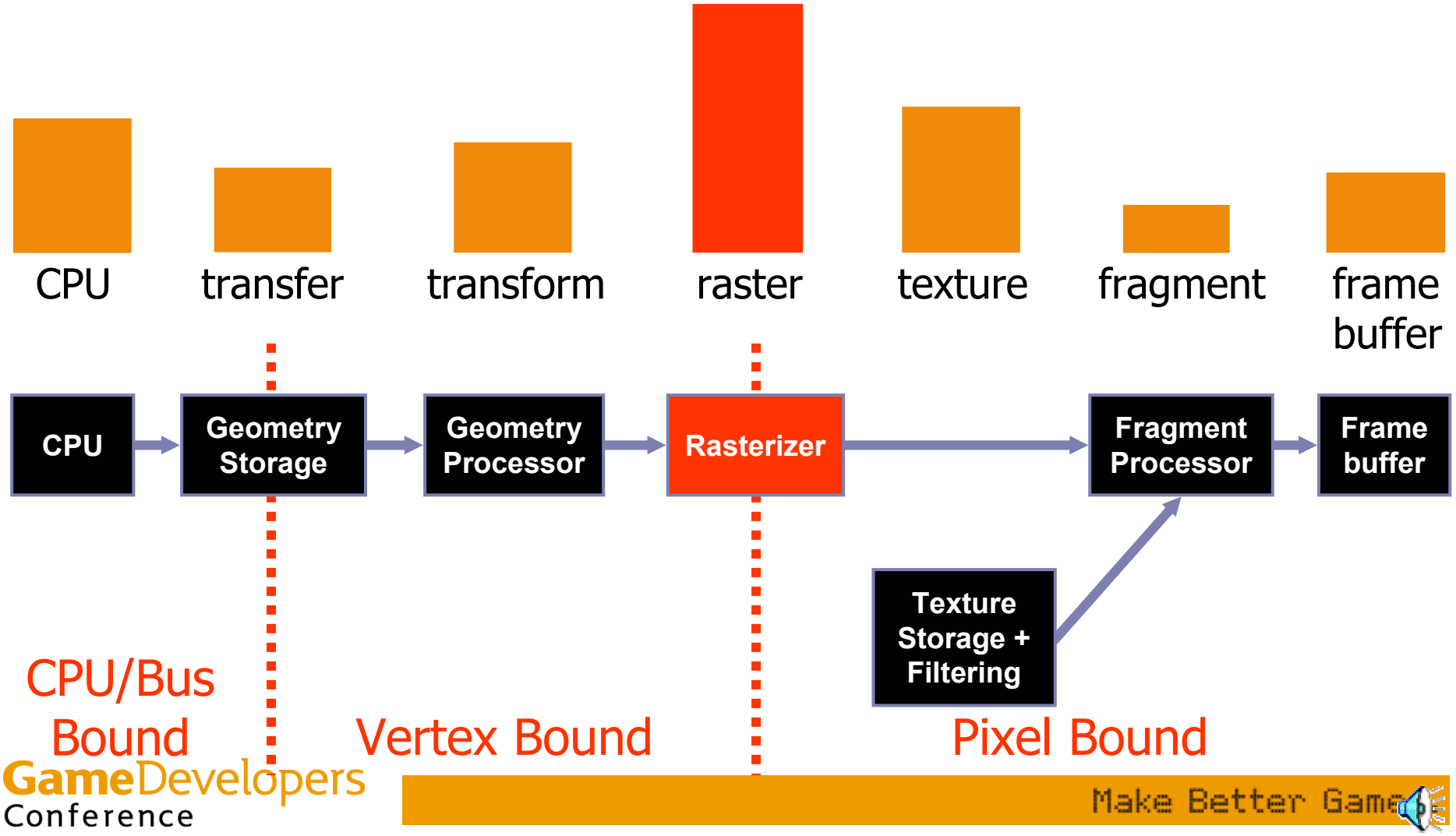
Too Much Vertex Computation: Vertex Programs

- 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 triangle fans/strips over lists

Rasterization Bottlenecks



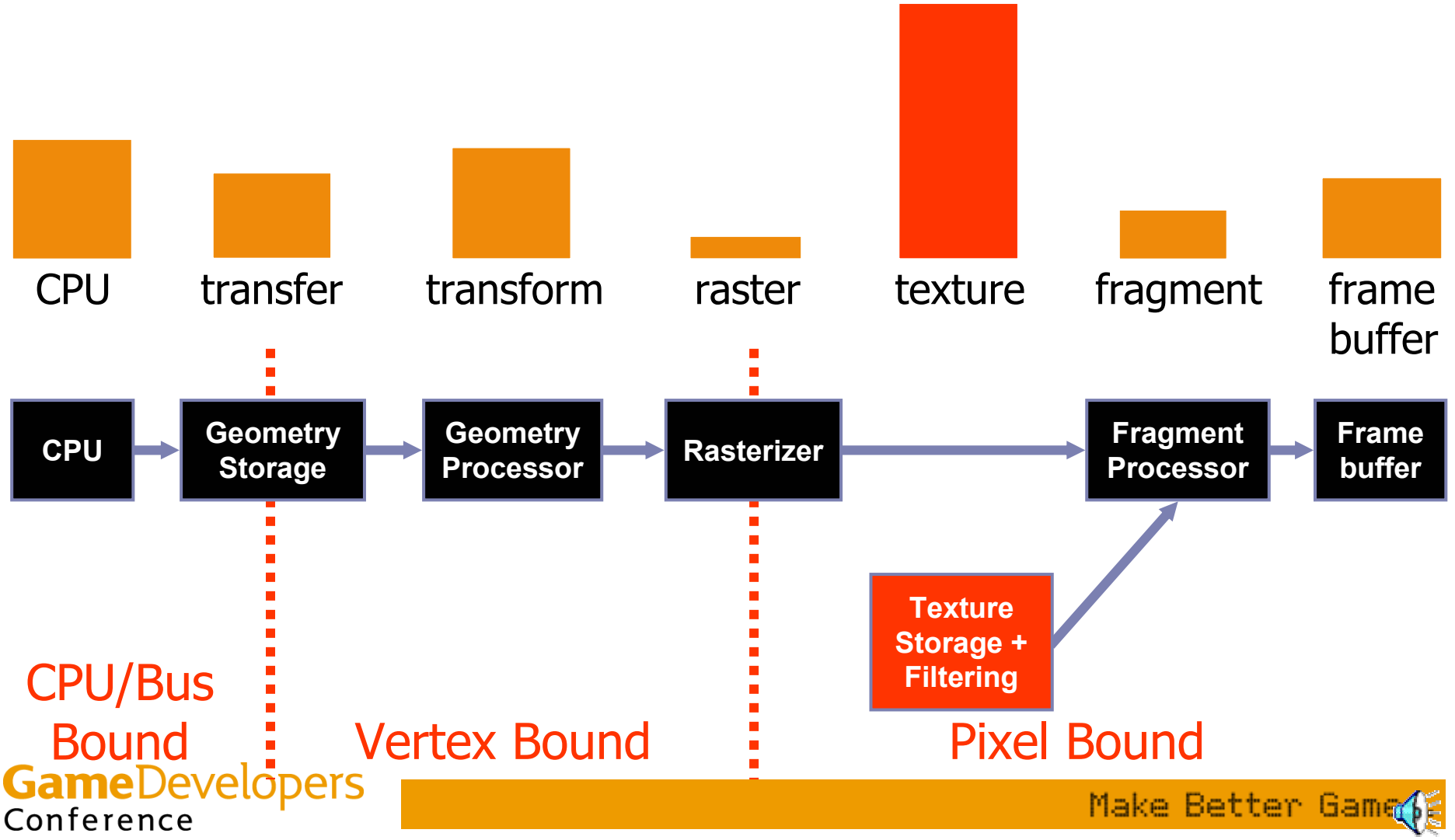
Rasterization

- 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
- Use scissor to minimize superfluous fragment generation for stencil shadow volumes
- Avoid polygon offset unless you really need it
- NVIDIA advice
  - use depth bounds test 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
  - New GPUs allow floating point 4xfp16 and 4xfp32 formats
- Compress textures:
  - Collapse monochrome channels into alpha
  - Use 16-bit color depth when possible (environment maps and shadow maps)
  - Use DXT compression

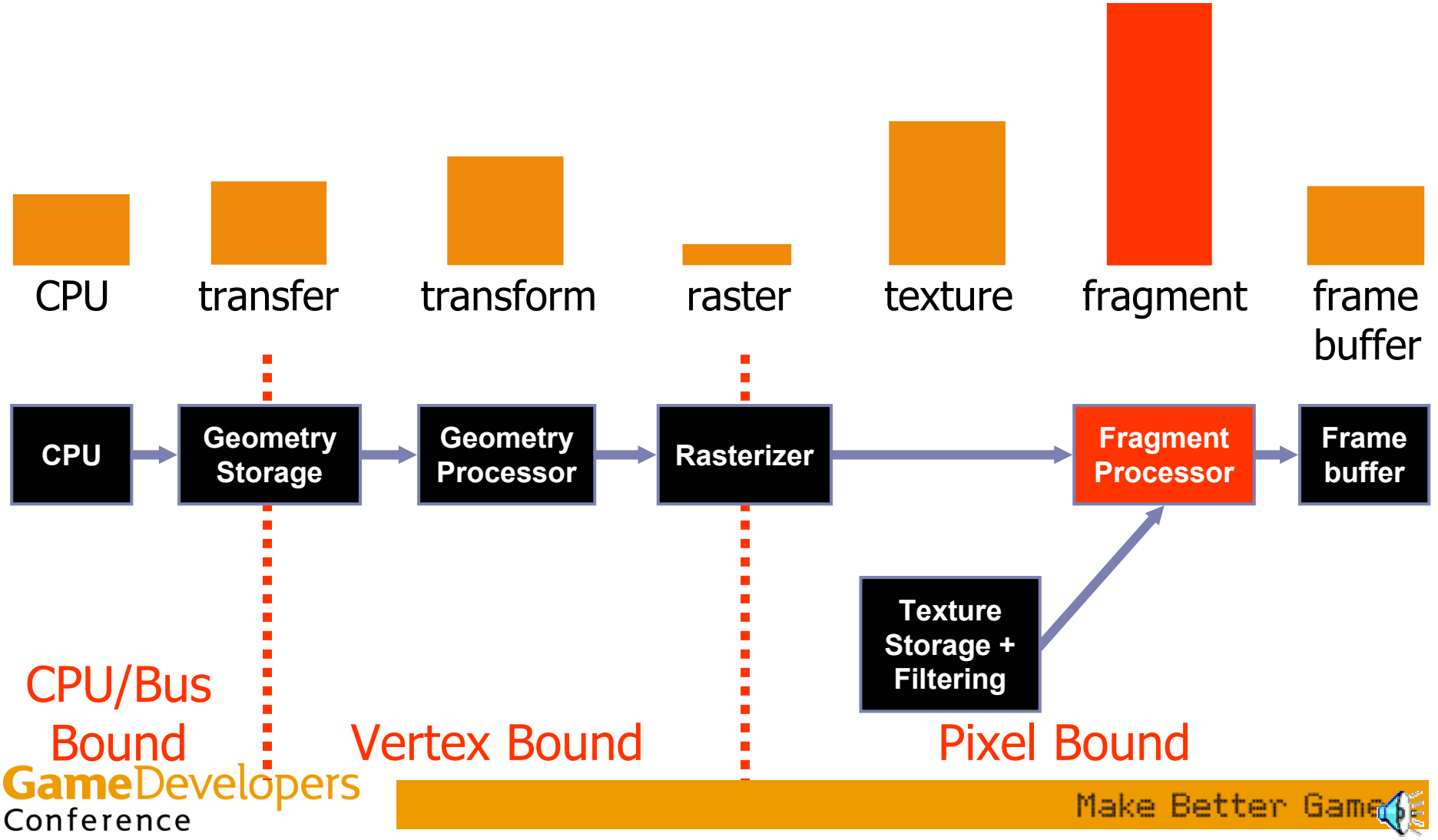
Poor Texture Cache Utilization

- Localize texture accesses
  - beware of dependent texturing
  - ALWAYS use mipmapping
  - use trilinear/aniso only when necessary (more later!)
- 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
- Similarly use anisotropic filtering judiciously
  - even more expensive than trilinear
  - not useful for environment maps (again, distortion)

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)

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

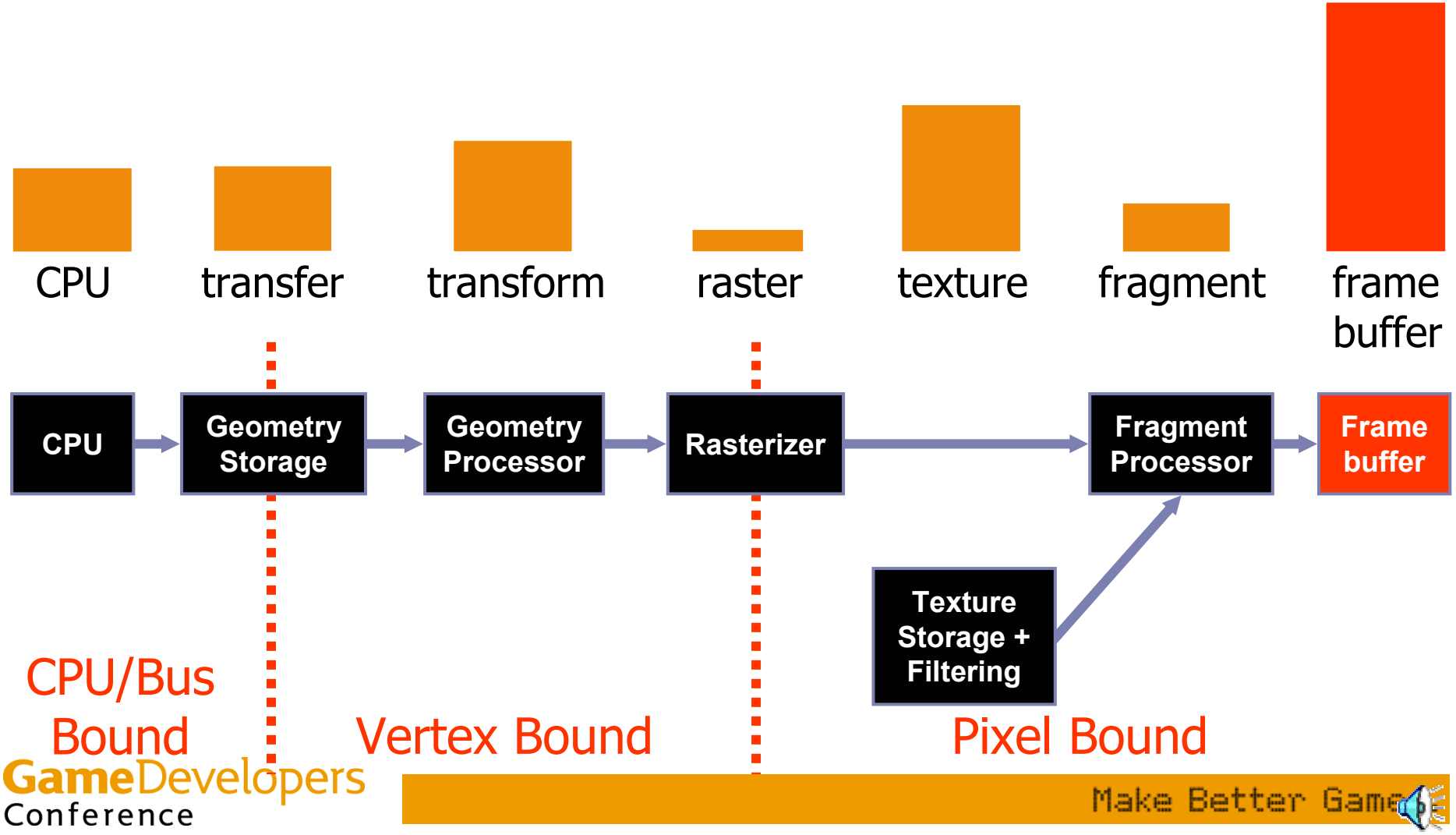
GeForceFX-specific Optimizations

- 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)

Radeon 9500+ Optimizations

- Understand Native vs. Non-Native Ops
  - SIN, COS, LIT – emulated
- Enable co-issue of scalar and vector instructions
  - Perform scalar math in the alpha channel
  - Only write to RGB when doing a 3-vec op
- Group non-dependent texture instructions
- Avoid unnecessary complex swizzles
- Tradeoff ALU/Texture instructions
  - Cubemap lookup versus normalize
  - SIN versus texture fetch

Framebuffer Bottlenecks



Minimizing Framebuffer Traffic

- Collapse multiple passes with longer shaders (not always a win)
- Turn off Z writes for transparent objects and multipassQuestion the use of floating point frame buffers
- Use 16-bit Z depth if you can get away with it
- 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
- Do not mask off only some color channels unless really necessary (NVIDIA only)

Pixel Rectangles (Blits)

- Copying pixels
  - Match formats as closely as possible
    - match size and components
    - Presence/lack of alpha is less important
  - Avoid non-identity pixel transfer operations
- Writing pixels
  - Match the format as closely as possible
    - Prefer BGRA order over RGBA
  - Avoid the non-packed 32-bit integer formats
- Reading pixels
  - Match the format as closely as possible
  - Avoid poorly aligned data
    - RGB as unsigned bytes
  - Avoid non-packed 32-bit integers
  - Use other alternatives when available (ccclusion query)

Finally… 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)
- Caveats:
  - need time for query to process
  - can add fillrate overhead

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

Questions, comments, feedback?

- John Spitzer, spit@nvidia.com
- Evan Hart, ehart@ati.com
- Credits
  - The NVIDIA developer technology team
  - The ATI ISV support team