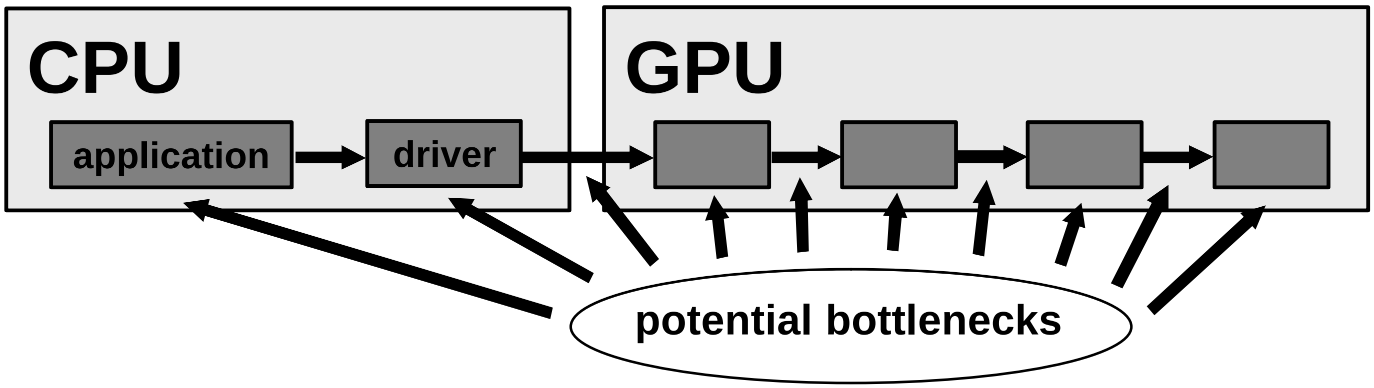
Optimizing the Graphics Pipeline

Cem Cebenoyan and Matthias Wloka

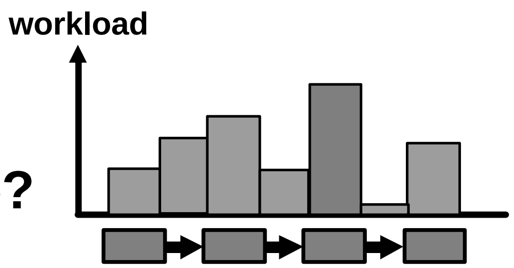
Overview



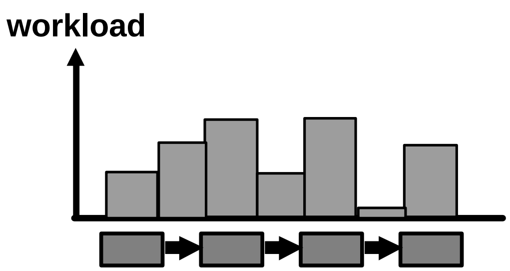
- The bottleneck determines overall throughput
- In general, the bottleneck varies over the course of an application and even over a frame
- For pipeline architectures, getting good performance is all about finding and eliminating bottlenecks

Locating and eliminating bottlenecks

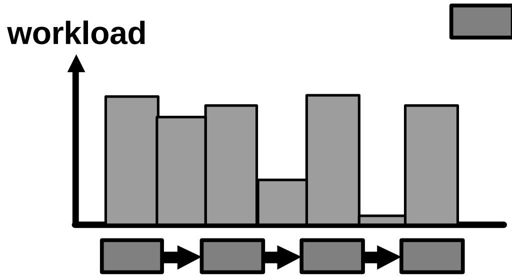
- Location: For each stage
  - Vary its workload
    - Measurable impact on overall performance?
  - Clock down
    - Measurable impact on overall performance?



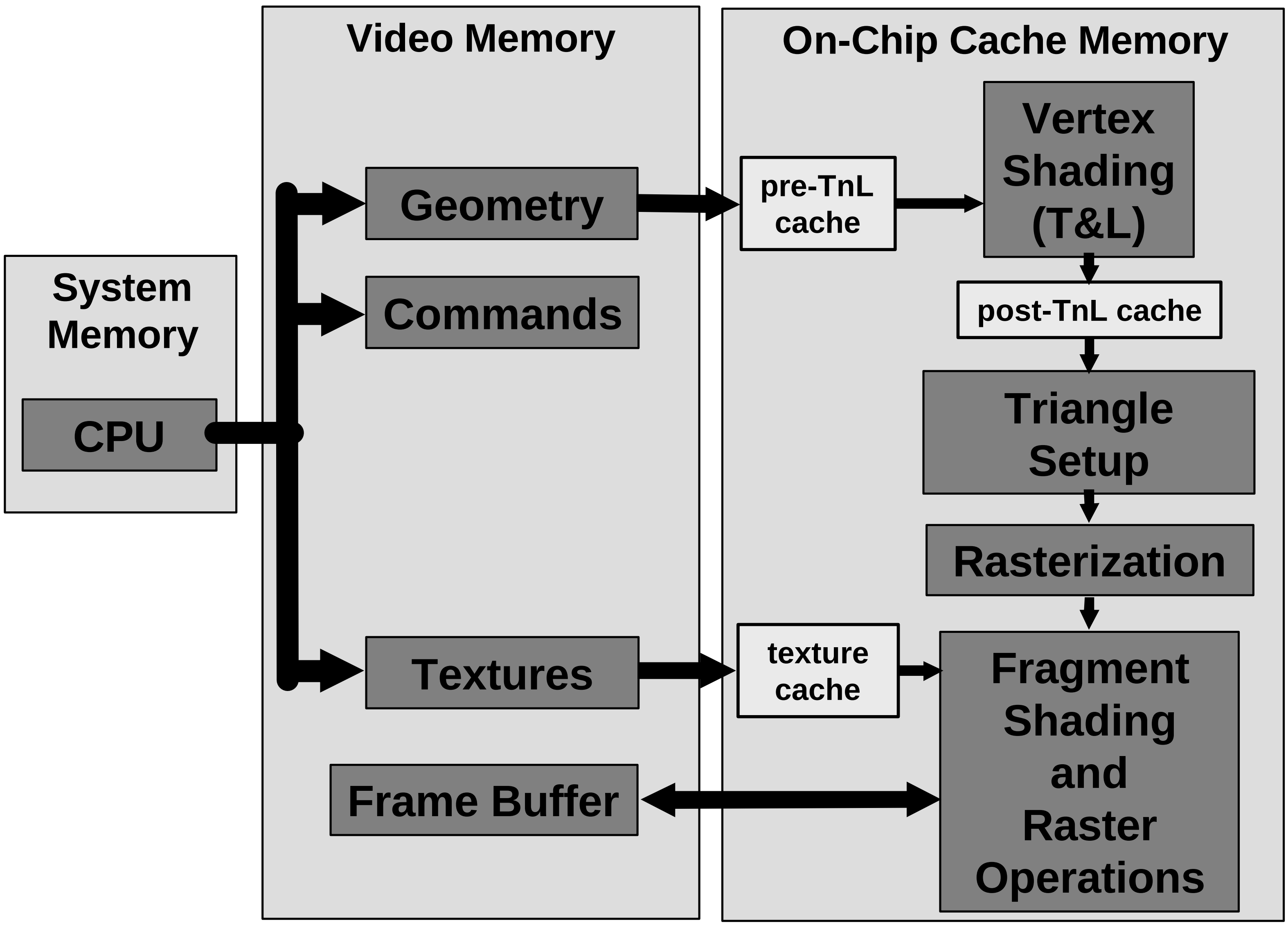
- Elimination:
  - Decrease workload of bottleneck:



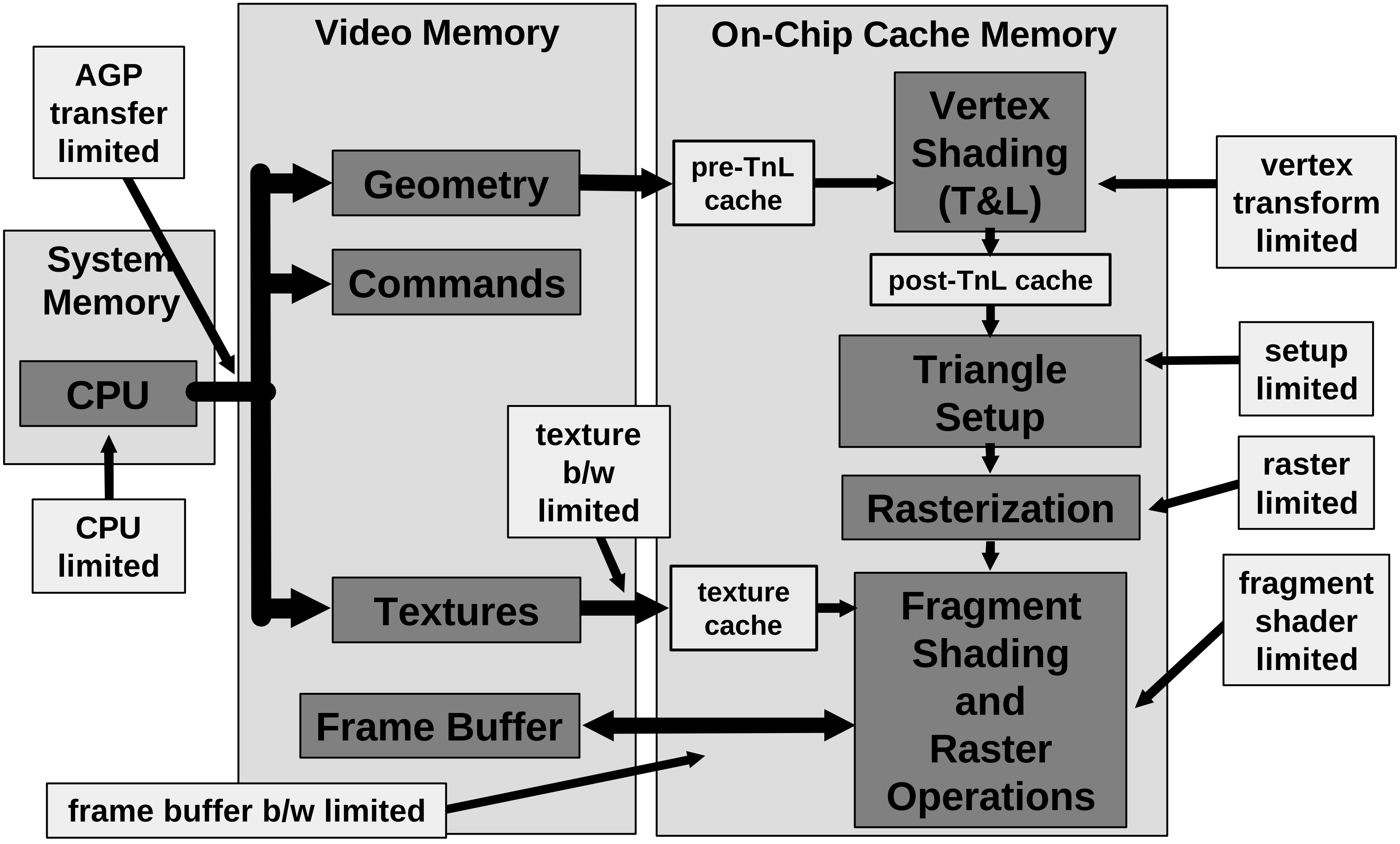
  - Increase workload of non-bottleneck stages:



Graphics rendering pipeline



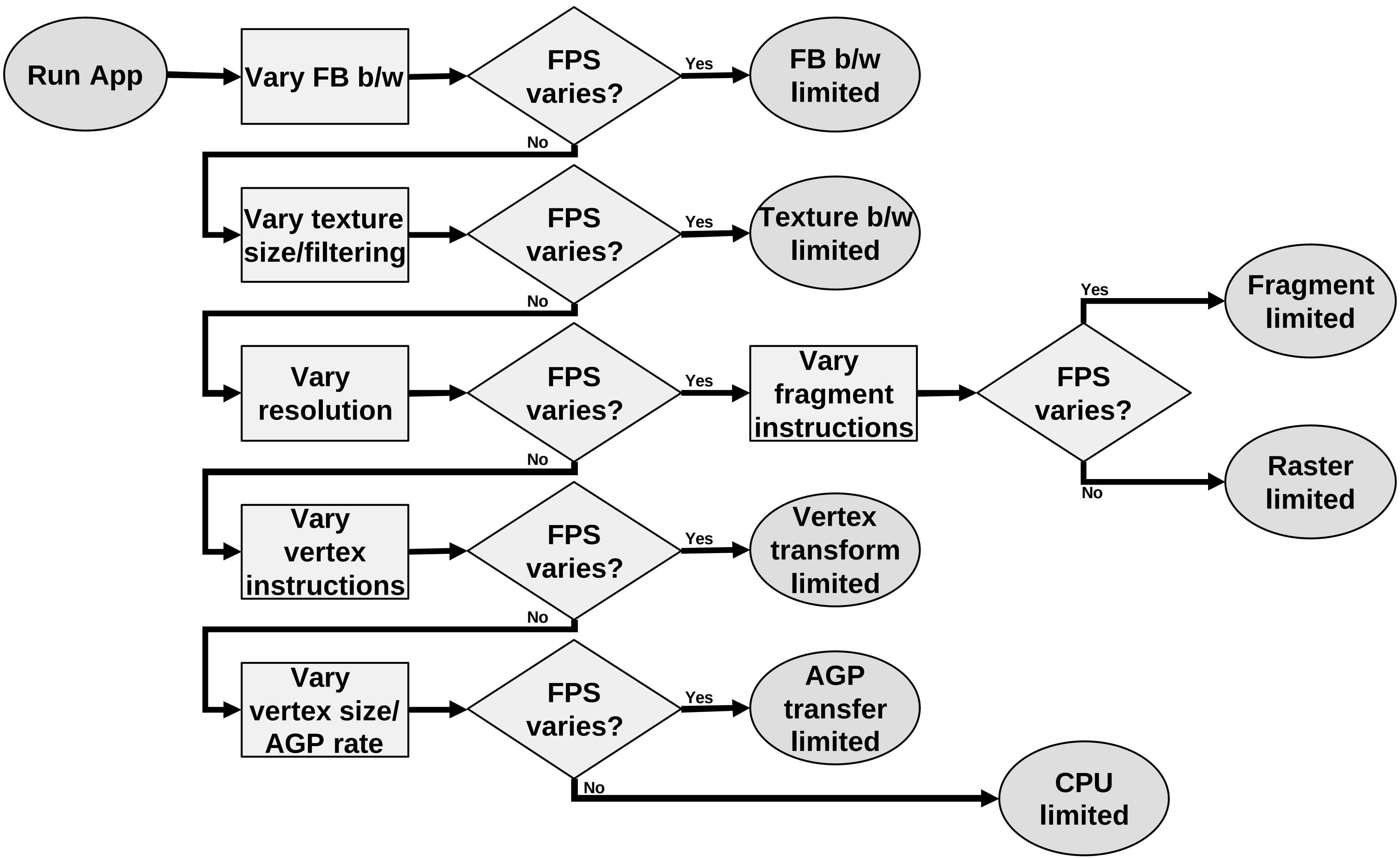
Potential Bottlenecks



Graphics rendering pipeline bottlenecks

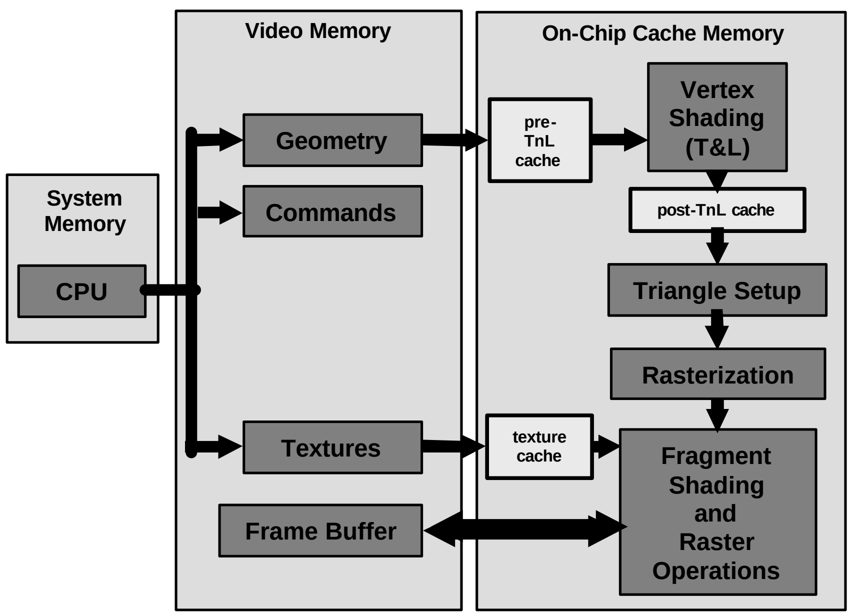
- The term “transform bound” often means the bottleneck is “anywhere before the rasterizer”
- The term “fill bound” often means the bottleneck is “anywhere after setup”
- Can be both transform and fill bound over the course of a single frame!

Bottleneck identification



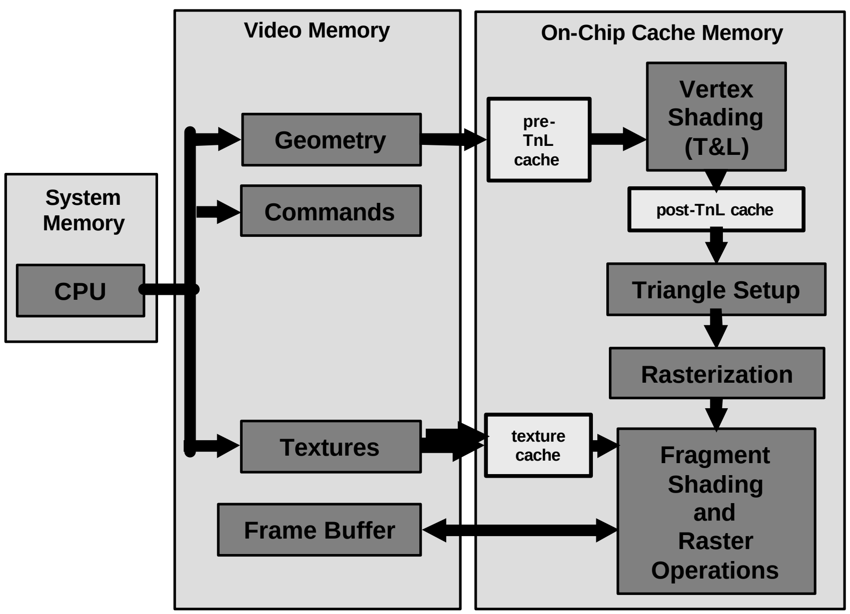
Frame Buffer B/W Limited

- Vary all render target color depths (16-bit vs. 32-bit)
  - If frame rate varies, application is frame buffer b/w limited



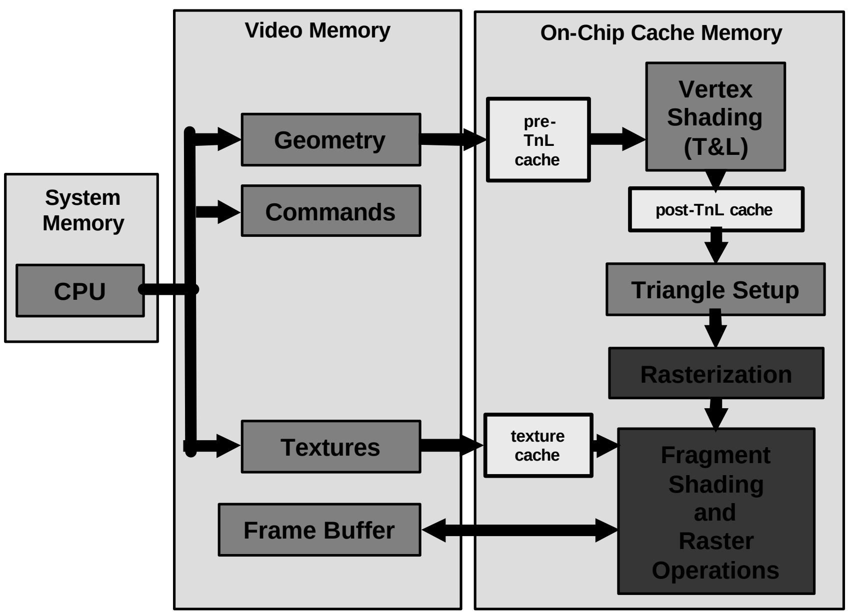
Texture B/W Limited

- Otherwise, vary texture sizes or texture filtering
  - Force MIPMAP LOD Bias to +10
  - Point filtering versus bilinear versus tri-linear
  - If frame rate varies, application is texture b/w limited



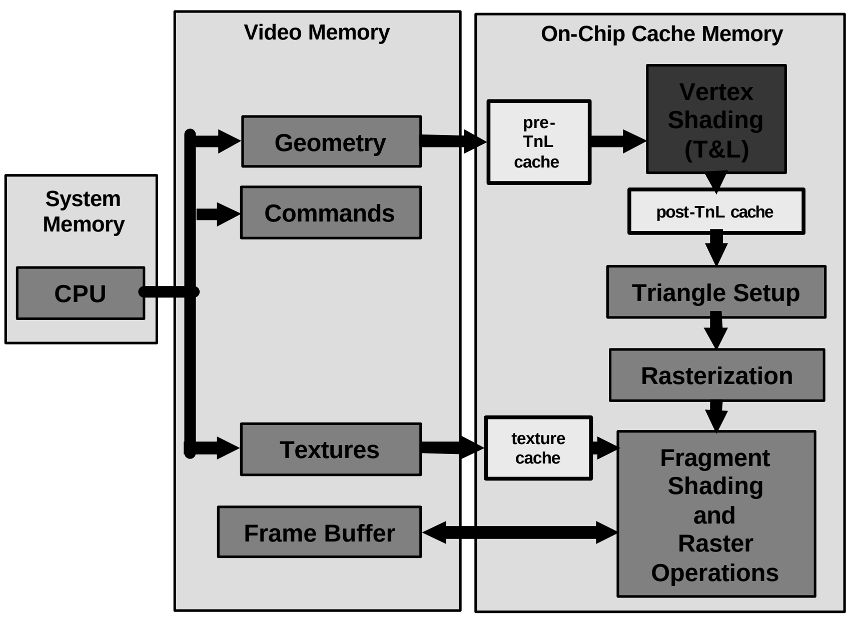
Fragment or Raster Limited

- Otherwise, vary all render target resolutions
  - If frame rate varies, vary number of instructions of your fragment programs
    - If frame rate varies, application is fragment shader limited
    - Otherwise, application is raster limited



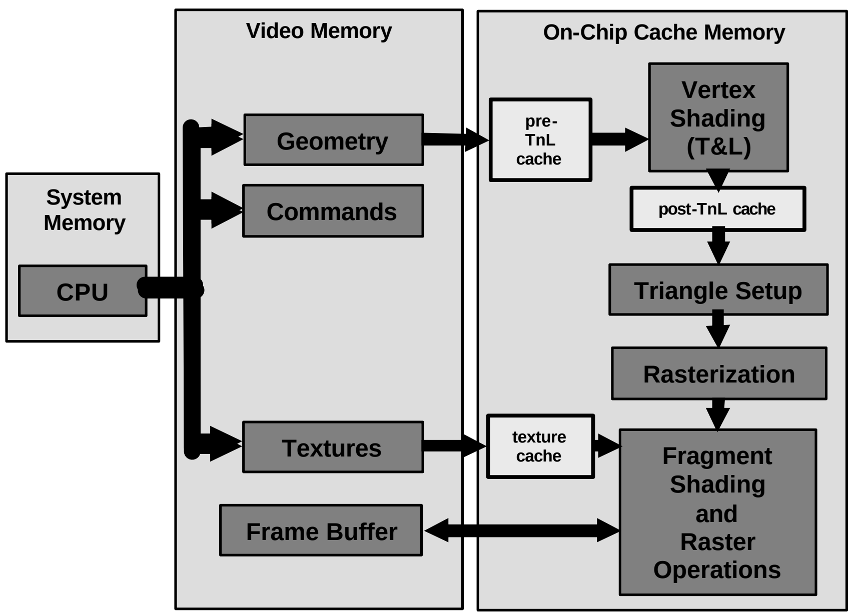
Vertex Transform Limited

- Otherwise, vary the number of instructions of your vertex programs
  - Careful: do not add instructions that are optimizable
  - If frame rate varies, application is vertex transform limited



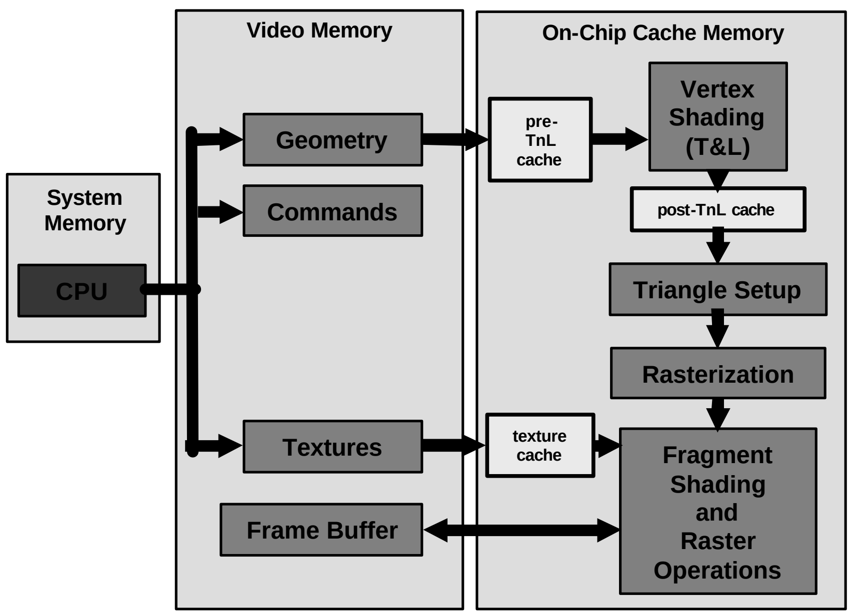
AGP Transfer Limited

- Otherwise, vary vertex format size or AGP transfer rate
  - If frame rate varies, application is AGP transfer limited



CPU Limited

- Otherwise, application is CPU limited



Bottleneck identification shortcuts!

- Run identical GPUs on different speed CPUs
  - If frame rate varies, application is CPU limited
    - Completely iff frame rate is proportional to CPU speed
- Force AGP to 1x from BIOS
  - If frame rate varies, application is AGP b/w limited
- Underclock your GPU
  - If slower core clock affects performance, application is vertex-transform, raster, or fragment-shader limited
  - If slower memory clock affects performance, application is texture or frame-buffer b/w limited

Overall optimization: Batching

- Eliminate small batches:
  - Use thousands of vertices per vertex buffer/array
  - Draw as many triangles per call as possible
    - thousands of triangles per call
  - ~50k DIP/s COMPLETELY saturate 1.5GHz Pentium 4
    - 50fps means 1k DIP/frame!
    - Up to you whether drawing 1k tri/frame or 1M tri/frame
  - Use degenerate triangles to join strips together
  - Use texture pages
  - Use a vertex shader to batch instanced geometry

Overall optimization: Indexing, sorting

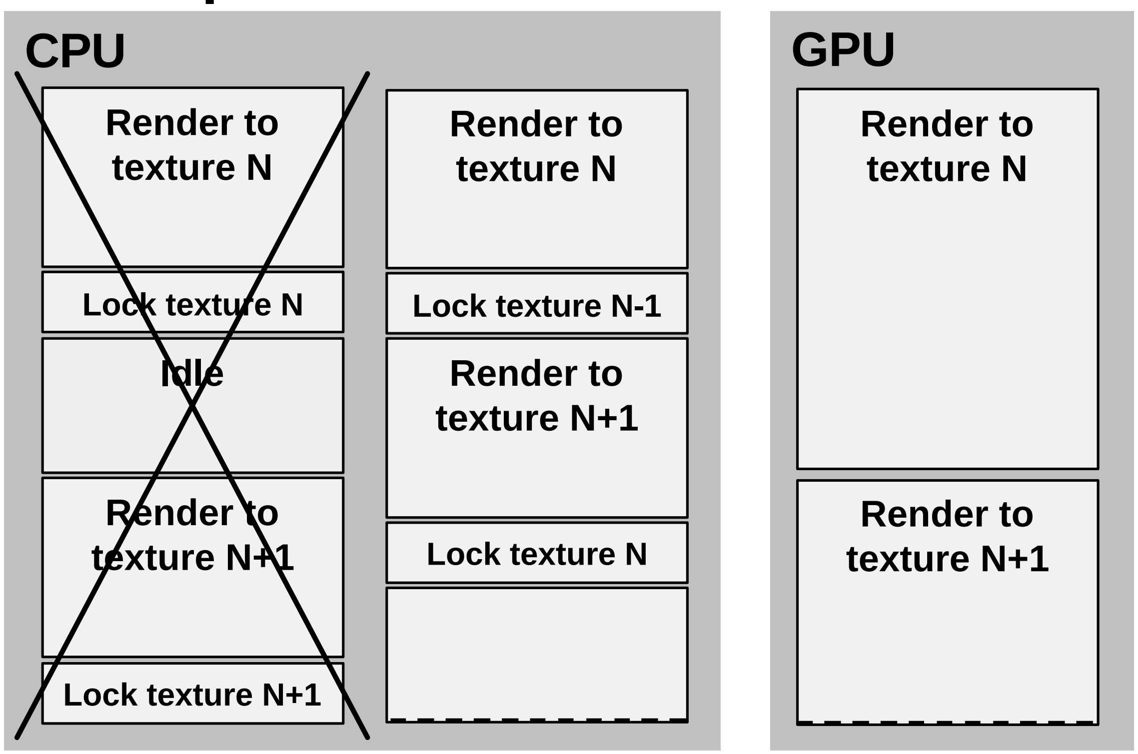
- **Use indexed primitives (strips or lists)**
  - **Only way to use the pre- and post-TnL cache!**
  - **(Non-indexed *strips* also use the cache)**
- **Re-order vertices to be sequential in use**
  - **To maximize cache usage!**
- **Lightly sort objects front to back**
- **Sort batches per texture and render states**

Overall optimization: Occlusion query

- **Use occlusion query to protect vertex and pixel throughput:**
  - **Multi-pass rendering:**
    - **During the first pass, attach a query to every object**
    - **If not enough pixels have been drawn for an object, skip the subsequent passes**
  - **Rough visibility determination:**
    - **Draw a quad with a query to know how much of the sun is visible for lens flare**
    - **Draw a bounding box with a query to know if a portal or a complex object is visible and if not, skip its rendering**

Overall optimization: Beware of resource locking!

- A call that locks a resource (Lock, glReadPixels) is potentially blocking if misplaced:
  - CPU is idling, waiting for the GPU to flush
- Avoid it if possible
- Otherwise place it so that the GPU has time to flush:



CPU bottlenecks: Causes

- Application limited:
  - Game logic, AI, network, file I/O
  - Graphics should be limited to simple culling and sorting
- Driver or API limited: Something is wrong!
  - Off the fast path
  - Pathological use of the API
  - Small batches
- Most graphics applications are CPU limited
  - Most graphics applications are CPU limited

CPU bottlenecks: Solutions

- Use CPU profilers (e.g., Intel's VTune)
  - Driver should spend most of its time idling
    - Easy to detect by looking at assembler: idle loop
- Increase batch-sizes aggressively
  - At the expense of the GPU!
- For rendering
  - Prefer GPU brute-force, but simple on CPU
  - Avoid smart (but expensive) CPU algorithms designed to reduce render load

AGP transfer bottlenecks

- Unlikely bottleneck for AGP4x
  - AGP8x is here
- Too much data crosses the AGP bus:
  - Useless data
    - Solution: Eliminate unused vertex attributes
    - Solution: Use 16-bit indices instead of 32-bit if possible
  - Too many dynamic vertices
    - Solution: Decrease number of dynamic vertices by using vertex shaders to animate static vertices, for example
  - Poor management of dynamic data
    - Solution: Use the right API calls
  - Overloaded video memory
    - Solution: Make sure frame buffer, textures and static vertex buffers fit into video memory

AGP transfer bottlenecks

- Data transferred in an inadequate format:
  - Vertex size should be multiples of 32 bytes
    - Solution: Adjust vertex size to multiples of 32 bytes:
      - Compress components and use vertex shaders to decompress
      - Pad to next multiple
  - Non-sequential use of vertices (pre-TnL cache)
    - Solution: Re-order vertices to be sequential in use
      - Use NVTriStrip

Optimizing geometry transfer

- Static geometry:
  - Create a write-only vertex buffer and only write to it once
- Dynamic geometry:
  - Create a dynamic vertex buffer
  - Lock with DISCARD at start of frame
    - Then append with NOOVERWRITE until full
  - Use NOOVERWRITE more often than DISCARD
    - Each DISCARD takes either more time or more memory
    - So NOOVERWRITE should be most common
  - Never use no flags
- Semi-dynamic geometry:
  - For procedural or demand-loaded geometry
  - Lock once, use for many frames
  - Try both static & dynamic methods

Vertex transform bottlenecks

- Unlikely bottleneck
  - Unless you have 1 Million Tri/frame (Cool!)
  - Or max out vertex shader limits (Cool!)
    - >128 vertex shader instructions
- Too many vertices
  - Solution: Use level of detail
  - But: Rarely a problem because GPU has a lot of vertex processing power
  - So: Don’t over-analyze your level of details determination or computation in the CPU
  - 2 or 3 static LODs are fine

Vertex transform bottleneck causes

- **Too much computation per vertex:**
  - **Vertex lighting with lots of or expensive lights or lighting model (local viewer)**
    - **Directional < point < spot**
  - **Texgen enabled or texture matrices aren’t identity**
  - **Vertex shaders with:**
    - **Lots of instructions**
    - **Lots of loop iterations or branching**
  - **Post-TnL vertex cache is under-utilized**
    - **Use nvTriStrip**

Vertex transform bottleneck solutions

- Re-order vertices to be sequential in use, use PostTnL cache
  - NVTriStrip
- Take per-object calculations out of the shader
  - compute in CPU and save as program constants
- Reduce instruction count via complex instructions and vector operations
  - Or use Cg
- Scrutinize every mov instruction
  - Or use Cg
- Consider using shader level of details
  - Do far-away objects really need 4-bone skinning?
- Consider moving per-vertex work to per-fragment
- Force increased screen-resolution and/or anti-aliasing!

Setup bottleneck

- Practically never the bottleneck
  - Except for specific performance-tests targeting it
- Speed influenced by:
  - The number of triangles
  - The number of vertex attributes to be rasterized
- To speed up:
  - Decrease ratio of degenerate to real triangles
  - But only if that ratio is substantial (> 1 to 5)

Rasterization bottlenecks

- It is the bottleneck if lots of large z-culled triangles
  - Rare
- Speed influenced by:
  - The number of triangles
  - The size of the triangles

GPU bottlenecks – fragment shader

- In past architectures, the fixed, then simply configurable nature of the shader made its performance match the rest of the pipeline pretty well
- In NV1X (DirectX 7), using more general combiners could reduce fragment shading performance, but often it was still not the bottleneck
- In NV2X (DirectX 8), more complex fragment shader modes introduced an even larger range of throughput in fragment shading
- NV3X (CineFX / DirectX 9) can run fragment shaders of 512 instructions (1024 in OpenGL)
  - Long fragment shaders create bottlenecks

GPU bottlenecks – fragment shader: Causes and solutions

- Too many fragments
  - Solution:
    - Draw in rough front-to-back order
    - Consider using a Z-only first pass
      - That way you only shade the visible fragments in subsequent passes
      - But: You also spend vertex throughput to improve fragment throughput
      - So: Don’t do this for fragments with a simple shader
      - Note that this can also help fb bandwidth

GPU bottlenecks – fragment shader: Causes and solutions

- **Too much computation per fragment**
  - **Solution:**
    - **Use fewer instructions by leveraging complex instructions, vector operations and co-issuing (RGB/Alpha)**
    - **Use a mix of texture and combiner instructions (they run in parallel)**
    - **Use an even number of combiner instructions**
    - **Use an even number of (simple) texture instructions**
    - **Use the alpha blender to help**
      - **SRCCOLOR*SRCALPHA for modulating in the dot3 result**
      - **SRCCOLOR*SRCCOLOR for a free squaring**
    - **Consider using shader level of detail**
      - **Turn off detail map computations in the distance**
    - **Consider moving per-fragment work to per-vertex**

CineFX fragment shader optimizations

- Additional guidance to maximize performance:
  - Use fp16 instructions whenever possible
    - Works great for traditional color blending
    - Use the _pp instruction modifier
  - Minimize temporary storage
    - Use 16-bit registers where applicable (most cases)
    - Reuse registers and use all components in each (swizzling is free)

GPU bottlenecks – texture: Causes and solutions

- **Textures are too big:**
  - **Overloaded texture cache: Lots of cache misses**
  - **Overloaded video memory: Textures are fetched from AGP memory**
  - **Solution:**
    - **Texture resolutions should be as big as needed and no bigger**
    - **Avoid expensive internal formats**
      - **CineFX allows 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, note that DXT1 quality is great on modern NV GPUs**

GPU bottlenecks – texture: Causes and solutions

- Texture cache is under-utilized: Lots of cache misses
  - Solution:
    - Localize texture access
      - Beware of dependent texture look-up
    - Use mipmapping:
      - Avoid negative LOD bias to sharpen: Texture caches are tuned for standard LODs
        - Sharpening usually causes aliasing in the distance
        - Prefer anisotropic filtering for sharpening
    - Beware of non-power of 2 textures
      - Often have worse caching behavior than power of 2

GPU bottlenecks – texture: Causes and solutions

- Too many samples per look-up
  - Trilinear filtering cuts fillrate in half
  - Anisotropic filtering can be even worse
    - Depending on level of anisotropy
    - The hardware is intelligent in this regard, you only pay for the anisotropy you use
  - Solution:
    - Use trilinear or anisotropic filtering only when needed:
      - Typically, only diffuse maps truly benefit
      - Light maps are too low resolution to benefit
      - Environment maps are distorted anyway
    - Reduce the maximum ratio of anisotropy
    - Often, using anisotropic reduces the need for trilinear

Fast Texture Uploads

- Use managed resources rather than your own scheme
  - Rely on the run-time and the driver *for most texturing needs*
- For truly dynamic textures:
  - Create with D3DUSAGE_DYNAMIC and D3DPOOL_DEFAULT
  - Lock them with D3DLOCK_DISCARD
  - Never read the texture!

GPU bottlenecks – frame buffer: Causes and solutions

- Too much read / write to the frame buffer:
  - Solution:
    - Turn off Z writes:
      - For subsequent passes of a multi-pass rendering scheme where you lay down Z in the first pass
      - For alpha-blended geometry (like particles)
    - But, do not mask off only some color channels:
      - It is actually slower because the GPU has to read the masked color channels from the frame buffer first before writing them again
    - Use alpha test (except when you mask off all colors)
    - Question the use of floating point frame buffers
      - These require much more bandwidth

GPU bottlenecks – frame buffer: Causes and solutions

- Solution (continued):
  - Use 16-bit Z depth if you don't use stencil
    - Many indoor scenes can get away with this just fine
  - 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

GPU bottlenecks – frame buffer: Causes and solutions

- Solution (continued):
  - Use hardware fast paths:
    - Buffer clears
      - Z buffer and stencil buffer are one buffer, so:
        - If you use the stencil buffer, clear the Z and stencil buffers together
        - If you don't use the stencil buffer, create Z-only depth surface (e.g. D24X8), otherwise it defeats Z clear optimizations
    - Z-cull is optimized for when Z-bias and alpha tests are turned off and stencil buffer is not used
  - Try using the new DirectX 9 constant color blend instead of a full-screen quad for tinting effects
    - D3DRS_BLENDFACTOR
    - Also standard in OpenGL 1.2

Conclusion

- Modern GPUs are programmable pipelines, as opposed to simply configurable, which means more potential bottlenecks, more complex tuning
- The goal is to keep each stage (including the CPU) busy creating interesting portions of the scene
- 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 inefficiencies to get things for free
  - Objects with expensive fragment shaders can often utilize expensive vertex shaders at little or no additional cost

Questions, comments, feedback?

- Cem Cebenoyan, cem@nvidia.com
- Juan Guardado, jguardado@nvidia.com
- Matthias Wloka, mwloka@nvidia.com
- Cyril Zeller, czeller@nvidia.com