# Compute Shaders

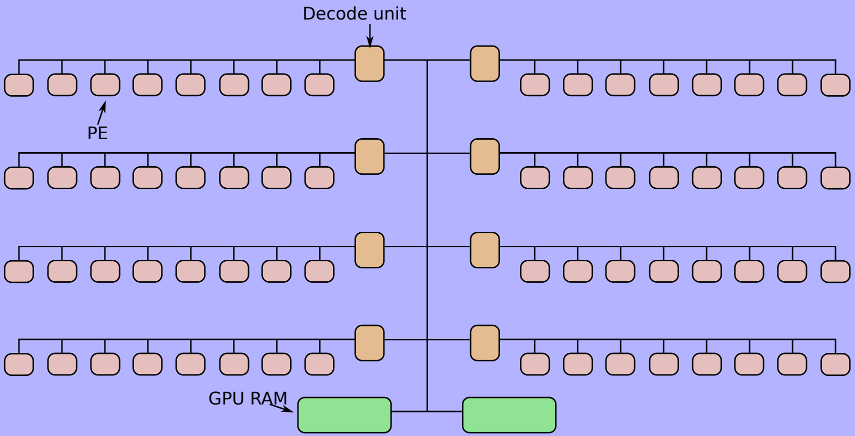
# Motivation

- You have a supercomputer inside of your computer: The GPU
  - GPU: `16,000,000,000,000` ops/sec (or more!) [nVidia Titan RTX GeForce 20 series: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units]
  - CPU: `74,000,000,000` ops/sec (Ryzen 7, 8 core)
- We want to take advantage of this
- But...Why use CPU's at all if GPU's so fast?

# Architecture

- Consider how instructions processed on typical CPU's: F,D,O,X,S
  - Instruction is fetched, then decoded, then operands are fetched, then it's executed, and finally, the results are stored
- Every clock tick, we advance the processing of an instruction down the pipeline
- So each instruction takes five cycles in this model
- Now consider GPU architecture...

# GPU Architecture



# Architecture

- To make circuitry easier to fabricate, a set of *processor elements* are grouped together
- What's a PE?
  - Circuits to do simple operations: Adder, multiplier, divider, bitwise ops
- PE can do arithmetic, but doesn't make decisions of *what* to do
  - Decode unit does that
- So all PE's of a group must do *same* instruction but with different data
  - SIMD: Like SSE, but writ large

# Quote

- “If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?” [Seymour Cray, early supercomputer pioneer: https://www.azquotes.com/author/23611-Seymour_Cray]
- CPU = Oxen, GPU PE = Chicken

# Upshot

- GPU works best if we have very regular operations, doing many data items at once
- Not so good if we have lots of independent operations, many decisions (“branchy”) code
  - Might have to execute *both sides* of the branch
  - Discard one set of results

# Setup

- Several different platforms available:
  - OpenCL
  - CUDA
  - DirectCompute/DX 12 Compute Shaders
  - OpenGL/Vulkan Compute Shaders
- We’ll use GL compute shaders here

# Prerequisites

- Download:
  - SDL 2: https://www.libsdl.org/download-2.0.php
  - Framework: framework.zip
- Unzip these somewhere

# Prerequisites

- Create a project file in your favorite IDE
  - Include directories: Add SDL's include folder
  - Library directories: Add SDL's library folder
  - Libraries: Add SDL2.lib and windowscodecs.lib
  - Add the framework code to your project
- When you run it, you should get a window with a GL rendering + FPS counter

# Program

- Right now, the program just draws an image to the screen using traditional rasterization techniques
- We'll use a compute shader to alter the rendered image
- But first, we must see what CS's look like

# Compute Shader

- Basic structure of CS:

```
layout(local_size_x=1,local_size_y=1,local_size_z=1) in;
void main(){
    uvec3 mynum = gl_GlobalInvocationID;
    ...do something...
}
```

# Invocations

- Since many graphics problems naturally have 2D or 3D structure, API includes built-in support for this
  - Allows us to organize our code to reflect logical structure of problem
- Each compute shader invocation has a "position" in a virtual "3D space"
- When we run CS, we must tell GPU how many invocations to do along each axis of this 3D space

# Dispatch

- Example: Suppose we've created a Program object with our compute shader code
- To use it:
  prog->use();
  prog->dispatch(100,1,1);
  - This just calls glDispatchCompute(100,1,1);
- Will run compute shader 100 times
  - mynum (in CS) will have values (0,0,0), (1,0,0), (2,0,0), ... , (99,0,0)

# Dispatch

- What if we do:
  prog->use();
  prog->dispatch(10,10,1);
- This will also run the compute shader 100 times
  - mynum will have values (0...9,0...9,0)

# Question

- What's the local_size text at the top of the compute shader?

# Workgroup

- GPU doesn't run one shader invocation at a time
  - Not efficient
  - Also, might want several invocations to communicate with each other
- Concept: Workgroup
  - Collection of several CS invocations
- Parameters to dispatch are really the number of *workgroups* to be executed
- Workgroups are defined in a 3D space as well

# Example

- Suppose workgroup size is (8,1,1)
- And we call dispatch(4,1,1)
- Total number of CS invocations: 32
  - mynum gets values (0,0,0)...(31,0,0)

# Example

- Or: If workgroup size is (2,4,8) and we dispatch(3,3,3)
- Total of (3*2)*(3*4)*(3*8) = 1,728 invocations
  - mynum.x ranges from 0...5
  - mynum.y ranges from 0...11
  - mynum.z ranges from 0...23

# Note

- Parameters passed to dispatch() must be ≤ 65535
- Workgroup size: x,y must be ≤ 1024; z must be ≤ 64
  - And WG x × WG y × WG z must be ≤ 1024
- If problem is larger: Call dispatch() several times
- Workgroups usually get mapped to *warps* or *wavefronts* in implementation-defined way

# Note

- If workgroup size is 1, entire warp is devoted to one thread execution
- Result: *Thread occupancy* is very low
- Let's look at an example now

# Example

- We'll need to alter the files "Globals.h", "setup.h", and "draw.h"
- The rest of the program can remain unchanged
- First task: Draw to a texture and then copy the texture to the screen

# Create

- In Globals.h: Add a variable for the FBO:
  std::shared_ptr<Framebuffer> fbo;
  - Remember from 2802: FBO allows us to render to a texture instead of the screen

# Create

- In setup(): Initialize the FBO
- This should be *after* setup creates the globs object:
  globs->fbo =
  std::make_shared<Framebuffer>(winwidth,winheight,1,GL_RGBA8);

# Use

- To use the FBO: We need to alter the draw function:
  - Rename the existing draw() function to "draw2()"
  - Add a new draw function:

```
void draw(){
    globs->fbo->setAsRenderTarget(false);
    draw2();
    globs->fbo->unsetAsRenderTarget();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glBindFramebuffer( GL_READ_FRAMEBUFFER, globs->fbo->fbo );
    glBindFramebuffer( GL_DRAW_FRAMEBUFFER, 0 );
    glBlitFramebuffer(0,0,globs->fbo->w,globs->fbo->h,
                      0,0,globs->fbo->w,globs->fbo->h,
                      GL_COLOR_BUFFER_BIT,
                      GL_NEAREST );
}
```

# Run It

- If you run the program, the output should look the same as before
  - On my system, it was slower: 859 fps (instead of 1017 fps)

# CS

- Now we're ready to incorporate our compute shader
- Add a global to Globals.h:
  Program cs{"cs.txt"};

# Draw

- Change draw to use the CS
- After the fbo->unsetRenderTarget() but before the glClear: Add some code:

```
globs->cs.use();
globs->fbo->texture->bindImage(0);
globs->cs.dispatch(globs->fbo->w, globs->fbo->h, 1 );
glMemoryBarrier( GL_ALL_BARRIER_BITS );
globs->fbo->texture->unbindImage(0);
```

# CS

- Finally, the compute shader itself: in shaders/cs.txt:

```
layout(local_size_x=1,local_size_y=1,local_size_z=1) in;
layout(binding=0,rgba8) uniform image2DArray img2;
void main(){
    ivec3 mynum = ivec3(gl_GlobalInvocationID);
    vec4 c = imageLoad( img2, mynum );
    if( length(c.rgb) < 0.1 ){
        c.rgb = vec3(0,1,0);
        imageStore(img2, mynum, c );
    }
}
```

# Explanation

- CS loads a single texel from the texture bound to image unit #0
- If it is close to black, we change the color to green
- Finally, we store the value

# Results

- On my machine, I get these results:
  - Original code: 859 frames per second
  - CS code: 92 frames per second
- Wow! Why so low?

# Problem

- Notice our workgroup size: We are using only one thread of every warp
  - I ran this on an nVidia card, so warp size is 32
  - On AMD, wavefront size is 64
- So only 1/32 = 3% of GPU being used
- What if we change the code...

# Changes

- Change first line of CS:
  `layout(local_size_x=64,local_size_y=1,local_size_z=1) in;`
- And change draw():
  `globs->cs.dispatch(globs->fbo->w/64, globs->fbo->h, 1 );`

# Results

- No CS: 859 frames per second
- CS, workgroup size=1: 92 frames per second
- CS, workgroup size=64: 727 frames per second
- That's better...But still only 84% of original results
  - Is it really *that* costly to change some pixel colors?

# Reasons

- Several things combine to make CS version of program slower
- How many operations can GPU do per second?
  - Switch render target (60,000 per sec)
  - Switch program (300,000/sec)
  - Switch pipeline state (glEnable/glDisable) (maybe 700,000/sec)
  - Switch textures (1,500,000/sec)
  - Switch VAO/UBO bindings (maybe 5,000,000/sec)
  - Update uniform (10,000,000/sec)
- Switching to/from CS can be more costly than just switching VS/FS programs

# Test

- How can we determine overhead of using CS?
- Can you think of a way (without looking at following slides?)

# Overhead

- One option: Comment out the csprog->dispatch() line in draw()
  - Result: 870 FPS
  - This is essentially the same as the no-cs result
    - It's actually faster!
    - Probably OS/scheduler overhead
- I suspect the GPU notices the CS is never dispatched and doesn't do any of the other setup work
  - Note: Explain concept of GPU ring buffer

# Overhead

- Next test: Put csprog->dispatch() back in and alter the shader itself
- Make first line be “return”
- Result: 835 FPS
- Analysis: What does this tell you?

# Analysis

- This tells us the cost of using the CS
- Again, the numbers:
  - No dispatch: 859 FPS
  - Dispatch, but CS returns immediately: 835 FPS
  - CS does real work: 727 FPS
- We pay some cost just to have a CS
- Most of the hit comes from the work the CS is doing

# The Birds and the Bees

- Or, where little pixels come from...

# SIMD

- Remember: We said GPU was SIMD
- Modern chipset manufacturers take this to the extreme
- Consider some example shader code...

# Shader

- Typical Lambertian shader:

```
layout(binding=0) uniform sampler2D tex;
uniform vec3 lightPosition;
in vec2 v_texCoord;
in vec3 v_normal;
in vec3 v_worldPos;
out vec4 color;
void main(){
    vec4 c = texture(tex, v_texCoord);
    vec3 L = normalize(lightPosition - v_worldPos);
    vec3 N = normalize(v_normal);
    float dp = dot(L,N);
    dp = max(0.0, dp);
    color.rgb = dp * c.rgb;
    color.a = 1.0;
}
```

- What does this look like at GPU instruction level?

# Code

- ```
  ;format: INSTRUCTION dest, args
  SUB tmp1.xyz, uniform[0].xyz, input[2].xyz ;lightPos - worldPos
  DOT tmp2.x, tmp1.xyz, tmp1.xyz             ;length squared
  RSQRT tmp2.x, tmp2.x                       ;one over length
  MUL tmp1.xyz, tmp2.xxx, tmp1.xyz           ;tmp1 gets unit L
  DOT tmp2.x, input[1].xyz, input[1].xyz     ;get sq. len of normal
  RSQRT tmp2.x, tmp2.x                       ;one over length
  MUL tmp2.xyz, tmp2.xxx, input[1].xyz       ;tmp2 holds unit N
  DOT tmp1.x, tmp1.xyz, tmp2.xyz             ;tmp1 gets dot(L,N)
  MAX tmp1.x, tmp1.x, 0                      ;tmp1 holds max(dot,0)
  TEXFETCH tmp0, 0, input[0]                 ;fetch from tex unit 0
  MUL output.xyz, tmp1.xxx, tmp0.xyz         ;dp * c.rgb
  MOV output.w, 1.0
  ```
- Notice: Compiler moved texture fetch as far down as it could. Why?
  ```

# GPU

- GPU starts executing shader code
- All threads in warp/wavefront proceed in tandem
  - Same instructions, but different data
- They're all busy...until they hit TEXFETCH
- This requires going out to memory
  - Up to this point, all data was in local registers

# Memory

- Memory access is slow
  - Even though it's local, dedicated RAM on GPU
  - Still requires arbitrating GPU bus, putting request out, waiting for response to come back
- Solution: GPU:
  - Queues up the TEXFETCH requests (one per thread in warp/wavefront)
  - Puts all 32 (64) threads on ice (suspends them)
  - Starts executing FS on brand new batch of 32 (64) pixels
- When those FS's hit TEXFETCH, put them on ice and repeat above process

# Result

- Eventually, GPU will run out of resources for suspending threads
  - The freezer gets full!
- So it pulls a batch out of suspended animation
  - Hopefully, results of TEXFETCH are available by now
- Begin executing
- When batch is finished, grab a new batch of pixels (or un-suspend a waiting group)

# Code

- This explains why compiler moved TEXFETCH down
- Want to make maximum possibility of overlapping work
- Imagine TEXFETCH was at top of code
  - We'd queue up the fetch, switch threads, queue another fetch, etc.
  - No overlap of memory fetch + computation
- Moral: CS's are best used if we have computation-heavy work to do

# CS

- Now we can return to our compute shader
  - Wasn't doing much computation
  - So most of the time, GPU is just queueing up memory read/write operations
- Why was this not a problem with FS?
  - CS's are more flexible than FS's
  - So GPU can make assumptions regarding what will and won't happen when it's doing FS's

# Detail

- One more detail: What's the glMemoryBarrier()?
  - Tells GPU that we need to make sure that all previous memory writes have completed before we go on to following code
- CS writes are *incoherent* by default
  - Not visible/published to rest of system until barrier is hit
    - Except for some frankly hard-to-remember exceptions...
- Best to put it in to make sure you get correct results

# Assignment

- Use the compute shader to compute a quarter size version of the original rendered scene.
- Draw the quarter-size version to the lower left quadrant of the window.
- The rest of the window should be black.

# Sources


- Khronos Corp. OpenGL reference pages & quick reference card.
- https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
- https://asteroidsathome.net/boinc/cpu_list.php
- https://www.khronos.org/opengl/wiki/Compute_Shader
- Cass Everitt and John McDonald. Beyond Porting. http://media.steampowered.com/apps/steamdevdays/slides/beyondporting.pdf (talk given in 2014)
- https://www.slideshare.net/CassEveritt/approaching-zero-driver-overhead
- https://www.khronos.org/opengl/wiki/ Shader_Storage_Buffer_Object
- https://www.khronos.org/opengl/wiki/Memory_Model#Incoherent_memory_access
- https://www.khronos.org/opengl/wiki/ GLAPI/glDeleteSync
- https://www.khronos.org/opengl/wiki/ Sync_Object
- https://www.khronos.org/opengl/wiki/Synchronization#Implicit_synchronization
- https://www.khronos.org/opengl/wiki/ GLAPI/glMapBufferRange
- https://www.khronos.org/opengl/wiki/ GLAPI/glBufferStorage
- https://www.khronos.org/opengl/wiki/ Buffer_Object_Streaming
- https://www.khronos.org/opengl/wiki/ Buffer_Object
- http://www.equasys.de/colorconversion.html
- https://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html
- John D. Cook. Converting color to grayscale. https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org