How does a GPU shader core work?

Unity Training Academy 2018-2019, #7
Aras Pranckevičius
Outline

- *All this will not be Unity specific!*
- Basically a rehash of “Running Code at a Teraflop” (Fatahalian’10)
- Basic ideas behind GPUs
- A look at some actual GPU designs
- Programming model implications
Gentle Intro to CPUs
What does a typical CPU do?

- A bunch of very simple operations, really fast
- 3GHz: 3 billion clock cycles every second
- Modern CPU core can do several operations per cycle
- The operations are simple though
- Many simple-looking constructs are lots of operations
Extremely common

- e.g. in JavaScript each and every member access is one
  - conceptually... JS engines do a lot of magic optimizations
- In C++, this oneliner:

```cpp
#include <unordered_map>
int get_from_dict(std::unordered_map<int, int>& dict, int val) {
    return dict[val];
}
```
hashtable lookup, in x64 assembly

- Via Compiler Explorer (godbolt.org), very useful site!

```assembly
get_from_dict:
    push rax
    mov dword ptr [rsp + 4], esi
    lea rsi, [rsp + 4]
    call std::__detail::_Map_base<...>::operator[](int const&)
    mov eax, dword ptr [rax]
    pop rcx
    ret

std::__detail::_Map_base<...>::operator[](int const&)
    push rbp
    push r15
    push r14
    push r13
    push r12
    push rbx
    push rax
    mov r13, rsi
    mov r14, rdi
    movsx r15, dword ptr [rsi]
    mov rcx, qword ptr [rdi]
    mov rdi, qword ptr [rdi + 8]
    xor edx, edx
    mov rax, r15
    div rdi

mov rbp, rdx
    mov rax, qword ptr [rcx + 8*rdx]
    test rax, rax
    je .LBB1_9
    mov rbx, qword ptr [rax]
    cmp r15d, dword ptr [rbx + 8]
    jne .LBB1_4
    test rbx, rbx
    jne .LBB1_3
    jmp .LBB1_9

.LBB1_4: # =>This Inner Loop Header: Depth=1
    mov rcx, rbx
    mov rbx, qword ptr [rbx]
    test rbx, rbx
    je .LBB1_9
    movsx rsi, dword ptr [rbx + 8]
    xor edx, edx
    mov rax, rsi
    div rdi
    cmp rdx, rbp
    jne .LBB1_9
    cmp r15d, esi
    jne .LBB1_4
    test rcx, rcx
```
hashtable lookup, in x64 assembly, ...

...that was like 180 instructions :/
Something simpler

• "Diffuse" lighting

```c
struct float3 { float x, y, z; }
inline float dot(float3 a, float3 b) { return a.x*b.x+a.y*b.y+a.z*b.z; }
inline float max(float a, float b) { return a > b ? a : b; }

float3 lightDir;
float3 diffuse(float3 normal, float3 albedo)
{
    float d = dot(lightDir, normal);
    d = max(d, 0.0f);
    albedo.x *= d; albedo.y *= d; albedo.z *= d;
    return albedo;
}
```
Something simpler

```c
struct float3 { float x, y, z; }
inline float dot(float3 a, float3 b) { return a.x * b.x + a.y * b.y + a.z * b.z;
inline float max(float a, float b) { return a > b ? a : b;
float3 lightDir;
float3 diffuse(float3 normal, float3 albedo)
{
  float d = dot(lightDir, normal);
  d = max(d, 0.0f);
  albedo.x *= d; albedo.y *= d; albedo.z *= d;
  return albedo;
}
```

```assembly
diffuse(float3, float3):  # @diffuse(float3,
  movsd xmm4, qword ptr [rip + lightDir]  #
  mulps xmm4, xmm0
  movaps xmm0, xmm4
  shufps xmm0, xmm4, 229 # xmm0 = xmm0[1,1].
  addss xmm0, xmm4
  mulss xmm1, dword ptr [rip + lightDir+8]
  addss xmm1, xmm0
  xorps xmm0, xmm0
  maxss xmm1, xmm0
  movaps xmm0, xmm1
  shufps xmm0, xmm1, 224 # xmm0 = xmm0[0,0].
  mulps xmm0, xmm2
  mulss xmm1, xmm3
  ret
```
Typical processor instructions

- **Loads/stores**: move data in & out of registers
- **Math** ("ALU"): add, multiply, shift, or, ...
- **Branches**: “if”, calls, jumps
- **Other instructions**: *not gonna talk about them today :)*
Turns out GPUs have processor(s) too!

```c
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord / iResolution.xy;

    if (uv.x > 0.5)
        fragColor = vec4(sin(uv.x*iTime)*0.5+0.5, uv.y, 1.0, 1.0);
    else
        fragColor = vec4(0.0);
}
```
Turns out GPUs have processor(s) too!

- Same “diffuse” lighting in HLSL, compiled to DX11 ps_5_0
- Via Shader Playground [shader-playground.timjones.io](https://shader-playground.timjones.io)

```hlsln
float3 lightDir;
float3 diffuse(
    float3 normal : T0,
    float3 albedo : T1) : SV_Target0
{
    float d = dot(lightDir, normal);
    d = max(d, 0.0f);
    albedo *= d;
    return albedo;
}
```
Inside every GPU there are (many) small CPUs

- They can do instructions (math, branches, load/store, ...)
- They have registers to hold values/variables
GPUs have a lot of other stuff

- Tons of non-CPU-like hardware for graphics:
  - Texture samplers,
  - Geometry setup engines,
  - Rasterizers,
  - Tessellators,
  - Blending / output merging,
  - *(new!)* Raytracing BVH traversers / triangle intersectors
  - ...
- All of that we’re **not** going to talk about today :)}
Part 1

Throughput Oriented Architecture
Throughput vs Latency

- **Throughput** ("bandwidth"): how much stuff can be done per unit of time.
- **Latency** ("ping"): how long does it take to do a thing.

Same latency!
Throughput vs Latency

- **Throughput** ("bandwidth"): how much stuff can be done per unit of time.
- **Latency** ("ping"): how long does it take to do a thing.

Same throughput!
What’s a GPU?

- A heterogeneous computing chip, highly tuned for graphics

Will mostly talk about these
Shader programming model

- Items (pixels, vertices, ...) are processed **independently**,
- **Many** items processed at once,
- But there is **no** explicit parallel programming model

```cpp
SamplerState smp;
Texture2D tex;
float3 lightDir;
float3 PSMain(float3 normal : T0, float2 uv : T1) : SV_Target0
{
    float3 albedo = tex.Sample(smp, uv).rgb;
    float d = dot(lightDir, normal);
    d = max(d, 0.0f);
    albedo *= d;
    return albedo;
}
```
Compiled shader

SamplerState smp;
Texture2D tex;
float3 lightDir;
float3 PSMain(float3 normal : T0, float2 uv : T1) : SV_Target0
{
    float3 albedo = tex.Sample(smp, uv).rgb;
    float d = dot(lightDir, normal);
    d = max(d, 0.0f);
    albedo *= d;
    return albedo;
}

sample r0.yzw, v1.xyxx, t0.wxyz, s0
dp3 r0.x, cb0[0].xyzx, v0.xyzx
max r0.x, r0.x, l(0.0)
mul o0.xyz, r0.xxxx, r0.yzwy
Compiled shader (scalar)

1 pixel input data (normal, UV)

1 pixel output data (color)

```
sample r0.yzw, v1.xyxx, t0.wxyz, s0
dp3 r0.x, cb0[0].xyzx, v0.xyzx
max r0.x, r0.x, l(0.0)
mul o0.xyz, r0.xxxx, r0.yzwy
```
Shader execution

```
sample r0...r2, v1.xy, t0, s0
mul r3, cb0[0], v0x
madd r3, cb0[1], v0y, r3
madd r3, cb0[2], v0z, r3
max r3, r3, l(0.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
```
Shader execution

Fetch/Decode

ALU (Execute)

Execution Context

```
sample r0…r2, v1.xy, t0, s0
mul  r3, cb0[0], v0x
madd r3, cb0[1], v0y, r3
madd r3, cb0[2], v0z, r3
max  r3, r3, 1(0.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
```
Shader execution

Fetch/ Decode

ALU (Execute)

Execution Context

Reg  Reg
Reg  Reg
Reg  Reg

sample r0...r2, v1.xy, t0, s0
mul  r3, cb0[0], v0x
madd r3, cb0[1], v0y, r3
madd r3, cb0[2], v0z, r3
max  r3, r3, 1(0.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
Shader execution

Fetch/Decode

ALU (Execute)

Execution Context

- Reg
- Reg
- Reg
- Reg

sample r0…r2, v1.xy, t0, s0
mul r3, cb0[0], v0x
madd r3, cb0[1], v0y, r3
madd r3, cb0[2], v0z, r3
max r3, r3, 1(0.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
Shader execution

Fetch/Decode

ALU (Execute)

Execution Context

```
sample r0...r2, v1.xy, t0, s0
mul r3, cb0[0], v0x
madd r3, cb0[1], v0y, r3
madd r3, cb0[2], v0z, r3
max r3, r3, l(0.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
```
CPU style cores

- Fetch/Decode
- ALU (Execute)
- Execution Context
  - Reg
  - Reg
  - Reg
  - Reg

Big data cache
- Memory prefetching
- Fancy branch prediction
- Out-of-order execution
CPU style cores

- Intel Skylake core

CPU style cores

- Intel Skylake core

Slim it down!

Idea #1:

**Remove** components that speed up running **one** thing at a time
Let’s have two cores (2 pixels at once)

```
sample r0...r2, v1.xy, t0, s0
mul r3, cb0[0], v0x
madd r3, cb0[1], v0y, r3
madd r3, cb0[2], v0z, r3
```
Four cores (4 pixels at once)

sample r0…r2, v1.xy, t0, t0
mul r3, cb0[0], v0x
add r3, cb0[1], v0y
add r3, cb0[2], v0z
Sixteen cores! (16 at once)
What if they all would execute the same code?

sample  r0...r2,  v1.xy,  t0, s0
mul     r3, cb0[0],  v0x
madd    r3, cb0[1],  v0y,  r3
madd    r3, cb0[2],  v0z,  r3
max     r3, r3,  l(0.0)
mul     o0,  r0,  r3
mul     o1,  r1,  r3
mul     o2,  r2,  r3
Recall: simple execution core

Fetch/Decode

ALU (Execute)

Execution Context

Reg Reg
Reg Reg
Reg Reg

Reg Reg

Reg Reg
Execute same code for many ALUs

Idea #2:
Use same fetch/decode for many ALUs.

SIMD processing
Modifying the shader

**Original shader:**
Processes **one** pixel using **scalar** ops on **scalar** registers

```
sample r0...r2, v1.xy, t0, s0
mul r3, cb0[0], v0x
madd r3, cb0[1], v0y, r3
madd r3, cb0[2], v0z, r3
max r3, r3, 1(0.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
```
Modifying the shader

New compiled shader:
Processes 8 pixels using 8-vector ops on 8-vector registers

```
v8sample vr0...vr2, v1.xy, t0, s0
v8mul  vr3, cb0[0], v0x
v8madd vr3, cb0[1], v0y, vr3
v8madd vr3, cb0[2], v0z, vr3
v8max  vr3, vr3, 1(0.0)
v8mul  vo0, vr0, vr3
v8mul  vo1, vr1, vr3
v8mul  vo2, vr2, vr3
```
Modifying the shader

```
Fetch/Decode

ALU1   ALU2   ALU3   ALU4
ALU5   ALU6   ALU7   ALU8

Ctx    Ctx    Ctx    Ctx
Ctx    Ctx    Ctx    Ctx

Shared Context Data

v8sample   vr0...vr2, v1.xy, t0, s0
v8mul       vr3, cb0[0], v0x
v8madd      vr3, cb0[1], v0y, vr3
v8madd      vr3, cb0[2], v0z, vr3
v8max       vr3, vr3, l(0.0)
v8mul       vo0, vr0, vr3
v8mul       vo1, vr1, vr3
v8mul       vo2, vr2, vr3
```
128 pixels in parallel

- 16 of such cores:
  - 128 pixels in parallel,
  - Each core works on 8 pixels,
  - Executing the same code on each.
128 [pixels, vertices, items, ...] in parallel

- Not only pixels:
  - Vertices
  - Primitives
  - DX Compute threads
  - OpenCL work items
  - CUDA threads
  - ...
Scale GPUs by varying # of cores

- “High”: 16 cores, \( \times 8 \) ALUs, \( = 128 \) items
- “Mid”: 8 cores, \( \times 8 \) ALUs, \( = 64 \) items
- “Low”: 4 cores, \( \times 8 \) ALUs, \( = 32 \) items
What about branches?

// regular code

if (x > 0.5) {
    y = sqrt(x); 
    c = y * foo;
} else {
    c = 0;
}

// regular code again
What about branches?

```c
// regular code
if (x > 0.5) {
    y = sqrt(x);
    c = y * foo;
} else {
    c = 0;
}
// regular code again
```
What about branches?

ALUs that don’t pass the condition get “masked out”, i.e. they don’t write the result.

```
// regular code
if (x > 0.5) {
    y = sqrt(x);
    c = y * foo;
} else {
    c = 0;
}
// regular code again
```
What about branches?

Not all ALUs do useful work!
Worst case: 1/8th performance.
This is one of reasons why “branching might be bad on GPU”
What about branches?

```c
// regular code
if (x > 0.5) {
  y = sqrt(x);
  c = y * foo;
} else {
  c = 0;
}
// regular code again
```
SIMD processing vs SIMD instructions

- Not necessarily the same!
- Most CPUs (Intel, ARM): explicit SIMD instructions
  - SSE, AVX, NEON
- Most GPUs: scalar instructions, vectorization mostly hidden
  - In practice, 16/32/64 items run the same instruction stream
  - NVIDIA calls them “warps”, AMD “wavefronts”
Uh-oh!

Stalls!
Stalls!

When an instruction can not run since results are not available yet

e.g. texture fetch latency - 100s to 1000s of clock cycles

We have removed all the fancy caches & OoO execution that helps with stalls on the CPU...
But we are computing lots of items at once...

*Like, really lots of them.*

**Idea #3:**

**Interleave** processing of **many** pixels on a **single** core, to avoid stalls due to high-latency operations.
## Hiding stalls

<table>
<thead>
<tr>
<th>Time (clocks)</th>
<th>Pixels 1..8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Fetch/Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU1</td>
</tr>
<tr>
<td>ALU5</td>
</tr>
</tbody>
</table>

### Pixels 1..8

- **1..8**

### Fetch/Decode

- ALU1
- ALU2
- ALU3
- ALU4
- ALU5
- ALU6
- ALU7
- ALU8
Hiding stalls

Time (clocks)

 Pixels 1..8

 Pixels 9..16

 Pixels 17..24

 Pixels 25..32

Fetch/Decode

ALU1  ALU2  ALU3  ALU4

ALU5  ALU6  ALU7  ALU8

unity
Hiding stalls

**Time (clocks)**

Pixels 1..8

Pixels 9..16

Pixels 17..24

Pixels 25..32

Runnable

Stall!

Fetch/Decode

ALU1   ALU2   ALU3   ALU4

ALU5   ALU6   ALU7   ALU8

1  2

3  4

Pixels 1..8

Pixels 9..16

Pixels 17..24

Pixels 25..32
Hiding stalls

Time (clocks)

Pixels 1..8  Pixels 9..16  Pixels 17..24  Pixels 25..32

Stall!  Stall!  Stall!  Stall!

Runnable  Runnable  Runnable  Runnable

Fetch/Decode

ALU1  ALU2  ALU3  ALU4
ALU5  ALU6  ALU7  ALU8
Hiding stalls

Time (clocks)

Pixels 1..8

Pixels 9..16

Pixels 17..24

Pixels 25..32

Fetch/Decode

ALU1 ALU2 ALU3 ALU4

ALU5 ALU6 ALU7 ALU8

1 2 3 4
Hiding stalls

Time (clocks)

Pixels 1..8

Pixels 9..16

Pixels 17..24

Pixels 25..32

Fetch/Decode

ALU1  ALU2  ALU3  ALU4

ALU5  ALU6  ALU7  ALU8
Context storage

Fetch/Decode

ALU1  ALU2  ALU3  ALU4
ALU5  ALU6  ALU7  ALU8

Space for context storage (e.g. 256KB)
Small contexts, can fit 12. Good latency hiding.

<table>
<thead>
<tr>
<th>Fetch/Decode</th>
<th>ALU1</th>
<th>ALU2</th>
<th>ALU3</th>
<th>ALU4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU5</td>
<td>ALU6</td>
<td>ALU7</td>
<td>ALU8</td>
<td></td>
</tr>
</tbody>
</table>

1  2  3  4  5  6
7  8  9 10 11 12
Medium contexts, can fit 6. Average latency hiding.
Big contexts, can fit 2. Bad latency hiding.
Making contexts smaller is important optimization

- Basically means “how many registers/space a shader might need”
  - i.e. max. amount of live temporary variables, etc.
  - More temporaries -> larger context -> worse latency hiding
- This is another reason for “branching might be bad”
  - Even if branch is not taken, register space (context size) still allocated!
Summary: 3 key ideas

- Slim down execution cores & put lots of them
- Put many ALUs into each core (execute same instruction stream)
- Avoid stalls by interleaving execution of many groups of items
Part 2

A brief look at actual GPUs
Caveat emptor

- I’m simplifying a lot
- And not mentioning a lot of other stuff:
  - Special instructions that aren’t simple mul/add/…
  - “Actually scalar” registers/instructions
  - Half-precision (FP16), integers
  - Caches
  - Instruction scheduling/dispatch details
NVIDIA GeForce 1080Ti (GP102 Pascal)

- NVIDIA speak:
  - 3584 ALUs ("CUDA cores")
  - 28 Cores (SMs - “Streaming Multiprocessors”)
  - Each SM is 128 ALUs, in a 4x32 configuration
  - 32 threads ("warp") share the same instruction stream
  - SM: 256KB register space, 96KB shared mem, 48KB L1 cache
  - 10.6 TFLOPS at 1.48GHz

https://db.thegpu.guru/card/GTX%201080%20Ti
GTX 1080 whitepaper
AMD RX Vega 64 (Vega 10)

- AMD speak:
  - 4096 ALUs (“Stream Processors”)
  - 64 Cores (NCUs - “Next-gen Compute Units”)
  - Each CU is 64 ALUs, in a 4x16 configuration
  - 64 threads (“wavefront”) share the same instruction stream
  - CU: 256KB register space, 64KB shared mem, 16KB L1 cache
  - 10.2 TFLOPS at 1.25GHz

https://db.thegpu.guru/card/RX%20Vega%2064

Vega 64 whitepaper
CPU, why not? AMD ThreadRipper 2990WX

- Handwaving it a lot, and ignoring everything about integers:
  - 32 Cores
  - Each Core has \(\sim 16\) FP ALUs (4x 128bit)
  - So we could say “512 ALUs” in total
  - 4 ALUs share the same instruction stream (SSE)
  - Core: 2.5KB FP registers, 96KB L1, 512KB L2 cache
  - \(\sim 1.5\) TFLOPS at 3GHz

WikiChip Zen+ info
Point is:

- Conceptually, the ideas here are all very similar
  - Marketing terms used are all different,
  - But they often talk about similar ideas / approaches
Part 3

Feeding the beast, and programming implications
CPU style:

- Optimized to **minimize latency**
  - Make **single** thread run as fast as possible
- **Big caches**
- RAM optimized for latency
  - DDR4 2400MHz - 14ns, 60GB/s
- Modern CPUs are awesome at running even “crap” code!
GPU style:

- Optimized to **maximize throughput**
  - Do lots of similar/same things at once
  - Small caches, big register files instead (*enables interleaving*)
  - RAM optimized for bandwidth
    - GDDR5X 1375MHz - ~50ns, 460GB/s
Feeding the beast

- High-end GPU has ~\textbf{40 times} math “power” than an 8-core CPU,
- But only ~\textbf{7 times} more memory bandwidth.

- Overcoming \textbf{bandwidth limits} is a common challenge for GPU development & optimization
  - Actually that often is true on CPUs — often the bottleneck is memory bandwidth/latency, not “computation”
Also, GPUs are different from CPUs

- AMD Vega 64 has total **16MB of registers**
- Each CU ("Core") in there can have 2048 “threads” in flight
  - 64 cores = GPU might be working on **131 thousand things**

> one of the most mindbending things about GPUs is the realization that your register file is much bigger than your cache and what this means

https://twitter.com/FioraAeterna/status/708164267938164736
How to reduce bandwidth requirements?

- Request data less often: “do more math” instead
- Reuse data more often
  - Implicit sharing via caches
  - Explicit sharing via local memory
  - Explicit sharing via wave operations
GPU caches

- Texture samplers have *(small)* caches
  - Neighboring pixels often read neighboring texels
  - This is why *not* having mipmaps is bad for perf
    - Far away textures start trashing the texel cache
- General memory loads have *(small)* caches
  - e.g. 48KB per SM on GTX 1080
Sharing data between ALUs/threads

Items (pixels, vertices, ...) are processed independently, but there is **no** explicit parallel programming model.

**LIES!**
GPU (compute) local memory

- “groupshared” in DirectX, “threadgroup” in Metal, ...
  - All threads in a “group” can access it,
  - Synchronization you must do yourself
    - Atomics, barriers, ...

- Similar latency hiding aspects as with register usage
  - e.g. 96KB shared memory per SM on GTX 1080
  - The more a thread group uses, the fewer groups can be “in flight”
Quad Operations

- Pixel shaders always execute in 2x2 quads
  - They need to read from textures,
  - And so they need to know which mipmap level to read from,
  - And that is done by knowing how “far away” neighbor pixel UVs are,
  - And that is computed by calculating their derivatives inside 2x2 block
- If we know this, we can (ab)use derivative instructions
  - And share some data between up to 4 pixel shader invocations
  - Some APIs make it explicit, e.g. Metal 2 has “quad operations” instructions
(fairly new!) Wave Operations

- If we have “vector” registers in a GPU core, executing same code...
- ...we could allow some “communication” within a vector, e.g.:
  - “set all lanes to the minimum value across whole vector”,
  - “swap every 2nd lane of the vector”,
  - ...etc.
- Only exposed very recently: DX12 SM6.0, Vulkan 1.1, Metal 2
  - Very useful list of resources: @KostasAAA twitter thread
Further reading & Resources

- “Running Code at a Teraflop: How a GPU Shader Core Works”, Kayvon Fatahalian 2010
- “A trip through the Graphics Pipeline”, Fabian Giesen 2011
- “Optimizing GPU occupancy and resource usage with large thread groups”, Seb Aaltonen 2017
- Scalarization & Wave Intrinsics Resources, twitter thread by Kostas Anagnostou 2018
- “Scheduling the Graphics Pipeline”, Jonathan Ragan-Kelley 2011
- “The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload”, Louis Bavoil 2018
- “Optimize your engine using compute”, Lou Kramer 2018
- “Life of a triangle - NVIDIA's logical pipeline”, Christoph Kubisch 2015
- “GPU-Driven Rendering”, Kubisch & Boudier 2016
Further reading & Resources

- Fabian Giesen has excellent in-depth video series on architecture (2018):
  - CPU microarchitecture [https://www.youtube.com/watch?v=oDrorJar0kM](https://www.youtube.com/watch?v=oDrorJar0kM)
  - Superscalar pipelines, stalls, interrupts [https://www.youtube.com/watch?v=rFZsP1Cy_HA](https://www.youtube.com/watch?v=rFZsP1Cy_HA)
  - SRAM arrays, register files, caches [https://www.youtube.com/watch?v=OKuKOUOJE7c](https://www.youtube.com/watch?v=OKuKOUOJE7c)
  - Microcode [https://www.youtube.com/watch?v=JpQ6QVgtyGE](https://www.youtube.com/watch?v=JpQ6QVgtyGE)
  - Many of things above applicable to GPUs, e.g. SRAM lecture lets you understand what “bank conflicts” are and why they are

- Compiler Explorer, Matt Godbolt
- Shader Playground, Tim Jones
Questions?

“Julia - Quaternion” by iq
https://www.shadertoy.com/view/MsfGRr