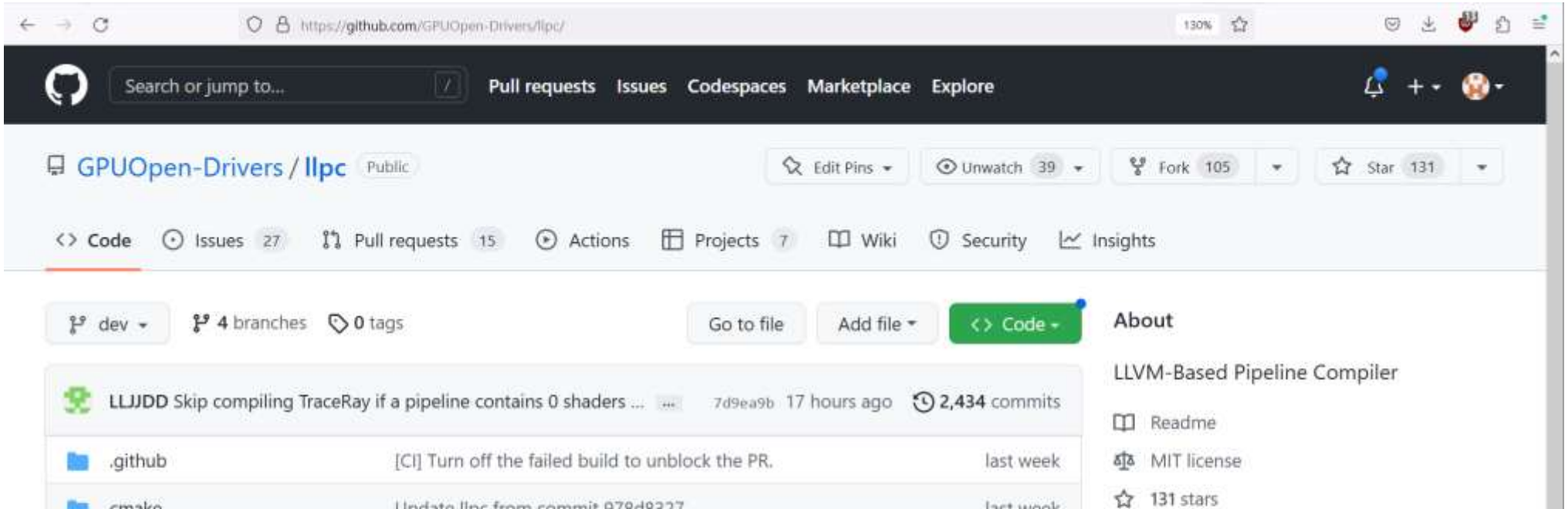
Code Generation for AMD GPUs

Nicolai Hähnle

AMD
together we advance_

About Me

- Got into open-source 3D driver development for the ATI R300 via reverse engineering
- PhD in mathematics at EPFL in Lausanne
- Research in (integer) linear optimization, combinatorial optimization, and electronic design automation
- At AMD in various roles since 2015
- Today: Architect on the Shader Compiler Team



Agenda

- The GPU programming ecosystem
- What is a GPU?
- Generating code for a GPU

The GPU Programming Ecosystem

A Graphics Programmer's View

- Write shaders in a high-level language: HLSL, GLSL, growing long tail
- Compile to an API-specific, vendor-agnostic representation: SPIR-V, DXIL, WGSL
- Pass this representation to the API at application runtime
  - vkCreateGraphicsPipelines() etc.
- Get an opaque handle to the resulting "pipeline"
  - Pipelines contain GPU binaries, but also:
  - Binding information: How to access "global" variables
  - Fixed function state: Vertex input state, pixel blending modes, etc.

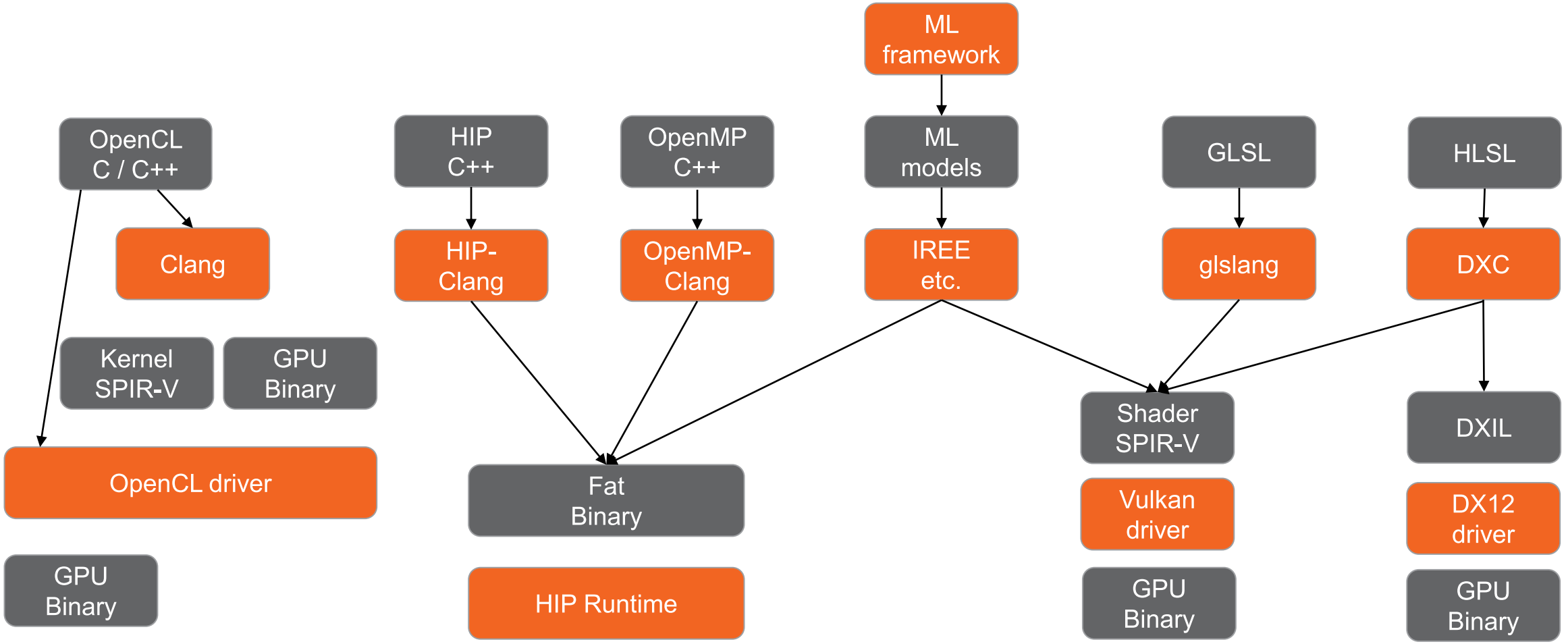
```
Buffer<float4> inputBuffer;
RWBuffer<float> outputBuffer;

[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    outputBuffer[did.x] = inputBuffer[did.x][did.y];
}
```

A Compute Programmer's View

- "Kernels" instead of "shaders"
- Details depend on the API / programming environment
- OpenCL
  - Separate source files, like graphics APIs
  - But kernels can optionally be precompiled to GPU binary
- Most other environments aim for single-source compilation and fat binaries
  - HIP/CUDA
    - Annotated functions are compiled as kernels under the hood by a (semi-custom) C++ compiler
    - Invoke kernels using magic "triple chevron" syntax or via a more traditional LaunchKernel runtime API function that accepts a (semi-magic) function pointer
  - OpenMP
    - #pragma omp on loops in C/C++/Fortran
    - Compiler splits code below function granularity and inserts code to invoke kernels
  - SYCL
  - C++AMP
  - etc.

```
__global__
void bodyForce(float4 *p, float4 *v, float dt, int n) {
  int i = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
  if (i < n) {
    float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

    for (int tile = 0; tile < hipGridDim_x; tile++) {
      __shared__ float3 spos[BLOCK_SIZE];
      float4 tpos = p[tile * hipBlockDim_x + hipThreadIdx_x];
      spos[hipThreadIdx_x] = make_float3(tpos.x, tpos.y, tpos.z);
      __syncthreads();

      for (int j = 0; j < BLOCK_SIZE; j++) {
        float dx = spos[j].x - p[i].x;
        float dy = spos[j].y - p[i].y;
        float dz = spos[j].z - p[i].z;
        float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
        float invDist = 1.0f / sqrtf(distSqr);
        float invDist3 = invDist * invDist * invDist;

        Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
      }
      __syncthreads();
    }

    v[i].x += dt*Fx; v[i].y += dt*Fy; v[i].z += dt*Fz;
  }
}
```

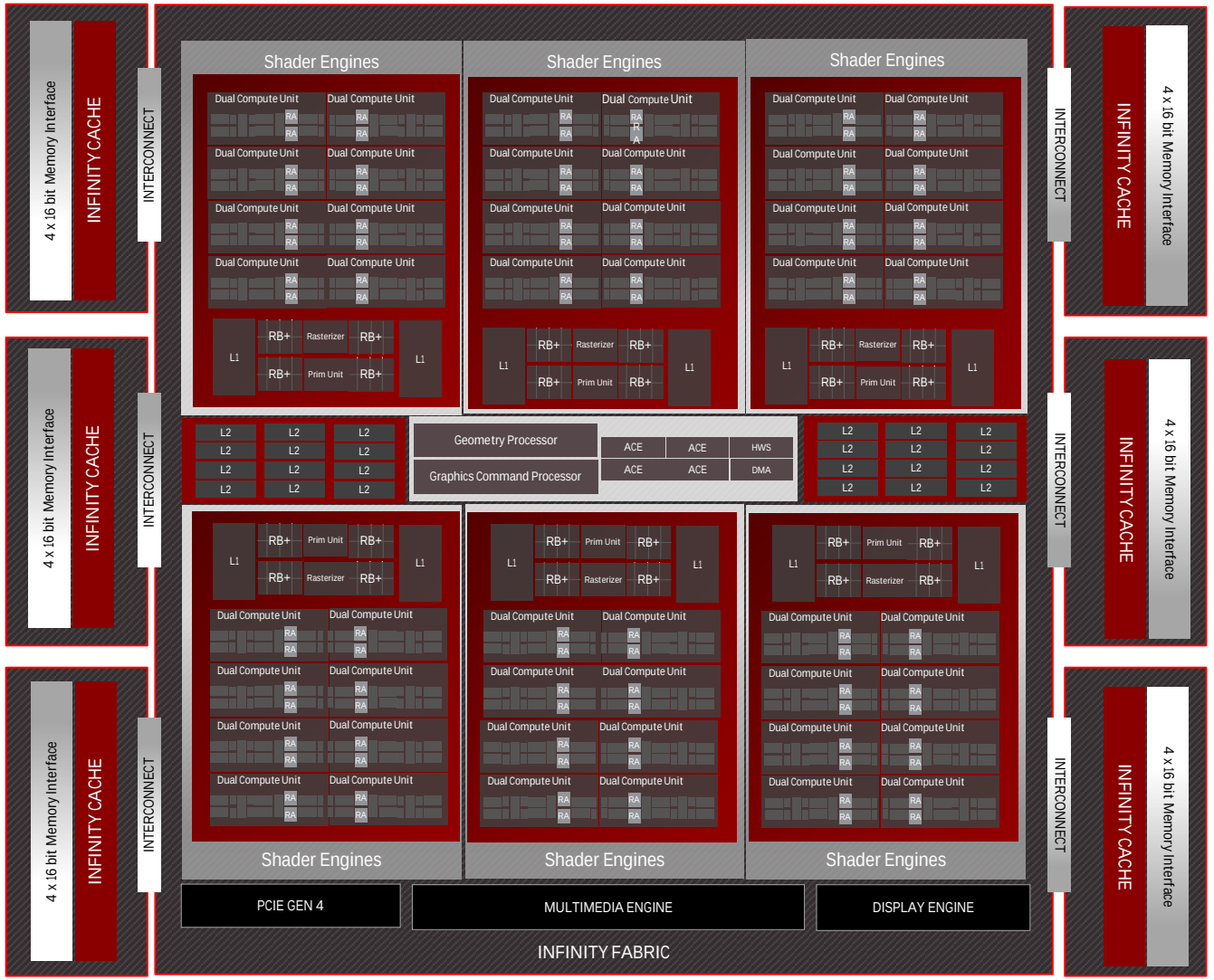
Some Paths to GPU binaries



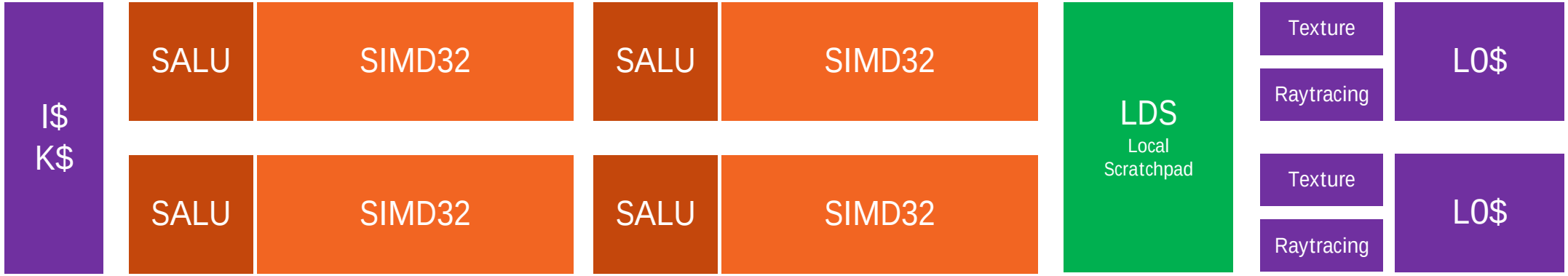
And many more…

What is a GPU?

High-level view



The RDNA Workgroup Processor



- Think of an RDNA WGP as a CPU with:
  - 4 cores: Each with a 32-wide SIMD unit
  - Each core supports deep SMT – up to 16 “threads” or “waves” per core
- Wave32 and Wave64 modes
  - Wave64 operates on 64-wide vectors
  - In the general case by issuing vector instructions twice (though there are significant optimizations)

RDNA ISA

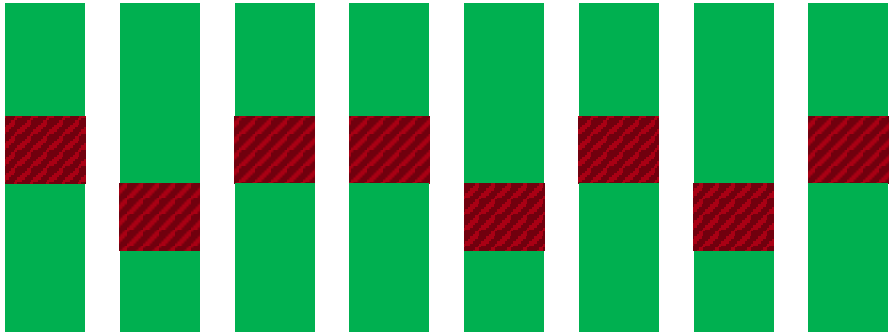
- ~106 32-bit scalar registers
- 256 vector registers
  - 32x32-bit or 64x32-bit depending on wave mode
- Register files are arrays
  - Successive registers can be combined to 64-bit and larger values
  - Some alignment requirements apply
  - Indirect indexing is possible
- Large set of scalar and vector ALU instructions
- Scalar branch instructions
- Full set of vector memory instructions
  - Full scatter/gather capabilities
  - Image format conversion and texture sampling
  - Raytracing acceleration
- Scalar loads for constant data

```
v_cmp_nle_f32    vcc, 0, v12
v_cndmask_b32    v19, -v20, v20, s[0:1]
v_add_f32        v20, |v16|, |v15|
v_cmp_le_f32     s[0:1], 0, v15
v_cndmask_b32    v13, v13, v18, vcc
v_mul_f32        v24, v10, v10
v_cndmask_b32    v14, v14, v19, vcc
v_sub_f32        v17, 1.0, v20
v_cndmask_b32    v18, -v21, v21, s[0:1]
```

```
    s_add_i32        s11, s11, 1
    ...
    s_cmpk_lg_i32    s28, 0x100
    s_cbranch_scc0   .LBB0_8
BB0_4:
    ...
    s_cmp_gt_u32     s11, 11
    ...
    s_cbranch_scc0   .LBB0_6
```

```
s_load_b128      s[20:23], s[24:25], 0x30
v_mul_f32        v7, s0, v1
v_mul_f32        v8, s1, v8
...
s_waitcnt        lgkmcnt(0)
image_sample_lz  v[9:10], v[7:8], s[4:11], s[20:23]
                 dmask:0x3 dim:SQ_RSRC_IMG_2D
```

SIMT Execution



```
void main()
{
    if (cb.material == 0) {
        color = texture (sampler2D (t0, s0), uv);
    } else {
        color = vec4(1,0,0,1);
    }
}
```

- “Threads” are mapped to lanes of a vector – the wave
- Program counter and instruction fetch, decode, and issue is per wave – not per lane
- Prefer terminology “wave / subgroup” and “lane / invocation” unless clear from the context
- In RDNA:
  - Every vector instruction is implicitly masked by the EXEC register
  - Control flow is implemented by scalar instructions operating on EXEC

```
    s_mov_b32       s5, exec_lo
    ...
    v_cmpx_eq_b32   0, v11
    s_cbranch_execz .LBB0_2
    < then block >
BB0_2:
    s_xor_b32       exec_lo, exec_lo, s5
    s_cbranch_execz .LBB0_3
    < else block >
BB0_3:
    s_or_b32        exec_lo, exec_lo, s5
```

Instruction Execution: CPU vs. GPU

	CPU	GPU
Branch prediction	✓	×
Speculative execution	✓	×
Out-of-order execution	✓	×
Register renaming	✓	×

Why?

Memory Accesses

- Memory instructions increment counters at issue
- Counters are decremented when instructions retire
- Different counters used for different instruction classes
- Some instruction classes remain in-order within the class, others can retire out of order
- Software must wait explicitly for those counters

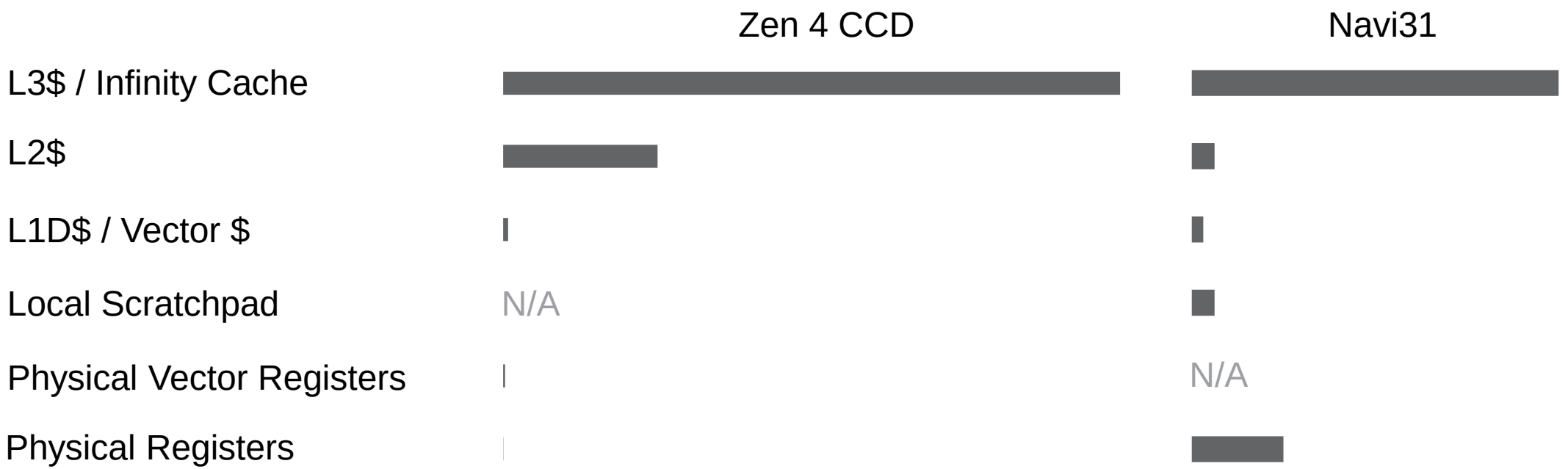
- Example:

s_waitcnt vmcnt(4)

Wait until the “vmcnt” counter is <= 4

```
s_load_b128       s[20:23], s[24:25], 0x30
v_mul_f32         v7, s0, v1
v_mul_f32         v8, s1, v8
...
s_waitcnt         lgkmcnt(0)
image_sample_lz   v[9:10], v[7:8], s[4:11], s[20:23]
                  dmask:0x3 dim:SQ_RSRC_IMG_2D
v_fma_f32         v11, 0.5, s16, v7
image_sample_lz   v[13:14], v[11:12], s[4:11], s[20:23]
                  dmask:0x3 dim:SQ_RSRC_IMG_2D
v_fmac_f32_e64    v8, s17, 0.5
image_sample_lz   v[15:16], v[7:8], s[4:11], s[20:23]
                  dmask:0x3 dim:SQ_RSRC_IMG_2D
v_mov_b32_e32     v12, v8
image_sample_lz   v[7:8], v[11:12], s[4:11], s[20:23]
                  dmask:0x3 dim:SQ_RSRC_IMG_2D
s_buffer_load_b128 s[20:23], s[12:15], 0x40
...
s_waitcnt lgkmcnt(0)
buffer_load_b64 v[0:1], v0, s[16:19], 0 offen offset:8
s_waitcnt vmcnt(4)
v_add_f32_e64 v11, |v10|, |v9|
```

Memory Hierarchy: CPU vs. GPU



- Diagram shows total storage sizes for unharvested configurations (all 8 cores / all 48 WGPs)
- CPU and GPU shown at different scales
- Why the qualitative difference? How does it affect code execution and code generation?

Generating Code for a GPU

Generating Code for a GPU – Selected Topics

- Implementing graphics APIs
- Satisfying the ISA: Wait insertion
- Control flow lowering
- Register allocation and memory instruction scheduling

Implementing graphics APIs

A Very Real Example

SPIR-V

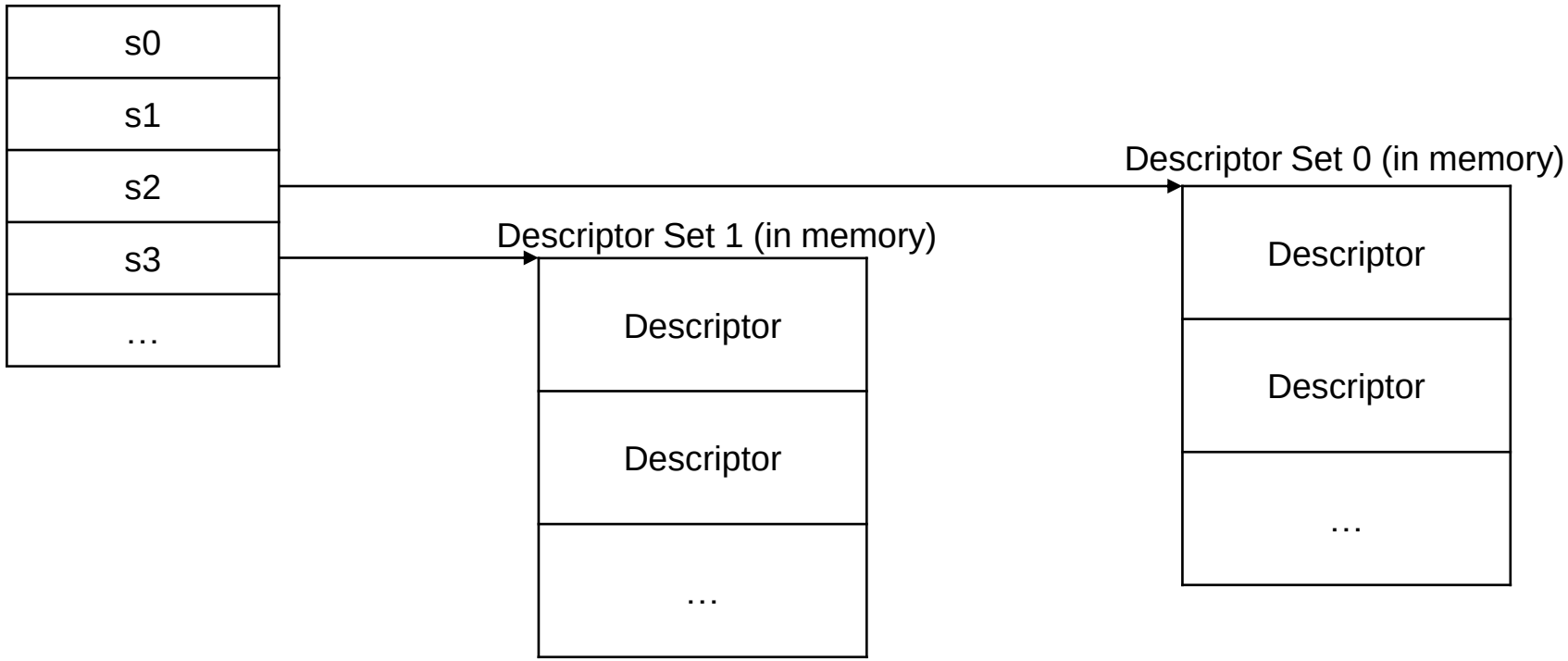
```
               OpDecorate %210 DescriptorSet 1
               OpDecorate %210 Binding 0

...

%_struct_208 = OpTypeStruct %v4float %v4float
%_ptr_Uniform__struct_208 = OpTypePointer Uniform %_struct_208
        %210 = OpVariable %_ptr_Uniform__struct_208 Uniform

...

        %213 = OpAccessChain %_ptr_Uniform_v4float %210 %int_0
        %214 = OpLoad %v4float %213
        %215 = OpVectorShuffle %v2float %214 %214 2 3
        %216 = OpFMul %v2float %207 %215
```


RDNA ISA

```
s_getpc_b64 s[6:7]
s_mov_b32 s24, s3
s_mov_b32 s25, s7
...
s_load_b128 s[16:19], s[24:25], 0x0
...
s_waitcnt lgkmcnt(0)
s_buffer_load_b64 s[0:1], s[16:19], 0x8
s_waitcnt lgkmcnt(0)
v_mul_f32_e32 v2, s0, v1
v_mul_f32_e32 v3, s1, v3

...

amdpal.pipelines: [ {
    ...
    .registers: {
        ...
        COMPUTE_USER_DATA_0                    0x0000000010000000
        COMPUTE_USER_DATA_2                    0x0000000000000000
        COMPUTE_USER_DATA_3                    0x0000000000000001 }
```

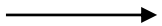
Shader ABI

- Some registers are initialized by hardware when a wave is launched
- The compiler and driver collaborate to configure this initialization



Trip through the compiler pipeline

```
               OpDecorate %210 DescriptorSet 1
               OpDecorate %210 Binding 0

...

%_struct_208 = OpTypeStruct %v4float %v4float
%_ptr_Uniform__struct_208 = OpTypePointer Uniform %_struct_208
        %210 = OpVariable %_ptr_Uniform__struct_208 Uniform

...

        %213 = OpAccessChain %_ptr_Uniform_v4float %210 %int_0
        %214 = OpLoad %v4float %213
        %215 = OpVectorShuffle %v2float %214 %214 2 3
        %216 = OpFMul %v2float %207 %215
```



```
@1 = external addrspace(7) constant <{ [4 x float], [4 x float] }>,
     !spirv.Resource !1, !spirv.Block !2

...

  %59 = load <4 x float>, ptr addrspace(7) @1, align 16
  %60 = shufflevector <4 x float> %59, <4 x float> %59, <2 x i32> <i32 2, i32 3>
  %61 = fmul reassoc nnan nsz arcp contract afn <2 x float> %58, %60

...

!1 = !{i32 1, i32 0, i32 0}
!2 = !{{ i64, { i64, i64 } }
       { i64 70368744177664,
         { i64, i64 } { i64 8796093022208, i64 8796093022224 } }}
```

```
/// Metadata for shader block.
union ShaderBlockMetadata {
  struct {
    unsigned offset : 32;      // Offset (bytes) in block
    unsigned IsMatrix : 1;     // Whether it is a matrix
    unsigned IsRowMajor : 1;   // Whether it is a "row_major" qualified matrix
    unsigned MatrixStride : 6; // Matrix stride, valid for matrix
    unsigned Restrict : 1;     // Whether "restrict" qualifier is present
    unsigned Coherent : 1;     // Whether "coherent" qualifier is present
    unsigned Volatile : 1;     // Whether "volatile" qualifier is present
    unsigned NonWritable : 1;  // Whether "readonly" qualifier is present
    unsigned NonReadable : 1;  // Whether "writeonly" qualifier is present
    unsigned IsPointer : 1;    // Whether it is a pointer
    unsigned IsStruct : 1;     // Whether it is a structure
    unsigned Unused : 17;
  };
  uint64_t U64All;
};
```

Trip through the compiler pipeline

```
@1 = external addrspace(7) constant <{ [4 x float], [4 x float] }>,
     !spirv.Resource !1, !spirv.Block !2

...

  %59 = load <4 x float>, ptr addrspace(7) @1, align 16
  %60 = shufflevector <4 x float> %59, <4 x float> %59, <2 x i32> <i32 2, i32 3>
  %61 = fmul reassoc nnan nsz arcp contract afn <2 x float> %58, %60

...

!1 = !{i32 1, i32 0, i32 0}
!2 = !{{ i64, { i64, i64 } }
       { i64 70368744177664,
         { i64, i64 } { i64 8796093022208, i64 8796093022224 } }}
```

```
%72 = call ptr addrspace(7) (...) @lgc.create.load.buffer.desc.p7(i32 1, i32 0, i32 0, i32 0)
%73 = call ptr @llvm.invariant.start.p7(i64 -1, ptr addrspace(7) %72)
...
%81 = load <4 x float>, ptr addrspace(7) %72, align 16
%82 = shufflevector <4 x float> %81, <4 x float> %81, <2 x i32> <i32 2, i32 3>
%83 = fmul reassoc nnan nsz arcp contract afn <2 x float> %80, %82
```

Trip through the compiler pipeline

```
%72 = call ptr addrspace(7) (...) @lgc.create.load.buffer.desc.p7(i32 1, i32 0, i32 0, i32 0)
%73 = call ptr @llvm.invariant.start.p7(i64 -1, ptr addrspace(7) %72)
...
%81 = load <4 x float>, ptr addrspace(7) %72, align 16
%82 = shufflevector <4 x float> %81, <4 x float> %81, <2 x i32> <i32 2, i32 3>
%83 = fmul reassoc nnan nsz arcp contract afn <2 x float> %80, %82
```

```
  %10 = call ptr addrspace(4) @lgc.descriptor.table.addr(i32 6, i32 6, i32 1, i32 0, i32 -1) #3
  %11 = getelementptr i8, ptr addrspace(4) %10, i32 0
  %12 = load <4 x i32>, ptr addrspace(4) %11, align 16
  %13 = call ptr addrspace(7) @lgc.late.launder.fat.pointer(<4 x i32> %12) #3
  ...
  %31 = load <4 x float>, ptr addrspace(7) %13, align 16
  %32 = shufflevector <4 x float> %31, <4 x float> poison, <2 x i32> <i32 2, i32 3>
  %33 = fmul reassoc nnan nsz arcp contract afn <2 x float> %30, %32

...

!lgc.user.data.nodes = !{!4, !5, !6, !7, !8, !9, !10, !11, !12, !13, !14, !15, !16, !17, !18}

!4 = !{!"DescriptorTableVaPtr", i32 7, i32 0, i32 1, i32 4}
...
!9 = !{!"DescriptorTableVaPtr", i32 7, i32 1, i32 1, i32 5}
!10 = !{!"DescriptorBuffer", i32 6, i32 0, i32 4, i32 1, i32 0, i32 4}
...
```

Trip through the compiler pipeline

```
%10 = call ptr addrspace(4) @lgc.descriptor.table.addr(i32 6, i32 6, i32 1, i32 0, i32 -1) #3
%11 = getelementptr i8, ptr addrspace(4) %10, i32 0
%12 = load <4 x i32>, ptr addrspace(4) %11, align 16
%13 = call ptr addrspace(7) @lgc.late.launder.fat.pointer(<4 x i32> %12) #3
...
%31 = load <4 x float>, ptr addrspace(7) %13, align 16
%32 = shufflevector <4 x float> %31, <4 x float> poison, <2 x i32> <i32 2, i32 3>
%33 = fmul reassoc nnan nsz arcp contract afn <2 x float> %30, %32
```

```
define dllexport amdgpu_cs void @lgc.shader.CS.main(
    i32 inreg %globalTable, i32 inreg %perShaderTable,
    i32 inreg %descTable0, i32 inreg %descTable1, ...) #0 !lgc.shaderstage !20 {
.entry:
  %0 = call i64 @llvm.amdgcn.s.getpc()
  %1 = bitcast i64 %0 to <2 x i32>
  %2 = insertelement <2 x i32> %1, i32 %descTable1, i64 0
  %3 = bitcast <2 x i32> %2 to i64
  %4 = inttoptr i64 %3 to ptr addrspace(4)
  ...
  %16 = getelementptr i8, ptr addrspace(4) %4, i32 0
  %17 = load <4 x i32>, ptr addrspace(4) %16, align 16
  %18 = call ptr addrspace(7) @lgc.late.launder.fat.pointer(<4 x i32> %17) #3
  %19 = call ptr @llvm.invariant.start.p7(i64 -1, ptr addrspace(7) %18)
  ...
  %34 = load <4 x float>, ptr addrspace(7) %18, align 16
  %35 = shufflevector <4 x float> %34, <4 x float> poison, <2 x i32> <i32 2, i32 3>
  %36 = fmul reassoc nnan nsz arcp contract afn <2 x float> %33, %35
```

Trip through the compiler pipeline

```
define dllexport amdgpu_cs void @lgc.shader.CS.main(
    i32 inreg %globalTable, i32 inreg %perShaderTable,
    i32 inreg %descTable0, i32 inreg %descTable1, ...) #0 !lgc.shaderstage !20 {
.entry:
  %0 = call i64 @llvm.amdgcn.s.getpc()
  %1 = bitcast i64 %0 to <2 x i32>
  %2 = insertelement <2 x i32> %1, i32 %descTable1, i64 0
  %3 = bitcast <2 x i32> %2 to i64
  %4 = inttoptr i64 %3 to ptr addrspace(4)
  ...
  %16 = getelementptr i8, ptr addrspace(4) %4, i32 0
  %17 = load <4 x i32>, ptr addrspace(4) %16, align 16
  %18 = call ptr addrspace(7) @lgc.late.launder.fat.pointer(<4 x i32> %17) #3
  %19 = call ptr @llvm.invariant.start.p7(i64 -1, ptr addrspace(7) %18)
  ...
  %34 = load <4 x float>, ptr addrspace(7) %18, align 16
  %35 = shufflevector <4 x float> %34, <4 x float> poison, <2 x i32> <i32 2, i32 3>
  %36 = fmul reassoc nnan nsz arcp contract afn <2 x float> %33, %35
```

```
%0 = call i64 @llvm.amdgcn.s.getpc()
%extelt.offset = lshr i64 %0, 32
%.i1 = trunc i64 %extelt.offset to i32
%.upto0 = insertelement <2 x i32> poison, i32 %descTable1, i64 0
%1 = insertelement <2 x i32> %.upto0, i32 %.i1, i64 1
%2 = bitcast <2 x i32> %1 to i64
%3 = inttoptr i64 %2 to ptr addrspace(4)
...
%10 = load <4 x i32>, ptr addrspace(4) %3, align 16
...
%14 = call <4 x i32> @llvm.amdgcn.s.buffer.load.v4i32(<4 x i32> %10, i32 0, i32 0), !invariant.load !21
%15 = bitcast <4 x i32> %14 to <4 x float>
%.i223 = extractelement <4 x float> %15, i64 2
%.i3 = extractelement <4 x float> %15, i64 3
%.i024 = fmul reassoc nnan nsz arcp contract afn float %.i223, %.i021
%.i125 = fmul reassoc nnan nsz arcp contract afn float %.i3, %.i122
```

Trip through the compiler pipeline

```
%0 = call i64 @llvm.amdgcn.s.getpc()
%extelt.offset = lshr i64 %0, 32
%.i1 = trunc i64 %extelt.offset to i32
%.upto0 = insertelement <2 x i32> poison, i32 %descTable1, i64 0
%1 = insertelement <2 x i32> %.upto0, i32 %.i1, i64 1
%2 = bitcast <2 x i32> %1 to i64
%3 = inttoptr i64 %2 to ptr addrspace(4)
...
%10 = load <4 x i32>, ptr addrspace(4) %3, align 16
...
%14 = call <4 x i32> @llvm.amdgcn.s.buffer.load.v4i32(<4 x i32> %10, i32 0, i32 0), !invariant.load !21
%15 = bitcast <4 x i32> %14 to <4 x float>
%.i223 = extractelement <4 x float> %15, i64 2
%.i3 = extractelement <4 x float> %15, i64 3
%.i024 = fmul reassoc nnan nsz arcp contract afn float %.i223, %.i021
%.i125 = fmul reassoc nnan nsz arcp contract afn float %.i3, %.i122
```

```
%0 = call i64 @llvm.amdgcn.s.getpc()
%extelt.offset = lshr i64 %0, 32
%.i1 = trunc i64 %extelt.offset to i32
%.upto0 = insertelement <2 x i32> poison, i32 %descTable1, i64 0
%1 = insertelement <2 x i32> %.upto0, i32 %.i1, i64 1
%2 = bitcast <2 x i32> %1 to i64
%3 = inttoptr i64 %2 to ptr addrspace(4), !amdgpu.uniform !5
...
%7 = load <4 x i32>, ptr addrspace(4) %3, align 16
...
%11 = call <2 x i32> @llvm.amdgcn.s.buffer.load.v2i32(<4 x i32> %7, i32 8, i32 0), !invariant.load !5
%12 = shufflevector <2 x i32> %11, <2 x i32> poison, <4 x i32> <i32 undef, i32 undef, i32 0, i32 1>
%13 = bitcast <4 x i32> %12 to <4 x float>
%.i223 = extractelement <4 x float> %13, i64 2
%.i3 = extractelement <4 x float> %13, i64 3
%.i024 = fmul reassoc nnan nsz arcp contract afn float %.i223, %.i021
%.i125 = fmul reassoc nnan nsz arcp contract afn float %.i3, %.i122
```

Trip through the compiler pipeline

```
%0 = call i64 @llvm.amdgcn.s.getpc()
%extelt.offset = lshr i64 %0, 32
%.i1 = trunc i64 %extelt.offset to i32
%.upto0 = insertelement <2 x i32> poison, i32 %descTable1, i64 0
%1 = insertelement <2 x i32> %.upto0, i32 %.i1, i64 1
%2 = bitcast <2 x i32> %1 to i64
%3 = inttoptr i64 %2 to ptr addrspace(4), !amdgpu.uniform !5
...
%7 = load <4 x i32>, ptr addrspace(4) %3, align 16
...
%11 = call <2 x i32> @llvm.amdgcn.s.buffer.load.v2i32(<4 x i32> %7, i32 8, i32 0), !invariant.load !5
%12 = shufflevector <2 x i32> %11, <2 x i32> poison, <4 x i32> <i32 undef, i32 undef, i32 0, i32 1>
%13 = bitcast <4 x i32> %12 to <4 x float>
%.i223 = extractelement <4 x float> %13, i64 2
%.i3 = extractelement <4 x float> %13, i64 3
%.i024 = fmul reassoc nnan nsz arcp contract afn float %.i223, %.i021
%.i125 = fmul reassoc nnan nsz arcp contract afn float %.i3, %.i122
```

```
%59:sgpr_32 = COPY $sgpr3
...
%65:sreg_64 = S_GETPC_B64
%66:sreg_32 = COPY %65.sub1:sreg_64
%4:sgpr_128 = S_LOAD_DWORDX4_IMM %67:sgpr_64, 0, 0 :: (invariant load (s128) from %ir.3, addrspace 4)
...
%76:sreg_64_xexec = S_BUFFER_LOAD_DWORDX2_IMM %4:sgpr_128, 8, 0 :: (dereferenceable invariant load (s64))
%10:sgpr_32 = COPY %76.sub1:sreg_64_xexec
%9:sgpr_32 = COPY %76.sub0:sreg_64_xexec
%11:vgpr_32 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e64 0, %9:sgpr_32, 0, killed %74:vgpr_32, 0, 0, implicit $mode, implicit $exec
%12:vgpr_32 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e64 0, %10:sgpr_32, 0, killed %75:vgpr_32, 0, 0, implicit $mode, implicit $exec
```

Trip through the compiler pipeline

```
%59:sgpr_32 = COPY $sgpr3
...
%65:sreg_64 = S_GETPC_B64
%66:sreg_32 = COPY %65.sub1:sreg_64
%4:sgpr_128 = S_LOAD_DWORDX4_IMM %67:sgpr_64, 0, 0 :: (invariant load (s128) from %ir.3, addrspace 4)
...
%76:sreg_64_xexec = S_BUFFER_LOAD_DWORDX2_IMM %4:sgpr_128, 8, 0 :: (dereferenceable invariant load (s64))
%10:sgpr_32 = COPY %76.sub1:sreg_64_xexec
%9:sgpr_32 = COPY %76.sub0:sreg_64_xexec
%11:vgpr_32 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e64 0, %9:sgpr_32, 0, killed %74:vgpr_32, 0, 0, implicit $mode, implicit $exec
%12:vgpr_32 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e64 0, %10:sgpr_32, 0, killed %75:vgpr_32, 0, 0, implicit $mode, implicit $exec
```



```
renamable $sgpr6_sgpr7 = S_GETPC_B64
$sgpr24 = S_MOV_B32 killed $sgpr3
$sgpr25 = S_MOV_B32 $sgpr7
...
renamable $sgpr16_sgpr17_sgpr18_sgpr19 = S_LOAD_DWORDX4_IMM renamable $sgpr24_sgpr25, 0, 0 :: (invariant load (s128) from %ir.3, addrspace 4)
...
renamable $sgpr0_sgpr1 = S_BUFFER_LOAD_DWORDX2_IMM renamable $sgpr16_sgpr17_sgpr18_sgpr19, 8, 0 :: (dereferenceable invariant load (s64))
renamable $vgpr2 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e32 $sgpr0, killed $vgpr1, implicit $mode, implicit $exec
renamable $vgpr3 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e32 $sgpr1, killed $vgpr3, implicit $mode, implicit $exec
```

Trip through the compiler pipeline

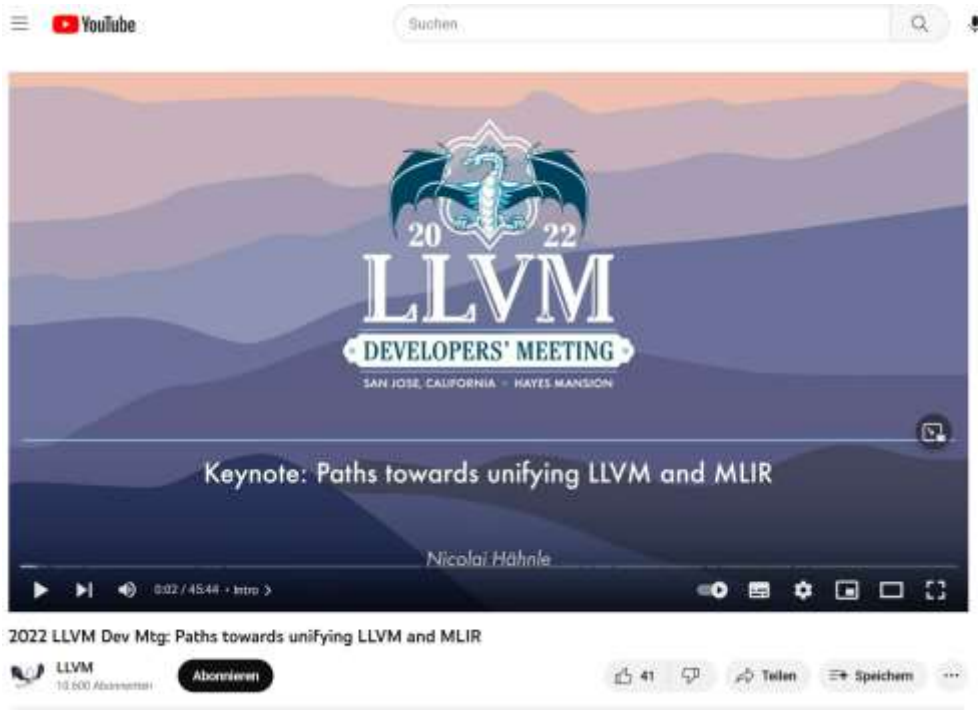
```
renamable $sgpr6_sgpr7 = S_GETPC_B64
$sgpr24 = S_MOV_B32 killed $sgpr3
$sgpr25 = S_MOV_B32 $sgpr7
...
renamable $sgpr16_sgpr17_sgpr18_sgpr19 = S_LOAD_DWORDX4_IMM renamable $sgpr24_sgpr25, 0, 0 :: (invariant load (s128) from %ir.3, addrspace 4)
...
renamable $sgpr0_sgpr1 = S_BUFFER_LOAD_DWORDX2_IMM renamable $sgpr16_sgpr17_sgpr18_sgpr19, 8, 0 :: (dereferenceable invariant load (s64))
renamable $vgpr2 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e32 $sgpr0, killed $vgpr1, implicit $mode, implicit $exec
renamable $vgpr3 = nnan nsz arcp contract afn reassoc nofpexcept V_MUL_F32_e32 $sgpr1, killed $vgpr3, implicit $mode, implicit $exec
```



```
s_getpc_b64 s[6:7]
s_mov_b32 s24, s3
s_mov_b32 s25, s7
...
s_load_b128 s[16:19], s[24:25], 0x0
...
s_waitcnt lgkmcnt(0)
s_buffer_load_b64 s[0:1], s[16:19], 0x8
s_waitcnt lgkmcnt(0)
v_mul_f32_e32 v2, s0, v1
v_mul_f32_e32 v3, s1, v3
```

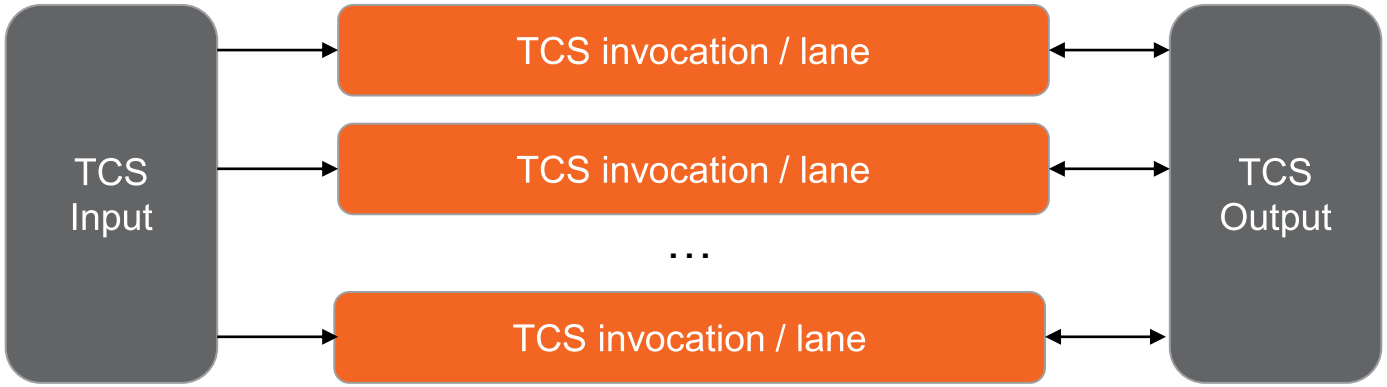
A Word on Intermediate Representations

- Our compiler uses a kind of “LLVM+X”
  - LLVM IR with additional “intrinsics” and metadata for GPU-specific operations
- Designing an IR is an art
  - A lot of the compiler flow is concerned with lowering these operations away
  - The IR must fit the lowering flow and vice versa
  - It must also serve other transforms (“optimizations”!)
- Implementing an IR is an art
  - Ergonomics matter
  - Compile-time matters
- MLIR is a great innovation in this area
  - Do the “IR implementation” only once (the “substrate”)
  - Developers can focus on designing purpose-built IRs (the “dialects”)
  - But MLIR doesn’t get everything right (yet?) – and is not compatible with LLVM…



Tessellation Control Shader Outputs

- Implementing graphics APIs is often a conceptually straightforward lowering process
- TCS outputs are a simple example of an exception



- Multiple TCS invocations can be launched for the same “patch” (triangle / quad)
- TCS invocations can (but often don’t) communicate via TCS outputs

TCS Output Lowering

- TCS outputs must ultimately be written to memory, from where they are read by the next pipeline stage
- We could just translate TCS output accesses to memory accesses

- Challenge: If TCS outputs are read back from TCS, reading them from memory is quite slow
- Solution:
  - TCS outputs go to LDS (local scratchpad)
  - At the end of TCS, all outputs are read back from LDS and then written to memory

- **But:** Should we always do this, or only if an output is read from?
  - Ordering between invocations still matters if there are aliased stores
  - Consider cacheline effects

Wait Insertion

Wait Insertion

- Late compiler pass to insert s_waitcnt instructions required for correctness
  - Why a late pass?
  - What does this mean for IR design?
- Algorithmically:
  - On-the-fly computation of “pending” registers and associated counter values within basic blocks
  - Fix point iteration for finding “pending” state at basic block boundaries
  - (Why) Do we need a fix point iteration?

```
s_load_b128      s[20:23], s[24:25], 0x30
v_mul_f32        v7, s0, v1
v_mul_f32        v8, s1, v8
...
s_waitcnt        lgkmcnt(0)
image_sample_lz  v[9:10], v[7:8], s[4:11], s[20:23]
                 dmask:0x3 dim:SQ_RSRC_IMG_2D
v_fma_f32        v11, 0.5, s16, v7
image_sample_lz  v[13:14], v[11:12], s[4:11], s[20:23]
                 dmask:0x3 dim:SQ_RSRC_IMG_2D
v_fmac_f32_e64   v8, s17, 0.5
image_sample_lz  v[15:16], v[7:8], s[4:11], s[20:23]
                 dmask:0x3 dim:SQ_RSRC_IMG_2D
v_mov_b32_e32    v12, v8
image_sample_lz  v[7:8], v[11:12], s[4:11], s[20:23]
                 dmask:0x3 dim:SQ_RSRC_IMG_2D
s_buffer_load_b128 s[20:23], s[12:15], 0x40
...
s_waitcnt lgkmcnt(0)
buffer_load_b64 v[0:1], v0, s[16:19], 0 offen offset:8
s_waitcnt vmcnt(4)
v_add_f32_e64 v11, |v10|, |v9|
```

Tweaking the Fix Point by Waiting Earlier

```
    buffer_load_b128   v[0:3], v0, s[16:19], 0 offen
    ...
.LBB0_6:
    ...
    s_waitcnt          vmcnt(0)
    buffer_store_b128  v[0:3], v1, s[20:23], 0 offen
    ...
    v_add_u32          v1, v1, s5
    ...
    s_cbranch_scc0     .LBB0_6
```

```
    buffer_load_b128   v[0:3], v0, s[16:19], 0 offen
    ...
    s_waitcnt          vmcnt(0)
.LBB0_6:
    ...
    buffer_store_b128  v[0:3], v1, s[20:23], 0 offen
    ...
    v_add_u32          v1, v1, s5
    ...
    s_cbranch_scc0     .LBB0_6
```

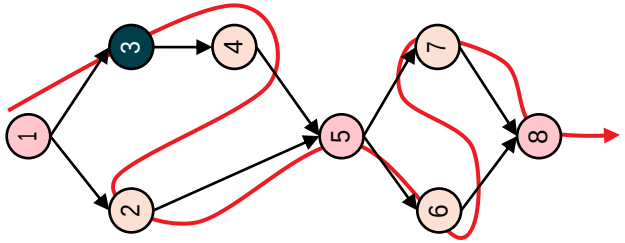
- Fix a false stall that affected GCN by moving the wait to before the loop
- RDNA addressed this issue by separating counters for loads vs. stores

Control Flow Lowering

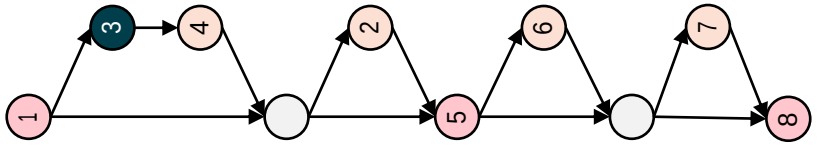
Control Flow Lowering

- The wave must follow a control-flow path that encompasses the paths of all individual threads that have been mapped to its lanes
- Compiler transforms the CFG accordingly and inserts bitmask manipulation code
- Analysis which values are uniform vs. divergent
  - Fix point iteration with some tricky aspects due to “temporal divergence”
  - Can sometimes avoid the CFG transform if branch conditions are uniform

Original CFG



Wave-transformed,
assuming divergent branches



Register Allocation and Memory Instruction Scheduling

Register Allocation Size Matters

- RDNA has 256 architectural vector registers that can be accessed by instructions
- There are typically 1024 physical vector registers per SIMD
- Most shaders use far less than 256 vector registers
- Vector registers are allocated to waves at launch according to their register size
- This affects **occupancy**, with profound impact on performance
- Example:
  - With 64 VGPRs in Wave32 mode, all 16 wave slots can be used
  - With 64 VGPRs in Wave64 mode, only 8 wave slots can be used
  - With 80 VGPRs in Wave32 mode, only 12 wave slots can be used

Latency hiding via Occupancy vs. Instruction Scheduling

- Memory latency is high, so hiding that latency is important
- Idea #1: Move loads as early as possible?
  - Other instructions of the wave can execute while the load is in flight
  - Normally an improvement if it doesn't change the register budget
  - What if it requires us to grow the register budget?
- Idea #2: Maximize the number of loads in flight (memory-level parallelism)
  - There are broadly two ways of doing that:
    - Increase the number of waves in flight → want a smaller register budget
    - Increase the number of loads in flight per wave → want a larger register budget
  - Interacts with loop unrolling
    - One of the major effects of loop unrolling on the GPU is that it can unlock memory parallelism
    - Like out-of-order execution on a CPU
- In practice:
  - Do a bit of everything
  - Allocation granularity and "occupancy boundaries" are often in our favor
  - Second-order effects on the cache hierarchy can easily throw a wrench in any theory

Thank you!

COPYRIGHT AND DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


AMD