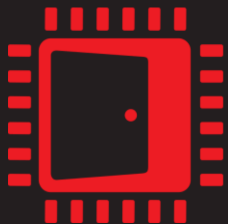
RADEON

A TRIP DOWN THE GPU COMPILER PIPELINE

DR. NICOLAI HÄHNLE, DR. MATTHÄUS G. CHAJDAS



AMD
GPUOpen

Frontend Islands

Static Single Harbour

Backend Lair

QUICK!: HOW MANY FLOAT MULTIPLIES ARE IN THIS SHADER?

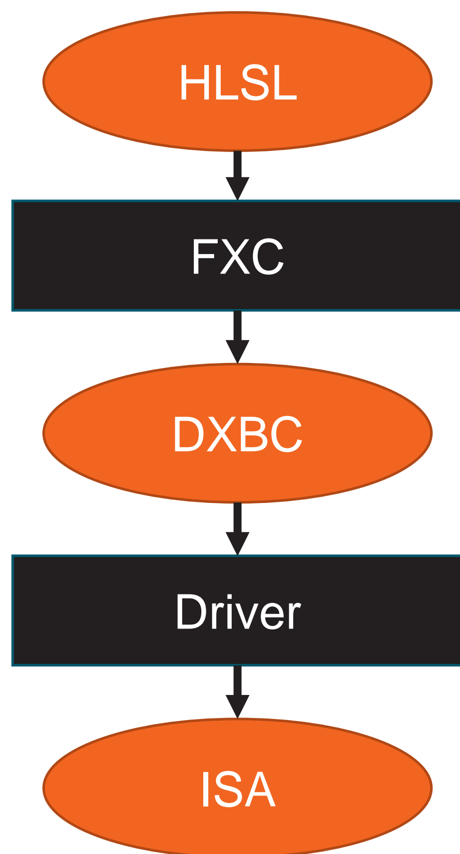
```
Buffer<float4> inputBuffer;
RWBuffer<float> outputBuffer;

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

QUICK!: HOW MANY FLOAT MULTIPLIES ARE IN THIS SHADER?

Answer: 4

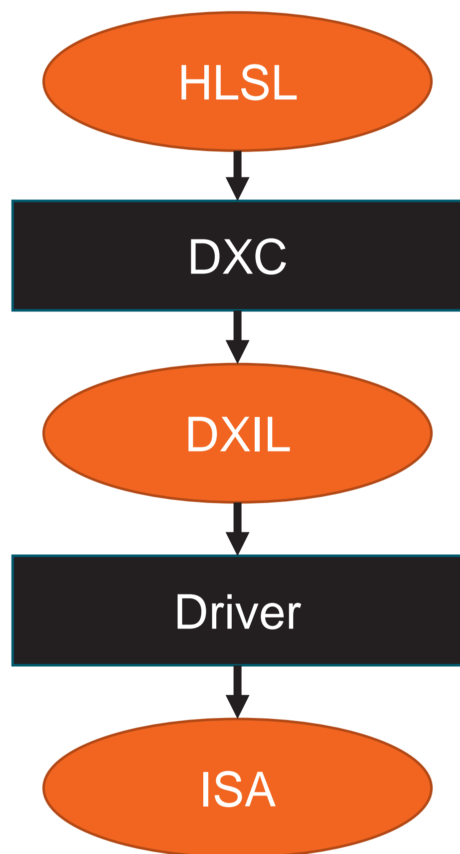
```
buffer_load_format_xyzw  v[0:3], v8, s[4:7], 0 idxen
tbuffer_load_format_xyzw  v[4:7], v4, s[12:15], 0 idxen
format:[BUF_FMT_32_32_32_32_FLOAT]
s_waitcnt     vmcnt(0)
v_mul_f32     v0, v0, v4
v_mac_f32     v0, v1, v5
v_mac_f32     v0, v2, v6
v_mac_f32     v0, v3, v7
v_mov_b32     v1, v0
v_mov_b32     v2, v0
v_mov_b32     v3, v0
buffer_store_format_xyzw  v[0:3], v8, s[8:11], 0 idxen
```

COMPILER PIPELINE: DX11



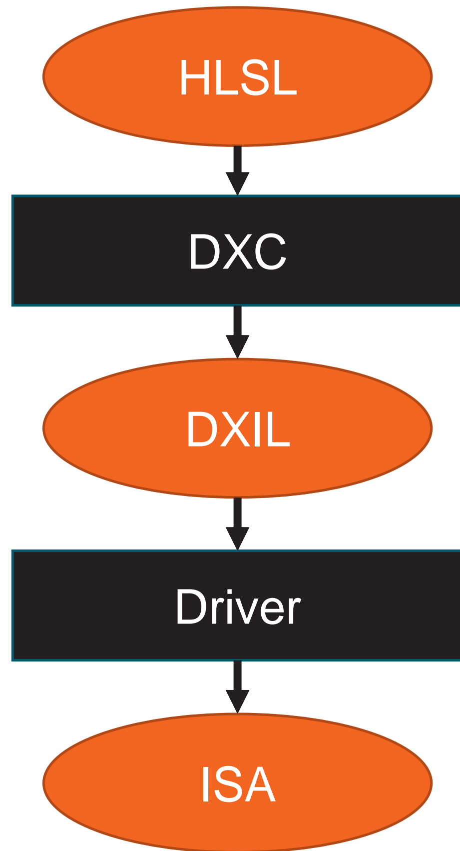

```
cs_5_0
dcl_globalFlags refactoringAllowed
dcl_immediateConstantBuffer { { 1.000000, 0, 0, 0},
                              { 0, 1.000000, 0, 0},
                              { 0, 0, 1.000000, 0},
                              { 0, 0, 0, 1.000000} }

…

ld_indexable(buffer)(float,float,float,float) r0.xyzw, vThreadID.xxxx,
                                              t0.xyzw
mov r1.x, vThreadID.y
dp4 r0.x, r0.xyzw, icb[r1.x + 0].xyzw
store_uav_typed u0.xyzw, vThreadID.xxxx, r0.xxxx
ret
```

COMPILER PIPELINE: DX12




```
…
  %6 = call … @dx.op.bufferLoad.f32(i32 68, %dx.types.Handle %3, i32 %4, i32 undef)
  %7 = extractvalue %dx.types.ResRet.f32 %6, 0
  %8 = extractvalue %dx.types.ResRet.f32 %6, 1
  %9 = extractvalue %dx.types.ResRet.f32 %6, 2
  %10 = extractvalue %dx.types.ResRet.f32 %6, 3
  %11 = getelementptr inbounds [4 x float], [4 x float]* %1, i32 0, i32 0
  store float %7, float* %11, align 4
  %12 = getelementptr inbounds [4 x float], [4 x float]* %1, i32 0, i32 1
  store float %8, float* %12, align 4
  %13 = getelementptr inbounds [4 x float], [4 x float]* %1, i32 0, i32 2
  store float %9, float* %13, align 4
  %14 = getelementptr inbounds [4 x float], [4 x float]* %1, i32 0, i32 3
  store float %10, float* %14, align 4
  %15 = getelementptr inbounds [4 x float], [4 x float]* %1, i32 0, i32 %5
  %16 = load float, float* %15, align 4
  call void @dx.op.bufferStore.f32(i32 69, %dx.types.Handle %2, i32 %4, i32 undef,
                                   float %16, float %16, float %16, float %16, i8 15)
…
```

COMPILER PIPELINE: DX12



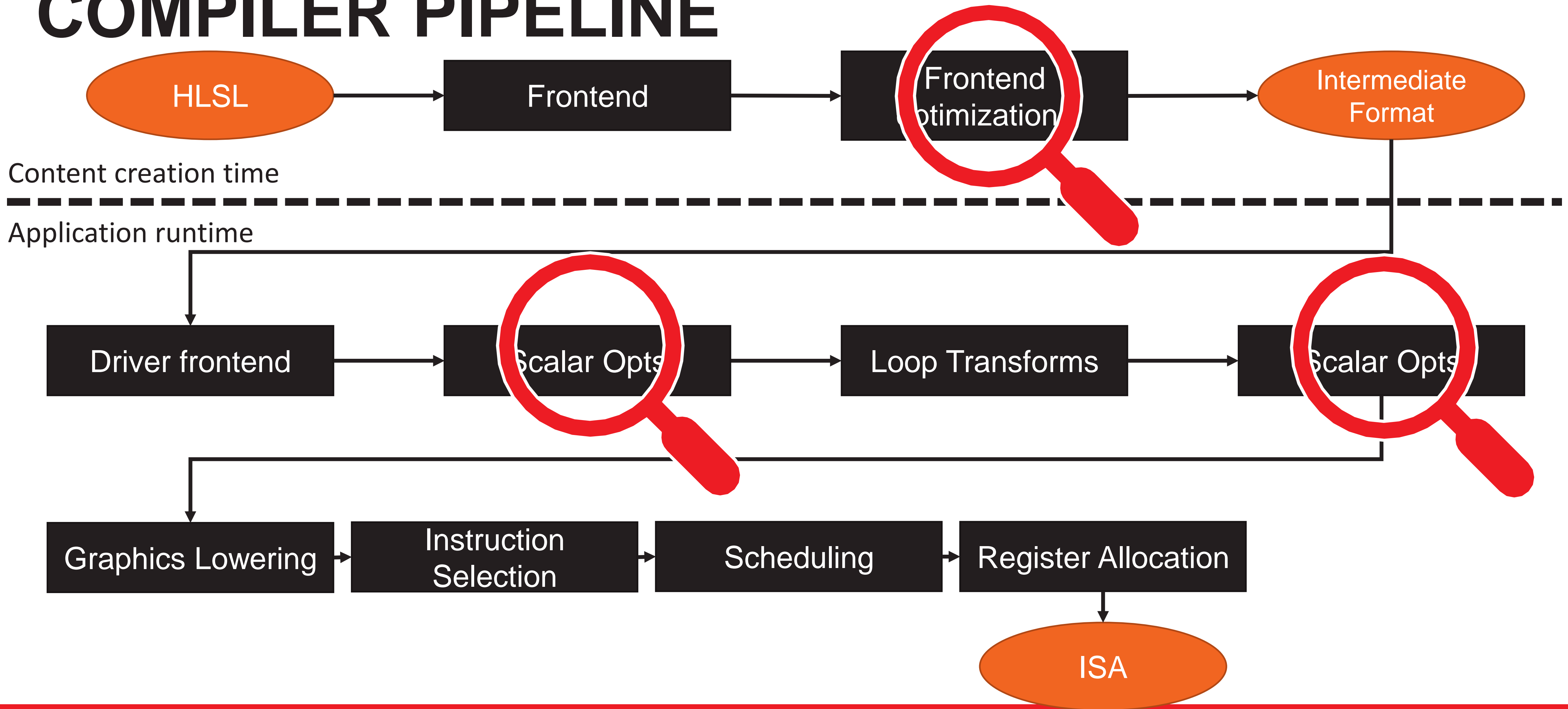

```
  …
  v_cmp_eq_i32  s[4:5], v0, 0
  v_cmp_eq_i32  s[6:7], v0, 1
  v_cmp_eq_i32  vcc, 2, v0
  s_waitcnt     lgkmcnt(0)
  buffer_load_format_xyzw  v[0:3], v4, s[8:11], 0 idxen
  s_waitcnt     vmcnt(0)
  v_cndmask_b32  v2, v3, v2, vcc
  v_cndmask_b32  v1, v2, v1, s[6:7]
  v_cndmask_b32  v0, v1, v0, s[4:5]
  v_mov_b32     v1, v0
  v_mov_b32     v2, v0
  v_mov_b32     v3, v0
  buffer_store_format_xyzw  v[0:3], v4, s[0:3], 0 idxen glc
…
```

Frontend Islands

Static Single Harbour

Backend Lair

COMPILER PIPELINE



WHY IR?

- Unambigious parsing
- Simpler type system

```
switch (commands[did.x]) {
  case 0: result = i.x + i.y; break;
  case 1: result = i.x - i.y; break;
  case 2: result = i.x * i.y; break;
  case 3: result = i.x / i.y; break;
  default: break;
}
```

```
; <label>:10
  %11 = fadd fast float %6, %7
  br label %18

; <label>:
  %13 = fsub fast float %6, %7
  br label %18

; <label>:
  %15 = fmul fast float %6, %7
  br label %18

; <label>:
  %17 = fdiv fast float %6, %7
  br label %18
```

TRANSLATING TO IR

- Vulkan®, Direct3D®11 and Direct3D®12 all go through an intermediate representation (SPIR-V, DXBC, DXIL)
- There is at least one more intermediate representation in the compiler
- This is (generally) a lossy conversion!
- For instance:
  - Types can change (as we saw, buffer types are all the same)
  - Things can disappear
  - Expressions can change

LEGALIZATION

- Makes code “legal”
- Sometimes the high-level language is more expressive than the IR/backend
- Code needs to get “legalized” to be valid IR level code
- Simple example:
  - Imagine division is forbidden at the IR level.
  - Solution: Replace all divisions with multiply by reciprocal

LEGALIZATION

```
Buffer<float4> inputBuffer0;
RWBuffer<float> outputBuffer;

float LoadAndProcess(Buffer<float4> b, int i)
{
  return dot(b[i], b[i]);
}

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
  outputBuffer[did.x] = LoadAndProcess(inputBuffer0, did.x);
}
```

LEGALIZATION

- It is illegal to pass a buffer into a function
- The IR **can** express it, but validation will fail

```
      OpStore %param_var_i %34
%35 = OpFunctionCall %float %LoadAndProcess %param_var_b %param_var_i
%37 = OpAccessChain %_ptr_Function_uint %did %int_0
%38 = OpLoad %uint %37
%39 = OpLoad %type_buffer_image_0 %outputBuffer
      OpImageWrite %39 %38 %35 None
```

```
%LoadAndProcess = OpFunction %float None %41
%b = OpFunctionParameter %_ptr_Function_type_buffer_image
%i = OpFunctionParameter %_ptr_Function_int
```

LEGALIZATION

- Legalization force-inlines the code
- Function call is removed; no parameter passing required, everyone is happy!

```
%21 = OpBitcast %int %20
%22 = OpBitcast %uint %21
%23 = OpImageFetch %v4float %19 %22 None
%24 = OpImageFetch %v4float %19 %22 None
%25 = OpDot %float %23 %24
%26 = OpLoad %type_buffer_image_0 %outputBuffer
      OpImageWrite %26 %20 %25 None
```

BUFFER TYPES CAN CHANGE

- High-level language

```
Buffer<float> inputBuffer0 : register(t0);
RWBuffer<float> outputBuffer : register(u0);

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
  outputBuffer[did.x] = inputBuffer0[did.x];
}
```

BUFFER TYPES CAN CHANGE

- IR level

```
; inputBuffer0                      texture     f32           buf      T0              t0     1
; outputBuffer                          UAV     f32           buf      U0              u0     1
```

- DXASM

```
dcl_resource_buffer (float,float,float,float) T0[0:0], space=0
dcl_uav_typed_buffer (float,float,float,float) U0[0:0], space=0
```

- DXIL

```
%"class.Buffer<float>" = type { float }
%"class.RWBuffer<float>" = type { float }
…
call void @dx.op.bufferStore.f32(i32 69, %dx.types.Handle %1, i32 %3, i32 undef,
    float %5, float %5, float %5, float %5, i8 15)
```

BUFFER TYPES CAN CHANGE

- API flexibility sometimes requires compiler sacrifices
- Generated ISA

```
v_mov_b32     v3, v2
v_mov_b32     v4, v2
v_mov_b32     v5, v2
buffer_store_format_xyzw  v[2:5], v1, s[0:3], 0 idxen glc // 00000000004C: E01C6000 80000201
```

- 4 Registers get initialized and passed, even though you and I know this is not needed

Frontend Islands

Static Single Harbour

Backend Lair

STATIC SINGLE ASSIGNMENT FORM (SSA)

- An important concept in compiler theory you need to know about
- Modern compilers use SSA
- Every variable gets assigned exactly once

```
x = 23;
y = 37;
x = x * y;
…
z = x;
```

```
x_0 = 23;
y_0 = 37;
x_1 = x_0 * y_0;
…
z_0 = x_1;
```

SSA – BRANCHES & LOOPS

- SSA obviously doesn't work for branches and loops without this one weird trick
- Phi-nodes to the rescue

```
x = 23;
if (y) {
  x = 42;
}
z = x;
```

```
x_0 = 23;
L1: if (y_0) {
  x_1 = 42;
}
z = Φ(L1, x_0, x_1);
```

SSA – PHI NODES

- Compiler needs to resolve Phi nodes
- This is where things get tricky, more so the more ways variables enter the Phi node
  - Loops with multiple exits
  - Switch statements
  - Complex control flow

SSA – PHI NODES

- How many inputs does the Phi node have after this switch?

```
float result = 0;
float2 i = inputs[did.x];

switch (commands[did.x]) {
  case 0: result = i.x + i.y; break;
  case 1: result = i.x - i.y; break;
  case 2: result = i.x * i.y; break;
  case 3: result = i.x / i.y; break;
  default: break;
}

   return result;
```

SSA – PHI NODES

- Correct – 5!
- 4 case labels and the value before the switch in case you hit the `default`

```
%27 = phi float [ %12, %9 ], [ %16, %13 ], [ %20, %17 ], [ %24, %21 ], [ 0.000000e+00, %25 ]
```

SCALAR OPTIMIZATIONS

- Dead-code elimination
- Constant folding
- Common sub-expression elimination
- Global value numbering
- Strength reduction

CONSTANT FOLDING

- Trivial with SSA

```
#define BLOCK_WIDTH (6)
#define BLOCK_HEIGHT (6)

[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    uint2 blockSize = uint2(BLOCK_WIDTH, BLOCK_HEIGHT);
    uint2 shifted = did.xy + blockSize / 2;
    float4 color = input[shifted];
    output[did.xy] = color;
}
```

Oh no, a division!

CONSTANT FOLDING

```
%uint_6 = OpConstant %uint 6
     %9 = OpConstantComposite %v2uint %uint_6 %uint_6
%uint_2 = OpConstant %uint 2
    %11 = OpConstantComposite %v2uint %uint_2 %uint_2

    %39 = OpLoad %v3uint %did
    %40 = OpVectorShuffle %v2uint %39 %39 0 1

    %42 = OpUDiv %v2uint %9 %11
    %43 = OpIAdd %v2uint %40 %42
```

Direct frontend translation

CONSTANT FOLDING

```
%uint_3 = OpConstant %uint 3
    %12 = OpConstantComposite %v2uint %uint_3 %uint_3


    %39 = OpLoad %v3uint %did
    %40 = OpVectorShuffle %v2uint %39 %39 0 1


    %43 = OpIAdd %v2uint %40 %12
```

COMMON SUB-EXPRESSION ELIMINATION (CSE)

```
float a = (inputBuffer[did.x].x + inputBuffer[did.x].y) *
          inputBuffer[did.x].z;
float b = (inputBuffer[did.x].x + inputBuffer[did.x].y) *
          inputBuffer[did.x].w;
outputBuffer[did.x] = float2(a, b);
```

COMMON SUB-EXPRESSION ELIMINATION (CSE)

```
…
%21 = OpImageFetch %v4float %20 %19 None
%22 = OpCompositeExtract %float %21 0
%23 = OpImageFetch %v4float %20 %19 None
%24 = OpCompositeExtract %float %23 1
%25 = OpFAdd %float %22 %24
%26 = OpImageFetch %v4float %20 %19 None
%27 = OpCompositeExtract %float %26 2
%28 = OpFMul %float %25 %27
%29 = OpImageFetch %v4float %20 %19 None
%30 = OpCompositeExtract %float %29 0
%31 = OpImageFetch %v4float %20 %19 None
%32 = OpCompositeExtract %float %31 1
%33 = OpFAdd %float %30 %32
%34 = OpImageFetch %v4float %20 %19 None
%35 = OpCompositeExtract %float %34 3
%36 = OpFMul %float %33 %35
%37 = OpCompositeConstruct %v2float %28 %36
…
```

COMMON SUB-EXPRESSION ELIMINATION (CSE)

```
…
%21 = OpImageFetch %v4float %20 %19 None
%22 = OpCompositeExtract %float %21 0

%24 = OpCompositeExtract %float %21 1
%25 = OpFAdd %float %22 %24

%27 = OpCompositeExtract %float %21 2
%28 = OpFMul %float %25 %27

%30 = OpCompositeExtract %float %21 0

%32 = OpCompositeExtract %float %21 1
%33 = OpFAdd %float %30 %32

%35 = OpCompositeExtract %float %21 3
%36 = OpFMul %float %33 %35
%37 = OpCompositeConstruct %v2float %28 %36
…
```

AMD GPUOpen

COMMON SUB-EXPRESSION ELIMINATION (CSE)

```
…
%21 = OpImageFetch %v4float %20 %19 None
%22 = OpCompositeExtract %float %21 0

%24 = OpCompositeExtract %float %21 1
%25 = OpFAdd %float %22 %24

%27 = OpCompositeExtract %float %21 2
%28 = OpFMul %float %25 %27




%33 = OpFAdd %float %22 %24

%35 = OpCompositeExtract %float %21 3
%36 = OpFMul %float %33 %35
%37 = OpCompositeConstruct %v2float %28 %36
…
```

COMMON SUB-EXPRESSION ELIMINATION (CSE)

```
…
%21 = OpImageFetch %v4float %20 %19 None
%22 = OpCompositeExtract %float %21 0

%24 = OpCompositeExtract %float %21 1
%25 = OpFAdd %float %22 %24

%27 = OpCompositeExtract %float %21 2
%28 = OpFMul %float %25 %27






%35 = OpCompositeExtract %float %21 3
%36 = OpFMul %float %23 %35
%37 = OpCompositeConstruct %v2float %28 %36
…
```

COMMON SUB-EXPRESSION ELIMINATION (CSE)

```
buffer_load_format_xyzw  v[0:3], v4, s[4:7], 0 idxen
s_waitcnt     vmcnt(0)
v_add_f32     v1, v0, v1
v_mul_f32     v0, v2, v1
v_mul_f32     v1, v3, v1
buffer_store_format_xyzw  v[0:3], v4, s[0:3], 0 idxen glc
```

GLOBAL VALUE NUMBERING (GVN)

- Track value equivalence across basic blocks

```
[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    float4 color;
    if (did.x == did.y) {
        color = input[did.x + did.y];
    } else {
        color = input[2 * did.x];
    }
    output[4 * did.x + did.y] = color;
}
```

GLOBAL VALUE NUMBERING (GVN)

- Track value equivalence across basic blocks

```
[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    float4 color;
    if (did.x == did.y) {
        color = input[did.x + did.x];
    } else {
        color = input[2 * did.x];
    }
    output[4 * did.x + did.y] = color;
}
```

Equal to 2 * did.x

GLOBAL VALUE NUMBERING (GVN)

- Track value equivalence across basic blocks

```
[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    float4 color;
    color = input[did.x + did.x];
    if (did.x == did.y) {
    } else {
    }
    output[4 * did.x + did.y] = color;
}
```

GLOBAL VALUE NUMBERING (GVN)

- Track value equivalence across basic blocks

```
[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    float4 color;
    color = input[did.x + did.x];
    output[4 * did.x + did.y] = color;
}
```

GLOBAL VALUE NUMBERING (GVN)

- Track value equivalence across basic blocks

```
[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    float4 color;
    color = input[did.x << 1];
    output[(did.x << 2) + did.y] = color;
}
```

GLOBAL VALUE NUMBERING (GVN)

- Track value equivalence across basic blocks

```
v_lshlrev_b32  v2, 1, v0
buffer_load_format_xyzw  v[2:5], v2, s[4:7], 0 idxen
v_lshl_add_u32  v1, v0, 2, v1
s_waitcnt     vmcnt(0)
buffer_store_format_xyzw  v[2:5], v1, s[8:11], 0 idxen
```

STRENGTH REDUCTION

- Replace expensive operations by cheaper ones

```
[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    output[did.xy] = input[did.xy] / scale;
}
```

STRENGTH REDUCTION

- Replace expensive operations by cheaper ones

```
image_load     v[3:6], [v2,v1], s[20:27] dmask:0xf dim:SQ_RSRC_IMG_2D unorm
s_buffer_load_dword  s0, s[12:15], null
s_waitcnt      lgkmcnt(0)
v_rcp_f32      v0, s0
s_waitcnt      vmcnt(0)
v_mul_f32      v3, v3, v0
v_mul_f32      v4, v4, v0
v_mul_f32      v5, v5, v0
v_mul_f32      v6, v6, v0
image_store    v[3:6], [v2,v1], s[4:11] dmask:0xf dim:SQ_RSRC_IMG_2D unorm
```

Reciprocal instead of division

Shared by all components (CSE)

STRENGTH REDUCTION

- Replace expensive operations by cheaper ones

```
[numthreads(8,4,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    output[did.xy] = input[did.xy / 3];
}
```

Integer division by constant

STRENGTH REDUCTION

- Replace expensive operations by cheaper ones

```
v_mul_hi_u32  v0, v2, lit(0xaaaaaaab)
v_mul_hi_u32  v3, v1, lit(0xaaaaaaab)
v_lshrrev_b32  v0, 1, v0
v_lshrrev_b32  v3, 1, v3
s_waitcnt     lgkmcnt(0)
image_load    v[3:6], [v0,v3], s[16:23] dmask:0xf dim:SQ_RSRC_IMG_2D unorm
s_waitcnt     vmcnt(0)
image_store   v[3:6], [v2,v1], s[4:11] dmask:0xf dim:SQ_RSRC_IMG_2D unorm
s_endpgm
```

Multiply-and-shift

Frontend Islands

Static Single Harbour

Backend Lair

THE BACKEND

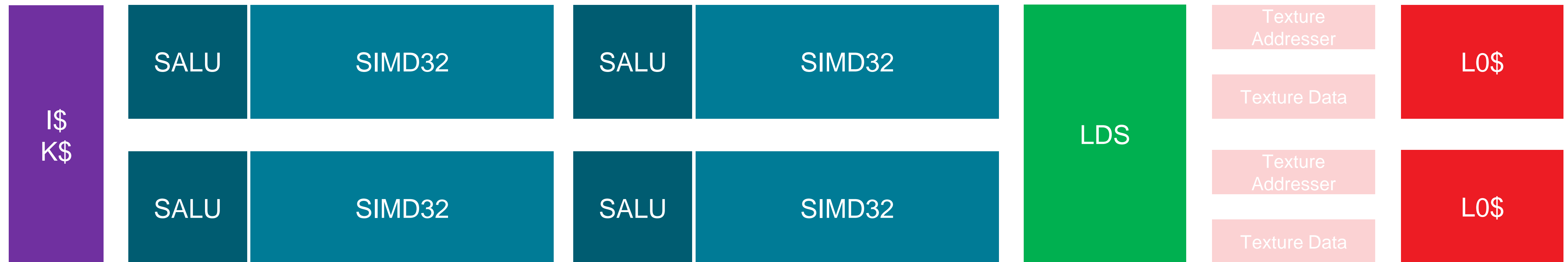
- Here, the IR meets the HW
- The backend is designed to
  - Perform HW-specific optimizations
  - Translate high-level constructs into HW instructions
  - Optimize

MEMORY ACCESS

- The backend compiler knows which caches are used for every instruction
- Everything before the backend compiler assumes a unified, coherent cache
- I.e. write-followed-by-read just works

MEMORY ACCESS

- On RDNA, we have various separate caches
- K$ is read-only, shared among all SIMD32; L0$ is read-write, also shared, but not **coherent** with K$



MEMORY ACCESS

MEMORY ACCESS

```
RWStructuredBuffer<int> inputBuffer0 : register(u0);
cbuffer inputBuffer1 : register(b0)
{
  int constants[64];
};
RWBuffer<float4> outputBuffer : register(u1);

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
  outputBuffer[did.x] = inputBuffer0[did.y] + constants[did.y];
}
```

MEMORY ACCESS

- `inputBuffer0` could alias with `outputBuffer`: Must go through L0$
- `inputBuffer1` does not alias with `outputBuffer`: Can go through K$

```
buffer_load_dword  v1, v1, s[4:7], 0 idxen                 // 000000000014: E0302000 80010101
s_buffer_load_dword  s0, s[12:15], s1                      // 00000000001C: F4200006 02000000
```

MEMORY ACCESS

- HW can also convert formats along the way
- This depends on what type of load it is!

BUFFER TYPES MATTER

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

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

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

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

```
…
buffer_load_format_xyzw  v[0:3],
    v4, s[8:11], 0 idxen
s_waitcnt      vmcnt(0)
v_cndmask_b32  v2, v3, v2, vcc
v_cndmask_b32  v1, v2, v1, s[6:7]
v_cndmask_b32  v0, v1, v0, s[4:5]
…
```

```
…
s_waitcnt      lgkmcnt(0)
buffer_load_dword  v0, v0, s[8:11], 0 offen
s_load_dwordx4  s[0:3], s[0:1], null
s_waitcnt      vmcnt(0)
v_mov_b32      v1, v0
v_mov_b32      v2, v0
v_mov_b32      v3, v0
…
```

EXECUTION MODEL

- A GPU doesn't execute code like a CPU

EXECUTION MODEL

- Scalar ALU core with it’s on scalar register file
- Vector ALU attached to it with a separate vector register file (with one register per lane)
- VALU is turned on/off using an execution mask

EXECUTION MODEL

- Scalar branch – branch over scalar value

```
cbuffer inputBuffer0 : register(b0) {
  int constants[64];
};
RWByteAddressBuffer outputBuffer : register(u0);

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
  if (constants [0]) {
    outputBuffer.Store(did.x, 3);
  } else {
    outputBuffer.Store (did.x, 42);
  }
}
```

Uniform across the wave

EXECUTION MODEL

- Branches can only happen on SALU
- Scalar comparison, scalar branch

```
s_buffer_load_dword  s0, s[8:11], null
s_waitcnt     lgkmcnt(0)
s_cmp_eq_u32  s0, 0
s_cbranch_scc1  label_0030
 v_mov_b32     v1, 3
 buffer_store_dword  v1, v0, s[4:7], 0 offen
 s_endpgm
label_0030:
 v_mov_b32     v1, 42
 buffer_store_dword  v1, v0, s[4:7], 0 offen
s_endpgm
```

EXECUTION MODEL

- Scalar branching, execution mask is fully on/off

EXECUTION MODEL

- Non-uniform branch

```
Buffer<int> inputBuffer0 : register(t0);
RWByteAddressBuffer outputBuffer : register(u0);

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
  if (inputBuffer0[did.x]) {
    outputBuffer.Store(did.x, 3);
  } else {
    outputBuffer.Store (did.x, 42);
  }
}
```

Varying across the wave

EXECUTION MODEL

- Non-uniform data – branch is no longer uniform unless it happens that all lanes produce the same result

```
buffer_load_format_x  v1, v0, s[4:7], 0 idxen
s_waitcnt     vmcnt(0)
v_cmp_ne_u32  vcc_lo, 0, v1
  s_and_saveexec_b32  s0, vcc_lo
  v_mov_b32     v1, 3
s_cbranch_execz  label_0030
  buffer_store_dword  v1, v0, s[8:11], 0 offen
label_0030:
  s_andn2_b32   exec_lo, s0, exec_lo
  v_mov_b32     v1, 42
s_cbranch_execz  label_0044
  buffer_store_dword  v1, v0, s[8:11], 0 offen
label_0044:
  s_endpgm
```

Vector comparison using VGPRs

Scalar branch if all are off

Scalar branch if all are off

EXECUTION MODEL

- Non-uniform control flow == execution mask it not all zero or one

EXECUTION MODEL

- Could also remove the optimization to prefer a scalar jump without loss of functionality

```
buffer_load_format_x  v1, v0, s[4:7], 0 idxen
s_waitcnt     vmcnt(0)
v_cmp_ne_u32  vcc_lo, 0, v1
  s_and_saveexec_b32  s0, vcc_lo
  v_mov_b32     v1, 3
  buffer_store_dword  v1, v0, s[8:11], 0 offen
label_0030:
  s_andn2_b32   exec_lo, s0, exec_lo
  v_mov_b32     v1, 42
  buffer_store_dword  v1, v0, s[8:11], 0 offen
s_endpgm
```

Vector comparison using VGPRs

GENERAL RULES

- Combining varying & uniform values …

Input 1	Input 2	Output
Varying	Varying	Varying
Uniform	Varying	Varying
Varying	Uniform	Varying
Uniform	Uniform	Uniform

- This includes loads (i.e. load using varying index – varying load!)
- All of this is automatic

UNIFORMITY AND BUFFERS

- Loading a constant using a uniform index (`did.y` same for all threads in a wave)

```
cbuffer Constants {
    float constants[256];
}
RWStructuredBuffer<float> output;

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    output[did.x] = constants[did.y];
}
```

UNIFORMITY AND BUFFERS

- Yields scalar load as expected

```
s_and_b32      s0, s13, lit(0x00ffffff)
s_lshl_b32     s0, s0, 4
s_buffer_load_dword  s0, s[8:11], s0
v_mad_u32_u24  v0, s12, 64, v0
s_waitcnt      lgkmcnt(0)
v_mov_b32      v1, s0
buffer_store_dword  v1, v0, s[4:7], 0 idxen
```

UNIFORMITY AND BUFFERS

- Loading a constant using a varying index

```
cbuffer Constants {
    float constants[256];
}
RWStructuredBuffer<float> output;

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
    output[did.x] = constants[did.x];
}
```

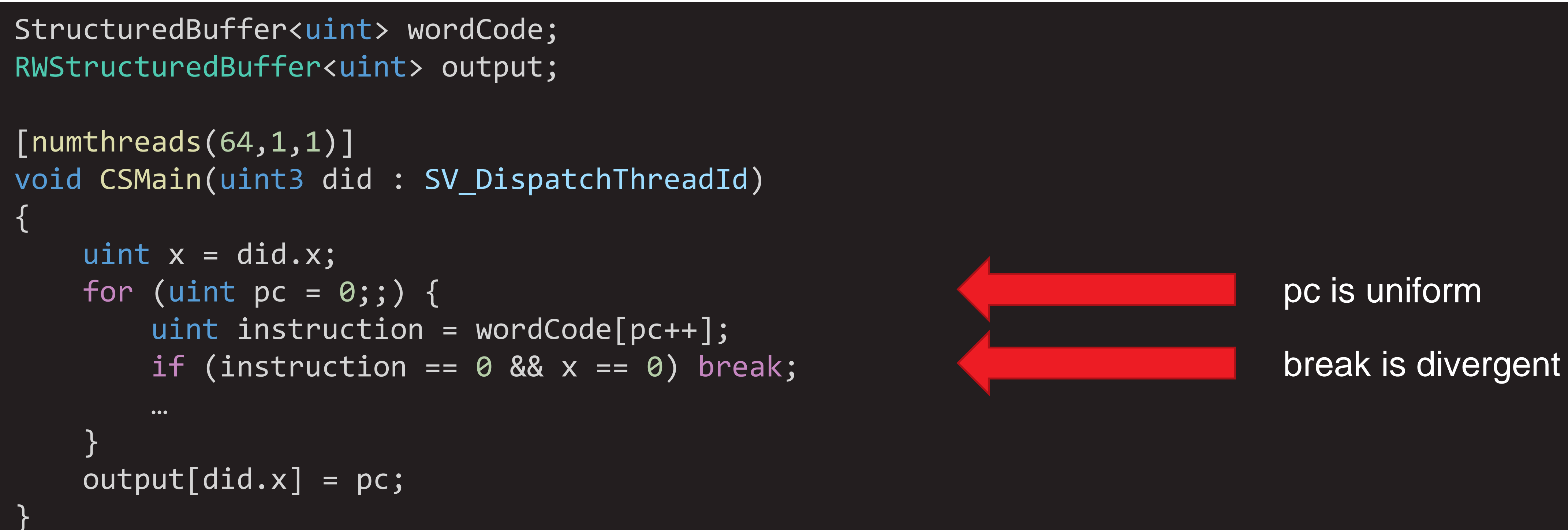
UNIFORMITY AND BUFFERS

- Yields a non-scalar load

```
v_mad_u32_u24  v0, s12, 64, v0
tbuffer_load_format_x  v1, v0, s[8:11], 0 idxen …
s_waitcnt      vmcnt(0)
buffer_store_dword  v1, v0, s[4:7], 0 idxen
```

UNIFORMITY AND CONTROL FLOW

- Propagating divergence gets tricky with control flow



UNIFORMITY AND CONTROL FLOW

```
    v_mov_b32 v1, v0
    s_mov_b32 s9, 0
    s_mov_b32 s10, 0
    s_mov_b32 s4, exec_lo
label_header:
    s_buffer_load_dword s8, s[4:7], s10
    s_add_i32 s9, s9, 1
    s_add_i32 s10, s10, 4
    s_waitcnt lgkmcnt(0)
    v_or_b32_e32 v2, s8, v1
    v_cmp_eq_u32_e32 vcc_lo, 0, v2
    v_mov_b32_e32 v2, s9
    s_andn2_b32 exec_lo, exec_lo, vcc_lo
    s_cbranch_execz label_exit
    ...
label_exit:
    s_mov_b32 exec_lo, s3
    v_lshlrev_b32_e32 v0, 2, v0
    buffer_store_dword v2, v0, s[0:3], 0 offen
```

Setup pc

Fetch instruction & update pc

Save into a vector register

Store the divergent value

CROSSING THE BRIDGE

- Going from scalar to vector is a simple broadcast
- Going from vector to scalar register is not so simple
  - What-if values are not uniform in the register file?
  - Commonly known as “scalarization”
- AKA “`ReadFirstLane`”, “`ReadLane`”

AUTOMATIC SCALARIZATION

- Sometimes, the compiler can do it automatically
- In the following example, `inputBuffer0[did.x]` is non-uniform, but `outputBuffer[0]` clearly is uniform
- Goal: Compute sum *across* VGPRs, then execute a single atomic on one lane

```
Buffer<int> inputBuffer0;
RWBuffer<int> outputBuffer;

[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
  InterlockedAdd(outputBuffer[0], inputBuffer0[did.x]);
}
```

AUTOMATIC SCALARIZATION

```
v_add_nc_u32  v2, v2, v2 row_shr:1
v_add_nc_u32  v2, v2, v2 row_shr:2
v_add_nc_u32  v2, v2, v2 row_shr:4
v_add_nc_u32  v2, v2, v2 row_shr:8
s_mov_b32     exec_lo, lit(0xffff0000)
s_movk_i32    s1, 0xffff
v_permlanex16_b32  v3, v2, -1, s1 fi:1
v_add_nc_u32  v2, v3, v2
s_mov_b32     exec_lo, s0
v_readlane_b32  s0, v2, 31
v_mov_b32     v0, 0
v_mov_b32     v1, s0
s_ff1_i32_b64  s0, exec
s_lshl_b64    s[0:1], 1, s0
s_and_saveexec_b64  s[0:1], s[0:1]
buffer_atomic_add  v1, v0, s[8:11], 0 idxen
```

Reduce using cross-lane operations

Load result into SGPR

Turn off all lanes but one

SCALARIZATION

- If we know the VALU only has a few separate values, we could process them all in turn using the SALU
- “Waterfall loop”
- Use case on RDNA: Texture descriptors live in scalar registers. What if we want to load different textures per lane?

```
float4 result = textureArray1[
  NonUniformResourceIndex(index)].SampleGrad (textureSampler, uv, grad.xy, grad.zw);
```

AUTOMATIC SCALARIZATION

```
label_00C0:
  s_waitcnt     vmcnt(0)
  v_readfirstlane_b32  s6, v6
  v_cmp_eq_u32  vcc, s6, v6
  s_and_saveexec_b32  s7, vcc_lo
  s_cbranch_execz  label_00FC
  s_load_dwordx8  s[12:19], s[0:
  s_waitcnt     lgkmcnt(0)
  s_waitcnt_depctr  0xffe3
  image_sample_d  v[6:9], [v2,v3,v4,v5,v0,v1], s[12:19], s[24:27] dmask:0xf dim:SQ_RSRC_IMG_2D
  s_andn2_b32   s4, s4, exec_lo
  s_cbranch_scc0  label_010C
label_00FC:
  v_nop
  s_mov_b32     exec_lo, s7
  s_and_b32     exec_lo, exec_lo
  s_branch      label_00C0
label_010C:
```

Promote current index to scalar; turn on all lanes which use the same value

Load descriptor

Check if done, of so, terminate

Disable lanes and loop again

MANUAL SCALARIZATION

- Manual scalarization – if you know the data will be uniform for all threads in a wave, you can convert a VGPR to a SGPR yourself

```
[numthreads(64,1,1)]
void CSMain(uint3 did : SV_DispatchThreadId)
{
  if (WaveReadLaneFirst (inputBuffer0[did.x])) {
    outputBuffer.Store(did.x, 3);
  } else {
    outputBuffer.Store (did.x, 42);
  }
}
```

MANUAL SCALARIZATION

```
v_readlane_b32  s0, v1, s0
s_mov_b32      s2, s5
s_mov_b32      s3, s1
s_load_dwordx4  s[4:7], s[2:3], null
s_cmp_eq_i32  s0, 0
s_cbranch_scc1  label_0070
  v_mov_b32      v1, 3
  s_waitcnt      lgkmcnt(0)
  buffer_store_dword  v1, v0, s[4:7], 0 offen glc
  s_endpgm
label_0070:
  v_mov_b32      v1, 42
  s_waitcnt      lgkmcnt(0)
  s_waitcnt_depctr  0xffe3
  buffer_store_dword  v1, v0, s[4:7], 0 offen glc
  s_endpgm
```

Promote value from first active lane to SGPR

Fully scalar branch as a result

TAKEAWAY

- It’s a long way from high level language to ISA
- Translation to IR is generally lossy
- SSA form makes certain optimizations trivial, especially around constants and dead-code
- The backend has to extract uniform/varying information from a high level language and IR which doesn’t care

DISCLAIMER

© *2020* Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

This dragon image is owned by Apple Inc. and is available for your download and use royalty-free. By downloading this image, Apple grants you, and you accept, a non-exclusive license to use this image. All right, title and interest in the image, including the copyright therein, is retained by Apple.

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 for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, 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.

THAT‘S ALL FOLKS!