Vulkanised 2025
The 7th Vulkan Developer Conference
Cambridge, UK | February 11-13, 2025

What is Maximal Reconvergence And Why It Matters

Hugo Devillers, Universität des Saarlandes

Maximal Reconvergence Gives A Well-Defined Answer To:

"Which invocations in my Subgroup are active for the purposes of non-uniform group operations ?"

And what does that question even mean ?!

Caveats

- This is a deeply technical subject
  - I have to fit within 25 minutes
  - I cannot possibly say everything I could say about this
- The specification uses very precise language
  - I will resort to oversimplifications and *lies*
- This talk will focus on the background
  - And not so much the precise spec behaviour
  - This will hopefully be more educational than the alternative

What's an invocation ?

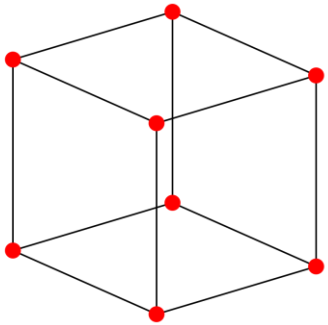
*A single execution of an entry point in a SPIR-V module [...] For example, in compute execution models, a single invocation operates only on a **single work item**, or, in a vertex execution model, a single invocation operates only on a **single vertex**.*

SPIR-V specification (1.6) Section 2.2.5 Control Flow

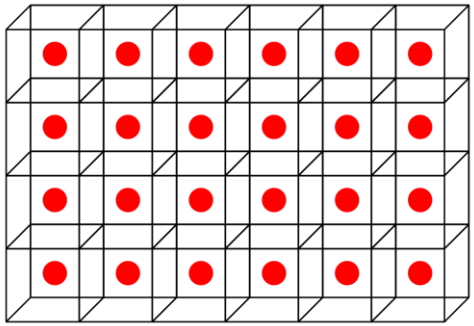
- One work item in a compute shader dispatch
- One vertex in the vertex stage
- One fragment in the fragment stage
- One ray in a raytracing pipeline stage

It's also known as a thread, or lane

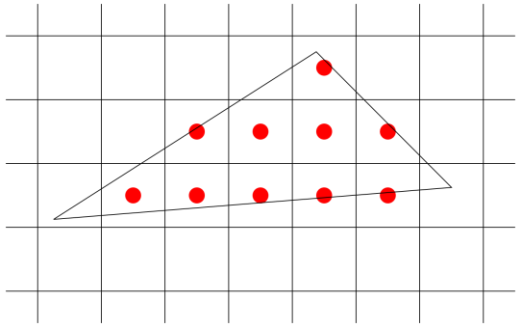
It corresponds to the smallest granularity of execution.
Shaders are programs written at this level.



CmdDrawIndexed(8, 1)



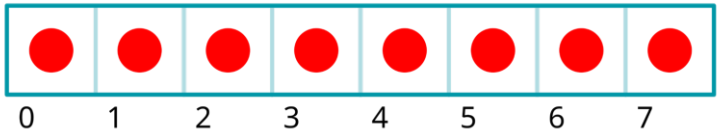
CmdDispatch(6, 4, 1)



Rasterized fragments

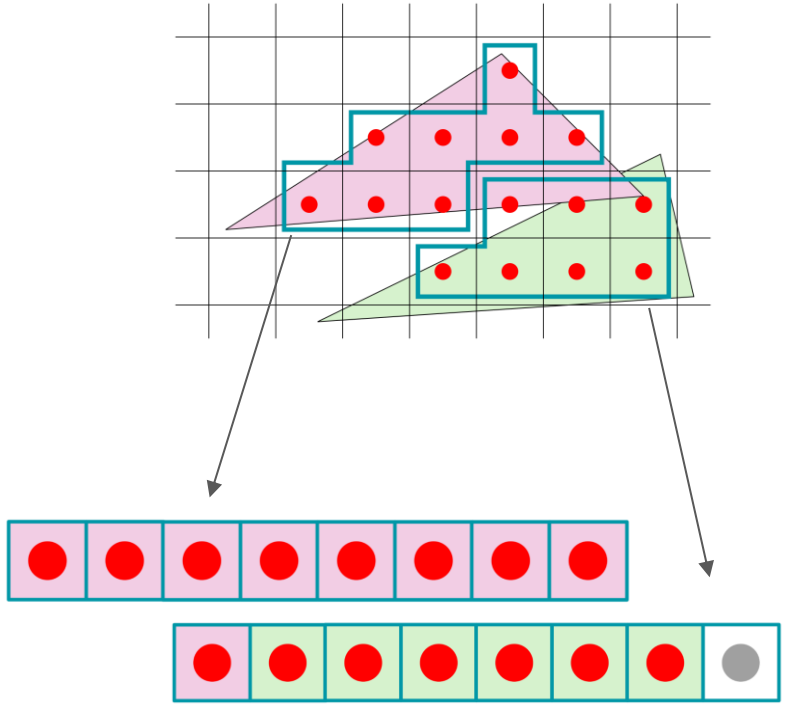
What is a Subgroup ?

- GPUs are **parallel** computers
  - Many Invocations executing in parallel is how very high-performance is achieved
- Invocations are grouped in Subgroups
- Subgroups are also known as waves or wavefronts
- Invocations within a subgroup get a local ID
  - Unlike workgroups, 1-dimensional ID only



- Invocations amongst a Subgroup can efficiently **synchronize** and **exchange data**

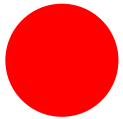
Example: Fragments from two triangles



Note that Fragments from different triangles can end up in the same subgroup !

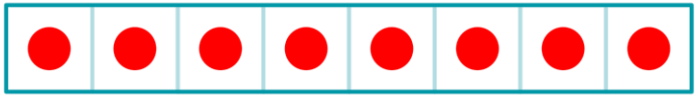
Scopes: Hierarchical parallelism in compute stages

Invocation

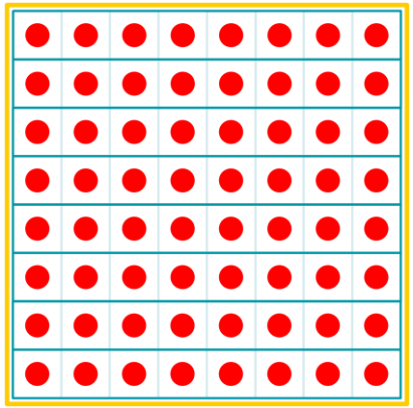


Bottom scope, unit of execution programmed through shaders

Subgroup



Small group of threads whose size is dictated by hardware details.



Workgroup

3D grid, sized by the programmer.

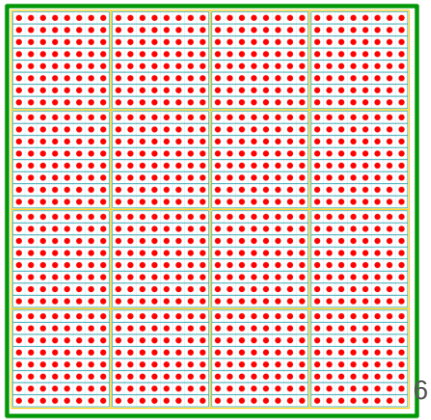
Has shared memory.

Made of subgroups

CrossWorkgroup

Size determined at dispatch, defined as a 3d grid of workgroups.

Top-level scope



Subgroup Execution Example

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

- This is a small compute shader that adds two adjacent numbers together for each work-item.
- We're going to run 4 invocations of it, packed together into one subgroup.
- We'll step through the code and look at the values of the variables for each invocation at each step.

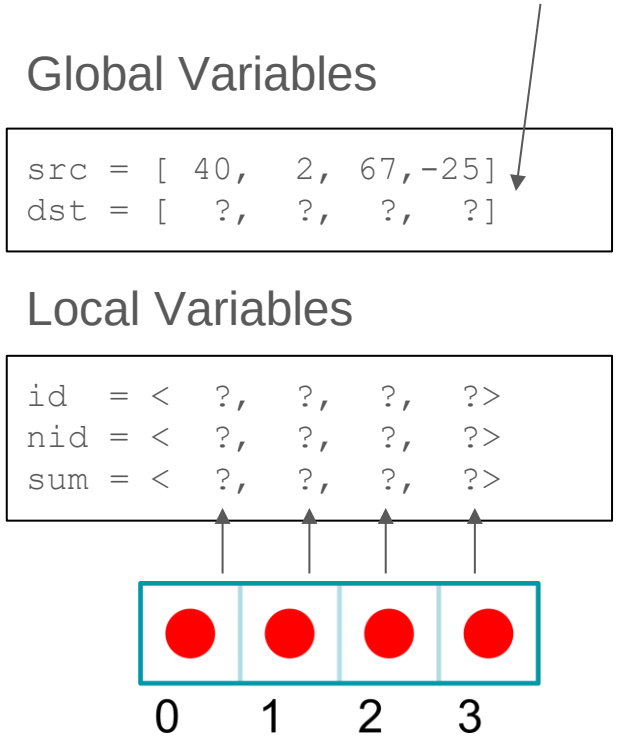
Subgroup Execution Example

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

We're executing this instruction

Value is the same for all invocations



Local variables have one value per invocation (Subgroup size = 4)

Subgroup Execution Example

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  ?,  ?,  ?,  ?>
sum = <  ?,  ?,  ?,  ?>
```

Subgroup Execution Example

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  1,  2,  3,  0>
sum = <  ?,  ?,  ?,  ?>
```

Subgroup Execution Example

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  1,  2,  3,  0>
sum = < 42, 69, 42, 15>
```

Subgroup Execution Example

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

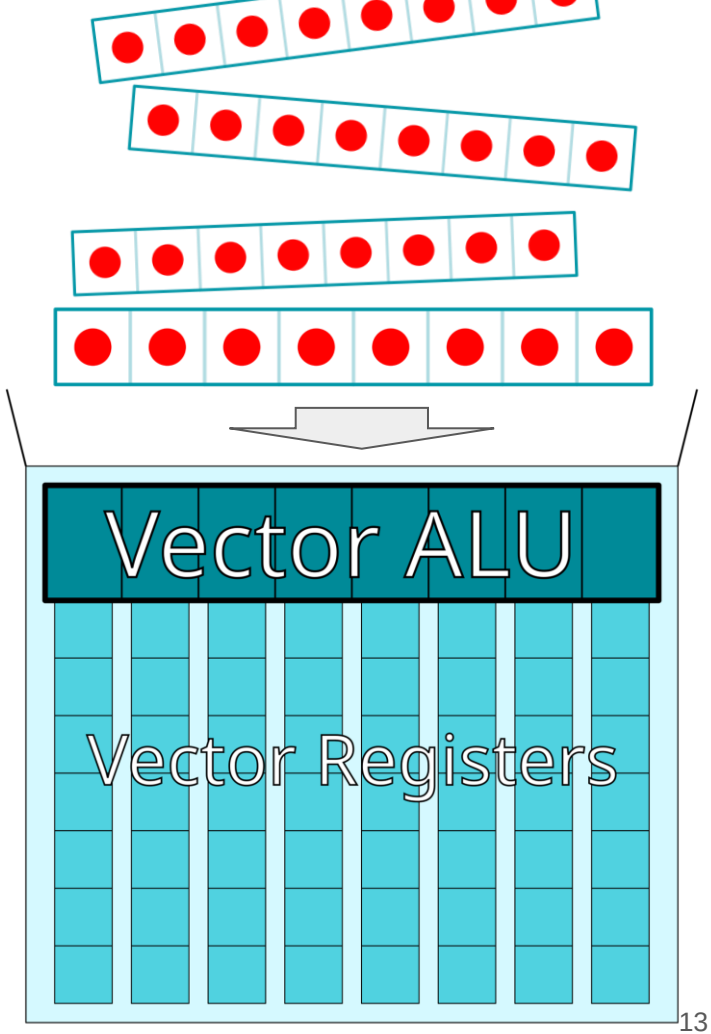
```
src = [ 40,  2, 67,-25]
dst = [ 42, 69, 42, 15]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  1,  2,  3,  0>
sum = < 42, 69, 42, 15>
```

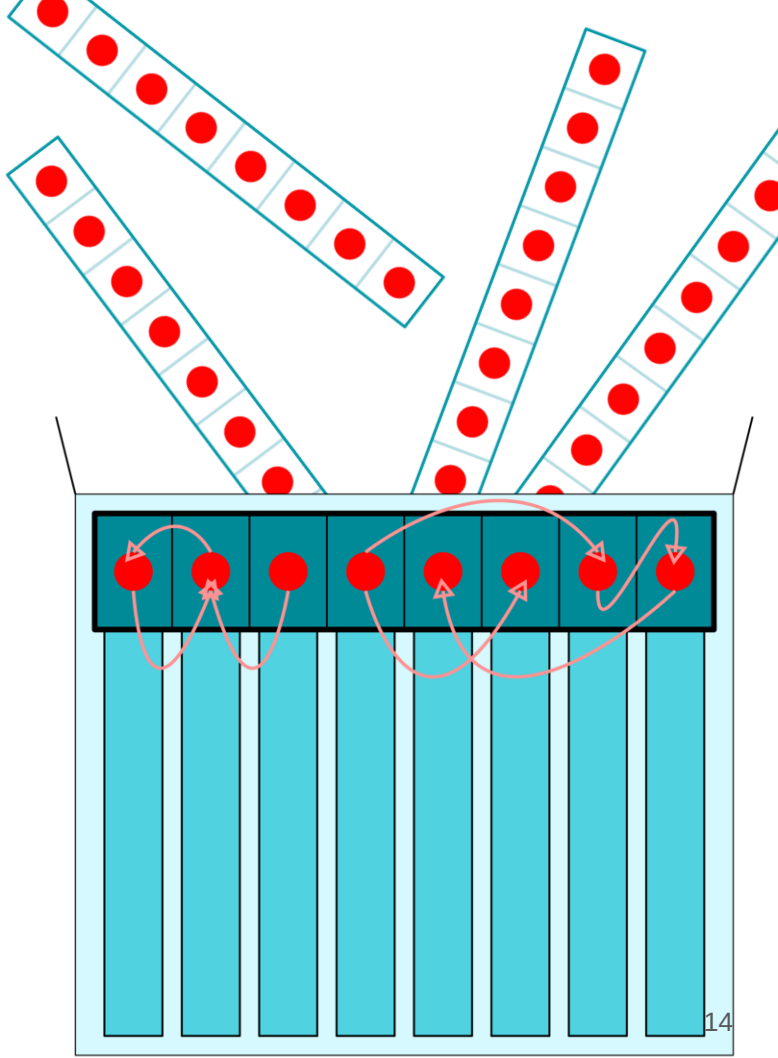
What are subgroups really ?

- Subgroups are SIMD ?
- This is a good **approximation** of reality for the purposes of developing our mental model
- Equating subgroups to SIMD is a lie
  - Not all hardware is actually SIMD.
  - There's nothing preventing the compiler from mapping subgroups to hardware in a non-1:1 fashion, even on SIMD hardware
- It is a useful lie…



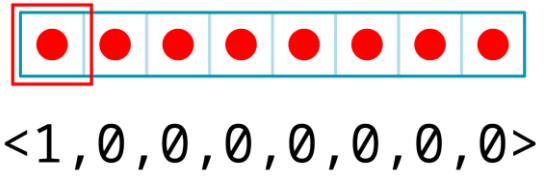
Group Operations

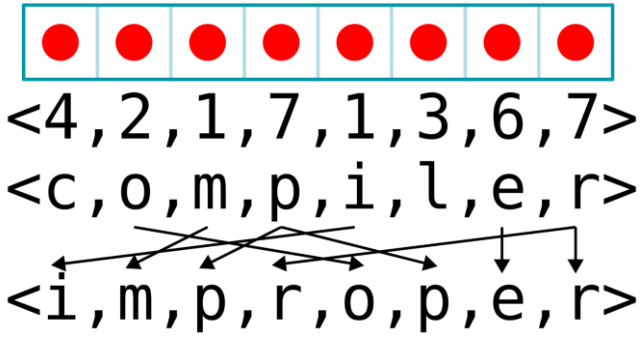
- Invocations can use Group Operations to exchange data efficiently within their Subgroup.
- Subgroup Operations are efficient because the invocations are all executing “together”
- The data is available locally without going to shared or global memory.



An overview of (Sub)Group Operations in Vulkan…

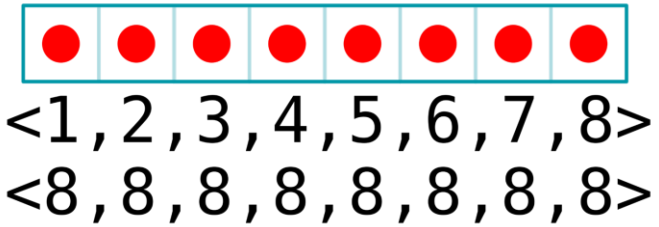
- **Elect**: Picks one thread as the “leader”
  - Result is a value that’s false for all threads but one
- **Ballot**: Vote and obtain a bitmask of the answers





- **Shuffle**: Exchange data with other invocations directly
  - Shuffle up/down: talk to your neighboring thread
  - Shuffle with ID: explicitly send/receive messages within the subgroup

- **Arithmetic**: Compute something amongst participating threads
  - Reductions
  - Inclusive and exclusive add



Group operation example: subgroupMax()

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  id = subgroupMax(id);
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  ?,  ?,  ?,  ?>
nid = <  ?,  ?,  ?,  ?>
sum = <  ?,  ?,  ?,  ?>
```

Group operation example: subgroupMax()

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  id = subgroupMax(id);
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  ?,  ?,  ?,  ?>
sum = <  ?,  ?,  ?,  ?>
```

Group operation example: subgroupMax()

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  id = subgroupMax(id);
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  3,  3,  3,  3>
nid = <  ?,  ?,  ?,  ?>
sum = <  ?,  ?,  ?,  ?>
```

Group operation example: subgroupMax()

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  id = subgroupMax(id);
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  3,  3,  3,  3>
nid = <  0,  0,  0,  0>
sum = <  ?,  ?,  ?,  ?>
```

Group operation example: subgroupMax()

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  id = subgroupMax(id);
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [  ?,  ?,  ?,  ?]
```

Local Variables

```
id  = <  3,  3,  3,  3>
nid = <  0,  0,  0,  0>
sum = < 15, 15, 15, 15>
```

Group operation example: subgroupMax()

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  id = subgroupMax(id);
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [ 15, 15, 15, 15]
```

Local Variables

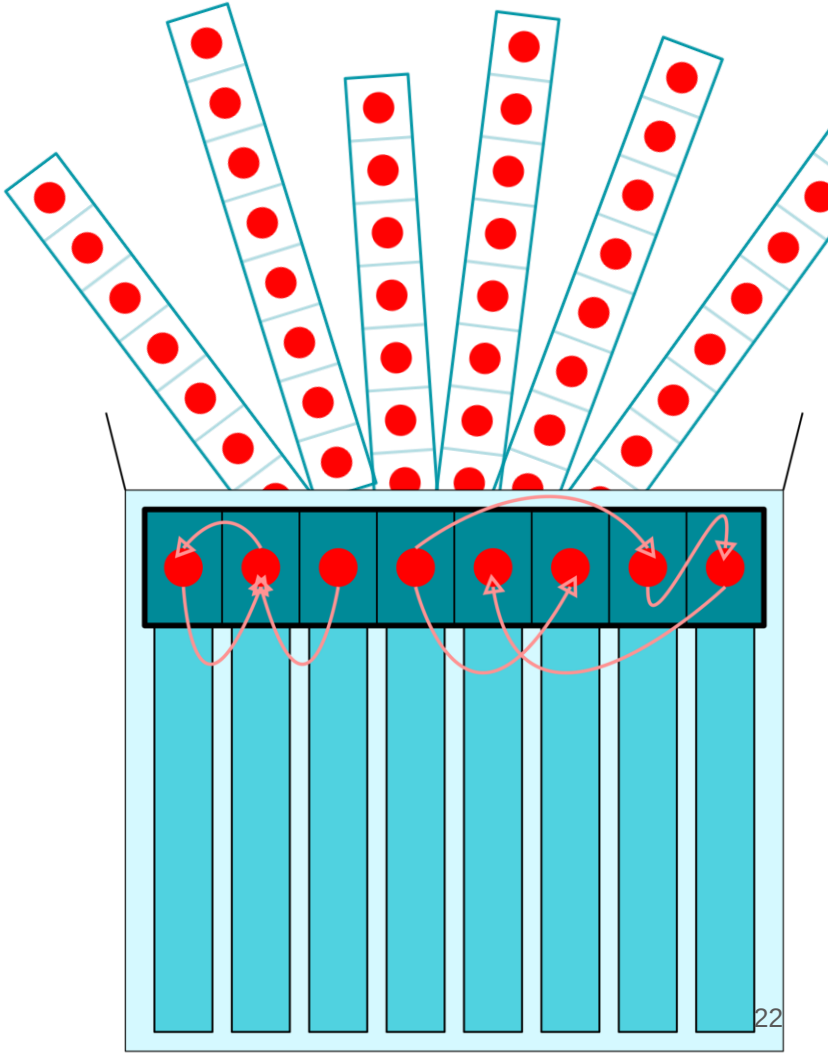
```
id  = <  3,  3,  3,  3>
nid = <  0,  0,  0,  0>
sum = < 15, 15, 15, 15>
```

Group Operations

- Invocations can use Group Operations to exchange data efficiently within their Subgroup.

That is the group operation in non-uniform group operations !

- But what does it mean for something to be uniform ?



What is uniformity really ?

Part 1: Variable uniformity

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = subgroupMax(id + 1) % N;
  int sum = src[id] + src[nid];
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [ 80, 17, 82, 15]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  0,  0,  0,  0>
sum = < 80, 17, 82, 15>
```

Some variables have the same value for all invocations in the subgroup

They are subgroup-uniform

What is uniformity really ?

Part 2: Control-flow uniformity

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 2) % N;
  int sum = src[id] + src[nid];
  if (sum < 0)
    sum = 0;
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [ 80, 17, 82,  0]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  ?,  ?,  ?,  ?>
sum = <  ?,  ?,  ?,  ?>
```

What is uniformity really ?

Part 2: Control-flow uniformity

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 2) % N;
  int sum = src[id] + src[nid];
  if (sum < 0)
    sum = 0;
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [ 80, 17, 82,  0]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  2,  3,  0,  1>
sum = <  ?,  ?,  ?,  ?>
```

What is uniformity really ?

Part 2: Control-flow uniformity

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 2) % N;
  int sum = src[id] + src[nid];
  if (sum < 0)
    sum = 0;
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [ 80, 17, 82,  0]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  2,  3,  0,  1>
sum = <107,-23,107,-23>
```

What is uniformity really ?

Part 2: Control-flow uniformity

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 2) % N;
  int sum = src[id] + src[nid];
  if (sum < 0)
    sum = 0;
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [ 80, 17, 82,  0]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  2,  3,  0,  1>
sum = <107,-23,107,-23>
```

Something weird happens next …

What is uniformity really ?

Part 2: Control-flow uniformity

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 2) % N;
  int sum = src[id] + src[nid];
  if (sum < 0)
    sum = 0;
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [ 80, 17, 82,  0]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  2,  3,  0,  1>
sum = <107,  0,107,  0>
```

```
active_invocations ::= <0, 1, 0, 1>
```

This code only runs for a **subset** of the subgroup !
This is known as divergence, or non-uniform control-flow.

What is uniformity really ?

Part 2: Control-flow uniformity

```
#define N 4
layout(scalar, set = 0, binding = 1)
buffer src { int array[N]; };
layout(scalar, set = 0, binding = 1)
buffer dst { int array[N]; };

void main() {
  int id = gl_SubgroupInvocationID.x;
  int nid = (id + 1) % N;
  int sum = src[id] + src[nid];
  if (sum < 0)
    sum = 0;
  dst[id] = sum;
}
```

Global Variables

```
src = [ 40,  2, 67,-25]
dst = [107,  0,107,  0]
```

Local Variables

```
id  = <  0,  1,  2,  3>
nid = <  2,  3,  0,  1>
sum = <107,  0,107,  0>
```

We return to uniform control-flow.

```
active_invocations ::= <1, 1, 1, 1>
```

This is known as reconvergence.

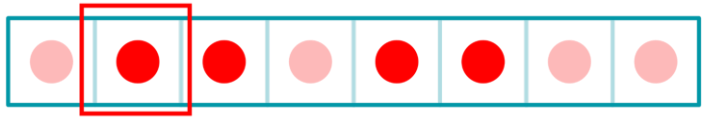
`Uniform`-ity summarized

- A variable is S-uniform for a scope S if, for all active invocations they stored the same value
  - Example: `x = <0, 0, 0, 0>` is a subgroup-uniform value for a subgroup size of 4
- A tangle is a set of invocations executing together as one
- Execution always starts out executing with subgroup-uniform control flow.
  - The initial tangle is the set of all invocations in the subgroup.
- Control-flow is uniform for a scope S if there is only one tangle
- Nested scopes lead to transitivity:
  - If something is workgroup-uniform, it is also subgroup-uniform
- ⚠ In graphics stages, draw-uniform does not imply subgroup-uniform
  - Different draws can legally be packed into a single subgroup

Non-uniform Group Operations

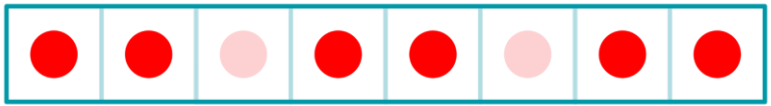
- What happens when a group operation executes in non-uniform control flow ?
- Only the active invocations participate
- Inactive invocations…
  - Cannot be elected
  - Always return false when voting
  - Are skipped entirely for reductions
  - Return undefined values for shuffles

electFirst()



<0,1,0,0,0,0,0,0>

subgroupAdd()

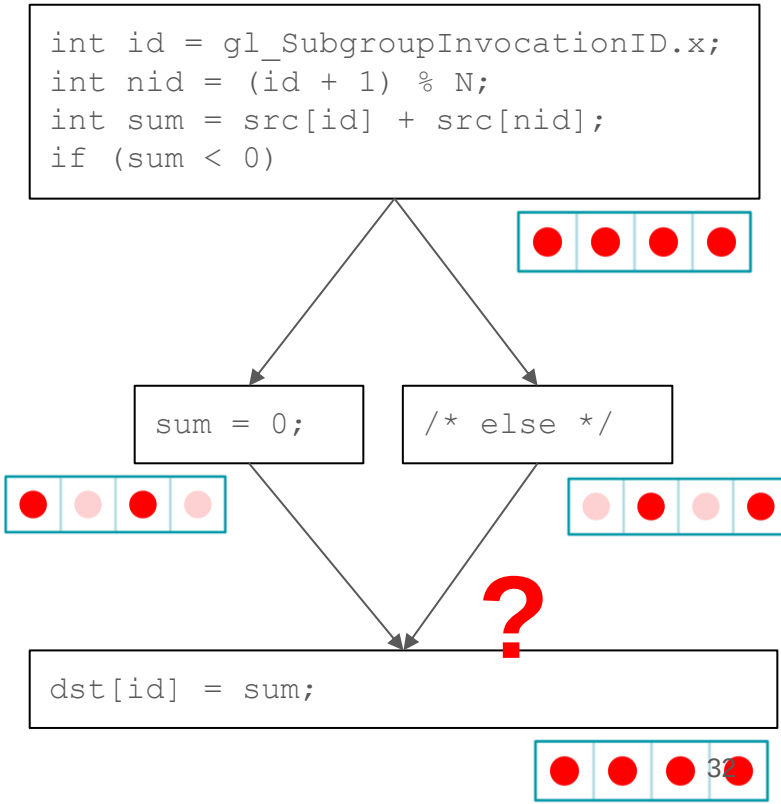


< 8, 2, ?,10,-5, ?, 2, 0>

<17,17,17,17,17,17,17,17>

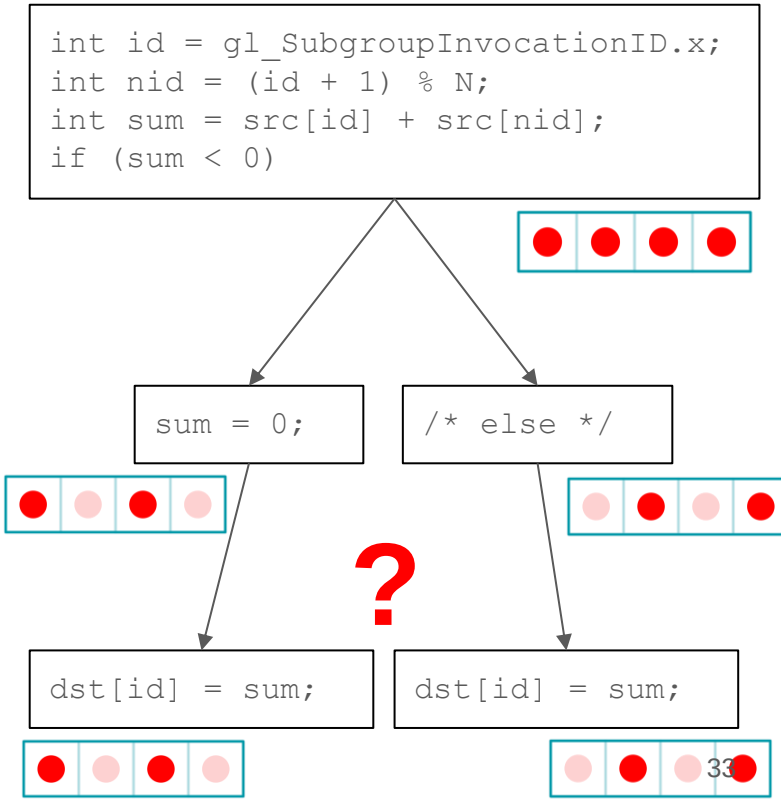
Divergence and Reconvergence

- Execution always starts out executing with subgroup-uniform control flow.
  - The initial tangle is the set of all invocations in the subgroup.
- Divergence happens when there is a branch with a non-uniform condition
  - The tangle gets split into different sets, based on the branch destination
- Reconvergence happens at after a divergent branch.
  - The two or more tangles get merged
- Precisely when reconvergence happens is typically left undefined.
  - Including Vulkan up to this point …



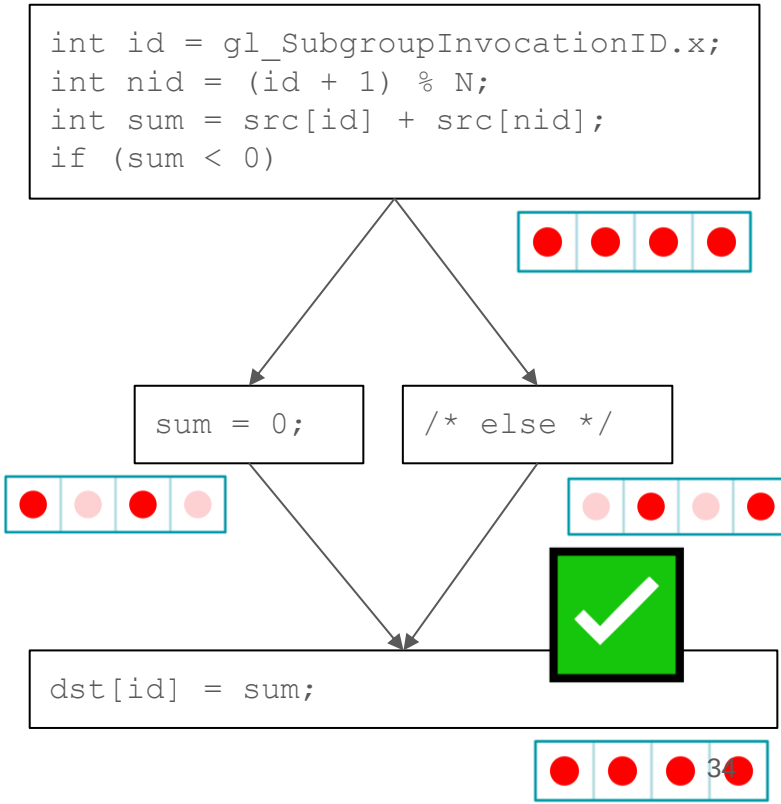
Divergence and Reconvergence

- Execution always starts out executing with subgroup-uniform control flow.
  - The initial tangle is the set of all invocations in the subgroup.
- Divergence happens when there is a branch with a non-uniform condition
  - The tangle gets split into different sets, based on the branch destination
- Reconvergence happens at after a divergent branch.
  - The two or more tangles get merged
- Precisely when reconvergence happens is typically left undefined.
  - Including Vulkan up to this point …



Divergence and Reconvergence

- Divergence happens when there is a branch with a non-uniform condition
  - The tangle gets split into different sets, based on the branch destination
- Reconvergence happens at after a divergent branch.
  - The two or more tangles get merged
- Precisely when reconvergence happens is typically left undefined.
  - Including Vulkan up to this point …
- **Maximal Reconvergence gives a robust answer to this question !**

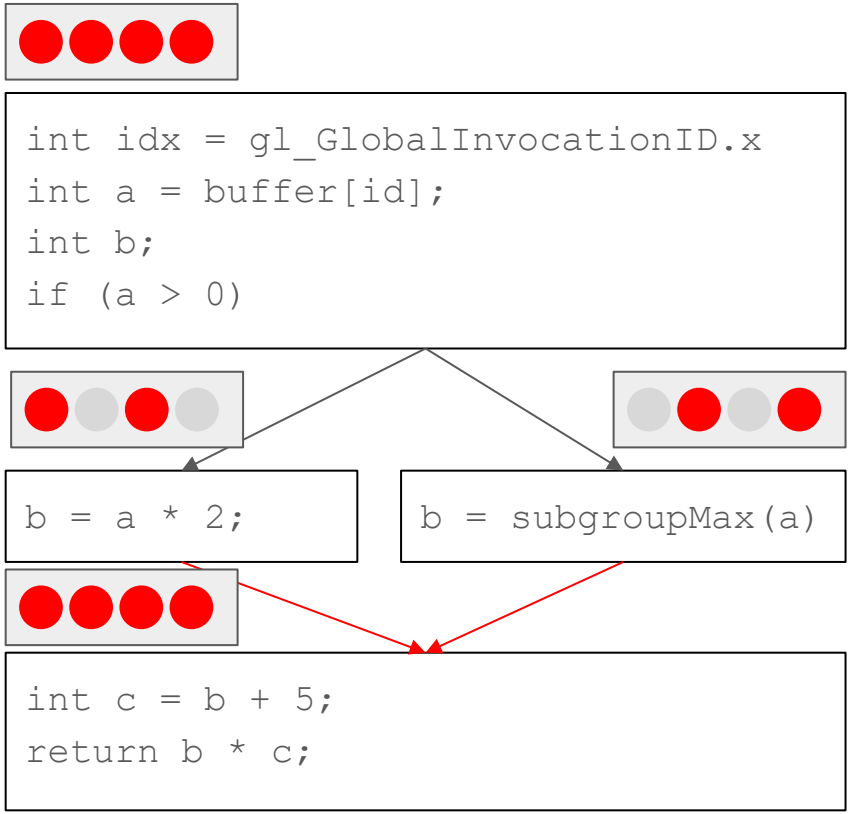


So what are the maximal reconvergence rules ?

- They are defined with regards to structured control flow constructs:
  - If / switch
  - For / while
  - Return
  - Break / continue
- We diverge when we enter a level of nesting, and we reconverge when we exit to an outer level of nesting.
  - They follow programmer intuition
- Wherever possible, we reconverge such that the tangle is as big as possible, to maximise efficiency.
  - That’s why it’s called *maximal* reconvergence.

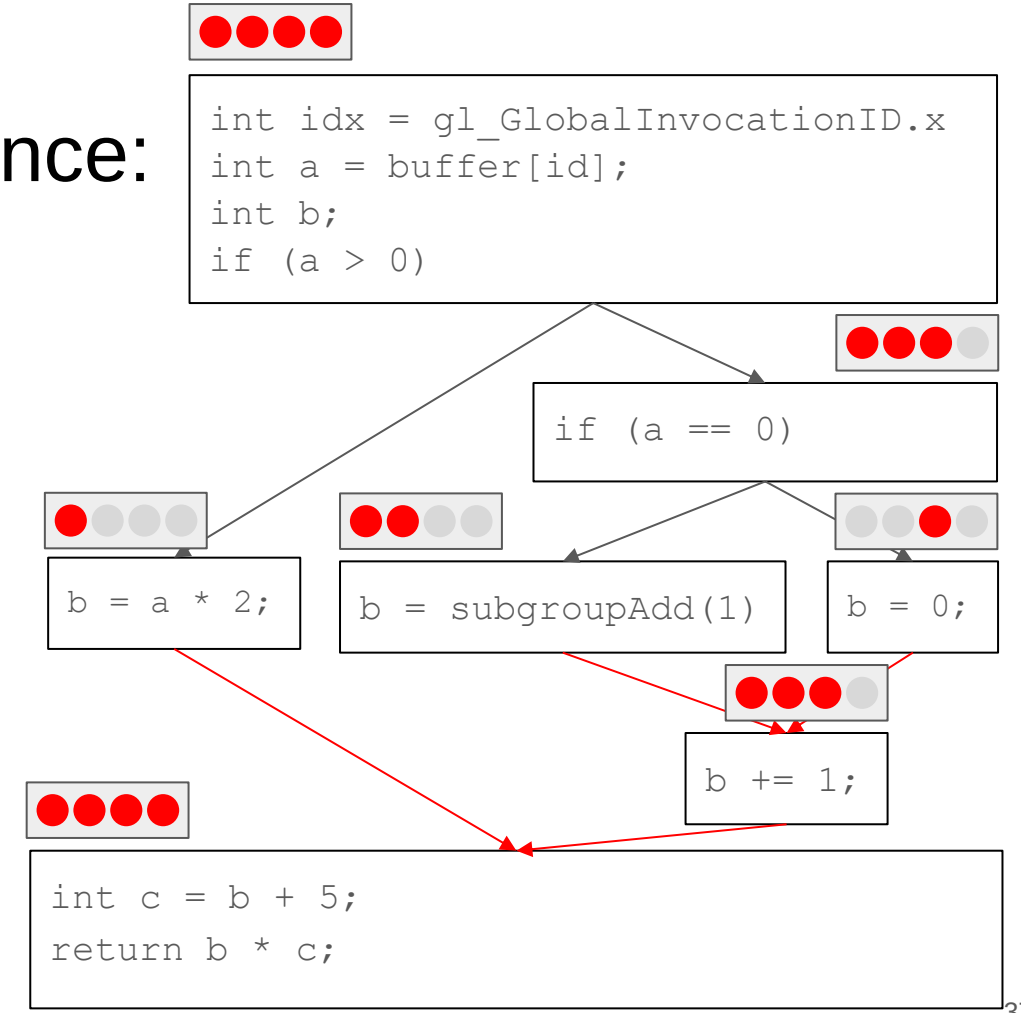
Maximal Reconvergence: `If` conditionals

```
int idx = gl_GlobalInvocationID.x
int a = buffer[id];
int b;
if (a > 0)
  b = a * 2;
else
  b = subgroupMax(a);
int c = b + 5;
return b * c;
```



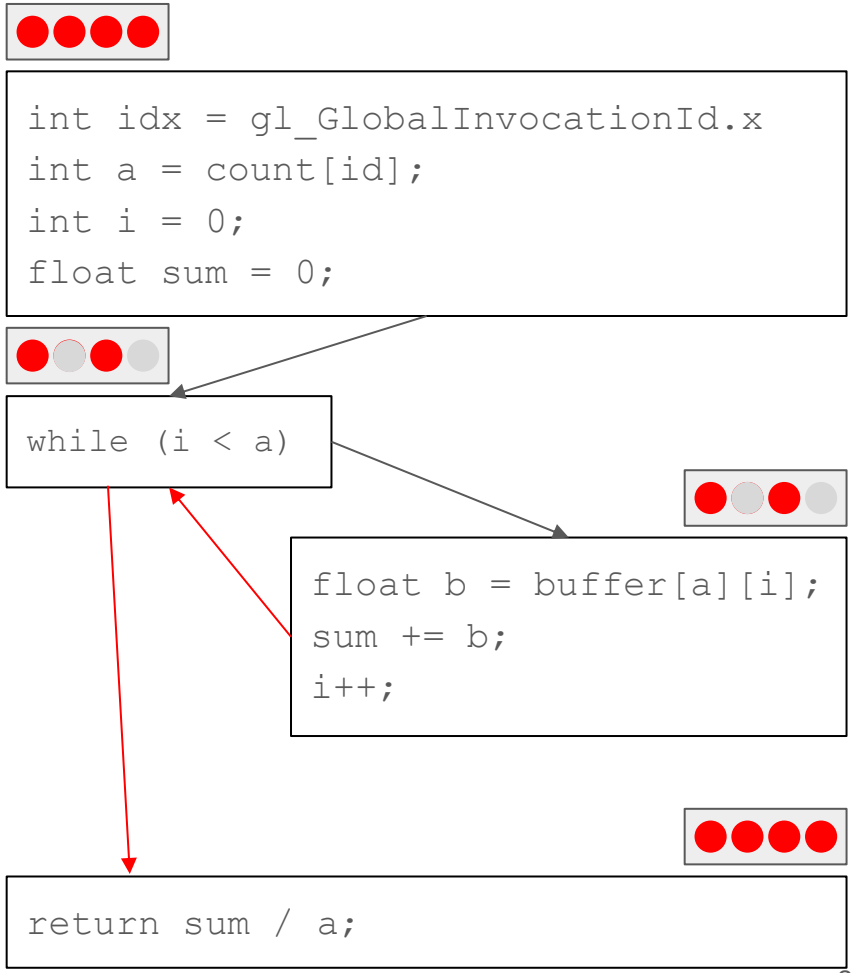
Maximal Reconvergence: Nested `If`s

```
int idx = gl_GlobalInvocationID.x
int a = buffer[id];
int b;
if (a > 0)
  b = a * 2;
else
  if (a == 0)
    b = subgroupAdd(1);
  else
    b = 0;
  b += 1;
int c = b + 5;
return b * c;
```



Maximal Reconvergence: `While` and `For` loops

```
int idx = gl_GlobalInvocationId.x
int a = count[id];
int i = 0;
float sum = 0;
while (i < a)
  float b = buffer[a][i];
  sum += b;
  i++;
return sum / a;
```



Application 1: Efficiently appending to a buffer

```
layout(scalar, set = 0, binding = 0)
buffer dst { int size; int array[N]; };

void main() {
  int data = ...
  int slot = atomicAdd(dst.size, 1);
  dst.array[slot] = data;
}
```

Straightforward: we increment the destination size atomically and use the old size to know where to write the data

```
layout(scalar, set = 0, binding = 0)
buffer dst { int size; int array[N]; };

void main() {
  int data = ...
  int count = subgroupBallotBitCount(true);
  int index = subgroupExclusiveAdd(1);
  int base;
  if (subgroupElect()) /* diverge here */ {
          base_slot = atomicAdd(dst.size,
count);
  }
  // reconverge here
  base = subgroupBroadcastFirst(base_slot);
  int slot = base + index;
  dst.array[slot] = data;
}
```

While we do more work within the subgroup, we only do one atomic add in the end

Application 2: Scalarization loop

```
uniform sampler2D textures[];

flat in int texId;
out vec4 fragColor;

void main() {
  while (true) {
    int elected = subgroupBroadcastFirst(texId);
    if (elected == textureId) {
      fragColor = texture(textures[texId]);
      break;
    }
  }
}
```

In this pattern, we iterate over the *unique values* of a non-uniform variable, with only the threads that have the same value active, making that value (textureId) dynamically subgroup-uniform !

textures[texId]:

Iteration 0

Iteration 1

Iteration 2

Why does any of this matter ?

In Theory…

- Non-Uniform Group Operations let you observe divergence …
- If reconvergence is undefined, they’re undefined too !

In practice…

- Simple examples will often happen to work, even without MR
  - But complex ones will blow up in your face when you least expect it
- Using certain operations with non-uniform data leads to UB/crashes
- This isn’t just a performance concern, but a **correctness** one

The Killer Feature You Never Heard About

- In Summary:

  “Subgroups now work the way you thought they did”

  Tom Olson, Vulkanized 2024

- Behaviour follows code structure
- Allowing you to write collaborative algorithms in a natural way
- Compiler optimizations do not get in the way
- Belongs on the shortlist of why Vulkan is Awesome
  - Alongside `buffer_device_address` and `descriptorIndexing`

The Killer Feature You Never Heard About

- OpenCL: Doesn’t have non-uniform group ops.
- OpenGL: You’re at the mercy of the compiler.
- DirectX: You’re at the mercy of the compiler.
- Metal: You’re at the mercy of the compiler.
- CUDA: Works under a different model.
  - You have to manually carry a mask that represents invocations in the subgroup you communicate with

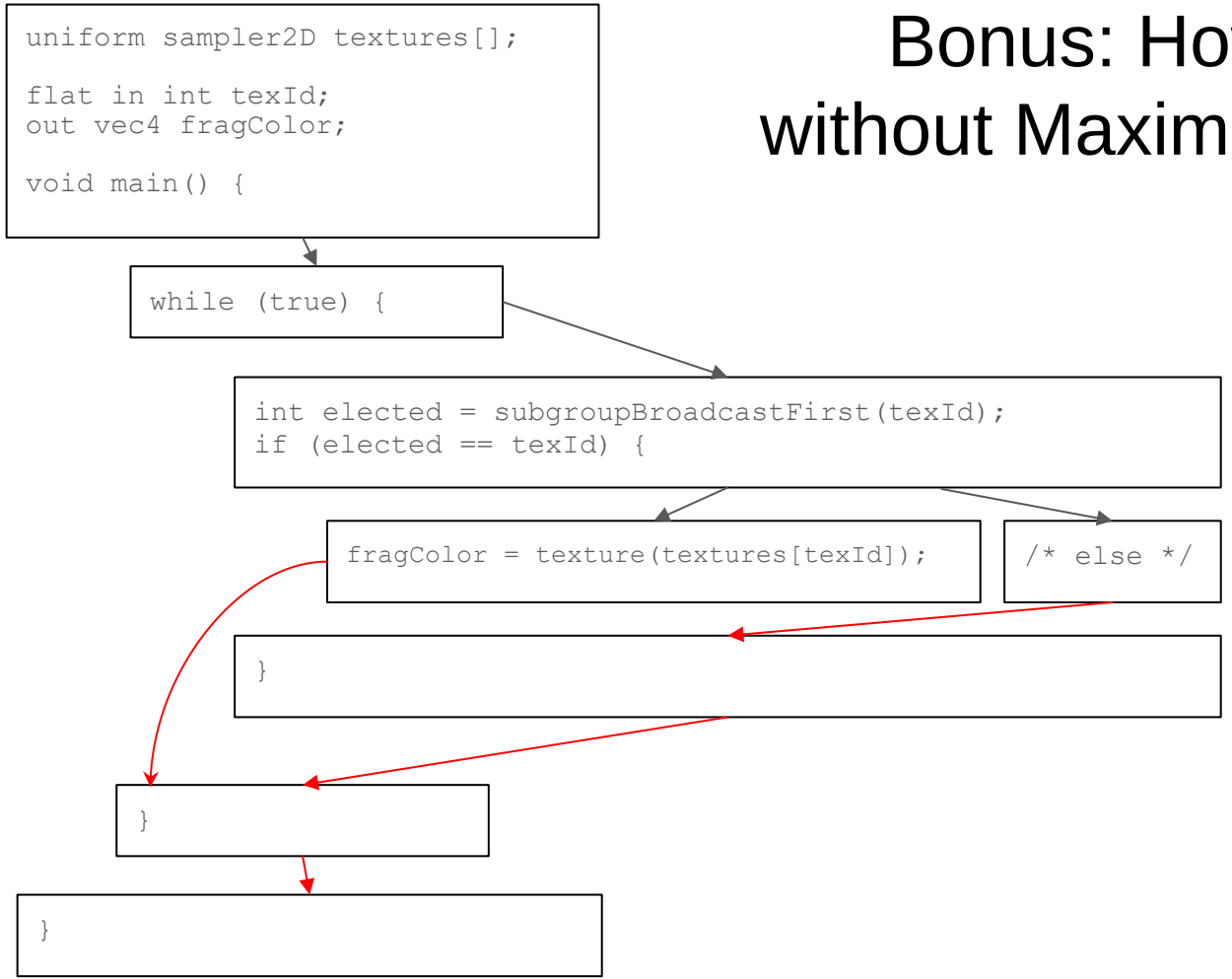
Vulkan is the **only API** with this feature today !

References & Suggested Reading

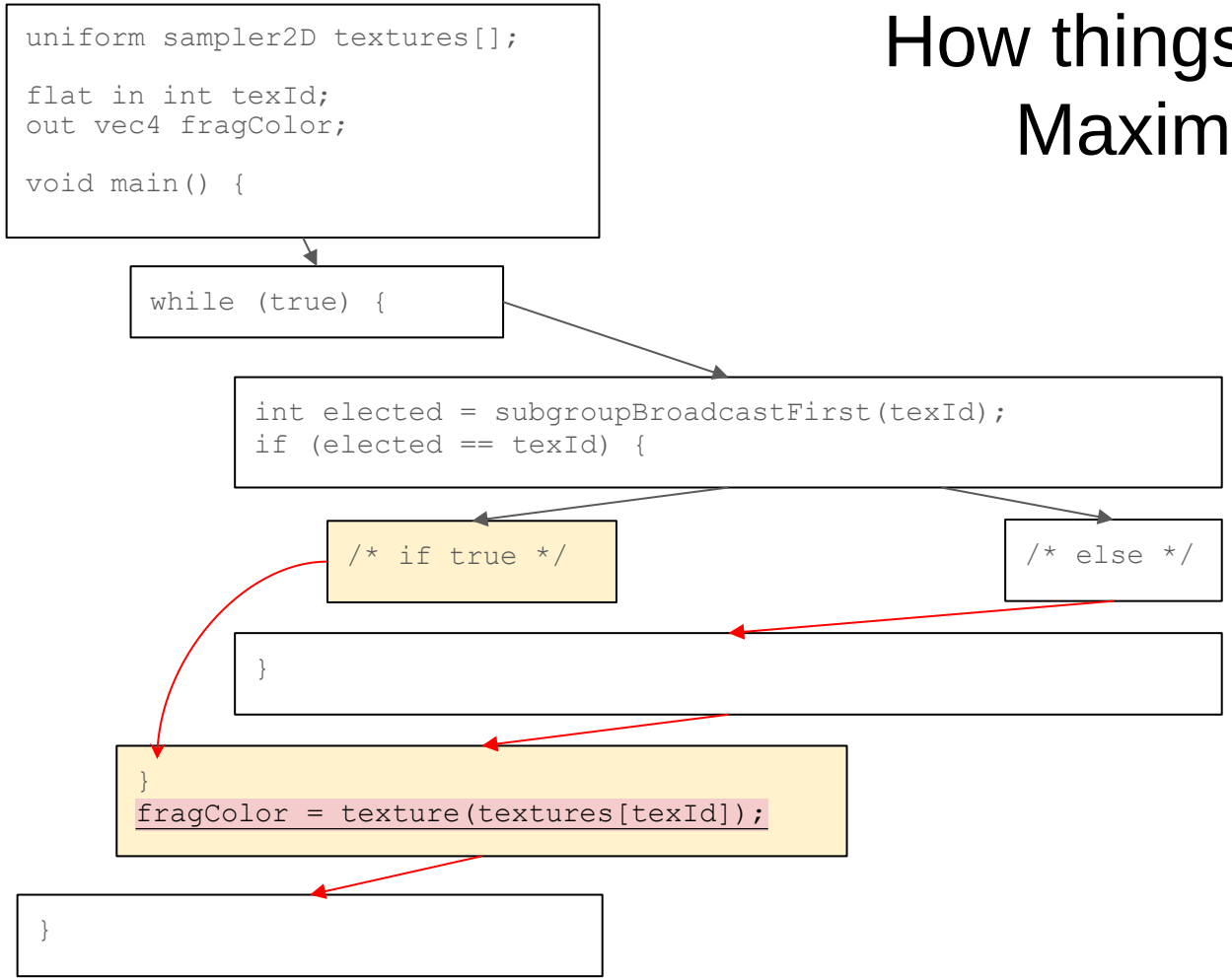
- Alan Baker's blog post https://www.khronos.org/blog/khronos-releases-maximal-reconvergence-and-quad-control-extensions-for-vulkan-and-spir-v
- Official SPIR-V extension specification https://github.com/KhronosGroup/SPIRV-Registry/blob/main/extensions/KHR/SPV_KHR_maximal_reconvergence.asciidoc
- Faith Ekstrand's blog post on implementing MR on NVK https://www.collabora.com/news-and-blog/blog/2024/04/25/re-converging-control-flow-on-nvidia-gpus/

Questions Time

Bonus: How things go wrong without Maximal Reconvergence



How things go wrong without Maximal Reconvergence



- Without maximal reconvergence, the compiler moves this texture sampling outside the loop !
- Which defeats the point: we sample at the end with a non-uniform `texId`