Vulkan Subgroup Tutorial

March 13, 2018 vulkan (/news/tags/tag/vulkan), glsl (/news/tags/tag/glsl)

Subgroups are an important new feature in Vulkan 1.1 because they enable highly-efficient sharing and manipulation of data between multiple tasks running in parallel on a GPU. In this tutorial, we will cover how to use the new subgroup functionality.

Modern heterogeneous hardware like GPUs gain performance by using parallel hardware and exposing a parallel programming model to target this hardware. When a user wants to run `N` parallel tasks for their algorithm, a GPU would divide this `N`-sized workload between the compute units of that GPU. Each compute unit of the GPU is then capable of running one or more of these parallel tasks concurrently. In Vulkan, we refer to the data that runs on a single compute unit of a GPU as the *local workgroup*, and an individual parallel task as an *invocation*.

Vulkan 1.0 already exposes a method to share data between the invocations in a local workgroup via *shared memory*, which is exposed only in compute shaders. Shared memory allows for invocations within the local workgroup to share some data via memory that is faster to access than reading and writing to buffer memory, providing a mechanism to share data in a performance sensitive context.

Vulkan 1.1 goes further and introduces a mechanism to share data between the invocations that run in parallel on a single compute unit. These concurrently running invocations are named the *subgroup*. This subgroup allows for the sharing of data between a much smaller set of invocations than the local workgroup could, but at a significantly higher performance.

While shared memory is only available in compute shaders, sharing data via subgroup operations is allowed in all shader stages via optionally supported stages as we'll explain below.

How to Query for Subgroup Support

In Vulkan 1.1, a new structure has been added for querying the subgroup support of a physical device:

```
struct VkPhysicalDeviceSubgroupProperties {
  VkStructureType        sType;
  void*                  pNext;
  uint32_t               subgroupSize;
  VkShaderStageFlags     supportedStages;
  VkSubgroupFeatureFlags supportedOperations;
  VkBool32               quadOperationsInAllStages;
};
```

To get the subgroup properties of a physical device:

```
VkPhysicalDevice physicalDevice = ...; // A previously retrieved physical device

VkPhysicalDeviceSubgroupProperties subgroupProperties;
subgroupProperties.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIE
S;
subgroupProperties.pNext = NULL;

VkPhysicalDeviceProperties2 physicalDeviceProperties;
physicalDeviceProperties.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;
physicalDeviceProperties.pNext = &subgroupProperties;

vkGetPhysicalDeviceProperties2(physicalDevice, &physicalDeviceProperties);
```

The fields of `VkPhysicalDeviceSubgroupProperties` are:

- `subgroupSize` - how many invocations are in a single subgroup of this device.
- `supportedStages` - which shader stages support subgroup operations.
- `supportedOperations` - which subgroup operations are supported.
- `quadOperationsInAllStages` - if `supportedOperations` contains `VK_SUBGROUP_FEATURE_QUAD_BIT`, do the quad operations work in all `supportedStages` or only in fragment and compute stages.

There are some minimal guarantees that all Vulkan 1.1 physical devices must support:

- `subgroupSize` must be at least 1.
- `supportedStages` must include the `VK_SHADER_STAGE_COMPUTE_BIT` stage - all compute shaders can use subgroup functionality on all Vulkan 1.1 devices.
- `supportedOperations` must include `VK_SUBGROUP_FEATURE_BASIC_BIT` operations, all other categories are optional.

Subgroup operations are pretty useless with a `subgroupSize` of 1, but it does mandate that a shader that uses subgroup functionality **must** be consumable by all Vulkan 1.1 drivers - which is good news for our users who don't want to ship more than one shader to take advantage of the new functionality!

How Subgroups are Formed on Hardware

Subgroups have some characteristics that are true on all hardware that supports the functionality.

Invocations within a subgroup can be active or inactive. An invocation is active if it is, for want of a better word, *active* within the subgroup - EG. it is doing actual calculations or memory accesses. An invocation is inactive if the opposite is true - for one reason or another, the invocation is not doing anything useful.

So what cases could cause an invocation to be inactive?

Small WorkgroupSize

Let us assume we've got a device that has a `subgroupSize` that is greater than 1. Now, let's have a look at the following compute shader:

```
#version 450

layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

void main() {
    // Do work!
}
```

In the above case, we are **guaranteed** to have inactive invocations in every subgroup. Why? you ask: If you specify a workgroup size that is less than the subgroup size, you are guaranteed to have inactive invocations within the subgroup.

Now, remembering that a subgroup is a way to expose a set of concurrently running invocations, we've basically committed ourselves to be underutilizing the hardware in the above example!

If the `subgroupSize` of the device was 2, we're executing at 50% capacity, 4 we're executing at 25% capacity, on NVIDIA (which has a `subgroupSize` of 32) we're executing at 3.1% capacity, and on AMD (which has a `subgroupSize` of 64) we're executing at 1.6% capacity!

Even if you don't use any other part of the new subgroup functionality, you should aim to make your local workgroup be at least the size of the subgroup in most situations.

Dynamic Branching

Lets take the following snippet of a branch:

```
float x = ...; // Some previously set variable
if (x < 0.0f) {
    x = x * -1.0f + 42.0f;
} else {
    x = x - 13.0f;
}
```

In the above example, if some invocations in the subgroup have an `x` with a value less than 0, and some have a value of `x` that is **not** less than 0, some invocations will enter the if branch, and others the else branch. Within the if branch, all invocations that had a value of x that is not less than 0 will be inactive. Likewise, in the else branch, all invocations that had a value of x that is less than 0 will inactive.

Active Invocations are Happy Invocations

From the information provided above, I hope it's obvious that keeping invocations active, and thus doing *actual* work, is the key to keeping your GPU happy.

GL_KHR_shader_subgroup

Alongside the Vulkan 1.1 core subgroup functionality, a new GLSL extension *GL_KHR_shader_subgroup* which is available here (https://github.com/KhronosGroup/GLSL/blob/master/extensions/khr/GL_KHR_shader_subgroup.txt), and is usable in glslang (https://github.com/KhronosGroup/glslang/pull/1277) too.

GL_KHR_shader_subgroup exposes new subgroup built-in variables and new subgroup built-in functions. These functions are separated into categories that match the `supportedOperations` enumeration of Vulkan 1.1.

Each category has a corresponding extension to enable. Enabling any category other than `GL_KHR_shader_subgroup_basic` will implicitly enable `GL_KHR_shader_subgroup_basic` also.

We'll use the generic typename `T` to denote `bool`, `bvec2`, `bvec3`, `bvec4`, `int`, `ivec2`, `ivec3`, `ivec4`, `uint`, `uvec2`, `uvec3`, `uvec4`, `float`, `vec2`, `vec3`, `vec4`, `double`, `dvec2`, `dvec3`, and `dvec4` types.

#extension GL_KHR_shader_subgroup_basic

The first category is `GL_KHR_shader_subgroup_basic`. The basic category introduces the built-in subgroup variables, and a few built-in functions too.

In the compute shader stage only:

- `gl_NumSubgroups` is the number of subgroups within the local workgroup.
- `gl_SubgroupID` is the ID of the subgroup within the local workgroup, an integer in the range [`0` .. `gl_NumSubgroups`).

In all supported stages:

- `gl_SubgroupSize` is the size of the subgroup, which matches the `subgroupSize` field of `VkPhysicalDeviceSubgroupProperties` mentioned previously.
- `gl_SubgroupInvocationID` is the ID of the invocation within the subgroup, an integer in the range [`0` .. `gl_SubgroupSize`).
- `gl_SubgroupEqMask`, `gl_SubgroupGeMask`, `gl_SubgroupGtMask`, `gl_SubgroupLeMask`, and `gl_SubgroupLtMask` are variables that can be used in conjunction with a subgroupBallot result.

The basic category also introduces the various barrier functions to control execution and memory accesses across the subgroup:

- `void subgroupBarrier()` performs a full memory and execution barrier - basically when an invocation returns from `subgroupBarrier()` we are guaranteed that every invocation executed the barrier before any return, and all memory writes by those invocations are visible to all invocations in the subgroup.
- `void subgroupMemoryBarrier()` performs a memory barrier on all memory types (buffers, images, and shared memory). A memory barrier enforces that the ordering of memory operations by a single invocation as seen by other invocations is the same. For example, if I wrote `42` to the 0'th element of a buffer, called `subgroupMemoryBarrier()`, then wrote `13` to the 0'th element of the same buffer, no other invocation would read a `42` after previously reading `13`.
- `void subgroupMemoryBarrierBuffer()` performs a memory barrier on just the buffer variables accessed.
- `void subgroupMemoryBarrierShared()` performs a memory barrier on just the shared variables accessed.

- `void subgroupMemoryBarrierImage()` performs a memory barrier on just the image variables accessed.
- `bool subgroupElect()` - exactly one invocation within the subgroup will return true, the others will return false. The invocation that returns true is always the one that is active with the lowest `gl_SubgroupInvocationID`.

The basic category is the building block for the other categories. In isolation, most of what we could do with the basic category could be done with the local workgroup synchronization primitives that GLSL already has, but when we later combine it with other functionality the power of the functionality will become clear.

#extension GL_KHR_shader_subgroup_vote

The vote category introduces built-in functions that allow invocations to vote on whether boolean conditions were met across the subgroup:

- `bool subgroupAll(bool value)` - returns true if all active invocations have `value == true`.
- `bool subgroupAny(bool value)` - returns true if any active invocation has `value == true`.
- `bool subgroupAllEqual(T value)` - returns true if all active invocations have a `value` that is equal.

These built-ins are a superset of the ones previously provided in ARB_shader_group_vote (https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_group_vote.txt).

The vote category is *seriously* useful in code that has branching. Let's say we have the following shader:

```
#version 450

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout(std430, set=0, binding=0) buffer layout_foo {
  int foo[];
};

layout(std430, set=0, binding=1) buffer layout_bar {
  int bar[];
};

void main() {
  if (foo[gl_GlobalInvocationID.x] < bar[gl_GlobalInvocationID.x]) {
    // x
  } else {
    // y
  }
}
```

And let's assume that you *know* that 80% of the time the invocations will all either enter the if statement and execute x, or enter the else statement and execute y. We can now tell the compiler this information and thus allow it to optimize the code by:

```
#version 450

#extension GL_KHR_shader_subgroup_vote: enable

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout(std430, set=0, binding=0) buffer layout_foo {
  int foo[];
};

layout(std430, set=0, binding=1) buffer layout_bar {
  int bar[];
};

void main() {
  bool condition = foo[gl_GlobalInvocationID.x] < bar[gl_GlobalInvocationID.x];

  if (subgroupAll(condition)) {
    // all invocations in the subgroup are performing x
  } else if (!subgroupAny(condition)) {
    // all invocations in the subgroup are performing y
  } else {
    // Invocations that get here are doing a mix of x & y so have a fallback
  }
}
```

#extension GL_KHR_shader_subgroup_ballot

The ballot category introduces built-in functions that allow invocations to do limited sharing of data across the invocations of a subgroup:

- `T subgroupBroadcast(T value, uint id)` broadcasts the `value` whose `gl_SubgroupInvocationID == id` to all other invocations (id must be a compile time constant).
- `T subgroupBroadcastFirst(T value)` broadcasts the `value` whose `gl_SubgroupInvocationID` is the lowest active to all other invocations.
- `uvec4 subgroupBallot(bool value)` each invocation contributes a single bit to the resulting uvec4 correponding to `value`.

- `bool subgroupInverseBallot(uvec4 value)` returns true if this invocations bit in `value` is true.
- `bool subgroupBallotBitExtract(uvec4 value, uint index)` returns true if the bit corresponding to `index` is set in `value` .
- `uint subgroupBallotBitCount(uvec4 value)` returns the number of bits set in `value` , only counting the bottom `gl_SubgroupSize` bits.
- `uint subgroupBallotInclusiveBitCount(uvec4 value)` returns the inclusive scan of the number of bits set in `value` , only counting the bottom `gl_SubgroupSize` bits (we'll cover what an inclusive scan is later).
- `uint subgroupBallotExclusiveBitCount(uvec4 value)` returns the exclusive scan of the number of bits set in `value` , only counting the bottom `gl_SubgroupSize` bits (we'll cover what an exclusive scan is later).
- `uint subgroupBallotFindLSB(uvec4 value)` returns the lowest bit set in `value` , only counting the bottom `gl_SubgroupSize` bits.
- `uint subgroupBallotFindMSB(uvec4 value)` returns the highest bit set in `value` , only counting the bottom `gl_SubgroupSize` bits.

These built-ins are a superset of the ones previously provided in ARB_shader_ballot (https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_ballot.txt).

The ballot category has two groups of functionality - the ability to send a value from one invocation to the others within a subgroup, and a more powerful form of voting in the form of the `ballot` built-in.

Let's take our previous example, and let's say you want to only perform x if at least a quarter of the invocations want to perform x. In other words, most of the time you are fine with performing y, but if enough say otherwise you'll reluctantly run x. Let's make use of our ballot built-ins for this:

```
#version 450

#extension GL_KHR_shader_subgroup_ballot: enable

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout(std430, set=0, binding=0) buffer layout_foo {
  int foo[];
};

layout(std430, set=0, binding=1) buffer layout_bar {
  int bar[];
};

void main() {
  uvec4 ballot = subgroupBallot(foo[gl_GlobalInvocationID.x] < bar[gl_GlobalInvo
cationID.x]);

  if (gl_SubgroupSize <= (subgroupBallotBitCount(ballot) * 4)) {
    // all invocations in the subgroup are performing x
  } else {
    // all invocations in the subgroup are performing y
  }
}
```

Another example, lets say you want to load a value such that the value is the same across the entire subgroup:

```
#version 450

#extension GL_KHR_shader_subgroup_ballot: enable

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout(std430, set=0, binding=0) buffer layout_foo {
  vec4 foo[];
};

void main() {
  vec4 value;

  if (subgroupElect()) {
    uint index; // some complicated index
    value = foo[index];

    // could even do some complicated math on value
  }

  // Tell everyone else in the subgroup the value
  value = subgroupBroadcastFirst(value);

  // Every invocation in the subgroup now has the same value
}
```

#extension GL_KHR_shader_subgroup_arithmetic

The arithmetic category introduces built-in functions that allow invocations to perform some simple operations across the invocations of a subgroup:

- `T subgroupAdd(T value)` returns the summation of all active invocations `value` 's across the subgroup.
- `T subgroupMul(T value)` returns the multiplication of all active invocations `value` 's across the subgroup.
- `T subgroupMin(T value)` returns the minimum `value` of all active invocations `value` 's across the subgroup.
- `T subgroupMax(T value)` returns the maximum `value` of all active invocations `value` 's across the subgroup.

- `T subgroupAnd(T value)` returns the binary and of all active invocations `value` 's across the subgroup.
- `T subgroupOr(T value)` returns the binary or of all active invocations `value` 's across the subgroup.
- `T subgroupXor(T value)` returns the binary xor of all active invocations `value` 's across the subgroup.

These operations perform what is termed a *reduction* operation - each subgroup invocation takes a number of `value` 's and performs an operation on them. Two other sets of operations that are supported are inclusive and exclusive scan. To understand scan operations, let's look at an approach that is not optimal to implementing an add-scan:

```
#version 450

#extension GL_KHR_shader_subgroup_ballot: enable

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout(std430, set=0, binding=0) buffer layout_foo {
  float foo[];
};

void main() {
  uint foo_index = subgroupBroadcastFirst(gl_GlobalInvocationID.x);
  float value = 0;

  for (uint i = 0; i < gl_SubgroupSize; i++) {
#ifdef INCLUSIVE_SCAN
    if (i == (gl_SubgroupInvocationID + 1)) {
      break;
    }
#else//EXCLUSIVE_SCAN
    if (i == gl_SubgroupInvocationID) {
      break;
    }
#endif

    value += foo[foo_index + i];
  }
}
```

In the example above, we've implemented a scan operation using a loop. What in effect happens, is that invocations will only perform the requested operation with active invocations in the subgroup that have a `gl_SubgroupInvocationID` that is less than themselves. The difference between the inclusive and exclusive variants is that the inclusive variants will use the value they provide as part of their own result, whereas exclusive variants do not use the value they provide to calculate their own results.

- `T subgroupInclusiveAdd(T value)` returns the inclusive scan summation of all active invocations `value` 's across the subgroup.
- `T subgroupInclusiveMul(T value)` returns the inclusive scan the multiplication of all active invocations `value` 's across the subgroup.
- `T subgroupInclusiveMin(T value)` returns the inclusive scan the minimum `value` of all active invocations `value` 's across the subgroup.
- `T subgroupInclusiveMax(T value)` returns the inclusive scan the maximum `value` of all active invocations `value` 's across the subgroup.
- `T subgroupInclusiveAnd(T value)` returns the inclusive scan the binary and of all active invocations `value` 's across the subgroup.
- `T subgroupInclusiveOr(T value)` returns the inclusive scan the binary or of all active invocations `value` 's across the subgroup.
- `T subgroupInclusiveXor(T value)` returns the inclusive scan the binary xor of all active invocations `value` 's across the subgroup.
- `T subgroupExclusiveAdd(T value)` returns the exclusive scan summation of all active invocations `value` 's across the subgroup.
- `T subgroupExclusiveMul(T value)` returns the exclusive scan the multiplication of all active invocations `value` 's across the subgroup.
- `T subgroupExclusiveMin(T value)` returns the exclusive scan the minimum `value` of all active invocations `value` 's across the subgroup.
- `T subgroupExclusiveMax(T value)` returns the exclusive scan the maximum `value` of all active invocations `value` 's across the subgroup.
- `T subgroupExclusiveAnd(T value)` returns the exclusive scan the binary and of all active invocations `value` 's across the subgroup.
- `T subgroupExclusiveOr(T value)` returns the exclusive scan the binary or of all active invocations `value` 's across the subgroup.
- `T subgroupExclusiveXor(T value)` returns the exclusive scan the binary xor of all active invocations `value` 's across the subgroup.

For each of the reduction operations we showed above, there are equivalent inclusive and exclusive scan variants too.

So where are these operations useful?

There are places where you want to scan an entire data set and perform an operation on the entire set. For example, you might want to know of the millions of data points, which is the largest? To do this, you typically need to use atomic operations to compare and update a single value multiple times. The problem is that atomics are expensive - resulting in a significant performance drop as compared to normal memory accesses. The single most sane way to reduce the cost of atomic operations is to simply *do less of them*, and this is where our subgroup arithmetic operations can come in. Let's assume we want to get the maximum value across our entire data set:

```
#version 450

#extension GL_KHR_shader_subgroup_arithmetic: enable

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout(std430, set=0, binding=0) buffer layout_foo {
  uint foo[];
};

layout(std430, set=0, binding=1) buffer layout_bar {
  uint bar;
};

void main() {
  uint value = subgroupMax(foo[gl_GlobalInvocationID.x]);

  // A single invocation in the subgroup will do the atomic operation
  if (subgroupElect()) {
    atomicMax(bar, value);
  }
}
```

Before we had subgroup operations, we'd have performed one atomic operation per data point we wanted to consider. Now, we are performing `1/gl_SubgroupSize` of the atomic operations - a whopping 32x drop in atomic operations on NVIDIA, and 64x on AMD!

A really cool example of where scan operations are useful is when you are using a compute shader to cull triangles (see Graham Wihlidal's 2016 GDC talk Optimizing the Graphics Pipeline with Compute (https://www.ea.com/news/optimizing-the-graphics-pipeline-with-compute)). Reducing triangles means you are running fewer fragment shaders which can be a huge win. The basis of the

triangle reduction code is that each invocation is going to consider some set of triangles and decide whether its worth including the triangles or not. For our example we'll simplify the shader somewhat to keep it short:

```
#version 450

#extension GL_KHR_shader_subgroup_ballot: enable
#extension GL_KHR_shader_subgroup_arithmetic: enable

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

struct PerVertexData {
  float x;
  float y;
  float z;
  float u;
  float v;
};

layout(std430, set=0, binding=0) buffer layout_in_triangles {
  PerVertexData in_triangles[];
};

layout(std430, set=0, binding=1) buffer layout_out_triangles {
  uint out_triangles_size; // Needs to be set to 0 before shader invocation
  PerVertexData out_triangles[];
};

void main() {
  // Get the 3 vertices for our triangle
  PerVertexData x = in_triangles[gl_GlobalInvocationID.x * 3 + 0];
  PerVertexData y = in_triangles[gl_GlobalInvocationID.x * 3 + 1];
  PerVertexData z = in_triangles[gl_GlobalInvocationID.x * 3 + 2];

  // Check whether the triangle should be culled or not
  bool care_about_triangle;

  uint vertices_to_keep = care_about_triangle ? 3 : 0;

  uint local_index = subgroupExclusiveAdd(vertices_to_keep);

  // Find out which active invocation has the highest ID
  uint highestActiveID = subgroupBallotFindMSB(subgroupBallot(true));
```

```
  uint global_index = 0;

  // If we're the highest active ID
  if (highestActiveID == gl_SubgroupInvocationID) {
    // We need to carve out a slice of out_triangles for our triangle
    uint local_size = local_index + vertices_to_keep;

    global_index = atomicAdd(out_triangles_size, local_size);
  }

  global_index = subgroupMax(global_index);

  if (care_about_triangle) {
    out_triangles[global_index + local_index + 0] = x;
    out_triangles[global_index + local_index + 1] = y;
    out_triangles[global_index + local_index + 2] = z;
  }
}
```

#extension GL_KHR_shader_subgroup_shuffle

The shuffle category introduces built-in functions that allow invocations to perform more extensive data sharing across the invocations of a subgroup.

- `T subgroupShuffle(T value, uint index)` returns the `value` whose `gl_SubgroupInvocationID` is equal to `index`.
- `T subgroupShuffleXor(T value, uint mask)` returns the `value` whose `gl_SubgroupInvocationID` is equal to the current invocations `gl_SubgroupInvocationID` xor'ed with `mask`.

Shuffle performs the same action as `subgroupBroadcast`, with the difference that the index whose value we want to get can be specified dynamically.

The main benefit of the shuffle built-ins is the ability to do cross-invocation sharing in all shader stages. Compute shaders already give us a mechanism to do this with shared memory, but the shuffle built-ins give us a way to do something similar in other shader stages to:

```
#version 450

#extension GL_KHR_shader_subgroup_shuffle: enable

layout(location = 0) flat in uint index;
layout(location = 1) in vec4 x;
layout(location = 2) in float blendFactor;

layout(location = 0) out vec4 data;

void main() {
    vec4 blendWith = subgroupShuffle(x, index);

    data = mix(x, blendWith, blendFactor);
}
```

In general, if you have a compile time constant value for the index, you should use `subgroupBroadcast` as that may use more optimal hardware paths on some hardware.

Shuffle xor is a specialization of subgroup shuffle such that you know every invocation is going to trade its value with exactly one other invocation. Let's take the example where you've got your own fancy reduction algorithm that you want to apply to all members of the subgroup:

```
#version 450

#extension GL_KHR_shader_subgroup_shuffle: enable

layout(location = 0) in vec4 x;
layout(location = 1) in float additive;

layout(location = 0) out vec4 data;

void main() {
    vec4 temp = x;

    for (uint i = 1; i <= 128; i *= 2) {
        // The mask parameter of subgroupShuffleXor must either be a constant,
        // or if used within a loop it must derive from a loop counter whose
        // initial value is constant (i = 1), its stride must be constant
        // (i *= 2) and its loop end condition must be constant (i <= 128). So
        // instead of doing i <= gl_SubgroupSize above, we make the loop counter
        // check less than 128 (which is the maximum supported subgroup size),
        // and include an additional break here.
        if (gl_SubgroupSize == i) {
            break;
        }

        vec4 other = subgroupShuffleXor(temp, i);
        temp = temp * other + additive;
    }

    data = temp;
}
```

#extension GL_KHR_shader_subgroup_shuffle_relative

The shuffle relative category introduces built-in functions that allow invocations to perform shifted data sharing across the invocations of a subgroup.

- `T subgroupShuffleUp(T value, uint delta)` returns the `value` whose `gl_SubgroupInvocationID` is equal to the current invocations `gl_SubgroupInvocationID` minus `delta`.

- `T subgroupShuffleDown(T value, uint delta)` returns the `value` whose `gl_SubgroupInvocationID` is equal to the current invocations `gl_SubgroupInvocationID` plus `delta`.

These built-ins are yet further specializations of the shuffle built-ins. shuffle up and shuffle down are great if you want to concoct your own scan operations. Let's say we want a strided scan - EG. all odd invocations will do a scan together, and all even invocations too:

```
#version 450

#extension GL_KHR_shader_subgroup_shuffle_relative: enable

layout(location = 0) in vec4 x;

layout(location = 0) out vec4 data;

void main() {
    vec4 temp = x;

    // This is a custom strided inclusive scan!
    for (uint i = 2; i < gl_SubgroupSize; i *= 2) {
        vec4 other = subgroupShuffleUp(temp, i);

        if (i <= gl_SubgroupInvocationID) {
            temp = temp * other;
        }
    }

    data = temp;
}
```

Another *really cool* thing you can do with shuffle relative is reverse scans. As you may have realized, the inclusive and exclusive scan perform the scan operation from lowest to highest ID within the subgroup. There may well be situations where you want to do this in *reverse*, where you want the scan to be performed highest to lowest:

```
#version 450

#extension GL_KHR_shader_subgroup_shuffle_relative: enable

layout(location = 0) in vec4 x;

layout(location = 0) out vec4 data;

void main() {
    vec4 temp = x;

    // This is a custom reverse inclusive scan!
    for (uint i = 1; i < gl_SubgroupSize; i *= 2) {
        vec4 other = subgroupShuffleDown(temp, i);

        if ((gl_SubgroupSize - i) > gl_SubgroupInvocationID) {
            temp = temp * other;
        }
    }

    data = temp;
}
```

#extension GL_KHR_shader_subgroup_clustered

The clustered category takes the operations we introduced in the arithmetic category but allows only subsets of the invocations to interact with each other.

- `T subgroupClusteredAdd(T value, uint clusterSize)` returns the summation of all active invocations `value` 's across clusters of size `clusterSize` .
- `T subgroupClusteredMul(T value, uint clusterSize)` returns the multiplication of all active invocations `value` 's across clusters of size `clusterSize` .
- `T subgroupClusteredMin(T value, uint clusterSize)` returns the mininum `value` of all active invocations `value` 's across clusters of size `clusterSize` .
- `T subgroupClusteredMax(T value, uint clusterSize)` returns the maximum `value` of all active invocations `value` 's across clusters of size `clusterSize` .
- `T subgroupClusteredAnd(T value, uint clusterSize)` returns the binary and of all active invocations `value` 's across clusters of size `clusterSize` .

- `T subgroupClusteredOr(T value, uint clusterSize)` returns the binary or of all active invocations `value` 's across clusters of size `clusterSize` .
- `T subgroupClusteredXor(T value, uint clusterSize)` returns the binary xor of all active invocations `value` 's across clusters of size `clusterSize` .

The `clusterSize` must be a compile-time constant, a power of two, and at least one. You'll also get undefined results if `clusterSize` is greater than `gl_SubgroupSize` .

The main idea with clustered operations is that sometimes you want to only share data with a selection of your closest neighbors within the subgroup.

A really cool thing you can do with subgroup operations is implemented a high performance convolutional neural network. The clustered operations, in particular, can help us implement a specific part of the neural network, max pooling (https://en.wikipedia.org/wiki/Convolutional_neural_network#Max_pooling_shape). In max pooling, we want to take a large data set and compress it to a smaller data set. We do this by dividing our large data set into an NxN grid and outputting a single element, the maximum value within the NxN grid. In the below example, we'll use a 4x4 grid using clusters of size 16:

```
#version 450

#extension GL_KHR_shader_subgroup_clustered: enable

// Using a spec constant for the x dimension here, and we'll set this to the
// subgroup size in the Vulkan API so that the work group is the exact size of
// the subgroup
layout(local_size_x_id = 0, local_size_y = 1, local_size_z = 1) in;

layout(r16f, set = 0, binding = 0) uniform readonly image3D inImage;
layout(r16f, set = 0, binding = 1) uniform writeonly image3D outImage;

void main() {
  // We're going to perform a 4x4 max pooling operation. So we divide up our
  // subgroup size into chunks of 16. We'll split up the subgroup like so:
  //  x  0  1  2  3  4  5  6  7
  // y+------------------------
  // 0|  0  1  2  3 16 17 18 19
  // 1|  4  5  6  7 20 21 22 23
  // 2|  8  9 10 11 24 25 26 27
  // 3| 12 13 14 15 28 29 30 31

  uint numClusters = gl_SubgroupSize / 16;
  uint clusterID = gl_SubgroupInvocationID / 16;

  uint x = (clusterID * 4) + (gl_SubgroupInvocationID % 4);
  uint y = (gl_SubgroupInvocationID / 4) % 4;

  uvec3 inIndex = gl_WorkGroupID * uvec3(numClusters * 4, 4, 1) + uvec3(x, y,
 1);

  float load = imageLoad(inImage, ivec3(inIndex)).x;

  float max = subgroupClusteredMax(load, 16);

  if (0 == (gl_SubgroupInvocationID % 16)) {
    uvec3 outIndex = gl_WorkGroupID;
    outIndex.x = outIndex.x * numClusters + clusterID;

    imageStore(outImage, ivec3(outIndex), vec4(max));
```

```
  }
}
```

#extension GL_KHR_shader_subgroup_quad

The quad category introduces the concept of a subgroup quad, which is a cluster of size 4. This quad corresponds to 4 neighboring pixels in a 2x2 grid within fragment shaders. The quad operations allow for the efficient sharing of data within the quad.

- `T subgroupQuadBroadcast(T value, uint id)` returns the `value` in the quad whose `gl_SubgroupInvocationID` modulus `4` is equal to `id`.
- `T subgroupQuadSwapHorizontal(T value)` swaps `value`'s witin the quad horizontally.
- `T subgroupQuadSwapVertical(T value)` swaps `value`'s witin the quad vertically.
- `T subgroupQuadSwapDiagonal(T value)` swaps `value`'s witin the quad diagonally.

Quad operations are not restricted to just fragment stages though. In other stages, you can think of the quad as just a special case of the clustered operations where the cluster size is 4.

An example of where quad operations can be useful is to imagine you've got a fragment shader, and you also want to output a lower resolution image of the fragment result (the next smaller mip level). We can do this by including a single extra image store from the fragment shader, and use our quad subgroup functionality:

```
#version 450

#extension GL_KHR_shader_subgroup_quad: enable

layout(location = 0) in vec3 inColor;

layout(location = 0) out vec3 color;

layout(set = 0, binding = 0) uniform writeonly image2D lowResFrame;

void main() {
    vec3 lowResColor = inColor + subgroupQuadSwapHorizontal(inColor);
    lowResColor += subgroupQuadSwapVertical(lowResColor);

    // Store the color out to the framebuffer
    color = inColor;

    // Only the 0'th pixel of each quad will enter here and do the image store
    if (gl_SubgroupInvocationID == subgroupQuadBroadcast(gl_SubgroupInvocationI
D, 0)) {
      ivec2 coord = ivec2(gl_FragCoord.xy / 2.0f);
      imageStore(lowResFrame, coord, vec4(lowResColor, 1.0f));
    }
}
```

One More Thing

While the GLSL lovers among the readership will be delighted with these new GLSL additions, it'd be wrong if we forgot our HLSL loving brethren! Alongside the new GLSL functionality added to glslang, we're pleased to announce that all of the Shader Model 6.0 wave operations (https://msdn.microsoft.com/en-us/library/windows/desktop/mt733232(v=vs.85).aspx) are also supported in glslang's HLSL mode in the 1.1.70 release of LunarG's Vulkan SDK (https://vulkan.lunarg.com/sdk/home) and available in the glslang GitHub repository (https://github.com/KhronosGroup/glslang/pull/1277) and in DXC's GitHub repository (https://github.com/Microsoft/DirectXShaderCompiler/pull/1118).

Minor note: some of the built-ins (like the clustered and shuffle operations) are only available in GLSL shaders as HLSL has no corresponding built-ins we can map to.

About the Author

Neil Henning is the Principal Software Engineer, for Vulkan & SPIR-V at Codeplay Software Ltd., and Codeplay's lead representative to the Vulkan and SPIR-V working-groups as a member of Khronos. Neil dedicated 2 years of his blood, sweat, and tears as the primary engineer and author of Vulkan 1.1's new subgroup functionality. The best place to find Neil is on Twitter @sheredom (https://twitter.com/sheredom).

Authored by Neil Henning, Principal Software Engineer, for Vulkan & SPIR-V at Codeplay Software Ltd.