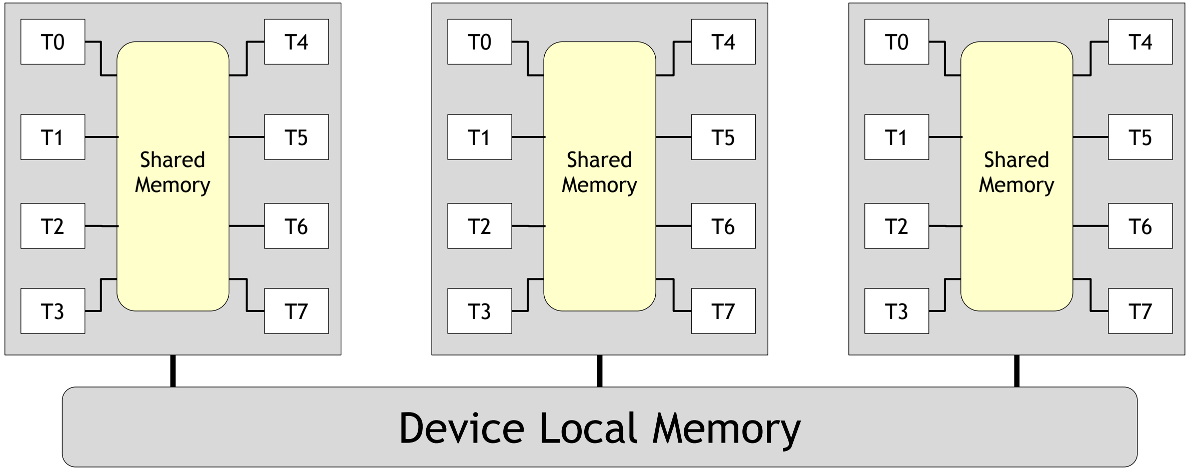
Vulkan Subgroups

Nuno Subtil
nsubtil@nvidia.com

Agenda

- What are subgroups and why they’re useful
- Subgroup overview
- Vulkan 1.1 Subgroup Operations
- Partitioned Subgroup Operations
- NV Implementation Details
- Tips
- Mapping to HLSL

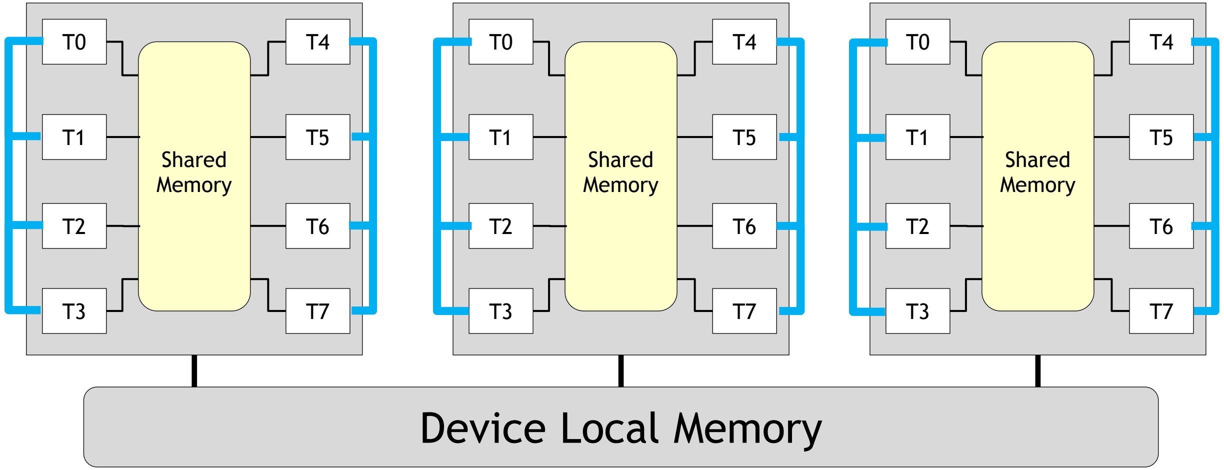
Subgroups?



Vulkan 1.0

- Threads execute in workgroups
- Each workgroup has some amount of (fast) shared memory
- Threads communicate via shared memory within workgroup

Subgroups!



Vulkan 1.0

- Threads execute in workgroups
- Each workgroup has some amount of (fast) shared memory
- Threads communicate via shared memory within workgroup

Vulkan 1.1

- Adds ***subgroups***: sets of threads within workgroup that can communicate directly
- Can be faster than shared memory

Subgroups!

- **All-to-all communication across sets of threads within a workgroup**
  - Equivalent to HLSL SM6 wave ops

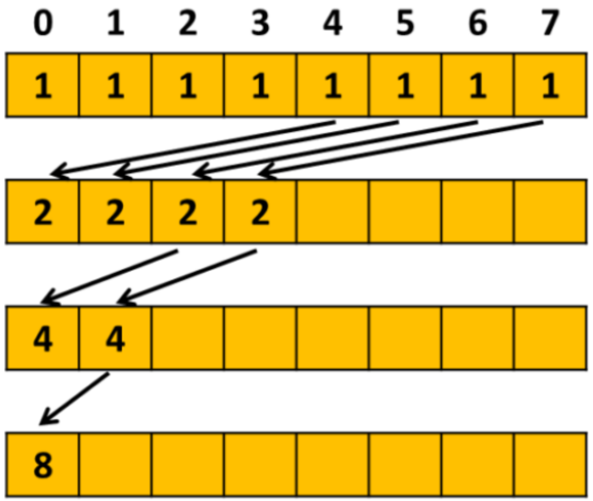
- **Can be more efficient than shared memory**
  - If the data you want to move is in registers, subgroups are typically faster
  - Implicit, finer-grained synchronization

- **May have lower latency**
  - Shared memory implies workgroup-wide synchronization
  - Subgroup operations only require synchronization within participating threads

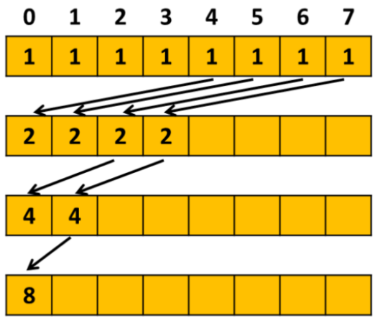
- **Exposed in all shader stages**
  - Compute support is required
  - Other stages are allowed depending on the implementation

Subgroup example: reduction

Simple parallel reduction example: sum of values across threads



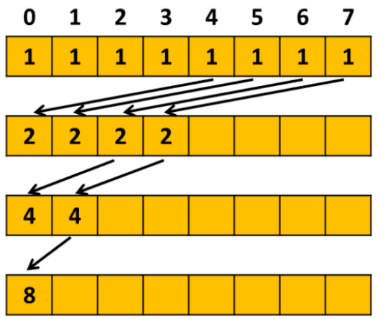
Subgroup example: reduction



Parallel reduction loop using shared memory

```
shared int s[WORKGROUP_SIZE];
…
int a = compute_local_value();
s[gl_WorkGroupID.x] = a;                   // memory write
memoryBarrierShared();                     // synchronize workgroup
if (current_thread_should_reduce()) {
  int b = s[gl_WorkGroupID.x + iterDelta]; // memory read
  perform_reduction_step(a, b);
  iterDelta /= 2;
}
```

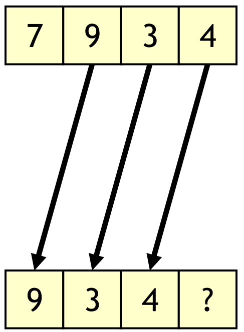
Subgroup example: reduction



Parallel reduction loop using shared memory

```
shared int s[WORKGROUP_SIZE];
…
int a = compute_local_value();
s[gl_WorkGroupID.x] = a;                   // memory write
memoryBarrierShared();                     // synchronize workgroup
if (current_thread_should_reduce()) {
  int b = s[gl_WorkGroupID.x + iterDelta]; // memory read
  perform_reduction_step(a, b);
  iterDelta /= 2;
}
```

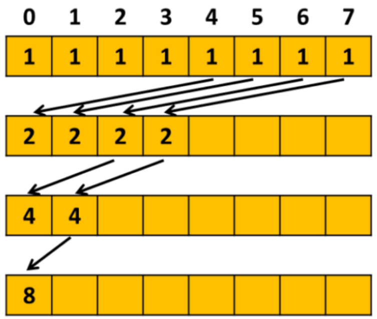
ShuffleDown (x, 1)



Parallel reduction loop using subgroups

```
int a = compute_local_value();
int b = subgroupShuffleDown(a, iterDelta); // synchronize subgroup
if (current_thread_should_reduce()) {
  perform_reduction_step(a, b);
  iterDelta /= 2;
}
```

Subgroup example: reduction



Parallel reduction loop using shared memory

```
shared int s[WORKGROUP_SIZE];
…
int a = compute_local_value();
s[gl_WorkGroupID.x] = a;                    // memory write
memoryBarrierShared();                      // synchronize workgroup
if (current_thread_should_reduce()) {
  int b = s[gl_WorkGroupID.x + iterDelta]; // memory read
  perform_reduction_step(a, b);
  iterDelta /= 2;
}
```

Workgroup-wide synchronization + two memory operations

Parallel reduction loop using subgroups

```
int a = compute_local_value();
int b = subgroupShuffleDown(a, iterDelta); // synchronize subgroup
if (current_thread_should_reduce()) {
  perform_reduction_step(a, b);
  iterDelta /= 2;
}
```

Fewer threads synchronize; no memory operations

When to use subgroups?

- **Quite a few algorithms may benefit**
  - Reductions: post-processing effects
    - Min/max/sum across range of data
    - Bloom, depth-of-field, motion blur are good candidates
  - List building: light culling
    - Reduce shared memory atomics, skip work across subgroup (e.g., entire subgroup decides no elements need to be added to the list)
  - Sorting
    - Bitonic sort can be accelerated via subgroups

When *not* to use subgroups?

- **Tradeoff between different kinds of latency**
  - Shared memory:
    - Workgroup-wide synchronization latency
    - Read/write latency (e.g., if backed by cache)
  - Subgroups:
    - Subgroup-wide synchronization latency
    - Potentially increased ALU/issue latency (may expand to multiple instructions)

- **Tradeoffs can be implementation dependent**

More details

- **Set of shader invocations (threads)**
  - ***Efficiently*** synchronize and share data with each other
  - Exposed **“as if”** running concurrently
    - Maps to warp (NV), wavefront (AMD)
    - Implementation *can* advertise smaller subgroup size than HW implements

- **Invocations in a subgroup may be *active* or *inactive***
  - **Active**: execution is being performed
  - **Inactive**: not being executed
    - Non-uniform flow control
    - Local workgroup size not a multiple of subgroup size
  - Can change throughout shader execution as control flow diverges and re-converges

Vulkan 1.1 API: Subgroup Properties

- A new structure to query subgroup support on a physical device
  - subgroupSize - number of invocations in each subgroup
    - *must be at least 1 (and <= 128)*
  - supportedStages - which shader stages support subgroup operations
    - *VK_SHADER_STAGE_COMPUTE_BIT is required*
  - supportedOperations - which subgroup operations are supported
    - *VK_SUBGROUP_FEATURE_BASIC_BIT is required*
  - quadOperationsInAllStages - do quads ops work in all stages or only fragment and compute

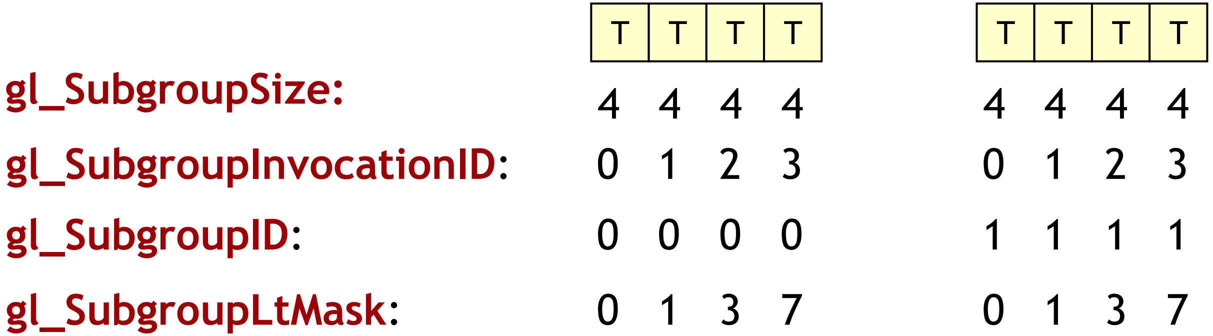
```
typedef struct VkPhysicalDeviceSubgroupProperties {
    VkStructureType          sType;
    void*                    pNext;
    uint32_t                 subgroupSize;
    VkShaderStageFlags       supportedStages;
    VkSubgroupFeatureFlags   supportedOperations;
    VkBool32                 quadOperationsInAllStages;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

```
typedef enum VkSubgroupFeatureFlagBits {
    VK_SUBGROUP_FEATURE_BASIC_BIT = 0x00000001,
    VK_SUBGROUP_FEATURE_VOTE_BIT = 0x00000002,
    VK_SUBGROUP_FEATURE_ARITHMETIC_BIT = 0x00000004,
    VK_SUBGROUP_FEATURE_BALLOT_BIT = 0x00000008,
    VK_SUBGROUP_FEATURE_SHUFFLE_BIT = 0x00000010,
    VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT = 0x00000020,
    VK_SUBGROUP_FEATURE_CLUSTERED_BIT = 0x00000040,
    VK_SUBGROUP_FEATURE_QUAD_BIT = 0x00000080,
    VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV = 0x00000100,
} VkSubgroupFeatureFlagBits;
```

Shader Built-in variables

- All supported stages
  - gl_SubgroupSize – size of the subgroup – matches the API property
  - gl_SubgroupInvocationID – ID of the invocation within the subgroup, [0..gl_SubgroupSize)
  - gl_Subgroup{Eq,Ge,Gt,Le,Lt}Mask
    - bitmask of all invocations as compared to the gl_SubgroupInvocationID of current invocation
    - Useful for working with subgroupBallot results (more on this later)

- Compute only
  - gl_NumSubgroups – number of subgroups in local workgroup
  - gl_SubgroupID – ID of subgroup within local workgroup, [0..gl_NumSubgroups)



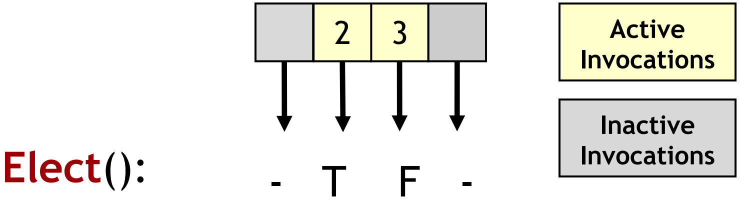
	T	T	T	T	T	T	T	T
gl_SubgroupSize:	4	4	4	4	4	4	4	4
gl_SubgroupInvocationID:	0	1	2	3	0	1	2	3
gl_SubgroupID:	0	0	0	0	1	1	1	1
gl_SubgroupLtMask:	0	1	3	7	0	1	3	7

Subgroup Basic Operations

- **Subgroup-wide barriers**
  - void subgroupBarrier() - Full memory and execution barrier
    - All active invocations sync and memory stores to ***coherent*** memory locations are completed
  - void subgroupMemoryBarrier()
    - Enforces ordering of all memory transactions by an invocation, as seen by other invocations in the subgroup
  - void subgroupMemoryBarrier{Buffer,Image,Shared}()
    - Enforces ordering on buffer, image, or shared (compute only) memory operations, respectively

Subgroup Vote Operations

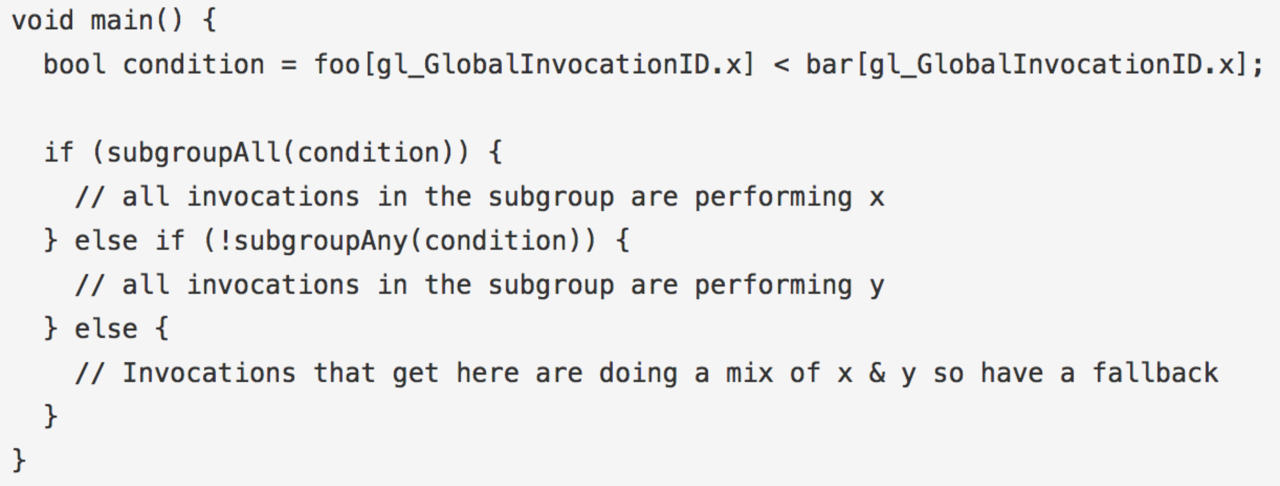
- Select one thread in a subgroup
  - bool subgroupElect()
    - Pick one active invocation, always the one with lowest gl_SubgroupInvocationID
    - Used for executing work on only one invocation



Subgroup Vote Operations

- **Determine if a Boolean condition is met across the entire subgroup**
  - bool subgroupAll(bool value)
    - true if all invocations have <value> == true
  - bool subgroupAny(bool value)
    - true if any invocation has <value> == true
  - bool subgroupAllEqual(T value)
    - true if all invocations have the same value of <value>

- **Useful for code that has branching**
  - Can do more optimal calculations if certain conditions are met



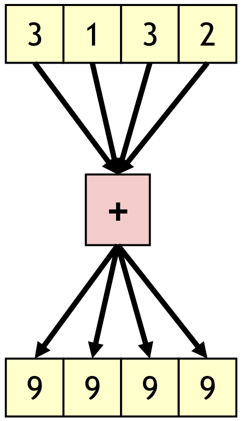
```
void main() {
  bool condition = foo[gl_GlobalInvocationID.x] < bar[gl_GlobalInvocationID.x];

  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
  }
}
```

Subgroup Arithmetic Operations

- Operations across all active invocations in a subgroup
  - T subgroup<op>(T value)
    - <op> = Add, Mul, Min, Max, And, Or, Xor
  - Reduction operations
    - Returns the result of the same calculation to each invocation

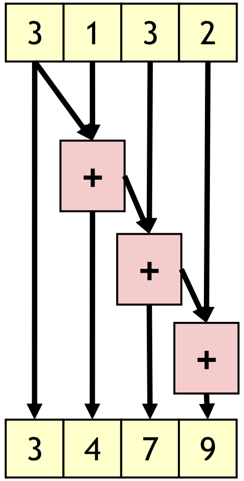
Add



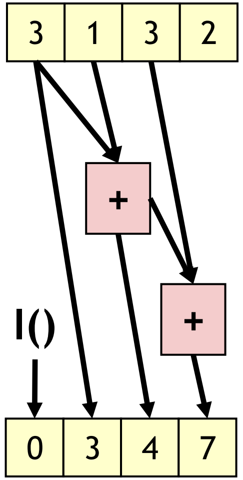
Subgroup Arithmetic Operations

- Operation on invocations with gl_SubgroupInvocationID less than self
  - T subgroupInclusive<op>(T value)
    - Includes own value in operation
  - T subgroupExclusive<op>(T values)
    - Excludes own value from operation
  - Inclusive or exclusive scan
    - Returns the result of different calculation to each invocation

InclusiveAdd



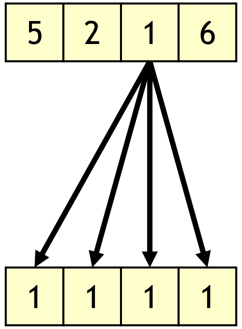
ExclusiveAdd



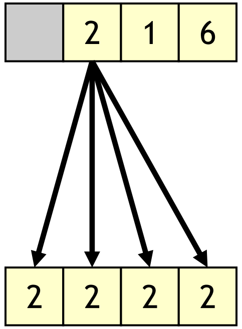
Subgroup Ballot Operations

- Allow invocations to do limited data sharing across a subgroup
  - **Broadcast** – value from one invocation to all invocations
    - T subgroupBroadcast(T value, uint id)
      - Broadcasts <value> from the invocation with gl_SubgroupInvocationID == id
      - <id> must be compile time constant
    - T subgroupBroadcastFirst(T value)
      - Broadcasts <value> from the invocation with lowest active gl_SubgroupInvocationID

Broadcast(x,2)

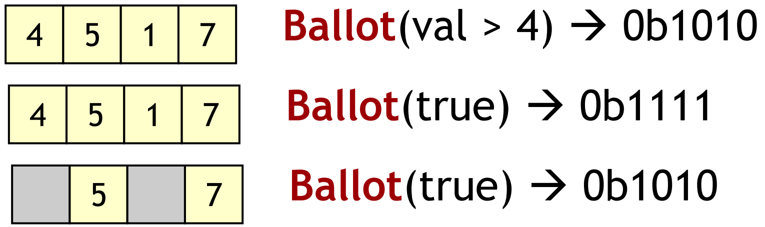


BroadcastFirst



Subgroup Ballot Operations

- Allow invocations to do limited data sharing across a subgroup
  - More powerful form of voting
    - uvec4 subgroupBallot(bool value)
      - Returns bitfield ***ballot*** with result of evaluating <value> in each invocation
      - Bit <i> == 1 means expression evaluated to true for gl_SubgroupInvocationID == i
      - Bit <i> == 0 means expression evaluated to false, or invocation inactive
      - uvec4 used in ballots is treated as a bitfield with gl_SubgroupSize significant bits
      - First invocation is in bit 0 of first vector component (.x), 32nd invocation in bit 0 of .y, etc.
      - Bits beyond gl_SubgroupSize are ignored
      - subgroupBallot(true) gives a bitfield of all the active invocations



4	5	1	7	**Ballot**(val > 4) → 0b1010
4	5	1	7	**Ballot**(true) → 0b1111
	5		7	**Ballot**(true) → 0b1010

Subgroup Ballot Operations

- **Ballot helper functions – to simplify working with uvec4 bitfield**
  - bool subgroupInverseBallot(uvec4 value)
    - Returns true if current invocation bit in <value> is set
  - bool subgroupBallotBitExtract(uvec4 value, uint index)
    - Returns true if bit in <value> that corresponds to <index> is 1
  - uint subgroupBallotBitCount(uvec4 value)
    - Returns the count of bits set to 1 in <value>
  - uint subgroupBallot{Exclusive,Inclusive}BitCount(uvec4 value)
    - Returns the exclusive/inclusive count of bits set to 1 in <value>
    - For bits with invocation ID < or <= the current invocation ID
  - uint subgroupBallotFind{LSB,MSB}(uvec4 value)
    - Returns the lowest/highest bit set in <value>

InverseBallot(0b1010) →

F	T	F	T

BallotBitCount(0b1101) →

3	3	3	3

BallotInclusiveBitCount(0b1101) →

1	1	2	3

== BallotBitCount(val & gl_SubgroupLeMask)

BallotExclusiveBitCount(0b1101) →

0	1	1	2

== BallotBitCount(val & gl_SubgroupLtMask)

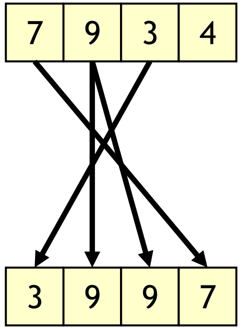
Subgroup Shuffle Operations

- More extensive data sharing across the subgroup
  - Invocations can read values from other invocations in the subgroup

- Shuffle
  - T subgroupShuffle(T value, uint id)
    - Returns <value> from the invocation with gl_SubgroupInvocationID == id
    - Like subgroupBroadcast, but <id> can be determined dynamically

Shuffle(x, index)

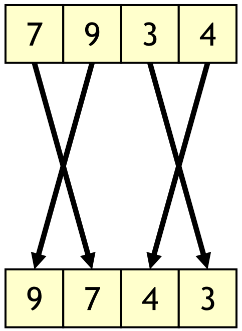
index = 2, 1, 1, 0



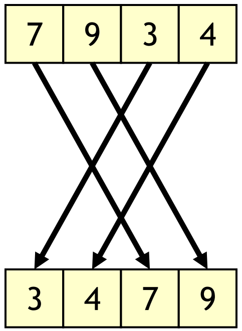
Subgroup Shuffle Operations

- **ShuffleXor**
  - T subgroupShuffleXor(T value, uint mask)
    - Returns <value> from the invocation with gl_SubgroupInvocationID == (mask ^ current)
    - Every invocation trades value with exactly one other invocation
    - Specialization of general shuffle
    - <mask> must be constant integral expression
    - Special conditions for using in a loop (basically needs to be unrollable)

ShuffleXor(x, 1)



ShuffleXor(x, 2)

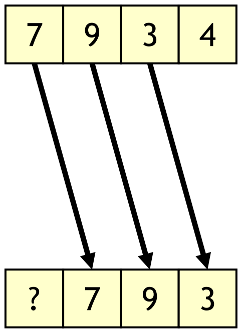


Subgroup Shuffle Relative Operations

- Enable invocations to perform shifted data sharing between subgroup invocations
  - T subgroupShuffleUp(T value, uint delta)
    - Returns <value> from the invocation with gl_SubgroupInvocationID == (current - delta)
  - T subgroupShuffleDown(T value, uint delta)
    - Returns <values> from the invocation with gl_SubgroupInvocationID == (current + delta)

- Useful to construct your own scan operations
  - Strided scan (e.g. even or odd invocations, etc.)

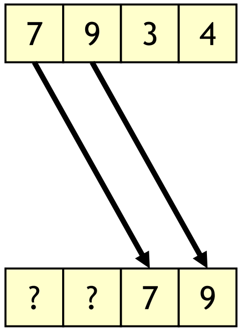
ShuffleUp(x, 1)



7 9 3 4

? 7 9 3

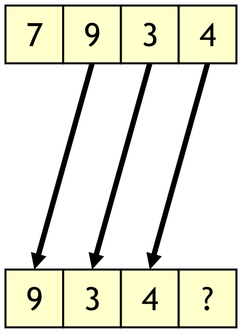
ShuffleUp(x, 2)



7 9 3 4

? ? 7 9

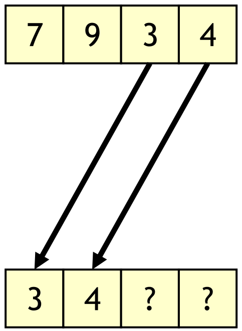
ShuffleDown (x, 1)



7 9 3 4

9 3 4 ?

ShuffleDown (x, 2)



7 9 3 4

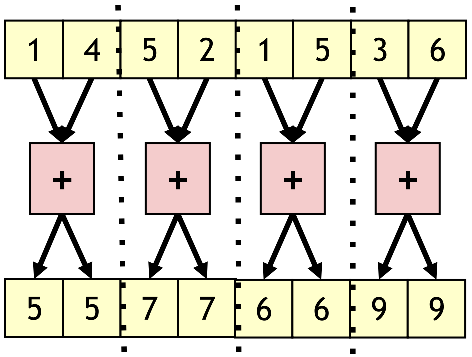
3 4 ? ?

Subgroup Clustered Operations

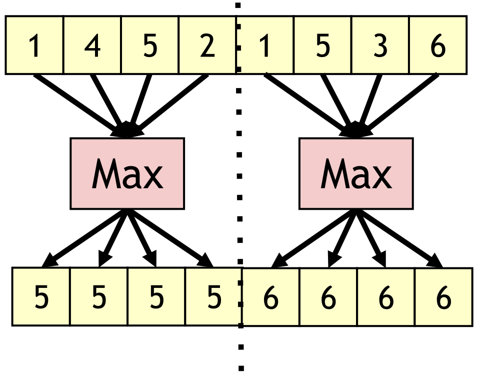
- **Perform arithmetic operations across a fixed partition of a subgroup**
  - T subgroupClustered<op>(T value, uint clusterSize)
    - <op> = Add, Mul, Min, Max, And, Or, Xor
    - clusterSize – size of partition
      - compile-time constant
      - power of 2, >= 1
    - Only active invocations in the partition participate

- **Sharing data only with a selection of your closest neighbors**
  - An algorithm that relies on a fixed size grid < gl_SubgroupSize
  - Eg: Convolution neural network – max pooling
    - Take large data set and compress to a smaller one
    - Divide data into NxN grid – N=clusterSize
    - Output maximum for each cluster

ClusteredAdd(x, 2)

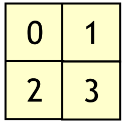


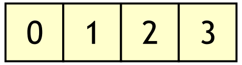
ClusteredMax(x, 4)



Subgroup Quad Operations

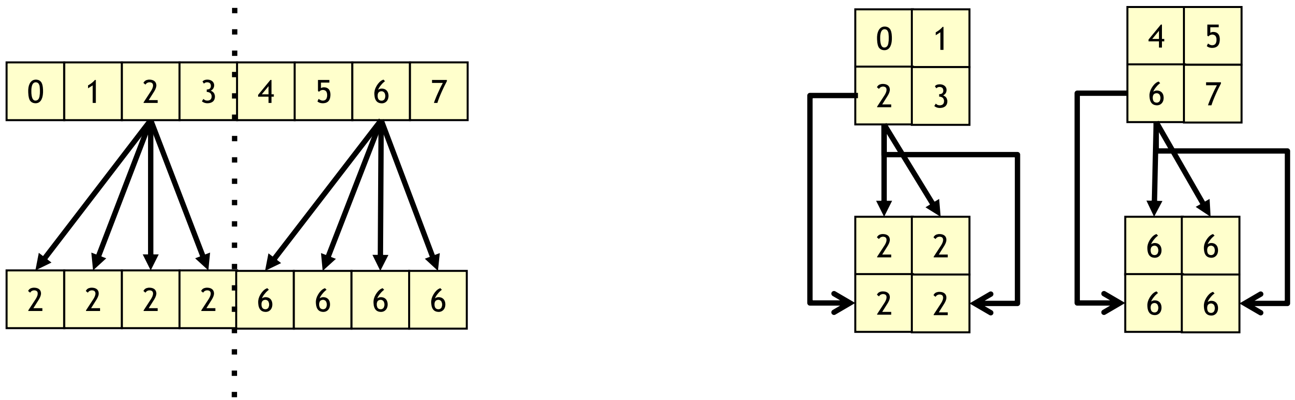
- Subgroup quad is a cluster of size 4
  - Neighboring pixels in a 2x2 grid in fragment shaders (ie derivative group)
  - Not restricted to fragment shaders,
  - Just a cluster of 4 in other stages (no defined layout)
    - Remember to check for support (quadOperationsInAllStages property)





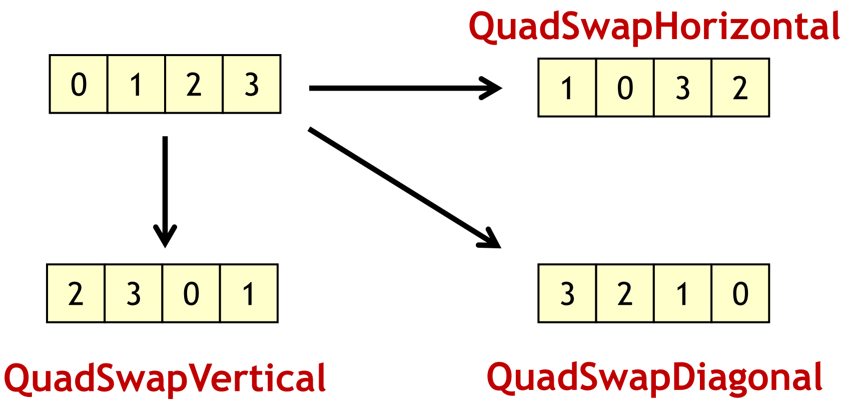
- Broadcast
  - T subgroupQuadBroadcast(T value, uint id)
    - Returns <value> from the invocation where gl_SubgroupInvocationID % 4 = <id>

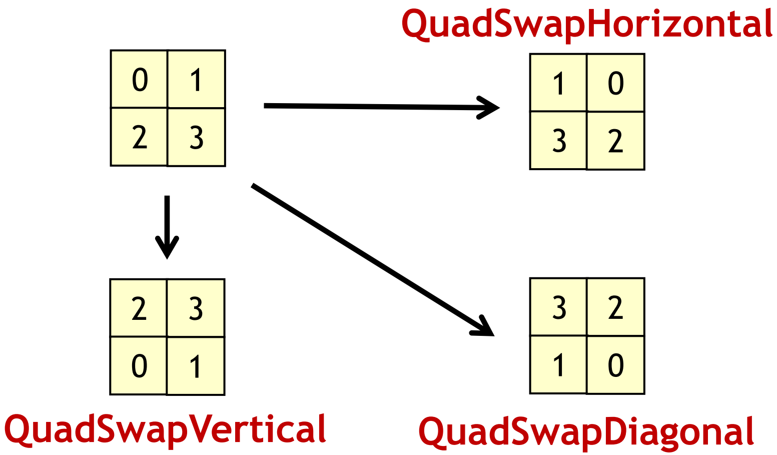
QuadBroadcast(x, 2)



Subgroup Quad Operations

- Swap
  - T subgroupQuadSwapHorizontal(T value)
    - Swap values horizontally in the quad
  - T subgroupQuadSwapVertical(T value)
    - Swap values vertically in the quad
  - T subgroupQuadSwapDiagonal(T value)
    - Swap values diagonally in the quad
  - Can easily construct a lower resolution image (2x2 filter)
    - See subgroup tutorial for details



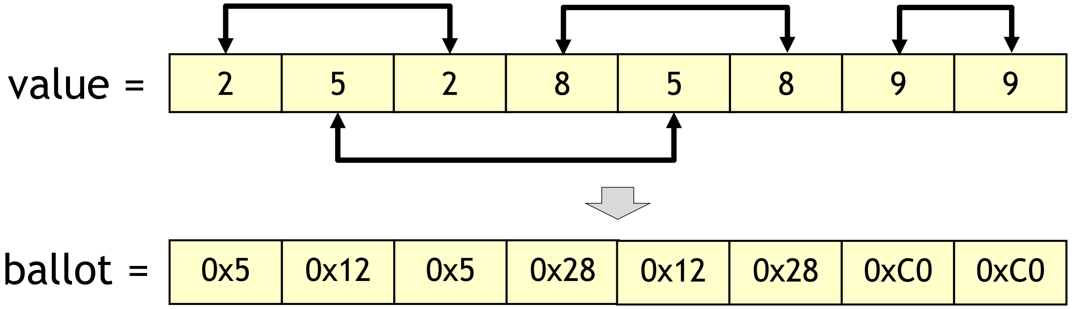


Subgroup Partitioned Operations (NV)

- **Perform arithmetic operations across a flexible set of invocations**
  - Generalization of clustering which does not need fixed-size clusters or offsets
  - VK_NV_shader_subgroup_partitioned /GL_NV_shader_subgroup_partitioned

- **Generate a partition**
  - uvec4 subgroupPartitionNV(T value)
    - Returns a **ballot** which is a partition of all invocations in the subgroup based on <value>
    - All invocations represented by the same ballot have the same <value>
    - All invocations in different ballots have different <value>

PartitionNV



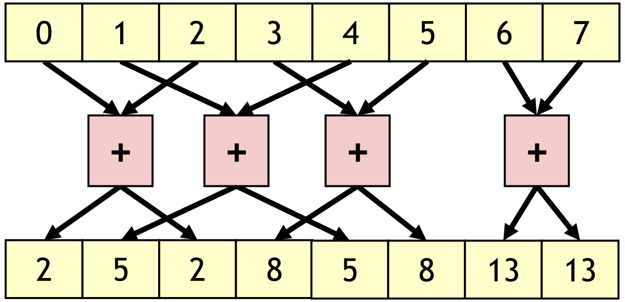
value =	2	5	2	8	5	8	9	9
ballot =	0x5	0x12	0x5	0x28	0x12	0x28	0xC0	0xC0

Subgroup Partitioned Operations (NV)

- **Operation on a partition**
  - T subgroupPartitionedInclusive<op>NV(T value, uvec4 ballot)
  - T subgroupPartitionedExclusive<op>NV(T value, uvec4 ballot)
  - T subgroupPartitioned<op>NV(T value, uvec4 ballot)
    - <op> is Add, Mul, Min, Max, And, Or, Xor
    - Inclusive scan, exclusive scan, reduction operate similar to clustered/arithmetic operations
    - <ballot> describes the partition – typically the result from subgroupPartitionNV
    - No restrictions on how the invocations are partitioned, except that the ballot values passed in must represent a “valid” partition

PartitionedAddNV(values, ballot)

ballot = 0x5, 0x12, 0x28, 0xC0



0 1 2 3 4 5 6 7

2 5 2 8 5 8 13 13

Subgroup Partitioned Operations

- **Why partitions?**
  - Shaders can't really predict that consecutive invocations will have related values
  - More useful to "discover" (subgroupPartitionNV) those invocations that are related, and then do subgroup operations on related invocations
  - E.g. Deferred shading, detect pixels with the same material or light

- **Any implementation that supports VK_SUBGROUP_FEATURE_ARITHMETIC_BIT can trivially support partitioned ops**
  - Loop over unique partition subsets, compute each in flow control
  - Cost = NumSubsets * costof(SUBGROUP_FEATURE_ARITHMETIC)
- **Some implementations can compute all subsets in parallel**
  - Cost = costof(SUBGROUP_FEATURE_ARITHMETIC)
  - More useful generalization of clustering, and at the same cost
- **Most implementations can probably do better than the trivial looping**

NVIDIA Implementation Details

- **Native hw instructions are essentially what is exposed in**
  - GL_NV_shader_thread_shuffle and GL_NV_shader_thread_group

- **shuffle/shuffleUp/shuffleDown/shuffleXor are fast instructions**
  - Essentially our primitives
  - Most other instructions are built up from these using relatively simple transforms
  - Don’t be afraid to use more general operations!
    - Can still be faster than composing from building blocks

- **All the subgroup operations are similar cost**
  - E.g. a REDUCE operation (subgroup<op>) is basically:

```
x = op(x, shuffleXor(x, 1));
x = op(x, shuffleXor(x, 2));
x = op(x, shuffleXor(x, 4));
x = op(x, shuffleXor(x, 8));
x = op(x, shuffleXor(x, 16));
```

Tips

- Make local workgroup be at least the size of the subgroup (compute),
  - Ideally integer multiples
  - Common subgroup sizes: 32 (NVIDIA, Intel), 64 (AMD)
- Subgroup size of 1 isn’t very useful, but makes a single code path possible
- Subgroup operations provide implicit subgroup execution barriers
- Operations only act on active invocations
- Be aware of inactive lanes or out of range invocation IDs
  - Reading gives undefined values in most cases!
- Helper invocations participate in subgroup operations

HLSL SM 6.0 Wave Ops Comparison

D3D Wave Ops

- Wave lane count: 4 - 128
- Required in pixel and compute shaders
  - Not supported in any other stages
- All or nothing functionality
- Types: half, float, double, int, uint, short, ushort, uint64 (as supported)

Vulkan Subgroups

- Subgroup size: 1 - 128
- Required in compute shaders
  - Optional in Frag, Vert, Tess, Geom stages
- Minimum functionality guaranteed, additional bundles of functionality
- Types: bool, float, double, int, uint
  - More types to be added in the future
- More complete set of intrinsics
  - Inclusive scan, clustered ops, etc.
  - Barriers
  - More helper routines

Availability

- **GLSL functionality**
  - Glslang - https://github.com/khronosgroup/glslang/

- **HLSL functionality**
  - Glslang - https://github.com/KhronosGroup/glslang
  - DXC - https://github.com/Microsoft/DirectXShaderCompiler/

- **SPIR-V 1.3**

- **Vulkan support**
  - https://vulkan.gpuinfo.org/ (under Device Properties)
  - NVIDIA Vulkan 1.1 drivers - http://www.nvidia.com/Download/index.aspx
  - AMD Vulkan 1.1 drivers
  - Intel Vulkan 1.1 drivers

References

- Vulkan Subgroup Tutorial
  - https://www.khronos.org/blog/vulkan-subgroup-tutorial

- GL_KHR_shader_subgroup GLSL extension
  - https://github.com/KhronosGroup/GLSL/blob/master/extensions/khr/GL_KHR_shader_subgroup.txt
- GL_NV_shader_subgroup_partitioned GLSL extension
  - https://github.com/KhronosGroup/GLSL/blob/master/extensions/nv/GL_NV_shader_subgroup_partitioned.txt
- HLSL Shader Model 6.0 (MSDN)
  - https://msdn.microsoft.com/en-us/library/windows/desktop/mt733232(v=vs.85).aspx
- DirectXShaderCompiler Wave Intrinsics
  - https://github.com/Microsoft/DirectXShaderCompiler/wiki/Wave-Intrinsics
- Reading Between the Threads: Shader Intrinsics
  - https://developer.nvidia.com/reading-between-threads-shader-intrinsics
- Faster Parallel Reductions on Kepler
  - https://devblogs.nvidia.com/faster-parallel-reductions-kepler/

Thank You

- Daniel Koch (NVIDIA)
- Neil Henning (AMD) @sheredom
- Lei Zhang (Google)
- Jeff Bolz (NVIDIA)
- Piers Daniell (NVIDIA)

HLSL / GLSL / SPIR-V Mappings

HLSL Intrinsic (Query)	GLSL Intrinsic	SPIR-V Op
WaveGetLaneCount() [4-128]	gl_SubgroupSize[1-128]	SubgroupSize decorated OpVariable
WaveGetLaneIndex	gl_SubgroupInvocationID	SubgroupId decorated OpVariable
WaveIsFirstLane()	subgroupElect()	OpGroupNonUniformElect

HLSL Intrinsic (Vote)	GLSL Intrinsic	SPIR-V Op
WaveActiveAnyTrue()	subgroupAny()	OpGroupNonUniformAny
WaveActiveAllTrue()	subgroupAll()	OpGroupNonUniformAll
WaveActiveBallot()	subgroupBallot()	OpGroupNonUniformBallot

HLSL Intrinsic (Broadcast)	GLSL Intrinsic	SPIR-V Op
WaveReadLaneAt()	subgroupBroadcast(const) / subgroupShuffle(dynamic)	OpGroupNonUniformBroadcast / OpGroupNonUniformShuffle
WaveReadLaneFirst()	subgroupBroadcastFirst()	OpGroupNonUniformBroadcastFirst

HLSL / GLSL / SPIR-V Mappings

HLSL Intrinsic (Reduction)	GLSL Intrinsic	SPIR-V Op
WaveActiveAllEqual()	subgroupAllEqual()	OpGroupNonUniformAllEqual
WaveActiveBitAnd()	subgroupAnd()	OpGroupNonUniformBitwiseAnd / OpGroupNonUniformLogicalAnd
WaveActiveBitOr()	subgroupOr()	OpGroupNonUniformBitwiseOr / OpGroupNonUniformLogicalOr
WaveActiveBitXor()	subgroupXor()	OpGroupNonUniformBitwiseXor / OpGroupNonUniformLogicalXor
WaveActiveCountBits()	subgroupBallotBitcount()	OpGroupNonUniformBallotBitCount
WaveActiveMax()	subgroupMax()	OpGroupNonUniform*Max
WaveActiveMin()	subgroupMin()	OpGroupNonUniform*Min
WaveActiveProduct()	subgroupMul()	OpGroupNonUniform*Mul
WaveActiveSum()	subgroupAdd()	OpGroupNonUniform*Add

HLSL / GLSL / SPIR-V Mappings

HLSL Intrinsic (Scan and Prefix)	GLSL Intrinsic	SPIR-V Op
WavePrefixCountBits()	subgroupBallotExclusiveBitCount()	OpGroupNonUniformBallotBitCount
WavePrefixSum()	subgroupExclusiveAdd()	OpGroupNonUniform*Add
WavePrefixProduct()	subgroupExclusiveMul()	OpGroupNonUniform*Mul

HLSL Intrinsic (Quad Shuffle)	GLSL Intrinsic	SPIR-V Op
QuadReadLaneAt()	subgroupQuadBroadcast()	OpGroupNonUniformQuadBroadcast
QuadReadAcrossDiagonal()	subgroupQuadSwapDiagonal()	OpGroupNonUniformQuadSwap
QuadReadAcrossX()	subgroupQuadSwapHorizontal()	OpGroupNonUniformQuadSwap
QuadReadAcrossY()	subgroupQuadSwapVertical()	OpGroupNonUniformQuadSwap