Hi, I'm Alex. One of the 6 Alexes at Snowed In Studios.

I'd like to thank the organizers of the conference for having me and for all of you for being here.

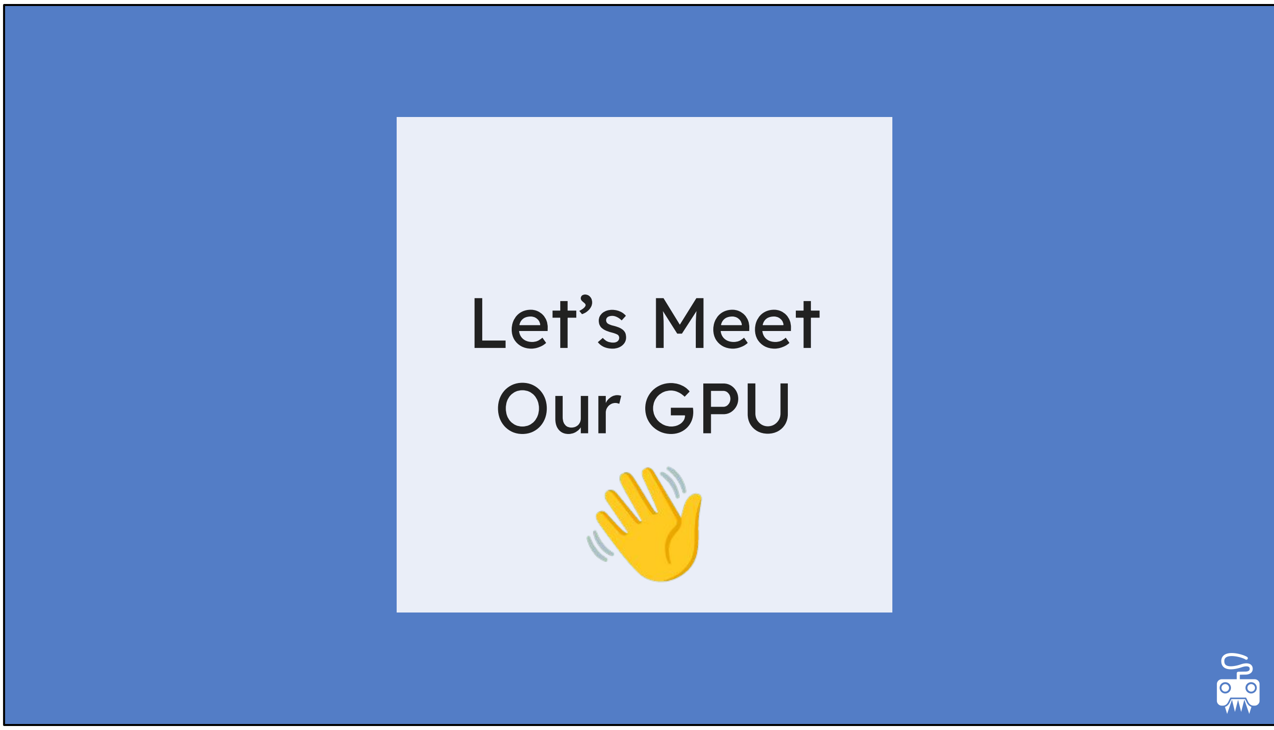
Today, I'll be talking about wave intrinsics. How they can be implemented and how you can use them to enable strange and powerful algorithms.

What Will You Get Out Of This Talk?

- An intuition for how wave intrinsics are implemented
- An intuition for when you want to use wave intrinsics
- Examples of common and novel use cases
- Ideas on how you can develop your own wave-based algorithms

As a part of this talk, I'm hoping that you will be able to develop an intuition for how existing wave intrinsics are implemented and when you want to use them.

On top of that, we will be looking at common and novel use cases of wave intrinsics.



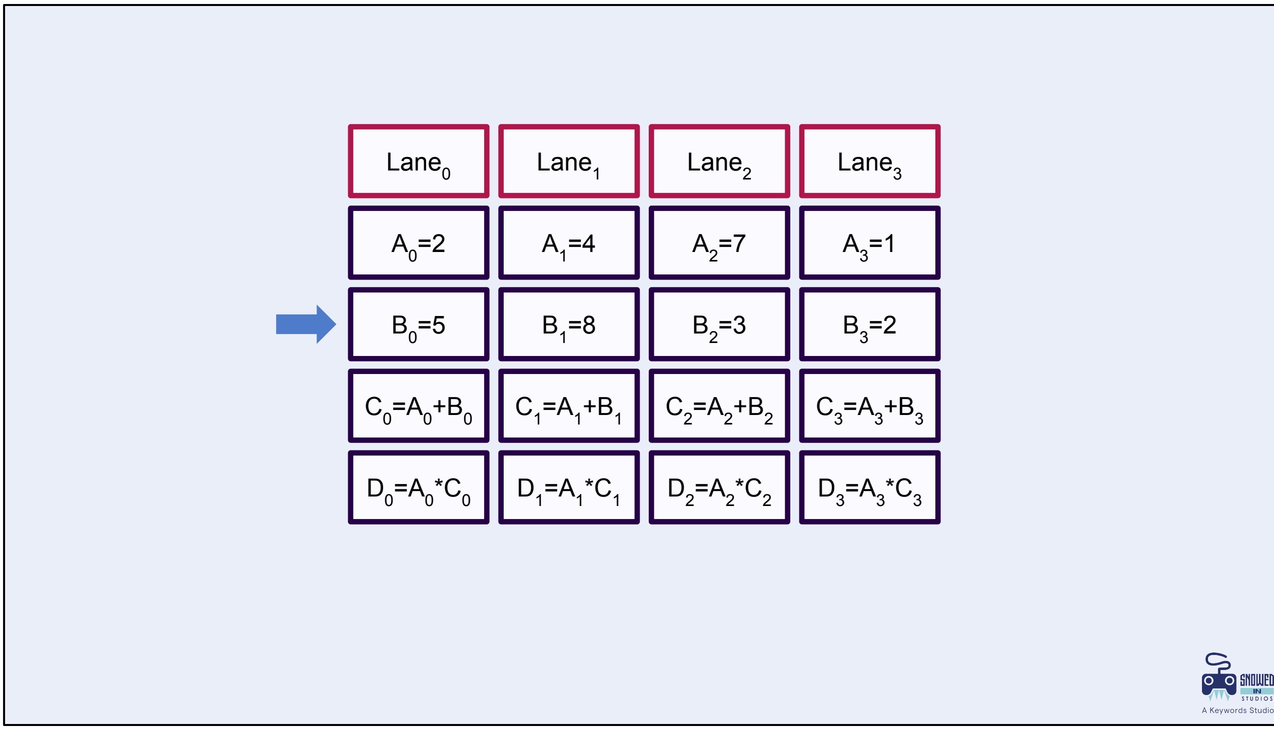
Let’s meet the imaginary GPU we’re going to be using for our demonstration today.

$Lane_0$	$Lane_1$	$Lane_2$	$Lane_3$
$A_0=2$	$A_1=4$	$A_2=7$	$A_3=1$
$B_0=5$	$B_1=8$	$B_2=3$	$B_3=2$
$C_0=A_0+B_0$	$C_1=A_1+B_1$	$C_2=A_2+B_2$	$C_3=A_3+B_3$
$D_0=A_0*C_0$	$D_1=A_1*C_1$	$D_2=A_2*C_2$	$D_3=A_3*C_3$

(Click through while talking) This mock GPU has 4 lanes each processing an independent stream of information.

It can do everything else a GPU can.

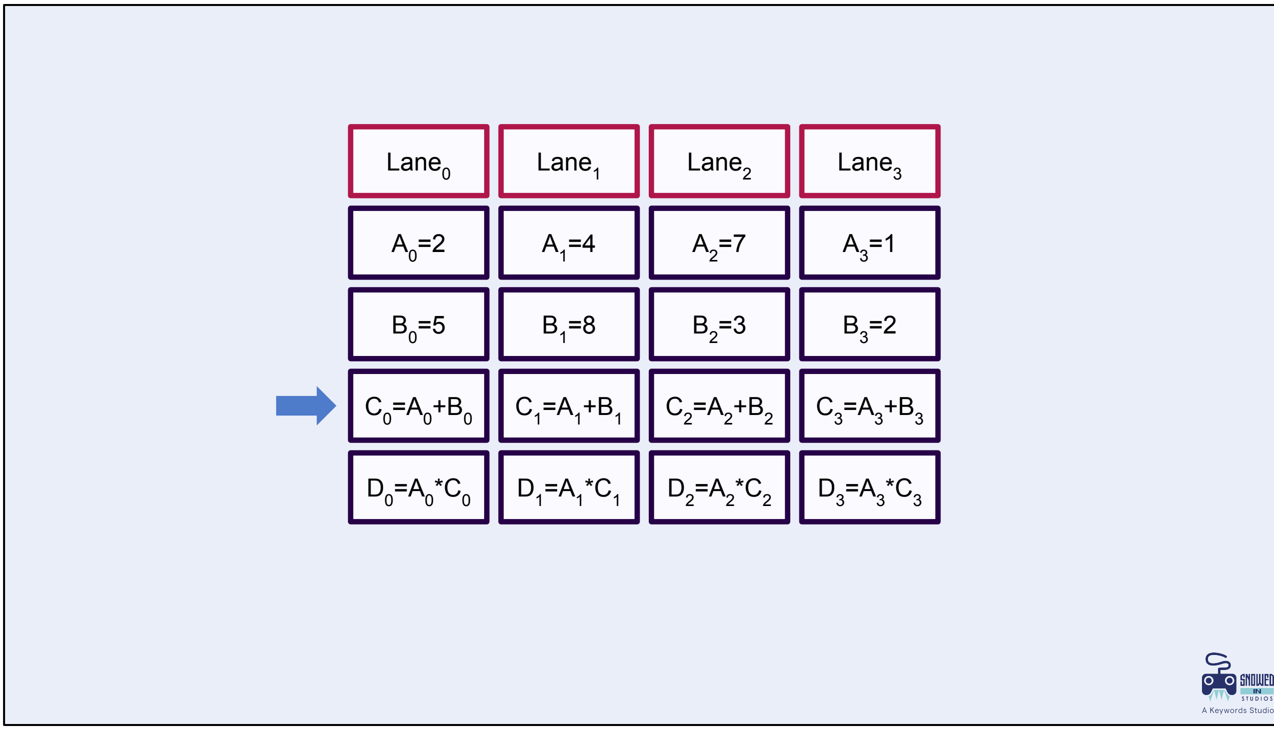
It just has fewer lanes than you would expect!



(Click through while talking) This mock GPU has 4 lanes each processing an independent stream of information.

It can do everything else a GPU can.

It just has fewer lanes than you would expect!



(Click through while talking) This mock GPU has 4 lanes each processing an independent stream of information.

It can do everything else a GPU can.

It just has fewer lanes than you would expect!

$Lane_0$	$Lane_1$	$Lane_2$	$Lane_3$
$A_0=2$	$A_1=4$	$A_2=7$	$A_3=1$
$B_0=5$	$B_1=8$	$B_2=3$	$B_3=2$
$C_0=A_0+B_0$	$C_1=A_1+B_1$	$C_2=A_2+B_2$	$C_3=A_3+B_3$
$D_0=A_0*C_0$	$D_1=A_1*C_1$	$D_2=A_2*C_2$	$D_3=A_3*C_3$

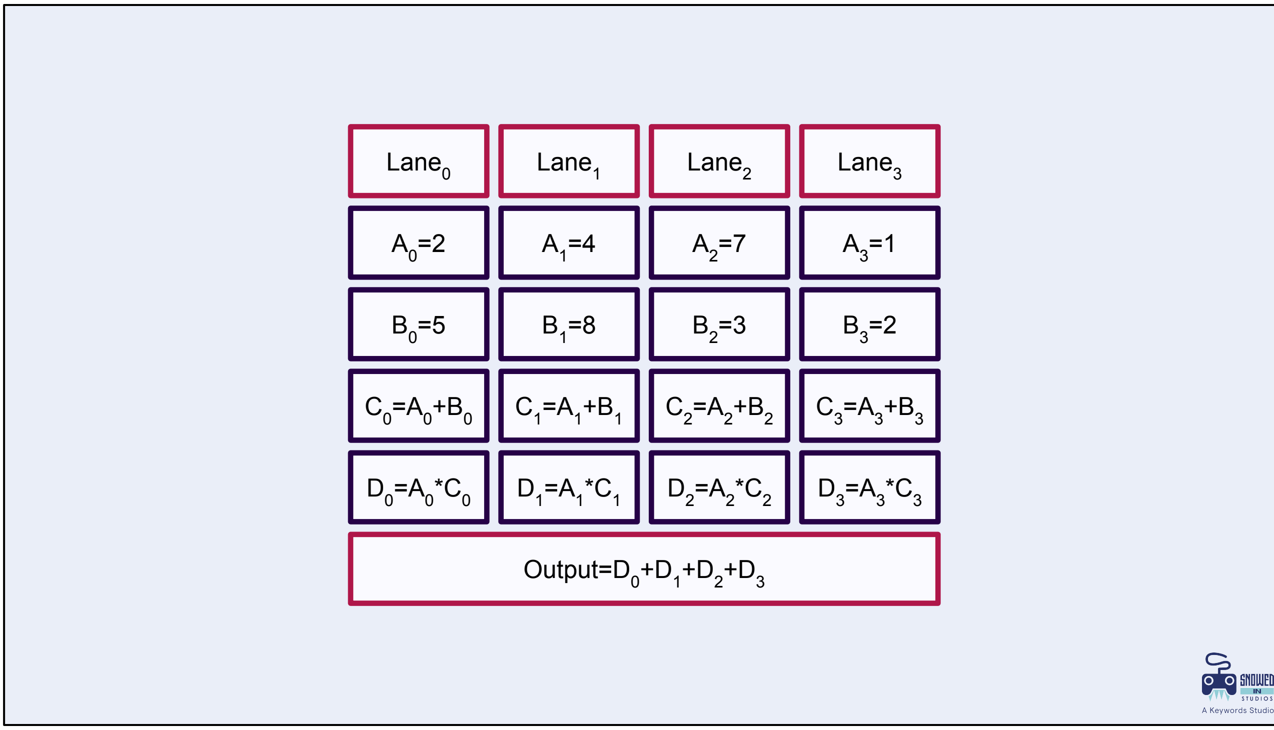
(Click through while talking) This mock GPU has 4 lanes each processing an independent stream of information.

It can do everything else a GPU can.

It just has fewer lanes than you would expect!

What if we want to share across these lanes?

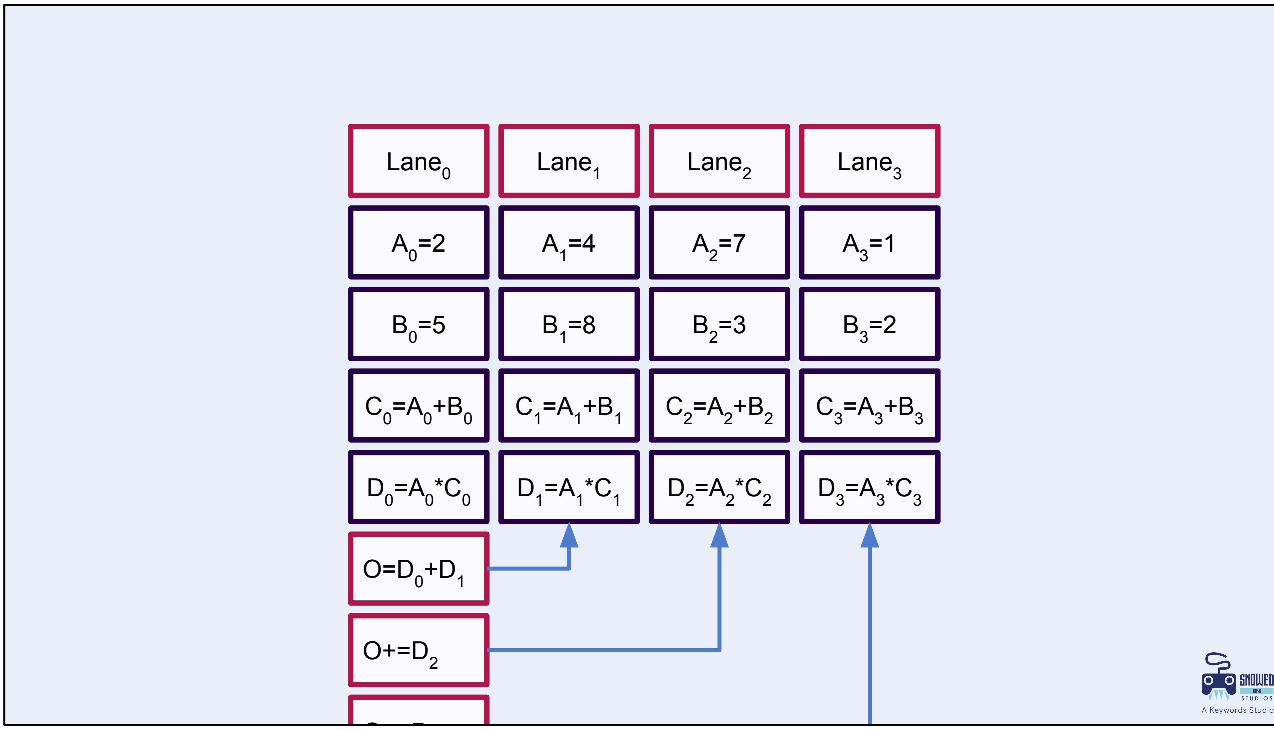
What if being completely independent isn't good enough?



Let’s say that at the end of our program, we want to accumulate the values of each of our lanes in a single value.

click

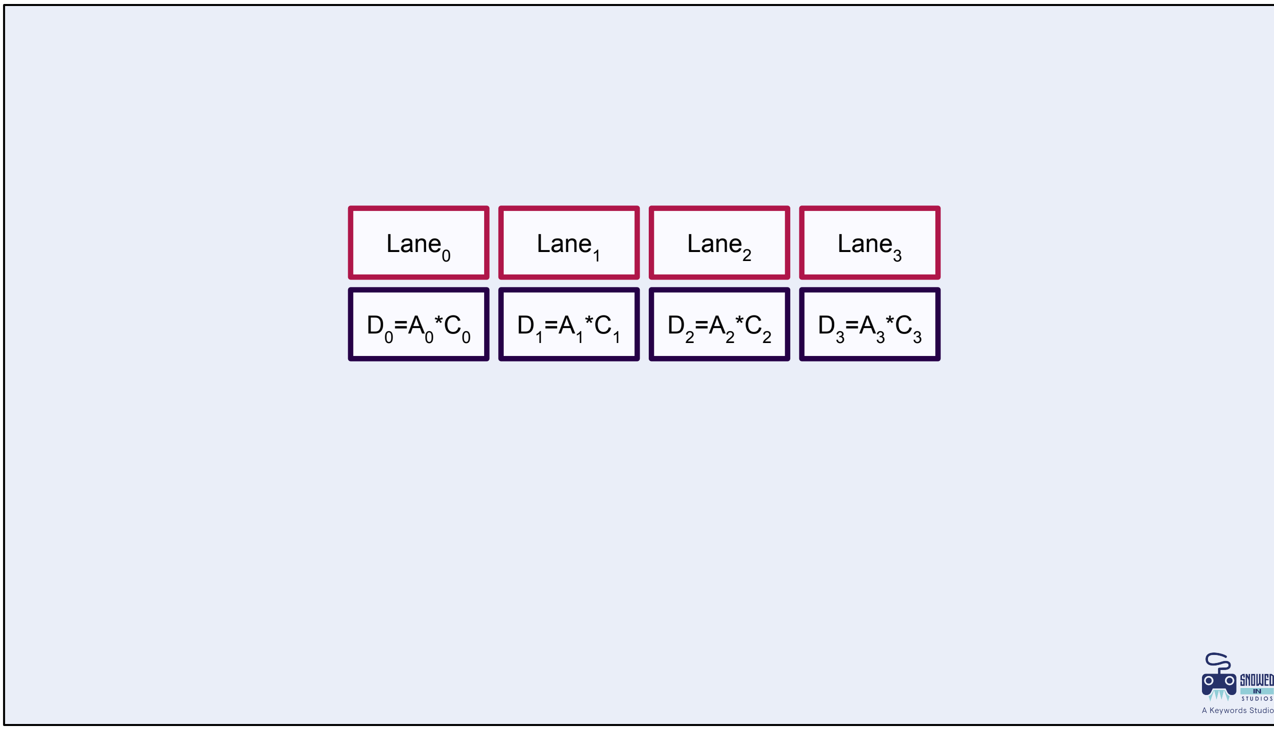
We could do that simply by iterating through each element in a single lane and adding them to our output.



Let's say that at the end of our program, we want to accumulate the values of each of our lanes in a single value.

click

We could do that simply by iterating through each element in a single lane and adding them to our output.



But there’s a better way.

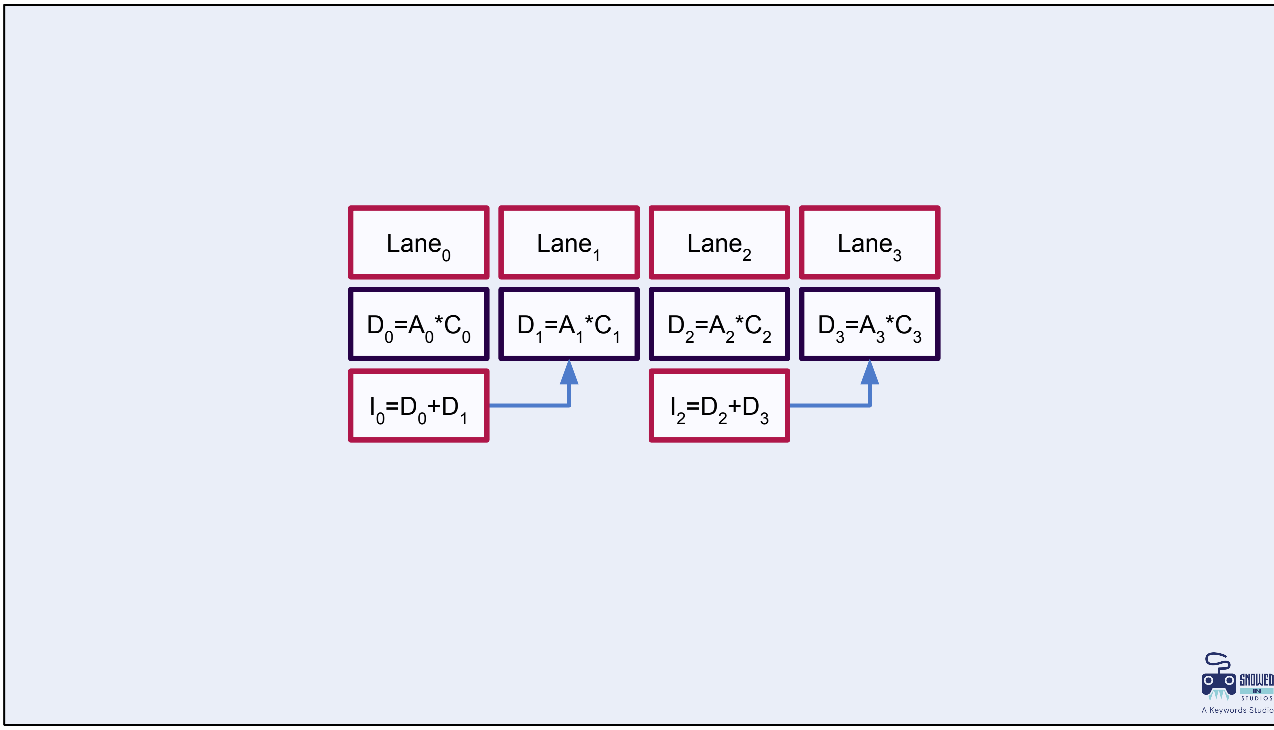
Instead, we can have some of our lanes participate in the final summation by having them accumulate the values of their neighbours.

Starting with their nearest neighbour.

click

And then doubling the distance to the next neighbour.

click



But there's a better way.

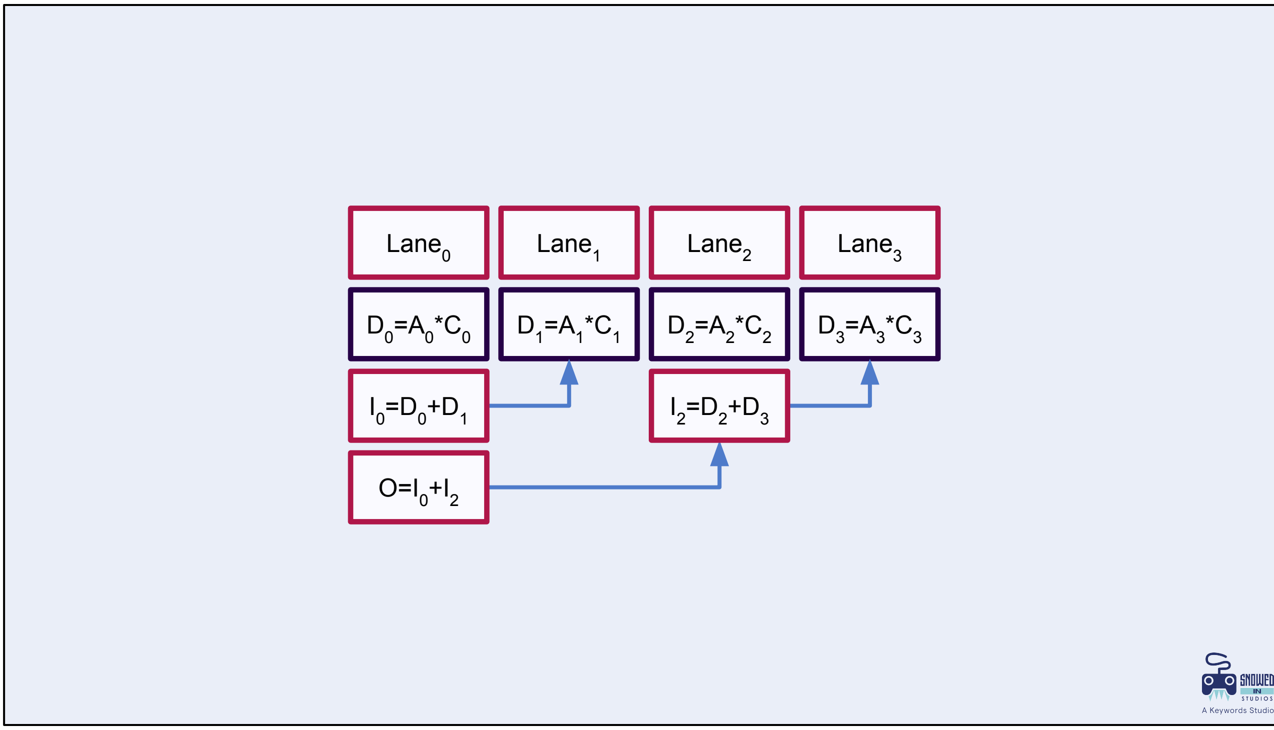
Instead, we can have some of our lanes participate in the final summation by having them accumulate the values of their neighbours.

Starting with their nearest neighbour.

click

And then doubling the distance to the next neighbour.

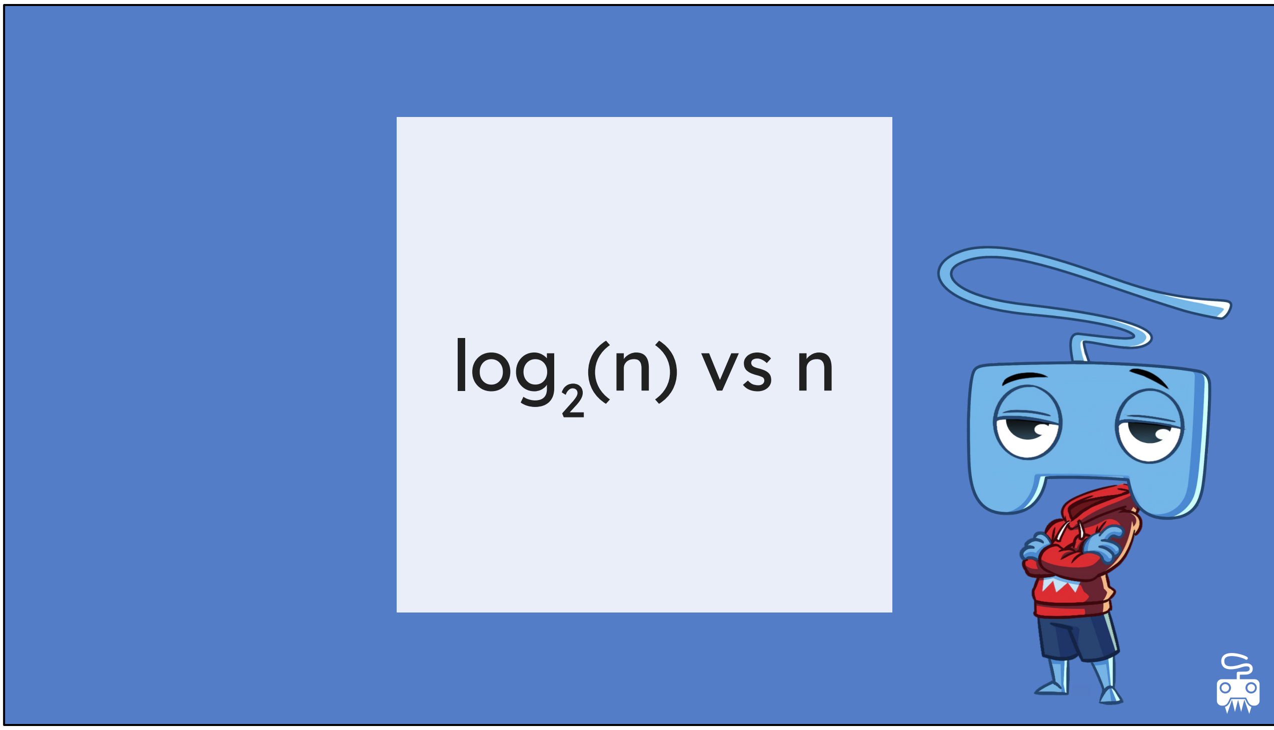
click



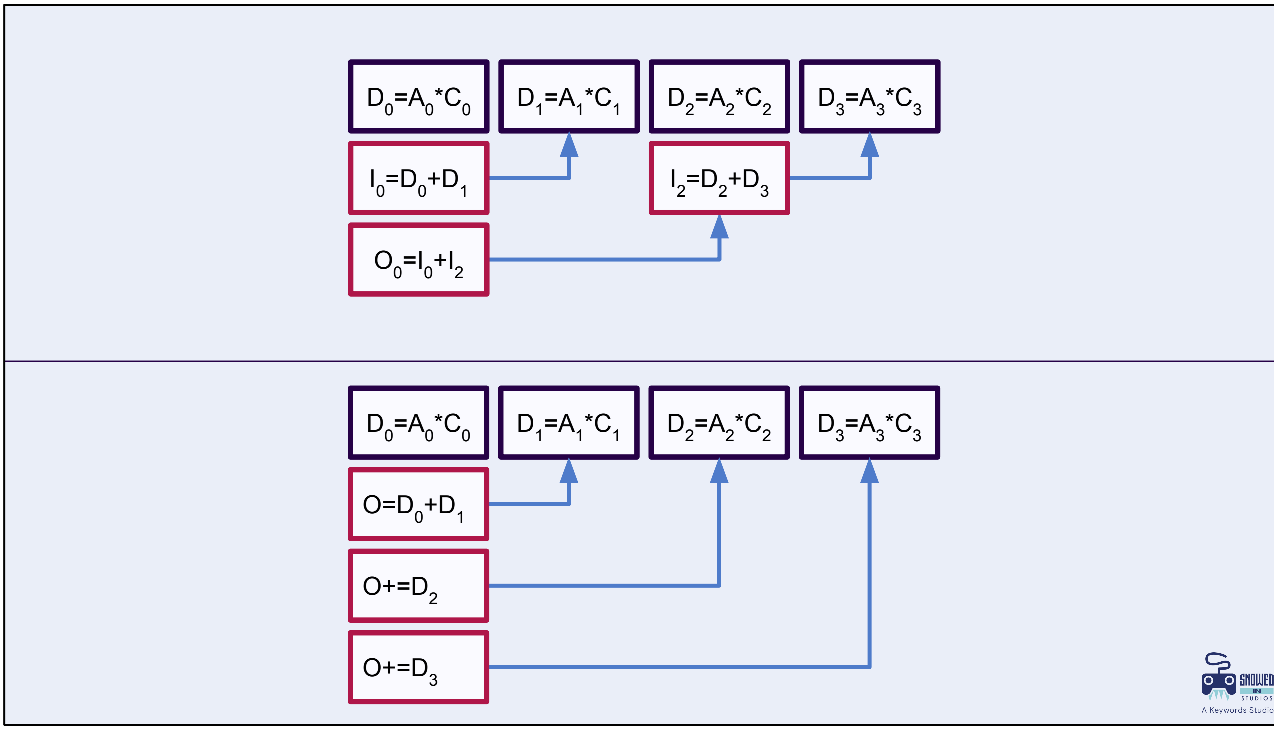
This summation ends at the second step but you can continue this process.

For 8 elements, the next stage would look at the neighbours at strides of 4.

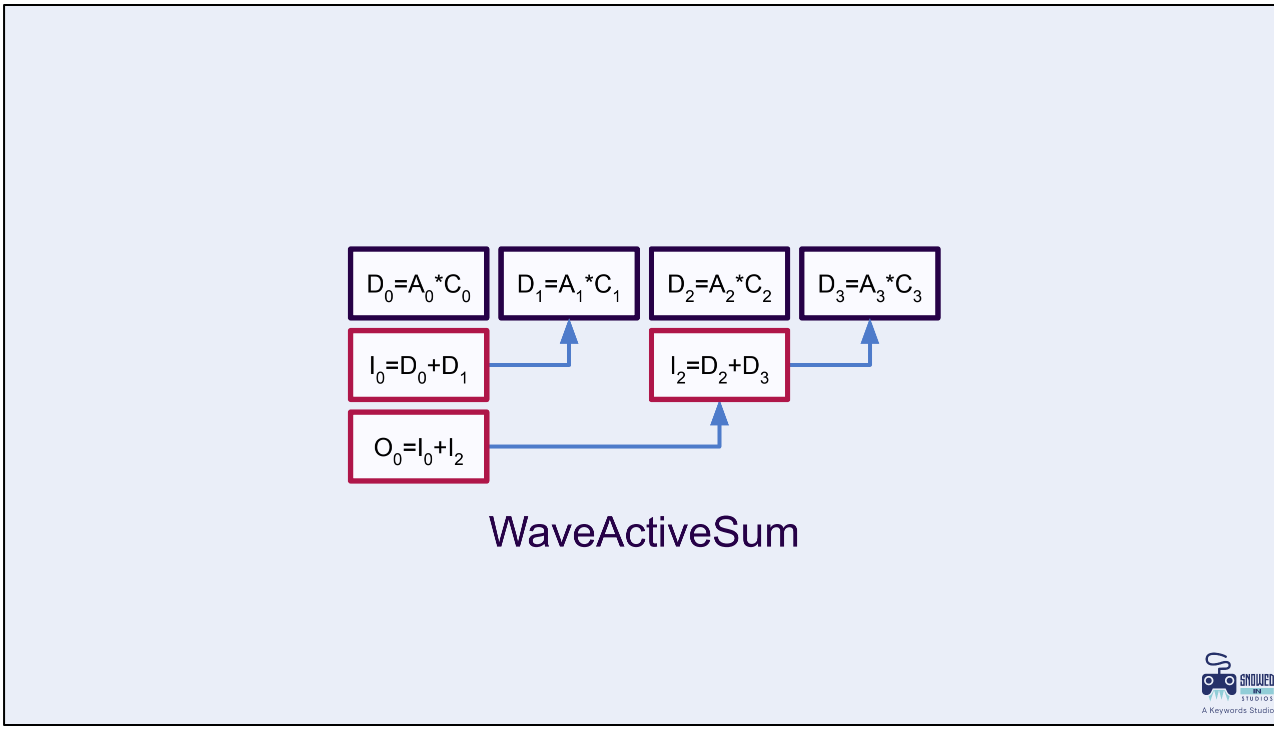
For 16, an additional stage with a stride of 8.



We’ve turned a linear problem into a logarithmic one.



We save 1 iteration. In the case of a wave of size 32, we save 26.



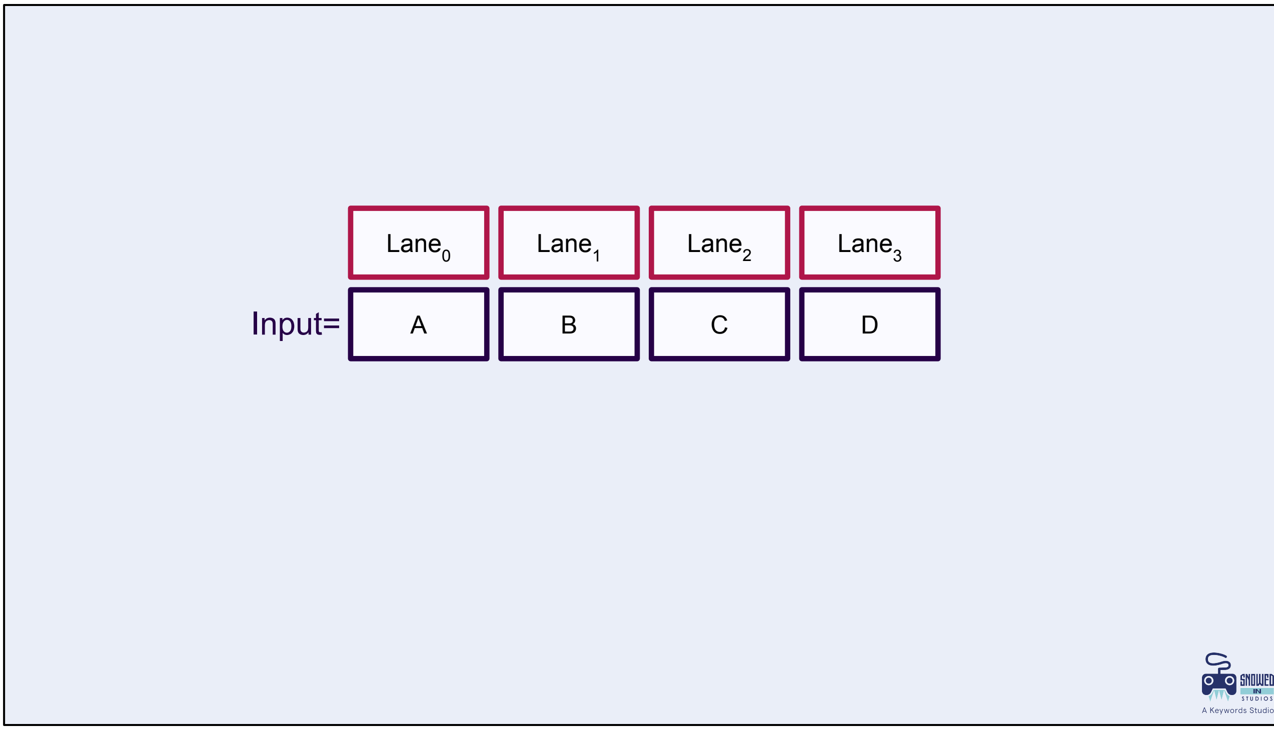
In HLSL, you may have accessed this operation using the `WaveActiveSum` intrinsic.

But why do we care about `WaveActiveSum` and its implementation?

Let’s take the algorithm one step further and use a common example used in many games.

Stream Compaction!

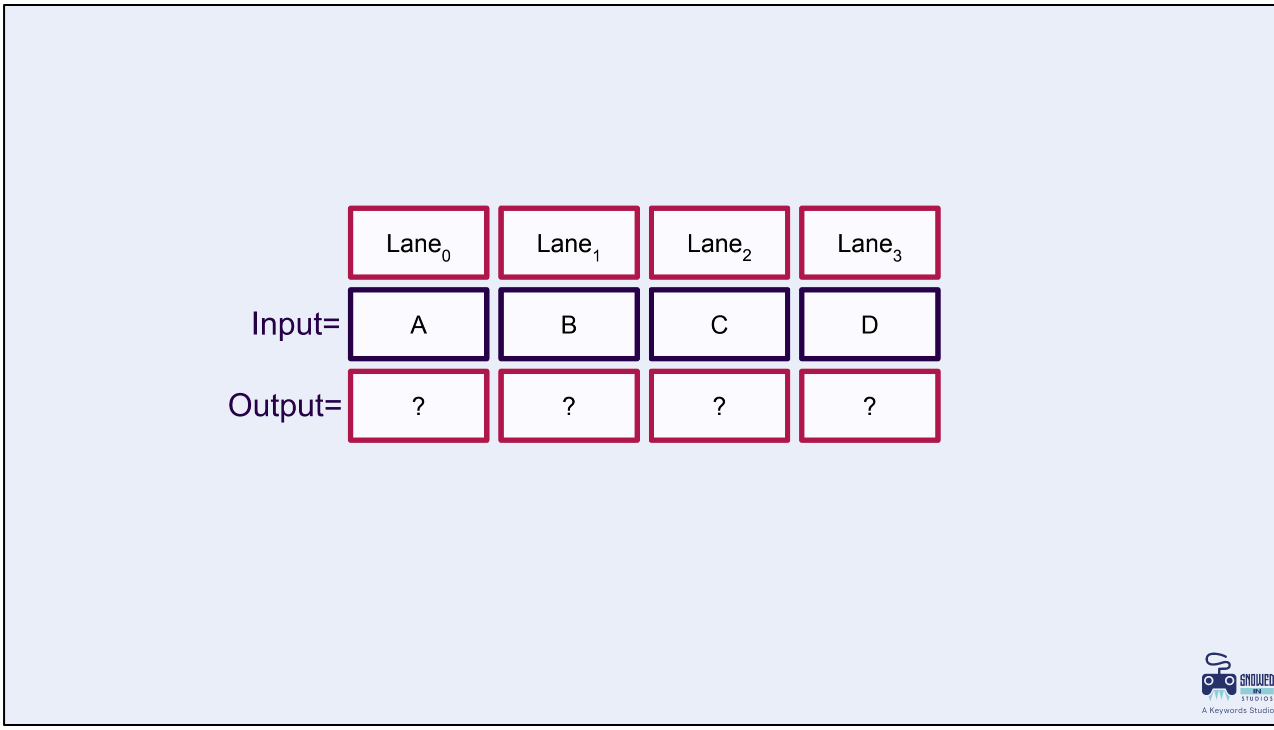
If you’ve done any form of GPU driven rendering, you almost definitely have used some form of stream compaction.



Let’s look at a short example.

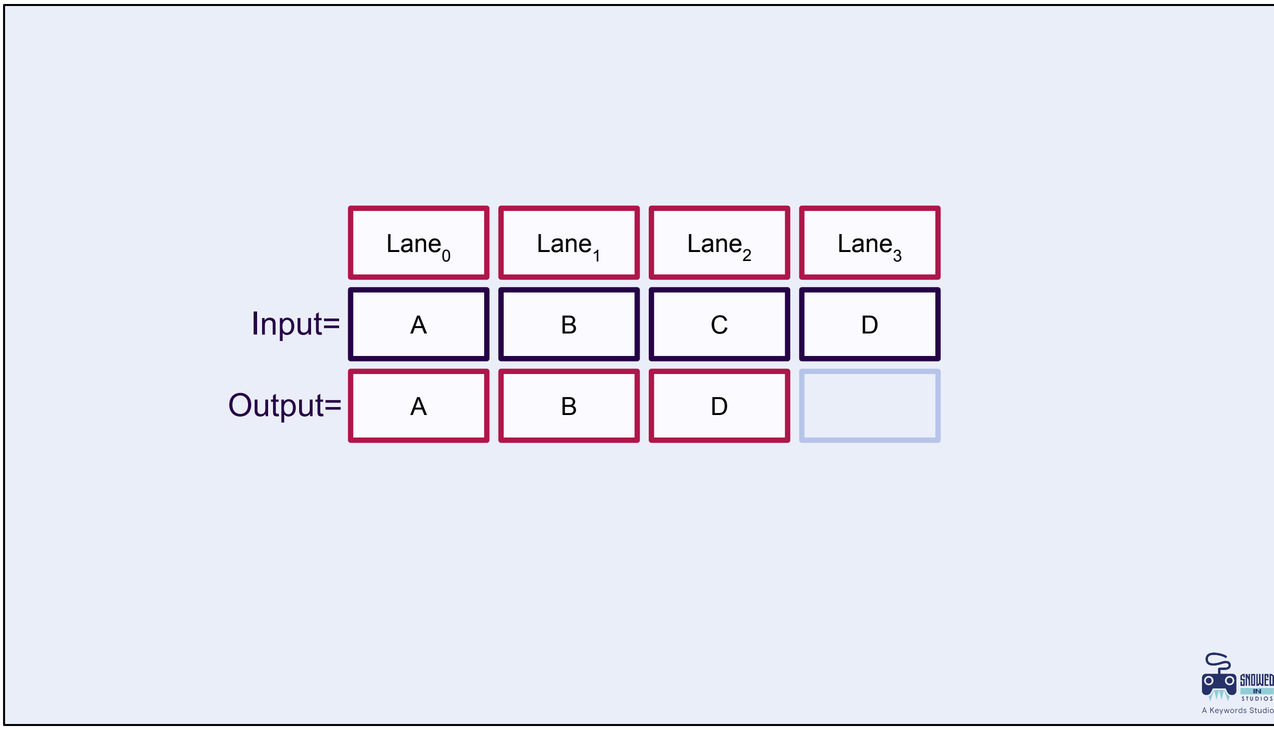
I have 4 letters.

But I want to write out only my favourite letters to an output buffer.



My favourite letters are A, B and D.

We don't like C.



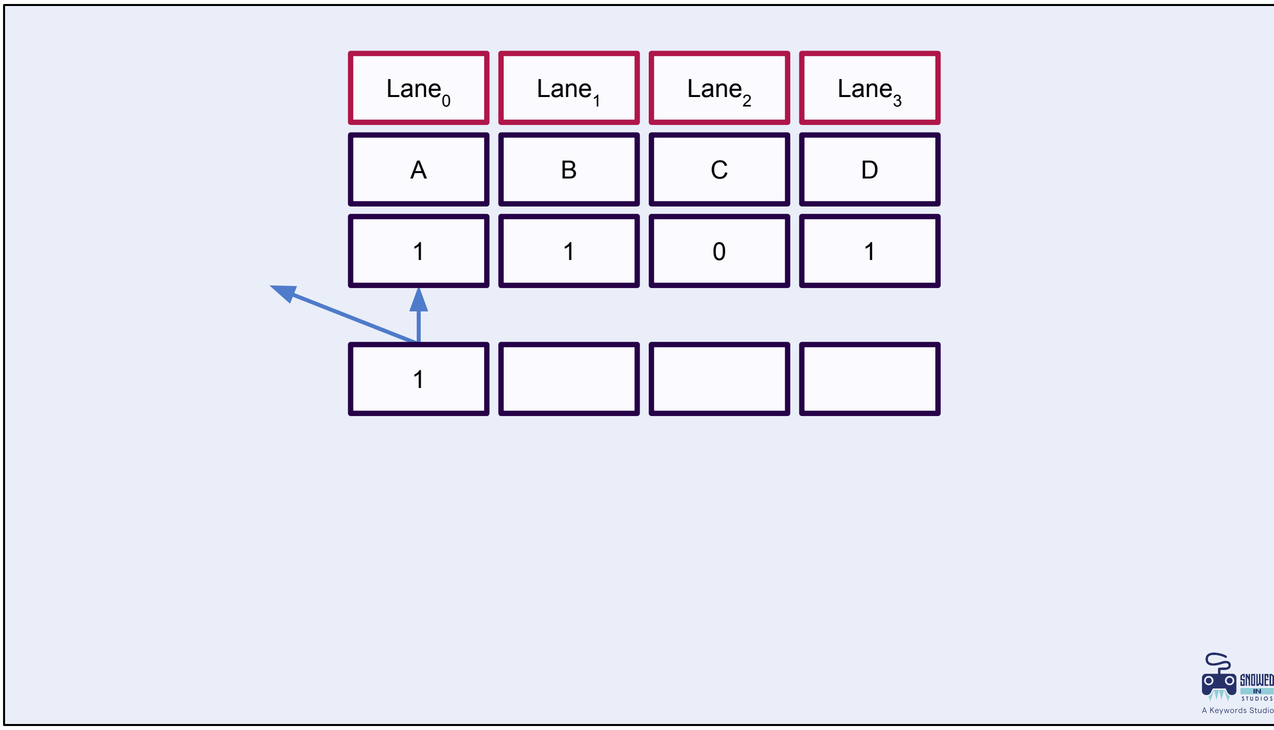
We know that our output buffer should look like this.

But how do we get there?

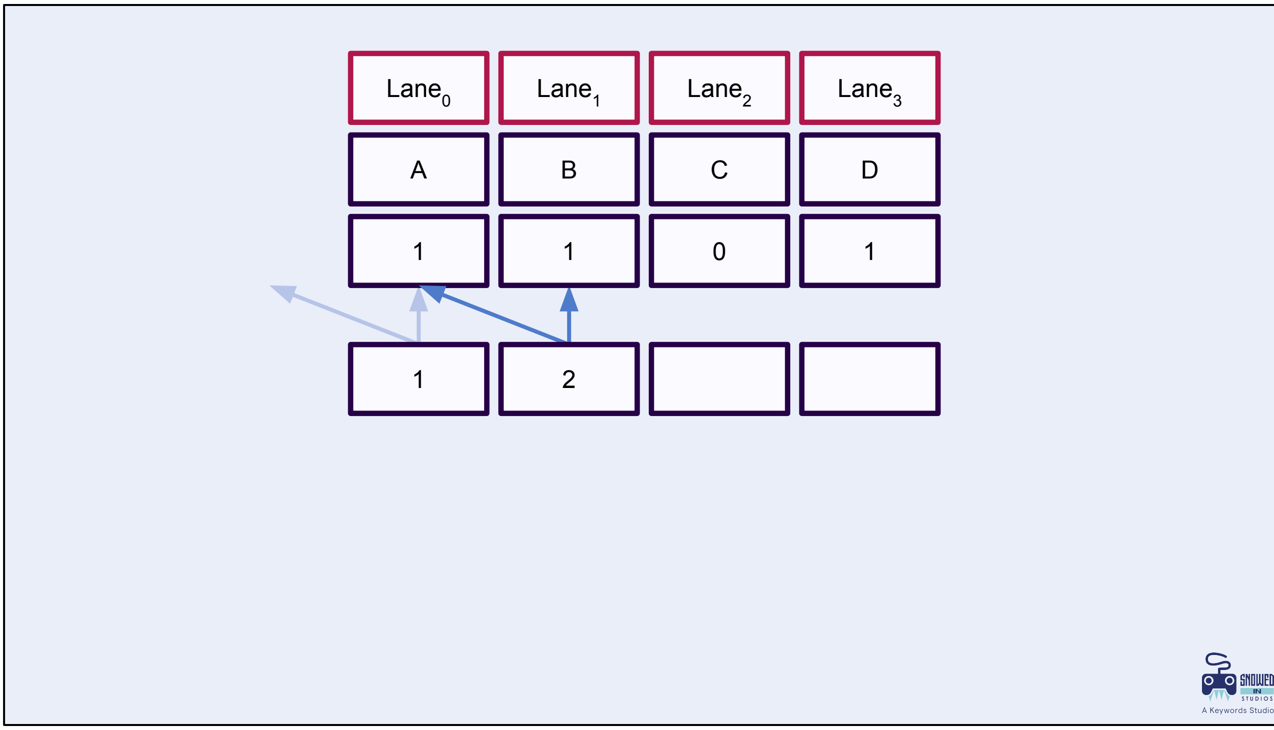
$Lane_0$	$Lane_1$	$Lane_2$	$Lane_3$
A	B	C	D
1	1	0	1

Let’s mark each lane with a 1 if its a favourite letter and a 0 if its not.

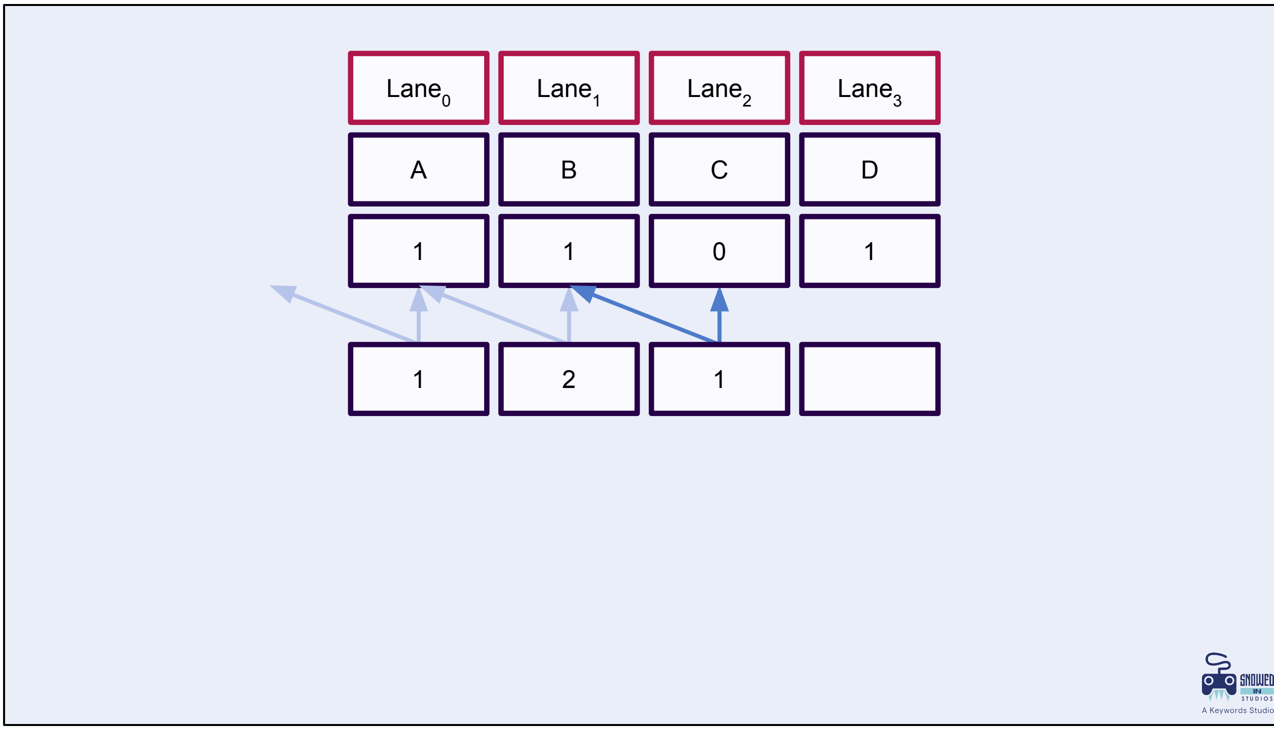
Maybe we just call an `isFavourite` function to get this result.



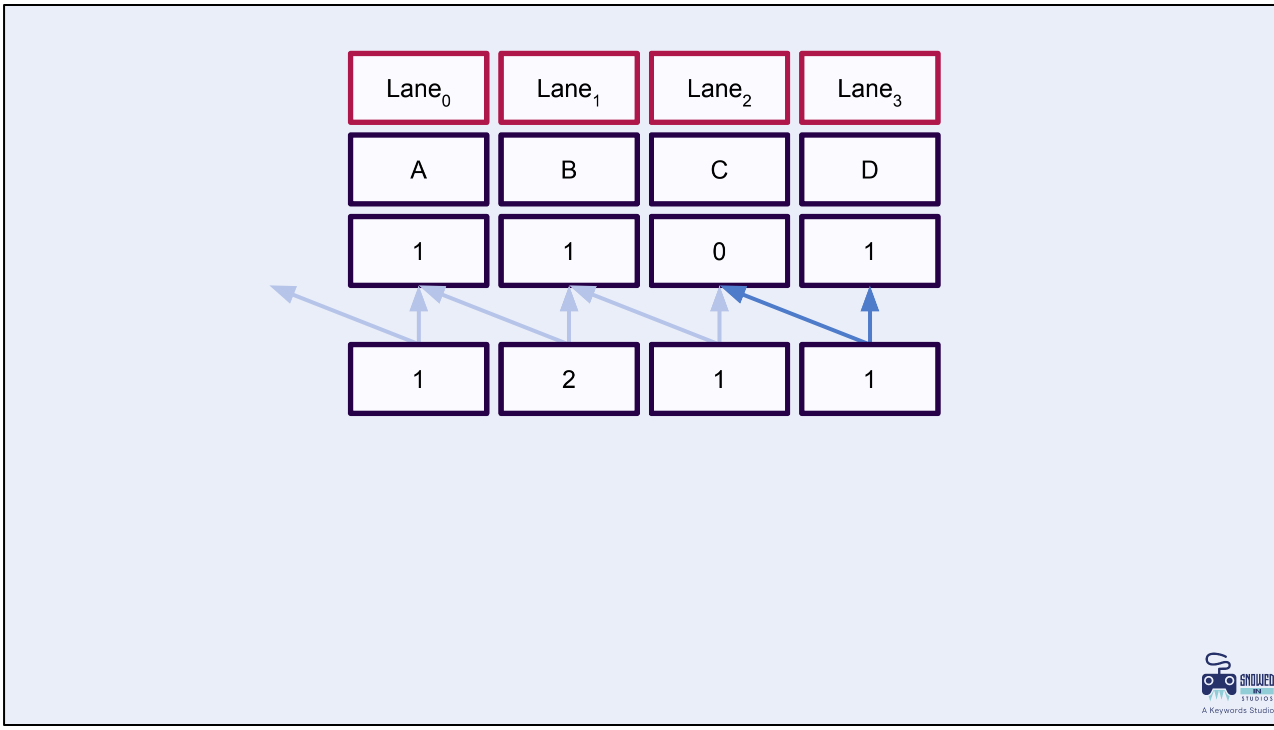
(Click through while talking) Starting from the same algorithm we used for WaveActiveSum, each neighbour reads and adds the value of its neighbour to itself.



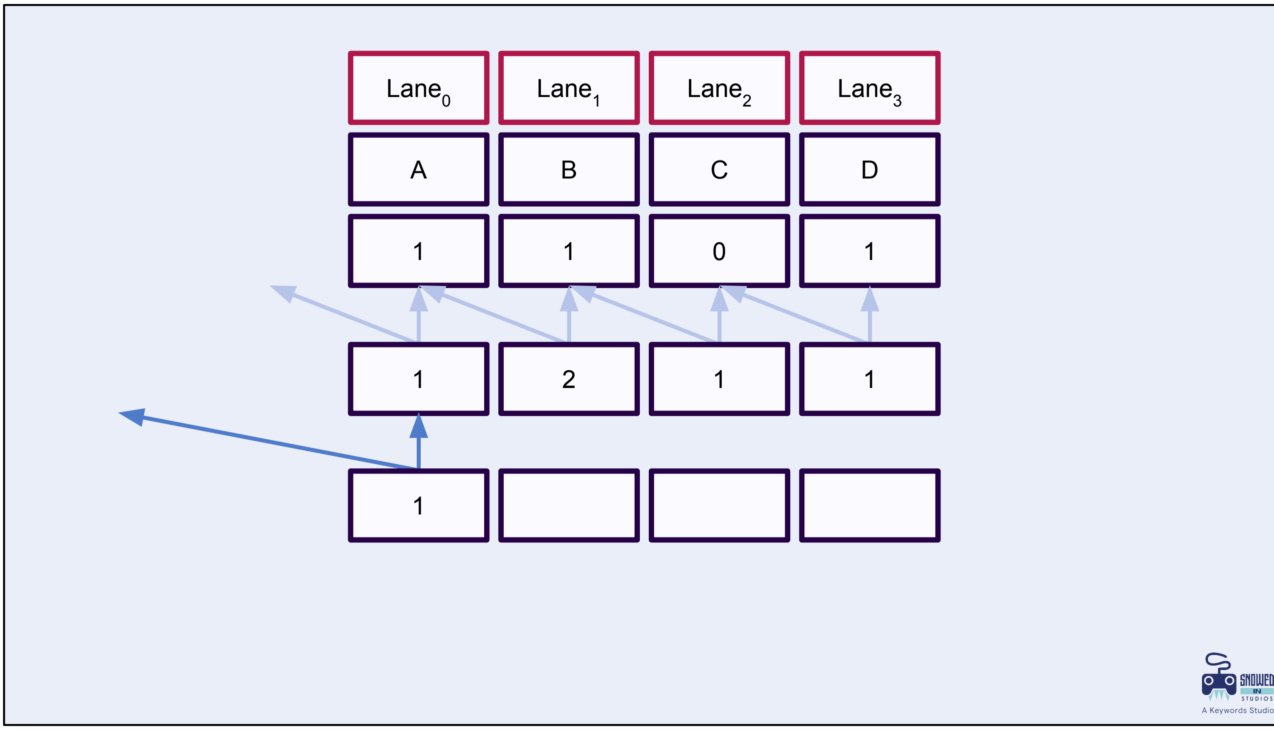
(Click through while talking) Starting from the same algorithm we used for WaveActiveSum, each neighbour reads and adds the value of its neighbour to itself.



(Click through while talking) Starting from the same algorithm we used for WaveActiveSum, each neighbour reads and adds the value of its neighbour to itself.

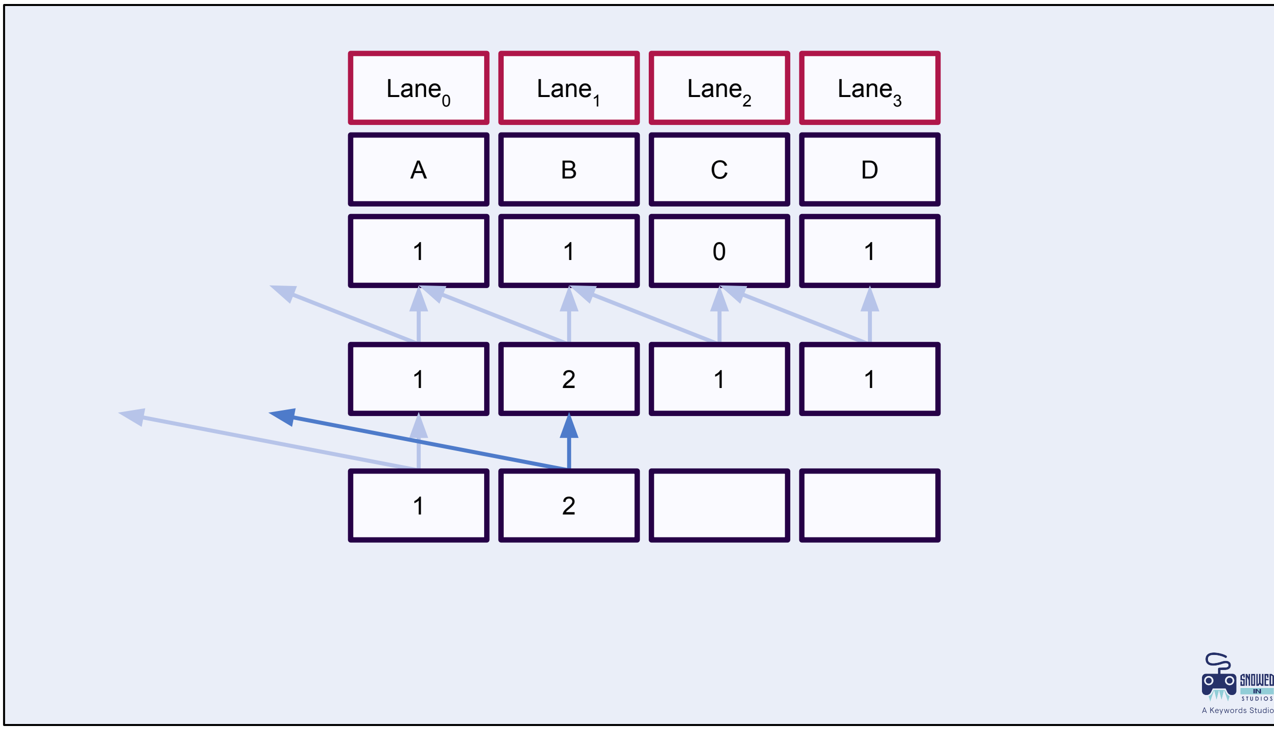


(Click through while talking) Starting from the same algorithm we used for WaveActiveSum, each neighbour reads and adds the value of its neighbour to itself.



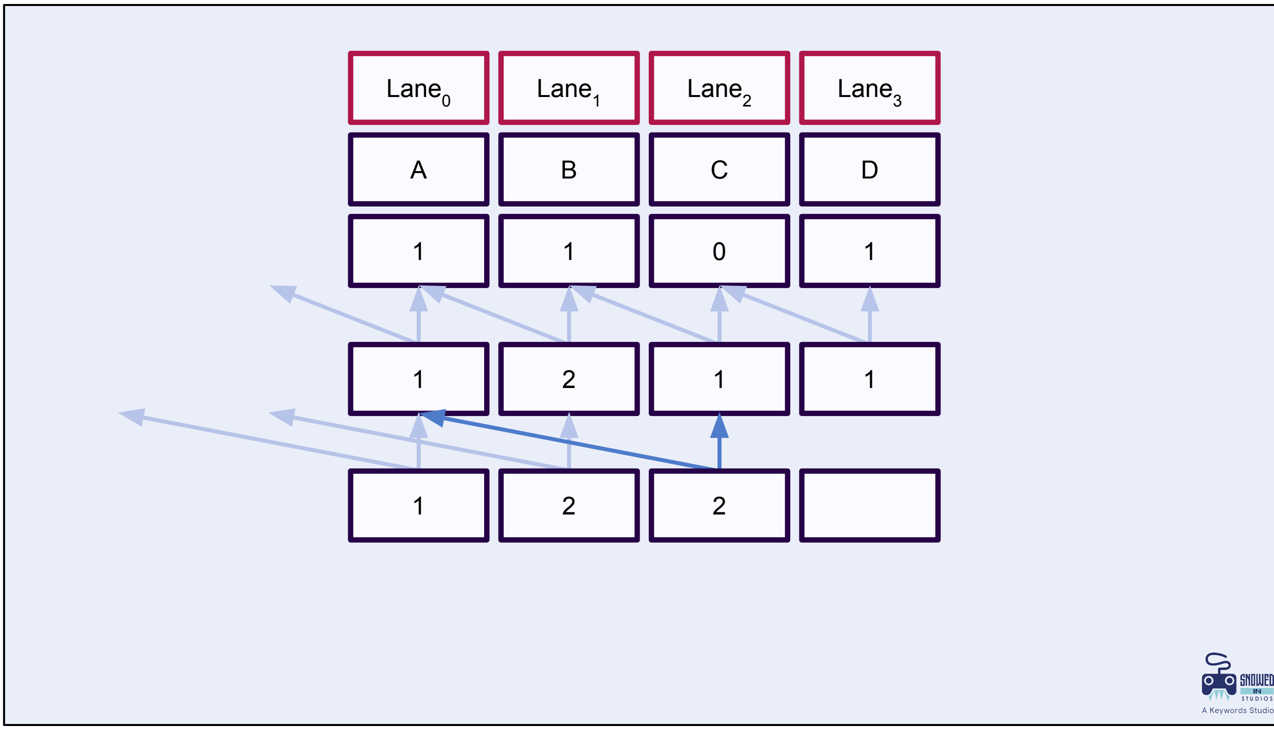
(Click through while talking) We then accumulate the results of our neighbours at twice the distance.

You'll notice that each value is slowly accumulating the sum of all of its preceding neighbours.



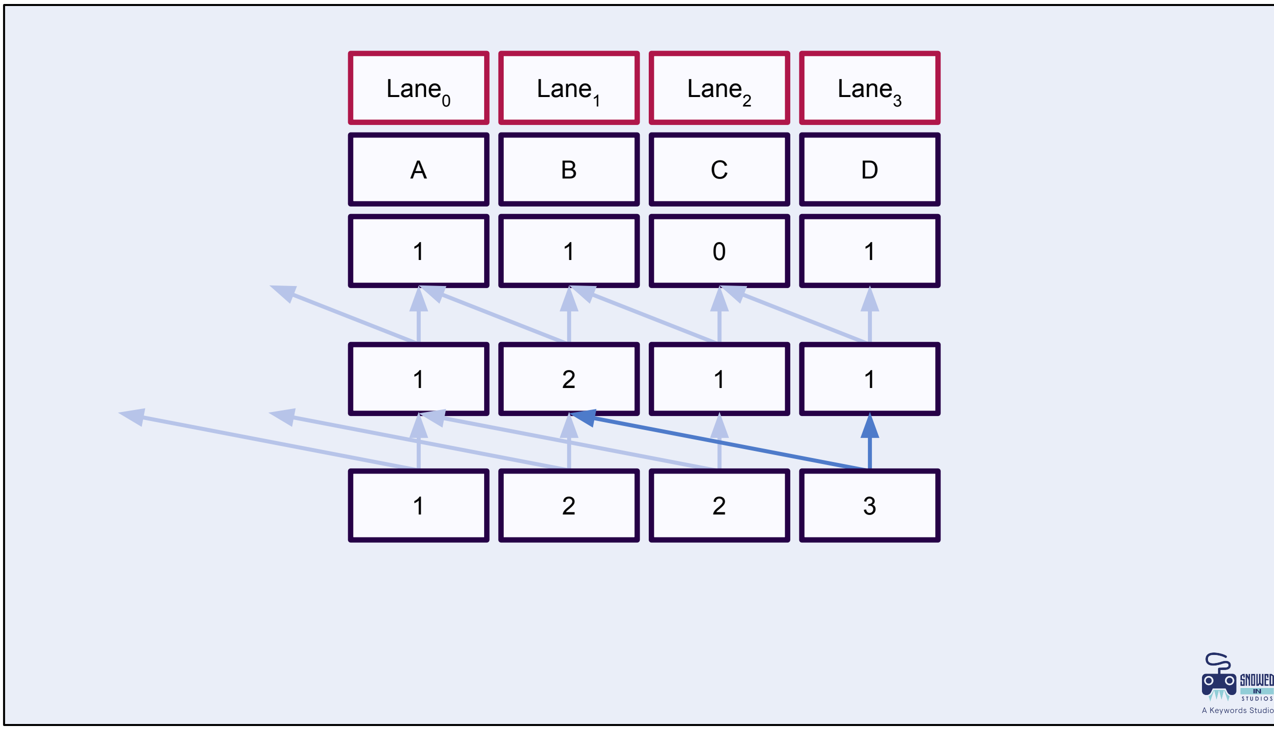
(Click through while talking) We then accumulate the results of our neighbours at twice the distance.

You’ll notice that each value is slowly accumulating the sum of all of its preceding neighbours.



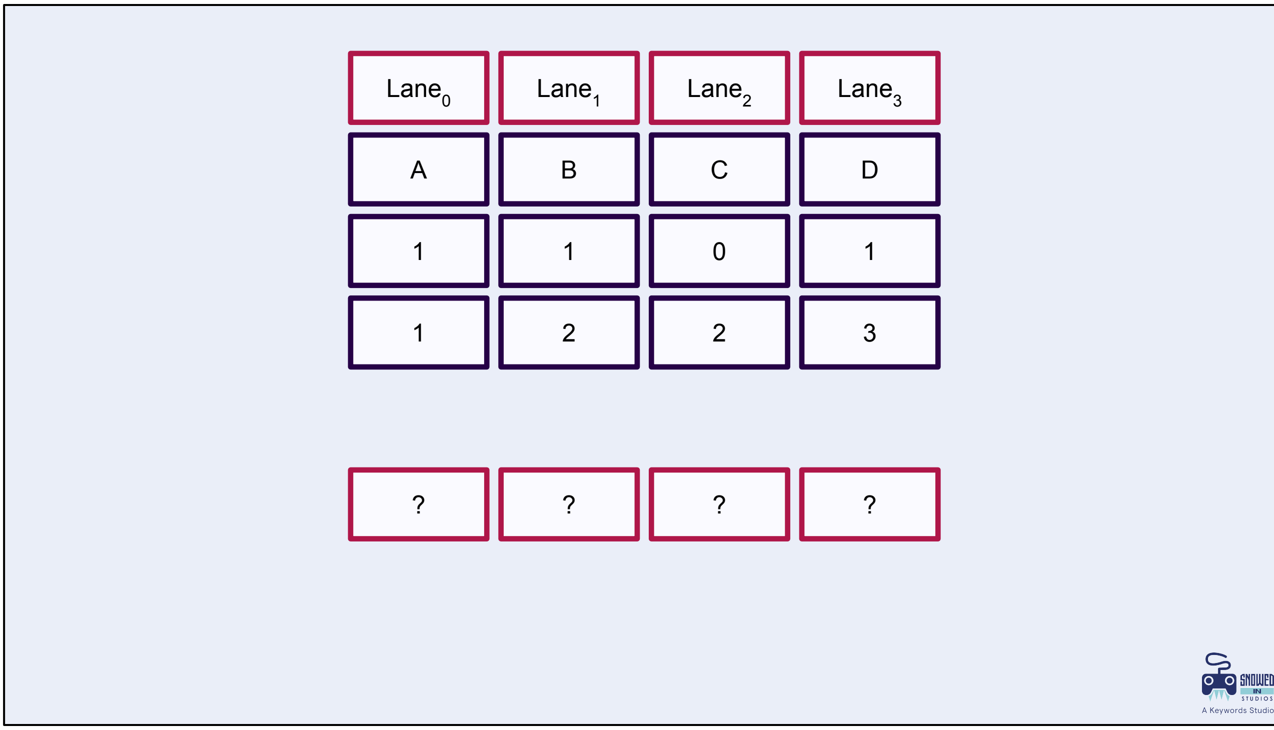
(Click through while talking) We then accumulate the results of our neighbours at twice the distance.

You'll notice that each value is slowly accumulating the sum of all of its preceding neighbours.



(Click through while talking) We then accumulate the results of our neighbours at twice the distance.

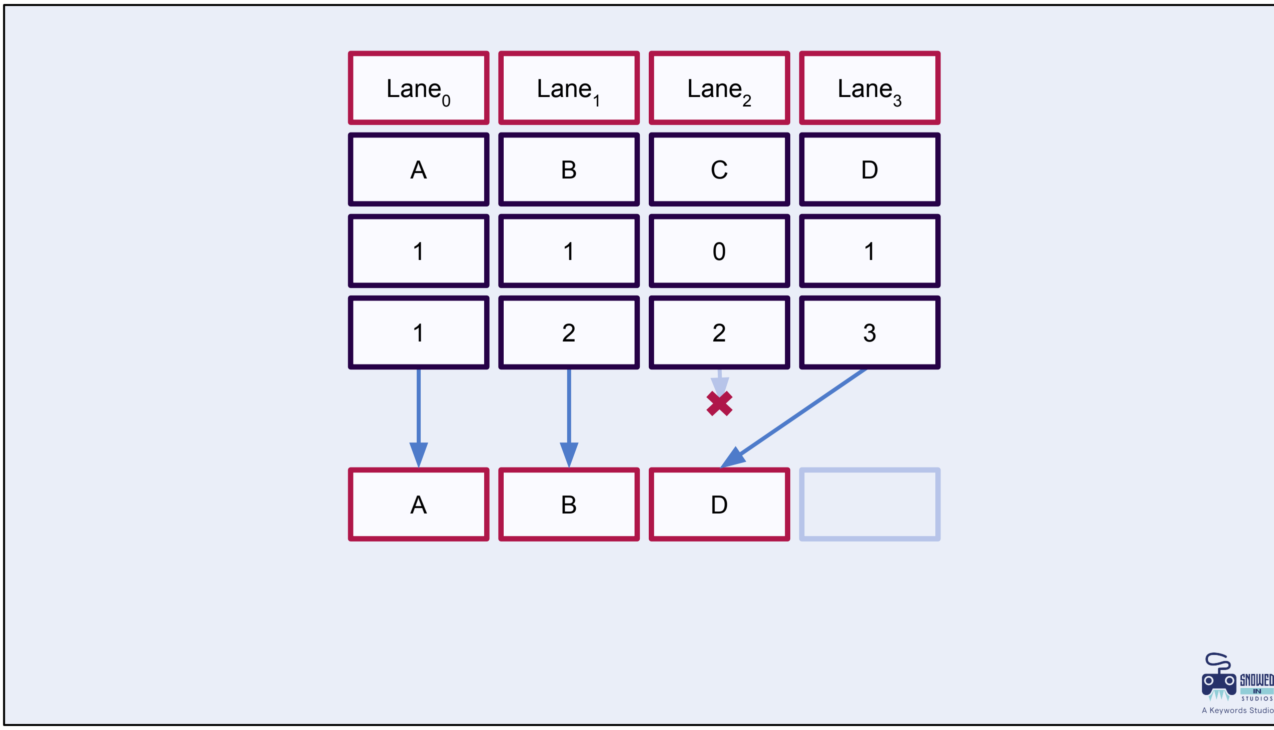
You'll notice that each value is slowly accumulating the sum of all of its preceding neighbours.



Once we’ve accumulated the results of our neighbours, we know how many values will be written before our own.

click

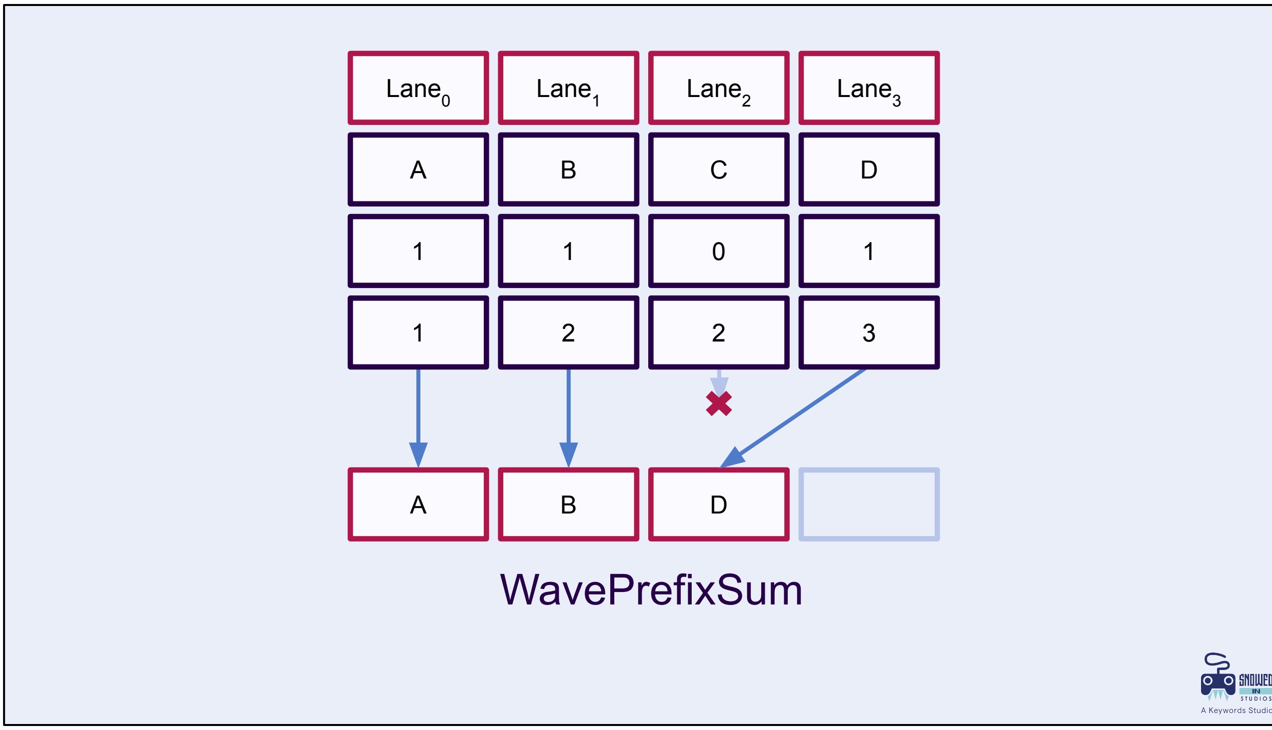
Then we subtract 1 from our sum and write to that index if it’s one of my favourite letters.



Once we've accumulated the results of our neighbours, we know how many values will be written before our own.

click

Then we subtract 1 from our sum and write to that index if it's one of my favourite letters.



To use this operation, you simply have to use `WavePrefixSum`.

Note to reader: We actually described an inclusive prefix sum which includes our current element, while WavePrefixSum is an exclusive prefix sum which excludes the current element.

Stream compaction, and as a result a prefix sum, is widely used for a variety of problems. Especially for GPU culling of meshlets, lights and other primitives.

Already Built For You!

- **WaveActiveSum->subgroupAdd**
- **WavePrefixSum->subgroupExclusiveAdd**
- **And more!**
  - https://github.com/microsoft/directxshadercompiler/wiki/wave-intrinsics
  - https://www.khronos.org/blog/vulkan-subgroup-tutorial

SNOWED IN STUDIOS
A Keywords Studio

There are many more intrinsics for you to use and some excellent resources for discovering how to use them.

The references slide later on has a wide variety of excellent reading material.

How Does It Compile?

(With AMD's Offline Compiler)

Now that we've had a bit of an overview. How do these wave intrinsics compile?

Knowing how these compile can provide you with valuable insight into the potential performance characteristics of your algorithm.

For the rest of this talk, disassembly will be from AMD's offline compiler.

Let's look at a simple program.

```
StructuredBuffer<int> Input;
RWStructuredBuffer<int> Output;

[numthreads(32,1,1)]
void CS()
{
    int input = Input[WaveGetLaneIndex()];
    bool selectInput = Select(input);
    uint outputIndex = WavePrefixSum(selectInput ? 1 : 0);

    if(selectInput)
    {
        Output[outputIndex] = input;
    }
}
```

This program represents the algorithm we discussed a moment ago.

WavePrefixSum

```
;     uint outputIndex = WavePrefixSum(selectInput ? 1 : 0);
    v_mov_b32_e32 v1, v8                                                //
    s_not_b64 exec, exec                                                //
    v_mov_b32_e32 v1, 0                                                 //
    s_not_b64 exec, exec                                                //
    s_or_saveexec_b64 s[4:5], -1                                        //
    s_mov_b32 s0, 0x6543210f                                            //
    v_lshlrev_b64 v[2:3], v0, 1                                         //
    v_permlane16_b32 v1, v1, s0, 0xedcba987 op_sel:[1,0]                //
    v_and_b32_e32 v5, 0x10001, v3                                       //
    v_readlane_b32 s0, v1, 16                                           //
    v_and_b32_e32 v4, 0x10001, v2                                       //
    v_and_b32_e32 v3, 0xffff0000, v3                                    //
    v_and_b32_e32 v2, 0xffff0000, v2                                    //
    v_writelane_b32 v1, s0, 48                                          //
    v_cmp_eq_u64_e64 s0, 0, v[4:5]                                      //
    v_mov_b32_e32 v4, 0                                                 //
    v_mov_b32_e32 v5, 0                                                 //
    v_writelane_b32 v1, 0, 16                                           //
    v_permlanex16_b32 v6, v1, 0, -1 op_sel:[1,0]                        //
    v_cndmask_b32_e64 v1, v6, v1, s0                                    //
    v_cmp_ne_u64_e64 s0, 0, v[2:3]                                      //
    v_add_nc_u32_dpp v6, v1, v1 row_shr:1 row_mask:0xf bank_mask:
    v_mov_b32_dpp v4, v1 row_shr:2 row_mask:0xf bank_mask:0xf  //
    v_mov_b32_dpp v5, v1 row_shr:3 row_mask:0xf bank_mask:0xf  //
    v_add3_u32 v1, v6, v4, v5                                           //
    v_add_nc_u32_dpp v1, v1, v1 row_shr:4 row_mask:0xf bank_mask:
    v_add_nc_u32_dpp v1, v1, v1 row_shr:8 row_mask:0xf bank_mask:
    v_permlanex16_b32 v4, v1, -1, -1 op_sel:[1,0]                       //
    v_cndmask_b32_e64 v2, 0, v4, s0                                     //
    v_cmp_lt_u32_e64 s0, 31, v0                                         //
    v_add_nc_u32_e32 v1, v2, v1                                         //
    v_readlane_b32 s6, v1, 31                                           //
    v_cndmask_b32_e64 v0, 0, s6, s0                                     //
    v_add_nc_u32_e32 v0, v1, v0                                         //
    s_mov_b64 exec, s[4:5]                                              //
    v_mov_b32_e32 v8, v0                                                //
```

~40 Instructions

If we look at the disassembly for `WavePrefixSum` you'll see that it's quite a bit of code!

Unfortunately, it's a little bit more complicated than "read your neighbour's value"...

I'll admit…

I lied to you.

We don't need to use `WavePrefixSum` for this problem at all.

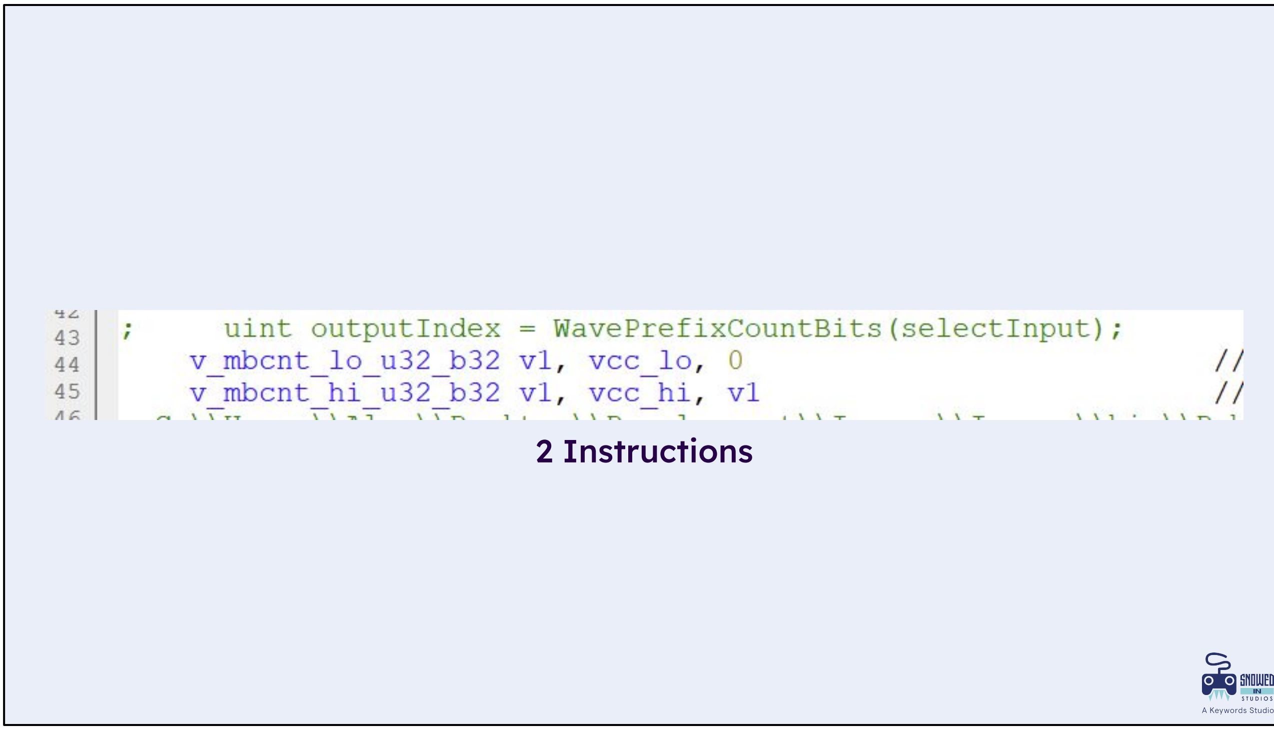
Since our values are 0 and 1, we can actually use the much leaner `WavePrefixCountBits`

```
StructuredBuffer<int> Input;
RWStructuredBuffer<int> Output;

[numthreads(32,1,1)]
void CS()
{
    int input = Input[WaveGetLaneIndex()];
    bool selectInput = Select(input);
    uint outputIndex = WavePrefixCountBits(selectInput);

    if(selectInput)
    {
        Output[outputIndex] = input;
    }
}
```

`WavePrefixCountBits` will provide us the same results.

But on RDNA, compiles to a much leaner two instructions.

click



`WavePrefixCountBits` will provide us the same results.

But on RDNA, compiles to a much leaner two instructions.

click

```
;     uint outputIndex = WavePrefixSum(selectInput ? 1 : 0);
    v_mov_b32_e32 v1, v8                                        //
    s_not_b64 exec, exec                                        //
    v_mov_b32_e32 v1, 0                                         //
    s_not_b64 exec, exec                                        //
    s_or_saveexec_b64 s[4:5], -1                                //
    s_mov_b32 s0, 0x6543210f                                    //
    v_lshlrev_b64 v[2:3], v0, 1                                 //
    v_permlane16_b32 v1, v1, s0, 0xedcba987 op_sel:[1,0]        //
    v_and_b32_e32 v5, 0x10001, v3                               //
    v_readlane_b32 s0, v1, 16                                   //
    v_and_b32_e32 v4, 0x10001, v2                               //
    v_and_b32_e32 v3, 0xffff0000, v3                            //
    v_and_b32_e32 v2, 0xffff0000, v2                            //
    v_writelane_b32 v1, s0, 48                                  //
    v_cmp_eq_u64_e64 s0, 0, v[4:5]                              //
    v_mov_b32_e32 v4, 0                                         //
    v_mov_b32_e32 v5, 0                                         //
    v_writelane_b32 v1, 0, 16                                   //
    v_permlanex16_b32 v6, v1, 0, -1 op_sel:[1,0]                //
    v_cndmask_b32_e64 v1, v6, v1, s0                            //
    v_cmp_ne_u64_e64 s0, 0, v[2:3]                              //
    v_add_nc_u32_dpp v6, v1, v1 row_shr:1 row_mask:0xf bank_mask:0
    v_mov_b32_dpp v4, v1 row_shr:2 row_mask:0xf bank_mask:0xf   //
    v_mov_b32_dpp v5, v1 row_shr:3 row_mask:0xf bank_mask:0xf   //
    v_add3_u32 v1, v6, v4, v5                                   //
    v_add_nc_u32_dpp v1, v1, v1 row_shr:4 row_mask:0xf bank_mask:0
    v_add_nc_u32_dpp v1, v1, v1 row_shr:8 row_mask:0xf bank_mask:0
    v_permlanex16_b32 v4, v1, -1, -1 op_sel:[1,0]               //
    v_cndmask_b32_e64 v2, 0, v4, s0                             //
    v_cmp_lt_u32_e64 s0, 31, v0                                 //
    v_add_nc_u32_e32 v1, v2, v1                                 //
    v_readlane_b32 s6, v1, 31                                   //
    v_cndmask_b32_e64 v0, 0, s6, s0                             //
    v_add_nc_u32_e32 v0, v1, v0                                 //
    s_mov_b64 exec, s[4:5]                                      //
    v_mov_b32_e32 v8, v0                                        //
```

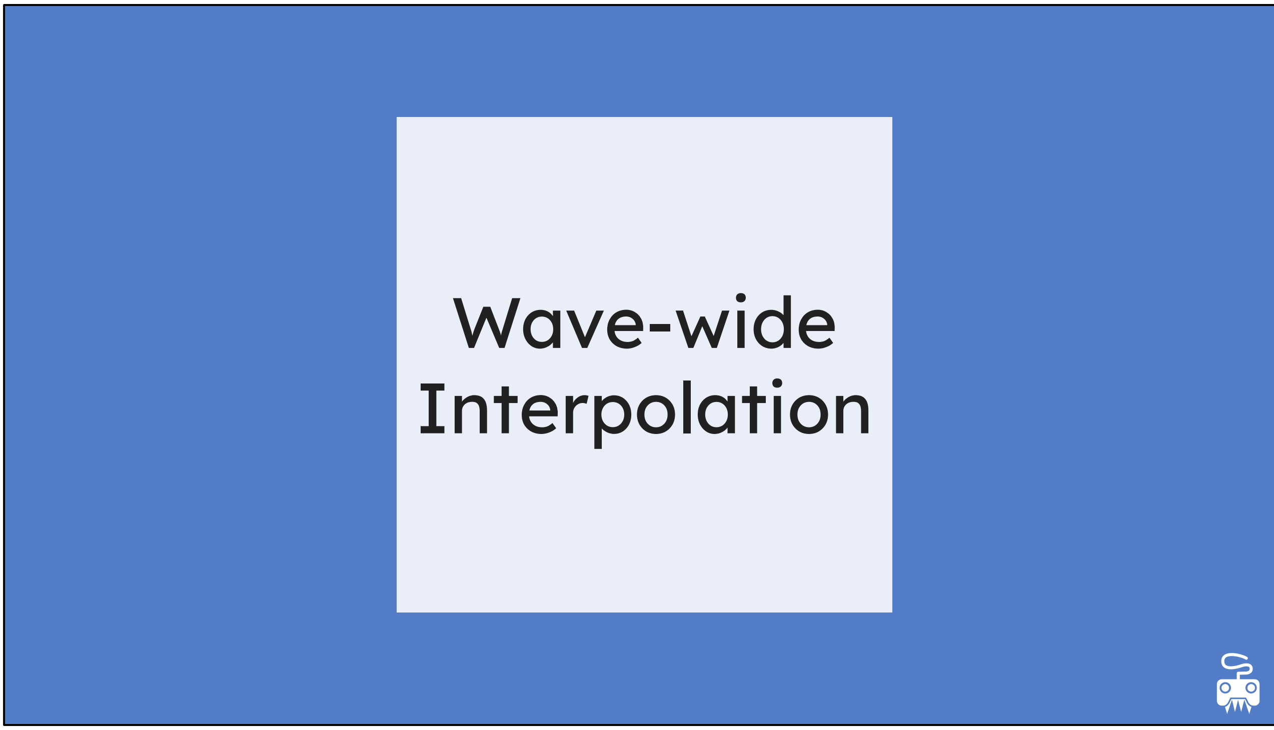
~40 Instructions

VS

```
;     uint outputIndex = WavePrefixCountBits(selectInput);
    v_mbcnt_lo_u32_b32 v1, vcc_lo, 0                            //
    v_mbcnt_hi_u32_b32 v1, vcc_hi, v1                           //
```

2 Instructions

I'll let you decide which one is likely to be faster…

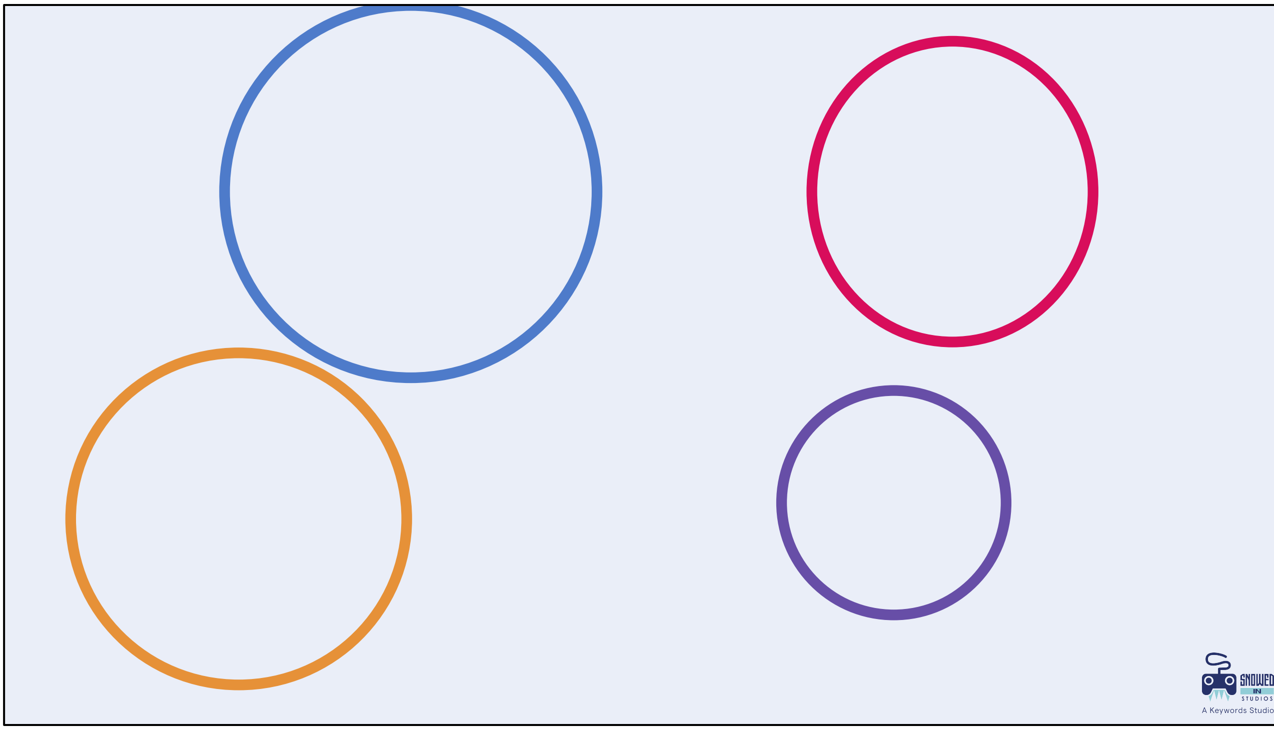


Now!

We’ve touched on some common use cases for wave intrinsics.

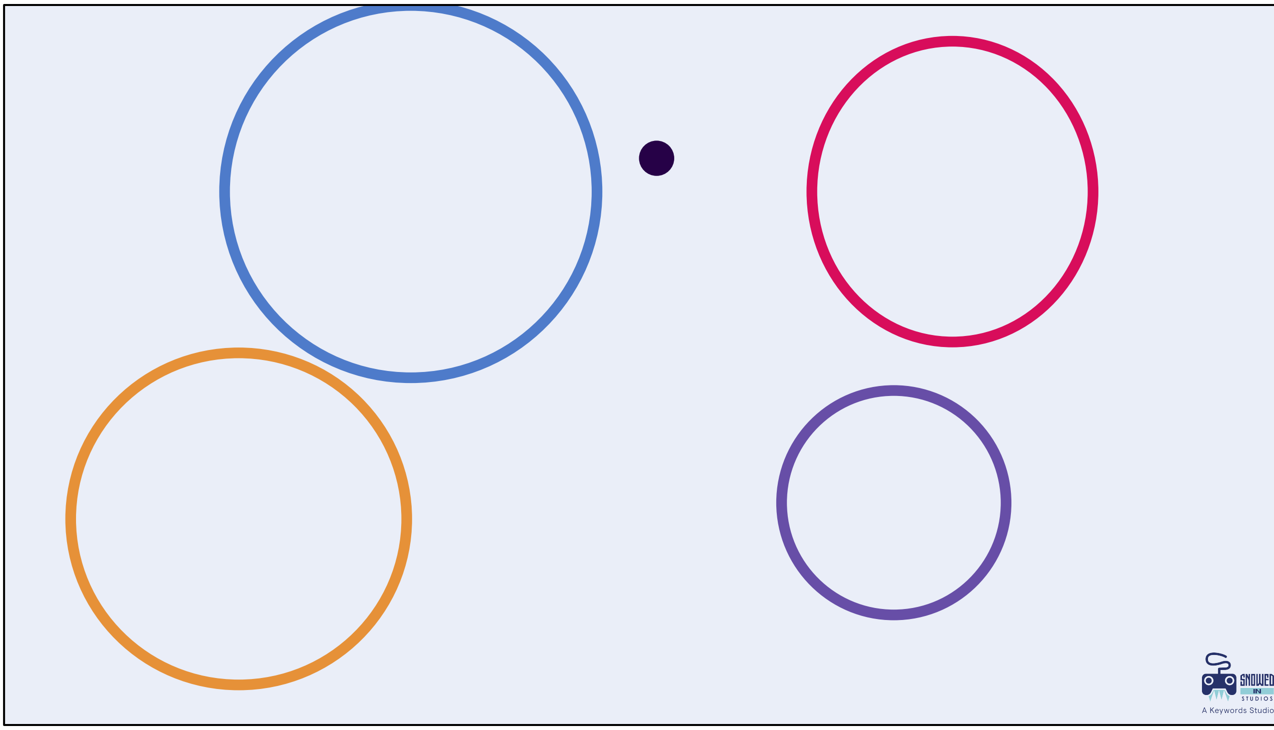
Let’s look at an algorithm we used on a project currently in production.

Let’s talk about wave-wide interpolation.



Imagine you have 4 spheres.

Each sphere has a radius and emits some sort of color.

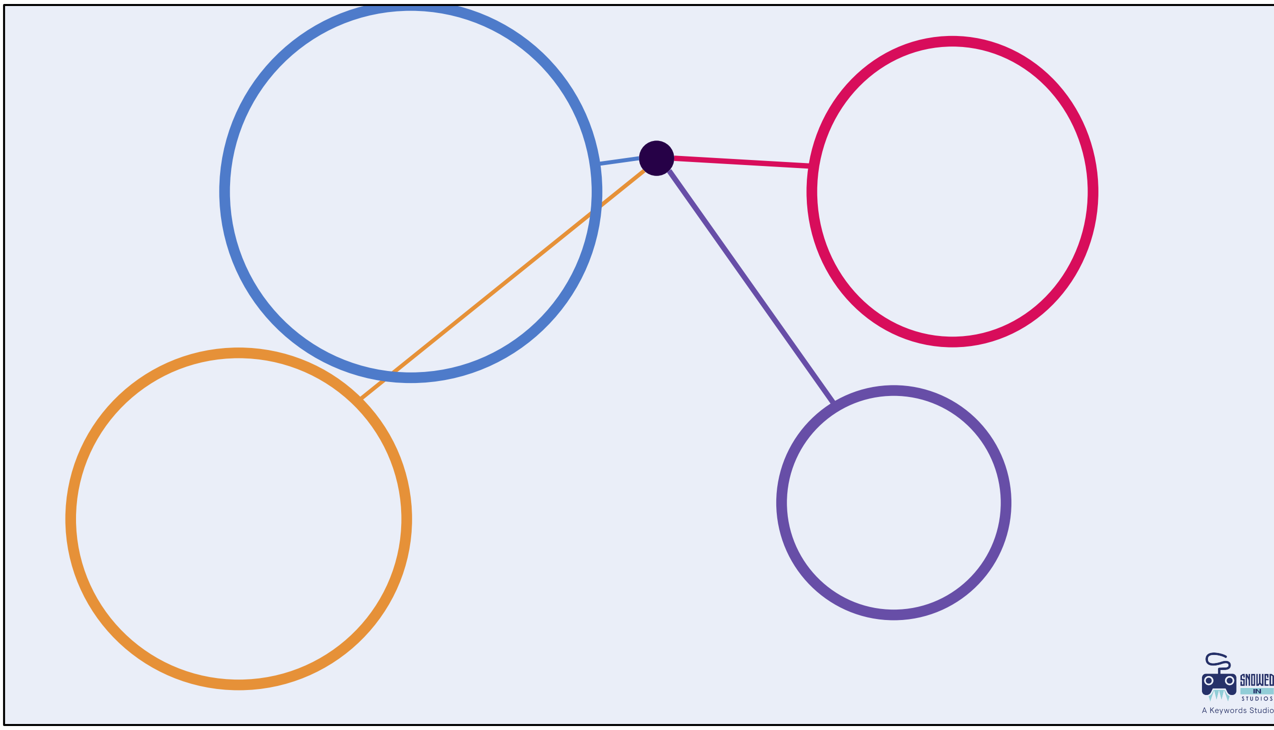


Now, let’s imagine that we have a point that wants to interpolate between the colors of each of these spheres.

click

For some reason, we don’t want a weighted average. We want a series of recursive linear interpolations based on some priority scheme.

This formulation of the problem is simple. We could do it on the CPU if we wanted to.

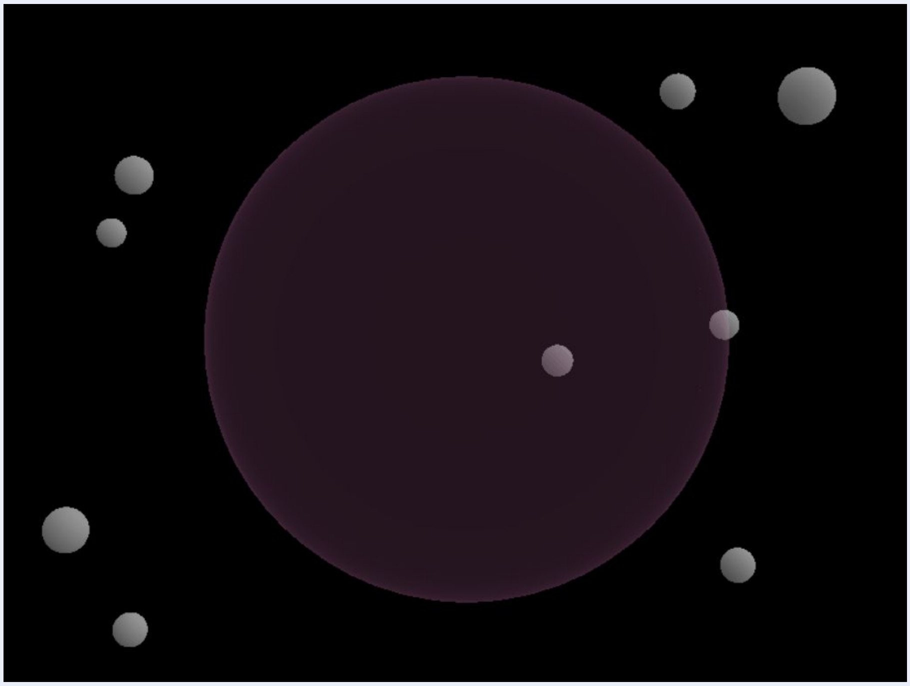


Now, let’s imagine that we have a point that wants to interpolate between the colors of each of these spheres.

click

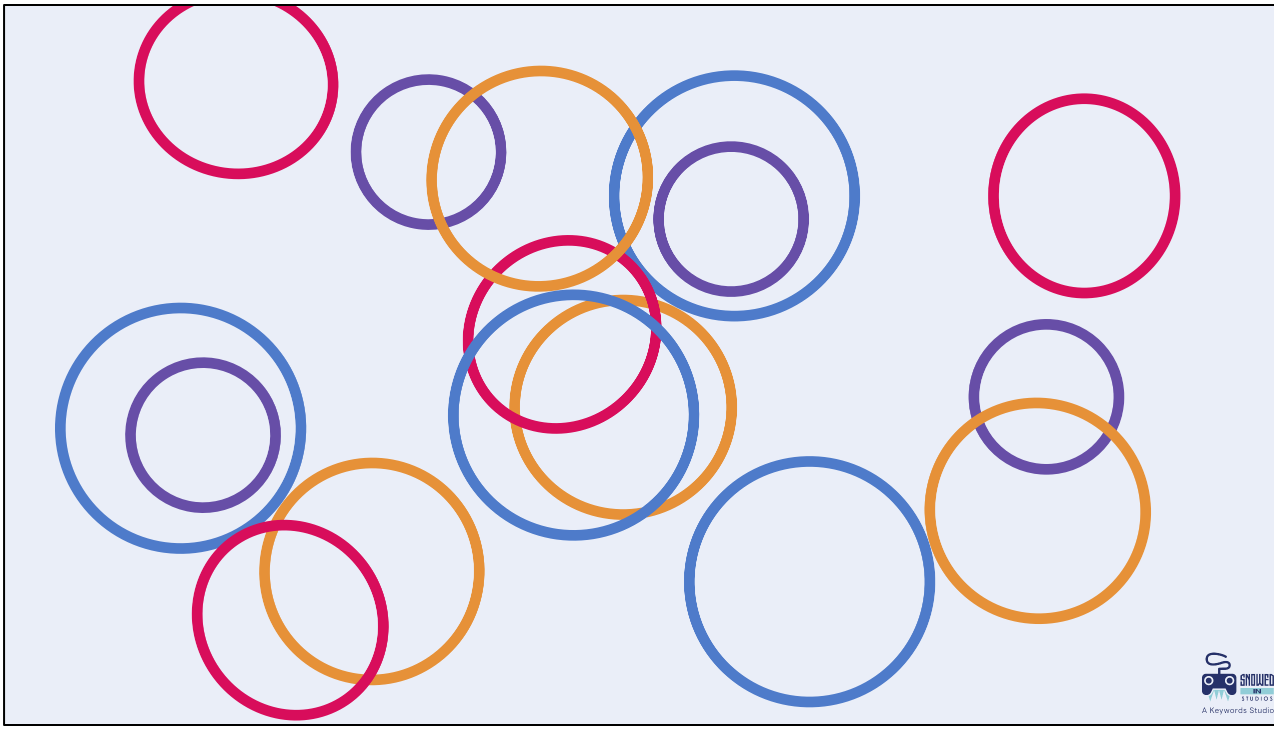
For some reason, we don’t want a weighted average. We want a series of recursive linear interpolations based on some priority scheme.

This formulation of the problem is simple. We could do it on the CPU if we wanted to.



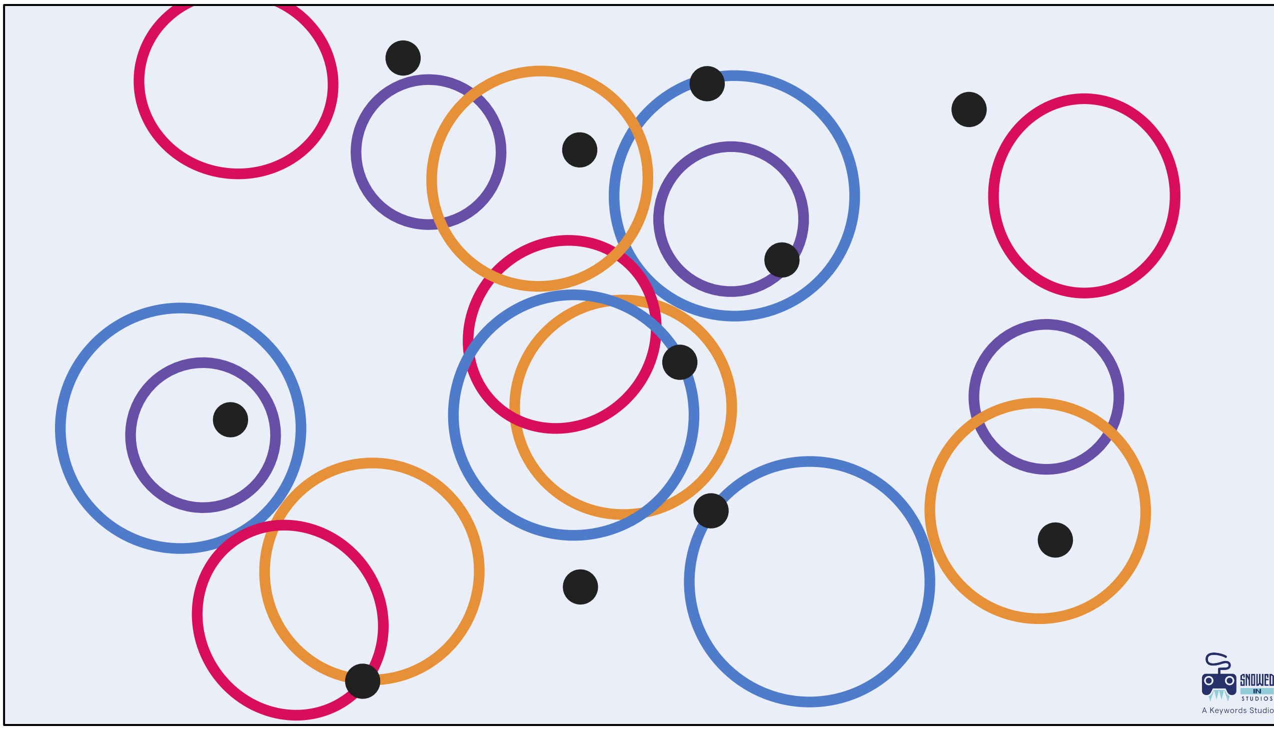
We want something that looks like this.

(Pause)



Instead, what if we have many more spheres.

Let’s say 128 of them.



And let’s say we have 128 points that each want to interpolate the colors of these 128 spheres.

# Points	# Spheres	Strategy	# Threads	Iterations / Thread	Approval
1 Million	4	Point Per Thread	1 Million	4	

An important detail in our process is for us to consider the amount of parallelism present in our problem.

This depends on the numbers of points and spheres.

If we have 1 million points and 4 spheres, then no problem!

click

We can just have one thread per point calculate its result with the 4 spheres.

# Points	# Spheres	Strategy	# Threads	Iterations / Thread	Approval
1 Million	4	Point Per Thread	1 Million	4	👍

An important detail in our process is for us to consider the amount of parallelism present in our problem.

This depends on the numbers of points and spheres.

If we have 1 million points and 4 spheres, then no problem!

click

We can just have one thread per point calculate its result with the 4 spheres.

# Points	# Spheres	Strategy	# Threads	Iterations / Thread	Approval
1 Million	4	Point Per Thread	1 Million	4	👍
128	128	Point Per Thread	128	128	?

But what if we only have 128 points and 128 spheres?

If we use the same strategy as above, we end up with 128 threads looping through 128 spheres.

Example RDNA GPU

- 26 Workgroup Processors (WGP)
- 4 SIMD32 per WGP
- 20 waves per SIMD32
- 26*4*20*32=**66,560** threads for 100% saturation

As you may have guessed, this isn't a very effective use of our GPU.

If we look at an example RDNA GPU with 26 Workgroup Processors, we can schedule up to 66 thousand threads.

128 threads is a mere 0.1% of our GPU's potential capacity.

# Points	# Spheres	Strategy	# Threads	Iterations / Thread	Approval
1 Million	4	Point Per Thread	1 Million	4	👍
128	128	Point Per Thread	128	128	?

click

# Points	# Spheres	Strategy	# Threads	Iterations / Thread	Approval
1 Million	4	Point Per Thread	1 Million	4	👍
128	128	Point Per Thread	128	128	👎

SNOWED IN STUDIOS
A Keywords Studio

click

# Points	# Spheres	Strategy	# Threads	Iterations / Thread	Approval
1 Million	4	Point Per Thread	1 Million	4	👍
128	128	Point Per Thread	128	128	👎
128	128	Point-Sphere Pair Per Thread?			

What if instead, we spawn a thread per point-sphere pair?

We would have each thread compute a single piece of our equation for each point and we would somehow combine the results.

And as a result, we be able to spawn *click* 16 thousand threads.

# Points	# Spheres	Strategy	# Threads	Iterations / Thread	Approval
1 Million	4	Point Per Thread	1 Million	4	👍
128	128	Point Per Thread	128	128	👎
128	128	Point-Sphere Pair Per Thread?	128*128=1638 4	1	👍

What if instead, we spawn a thread per point-sphere pair?

We would have each thread compute a single piece of our equation for each point and we would somehow combine the results.

And as a result, we be able to spawn *click* 16 thousand threads.

How?

click

Version #1

```
for(int p = 0; p < pointCount; p++)
{
    float3 pointPos = PointPositions[p];

    float3 color = float3(0.0f, 0.0f, 0.0f);
    for(uint i = 0; i < sphereCount; i++)
    {
        float influence = calcInfluence(spherePositions[i], pointPos);
        color = lerp(color, sphereColors[i], influence);
    }

    PointColors[p] = color;
}
```

If we were to write our problem as a serial program, it might look something like this.

Version #1

Per-Thread

```
for(int p = 0; p < pointCount; p++)
{
    float3 pointPos = PointPositions[p];

    float3 color = float3(0.0f, 0.0f, 0.0f);
    for(uint i = 0; i < sphereCount; i++)
    {
        float influence = calcInfluence(spherePositions[i], pointPos);
        color = lerp(color, sphereColors[i], influence);
    }

    PointColors[p] = color;
}
```

In our base version, we simply have one thread per-point.

This works well since we don’t have any dependencies between any loop iteration which means each iteration can run completely independently.

Version #2

Per-Thread **Group**

Per-Thread

```
for(int p = 0; p < pointCount; p++)
{
    float3 pointPos = PointPositions[p];

    float3 color = float3(0.0f, 0.0f, 0.0f);
    for(uint i = 0; i < sphereCount; i++)
    {
        float influence = calcInfluence(spherePositions[i], pointPos);
        color = lerp(color, sphereColors[i], influence);
    }

    PointColors[p] = color;
}
```

Instead, what if we spawn a thread group for each point and spawn a thread per sphere?

We're unrolling our loop where each individual iteration is its own thread.

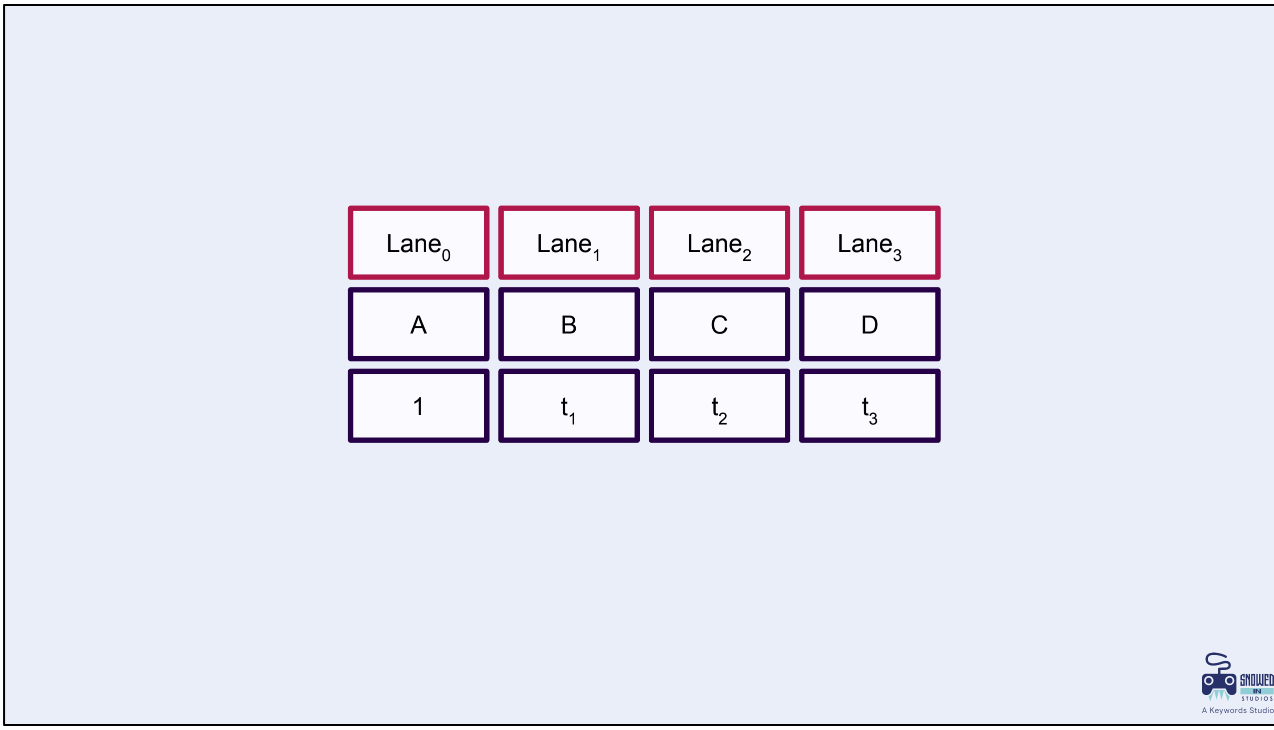
Version #2

```
for(int p = 0; p < pointCount; p++)
{
    float3 pointPos = PointPositions[p];

    float3 color = float3(0.0f, 0.0f, 0.0f);
    for(uint i = 0; i < sphereCount; i++)
    {
        float influence = calcInfluence(spherePositions[i], pointPos);
        color = lerp(color, sphereColors[i], influence);
    }

    PointColors[p] = color;
}
```

Cross-Thread Dependency

However, our lerp has a dependency between each loop iteration (aka thread).

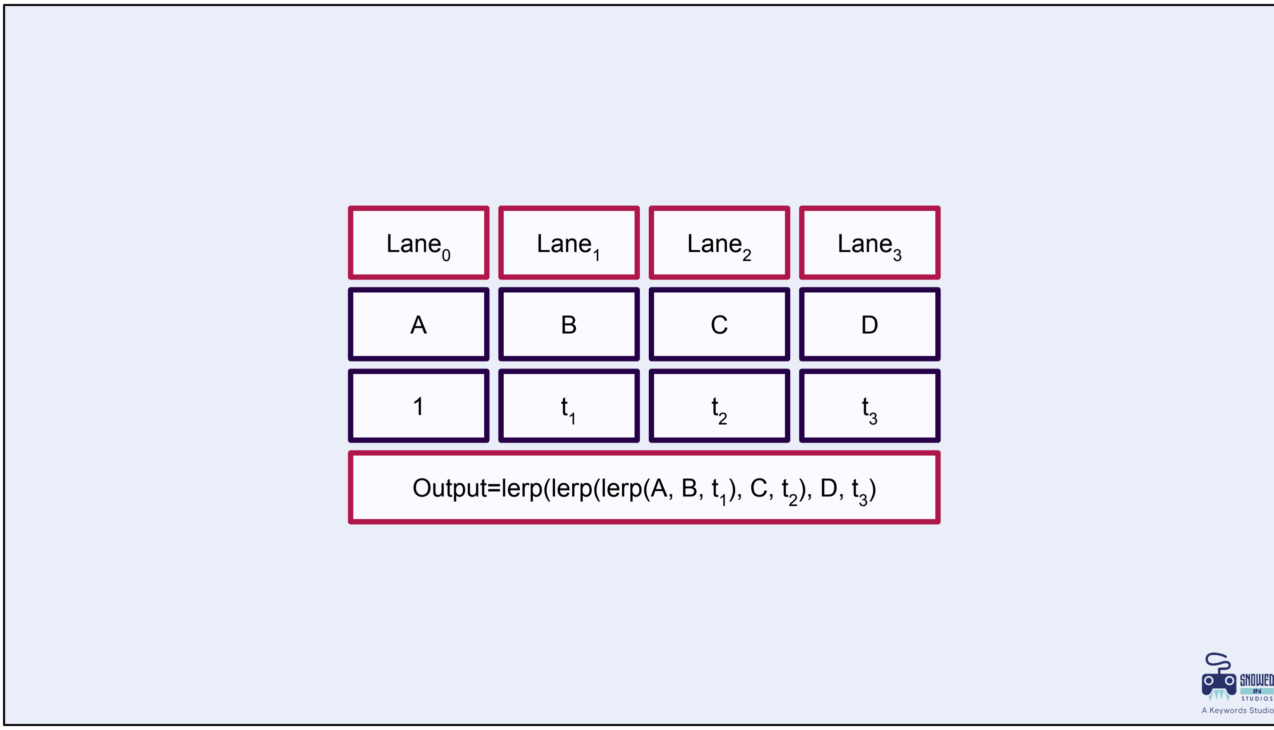
How do we address that?



If we look at our small 4 lane GPU again.

This set of lanes is looking to compute the result of the interpolation of A, B, C and D using 4 interpolation terms.

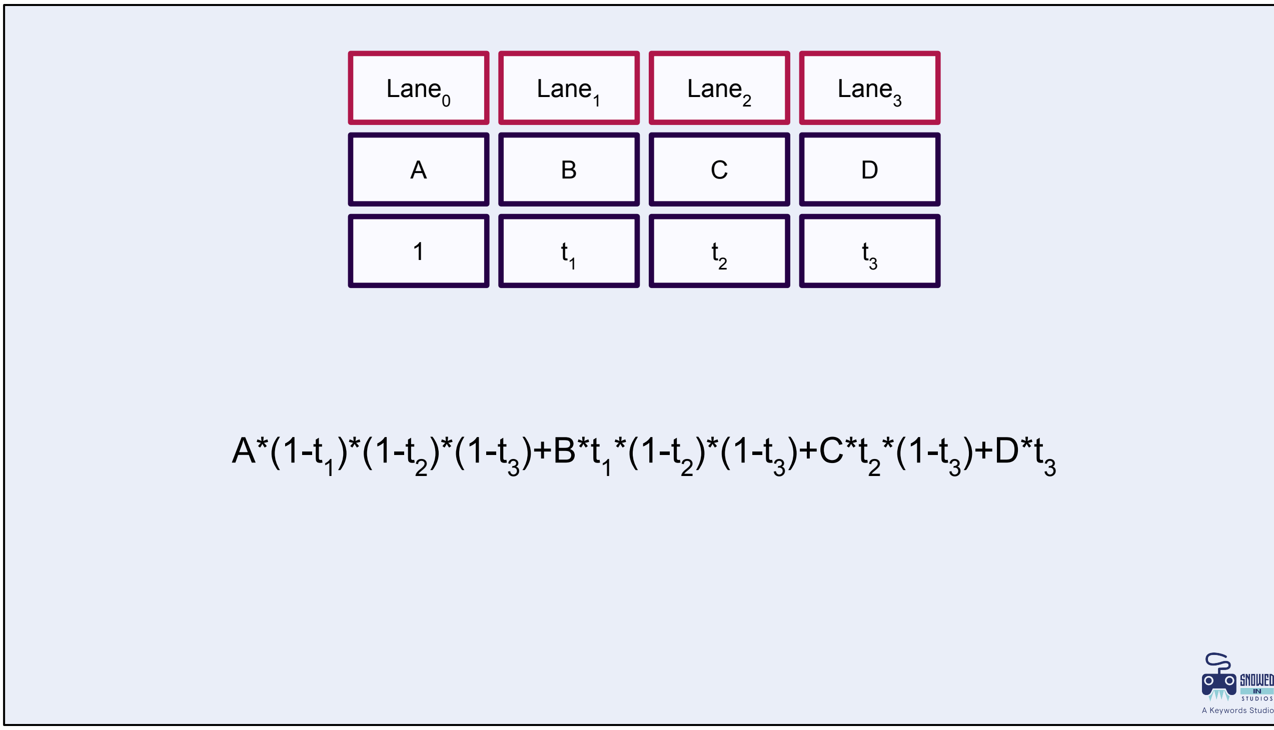
For convenience and so the algebra doesn't get too crazy, we've set Lane 0's interpolation term to 1.



We want to combine the results of our lanes such that we evaluate the expression above.

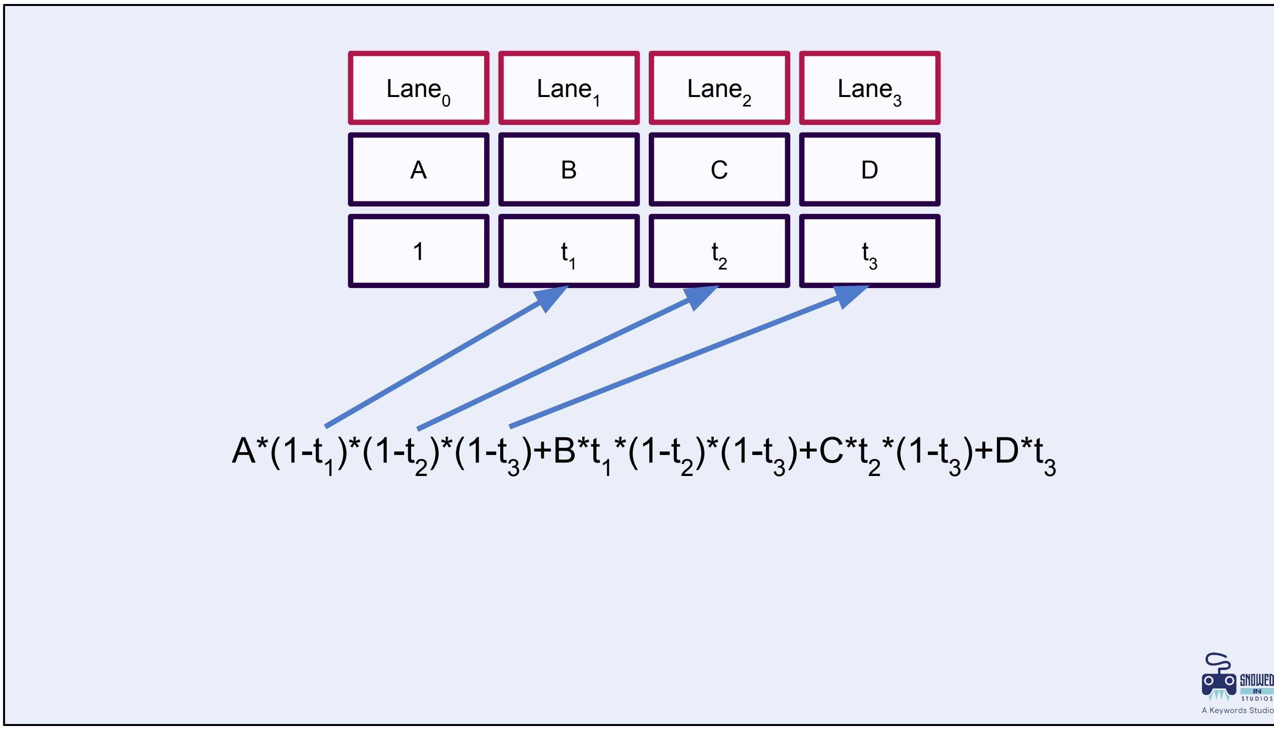
lerp(A, B, t_0)	$A*(1-t_0)+B*t_0$
lerp(lerp(A, B, t_1), C, t_2)	$A*(1-t_1)*(1-t_2)+B*t_1*(1-t_2)+C*t_2$
lerp(lerp(lerp(A, B, t_0), C, t_1), D, t_2)	$A*(1-t_1)*(1-t_2)*(1-t_3)+B*t_1*(1-t_2)*(1-t_3)+C*t_2*(1-t_3)+D*t_3$

If we expand each iteration of a recursive lerp, you might notice a pattern.



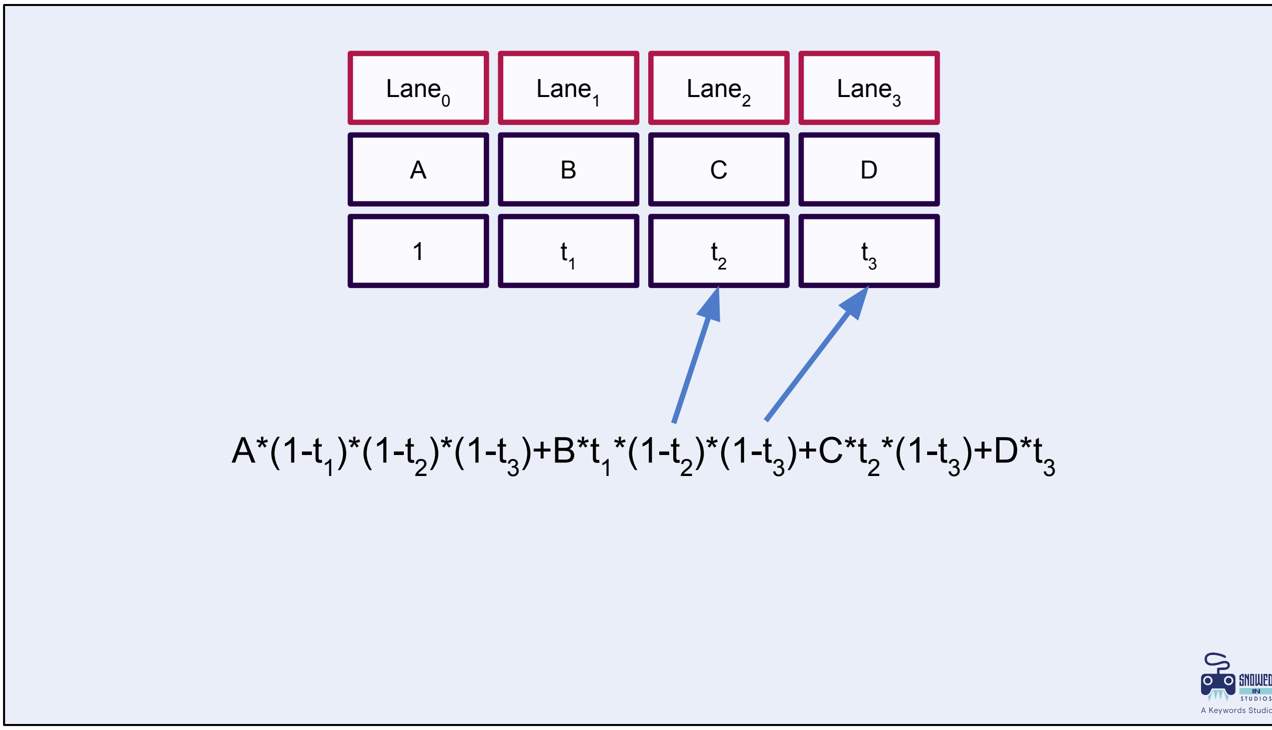
Each term is simply a combination of its interpolation factor and its next neighbour’s 1 minus interpolation term.

click
click
click



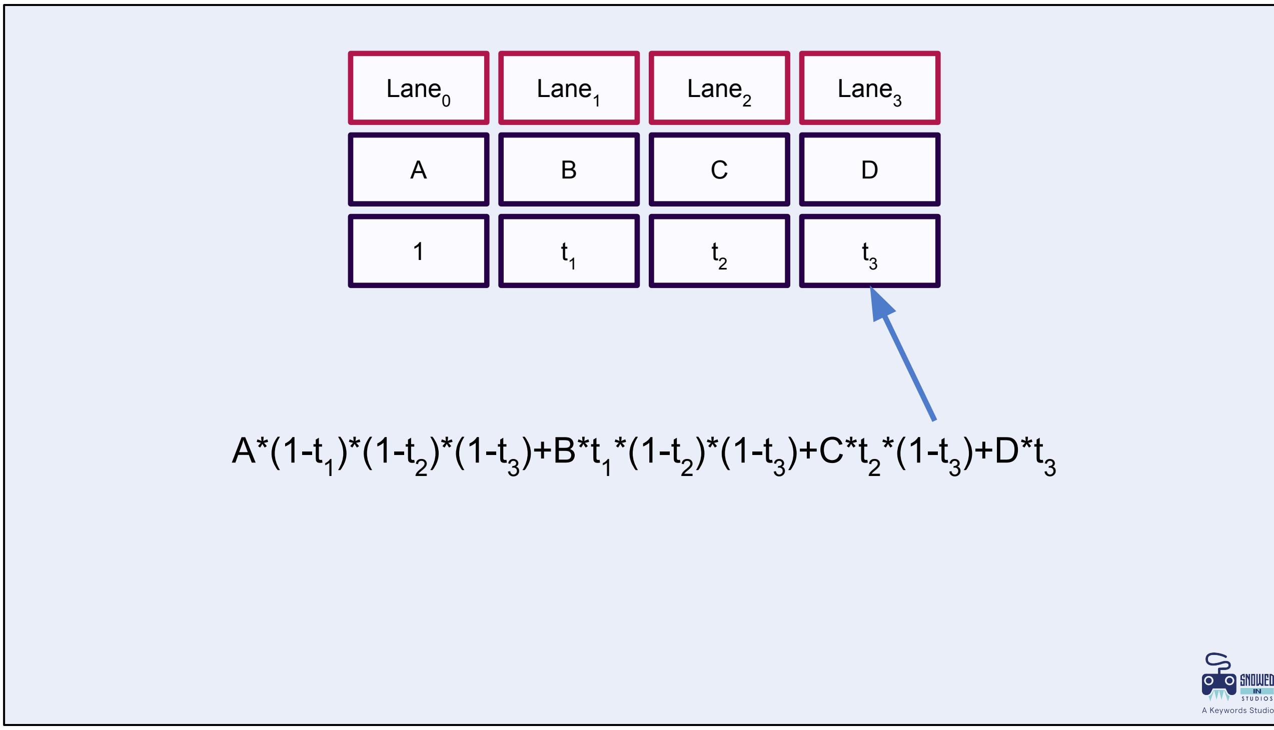
Each term is simply a combination of its interpolation factor and its next neighbour’s 1 minus interpolation term.

click
click
click



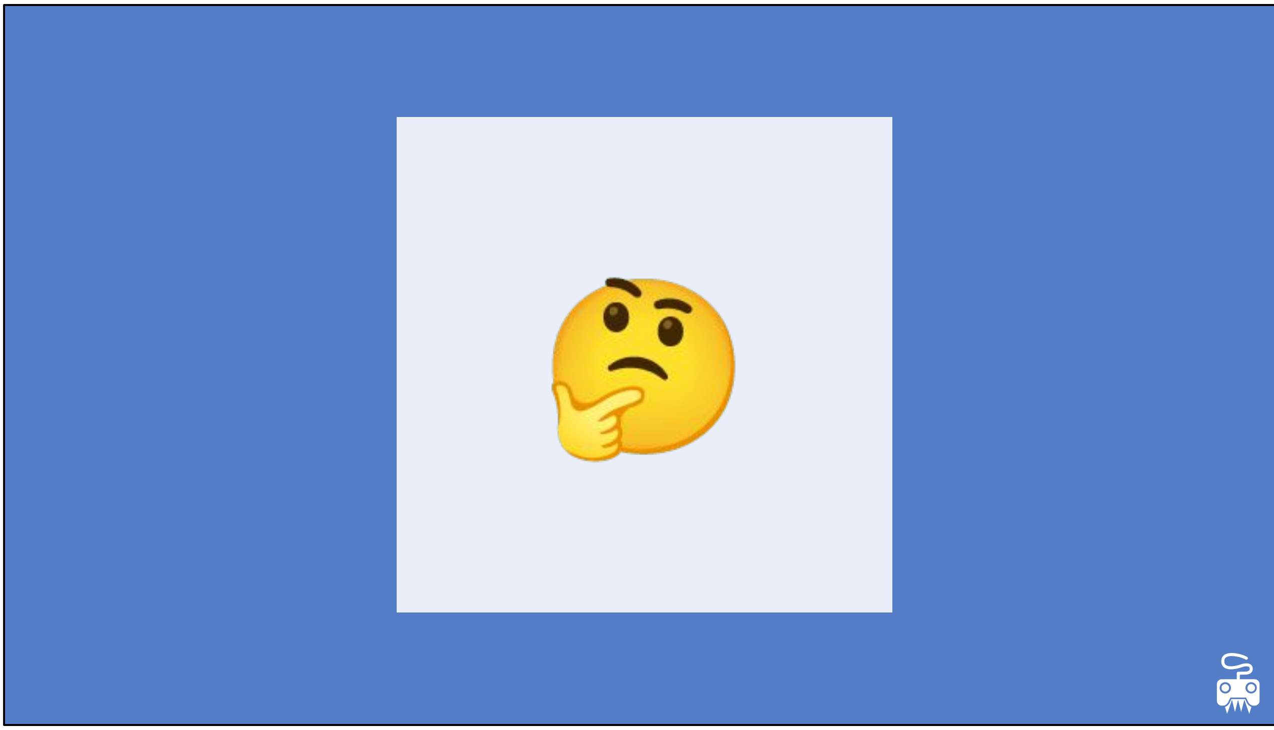
Each term is simply a combination of its interpolation factor and its next neighbour's 1 minus interpolation term.

click
click
click



Each term is simply a combination of its interpolation factor and its next neighbour's 1 minus interpolation term.

click
click
click



Which looks a lot like a prefix operation in the reverse order.

Lane$_0$	Lane$_1$	Lane$_2$	Lane$_3$
A	B	C	D
1	t_1	t_2	t_3

$$A*(1-t_1)*(1-t_2)*(1-t_3)+B*t_1*(1-t_2)*(1-t_3)+C*t_2*(1-t_3)+D*t_3$$

So perhaps we can simply reverse the order of our lane's values.

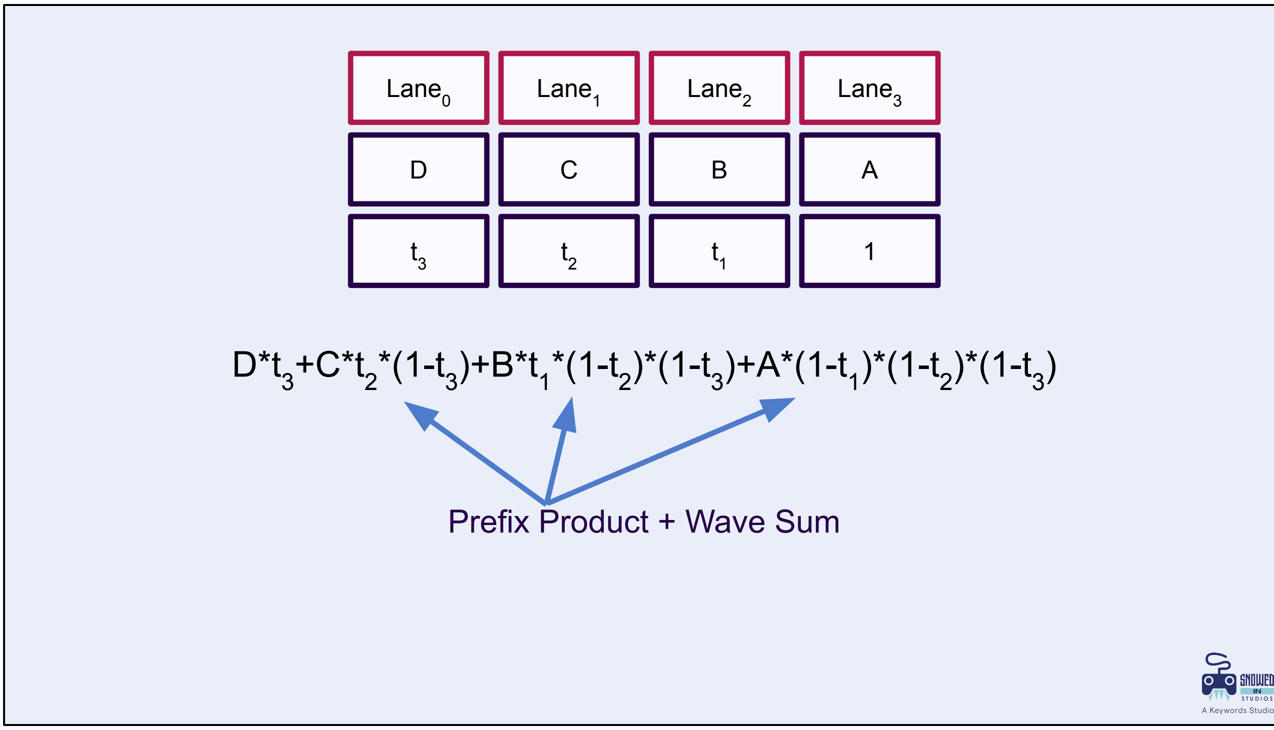
click

Lane$_0$	Lane$_1$	Lane$_2$	Lane$_3$
D	C	B	A
t_3	t_2	t_1	1

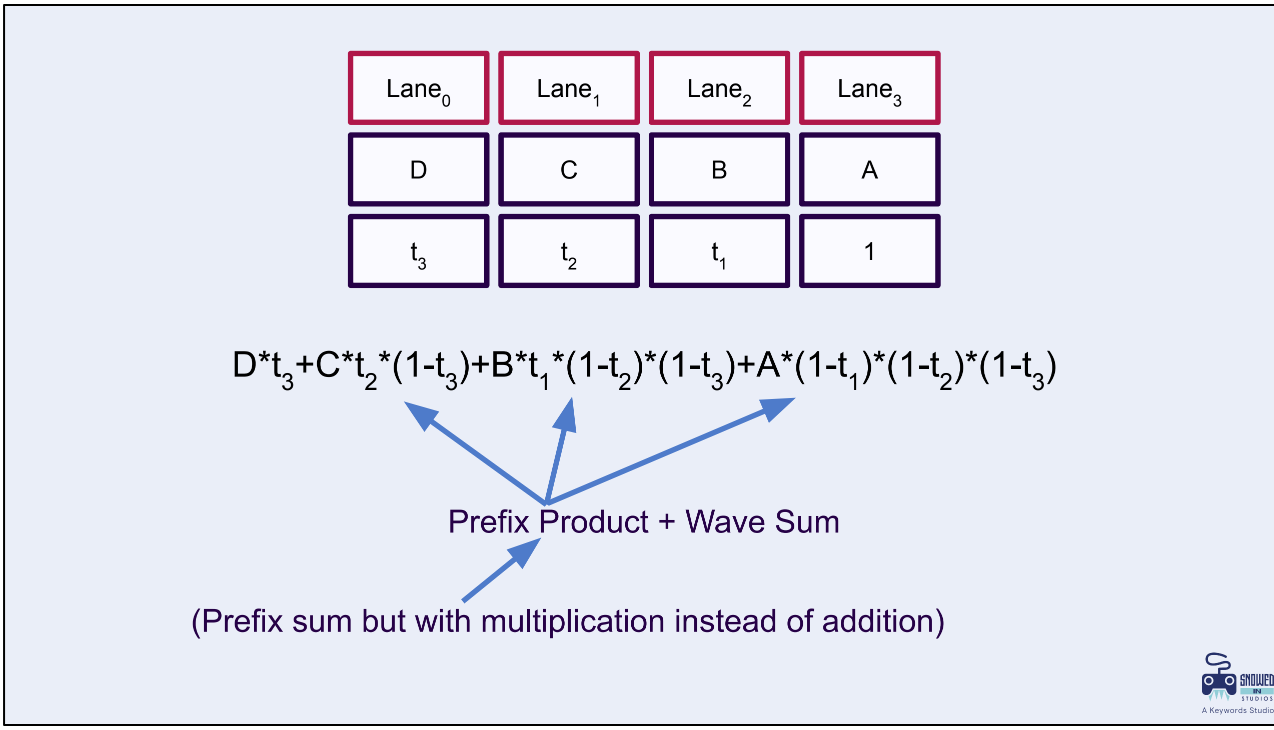
$$D*t_3+C*t_2*(1-t_3)+B*t_1*(1-t_2)*(1-t_3)+A*(1-t_1)*(1-t_2)*(1-t_3)$$

So perhaps we can simply reverse the order of our lane's values.

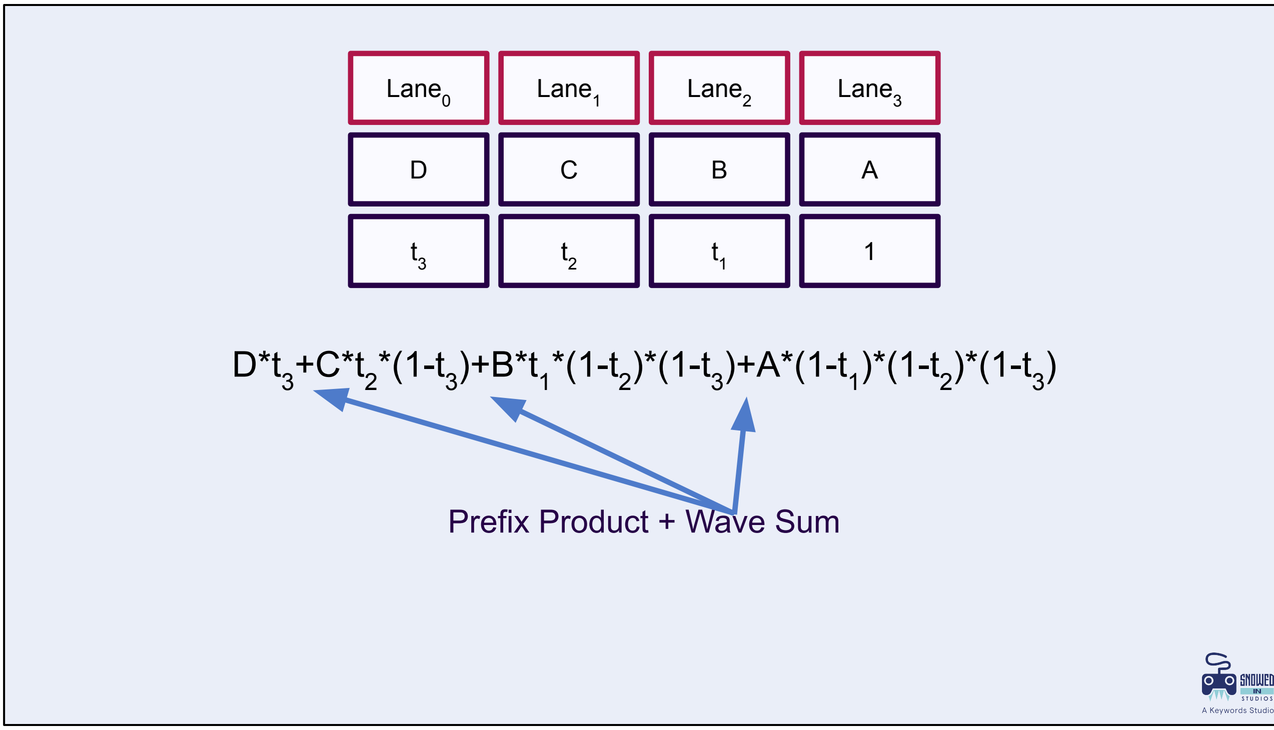
click



Then we can easily apply a prefix product.


$Lane_0$
$Lane_1$
$Lane_2$
$Lane_3$
D
C
B
A
t_3
t_2
t_1
1
$D*t_3+C*t_2*(1-t_3)+B*t_1*(1-t_2)*(1-t_3)+A*(1-t_1)*(1-t_2)*(1-t_3)$
Prefix Product + Wave Sum
(Prefix sum but with multiplication instead of addition)
A Keywords Studio

click



And once we’ve applied our prefix product, we simply want to combine our results using a wave sum!

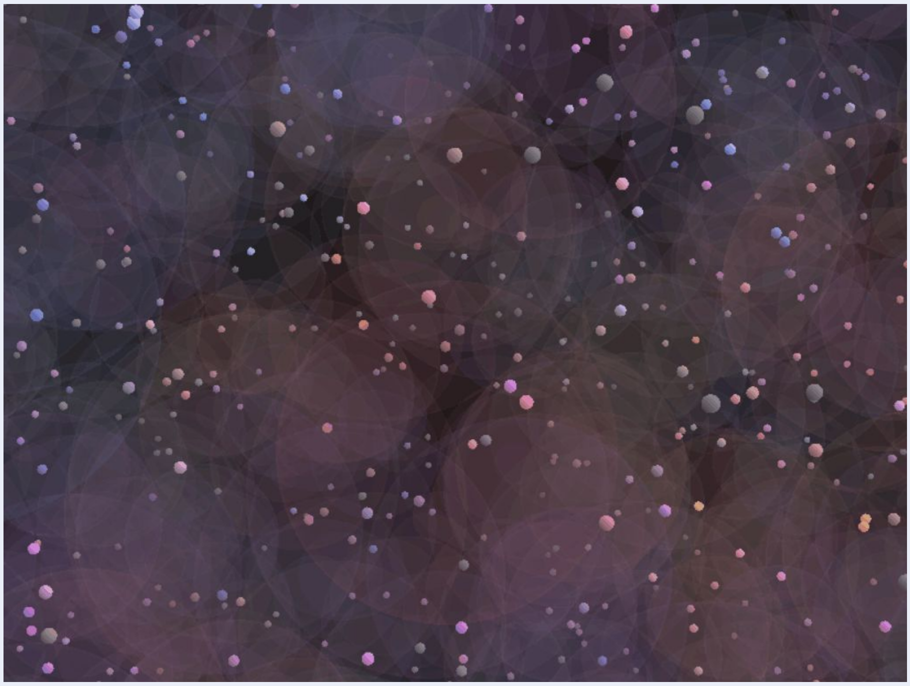
$$D*t_3+C*t_2*(1-t_3)+B*t_1*(1-t_2)*(1-t_3)+A*(1-t_1)*(1-t_2)*(1-t_3)$$

```
float WaveActiveLerp(float value, float t)
{
    float prefixProduct = WavePrefixProduct(1.0f - t);
    float laneValue = value * t * prefixProduct;
    float interpolation = WaveActiveSum(laneValue);

    return interpolation;
}
```

(Pause)

Through some simple algebra, we were able to identify a pattern that we could use to turn what might be a problem ill-suited for a GPU to one that can effectively saturate our GPU’s cores.

From 128 threads to 16 thousand threads through the use of wave intrinsics.



Here we’ve simulated a thousand spheres influencing a thousand points at twice the speed of the naive implementation.

1024 Spheres x 1024 Points			
	Naive	**Parallelized**	**Savings**
Intel(R) Iris(R) Xe Graphics	0.6ms	0.3ms	0.3ms
NVIDIA GeForce RTX 3070 Ti	0.4ms	0.1ms	0.3ms

As you can see, the savings can be drastic between 2x and 4x in savings on various GPUs.

A word of warning before you go forth.

Code Generation - Dynamic WaveReadLaneAt

```
;     uint outputIndex = WaveReadLaneAt(selectInput, WaveGetLaneIndex() + 1);
    v_cndmask_b32_e64 v3, 0, 1, vcc_lo                           // 000000000048:
0000000000000050 <L0>:
    v_readfirstlane_b32 s6, v2                                   // 000000000050:
    v_cmp_eq_u32_e64 s4, s6, v2                                  // 000000000054:
    s_and_saveexec_b64 s[4:5], s[4:5]                            // 00000000005C:
    v_readlane_b32 s6, v3, s6                                    // 000000000060:
    v_mov_b32_e32 v1, s6                                         // 000000000068:
    s_xor_b64 exec, exec, s[4:5]                                 // 00000000006C:
    s_cbranch_execnz L0                                          // 000000000070:
    s_mov_b64 exec, s[2:3]                                       // 000000000074:
; C:\\Users\\Alex\\Desktop\\Development\\Icecap\\Icecap\\bin\\Debug\\net8.0-wir
```

Waterfall loop! :(

Be mindful of your code generation. Using WaveReadLaneAt to do a shuffle boils down to a waterfall loop on AMD’s offline compiler.

click

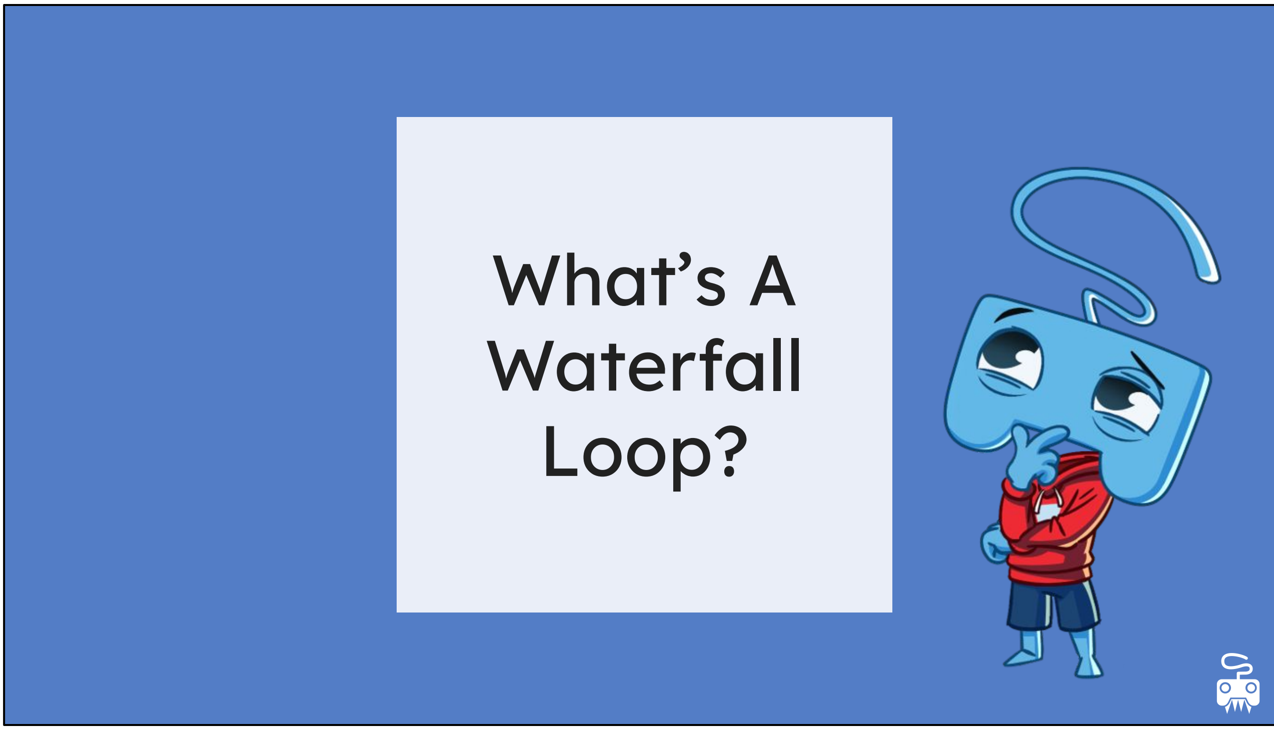
Code Generation - Spir-V Shuffle Up?

```
;     uint outputIndex = OpGroupNonUniformShuffleUp(vk::SubgroupScope, (uint)selectInput, 1);
    v_cndmask_b32_e64 v2, 0, 1, vcc_lo                     // 000000000048: D5010002 01A902
0000000000000050 <L0>:
    v_readfirstlane_b32 s6, v3                             // 000000000050: 7E0C0503
    v_cmp_eq_u32_e64 s4, s6, v3                            // 000000000054: D4C20004 000206
    s_and_saveexec_b64 s[4:5], s[4:5]                      // 00000000005C: BE842404
    v_readlane_b32 s6, v2, s6                              // 000000000060: D7600006 00000D
    v_mov_b32_e32 v1, s6                                   // 000000000068: 7E020206
    s_xor_b64 exec, exec, s[4:5]                           // 00000000006C: 89FE047E
    s_cbranch_execnz L0                                    // 000000000070: BF89FFF7
    s_mov_b64 exec, s[2:3]                                 // 000000000074: BEFE0402
```

Waterfall loop! :(

And even if you use the more constrained shuffle up spir-v instruction.

It still boils down to a waterfall loop.



But what's a waterfall loop?

```
for (;;)
{
    uint scalarValue = WaveReadLaneFirst(laneValue);
    [branch]
    if (scalarValue == laneValue)
    {
        // Do something.
        break;
    }
}
```

A waterfall loop iterates through each unique value within our wave.

If we have a wave with unique values A, B and D we will iterate 3 times.

Each lane that matches our first active lane will enter the branch.

Then they will mark themselves inactive.

Then each lane that matches the new first active lane will enter the branch.

Until every unique value was processed.

Waterfall Loop

- Iterates for each unique value in a wave
- Worst case: wave lane count number of iterations
- Best case: single iteration
- Useful if you want to reduce vector register pressure

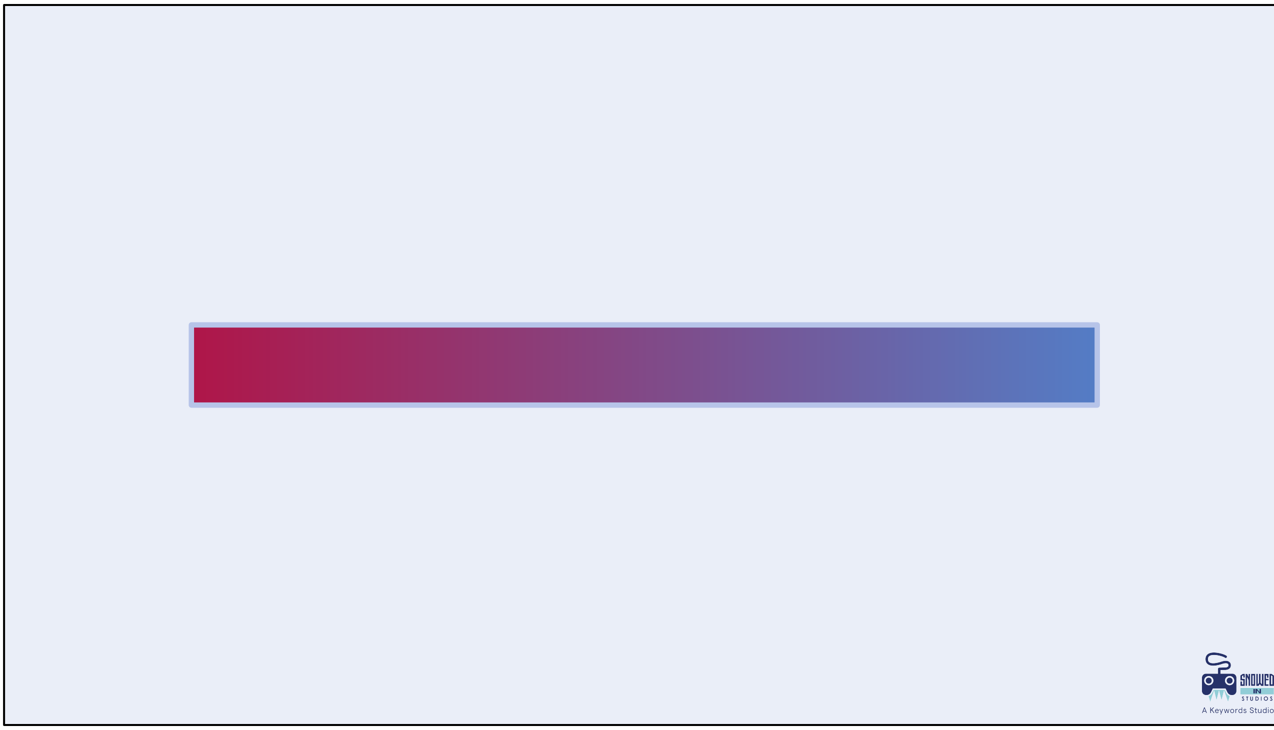
```
for (;;)
{
    uint scalarValue = WaveReadLaneFirst(laneValue);
    [branch]
    if (scalarValue == laneValue)
    {
        // Do something.
        break;
    }
}
```

In the best case, we iterate only once.

In the worst case when each lane in a wave has a different value, we iterate once for each lane.

Conclusions

click



A lens you can use to navigate your problem spaces, is to think of your GPU as a super powered loop unrolling machine.

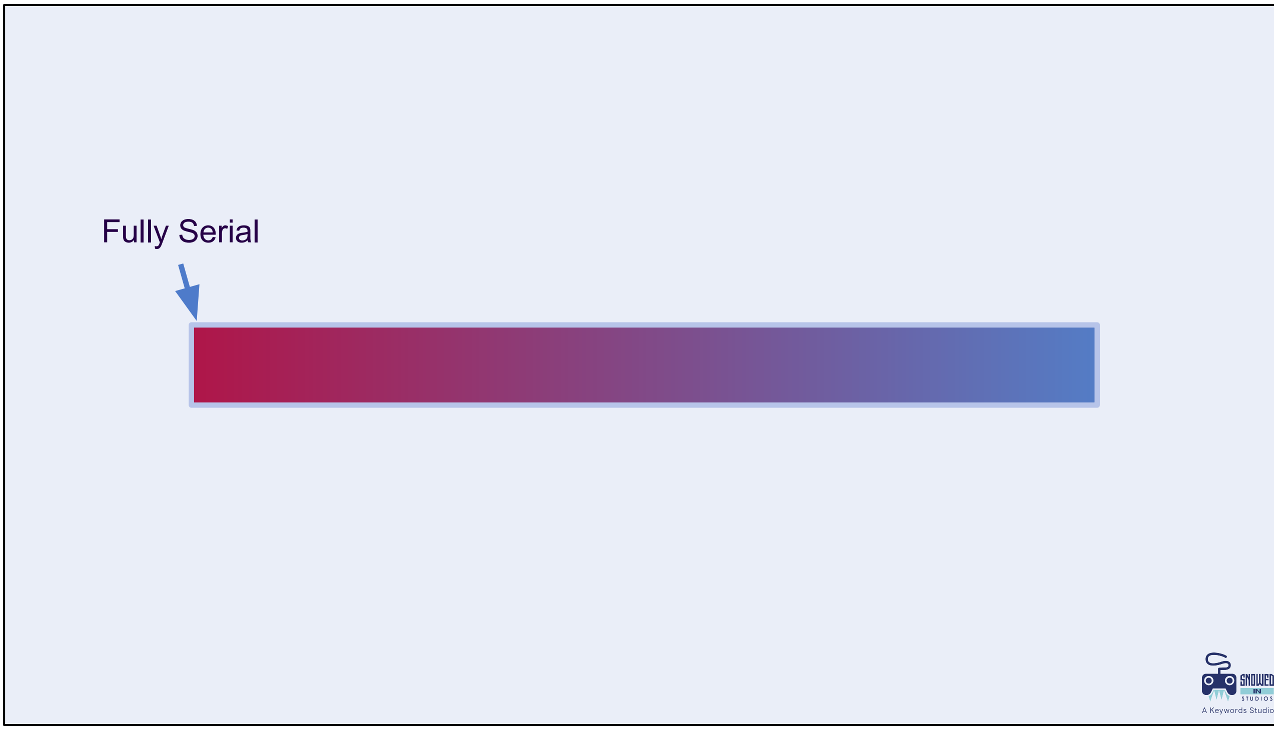
With that in mind, you can imagine that there’s a gradient spanning a fully serial program

click

to a fully unrolled program where each iteration of the loop is a thread.

click

With a vast number of options in between.



A lens you can use to navigate your problem spaces, is to think of your GPU as a super powered loop unrolling machine.

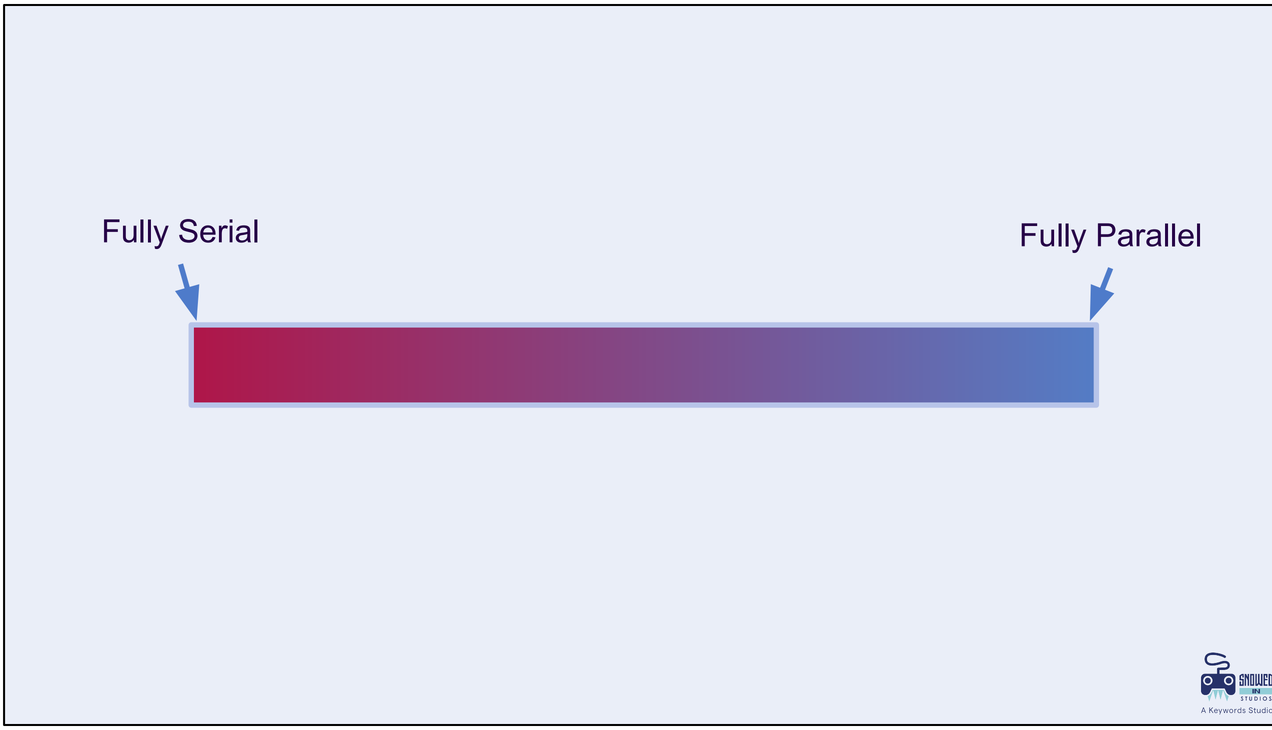
With that in mind, you can imagine that there's a gradient spanning a fully serial program

click

to a fully unrolled program where each iteration of the loop is a thread.

click

With a vast number of options in between.



A lens you can use to navigate your problem spaces, is to think of your GPU as a super powered loop unrolling machine.

With that in mind, you can imagine that there's a gradient spanning a fully serial program

click

to a fully unrolled program where each iteration of the loop is a thread.

click

With a vast number of options in between.

How To Develop Your Own?

- **Take inspiration from CPU SIMD and integer bit tricks**
  - Hacker's Delight - Henry S. Warren, Jr
  - Bit Twiddling Hacks - Sean Eron Anderson
  - SIMD at Insomniac Games - Andreas Fredriksson
- **Look at our friends in GPGPU**
- **Play around with the algebra**
  - Maybe there's a prefix or suffix operation hiding in there!
- **Explore the literature**
  - Goes back to the 70s!
    - A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations - Peter M. Kogge, Harold S. Stone

There's a wealth of inspiration that you can gain from many areas to develop your own algorithms.

Here are some spaces you can explore to get some ideas!

Where Can You Find Me?

Mastodon

https://mastodon.gamedev.place/@AlexSneezeKing

Blog

https://github.com/AlexSabourinDev/cranberry_blog

If you want to chat, you can find me on Mastodon.

And if you're interested in more topics like this, take a look at my blog!

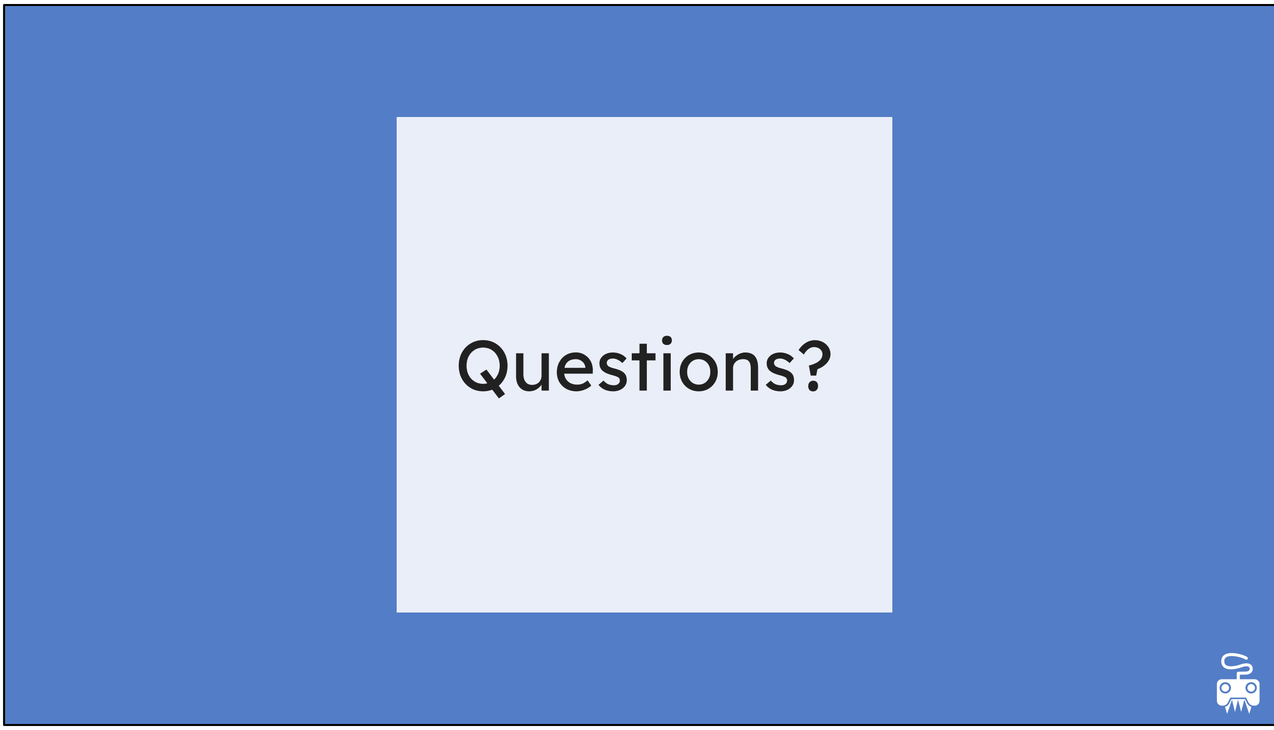
References

- [1] SIMD at Insomniac Games - Andreas Fredriksson
- [2] Wave Programming in D3D12 and Vulkan - David Lively, Holger Gruen
- [3] Stream compaction using wave intrinsics – Interplay of Light - Kostas Anagnostou
- [4] Compute shader wave intrinsics tricks - Marko Sreckovic
- [5] GPU Programming Primitives for Computer Graphics - Daniel Meister, Atsushi Yoshimura, Chih-Chen Kao
- [6] Optimizing for the Radeon RDNA architecture - Lou Kramer
- [7] Optimizing The Graphics Pipeline with Compute - Graham Wihlidal
- [8] Stream Reduction Operations for GPGPU Applications - Daniel Horn
- [9] A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations - Peter M. Kogge, Harold S. Stone
- [10] Bit Twiddling Hacks - Sean Eron Anderson
- [11] The Power Of Parallel Prefix - Clyde P. Kruskal, Larry Rudolph, Marc Snir
- [12] SIMD and SWAR Techniques
- [13] Prefix Sums and Their Applications - Guy E. Blelloch
- [14] AMD GCN Assembly: Cross-Lane Operations - Ben Sander
- [15] Reading Between The Threads: Shader Intrinsics
- [16] Adaptive Exposure from Luminance Histograms - Alex Tardif
- [17] https://bruop.github.io/exposure/ - Bruno Opsenica

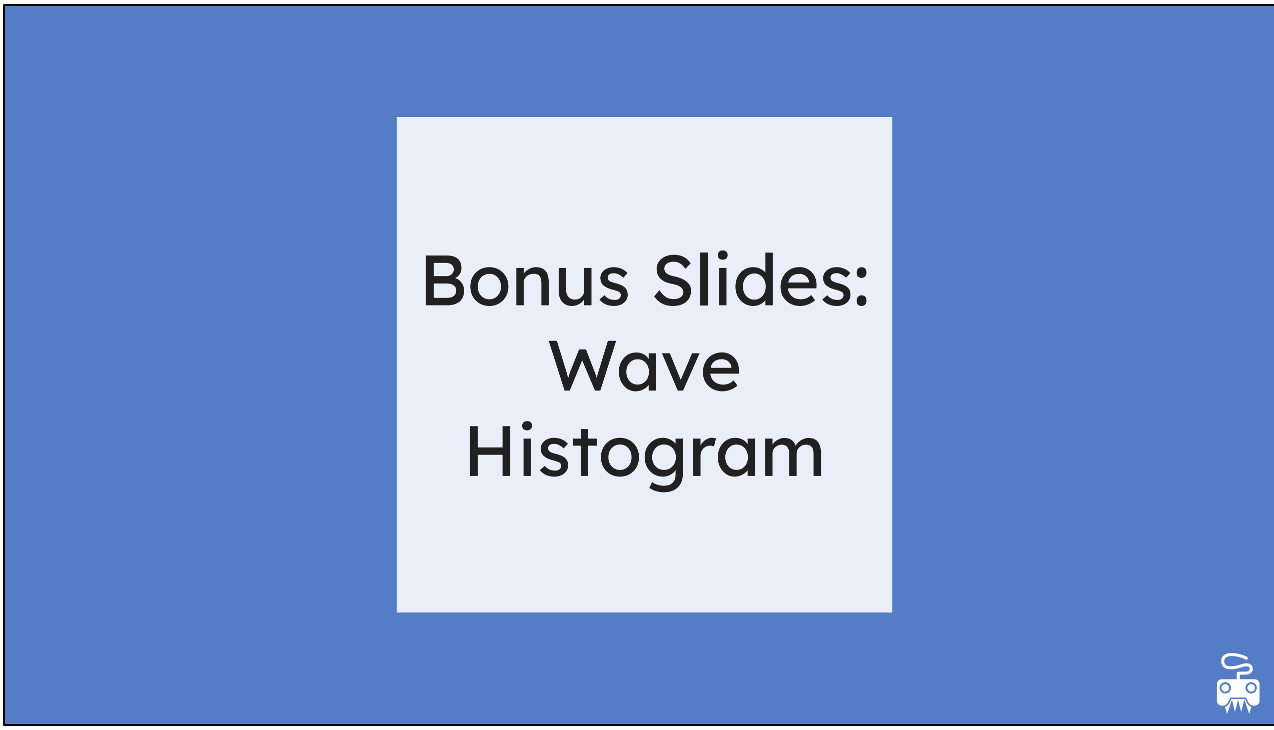
I recommend you take the time to explore some of these references. There's great insights in all of these.

Thank you all for your time!

If you have feedback for me on my presentation, I would love to hear from you later.



Any questions?



Let’s explore at a problem at a different scale.

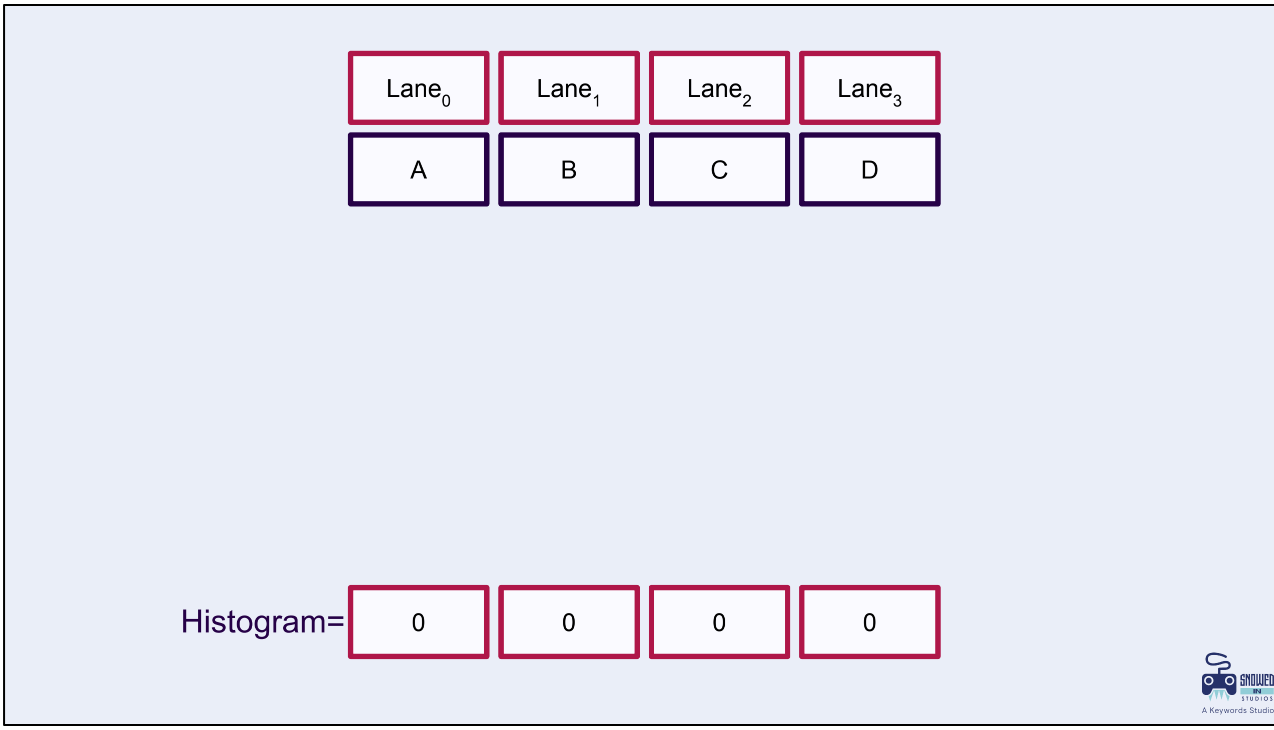
No more spheres. Instead, I want boxes.

Perhaps you’re working on histogram-based adaptive exposure or perhaps you just like counting things.

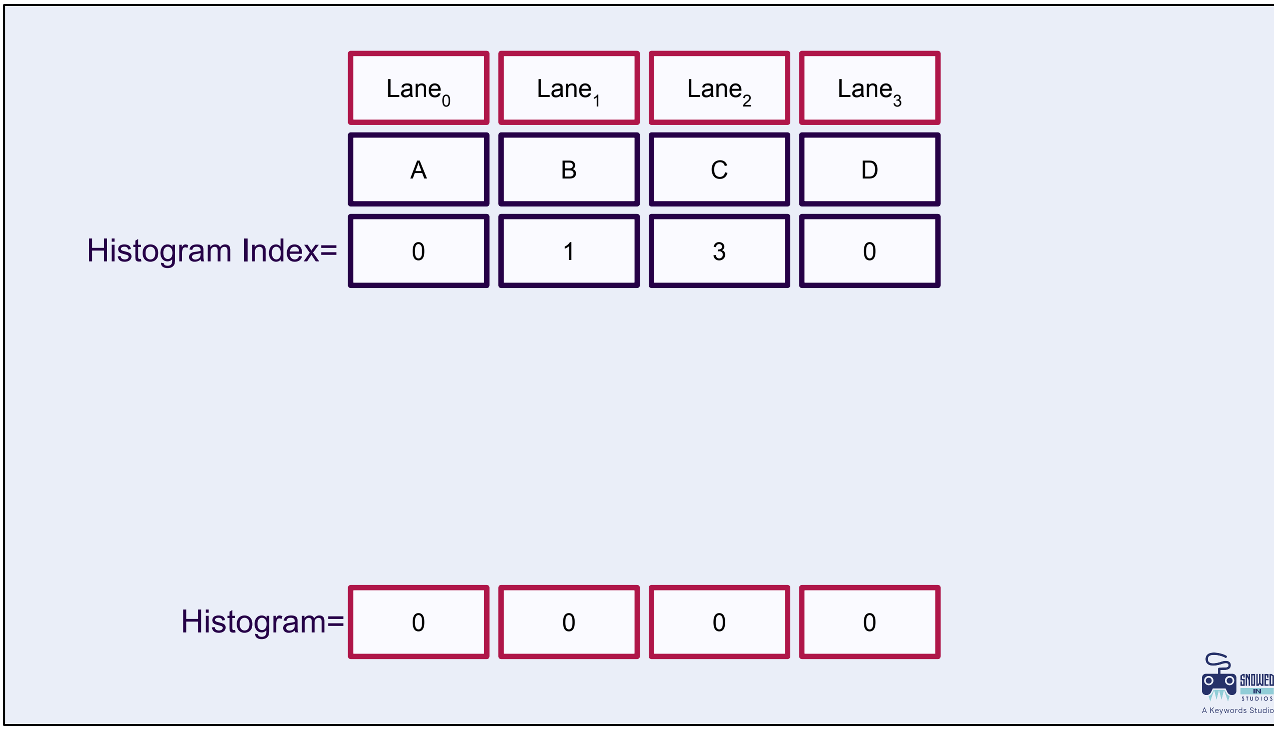
We’re going to investigate tool we can use to optimize atomic adds to a histogram.

```
[numthreads(1024, 1, 1)]
void CS(uint3 dispatchId : SV_DispatchThreadId)
{
    uint bucket = getHistogramBucket(Input[dispatchId.x]);
    InterlockedAdd(OutputHistogram[bucket], 1);
}
```

Here we have a simple program that adds to a histogram.

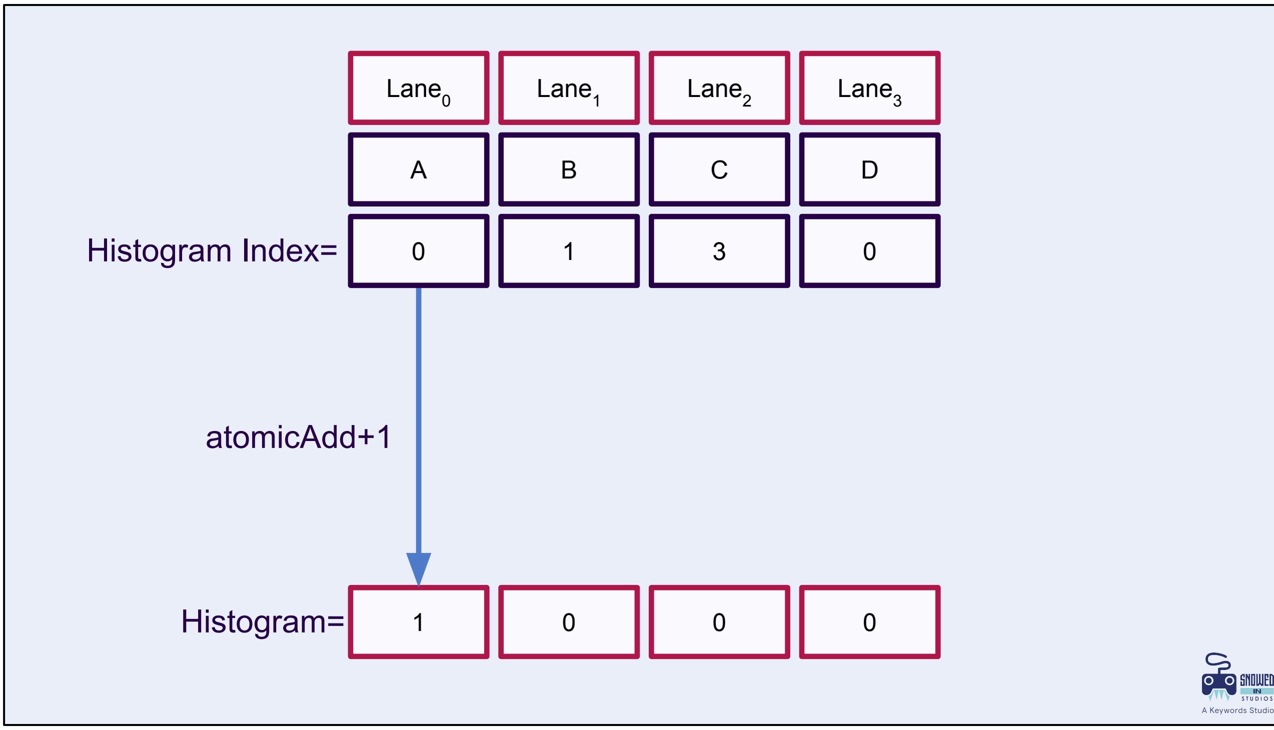


Once again, we want to look at our trusty 4 lane GPU.



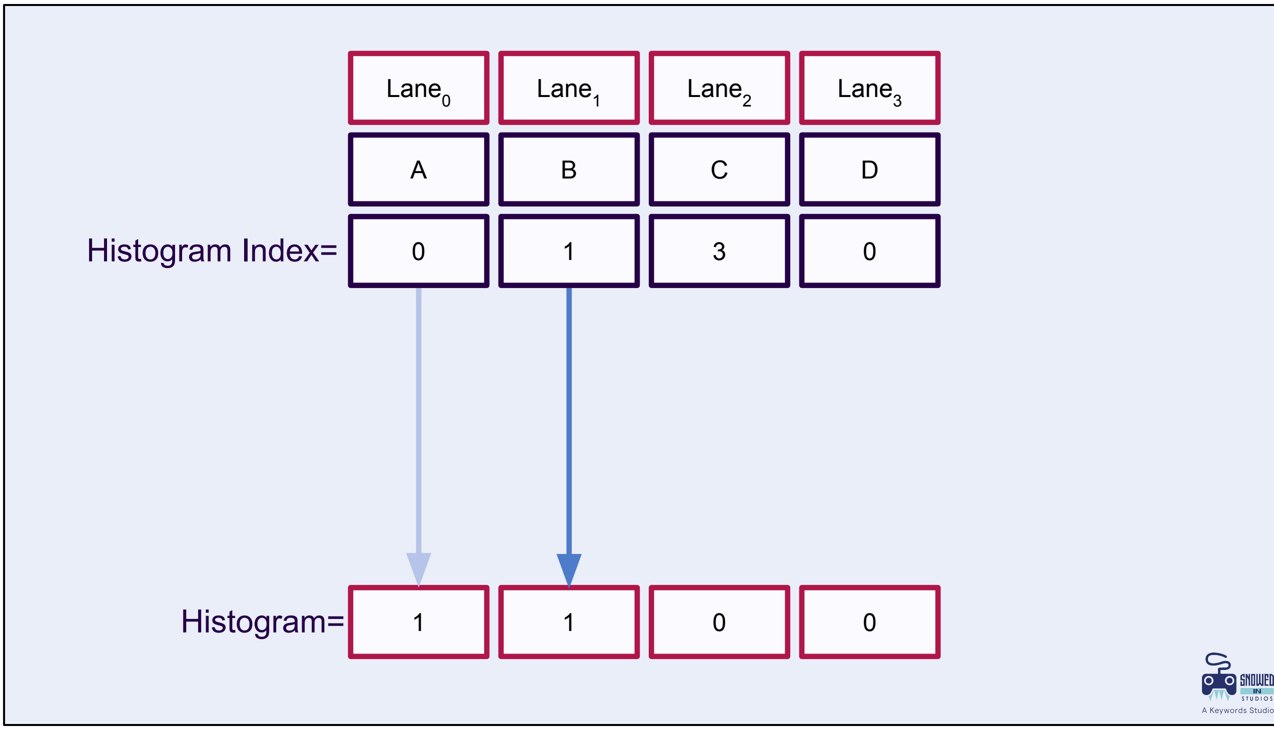
This program will assign an index to each value and each lane will add to their respective buckets using an atomic increment.

click
click
click
click



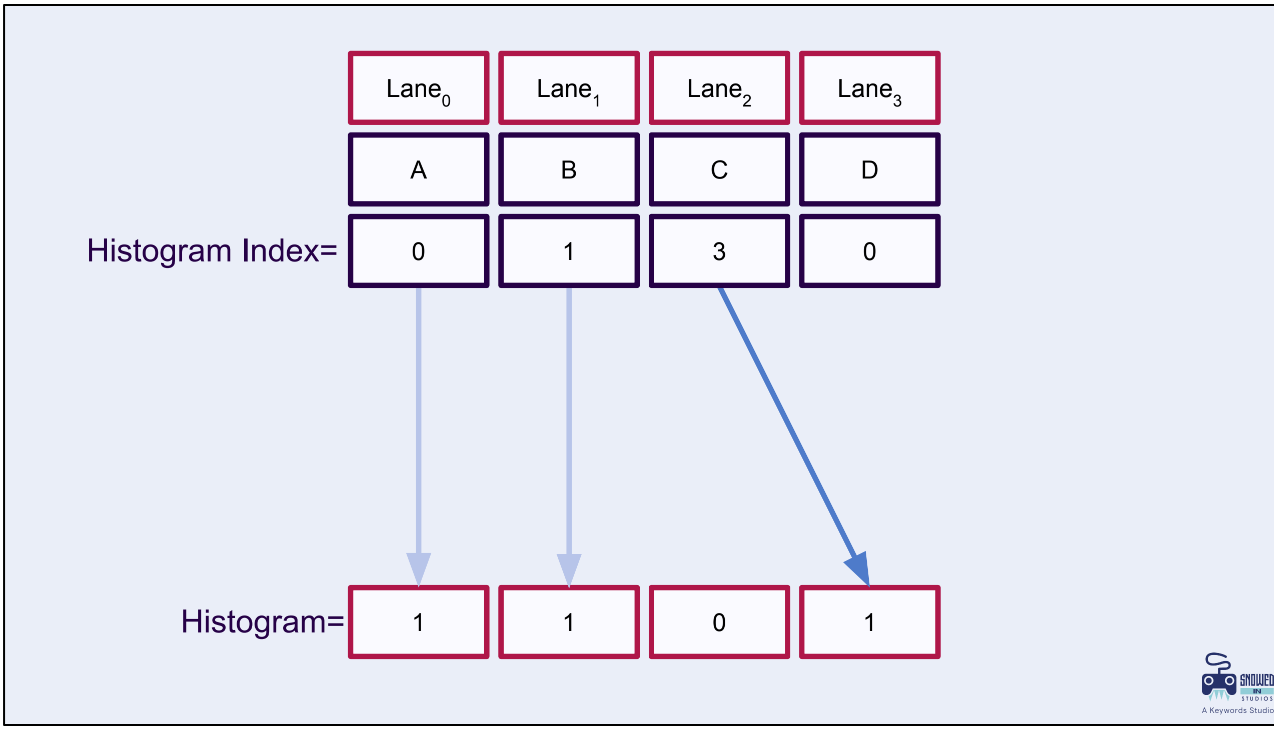
This program will assign an index to each value and each lane will add to their respective buckets using an atomic increment.

click
click
click
click



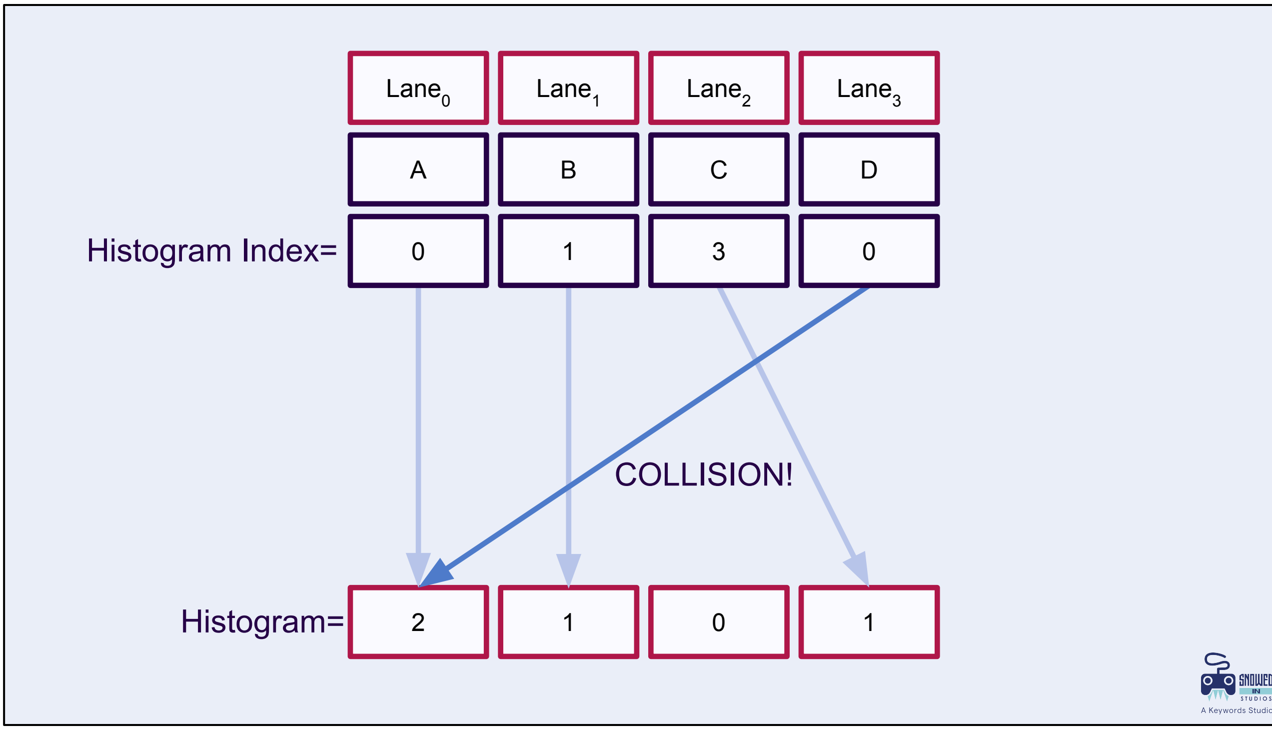
This program will assign an index to each value and each lane will add to their respective buckets using an atomic increment.

click
click
click
click



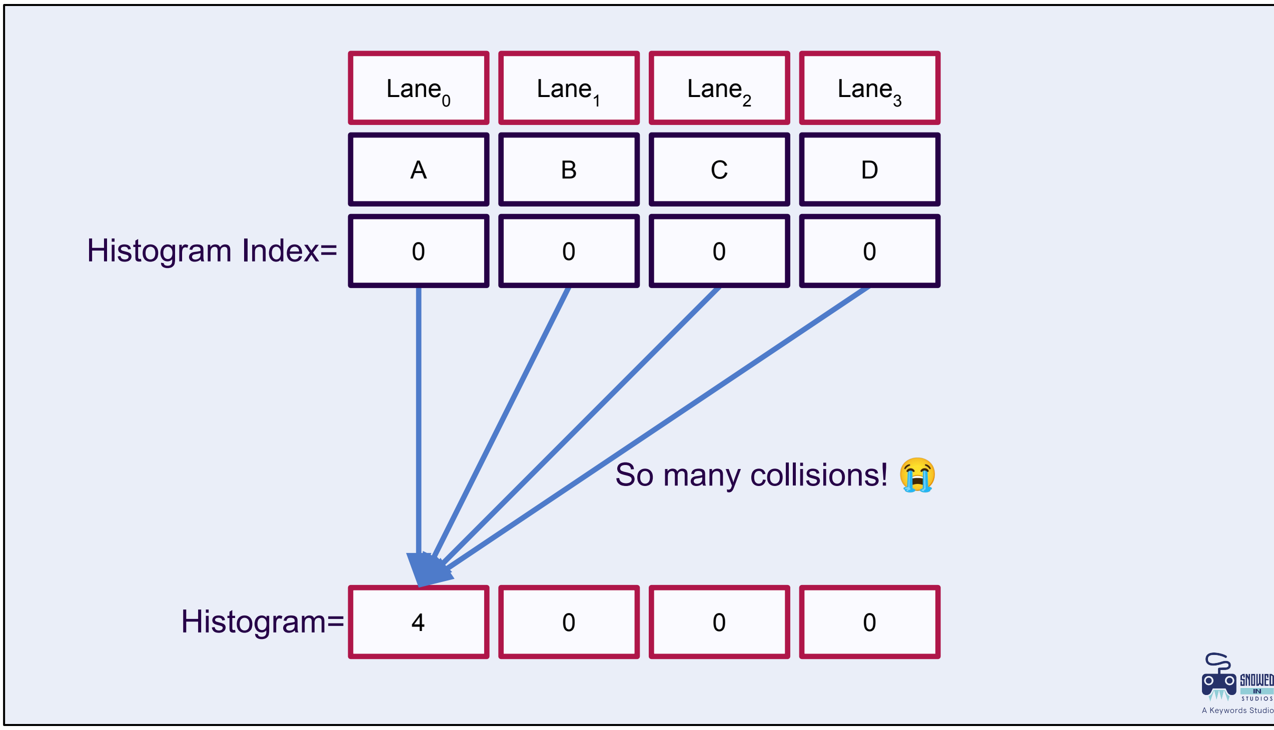
This program will assign an index to each value and each lane will add to their respective buckets using an atomic increment.

click
click
click
click



Sometimes we're going to have a collision.

This can cause a small, probably unnoticeable performance penalty.



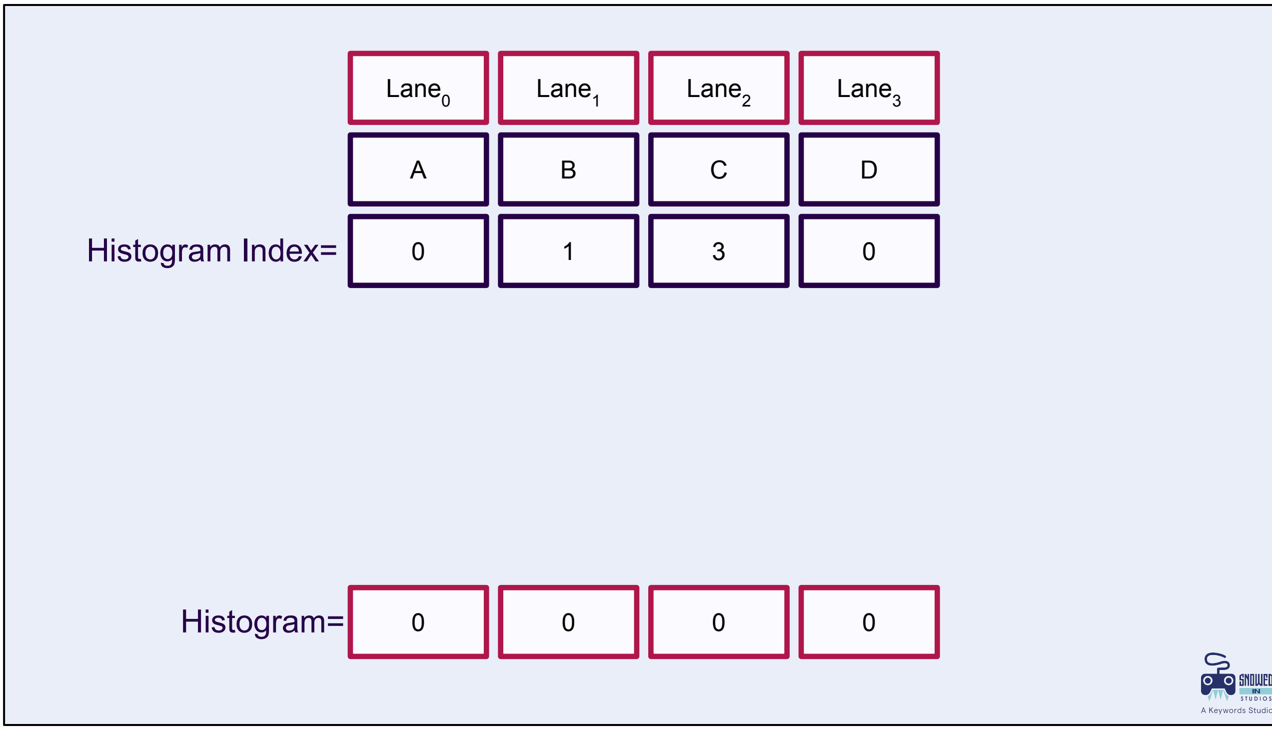
Unless everyone writes to the same location…

256 Bucket Histogram Benchmark

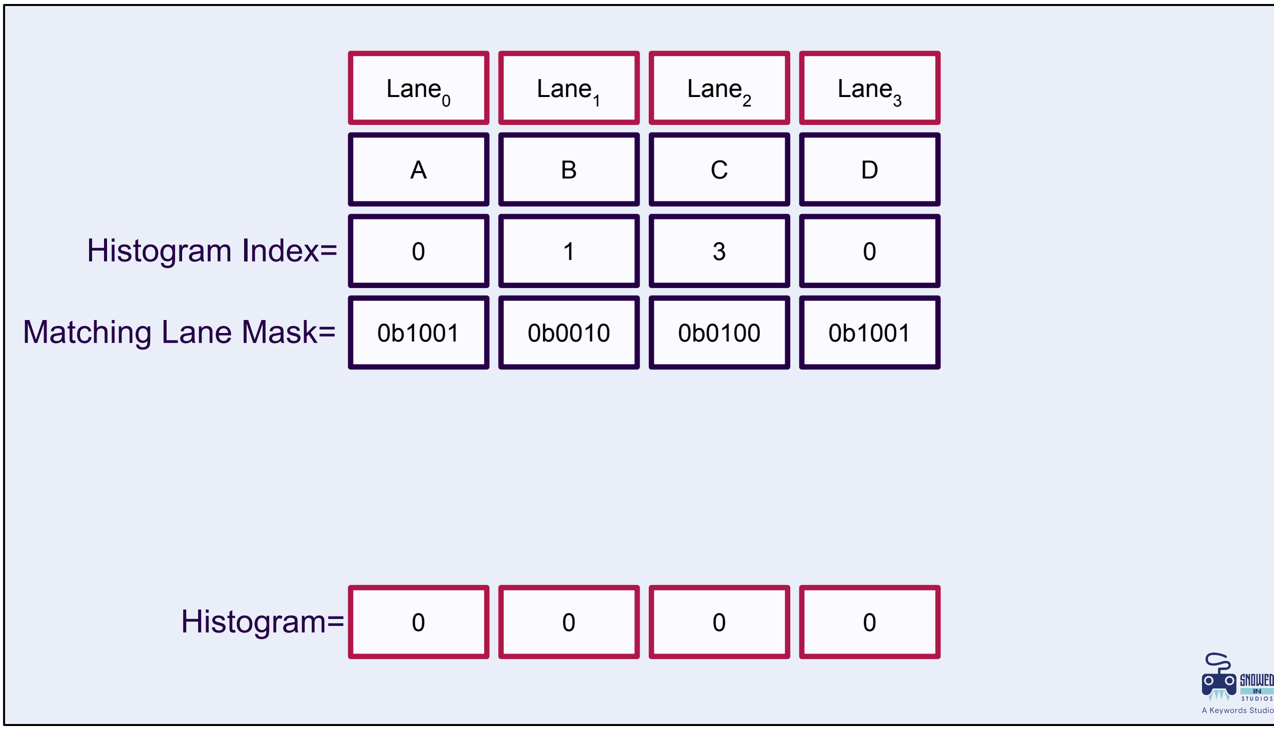
	Global Memory	+Groupshared
Intel(R) Iris(R) Xe Graphics - 16 Million Elements, 256 Buckets		
Minimal Collisions	30ms	1.5ms
Maximum Collisions	30ms	3.5ms
NVIDIA GeForce RTX 3070 Ti - 16 Million Elements, 256 Buckets		
Minimal Collisions	2ms	0.3ms
Maximum Collisions	2ms	2ms

Writing directly to global memory is not bound by our atomic throughput.

But when we use groupshared memory as an intermediate data store, we start seeing some performance penalties.

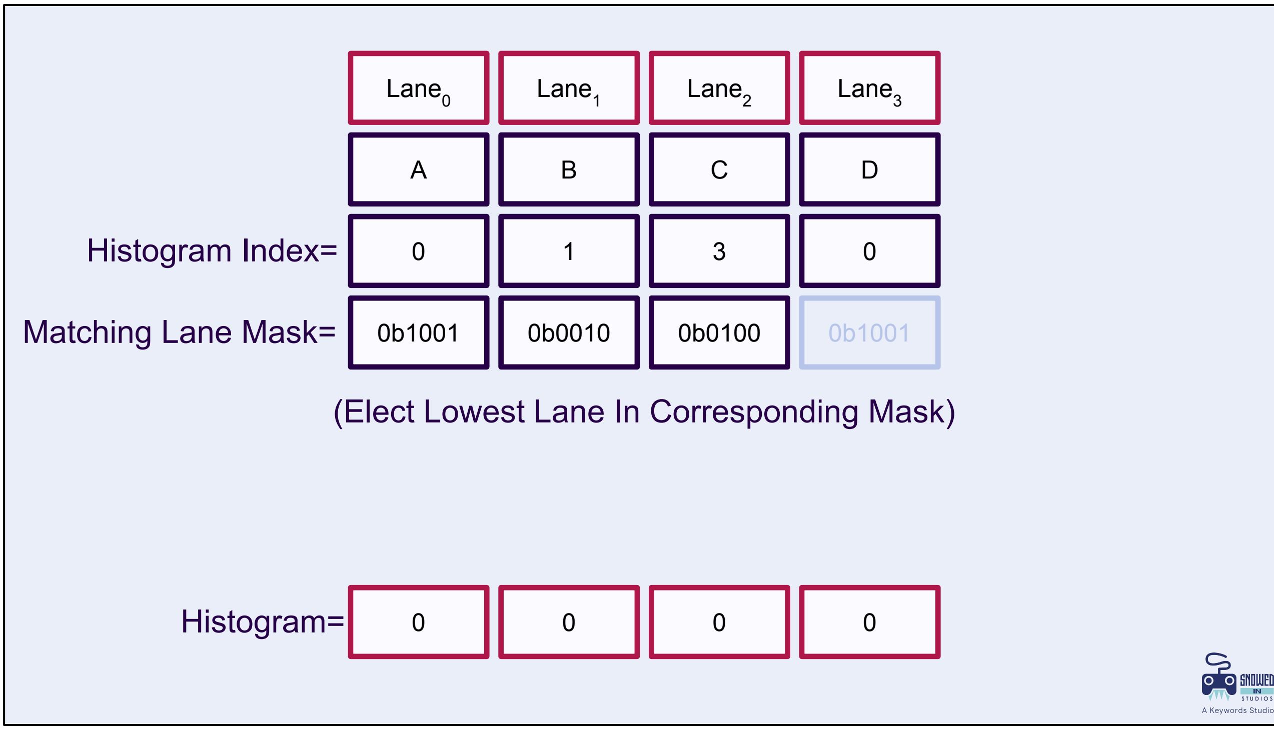


What we want, is some sort of “mask” that would tell us which lanes have the same value as our current lane.

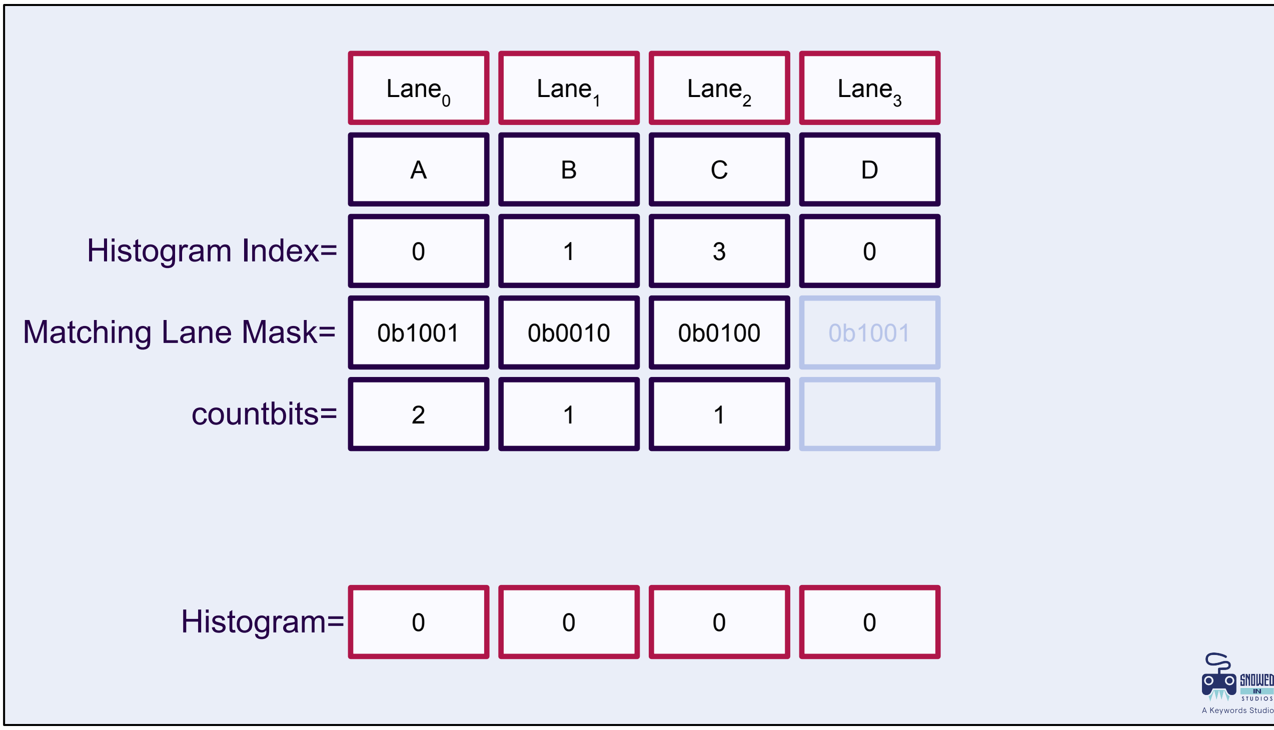


Once we have that mask, we want to elect the lowest lane in each mask.

In this case, note that lane 0 and lane 3 have the same mask since they want to write to the same histogram bucket.

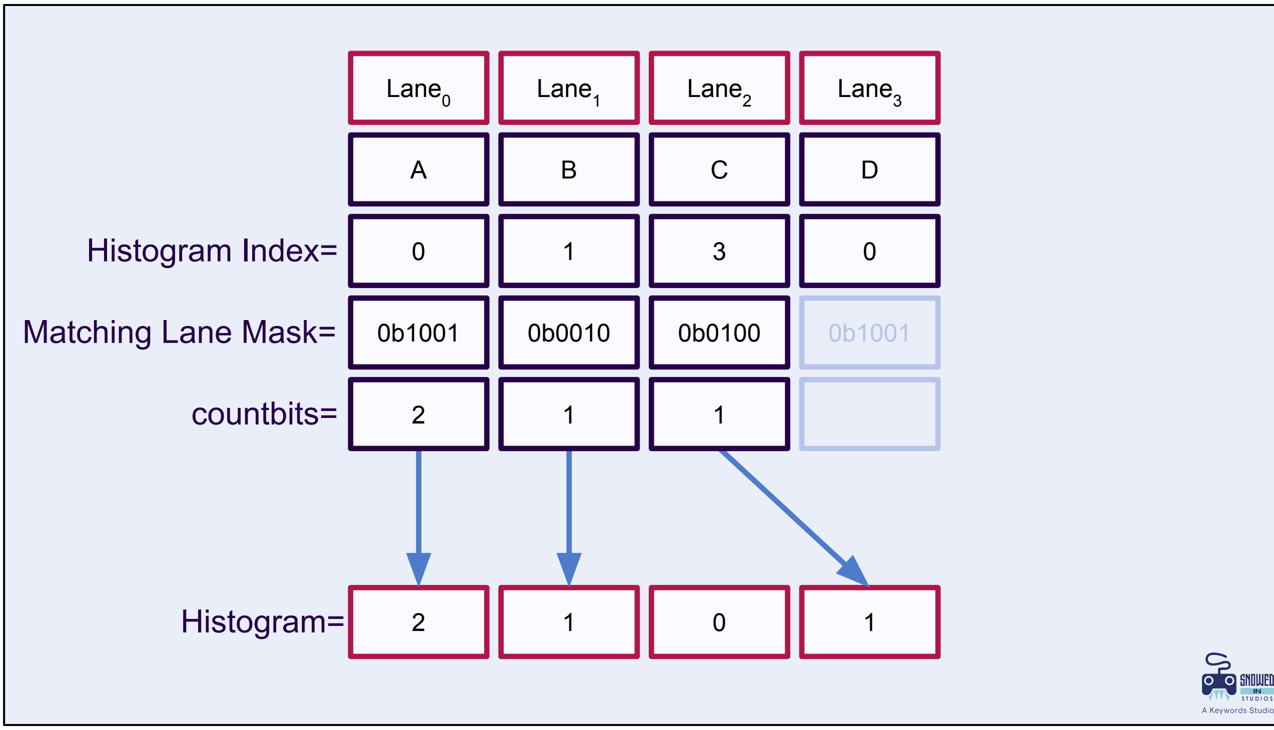


When we have a conflict, we selected Lane 0 as the lowest lane whose bit is set in its respective mask.



Then, it’s simply a matter of counting the bits in our mask and writing to the corresponding location in our histogram.

click



Then, it's simply a matter of counting the bits in our mask and writing to the corresponding location in our histogram.

click

That’s great and all, but how do we get that mask?

How Do We Get The Lane Mask?

- **We have a fixed histogram size (in this case 4 buckets)**

In this case, we have a fixed histogram size with 4 buckets.

How Do We Get The Lane Mask?

- **We have a fixed histogram size (in this case 4 buckets)**
- **We could simply compare each lane to all of its neighbours**
  - Wave lane count amount of shuffles 🙁

We could compare each lane with all of its neighbours.

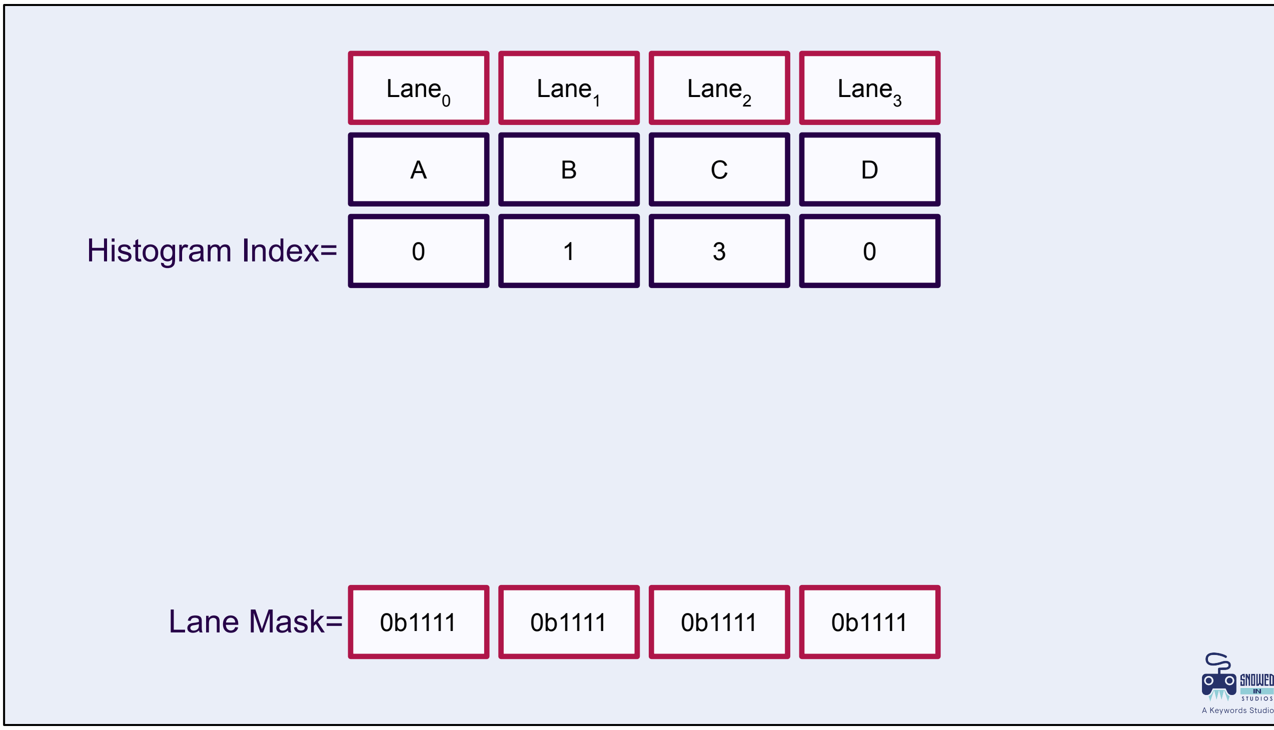
But then we would need an equivalent number of shuffles to the number of lanes on our GPU.

In this case, our “GPU” has 4 lanes. But in other cases we might see wave sizes of 32 or 64.

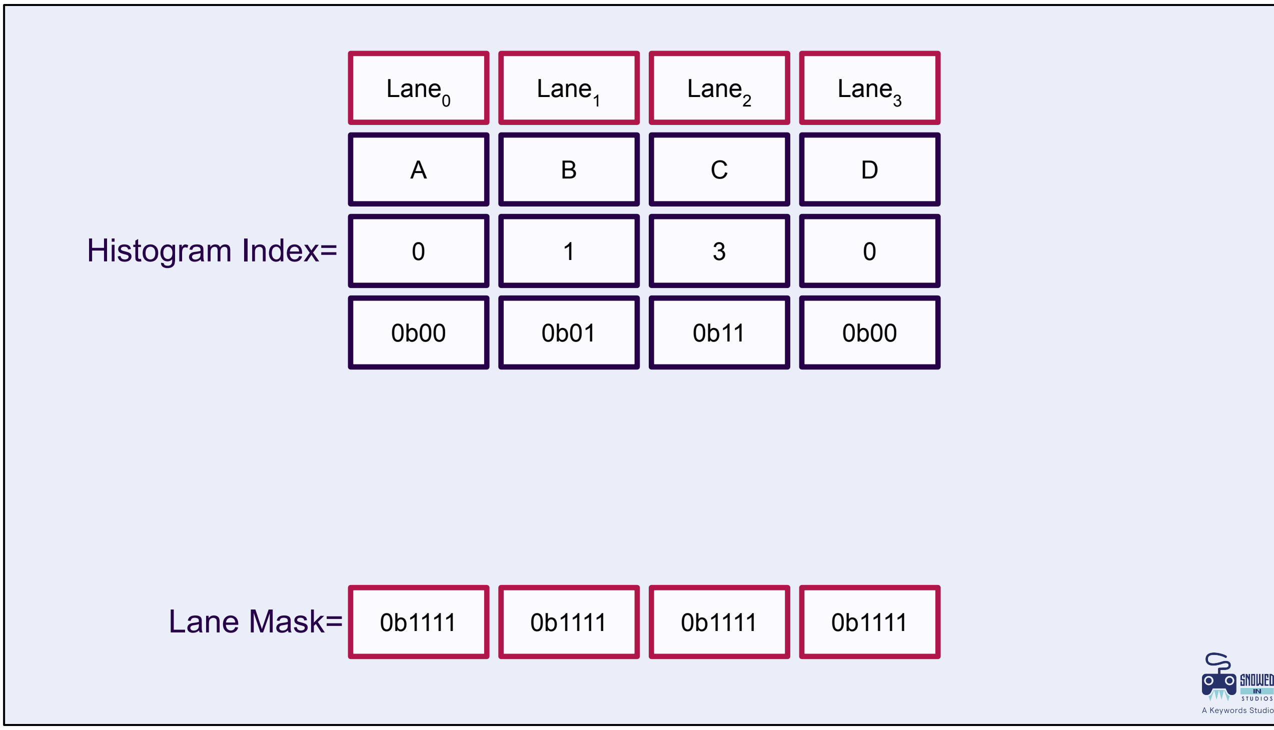
How Do We Get The Lane Mask?

- **We have a fixed histogram size (in this case 4 buckets)**
- **We could simply compare each lane to all of its neighbours**
  - Wave lane count amount of shuffles 🙁
- **What if we compare each lane bit by bit compare?**
  - Let’s look at an example

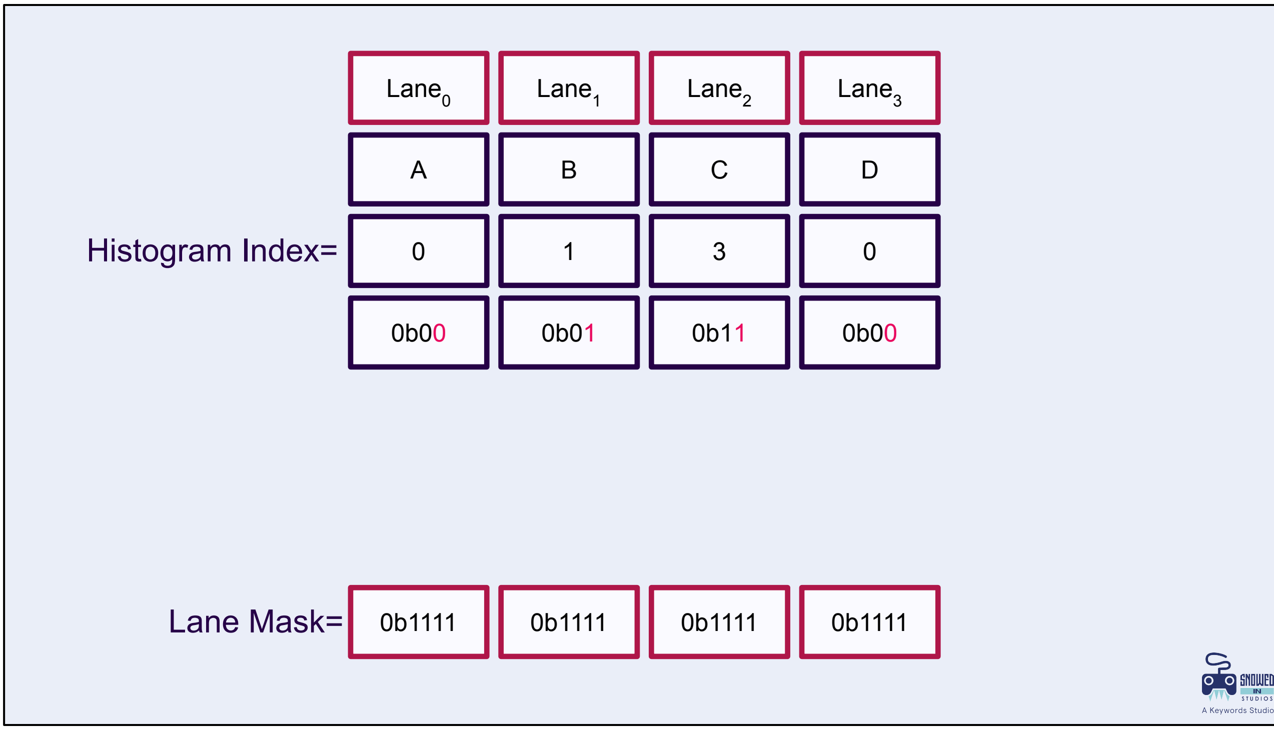
What if we compare each lane bit by bit?



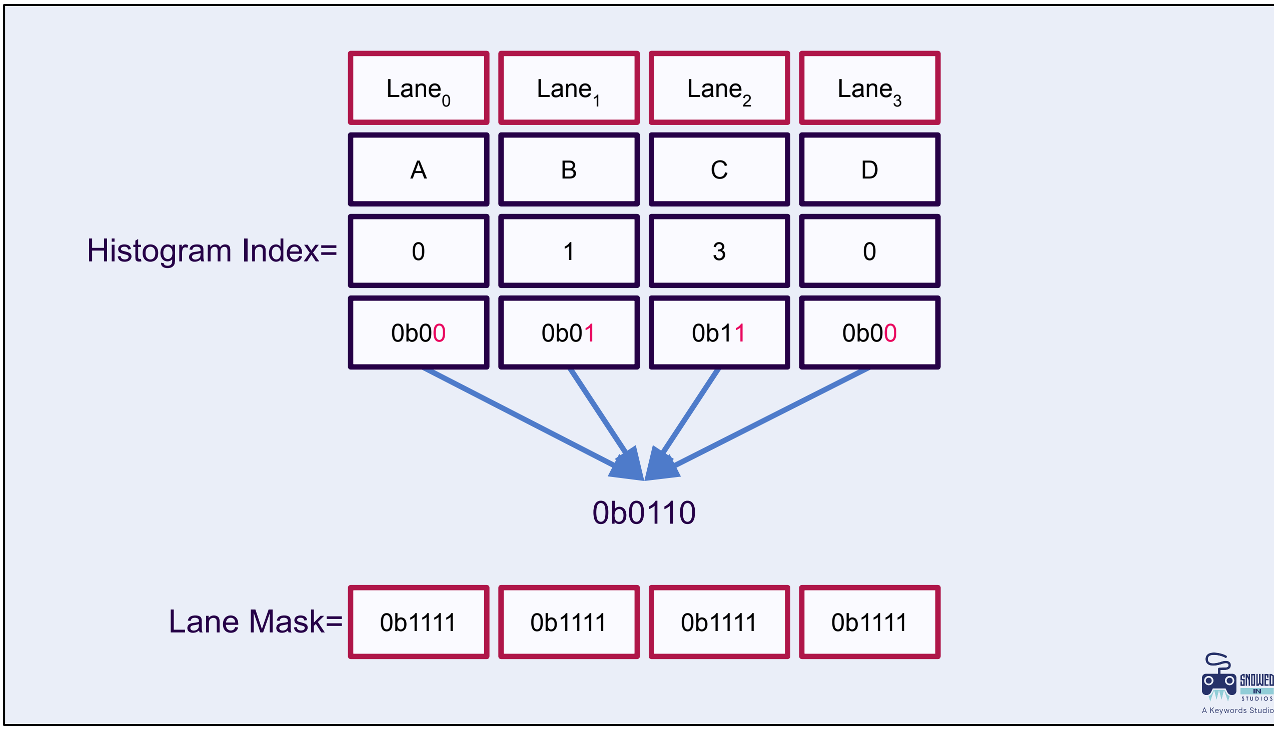
Let’s start by assigning each lane a unique “Lane Mask” value and set it to 1s.



Let’s convert our histogram index to binary.



We want to walk through the bits of that index one at a time.

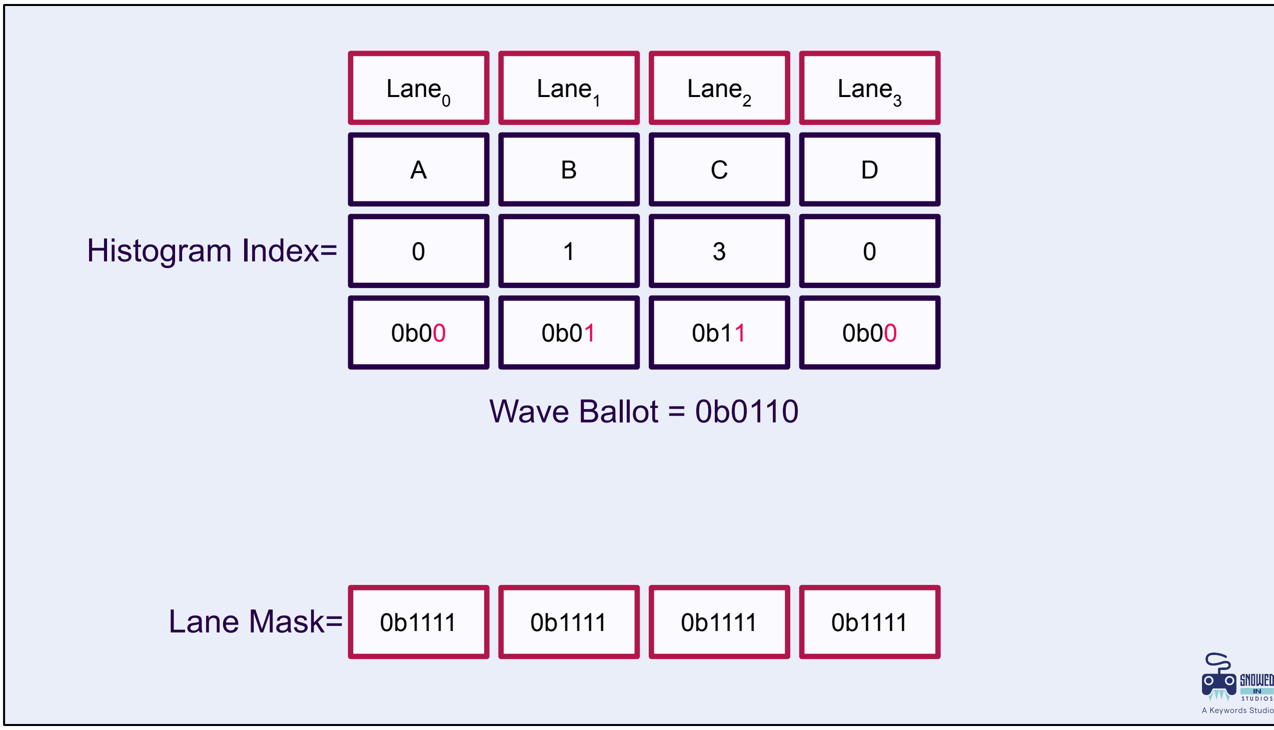


And we want to collect all the bits into a single value.

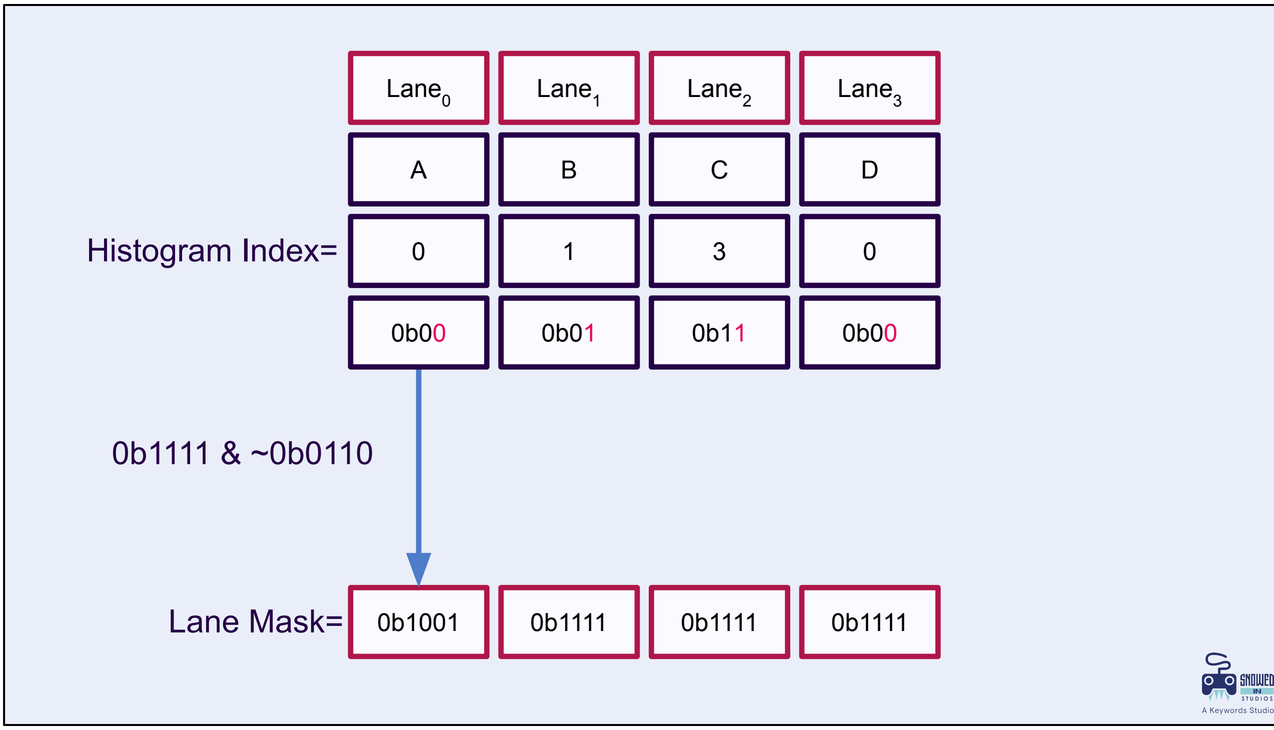
(Take some extra time to describe how this works and why we want it)

This can be achieved with a call to `WaveActiveBallot`.

WaveActiveBallot simply allows us to collect all of the bits in our lanes into a single integer value visible to all lanes at once.



Once we have this ballot, we have a mask that tells us which lane's first bits are 1 and which lane's bits are 0.

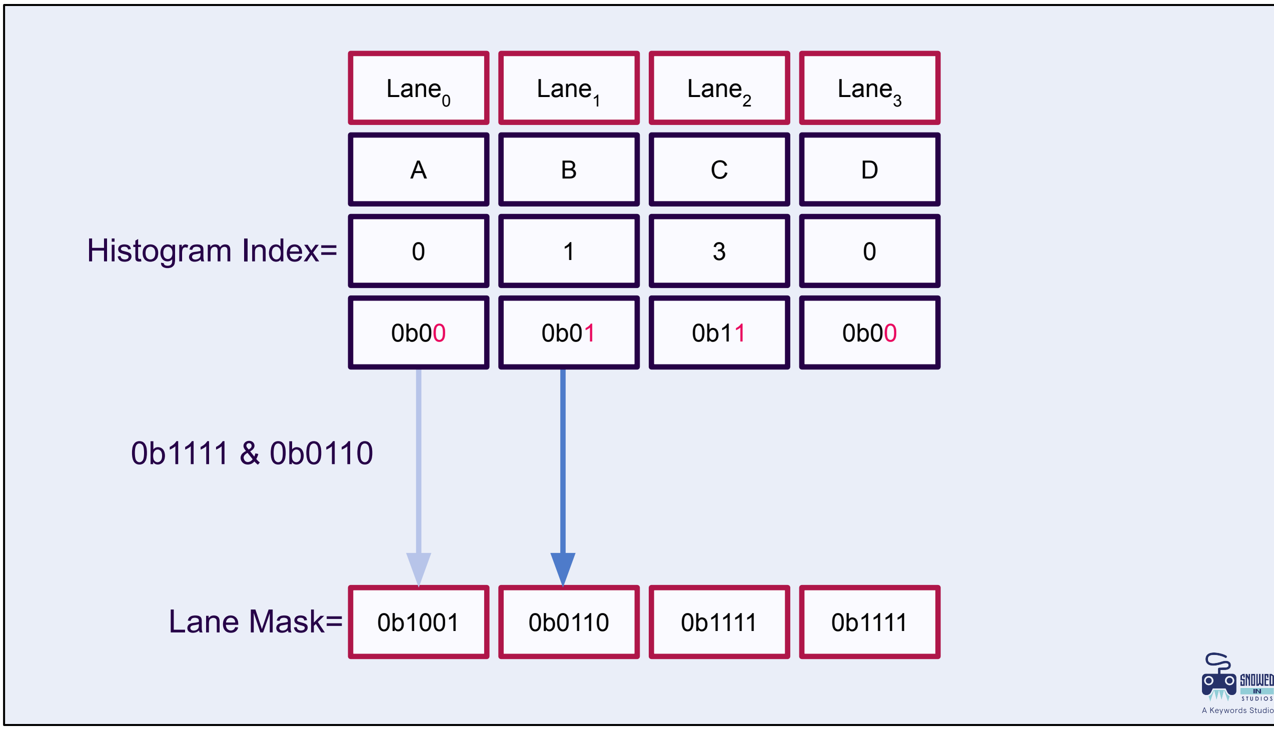


Since we’re trying to find out which lanes match our own.

We use this mask as a progressive equality check.

Since our current bit is zero, then we want to invert the ballot mask.

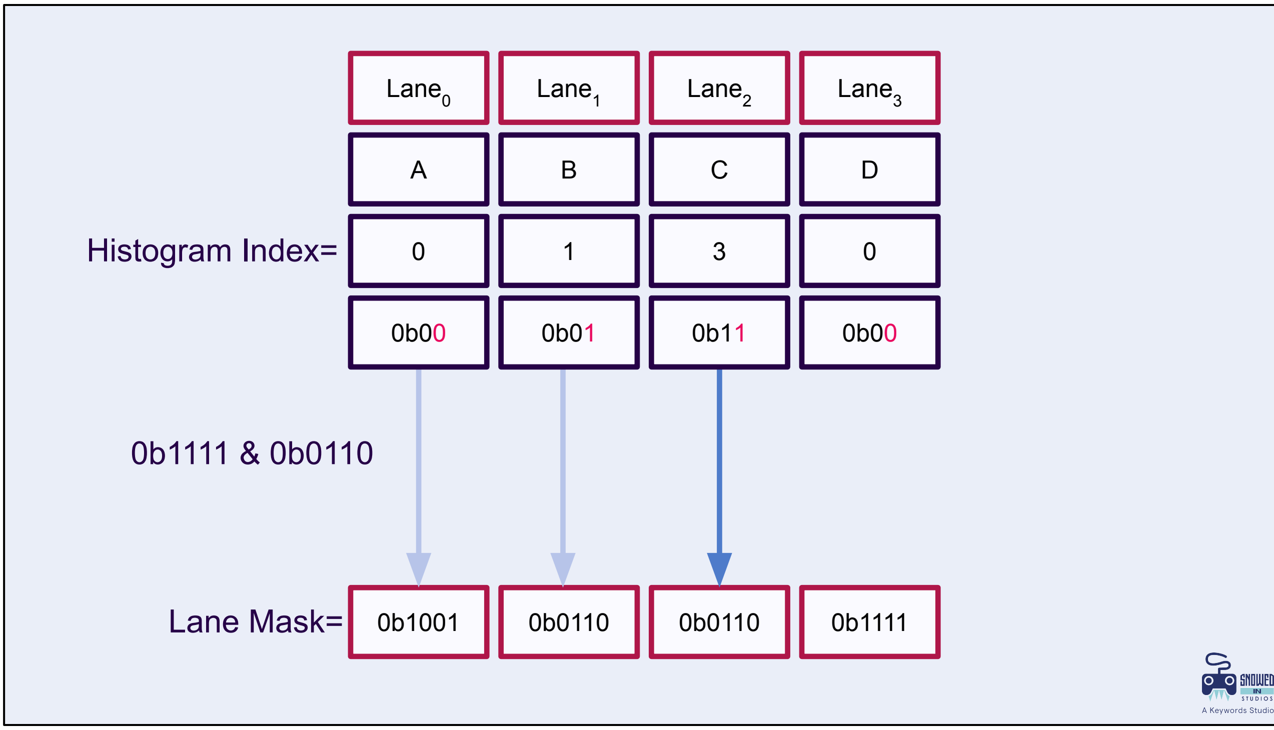
Each lane whose first bit was zero should get a “true” and each lane whose first bit was one should get a “false”.



Then we continue this process.

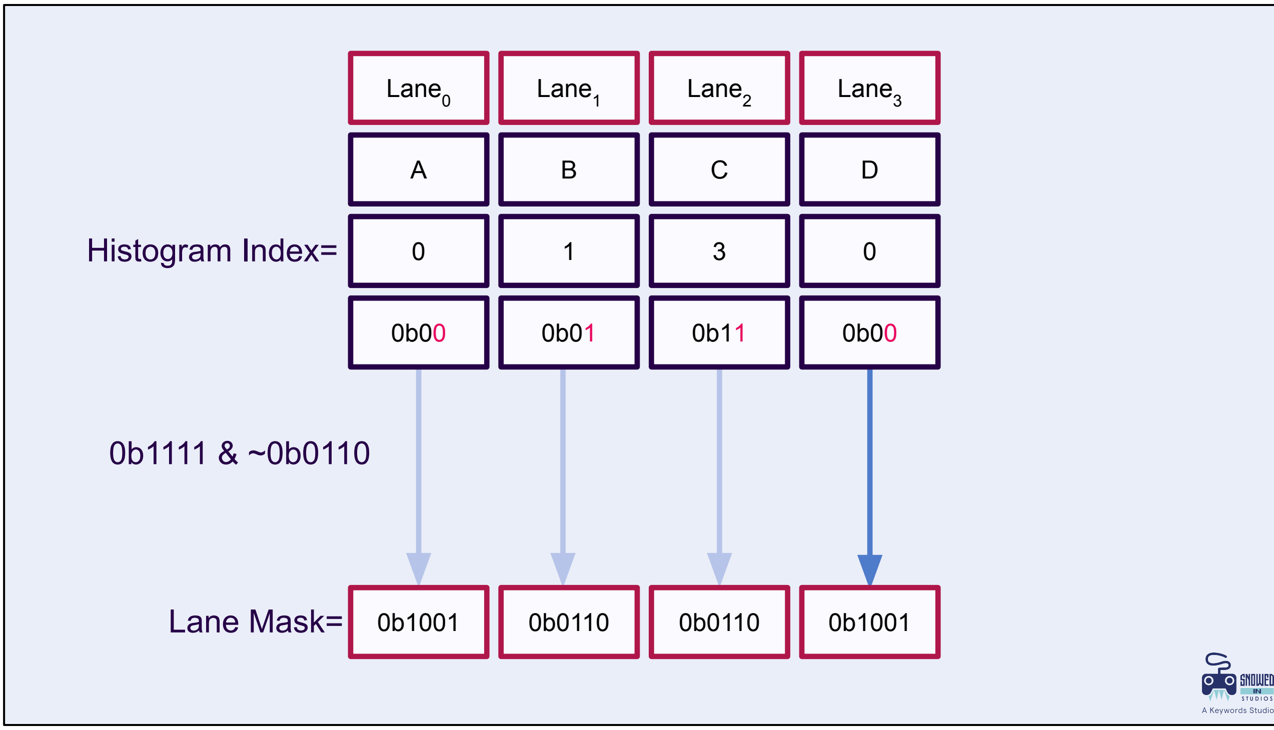
In this case, our first bit is one.

As a result, we keep our ballot mask as it is.



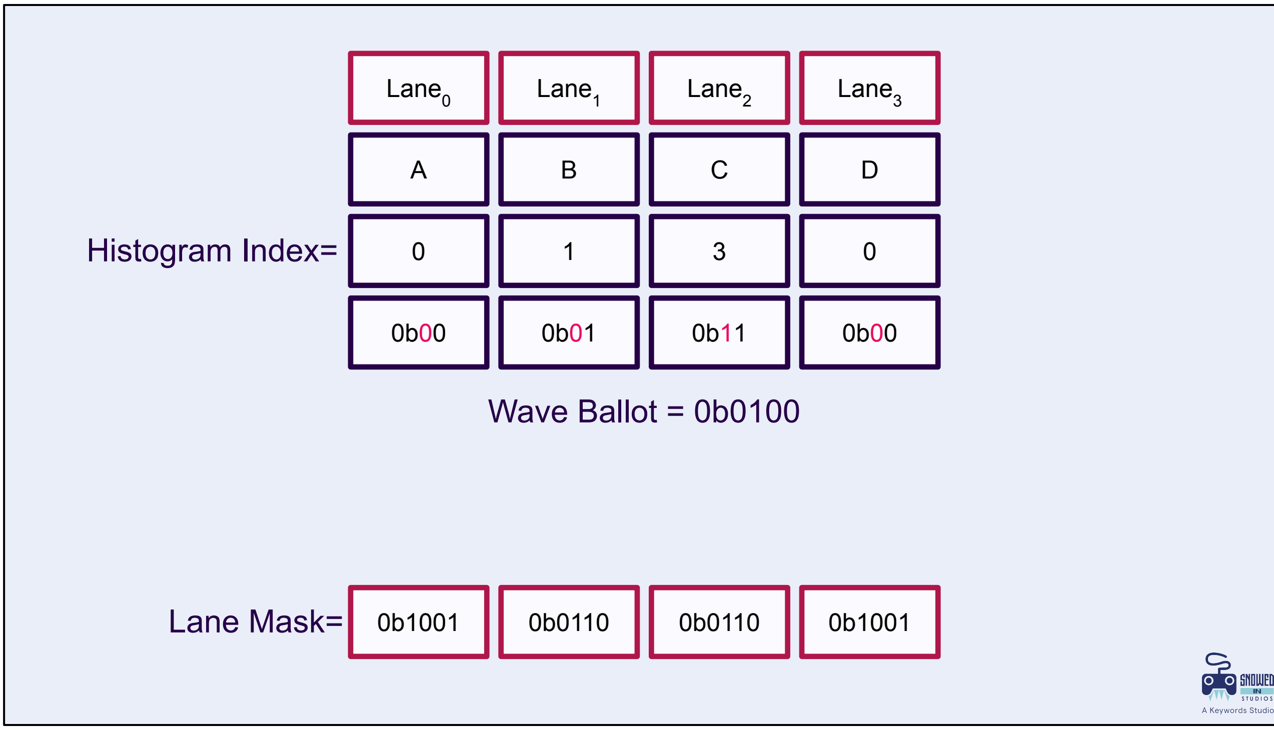
Lane0
Lane1
Lane2
Lane3
A
B
C
D
Histogram Index=
0
1
3
0
0b00
0b01
0b11
0b00
0b1111 & 0b0110
Lane Mask=
0b1001
0b0110
0b0110
0b1111

click



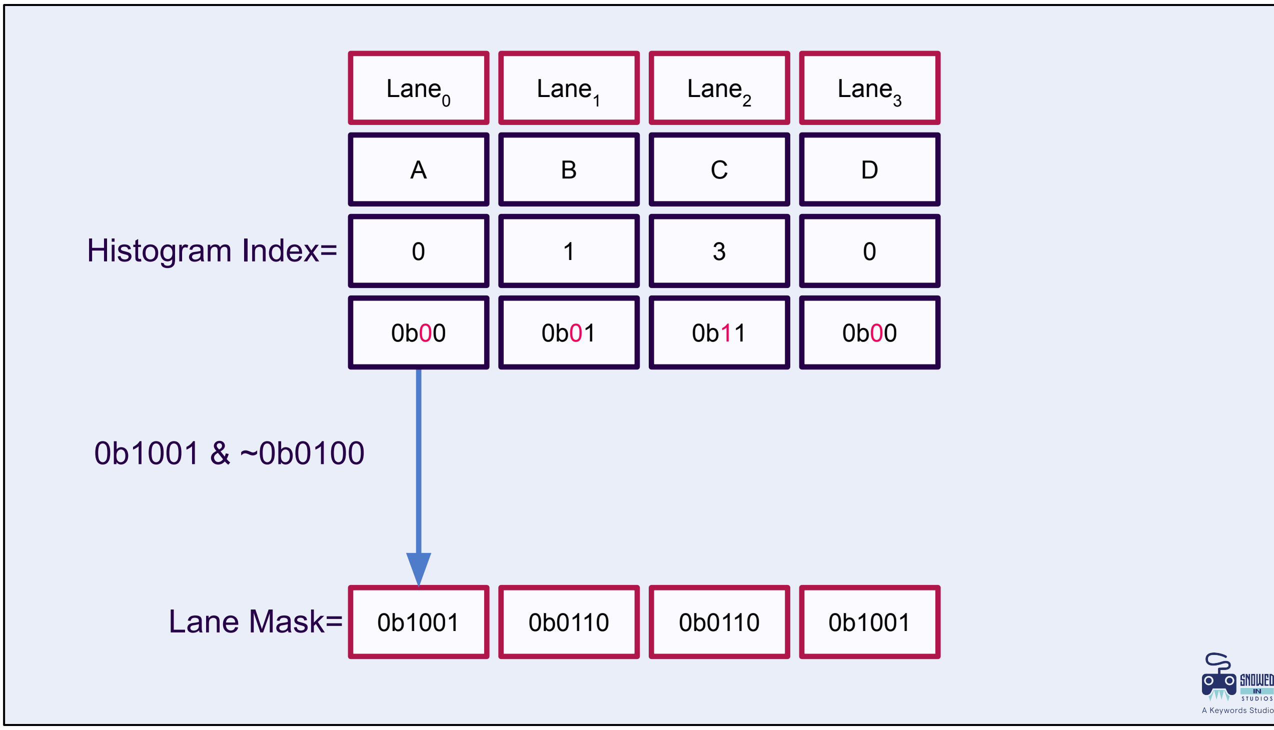
(Pause so people can look at the current masks)

click



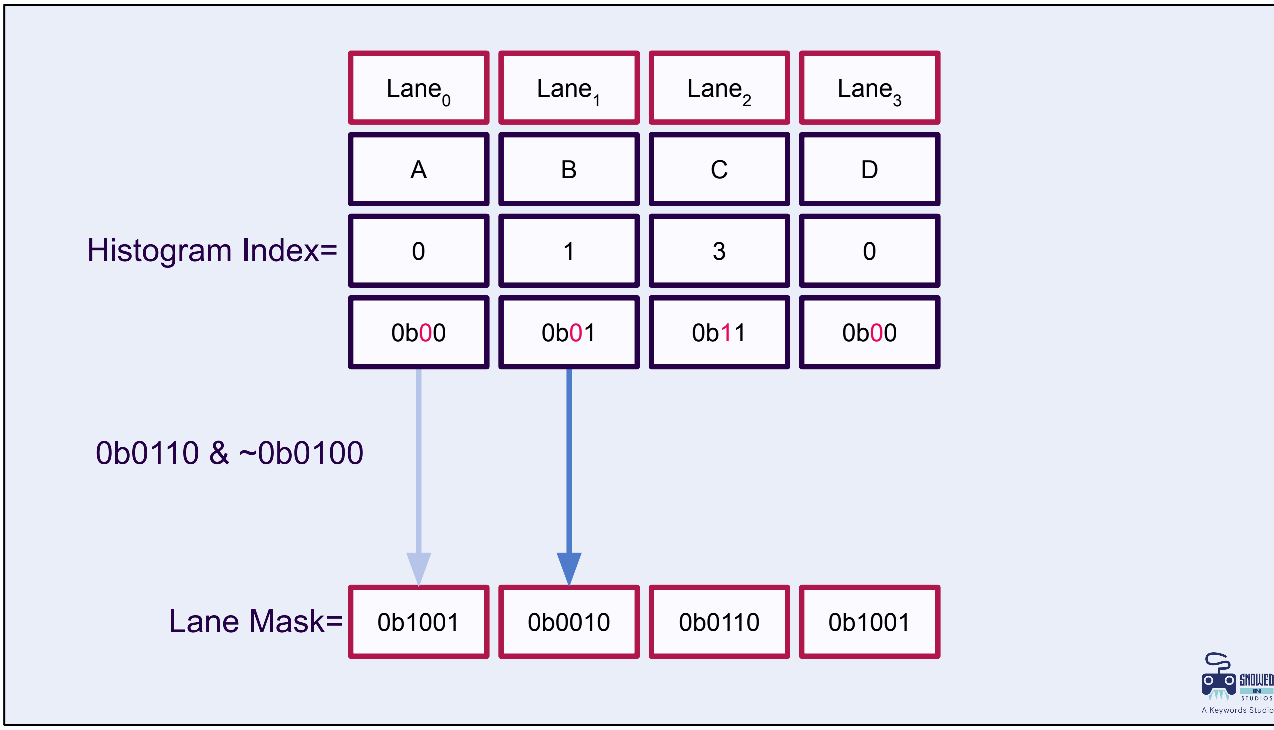
You'll notice that we're progressively disqualifying neighbouring lanes from matching our current one.

We repeat this process up to the number of bits that we want to use in our equality test.

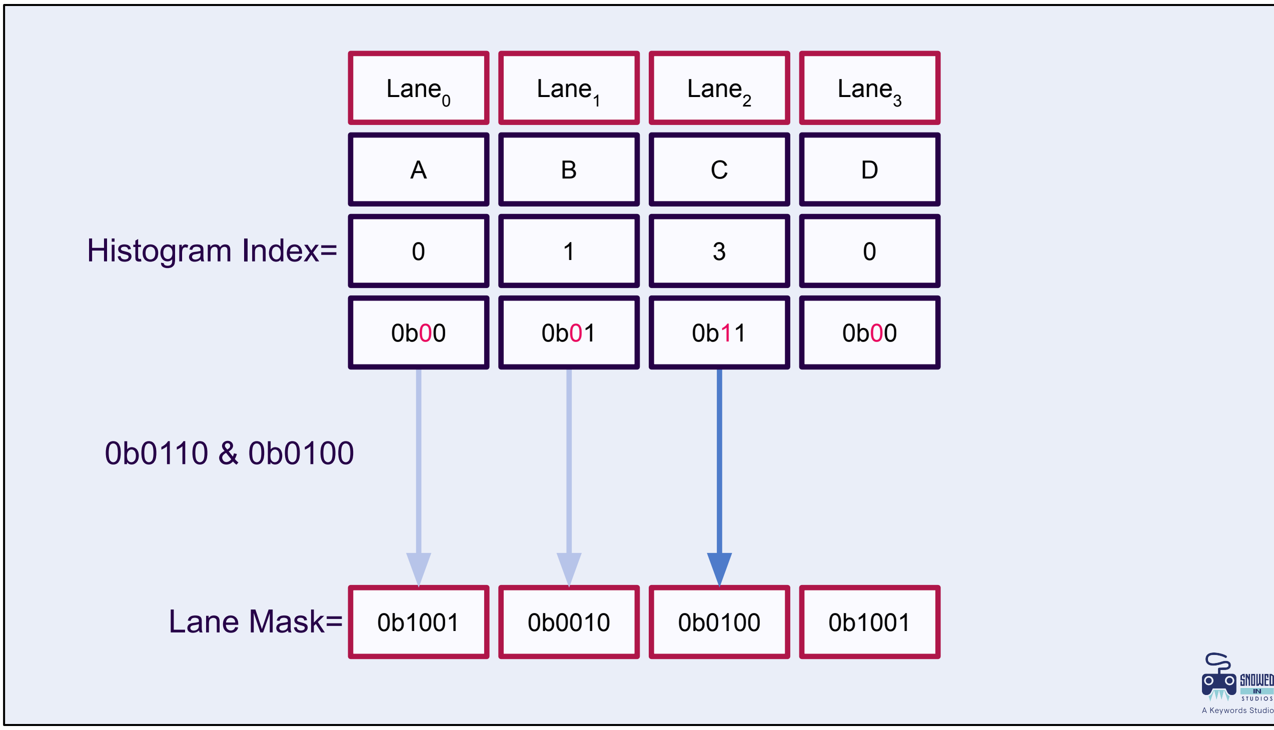


Lane0
Lane1
Lane2
Lane3
A
B
C
D
Histogram Index=
0
1
3
0
0b00
0b01
0b11
0b00
0b1001 & ~0b0100
Lane Mask=
0b1001
0b0110
0b0110
0b1001
A Keywords Studio

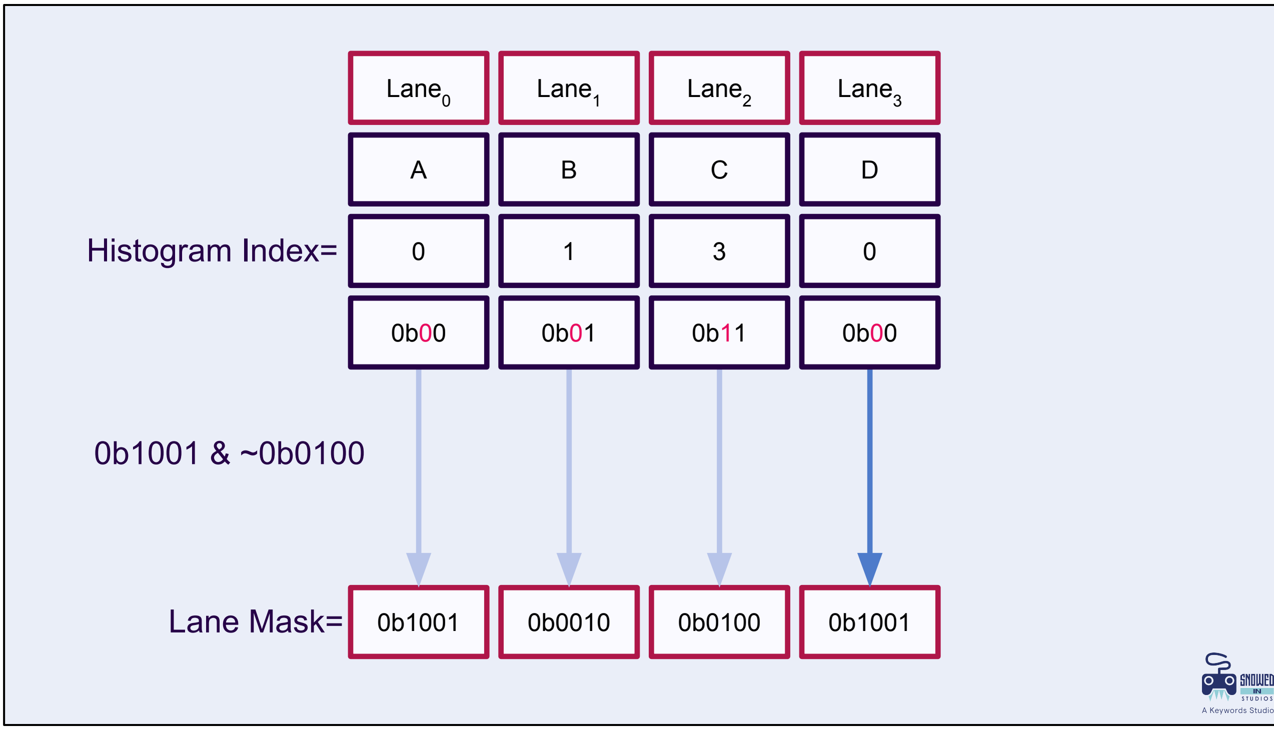
click


Lane0
Lane1
Lane2
Lane3
A
B
C
D
Histogram Index=
0
1
3
0
0b00
0b01
0b11
0b00
0b0110 & ~0b0100
Lane Mask=
0b1001
0b0010
0b0110
0b1001
A Keywords Studio

click

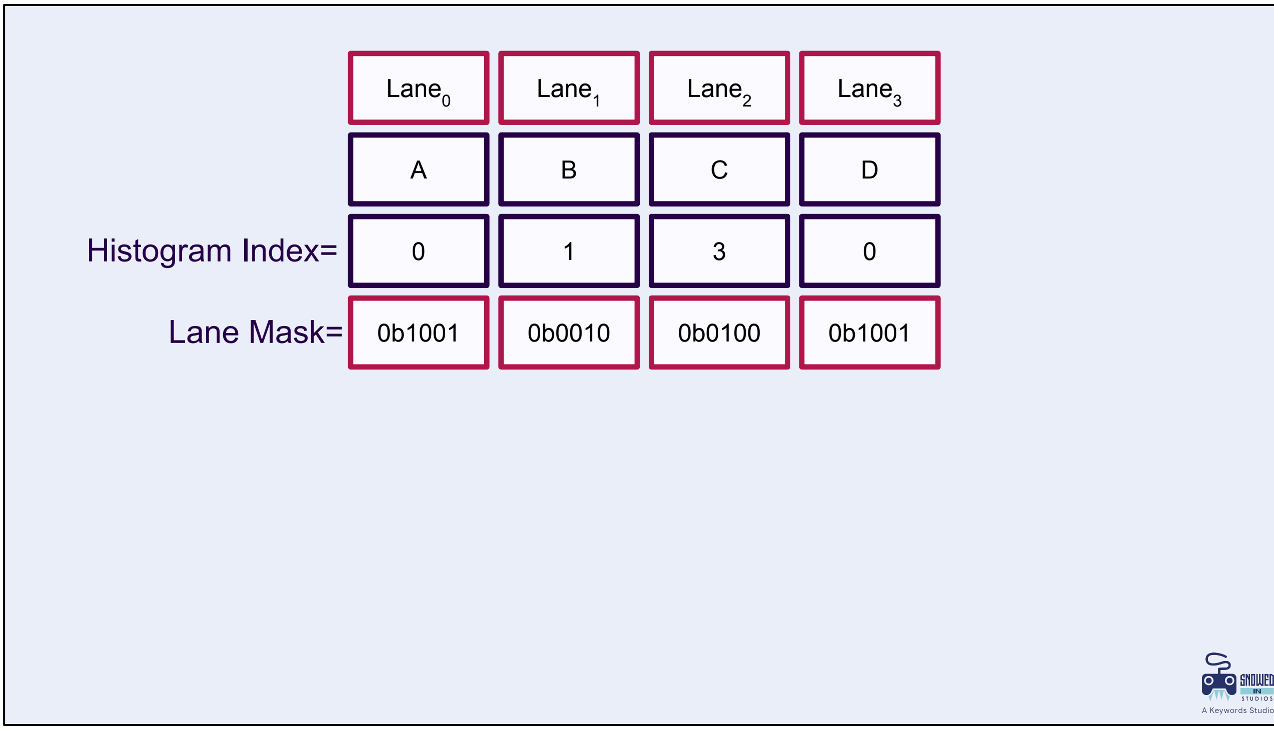

Lane0
Lane1
Lane2
Lane3
A
B
C
D
Histogram Index=
0
1
3
0
0b00
0b01
0b11
0b00
0b0110 & 0b0100
Lane Mask=
0b1001
0b0010
0b0100
0b1001

click

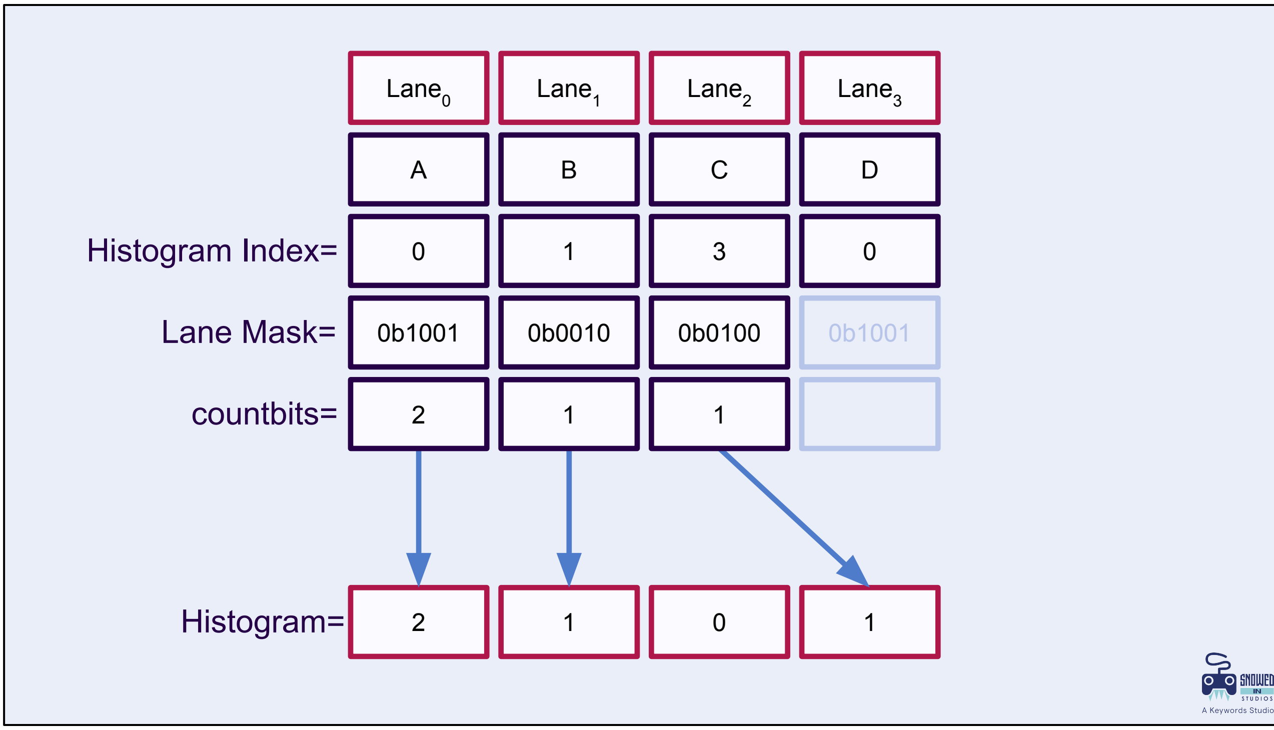


(Pause for a moment so people can look at the lane mask)

click



With two iterations, each lane now has a mask that represents the lane’s whose value match our own.



Now we simply go through the routine that we described above and write to our histogram.

How Do We Get The Lane Mask?

- **We have a fixed histogram size (in this case 4 buckets)**
- **We could simply compare each lane to all of its neighbours**
  - Wave lane count amount of shuffles 🙁
- **What if we compare each lane bit by bit compare?**
  - countbits(bucketCount) number of iterations
    - Supporting 8 bits takes 8 iterations 😁

For a histogram of 256 values, you simply need to iterate 8 times.

How Do We Get The Lane Mask?

- **We have a fixed histogram size (in this case 4 buckets)**
- **We could simply compare each lane to all of its neighbours**
  - Wave lane count amount of shuffles 🙁
- **What if we compare each lane bit by bit compare?**
  - countbits(bucketCount) number of iterations
    - Supporting 8 bits takes 8 iterations 😁
    - Supporting 32 bits takes 32 iterations 🙁

But for supporting 32 bits you need 32 iterations…

How Do We Get The Lane Mask?

- **We have a fixed histogram size (in this case 4 buckets)**
- **We could simply compare each lane to all of its neighbours**
  - Wave lane count amount of shuffles 🙁
- **What if we compare each lane bit by bit compare?**
  - countbits(bucketCount) number of iterations
    - Supporting 8 bits takes 8 iterations 😁
    - Supporting 32 bits takes 32 iterations 🙁
    - Depends on a fast WaveActiveBallot

A Keywords Studio

And this depends on a fast ballot method.

If requesting a ballot is secretly a waterfall loop, or even simply a reduction the cost of this approach soars.

```
;           uint4 ballot = WaveActiveBallot(laneBit);
    v_cmp_eq_u32_e64 s0, 0, v3
    v_cmp_ne_u32_e32 vcc_lo, 0, v3
```

Which thankfully, in the case of RDNA and AMD’s offline compiler is a nice and tidy 2 instructions.

```
uint4 WaveActiveMatch(uint value, uint bitCount)
{
    uint4 laneMask = uint4(0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF);
    for (uint i = 0; i < bitCount; i++)
    {
        uint laneBit = (value >> i) & 1;
        uint4 ballot = WaveActiveBallot(laneBit);

        // Make sure to flip our ballot if our bit is 0.
        laneMask.x &= laneBit ? ballot.x : ~ballot.x;
        laneMask.y &= laneBit ? ballot.y : ~ballot.y;
        laneMask.z &= laneBit ? ballot.z : ~ballot.z;
        laneMask.w &= laneBit ? ballot.w : ~ballot.w;
    }

    return laneMask;
}
```

Here is the algorithm that we've just described to capture the lane mask.

As well as the election process to determine which lanes will write to the histogram.

click
click

```
uint4 ballot = WaveActiveMatch(bucket, 8);
uint matchCount = WaveBallotBitCount(ballot);
uint electedLane = WaveBallotLSB(ballot);

[branch]
if (WaveGetLaneIndex() == electedLane)
{
    InterlockedAdd(OutputHistogram[bucket], matchCount);
}
```

Here is the algorithm that we’ve just described to capture the lane mask.

As well as the election process to determine which lanes will write to the histogram.

click
click

256 Bucket Histogram Benchmark

	Global Memory	+Groupshared	+Wave Match
Intel(R) Iris(R) Xe Graphics - 16 Million Elements, 256 Buckets			
Minimal Collisions	30ms	1.5ms	2ms
Maximum Collisions	30ms	3.5ms	2ms
NVIDIA GeForce RTX 3070 Ti - 16 Million Elements, 256 Buckets			
Minimal Collisions	2ms	0.3ms	0.4ms
Maximum Collisions	2ms	2ms	0.4ms

If we look at some performance numbers.

You'll note that regardless of the number of collisions, you get the same performance characteristics.

2ms on my intel integrated GPU and 0.4ms on my RTX 3070 Ti.

Using wave intrinsics you can be confident that you won't be surprised by sudden spikes in your histogram calculation.

Bonus Slides: Things I Dream About

Things I Dream About

- **More expressive intrinsics**
  - HLSL Shuffle Up/Shuffle Down/Xor Shuffle
    - Already available with subgroups operations!

Especially in HLSL, it would be nice to be able to access more expressive intrinsics.

Thankfully, for my primary use case I can use inline Spir-V to access these intrinsics.

Things I Dream About

- **More expressive intrinsics**
  - HLSL Shuffle Up/Shuffle Down/Xor Shuffle
    - Already available with subgroups operations!
- **Don’t waterfall loop my shuffles (please)**

Additionally, it can be difficult to develop more complex intrinsics when many intrinsics degenerate to a waterfall loop in a variety of cases.

Understandably, I can see that hardware limitations might lead to needing to use a waterfall loop.

But this is why I called them dreams.

Things I Dream About

- **More expressive intrinsics**
  - HLSL Shuffle Up/Shuffle Down/Xor Shuffle
    - Already available with subgroups operations!
- **Don’t waterfall loop my shuffles (please)**
- **More shader compilation tools**
  - RGA Good!
    - No source correlation for SPIR-V Debug Info
    - Currently using a modified version of LLPC to get source code correlation
  - Intel disassembly on Windows?

Finally, more shader compilation tools would be incredible.

RGA is a great tool, but the lack of source code correlation with graphics Spir-V makes it harder to correlate the disassembly to the source code that generated it.

You may have noticed that I have source correlations in the disassembly I’ve presented. I use a modified version of DXC and LLPC to allow for debug instructions to make it all the way to the binary.

On top of that, more tools for different vendors would be incredible.

Perhaps something like that exists! Let me know if that’s the case.