DXC shader compiler integration

Primary developer: @lukas.taparauskas (Core Kaunas / QoL team)
Slack: #devs-dxc-shader-compiler

TL;DR:

- Unity has used an old “FXC” HLSL compiler for years, but FXC is not getting any new GPU feature support, and has a lot of compilation performance issues. We’re adding a new “DXC” shader compiler as an option.
- **DXC works on DX12, Vulkan and Metal**. DX11 and GL/ES are not supported yet.
- Compile times on average are **4x faster** than the previous compiler.
- Compiled code size is 2x smaller on Vulkan, about the same on Metal, 3x larger on DX12.
- DXC is not 100% compatible with FXC with some HLSL language constructs. Some shaders might need small updates to make them compile.

User guide

Unity uses Microsoft’s “FXC” shader compiler (see manual) today. However it has several large issues:

- Microsoft stopped updating it years ago.
- FXC is limited to DX11 / shader model 5.0 feature set (no support for: ray-tracing shaders, wave/simd-group functions, barycentric coordinates, full FP16 support, etc.).
- Has shader compile time performance issues. Most notably: slow at inlining functions, and very slow at compiling anything that uses large arrays.
- Because FXC compiles into DX11 bytecode, translating that into Vulkan/Metal/OpenGL produces definitely “unreadable” shaders, and sometimes suboptimal shaders too.
- Using FXC on Mac/Linux is really cumbersome.

Microsoft has been developing a new DirectXShaderCompiler (DXC) as an open source HLSL compiler, with out of the box support for DX12 (DXIL) and Vulkan (SPIR-V) shaders. Targeting Metal is possible by going through SPIR-V + SPIRV-Cross.

We are **adding support for DXC as an optional shader compiler backend** in Unity. DXC primarily brings two benefits:

1. Can write shaders that use more modern GPU features (e.g. wave functions).
2. DXC often compiles code faster than FXC does; on many shaders we tried about 3-6 *times* faster.
3. Some shader constructs might result in faster shaders on non-D3D platforms (e.g. Metal or Vulkan).

However, do check out reasons **why DXC is not default yet** further in the doc.

How to enable DXC per shader or shader pass

DXC can either be enabled on a per shader pass basis by placing a **`#pragma use_dxc`** pragma into the `HLSLPROGRAM/ENDHLSL` or `CGPROGRAM/ENDCG` section.

It can also be enabled for all shader passes in a .shader file by placing the pragma into the `HLSLINCLUDE/ENDHLSL` or `CGINCLUDE/ENDCG` section.

You can also enable the pragma on a per graphics API basis by providing additional platform arguments (arguments similar to those used for `#pragma only_renderers`). For example, writing `#pragma use_dxc metal vulkan` will only enable DXC for the Metal and Vulkan graphics APIs, but not for D3D. If no arguments are provided, DXC will be enabled for all platforms that support it.

Note: if multiple `#pragma use_dxc` are placed into the same file, only the last one will be used. This can be useful when e.g. DXC is enabled for all supported platforms from the `HLSLINCLUDE` section with `#pragma use_dxc` with no arguments and then by placing a second DXC pragma into a particular shader pass (`HLSLPROGRAM` section) you can override the global DXC setting with different platforms for that pass.

How to enable DXC for compute shaders

The same **`#pragma use_dxc`** can also be used to enable DXC for compute shaders (.compute extension). Here the pragma can be placed on any empty line in the file. Like with regular shaders, here the pragma supports platform arguments and only the last pragma in the file is used.

How to make sure DXC is off

When working with DXC you might sometimes want to disable it either for a particular *.shader* file, a particular pass or even only for a particular API. This can be achieved with a **`#pragma never_use_dxc`**. This pragma also takes API arguments and it will make sure DXC is not used in the shader for those APIs regardless of if `#pragma use_dxc` is present in that shader snippet, if no API arguments are provided it will disable DXC for all APIs.

`#pragma never_use_dxc` can be placed in the same places where `#pragma use_dxc` can and similarly only the last instance of the pragma in a particular shader snippet will be respected.

Note: it does not matter if `#pragma never_use_dxc` is followed by `#pragma use_dxc`, these pragmas are parsed separately and only the last pragma of each in a particular shader snippet will be used.

#pragma require - enabling Shader Model 6.x features

The existing `#pragma require` has been updated with new arguments corresponding to some of the new features that DXC allows. The tentative list of these arguments is as follows (though it might need to be changed if a better mapping to hardware if found):

- **`WaveBasic`** - HLSL intrinsics corresponding to GL_KHR_shader_subgroup_basic[1]
- **`WaveVote`** - HLSL intrinsics corresponding to GL_KHR_shader_subgroup_vote[1]
- **`WaveBallot`** - HLSL intrinsics corresponding to GL_KHR_shader_subgroup_ballot[1]
- **`WaveMath`** - HLSL intrinsics corresponding to GL_KHR_shader_subgroup_arithmetic[1]
- **`WaveMultiPrefix`** - SM 6.5 wave intrinsics[2]
- **`QuadShuffle`** - HLSL intrinsics corresponding to GL_KHR_shader_subgroup_quad[1]
- **`Int64`** - 64 bit integer operations from shaders (without atomics)
- **`Native16Bit`** - use of `float16_t`, `int16_t`, `uint16_t` shader types for arithmetic operations (`half` doesn't get silently converted to 32bit float). Also allows use of 16 bit types in constant buffers and structured buffers. Available from Unity 2021.2a20.
- **`Barycentrics`** - SV_Barycentrics input semantic as well as GetAttributeAtVertex HLSL intrinsic[3]
- **`Int64BufferAtomics`** - 64-bit integer atomic operations on buffers that come standard with DirectX SM 6.6. Available from Unity 2022.1.0a16 for D3D12/Vulkan
- **Int64GroupsharedAtomics** - optional SM 6.6 64-bit integer atomics on groupshared variables. Available from Unity 2022.1.0a16 for D3D12/Vulkan

Adding any one of these to a shader/pass will implicitly enable DXC for that shader on DX, Vulkan and Metal platforms. Note that this can be overridden by `#pragma never_use_dxc` though the FXC compiler backend will likely not compile these if used.[4]

Usage of these features will adjust DX shader target model, Vulkan version/extensions and Metal version as needed.

(1) - Subset of this Vulkan instruction set which has corresponding HLSL and MSL intrinsics (see mapping of the new intrinsics between graphics APIs)
(2) - Includes WaveMatch and WaveMultiPrefix, these have no equivalents in Vulkan/Metal.
(3) - GetAttributeAtVertex currently causes a crash with the SPIR-V backend.
(4) - For FXC we use function overloading to emulate some intrinsics like *WaveReadLaneFirst* in HDRP shaders however these overloads are not allowed with DXC SPIR-V backend as they get interpreted as the real intrinsics and cause errors due to insufficient Vulkan version set during compilation (the version is only raised when the appropriate *#pragma require* is set)

#pragma target

With respect to new shader model targets which are available with DXC (i.e. SM6.x) we have decided to **NOT** expose new `#pragma target` arguments.

It has historically been a pain to maintain the `#pragma target` (see our crazy feature correspondence table for existing models between APIs) and we believe we have reached a point where this concept is no longer useful outside of D3D as features between modern graphics APIs don't have a linear correspondence and any of them can be either present or missing on a particular GPU model. Because of this we have exposed new features to shaders using `#pragma require`.

For these reasons #pragma target 5.0 will remain the largest pragma target we allow. And the SHADER_TARGET_AVAILABLE macro will remain defined to 50 when compiling with DXC.

Internal codebase only: Making DXC the default compiler

If someone is inclined to more thoroughly try out DXC it can be made the default compiler for supported APIs by changing one #define in our Unity codebase and recompile the editor. To do this you need to change `#define DXC_USAGE_MODE DXC_USAGE_MODE_OPT_IN` to `#define DXC_USAGE_MODE DXC_USAGE_MODE_OPT_OUT` in `Tools/UnityShaderCompiler/Compiler/CompilerDXC.h`. This will still allow you to disable DXC for particular shaders/APIs by using `#pragma never_use_dxc` as described in the dedicated section.

Status as of 2021.2 a10

Graphics API support

The following describes if/how our DXC backend is available on different APIs:

- **DX11** - running DXC shaders on DX11 does not work. In practice this means that the shaders will simply not run there, and the usual next subshader -> fallback -> pink error shader will kick in. This is because the DXIL bytecode is only accepted by DX12. **This is a big limitation as most of our users still primarily target DX11 and cannot benefit from DXC**. There are several ways we are thinking of how to approach this:
    - Separate compiled shader blobs for DX11 and DX12, use FXC for the former, DXC for the latter. Downside: we'd still be stuck with the slow compile times of FXC.
    - Try DXC -> SPIR-V -> SPIRV-Cross -> HLSL -> FXC -> DX11 DXBC. This sounds crazy, but maybe the last part where FXC is involved would be sufficiently "flattened"/"optimized" HLSL code already, and FXC compile time slowness would no longer be an issue.
    - Try writing a DX12 DXIL -> DX11 DXBC converter, for cases where it can be done (i.e. presuming the shader does not use any of the shader model 6.x features).
- **DX12** - **works**, with some caveats:
    - You can only compile DXC shaders for DX12 when using the Windows editor. This is because code signing and reflection required for DX12 shaders only exist in a native dxil.dll Windows DLL.
    - ~~Editor scene view wireframe does not work with DX12 shaders compiled with DXC~~ (fixed in 2021.2.0a19)
    - Early versions of Windows 10 / DX12 did not have DXIL support. So you can end up in a situation where DX12 generally works but does not accept DXC/DXIL shaders. This is because DXIL shipped only starting with Windows 10 version 1703 (2017 April).
- **Vulkan** - **works**, Under the hood, DXC emits SPIR-V, and we also do some resource remapping to match Unity runtime expectations.

    - The biggest caveat is that we currently require the `VK_EXT_scalar_block_layout` extension to maintain resource memory layout compatibility with DirectX (the support for this extension is growing but is not yet universal)
- **Metal** - **works** (tessellation shaders starting with 2021.2a20). Similar to Vulkan, DXC emits SPIR-V which we then convert to Metal Shading Language using SPIRV-Cross.
- **OpenGL & GLES** - Does not work; trying to use DXC on OpenGL will simply always fall back to the previous FXC+HLSLcc compiler. We are planning to look into the viability of using SPIRV-Cross to target OpenGL.

SRP support

- HDRP and URP version 11.x shaders have been fixed to work with DXC.
- ShaderGraph - At the moment, there is no way to enable DXC for ShaderGraph (as you cannot add shader pragmas there) also new functionality/intrinsics that are unique to DXC currently don't have corresponding graph nodes added.

DXC vs FXC

Compile time and shader size

We have measured shader compilation time and resulting data size on various (built-in, HDRP, URP, post-processing) shaders. **Here's the full spreadsheet.**

Compile times: DXC is generally significantly faster than the old backend e.g. it's **~3.2-6.8(1) times faster** when building the HDRP template depending on the backend.

Shader sizes (after compression):

- DX12 - HDRP output is ~1.7 times larger (though it's only 0.75 of FXC's output before compression, meaning that DXIL is just less compressible than DXBC). URP output is 4 times larger with DXC...
- Metal - output size is similar, though a bit larger for the URP template (1.35x larger)
- Vulkan - output size is about 2x smaller than the old compiler.

(1) - Timings for the top ~20 longest compiling shaders in the HDRP project template.

Shader differences/gotchas

The DXC compiler is not 100% backwards compatible with HLSL syntax accepted by FXC. The majority of HLSL constructs compile and work exactly as before, but not all of them. Some existing shaders might need to be edited to work with DXC.

Official DXC documentation lists some of the differences between FXC and DXC: **Porting shaders from FXC to DXC**

Here's a list of the main differences we found:

1. Requires more strict use of HLSL output semantics. For example the pixel shader's output has to be `SV_Target` and not `COLOR`. Also the vertex/domain/geometry stages have to have an `SV_Position` output semantic (the `POSITION` semantic will no longer work) to produce visuals.
2. Does not allow silent vector type truncation: `float a = uv.xy`
3. Float `VFACE` pixel shader input semantic is no longer supported; need to use bool `SV_IsFrontFace`.
4. Does not allow *writes* to constant buffer globals: where a shader, instead of using a temporary variable, writes into a global variable and later on reads the modified value. For example Unity's built-in instanced particle shaders rely on this; currently we disable DXC for them.
5. Any usage of writable resources ("UAVs" like `RWTexture`, `RWBuffer`, `RWStructuredBuffer` etc.) in a pixel shader requires to indicate their binding index using "`: register(uN)`" annotation, with N matching the binding index you use in C# SetRandomWriteTarget. This was technically required with FXC too, except in many cases it was automatically assigning the "most logical" binding index, which was "number of pixel shader color outputs + 1". DXC no longer does that.
6. Using `Texture2DMS<type>` does not work, you need to use `Texture2DMS<type, samples>.`
7. Does not support resources passed as shader function entry point arguments, e.g. `float4 frag(StructuredBuffer<float4> buffer) : SV_Target` will not work.
8. Unused variables are not stripped away from the global constant buffers when targeting Metal/Vulkan/OpenGL.
9. `atan2(0,0)` with DXC can result in PI/2, but results in NaN with FXC; both are correct since it's "undefined".
10. `round()` gotcha: FXC used to round constant 0.5 to 1.0, but a value 0.5 coming from a variable down to 0.0. DXC consistently rounds both to 0.0. However, for example Metal rounds 0.5 to either 0.0 or 1.0 depending on the device.
11. The FXC+HLSLcc compiler stack supported sub-pass inputs (Vulkan) / framebuffer fetch (Metal) as `inout` fragment arguments. This is not supported on DXC; the only way to do sub-pass inputs is via `UNITY_DECLARE_FRAMEBUFFER_INPUT_<type>, UNITY_READ_FRAMEBUFFER_INPUT_<type>`.
12. DXC does not support the HLSL `CalculateLevelOfDetail` function on Metal yet.

Why is DXC not the default compiler?

The DXC backend still has some limitations which prevent us from making it default. These are (roughly in the order from most important to least important):

1. **Does not work for DX11** - the new DXIL shader bytecode format is only accepted by DX12 (DX11 only accepts the FXC generated DXBC bytecode, while DX12 accepts both DXBC and DXIL). And due to the fact that there is currently no separation between DX11 and DX12 shader data (as it used to be identical with FXC). The latter means that we cannot just use DXC for DX12 and FXC for DX11 until their bytecode is split up.

2. **Vulkan** requires VK_EXT_scalar_block_layout device extension, which is not yet universally supported. We require this for DXC as we rely on forcing DX memory layout with `-fvk-use-dx-layout` compiler flag on Vulkan/Metal to maintain StructuredBuffer memory layout compatibility between APIs (the old backend would achieve the same by sidestepping the issue and making all structured buffers contain only UINT for Vulkan/Metal and relying on bitcasts to and from memory to achieve layout compatibility between APIs).
3. DXC Vulkan support for mobile is not yet sufficiently tested
4. Reduced precision scalar/vector/sampler types are not yet supported and thus DXC backends always run at full precision (e.g. Sampler2D_half, half and half4, etc. will have full float precision)
5. Cannot compile for DX12 from Mac/Linux.
6. ShaderGraph can't use DXC since it does not expose "pragma require".
7. ~~Metal tessellation is not implemented yet (done in 2021.2a20)~~.
8. DXC cannot be used for instanced particle shaders as they rely on legacy HLSL syntax of *writing* to constant buffers (it's just syntactic sugar, the buffers are not actually written to) so `#pragma never_use_dxc` is required to not break those.
9. DXC still has quite a few bugs which affect our shader codebase; see the list of open DXC issues below.
10. ~~iOS using `base_instance` will currently not compile with DXC (fixed in 2021.2a20)~~

Our open upstream DXC issues

1. [SPIR-V] SV_Position can no longer be declared to be min10float4, min16float4, float16_t4 or half4 after fix to issue 3736 (works with DXIL)
2. GLSL: High precision ops involving constants get implicitly demoted to mediump during translation
3. [SPIR-V] Resources declared in cbuffer scope result in validation errors instead of being moved out (works with FXC)
4. [SPIR-V] Trying to "write" to uniforms using the -Gec backwards compatibility mode produces invalid codegen (works with DXIL backend)
5. [SPIR-V] Using bool types or their arrays in Hull and Domain stage IO fails compilation (works with DXIL backend)
6. [SPIR-V] Patch constant function does not support out parameters (works with DXIL backend)
7. No warning/error when SV_Position semantic is missing from Vert/Domain/Geom output and POSITION is used instead
8. [SPIR-V] Omitting outputtopology Hull entry point attribute does not get caught (is caught by DXIL backend)
9. [SPIR-V] No error when Hull and patchconstantfunc input point counts don't match
10. [SPIR-V] Reapeated SV_DomainLocation semantics don't produce an error and can crash the compiler (works with the DXIL backend)

11. [SPIR-V] Using very large vert/hull/domain IO produce a silent compiler crash (works with DXIL backend)
12. [SPIR-V] Zero as control point template argument for InputPatch Hull input fails validation and asks to report a bug
13. [SPIR-V] [outputcontrolpoints(0)] or missing outputcontrolpoints Hull stage attribute crash the compiler (works with DXIL backend)
14. Some compile time const expressions can't be used to specify const array lengths (works with FXC)
15. DXC's preprocessor allows missing macro arguments and produces invalid output with -flegacy-macro-expansion (different from FXC)
16. Relative includes are handled differently from FXC - directories of previously included source files get searched
17. Using Texture2D<bool*>.Load results in validation errors with both DXIL and SPIR-V backends (not with FXC)
18. [SPIR-V] Arrays of empty structs in cbuffers produce validation errors when optimizations are disabled (asks to file a bug)
19. [SPIR-V] Silent compiler crash when using GetAttributeAtVertex
20. [SPIR-V] Crash when result of dot4add_u8packed or dot4add_i8packed is operated on by a literal
21. Plans to support DXIL reflection on non-windows platforms?
22. Array in cbuffer reported as unused by reflection
23. "error: reference to 'i' is ambiguous" where loop scoped variable should take priority. Other compilers have no issues

To get fixes for these issues into Unity after they're done it's required to update our fork of DXC by merging in the fixed upstream master and then adding the updated DXC library to the Unity repo for all the platforms and adding/fixing up any unit tests.

Closed upstream DXC/SPIRV-Cross issues

24. [SPIR-V] Using SV_Target and SV_Target0 in the same PS fails validation instead of printing an error
25. [SPIR-V] Structs with doubles violate 8B alignment requirement and fail validation when using dx layout
26. [SPIR-V] SV_ClipDistance variable not allowed to be of array type (works with DXIL backend and FXC)
27. ~~[SPIR-V] Matrices in constant buffers are unnecessarily padded when using DirectX memory layout~~
28. [SPIR-V] Compiler does not complain about writing to read only StructuredBuffer<> resources (DXIL backend does)
29. [SPIR-V] Writing to a read only Buffer<> resource produces validation errors (asks to report a bug)
30. [SPIR-V] Declaring SV_POSITION not as float4 produces validation errors and asks to report a bug
31. ~~[SPIR-V] Using RWTexture2D or RWBuffer with min precision types crashes the compiler~~
32. ~~[SPIR-V] Reading from a RWTexture2D with a min precision template argument produces validation errors (good with DXIL backend or FXC)~~

33. ~~MSL: gl_Position is undetectable in tessellation stage IO using the SPIRV-Cross reflection API~~
34. ~~Using legacy VFACE semantic does not work but compiler error is not printed (affects both DXIL and SPIR-V backends)~~
35. Reflection info says Texture2DMS<float4> is not multisampled unless explicit sample count is provided (different from FXC and SPIR-V backend)
36. ~~Silent crash or validation error when SV_InnerCoverage semantic is assigned to a float parameter instead of uint~~
37. ~~[DXIL] Decorating CS float argument with SV_DispatchThreadID semantic crashes the compiler (float2, float3 and float4 works)~~
38. ~~[SPIR-V] Assigning SV_DispatchThreadID to a float argument causes validation errors (does not happen with DXIL backend)~~
39. ~~MSL: No way to set msl_options.ios_support_base_vertex_instance from the C API~~
40. ~~[SPIR-V] Forward declaring a function inside another function causes a compiler crash~~
41. ~~[SPIR-V] Reading or writing a struct's member from a derived struct's method causes SPIR-V codegen errors~~
42. Can't have resources in entry point parameter list (works with FXC) **(Rejected)**
43. ~~[SPIR-V] Crash on assertion when compiling shaders containing SubpassInputs and global struct instances with resource only members~~
44. ~~[SPIR-V] Global struct variables containing only resources fail validation with '-O0' optimization level~~
45. ~~Optimized DXIL doesn't work correctly when memory barriers are placed in between identical conditions~~
46. ~~Invalid float4 to float assignment in codegen when using arrays of scalars (int[], float[]) in constant buffer~~
47. ~~MSL: Why do names starting with '_' followed by a digit e.g. "_1PixelColor" get changed to _m<num>?~~
48. ~~MSL: SubgroupSize becomes [[thread_execution_width]] instead of [[threads_per_simdgroup]] in compute shaders~~
49. ~~[SPIR-V] Silent crash when using any WaveMultiPrefix* intrinsic with first arg of type uint*~~
50. ~~MSL: RWBuffer<uint> is translated into a texture2d and buffer of atomic_int even though SPIRV-Cross can tell the original type~~
51. ~~[MSL] SPIRV-Cross assumes std140 layout and fails with SPIR-V generated with DXC's -fvk-use-dx-layout option~~
52. ~~[Question] How to tell which UBO members are actually used with the reflection API?~~
53. ~~[SPIR-V] global struct variables containing only resources get placed in uniform buffer~~

NB: Not all of these have been integrated into our DXC/SPIRV-Cross fork repositories yet so these issues are still present when using DXC with Unity (the ones already fixed in Unity are ~~striked out~~)