GPU Web 2020-10-19 VF2F Day 1

Chair: Corentin / Dean
Scribe: Ken / Austin
Location: Google Meet

Doc for Day 2

Doc for Day 3

Tentative agenda

- Status updates!
    -
- Method of ensuring GPUShaderModules can contain MTLLibraries #1064 (Myles)
- Capability-querying APIs for GPUAdapter
    - Make GPUAdapter.extensions a setlike interface #1098 (Kai)
    - Add a limit-querying API for GPUAdapter #1100 (Kai)
- D3D12 does not support SRC_COLOR used in SrcBlendAlpha slot #65 (Dzmitry?)
- Reconsider the name `OUTPUT_ATTACHMENT` #1153 (Corentin)
- Multiqueue (Dzmitry, Corentin)
    - Multi-Queue Investigation #1065
    - Strawman Multi-Queue Proposal #1066
    - Multi-queue proposal with explicit transfers #1073
- Proposal for importing Web platform images in WebGPU #1154
- PR burndown
    - Add aspect back to GPUTextureCopyView #873 (Austin)
    - createBindGroup: Require a superset of the layout's bindings #1061 (Corentin)
    - Add filtered texture and sampler binding types #1076 (Dzmitry)
    - GPUColor: remove sequence overload from the union #1079 (Kai)
- WGSL
    - Define the interface of an entry point in a WGSL program (#774)
    - Restrictions on function parameters (#1139)
    - shader programming model: permitted memory orders on control barriers (#232)
    - Issue with type converting module scoped variables (#1104)
    - Is Input/Output access one-way? (#1113)
    - Invariant qualifier (#893)
    - Method of ensuring GPUShaderModules can contain MTLLibraries (#1064)
    - Reorder expression sections (#1136)
    - Introduce operator precedence table (#1111)

    - Placement of read_only attribute (#1159)
    - what is the initial value of a workgroup variable? (#1137)
    - Entry point taking in/out as parameters (#1155)
    - Buffer indices should be unsigned (#1135)
    - Remove return requirement (#1156)
- Agenda for next meeting

Attendance

- Apple
    - Dean Jackson
    - Myles C. Maxfield
    - Robin Morisset
- Google
    - Austin Eng
    - Brandon Jones
    - Corentin Wallez
    - Dan Sinclair
    - David Neto
    - James Darpinian
    - John Kessenich
    - Kai Ninomiya
    - Ken Russell
    - Shrek Shao
    - Ryan Harrisson
    - Sarah Mashayekhi
- Intel
    - Yunchao He
- Kings Distributed Systems
    - Daniel Desjardins
    - Dominic Cerisano
    - Hamada Gasmallah
- Microsoft
    - Rafael Cintron
    - Natasha Lee
    - Sebastien Vandenberghe (first hour)
    - Alexis Vaginay (first hour)
    - Thomas Lucchini (first hour)
- Mozilla
    - Dzmitry Malyshau
    - Jeff Gilbert
- W3C
    - Francois Daoust

- Henrik Edstrom
- Michael Shannon
- Mehmet Oguz Derin
- Pelle Johnsen
- Timo de Kort

Status updates

- Apple
    - DJ: all 3 of us busy with other projects
    - Planning work for the next few months
    - Hopefully will have some time to do more WebGPU work in the coming weeks
    - RM: nothing more to add
- Google:
    - CW: API side: just sent status update with new APIs that Chrome now supports, and breaking changes
    - Highlights: experimental WGSL support (some crashes / wrong results, but testable, and will get better over time); demo later
    - Linux support! Should be coming in Chrome Dev Channel very soon. Might be slightly less robust than other OSs (our Vulkan implementation has had less attention overall), but working for the most part. Please report issues./
    - New APIs. writeTexture. Timestamp queries. Depth bias. Additional formats like RGB9_E5. Progress toward completing API surface.
    - MM: for experimental WGSL support are you going through SPIR-V on all platforms?
        - CW: Chrome does, in all cases right now. WSGL -> SPIR-V Cross -> platform shading languages. Dan Sinclair will give more updates later.
    - CW: <shows Austin's samples working in WGSL in Chromium Canary>
    - DS: slides (mirror)
        - Everything in spec parses. Missing lexical scoping. Landed struct name parsing. Still parsing “type struct” for now until downstream users update.
        - Lots of unit tests. Adding CTS validation tests; passing about half. Tint being updated before CTS tests are, or haven’t implemented that validation yet.
        - In-progress SPIR-V parser. Lagging what WGSL can support, like textures. On our roadmap; it’s just time.
        - Output: have SPIR-V, MSL, HLSL backends. SPIR-V most complete. Others don’t have e.g. textures yet. Some intrinsics like outer product are missing.
        - WGSL emission is complete. For the SPIR-V pass, so you can do SPIR-V to WGSL.

        - Chrome can accept WGSL. Always generates SPIR-V. Tint doesn't have reflection information which Dawn needs. Currently building reflection info into Tint. Once we have reflection, will switch Dawn over to using Tint.
        - Various validations, etc.
        - MM: on slide 2 last section says you're working on SPIR-V parser? Why do you need that?
        - DS: Tint can take SPIR-V and turn it into WGSL.
        - MM: and that's not web-exposed?
        - DS: no.
        - MM: on last slide you said you're in process of moving Chrome to using Tint to generate HLSL and MSL?
        - DS: yes. Tint needs to generate reflection information first. Once we have it will use it instead of SPIR-V cross.
        - MM: when that happens do you have long-term plans about skipping intermediate SPIR-V step?
        - DS: once we have that reflection information we'll switch to our backend generation.
        - MM: and that skips SPIR-V?
        - DS: yes, we can generate those languages like MSL and HLSL directly.
        - MM: cool.
    - Put some effort into writing spec text for almost all entry points (thanks Brandon!)
    - Pushed a lot of tests into the CTS (with other contributors) for validation and state tracking, to minimize interop issues.
    - DC: question about transpiling of WGSL to other languages: will GLSL be supported by this mechanism?
        - DS: will not be done in browser. You'll have to convert GLSL to SPIR-V and SPIR-V to WGSL. Browser will just accept WGSL.
        - Can use Naga or Tint to do this.
        - Naga has a GLSL frontend; Tint doesn't.
        - PJ: does Tint compile to WebAssembly?
        - DS: haven't tried, but no reason it wouldn't. Happy to receive build fixes.
- Intel:
    - slides presented by Yunchao (mirror)
    - DC: has query API been committed?
        - YH: yes.
        - DC: working on CTS tests for this?
        - YH: yes.
        - DC: in TensorFlow, your benchmarking between WebGL and WebGPU - did you use those query APIs?
        - YH: the reason we introduced query API into Dawn is for TensorFlow WebGPU backend, yes.
        - DC: WebGPU also has an OpenGL backend, if I understand. Did you also implement timestamp query for OpenGL?

        - YH: no. For WebGL we have a simple timer query API.
        - DC: I'm curious to know the performance of the OpenGL backend.
        - YH: we didn't do that. WebGL's query API isn't as accurate as WebGPU's.
        - DC: so the benchmark has to be taken with a grain of salt then?
        - YH: yes.
        - CW: two caveats about timestamp queries: (1) they currently return results in ticks, not milliseconds; and (2) we don't implement the zeroing of queries that we discussed as being necessary.
    - DM: issues in WebGPU TF porting: robustBufferAccess pass bringing big performance degradation? Thought that's just a feature you enable?
        - YH: it's just a feature, yes. When we add it, we need to check boundaries, and add clamp operations over every buffer access, right now. We see significant performance degradation from this.
        - DS: it's confusing. One thing is the Vulkan robust buffer access flag. Then there's the SPIR-V robust buffer access op. That adds clamp operations and is the cause of the 30% slowdown.
        - CW: that's something WGSL requires, that we clamp all array accesses. In WGSL it's just a min() operation, but we'll see what the cost of that is. Some drivers do really well with it, but some don't. Need to see if the performance issue can be worked around.
        - YH: my concern is that for TF/WebGPU workloads, the GPU isn't the bottleneck sometimes. So if you don't see a perf regression, that might not be true.
        - DM: if you don't use the feature you shouldn't need the pass?
        - DN: that's true for buffers. For GPU private memory, workgroup memory, for defined behavior you still need to clamp those indices. If you are running on a platform with the flag in the vulkan driver then you can rely on that.
        - DS: doesn't this change with Tint's change away from SPIR-V cross? Then when we turn on the Vulkan robust buffer access path, we won't need the clamping?
        - CW: thought for portability we decided to always clamp? Or am I misremembering?
        - DS: WGSL will always clamp an out-of-bounds array access to the last array element.
        - MM: what's the mechanism to ensure there is a last element?
        - DM: The minimum buffer size that we have.
        - MM: Error at which API call?
        - DM: If provided at bind group layout, it would be at bind group creation. Otherwise at draw time
        - DS: it's not an error, we're just clamping.
        - DM: respecting the min buffer size is an error.

        - MM: if you have an array of structures, you could pass a buffer smaller than the size of one of those struct members, and you couldn't clamp. Spec must also have affordances so every inside a buffer holds at least 1 item.
        - DN: yes.
    - DM: You said RBA made TF.js slower compared to WebGL is that correct?
        - YH: Actually slower than WebGPU without RBA, not comparing to WebGL
        - DM: but the WebGPU backend is slower than WebGL right now? Why is that?
        - YH: depends on the workload, right now for posenet/resnet the input is a video/camera, so for WebGL the video can go through the video_texture extension so it can be faster than copyImageBitmapToTexture.
        - YH: If input data is on the CPU (say tensor, data in ArrayBuffer), or we're able to keep everything on the GPU then WebGPU can be faster than WebGL.
        - DM: I see. Was wondering what WebGL would be doing differently with robust buffer access compared to WebGPU. Sounds like nothing.
- Microsoft (Edge):
    - RC: implemented context loss in Dawn and rest of Chromium; partners think this is important.
    - RC: couple of people working on SPIR-V->DXIL converter, so when we do more DX12-only APIs we can more easily use them. (Currently using fxc.)
    - DM: are you working on SPIR-V->DXIL outside Tint infrastructure?
    - RC: yes - we could use part of Mesa (have some SPIR-V->DXIL devs working on this for OpenCL) - considering adding graphics stuff to this code.
    - MM: if you take this path you'll go WGSL -> SPIR-V -> DXIL? Won't go directly?
    - RC: we will use Tint inside Edge. We'll parse WGSL and convert to SPIR-V.
    - MM: so you'll add another stage after Tint?
    - RC: yes.
    - DS: Tint doesn't do any optimization, where DXIL requires it.
    - MM: what's the reason to not upstream this DXIL work?
    - RC: I never said we wouldn't upstream it.
    - DS: upstream where?
    - RC: to Mesa and Chromium.
- Microsoft (Babylon.js):
    - TL (Thomas Lucchini from Babylon.js team): slides (mirror)
    - DC: will you expose WebGPU directly to end users?
    - SV: we're aiming to abstract it. Right now we use bgfx. Want to reduce number of abstraction layers.
    - DC: There's a project spun out from WebGPU-NAPI, which is WebIDL-NAPI; want to use WebGPU as the pilot project.
    - SV: it's long-term for us - after the first release.

    - CW: You say there's perf issues copying from Canvas/video. What path are you using to do that?
    - SV: we're not doing crazy stuff anymore. createImageBitmap. As soon as source is canvas / video, it gets really slow. On Alexey's machine, to do our GUI layer, we do the UI in a 2D canvas. When we do this in WebGL we reach 60 FPS. In WebGPU when we do this for a 2800-width canvas, looks like there's lots of overwriting. If videos are small, perf is OK, but larger videos are really slow. Got more and more experiences internally mostly with professional tools where video's an input. Also video processing with canvas. Need really fast path for video textures. It's our main bottleneck. Not as lean as we could have it even with WebGL. Can really feel the impact on low-end Chromebooks, etc.
    - DM: what do you use for shaders today?
        - SV: SPIR-V today. What you were discussing before: (Tint, Naga) - would be mandatory for us to ship to our installed base. We'd like for end users to autodetect WebGPU support and convert their GLSL shaders automatically. Would need 1 MB additional library, but would give them a migration path until they can have only WGSL in their application.
        - DM: currently writing in GLSL and converting to SPIR-V at some point?
        - SV: yes. Not a fast path but is friendly to the end user. Plan is to migrate our shaders to have both WebGL and WGSL versions, code splitted. But want to still support GLSL for end users' shaders.
        - MM: during this gradual rollout, you mentioned one of the steps is people write their shaders in GLSL and include an external dependency for the WebGPU engine. Which GLSL compiler do you aim to use?
        - SV: we're discussing that now. Probably either Naga or Tint. Then use WASM version of Tint to go from GLSL - > SPIRV. Don't know what the best choice would be for our end users. Would probably ask Corentin, Rafael, Kai, etc. We expect this path to work for other users too, not just us.
- Mozilla:
    - DM: slides (mirror)
    - DM: lots of updates, writeBuffer/writeTexture, new mapping API
    - Not currently running Google's samples because they use implicit bind group layout which we don't have yet. wgpu has BindGroupLayout support though.
    - CW: your textured cube demo, is that JavaScript or a Wasm-compiled Rust demo?
        - DM: that one's JavaScript.
- Spec Editors:
    - DM: small update on spec editing: slides (mirror)
    - DM: a lot of spec work done since last F2F in June. Mostly done by editors as well as Brandon, who's basically acting as editor nowadays and doing a lot of good things.
        - Malicious use considerations, internal usage flags, … (see slides)

        - Need to continue the pace we've had over the past 3 months. Need to describe render pipelines completely, how WGSL integrates, computation model, etc.
        - We invite everyone to help us specify this!
    - MM: what % of all the code you described is written in Rust?
        - DM: on the Gecko side, most of the code's plumbing / boilerplate. All the logic's written in Rust.
        - JG: the vast majority.
        - DM: we do still use SPIRV-Cross. Planning to deprecate it in favor of Naga. Will be a big migration away from C++.
        - MM: will Naga use SPIRV?
        - DM: Naga has an internal representation that we define. Probably similar to Tint's. SPIRV is just one of its frontends / backends. We skip it or compile it out if it's not involved.
        - MM: all of the WGSL compilers that will end up in browsers will have direct to MSL / HLSL paths that skip SPIR-V?
        - CW: assuming Safari does it as well, yes.
        - MM: yes, we would not go through SPIR-V.
        -
    -
    -
- Others?

Agenda for the rest of the F2F

- WebGL will meet for the first hour of the WebGPU on Thursday morning. If not too many topics, maybe we can remove that hour from the WebGPU schedule?
- CW: maybe today should be WGSL discussions; Wednesday entirely API discussions; Thursday cancel the first hour; then probably WGSL and whatever remains. Thoughts?
- DJ: sounds good.

WGSL

- Define the interface of an entry point in a WGSL program (#774)
- Restrictions on function parameters (#1139)
    - DS: Passing textures into functions?
    - DN: you can do it in SPIR-V. There's one case you can't do in GLSL - formatted read-only storage images or write-only storage images. But SPIR-V does allow you to do that. Currently as created the … is part of the … in WGSL, so as long as the type parameters line up it should work.
    - MM: if I have a function taking parameter of texture, can I do that?

- DS: you pass a pointer to the variable containing the texture. Like a reference parameters.
- MM: does that mean I need a load operation?
- DN: yes, correct.
- MM: that approach works in Metal, no problem. Can we do the same thing in HLSL? If so then we should allow it.
- DN: good question.
- DJ: could we agree to accept texture parameters, waiting on Greg, Rafael or Damyan to say no?
- DM: will we allow texture parameters under pointers, or with pointers?
- DN: I'd like them with pointers
- MM: opens them up to whole bunch of additional uses. Now that this seems legal, would like to re-discuss design. Entry point could receive all resources, then you could pass them around.
- DN: other bit of issue: you can't pass pointers to other storage classes. Can't pass base pointer to buffer, UBO, etc. Some extensions let you do it just for storage buffers. Bunch of different restrictions that paint you into a corner.
- MM: if you wanted to do the thing I described, then only textures could be parameters to entry points?
- DN: you couldn't do any of the buffers. Could do samplers and textures.
- MM: OK. So given that we have bind groups we prob don't want to have some members of bind groups in one place and some in others. So all resources should be globals. In SPIR-V you can make a function that's generic not over type, but over which texture it samples from?
- DN:yes.
- MM: but not which buffer it reads from?
- DN: yes.
- DJ: is there a benefit for non-entry-point functions to have texture parameters?
- DN: people in the community tell me the functionality's been around since 2004 and is something they'd like to do.
- MM: if HLSL can do it, we should have it, and vice versa.
- DN: sounds like a pragmatic and good direction.
- DJ: resolution: functions can have texture parameters, but not entry point functions.
- DM: I find it slightly unnecessary to have pointer to texture. Texture is "handle" storage type. No reason a handle couldn't be passed around.
- DN: can i copy it from one place to another? What are the value semantics for this thing? Gets confusing.
- MM: think this Q isn't very important. Caller doesn't have to add any code if parameter's a pointer or not, and neither does callee. We're talking about: should you have to write ptr<>?
- DN: yes.

    - DM: I didn't realize WGSL would have implicit loads for pointers. Why would we do that?
    - DN: because the signature of the texture built-in functions is that it takes an image, so you generate a load there.
    - DM: it's part of the implicit type conversions?
    - KN: sounds like it'd change the signature of the builtin function, not add an implicit load.
    - DN: depends on whether you add the … to the function or the texture itself.
    - MM: right. The caller only calls a builtin, doesn't matter how they do it.
    - DN: later, if we have arrays of textures & want to choose which of those elements gets passed into helper function, then do computation in caller, but function only gets the single element. Another argument for staying in pointer domain as long as possible.
    - MM: in one world we have pointers to texture object, passed by value. Or, we pass by pointer, have reference to texture, pass-by-reference semantics. Think these are identical.
    - DN: ...yeessss…
    - MM: wish we could rename pointers to references. ref instead of ptr?
    - DM: Myles said not a good idea to pass textures as entrypoint args - but what about inputs and outputs?
    - DJ: are we putting ptr<> in the function signature or not for textures?
    - DN: I think we are.
    - MM: I'm slightly leaning toward no, because it's more typing.
    - DM: I think we're not blocked on that, but on Microsoft's getting back to us on this topic.
    - RC: we'll take a look.
    - RC: I don't know the answer - will have to ask.
    - KN: given history of HLSL this could be one of those things you can do in practice, but only because of specific optimizations in fxc.
- shader programming model: permitted memory orders on control barriers (#232)
    - DN: we don't have acquire/release semantics on atomics. Point of control barriers: make sure stores from workgroup-mates become visible to other workgroup-mates.
    - DN: believe we only have relaxed atomics.
    - RM: are you saying we don't have release/acquire at all? Thought we didn't have either one alone.
    - DN: for control barriers, but not for atomics.
    - RM: ah yes, I agree with that.
    - MM: what is the question at hand?
    - DN: what memory orders does a control barrier apply? What loads/stores can't go across the barrier?
    - JG: there's a note saying we decided to not add memory barriers for MVP.

    - RM: I think that's a different thing. Do we need to make a decision or someone just has to write text for it?
    - DN: think this is part of that larger project. Don't think it's deep in its own sense.
    - MM: seems like there's nothing to decide, just figure out what's possible and not.
    - RM: I agree.
    - RM: I started looking at this over a year ago. Can try to find what I wrote and add it as a comment to the issue.
    - DJ: so we're waiting for Robin to add these notes, and discuss again?
    - JG: think we agreed someone just needs to write more spec text.
    - MM: maybe we need an umbrella issue tracking the relevant pieces of the memory model. Is that too much process?
    - DJ: I think we give this task to Robin.
    - RM: yes, I'll do it.
- Issue with type converting module scoped variables (#1104)
    - DS: easiest thing: disallow type conversions at module scope. Only valid at function scope. Can't do f32<1> at module scope, only in a function.
    - MM: seems funny and arcane. Compiler knows how to do this.
    - JG: requires well-defined idea of what's constexpr.
    - DS: if it were an identifier, couldn't do it at function scope anyway.
    - MM: think we already have well-defined idea of module scope definitions.
    - DN: we have the constructor for composites. Overloaded the syntax to also do value conversions. That's where this cropped up.
    - MM: think value conversions not much harder than what we already had, so seems natural to support this.
    - DN: usually the problem is, make sure all compilers do conversions exactly the same way. Extra work. Close to overflowing number of bits in mantissa, have to make sure you compile the right way.
    - RM: adding more things to constexpr can be added later in the language. First iteration in C++ was basic, and now the compiler can handle many more things. Think it's OK to defer the hard part
    - JG: doesn't seem hard
    - KR: Were a lot of discussion in WebGL about exactly what kind of constant propagation you could expect or require the compiler to do. The compiler could not always figure it out and we punted on requiring compilers to do a certain amount of optimization. Seems to be a similar discussion about how much analysis a compiler will do to prove an expression constant.
    - DN: could see the line being drawn at simple value conversions but not variable arithmetic. Valuable for user without lots of additional complexity.
    - MM: today in WGSL can I make a global float4 where one component is 3+4?
    - DS: no.
    - KR: I think it's a good decision to keep it simple and work on expanding the capabilities later.
    - JG: so this is resolved "nice to have?"

    - DS: we have to put in spec text to make sure we disallow this for now, and revisit it after MVP.
    - MM: in this example we'd ask authors to write vec3<float32<1.0, 2.0, 3.0>>?
    - JG: Could we introduce the idea of constant expression even if it's very simple right now?
    - MM: think we'll have to have some description of what is allowed on right-hand-side.
    - JG: think I'm proposing we should actually call it constexpr.
    - DN: There is something like that in the grammar.. but it's not connected ..
    - JG: make it so. Who'll do it?
    - MM: we can add it to the column in the project.
    - DJ: as long as it's noted in the issue and moved to Needs-Specificaiton, our army of volunteers will handle it.
- Is Input/Output access one-way? (#1113)
    - DM: think there is consensus there. Outputs also readable; think people accumulate them. Inputs only readable, outputs are read-write.
    - MM: if we made inputs arguments to the entry point, this problem would just go away, because you couldn't write to them (or, at least, would be the same as writing to the argument of any function.)
    - DN: parameters in C are local variables. Doesn't seem to me related to be the parameters of the function. Usually they're considered const.
    - MM: in a world where inputs were function params, behavior this question's asking about would fall out naturally.
    - DN: still disagree but think we don't need to litigate this.
    - DJ: we agree with this issue - accepting the last comment from David 21 days ago.
    - MM: will we mark input variables instead of "var" using "let"?
    - DN: no. "var" implies that you have storage somewhere ... pointer means the storage is somewhere else.
    - MM: do we have 2 definitions of const-ness?
    - DN: there will be a day where you have a volatile input which you can't write but must read multiple times.
    - MM: so a third definition of const-ness?
    - DN: depends on how you look at it. Input variable is storage that you as a shader can not modify. Right now, that storage's initialized before shader invocation begins. In future with raytracing, etc. might also have a variable you don't modify as shader, but outside agent might modify it.
    - DS: Myles, answer your question is yes: var, const, and var with storage classes. Input storage class => read-only.
    - MM: think this isn't worth arguing about. Programmer doesn't care whether they're storage or not. They just can't write to it. We can move on.
- Invariant qualifier (#893)

- DJ: last time we discussed this was in September. Had to look into Metal. Doesn't show up until Metal 2.2, which was macOS 10.15.
- MM: trying to figure out whether invariant can be in core. Seems it can't be. Enough devices / OSs wouldn't support it, where browsers want to ship WebGPU.
- JG: seems unbelievable to me but isn't invariant fairly core in GLSL? Is it just ignored?
- MM: it isn't ignored, but OpenGL and Metal are different APIs with different features sets.
- JG: so there are Metal-only devices that don't support invariant?
- MM: no, just devices where the Metal impl doesn't support it.
- DJ: there are some devices where OpenGL is implemented on Metal, and I suspect they implement invariant with something not exposed in the Metal API.
- JG: OK, just a harsh reality of which devices don't support this in Metal.
- MM: what's the story on HLSL and Vulkan?
- JG: supported and required on all of them. This is 1990s era functionality, which is why it surprised me it wasn't in Metal core.
- DM: seems silly that it's only blocked by variant of MSL. Lot of GPUs should support it. If we produce an error would this restriction be lifted?
- MM: if you produce an error all bets are off.
- DM: even Safari won't produce an error?
- MM: I can't comment on future releases but Safari's not planning to.
- DJ: we won't ship WebGPU on 10.15 or below, or iOS 13 or below.
- JG: but there are still devices that are above those versions that don't have this functionality?
- MM: no, the hardware supports it.
- KN: the hardware supports it when it supports OpenGL.
- JG: sounds like you won't support the device configurations where this would be problematic.
- MM: we don't plan to ship WebGPU on those configs.
- JG: so this is mainly an issue with Chrome, Firefox, or Edge.
- DJ: we don't worry about whether this will be in core or not.
- JG: do we have a firm idea of how many devices we'd forego?
- DJ: it's not necessarily devices - it's devices running a particular OS.
- MM: when we say "devices" we're talking about Macs specifically?
- JG: yes.
- DJ: Did we confirm all this with the Metal team?
- MM: yes.
- JG: my instinct would be to ask people to update their OS to get WebGPU. Maybe this is Mozilla / Google/ MSFT determining their tolerances. Think we should require invariant support.
- MM: that would be fantastic from our perspective. Up to the others to determine.
- JG: we'll schedule an offline chat.

    - DJ: this is just deciding core vs. extension? If others agree, they can decide what they prefer. Fine with Apple.
    - MM: possible to do that before this data before the remaining days of the VF2F are over?
    - DJ: depends on several factors.
- (discussion about DXIL emission from Tint / Naga)
- Method of ensuring GPUShaderModules can contain MTLLibraries (#1064)
    - MM: overall question: in WebGPU, if you want to use a shader, do you have to make two function calls? Turn source into ShaderModule (this takes no arguments today), and then pull out specific entry points from ShaderModule and turn it into pipeline. Second step takes a bunch of arguments.
    - MM: desire here: in Metal the first step can't do anything. All it can do is retain the string. Think true for HLSL as well. In Vk the first step can do a bunch of work. LLVM pass to create Vulkan object under the hood, for example.
    - MM: opened this issue to try to achieve parity between the 3 APIs. Letting all 3 do some work during compilation.
    - MM: in Metal, the first step would have to have access to a bunch of additional information it doesn't have today, like the pipeline layout, and other things.
    - MM: the way I originally framed this was, how are we going to pass this additional information? But now there's resistance to passing anything at all.
    - CW: same problem with D3D12 in a sense. On DXIL path, can produce multiple entry points at once, have it produce DXIL module. dxc can run llvm passes / optimizations on it. Bit of the same problem there.
    - CW: all graphics APIs work by: create a shader module / LLVM IR (SPIRV doesn't, but every driver uses an LLVM IR), then use that to create the pipeline. Real compilation / optimization for the GPU happens in this phase. Register alloc, instr scheduling.
    - CW: concern: intuition of WebGPU is that pipeline creation would be more expensive than shader module compilation. Not true on Metal, where shader module creation is in same order of magnitude as pipeline compilation.
    - CW: would be really nice for WebGPU shader module with multiple entry points to be compilable as Metal library with multiple entry points to factor out the common cost.
    - MM: Metal's optimized for throughput. Have lots of sources, pass them in as one big batch. High constant time. That's by design. You should pass in a lot of sources in the first stage. Not designed to be called with lots of tiny invocations.
    - CW: didn't mean to diminish Metal compiler. Being able to change pipeline layout, change sampling patterns, etc. - means you'll pay the startup cost of the Metal shader compiler many times.
    - CW: don't know how much information we'd need. Pipeline layout, sample mask, vertex state if you do programmable vertex pulling… ? Taking as a design constraint that one ShaderModule has to be one Metal library is a big constraint

going forward, because we'll break it the minute we do our first driver bug workaround and have to pass more information in.

- CW: problem is that we don't want to require the developer to have to do this, but would be nice for browser to coalesce libraries.
- CW: createReadyPipeline -> createReadyPipeline**s** ? Browser would see lots of shader modules together, and know it could compile them together. Pay the cost of the Metal compiler less, and the smarter the browser is.
- CW: solution where we need more info at ShaderModule creation time is unpalatable. Makes developers' life harder. This one is nice, would be good on its own merits. Do we want it now, later, or at all?
- MM: you said one of the things I was going to - createReadyPipelines seems useful, we'd want to have it even without this discussion. Particular case I'm worried about - common case - application has lots of shaders with lots of entry points, and they want to preprocess ahead of time but can't create pipelines ahead of time. Eg., new character comes on screen and have to create pipeline for it. Does this solution solve that case?
- CW: no. It does solve it in that you could precompile a dummy pipeline if you wanted to.
- MM: all of the solutions could allow that.
- MM: this is a problem important to us, we'd like to see some solution for it. Also, for sampleMask - think that can work with function constants, it's just scalar data. Can put in a placeholder during shader module creation and override it later.
- DM: still need to know whether to put it in the shader or not.
- CW: could put it in and just disable it.
- CW: rephrasing, createReadyPipelines helps, but does not solve all problems. Apps don't know what pipelines they'll use until they need them.
- MM: apps know enough about pipelines that they can set up argument buffers as structs ahead of time, but can't create pipelines ahead of time. This is the common case our Metal team is concerned about.
- CW: if they create pipelines at runtime, in the benchmarks I did by disabling driver cache, cost of creating pipeline was about equal to cost of compiling source for it.
- MM: you're right that we won't be able to move 100% of compilation ahead of time. Of that we can move, is that perf benefit worth it? I think the answer is yes.
- MM: you and I created benchmarks that shows a lot of work can be moved ahead of time. Different workloads, but similar ballpark results.
- CW: do you have a proposal beyond createReadyPipelines?
- MM: I don't have any new proposal. Proposal I made was: add add'l information to shader module creation that has info about pipeline layout. Handle sampleMask using function args. Handle vertex state with same mechanism for index buffer sanitization for when the index buffer's been created by the GPU in an earlier compute shader pass.

- CW: and for any type of workarounds for e.g. Metal bugs needing to inject shader code, then things are going to be really slow?
- MM: is that bug rare enough and you can detect ahead of time whether you need to do shader generation ahead of time. This is a perf improvement. If you can do it for 90% of shaders, that's fine, if the remainder need driver bug workarounds.
- DM: you're suggesting only pipeline layout is provided at shader module creation time?
- MM: I think that's what it boils down to.
- BJ: does this invalidate the code path for developers to query the pipeline layout later?
- MM: it wouldn't. The input to the API you're describing is a pipeline. The question this issue's about - the add'l info for creating pipeline, does that come in beginning or intermediate stage?
- BJ: right now as a developer I don't have to provide a pipeline layout, and BGLs are inferred from the shaders themselves. It's very convenient. If you have to provide layout to get the shader at all, invalidates that convenience API. I can continue to get it back via getBindGroupLayout, but no option to *not* specify it ahead of time.
- MM: I don't think that's correct because.. okay well I haven't come up with a finished proposal, but the flow you've described is one where the author does provide a pipeline layout, they just haven't done it explicitly. It's implicit and they ask what they have created. We can come up with some solution that preserves that flow.
- BJ: it sounds like there's enough information at shader module creation time to still provide that implicit pipeline layout if you don't specify it explicitly?
- MM: in any solution, that would be true, yes.
- BJ: my assumption would be that you'd need both stages, vertex and fragment, but if we don't need that, yes.
- DM: Are we limiting shader modules to a single layout?
- CW: it would be per entry point.
- DM: so dictionary mapping entry point -> layout mapping?
- MM: there are a couple of options I proposed, either passing it in, or encode it in the shader source themselves.
- CW: That option is really less interesting because there's important tricks you can do with BGLs where you can specify BGL that is a superset of what a pipeline is so you can use the same bind group with different BGLs as long as it is a subset.
- MM: Is that impossible with putting the def inside the source code? You can still put unused items there.
- CW: you can. It's more difficult to do. If you update one piece of source code you have to update multiple ones. Can't do it dynamically any more.
- MM: that's an argument I'm extremely sympathetic to. Dictionary approach would be better.

- DM: I still don't think we got Brandon's concern resolved. If you aren't providing the explicit pipeline layout at SM creation for vertex/frag stages, you can't derive that at SM creation b.c. you don't know how they're used together, so you can't build the combined layout of the pipeline.
- MM: I'd have to think about it more.
- BJ: that was specifically my concern.
- DM: think there's a simple solution though. In this case, if the user doesn't provide the specific layout, they don't care about how fast it'll be.
- BJ: could be interesting best practice. If you do care about perf, providing this info could speed up compilation. User friendly and optimization-friendly.
- MM: we do have precedent for this in minBufferSize.
- CW: What we've been discussing so far is a potential solution where you create the metal library at pipeline creation time. I still have measured concerns because not every browser is Safari. It's easier if you control the browser, the driver, and the workarounds. Having seen crazy workarounds going into ANGLE, I don't feel like 1 SM == 1 MetalLib will last very long at all. It seems to be a design constraint that is unusable by Chromium/FF in the long term b.c. we care about older versions of MacOS.
- KR: Concrete example: we spent over a week investigating alpha: false on context creation on intel macs. Doesn't handle color mask well, and if you turn off alpha, it kills perf. Only solution was to dynamically generate shaders based on whether you're writing to the default framebuffer and alpha: false. Lots of complex state and swapping shader programs. Huge performance cliff. Very likely similar issues that are unexpected will pop up on WebGPU.
- MM: I find this argument.. well I wish there were some analysis about Metal and not OpenGL, and I think probably Corentin's intuition about the number of bugs is different from my intuition.
- CW: We've already found maybe 10 bugs on Metal on various drivers. ex. depth-compare in a compute shader breaks on intel iris. We're very early and this is just Dawn testing a couple things, and we're already finding a lot of small bugs, have three workarounds, filed a few bugs, and quite a number of suppressions. It's a graphics API and there are many bugs. I don't doubt that the engineering is amazing, but there are inevitably going to be issues.
- CW: other thing: we're talking about perf. Need to talk about where perf happens. B/c of driver caching, browser caching, perf improvement only happens during cold load of website. Reduction of the problem. We have createReadyPipelines that can be used by websites seeing this as a big problem. This additional cost of compilation happens for dynamically generated pipelines for cold loads only. The cost is 2x as big, not 10x as big. Seems that createReadyPipelines solves most of the problems for us. Separately, have concerns about providing pipeline layouts during shader module creation. Think we should add createReadyPipelines and say that perf might not be as fast as it could be in all cases.

- MM: I think it's not particularly fair to say cold load is more important or hot load is. Both are super important. In the web context cold load is super important b.c. people browse to websites. Of the four sets of data you described, .?. all of them were wins. Three of the four had big wins. One had a small win.
- CW: if you count 2% as a win for pipeline compilation.
- MM: 3 of the 4 had big wins.
- CW: if you have createReadyPipelines, this problem's reduced to a tiny subset of the current set of problems, taking into consideration driver and browser caching. The problem's less important because it's much reduced. Saying we won't solve this means that people will use createReadyPipelines - drawbacks are less than drawbacks of requiring more information during ShaderModule creation, because it's inconvenient to the developer, designs us into a corner. createReadyPipelines has fewer drawbacks than the other.
- MM: Okay, not sure how to respond except what I said in the beginning. The case we're worried about is when you can't create the pipeline until runtime.
- JG: earlier someone talked about creating a dummy pipeline? Is that a viable approach?
- CW: it depends. Random fixed function state can cause a different pipeline shader to be emitted.
- JG: not sure there's a way to solve it, unless you surface a pure set of functions for all the cases. They might use createReadyPipeliens if they can take advantage of it, or on this particular system it doesn't provide an advantage and there are flexibility tradeoffs for doing it the other way. createReadyPipelines would give us the best chance of doing things up front for creation of a Metal library. Best-effort thing. We're trying really hard to do this, but don't think we can agree on a set of guarantees for the lifetime of this API where we don't have fallbacks to the pessimal pattern - take the shader text, but don't compile it until you know more about how it'll be used.
- JG: createReadyPipelines seems like the closest thing to a solution as we can get. I feel like any situation where…. where you create a dummy pipeline, we'd still capture that performance and let us do whole-pipeline rewriting.
- DM: Myles did you consider a kind of "hinting" API - you'll tell the shader module I'll use these entry points with this data?
- MM: That's one way to think about my proposal. Give this info, and if impl finds it can't compile right now, it defers it.
- CW: that's something you could pass to ShaderModule creation, but not binding on WebGPU validation.
- MM: so if they get it wrong they pay the cost twice?
- CW: yes. That it's a pure hint negates whether developers can provide this information up front. They don't need to.
- MM: interesting thought. Have to figure out how many devs would get it wrong, and how many would not supply it. If it's 90% of devs, no point.

    - JG: in the spirit of providing a hint, createReadyPipelines always seems like a hint. If you create a different ready pipeline later with same shader module with different details and the impl can reuse its previous pipeline, it can go ahead and do so.
    - CW: You just need to tell the pipeline that it's actually just a hint and don't compile for real.
    - JG: in the same spirit as today if you do multiple createReadyPipelines with same shader module. If we made 0 changes to the spec that's what I'd expect all impls to do today.
    - MM: I think we would probably be ok with a hint, provided that the hint is optional. There's a distinction between "I'm giving you nothing" vs. "I'm giving you something so please do something". Explicitly not causing double compilation would be a requirement.
    - CW: for clarity you're saying you'd be OK with a hint. Can you be more precise?
    - MM: Shader module creation taking an optional pipeline layout. If you provide it, impl might do some ahead-of-time compilation.
    - DM: we still need to see your proposal on how to get around the sampleMask - understand the vertex part, but not the sampleMask part.
    - CW: Concerned about vertex state
    - DM: Concerned about sample mask because it could ruin early depth and other graphics state.
    - MM: makes sense. I can write up something by Thursday. Wanted to mention, sample mask not particularly important for MVP. Post-MVP feature?
    - CW: we already have it. We're going to add stuff post-MVP, and if we impose this constraint and find we have to emulate the functionality later, ...
    - DN: If this is a hint, can we make sure that specialization constant or function constant values can play in early? Ability to expand the things you pass in would be great!
    - CW: sounds like you want #defines.
    - DN: no. :)
    - DN: layout might not be the only parameter.
    - MM: so intermediate object that would contain layout and maybe other things.
- Operator Precedence (Umbrella ☂️ issue #1146)
    - Reorder expression sections (#1136)
        - **Quick Links**
            - **Precedence before:**
                - gpuweb/pull/1111#issue comment-707941325 🔗
            - **Used to match:** HLSL, GLSL, C++
            - **Precedence after:**
                - gpuweb/pull/1111#issuecomment-705039742 🔗
            - **Matches:** Python, Swift, Rust
            - **What's different:** Bitwise ops get higher precedence
        - **Discussion**

    - Introduce operator precedence table (#1111)
    - Partially-ordered precedence (comment on #1146)
        - KN: proposed partial ordering
- what is the initial value of a workgroup variable? (#1137)

Agenda for next meeting

- Thought most of Wednesday would be for API, and Thursday for WGSL topics?
- MM: sounds good.
- Agreement to cancel first hour of Thursday's call so as to not conflict with WebGL concall.
- CW: like to call attention to #1154 - importing web platform images into WebGPU.
- DJ: WGSL people, please add more things to the project. Operator precedence is up next.