NXT buffer mapping

cwallez@google.com, kainino@google.com

Please read the “memory barrier” doc first!

The following assumes we use usage transitions like described in the doc.

Goals

Minimize the number of copies

Applications that use the GPU need to upload very large amounts of data to the GPU during the initialization, easily reaching hundreds of megabytes of data. After the loading is complete, comparatively little data is uploaded each frame, but upload performance is still paramount because it needs to happen 60 times per second or more.

The number of times data is copied before it reaches its final destination is the biggest factor in upload performance. In WebGL uploading a buffer follows these steps, each incurring a copy:

1. The buffer data is decompressed into an ArrayBuffer
2. gl.bufferSubData is called and the data is copied to shared memory (only for Chrome)
3. The driver’s glBufferSubData is called and the driver copies to an internal GPU-visible ring buffer.
4. A GPU copy is scheduled that copies from the ring buffer to GPU RAM

So data upload in WebGL goes through 3 or 4 copies depending on the browser. On the other hand the minimal number of copies possible is two, with one copy for the decompression, and one copy from CPU memory to GPU memory (assuming non-UMA). That’s our goal.

If WebGPU exposes a buffer mapping primitive, the application will be able to decompress directly into either shared-memory (for Chrome) or directly into GPU-visible memory, avoiding one copy. This gets us to three copies in Chrome, and two copies in other browsers, which is the best possible.

The same analysis holds for getting data from the GPU to the CPU, although in that case the amount of data transferred is often minimal.

Avoid data races

Since we assume the API tries to prevent data races on the GPU, it makes sense to also prevent data races between the CPU and the GPU for the exact same reasons. This means that the CPU should not be able to read or write buffer memory while the GPU is using the buffer.

Asynchronous GPU to CPU transfer

One issue in WebGL is that reading back data from the GPU via ReadPixels or GetBufferSubData is synchronous and can cause the CPU to wait for the GPU. To prevent such stalls, WebGPU should provide an asynchronous way to read back data from the GPU. WebGL is adding the same capabilities with an extension for GetBufferSubDataAsync.

Buffer Mapping in NXT

NXT has two buffer usages related to mapping, MapRead and MapWrite that allow mapping the buffer respectively for reading and writing. A single Map usage might have been enough but:

- We don’t see a compelling use for MapRead | MapWrite buffers
- The separation helps on D3D12 that has “upload” and “readback” heaps
- The separation could help NXT be smarter

For now, for simplicity, we require mappable buffers to be used only as transfer buffers. This means that, when creating a new buffer, the allowed-usage set has additional restrictions: the MapRead usage is allowed in combination with only TransferDst, and MapWrite is allowed in combination with only TransferSrc, and no other usages.

Here’s an example of mapping a buffer for reading:

```
// In NXT the API is callback based for now, but callback vs. closure vs. promise etc.
// is part of the cosmetics, please don’t focus on it.
void OnMapRead(nxt::BufferMapReadAsyncStatus status, const void* data, void* userPtr) {
    // Stuff
}

nxt::Buffer buffer;

// Currently the buffer is in flight

// The callback will only be called once the GPU is done using the buffer
buffer.TransitionUsage(nxt::BufferUsageBit::MapRead);
buffer.MapReadAsync(0 /*offset*/, 4 /*size*/, OnMapRead, nullptr /*userPtr*/);

// After the callback has been called
buffer.Unmap();
buffer.TransitionUsage(nxt::BufferUsageBit::TransferDst);
```

Buffer mapping for reading follows a number of rules (extends trivially to mapping for writing):

- A buffer is created in the unmapped state.
- MapReadAsync requires the buffer to have MapRead in its current usage.
- A valid call to MapReadAsync puts the buffer in the mapped state.
- Unmap is only valid if the buffer is in the mapped state.
- A valid Unmap puts the buffer in the unmapped state.
- A buffer cannot be transitioned while it is in the mapped state.
- The callback will be called exactly once.
- If Unmap is called before the callback, the callback is called with a “Canceled” status
- If the buffer in the mapped state is destroyed before the callback is called, the callback is called with a “Canceled” status.
- (hand wavy) the callback is called as soon as possible with either a “Success” status and a pointer to the data, or an “Error” status
- The pointer passed to the callback is only valid while the buffer is mapped
- We haven’t yet addressed the case where MapReadAsync is called multiple times

Both types of buffer mapping prevent data races with the GPU, because the buffer needs to be in the MapRead or MapWrite usage continuously until the async callback is called. To be used by the GPU during the same time, it would have to be in a different usage, which is not possible (or would cancel the callback).

In practice we haven’t implemented MapWriteAsync yet, and instead have a nxt::Buffer::SetSubData call that requires the TransferDst usage and acts as if the buffer’s data was immediately updated. While MapWriteAsync should be used for high-performance uploads, SetSubData should be very useful during application development. We suggest WebGPU keeps this convenience feature (or a MapWriteSync) even though it is suboptimal.

Proposal for a WebGPUMappedMemory object

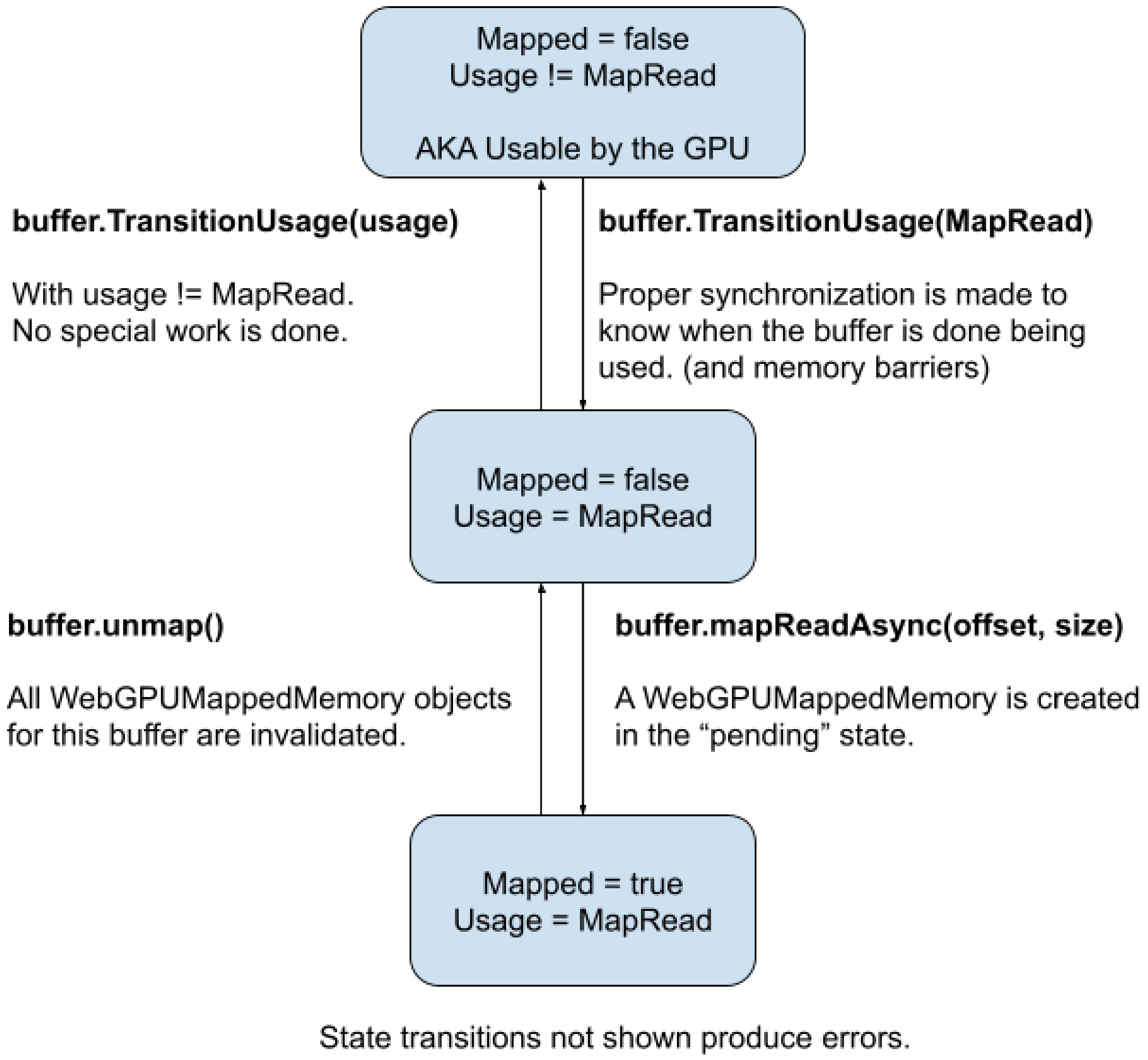
WebIDL:

```
// An object that supports both the poll and promise interface.
interface WebGPUMappedMemory {
    bool isPending();
    ArrayBuffer getPointer(); // not sure about the return type yet, could be SAB.

    // Implementing a .then method that acts like a promise’s .then method should
    // make this object interchangeable with a promise, even Promise.all should work!
    void then(SomeCallbackType);

    // This can be unmapped early but doesn’t put the buffer in the unmapped state.
    // Even if it was the last mapped memory (?)
    void unmap();
};

interface WebGPUBuffer {
    // “Memory barrier” API, like NXT in this proposal, but the only thing we need is
    // to ensure DRF between the CPU and the GPU.
    void transitionUsage(int usage);

    // This is only allowed in the MapRead usage. It puts the buffer in the mapped state
    // until it is unmapped, which disallows usage transitions (important for DRF).
    // Multiple of these can be created before an unmap().
    WebGPUMappedMemory mapReadAsync(int offset, int size);


    // Invalidates all WebGPUMappedMemory objects for this buffer, meaning that the
    // “promise” is resolved with a “cancelled” status, isPending() returns false and
    // getPointer() returns null.
    // After this, the buffer can be transitioned again.
    void unmap();
};
```

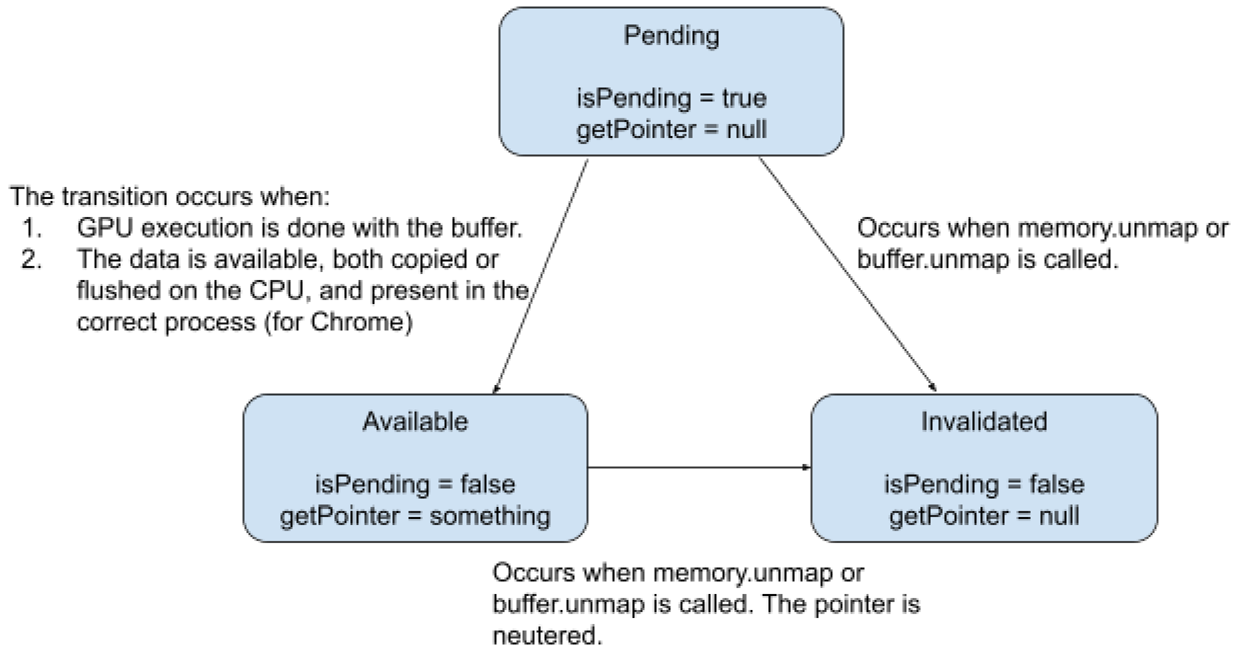
JS code (with preprocessor >_>)

```
buffer.transitionUsage(MapRead);
let mapped = buffer.mapReadAsync(offset, size); // is a WebGPUMappedMemory

#if SYNCHRONOUS
{
  // Wait for data to be available by polling, here showed by spinning but would
  // would be done in a setTimeout or a RAF.
  // /!\ This while loop is an image of what would be done via continuation in a
  // RAF / setTimeout. Looping without yielding back to the browser is NOT something
  // we would like to support.
  while (mapped.isPending()) {
    // spin
  }
  let ptr = mapped.getPointer();
  if (ptr) {
    // Do something with the data in ptr.
  } else {
    // Request was cancelled or had an error.
  }
  mapped.unmap();
}
#else
{
  // WebGPUMappedMemory can be "thenable" without actually being a Promise.
  // This seems to work with Promise.all, etc., too.
  mapped.then(ptr => {
    // ...
    mapped.unmap();
    buffer.transitionUsage(...);
  });
}
#endif
```

State machine for WebGPUBuffer:


Mapped = false
Usage != MapRead
AKA Usable by the GPU
buffer.TransitionUsage(usage)
With usage != MapRead.
No special work is done.
buffer.TransitionUsage(MapRead)
Proper synchronization is made to know when the buffer is done being used. (and memory barriers)
Mapped = false
Usage = MapRead
buffer.unmap()
All WebGPUMappedMemory objects for this buffer are invalidated.
buffer.mapReadAsync(offset, size)
A WebGPUMappedMemory is created in the “pending” state.
Mapped = true
Usage = MapRead
State transitions not shown produce errors.

State machine for WebGPUMappedMemory:


Pending
isPending = true
getPointer = null
The transition occurs when:
1. GPU execution is done with the buffer.
2. The data is available, both copied or flushed on the CPU, and present in the correct process (for Chrome)
Occurs when memory.unmap or buffer.unmap is called.
Available
isPending = false
getPointer = something
Invalidated
isPending = false
getPointer = null
Occurs when memory.unmap or buffer.unmap is called. The pointer is neutered.

Open questions

UMA / single write, single read resources

There are use-cases when we want the GPU to read from the CPU memory. This happens on Unified Memory Architecture (UMA) systems where both memories are the same, but is also useful with discrete GPUs when the application knows that a resource will be read just a few times (such as software decoded video, read as a storage buffer).

It would be possible to lift the restrictions on using MapRead and MapWrite with other usages, but we suggest this is made as an extension so that developers need to require it explicitly and know what they are doing.

Linear textures

Mapping could also be beneficial for textures that are read just a few times, but our proposal doesn't address this.

Efficient UMA alternatives

In the current proposal, the Map* usages are disallowed with anything but the correct Transfer usage because on discrete GPUs, CPU-visible memory would be allocated, which would be slow if you plan on accessing the memory many times on the GPU. For example using Map |

Vertex usage and mapping to upload the data would result in really slow vertex buffer access by the GPU.

However this enforce one extra copy on UMA systems which is bad. Here’s some alternatives on how to solve this extra copy on UMA.

1 - Add “Map*Staged” usages

Add a Map*Staged (MapReadStaged and MapWriteStaged) usages that are allowed with any other combination of usages and implies that a staging copy happens when mapping a resource and running on a discrete GPU.

Pros:

- Good for UMA as Map*Staged would simplify to mapping the buffer directly.
- The staged behavior is opt-in and explicit.

Cons:

- Adds redundancy on UMA systems where both Map* and Map*Staged are equivalent.
- Complexifies the API

2 - Add an UMA extension

Add an extensions that is exposed only on an UMA system. When the extension is enabled, Map* is allowed with any usage.

Pros:

- Allows the good path on UMA.
- The extension is enabled explicitly.
- Keeps the API simple.
- Applications are likely to test both code paths if it is an extension.

Cons:

- Requires defining the extension mechanism (could be a pro actually)
- Most systems are UMA, only NVIDIA and AMD are not. The most common case shouldn’t require an extension.

Lift restrictions on Map*

Lift restrictions on using Map* usages with other usages. Using Map* with anything else than transfer ends up staging on discrete GPUs. Expose an isUMA flag to let application avoid staging if they want to.

Pros:

- Simple.

- Works decently everywhere.
- Defaults to be fast on UMA which is the most common.

Cons:
- Magic happens on discrete when Map* isn't used just with Transfer* but this wouldn't be in the spec. Making it implicit knowledge about the implementation.
- Adds implicit staging on discrete.

Discussion

cwallez@
Solution 1) was our initial idea, but it feels clunky and ugly compared to 2) and 3). It is basically the same as 3) but separates the usages that are optimized for UMA and discrete.

2) and 3) seem both pretty good. Slight preference for 2) because it is the most explicit version, and UMA vs. discrete is a big enough difference that most applications will handle both cases efficiently. In addition 2) makes it easy to have a "storage mode" like MTLStorageMode that says that resource should preferably be accessible by the CPU or device local.

kainino@: conceptual pseudocode uploading data to a vertex buffer

2 - UMA extension	Unoptimized app	`B1 = mkbuffer(MapWrite | TransferSrc)`
		`B2 = mkbuffer(TransferDst | Vertex)`

	Optimized app	`If (uma):`
		`  B1 = mkbuffer(MapWrite | Vertex)`
		`Else:`
		`  B1 = mkbuffer(MapWrite | TransferSrc)`
		`  B2 = mkbuffer(TransferDst | Vertex)`

1 - Add Map*Staged	Unoptimized app	`B1 = mkbuffer(MapWriteStaged | Vertex)`

	Optimized app	`If (uma):`
		`  B1 = mkbuffer(MapWriteStaged | Vertex)`
		`Else:`
		`  B1 = mkbuffer(MapWrite | TransferSrc)`
		`  B2 = mkbuffer(TransferDst | Vertex)`

3 - Lift restrictions on Map*	Unoptimized app	`B1 = mkbuffer(MapWrite | Vertex)`

	Optimized app	`If (uma):`
		`  B1 = mkbuffer(MapWrite | Vertex)`
		`Else:`
		`  B1 = mkbuffer(MapWrite | TransferSrc)`
		`  B2 = mkbuffer(TransferDst | Vertex)`