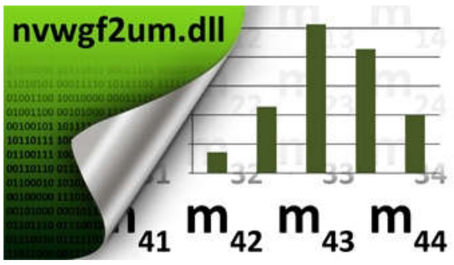
Constant Buffers without Constant Pain

By Holger Gruen, posted Jan 14 2015 at 10:38AM
(/#facebook) (/#twitter) (/#linkedin) (/#google_plus)
Tags:
GameWorks (/taxonomy/term/315), GameWorks Expert Developer (/category/tags/gameworks-expert-developer), DX12 (/taxonomy/term/278), DX11 (/category/tags/dx11)

Constant Buffers without Constant Pain



Since the release of DirectX 10 ™ 3D programmers have had to deal with constant buffers as the way of passing parameter constants to GPU shaders. Changing shader constants became a whole lot more tricky, as sub-optimal updates to constant buffers could have a very severe impact on the performance of a game.

Under DirectX 9 ™ approximately 80% of all command buffer traffic sent to the GPU was shader constant data. Smart game engines avoided falling into this trap by performing partial constant updates, while other engines would send all updates for every Draw() call – which is obviously extremely wasteful.

DirectX 10 ™ introduced the concept of dedicated constant buffers to reduce the amount of data that needs to be sent from application to runtime to driver to GPU. Game engines were supposed to turn the necessary constant changes into updates of a few small constant buffers. Unfortunately it turns out that a bunch of small updates per draw call can create severe bottlenecks.

The goal of this post is to revisit constant buffer usage patterns and to also take a look at some of the superior new partial constant buffer updates that DirectX 11.1 ™ in Window 8 ™ allows for. Hopefully this will help alleviate some of the pain still felt by developers.

Note that some of the data that is shared below has been gathered using NVidia drivers and GPUs, so some of it may be different using drivers and GPUs from other vendors.

Constant Buffer Basics

Before the advent of constant buffers, there wasn't any way to keep shader constants around when changing shaders. So a shader change did require setting all constants again, as their values were tied to the previous shader. Furthermore, if the same values had to be set for both the pixel and vertex shader stages, they had to be set twice, as constants could not be shared between stages.

Constant buffers changed this. They're objects that preserve the values of the stored shader constants until it becomes necessary to change them. A constant buffer can be bound to any number of pipeline stages at the same time. So from a conceptual point of view there is no need to store any constant twice. Unfortunately binding the same constant buffer to several shader stages will make updates more expensive (see Table 1 below), so in practice you may want to avoid this.

Constant Buffer Updates and Constant Buffer Renaming

If an application wants to change the contents of a constant buffer, usually the old contents are still needed because either the GPU is still accessing them or has not yet used/consumed them. As a consequence, to avoid stalling, the graphics driver returns a pointer to a different block of memory for each pair of calls to Map(MAP_WRITE_DISCARD)/Unmap() or a call to UpdateSubresource().

The driver uses a certain quota of memory (currently 128 MB for NVidia drivers) for constant buffer renaming. The accumulated memory size of all rename operations can grow so much that this causes the graphics driver to run out of renaming space. If this happens, the driver temporarily throttles the game until it has recovered. This throttling event should be avoided and should not happen often during a frame.

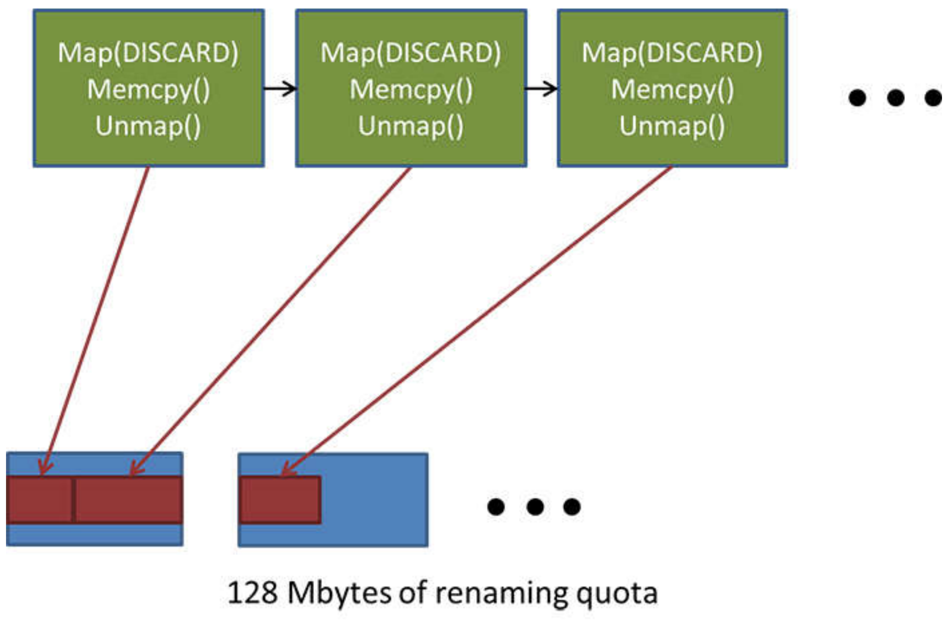


Figure 1 – Renaming a sequence of constant buffer updates

Example: 4096 bytes of constant buffer data (2048 for each, the pixel stage and the vertex stage) being updated for 10000 draw calls per frame, gets renamed to ~156MB (assuming 4 frames are in flight – e.g. for an SLI configuration). So clearly this exceeds the above mentioned limit of 128MB, causing the driver to throttle down.

Why Can Updating Constant Buffers be so painfully slow?

Let's talk about the cost in CPU cycles on a contemporary CPU (3.3GHz Intel ™ Core-i7™) for various constant buffer operations – see Table 1 or Figure 2 which offers a different view of the same data.

Please note that the absolute numbers quoted here are not that important, but rather the relative cost that matters.

Operation	Approximate number of CPU cycles	Comments
DrawIndexed()	36	
XXSetConstantBuffers()	114	Keep this in mind when using the same constant buffer across several shader stages.
Map(DISCARD)+Unmap()	256 x (number of shader stages that have the constant buffer bound)	Keep this in mind when using the same constant buffer across several shader stages
UpdateSubresource()	214 x (number of shader stages that have the constant buffer bound)	This number does not account for the cost of copying the actual data
cache miss when using a constant buffer	100-200	This is the cost you pay when a constant buffer is used for the first time or when it isn't in the cache anymore

Table 1 – CPU cycle cost of constant buffer operations

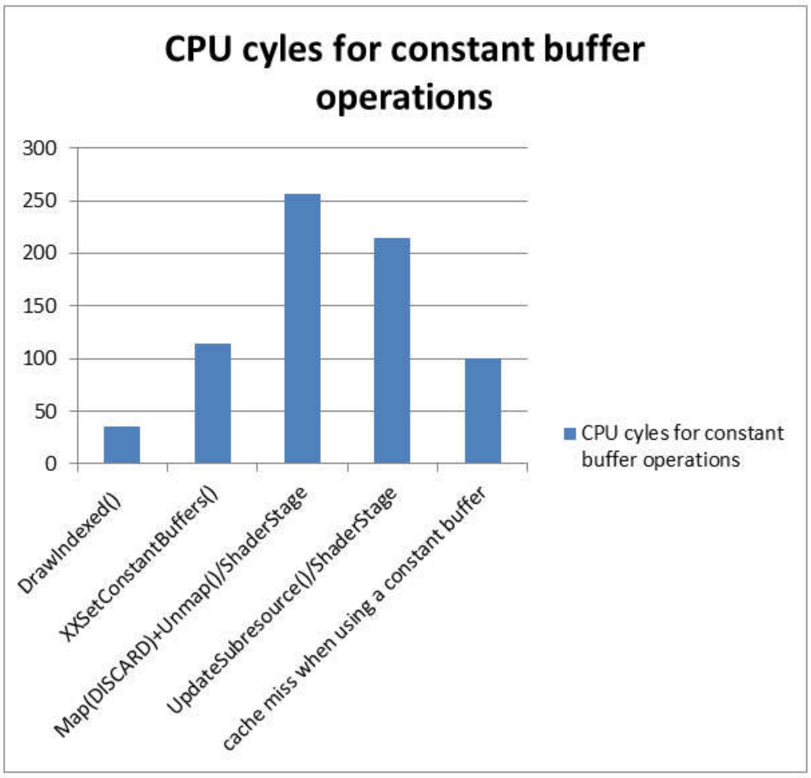


Figure 2 – CPU cycle cost of constant buffer operations

Directed tests (such as perfX2/overhead11/TDrawIndexed) measure the DrawIndexed() throughput in millions of DrawIndexed calls per second:

Result 1: One can issue ~90 million DrawIndexed() calls per second, if no constant buffer updates are made in between calls.

Result 2: If one updates just one constant buffer (e.g. using UpdateSubresource()) in between draw calls, the issue rate suddenly drops to ~11 million DrawIndexed() calls per second.

In other words: Changing a single constant buffer in between draw calls reduces the number of draw calls you can issue per frame by almost an order of a magnitude.

When a constant buffer gets updated it is advisable to first create a CPU side version of the buffer that sits in system memory. How to place the CPU side version most efficiently in memory will be described now.

Allocating Memory for Constant Buffer Updates

The memory you use to build your CPU-side version of a constant buffer shouldn't just be a local variable.

In order to make copying memory to constant buffers fast it makes sense to use _aligned_malloc() to allocate memory that is aligned to 16 byte boundaries, as this speeds up the necessary memcpy() operation from application memory to the memory returned by Map(). In a similar throw UpdateSubresource() will be able to perform the copy operation faster too.

So let's now talk about how you should ideally be updating constant buffers in-between draw calls.

Do's and Don'ts for updating constant buffers using DX10/DX11.0

1. Try to update only **one** constant buffer in between draw calls. This constant buffer should only be bound to a single shader stage, if possible. Where this is not possible, only set and update the absolute minimum number of constant buffers necessary to do the job.

2. Update constant buffers using one of these two methods:
    a. Map(MAP_WRITE_DISCARD) -> memcpy() the system memory data -> Unmap()
    Avoid reading from the mapped write combining memory block (checkout http://msdn.microsoft.com/en-us/library/windows/desktop/aa366786%28v=vs.85%29.aspx#PAGE_WRITECOMBINE (http://msdn.microsoft.com/en-us/library/windows/desktop/aa366786%28v=vs.85%29.aspx#PAGE_WRITECOMBINE)), as these reads are usually un-cached and can have an unexpectedly heavy performance impact if they happen more than a handful of times. Watch out for implicit reads that e.g. the '+=' operator introduces when being used on C++ objects that are stored in the mapped memory block. Ensure that writes to constant buffers are sequential. One way to make sure this happens is to memcpy() a local version of the buffer to the mapped memory location as indicated above.

Make sure to read: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476457%28v=vs.85%29.aspx (http://msdn.microsoft.com/en-us/library/windows/desktop/ff476457%28v=vs.85%29.aspx).

b. UpdateSubresource()
NVidia drivers map UpdateSubresource() to Map(MAP_WRITE_DISCARD)->memcpy()->Unmap(). This means you'll see the cost of the memcpy() if you profile the CPU cost of UpdateSubresource().

3. Don't update a subset of a larger constant buffer, as this increases the accumulated memory size more than necessary, and will cause the driver to reach the renaming limit more quickly. This piece of advice doesn't apply when you are using the DX11.1 features that allow for partial constant buffer updates.

4. Avoid switching between different constant buffers, as often as possible. Be aware of the cost for a cache miss of a fresh constant buffer.

Now that we have discussed how to update constant buffers under DX10 and DX11.0 let's take a look at what DX11.1 brings to the table.

Updating constant buffers using DX11.1 and above

DX11.1 adds API features that allow an application to manage constant buffer memory directly. It allows the creation of big constant buffers that can be filled with data for multiple rendering operations in one go.

This is possible in DX11.1, as the constant buffers for shader stages can be set in a way that only a subsection of a larger constant buffer ends up being utilized by a shader – e.g. by using PSSetConstantConstantBuffers1() for the pixel stage.

This basically allows the programmer to work around the renaming limit inside the driver. In an extreme case, one can prepare a big constant buffer that contains all the constants for one frame's worth of rendering.

This would lead to the following optimal usage pattern:

1. Map(MAP_WRITE_DISCARD) a big constant buffer
2. For all remaining draw calls– initially this is all of the draw calls
    a. Append per draw call constants to the mapped memory
    b. If constant buffer is full exit loop.
3. Unmap() the big constant buffer
4. For all draw calls that step 2 has generated constant data for
    a. set the right part of the big buffer as a constant buffer XXSetConstantBuffers1()
    b. Perform draw call
5. If all draw calls have not been processed restart at 1.

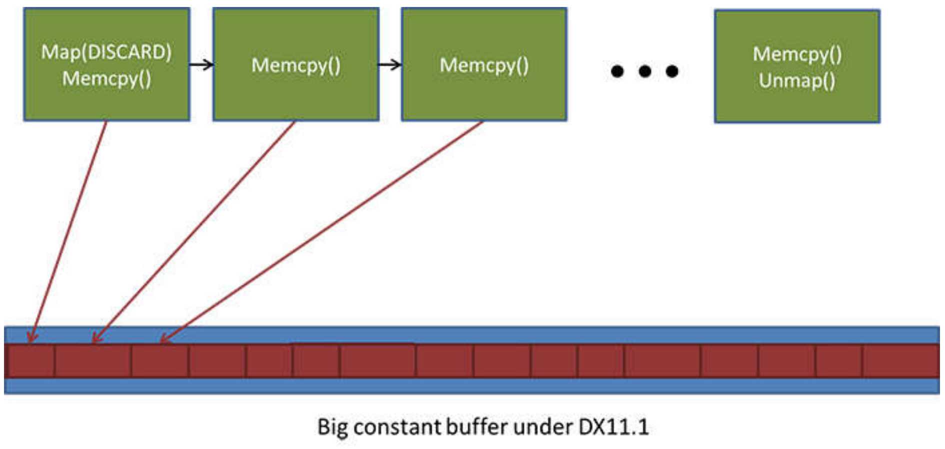


Figure 3 – How to fill a big constant buffer under DX11.1

Alternatively one can use the following pattern:

1. Use UpdateSubresource1(WRITE_DISCARD) to copy the constant data for the first of the remaining draw calls - initially this is all of them
2. For all remaining draw calls – initially this is all of them minus the first
    a. Use UpdateSubresource1(NOOVERWRITE) to copy constants for the current draw call
    b. If constant buffer is full exit loop.
3. For all draw calls that step 2 has generated constant data for
    a. set the right part of the big buffer as a constant buffer XXSetConstantBuffers1()
    b. b. Perform draw call
4. If not all draw calls have been processed restart at 1.

All constant buffer data that is passed to XXSetConstantBuffers1() needs to be aligned to 256 byte boundaries. You need to make sure that you update the memory inside the big constant buffers accordingly.

Please note that the DirectX 11.1 implementation of XXSetConstantBuffers1() and of UpdateSubResource1() do not work on Windows 7 ™. You can really only rely on proper support on Windows 8 ™ and above. See http://msdn.microsoft.com/en-us/Library/Windows/Hardware/dn653328%28v=vs.85%29.aspx#buffers (http://msdn.microsoft.com/en-us/Library/Windows/Hardware/dn653328%28v=vs.85%29.aspx#buffers)

No more pain?

I hope this little write up helps to digest the performance implications constant buffers create. The things to keep in mind are:

a. Be aware of the costs of the various constant buffer operations as presented in Table 1
b. Be aware of the do's and don'ts for updating constant buffers

If you can adhere to these guidelines hopefully all constant buffer pain will subside...