GPU Program Caching

Owner: dmurph, gman
Issues: crbug.com/88572,
Major CLs:

- 10534173 (in-memory storage only) -- reverted
- 10797055 re-submit, fixed extension detection
- 10812002 fixed new & old bugs with getTranslatedShader
- 10826073 small fix for behavior re-caching behavior, and fixed null pointer
- 10837009 adding histograms

Table of Contents:

Overview/Why?

Because of sandboxing, every time a page loads we translate, compile, and link its gpu shaders. While not every page defines shaders, the compositor uses some shaders that have to be recompiled for every tab. We should be able to cache previously linked programs, and reuse them when they are requested again.

We do this with a gpu program cache, which will employ both in-memory and disk-based caching to speed up this process.

Cache Levels

In-Memory Cache

Due to the unknown time of disk access (along with the IPC calls needed), all cache hits for binary programs will be from the in-memory cache. Loading from the disk-based cache will happen on startup.

Key Considerations

Our in-memory cache is stored in the gpu channel manager, so it's around for the lifetime of the gpu process. Because of this, we can assume that the binary representation of the same shader will not change during the lifetime of our in-memory cache (driver won't change, vendor won't change, etc). So our key can just consist of the untranslated shader source. Because we don't want unbounded key sizes, we'll use a the SHA1 hash of the source.

When referencing a gpu program (which contains two shaders), we also need to include the bound attribute location map in our key, as this affects the resulting binary and can be different with the same shaders. So we do a SHA1 of the two shader sha1's and then the bind attrib map.

Disk Cache

The disk cache will act as persistent storage for the in-memory cache. It will have the same maximum size as the in-memory cache, and all programs cached to the in-memory cache will also be sent to the disk cache.

One option for allowing a disk-based cache hits involves 'locking' the gpu program info and asynchronously fetching the binary while we continue execution. If any future calls involve that gpu program, then that call will wait till the async loading is done. However, because of the common pattern of checking the link status of the program immediately after linking it (so the

program is used immediately after the async call is initiated), this defeats the option.

Key Considerations

Since this will persist on disk, we need to include anything that will affect the binary representation of an untranslated shader. This includes changes to both the driver and possibly the translator in chromium. So we want to include:

- untranslated shader sources
- bound attribute location map
- glGetString(GL_VENDOR)
- glGetString(GL_RENDERER)
- Driver Version ID
- Vendor ID
- Chrome Build # *

* This is not available to us in the gpu project, this is only in the chrome project. If the disk-cache is residing in chrome, this should be fine.

Behavior

The disk cache has to add the ability of persisting the program cache across launches without causing any performance hits. Because of unknown disc access time (and the fact that just compiling and linking a program can be faster than getting the binary from the disk), we are never going to use the disk-based cache as a direct provider for the cached binaries. Instead, they will be loaded into the in-memory cache on startup.

For optimal behavior, the disk cache needs to:

- Load the binaries on the disk on startup.
    - Because binary size is around 1-20kb, and we'll be using IPC's, we can't load these all at once
    - Worst case scenario for disk cache is that we have a file per entry, so this shouldn't block the startup. Instead this needs to be done lazily on a separate thread
    - "Compatibility of key" check should happen before we send an IPC (above)
- Perform cache updates/writes asynchronously (never reading, just keeping it up to date with the in-memory cache)
- Delete on browser cache deletion

The implementation has to be mindful of the race conditions happening on startup, where the compositor is using shaders right away, which may or may not be supplied yet from disk. This also begs the question: Should we be able to flag certain programs as load-immediately-on-startup? Is it faster to grab these binaries for the compositor shaders from disk, or to just have them compile&link normally? This can be considered for the future, probably overkill though.

Eviction

Considerations

Being resource loading, the optimal eviction plan for a single page is MRU. This is because we never re-use the same program binary twice, we just load it once.
However, this cannot be used when considering we're running across multiple pages, because page access use does not follow a resource loading pattern, and background pages effectively limit the cache space available for the current page.

So a better eviction plan is LRU with pages and MRU for each page. So you would evict from the most recently used program on the least recently used page. Keep in mind that if a program is reused on an old tab in a new tab, that program would be removed from the old tab's MRU stack and put in the new tab's MRU stack.

Final choice

Because we can't reliably know what tab/page we're on in the gpu process, **we evict using the LRU policy**. This policy will only cause issues if we cannot fit all of the gpu programs of one page entirely in the cache, as then we'll be evicting the first programs cached on the page, and on reload we'll be re-compiling all of them. Currently the sizes of binaries are pretty manageable (Loading both Mini Ninjas and From Dust resulted in a little less than 6mb or binaries), but if this becomes an issue then the eviction plan should be as outlined in the paragraph above.

Status Storage

Before we fetch or save program binaries, we do a series of ‘status’ checks to avoid translating/compiling/linking shaders in case they are cached. So we store the following status information:

Status storage	Key	Value
Shader Compilation	SHA1(untranslated shader)*	- Compilation status (Success, Unknown) - Reference count of linked programs**
Program Linking	SHA1(SHA1(untranslated vertex shader)*** + SHA1(untranslated fragment shader) + attribute location binding map)	- Link status (Success, Unknown)

* SHA1 is used to avoid keeping copies of the unbounded untranslated shader in memory
** We keep a reference count so we know when to remove the shader compilation status when we evict programs (because the same shader can be used in multiple programs)
*** The linking key uses the SHA1s of the shaders so we can reference the shader compilation status during eviction w/o having access to the original untranslated shader source.

Binary Storage

In-Memory

After we finish linking a program or determine that a program is in the cache, we access the in-memory binary storage. The key for this storage will be the same as the program linking status key described above, and we store the following values in the cache:

- SHA1(untranslated vertex shader)
- SHA1(untranslated fragment shader)
- vertex shader attribute to shortened name map
- vertex shader uniform to shortened name map
- fragment shader attribute to shortened name map
- fragment shader uniform to shortened name map
- values from glGetProgramBinary:
    - length

    - format
    - data

We're storing the hashed shaders so we can correctly evict w/ the compilation status storage

Disk

The storage scheme is still undecided, but we need to accommodate the following:

- Store everything that's needed for the in-memory binary storage
- Store everything that's needed to create the in-memory keys
- Not load into memory values that don't match the key values described in DiskCache-KeyConsiderations

Histograms

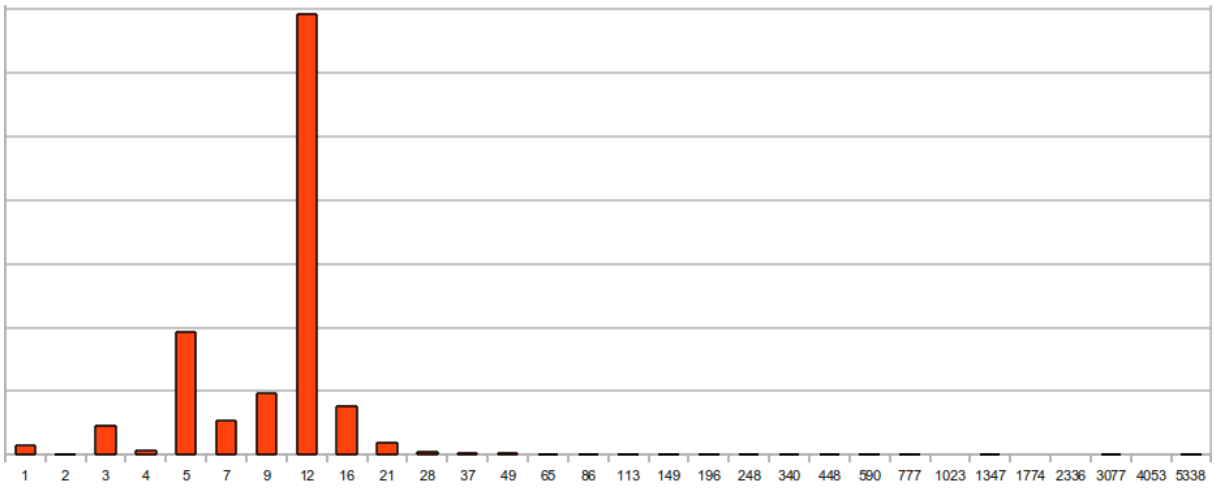
The following histograms would help us tune the cache:

- Program binary size - per linked binary, not per use
- Program cache size (before + after storage)
- Binary cache hit time
- Binary cache miss link time
- Status cache hit time
- Status cache miss compilation time

Note: Histograms of builds from Aug 20th to Aug 22nd for the binary cache hit time are incorrect.

Cache Size Distribution Calculation

Here's a quick graph of the normal cache usage on 8/21/12. X axis is size in KB, Y axis is client count.



I think the usage will go up as more web-based games played. The size of the cache after

launching From Dust was about 3 or 4 MB.

Code Structure

Main Classes

`program_cache` - This is the base program cache class, which holds the status information and has virtual methods for saving/loading the binary
`memory_program_cache` - This is an in-memory program cache, no disk backend
`program_cache_lru_helper` - This is a utility class to facilitate the lru policy

Tests

ProgramCacheLruHelper

- LRU eviction order w/o reuse
- LRU eviction order w/ reuse
- clear works correctly
- peek/pop work correctly (integrated into the order tests)

ProgramCache

- Compilation status storage, make sure key is copied
- Compilation unknown on source change
- Linked status storage, make sure key is copied
- Link unknown on source change for both vertex and fragment source
- Eviction w/o shader reuse
- Eviction w/ shader reuse
- Clear works correctly

MemoryProgramCache

- Correct gl calls on binary save, linked status correct
- Correct gl calls on load, attrib and uniform maps set correctly, and binary saved is the binary returned
- Different source doesn't return same program
- Different attrib mapping doesn't return same program
- Proper eviction on save when cache is full

ProgramManager

- Calls glCompile on compilation cache miss + sets status to success
- Does not set compilation status on error during compilation
- Does not compile when compilation status is success

- Links program on cache miss
- Compiles uncompiled reused shaders on cache miss + links
- Correct program cache hit (calls LoadLinkedProgram, doesn’t link or cache again)
- Compiles and links if LoadLinkedProgram returns false