`tangl` and `mangl`

Threaded OpenGL API Dispatch

Alexander Monakov

amonakov@ispras.ru

Institute for System Programming of Russian Academy of Sciences

X.Org Developers Conference, October 10, 2014

Talking Points

Threaded GL API dispatch

- Concept
- Implementation details
- Making it fast
- Making it faster
- Missing relevant features in OpenGL

Note the Footnote

Application makes API calls

- Store function IDs and arguments in a buffer
- Don't execute the actual function
- Return control to the application
- Have a secondary thread do the real work
  - Retrieve function IDs and args from the buffer
  - Execute the actual function
- . . . as long as postponing the side effects is fine

"Threaded"[1] refers to offloading the work to another thread

[1] "threaded dispatch" usually refers to a certain design of an interpreter loop

Not That Easy

You can't *naively* make an API call asynchronously when it

- ... returns a value
- ... dereferences pointers into application memory
  - pointer given in arguments
  - pointer escaped via previous calls
  - ... unless async behavior allowed by the spec (`glArrayElement`)
- ... specified to have a synchronizing effect (`glFinish`)
- ... just better be synchronous (`glXSwapBuffers`)

Solutions:

- Synchronize (stall until the secondary thread catches up)
  big hammer, always works
- If API call needs a const pointer to a small array, just copy it
- Use API semantics to your advantage in other ways

No Silver Bullet

Won't buy you anything if the application is

- . . . 100% GPU bound
- . . . 100% CPU bound *all outside the driver*
  not helping the bottleneck
- . . . 100% CPU bound *all in the driver*
  moving the bottleneck to another thread

Ideal case:

- CPU bound, 50% in GL driver on the critical path
- No API calls causing synchronization stalls

Ideal theoretical speedup is "about 2x"

Not Exactly New

Been done before:

- NVIDIA: `__GL_THREADED_OPTIMIZATIONS`, 2012
  (years after Windows driver got "Multicore Optimizations")
- Mesa: `anholt/glthread-5` branch

What's going to be new here

- Standalone, vendor-independent
- Will come with a stall profiler

Principles of Operation

To perform threaded offload, one needs:

- Secondary worker threads
- Mechanism to pass API call args
- Synchronization mechanism
- Producer/consumer stubs for each GL entrypoint

Workers

One worker thread for each application thread touching GL/GLX

- 1–1 producer-consumer correspondence
- Never touch libGL from original application threads
- When to spawn:

  In GLX calls, spawn worker if doesn't exist yet

  In GL calls, no need to care
- When to cleanup:

  when the corresponding application thread exits

  (using `pthread_key_create`)

Tried and discarded another approach:

- Spawn one worker per active context
- Turns out NVIDIA driver gets slower with `pthread_mutex_unlock` high in perf profiles
- Presumably attempts to protect internal datastructures with mutexes when mulithreaded, even with one context
- Exact logic is unclear
- Need to dlopen NVIDIA libGL from worker thread as well!

Workers

One worker thread for each application thread touching GL/GLX

- 1–1 producer-consumer correspondence
- Never touch libGL from original application threads
- When to spawn:

  In GLX calls, spawn worker if doesn't exist yet

  In GL calls, no need to care
- When to cleanup:

  when the corresponding application thread exits

  (using `pthread_key_create`)

Tried and discarded another approach:

- Spawn one worker per active context
- Turns out NVIDIA driver gets slower with `pthread_mutex_unlock` high in perf profiles
- Presumably attempts to protect internal datastructures with mutexes when mulithreaded, even with one context
- Exact logic is unclear
- Need to dlopen NVIDIA libGL from worker thread as well!

Buffers

One ring buffer for each producer-consumer pair

- Size/align 4MB/4MB — get a hugepage if lucky
- Data layout just natural:
  - Function ID followed by arguments
  - Variable-length arrays preceded by length
  - Primitive types aligned to their size
- Prescribe maximum argument size (e.g. 16K)
  - Useful to keep small `glBufferSubData` calls async
  - For larger sizes, make a synchronous call without copying

Synchronization

Threads occasionally need to suspend:

- Consumer: ring buffer empty
- Producer: ring buffer may overflow on next call
- Producer: when making a synchronous call

When one suspends, the other needs to wake it

Approach taken:

- For producer and consumer, maintain
  - Current pointer into ring buffer
  - "Suspended" flag
- Suspend/wakeup:
  - Futex operations on pointers
  - Fits almost[2] perfectly
  - Consumer: `sched_yield()` a few times before suspending

[2] needs endian-dependent hacks

Stubs

Need two stubs for each GL API entrypoint

- Almost 3000 functions (counting all extensions)
- Must have automatic codegen

Need formal API specs to do codegen

- Old GL specs: incomplete, deprecated
- New GL specs
  - XML
  - Not informative enough
- APITrace specs: very nice

Stubs

```
Function(ASYNC, Void, glVertex2f, ((GLfloat, x), (GLfloat, y)))
```

```
void glVertex2f (GLfloat x, GLfloat y)
{
  PFUNC(glVertex2f);
  PUT(x);
  PUT(y);
  PDONE;
}

static void worker_glVertex2f(void)
{
  GLfloat x;
  GLfloat y;
  CFUNC(glVertex2f);
  GET(x);
  GET(y);
  CDONE;
  CNEXT(glVertex2f)(x, y);
}
```

Stubs

```
Function(ASYNC, Void, glVertex2f, ((GLfloat, x), (GLfloat, y)))

void glVertex2f (GLfloat x, GLfloat y)
{
  PFUNC(glVertex2f);
  PUT(x);
  PUT(y);
  PDONE;
}

static void worker_glVertex2f(void)
{
  GLfloat x;
  GLfloat y;
  CFUNC(glVertex2f);
  GET(x);
  GET(y);
  CDONE;
  CNEXT(glVertex2f)(x, y);
}
```

Producer Stub Assembly

```
glVertex2f:
# Get thread-specific context (cheat: IE TLS)
        movq    current@gottpoff(%rip), %rax
        movq    %fs:(%rax), %rdi
# Get ring buffer pointer
        movq    256(%rdi), %rsi
# Save Function ID
        movl    $216, (%rsi)
# Advance ring buffer pointer
        leaq    16(%rsi), %rdx
# Save args
        movss   %xmm0, 4(%rsi)
        movss   %xmm1, 8(%rsi)
# Store ring buffer pointer and handle overflow
        jmp     producer_advance
```

Consumer Stub Assembly

```
worker_glVertex2f:
# Load args
        movss   4(%rbx), %xmm0
        movss   8(%rbx), %xmm1
# Advance ring buffer pointer
        leaq    16(%rbx), %rbx
# Jump to vendor libGL
        jmp     *%rax
```

Workers are very small thanks to custom ABI.
Use return register (`rax`) for driver function pointer
Use callee-saved registers (`rbx`, `r15`) for

- Ring buffer pointer
- Current context data (very rarely needed)

Only a matter of 3 global register vars (GCC extension)

Stall Profiler

Producer side can output stall timing statistics:

```
41   fps
92.1 syncs per frame
0    waits per frame (due to overflow)

sync: 78.2%
wait: 0%

glXSwapBuffers:            41      88.6%
glGetIntegerv:             1447    6.85%
glCheckFramebufferStatus:  1406    2.82%
glMapBufferRange:          592     1.02%
glBufferData:              143     0.326%
glTexImage3D:              5       0.124%
glGetError:                41      0.057%
```

Fake It Till You Make It

Fast offload not useful if you sync all the time

- Chances are, you will. . .
- . . . unless the application was heavily optimized with driver threading in mind
- Want some way to forgo syncs when possible

Ways to avoid thread syncs:

- Guess and hope for the best
  - `glGetError() {return GL_NO_ERROR;}`
  - `glCheckFramebufferStatus()` — likewise
- Try to track some GL state
  - Intercept `glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo)`
  - Answer `glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING)` queries

Fake It Till You Make It

Fast offload not useful if you sync all the time

- Chances are, you will. . .
- . . . unless the application was heavily optimized with driver threading in mind
- Want some way to forgo syncs when possible

Ways to avoid thread syncs:

- Guess and hope for the best
  - `glGetError() {return GL_NO_ERROR;}`
  - `glCheckFramebufferStatus()` — likewise
- Try to track some GL state
  - Intercept `glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo)`
  - Answer `glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING)` queries

Fake It Till You Make It

Fast offload not useful if you sync all the time

- Chances are, you will. . .
- . . . unless the application was heavily optimized with driver threading in mind
- Want some way to forgo syncs when possible

Ways to avoid thread syncs:

- Guess and hope for the best
  - `glGetError() {return GL_NO_ERROR;}`
  - `glCheckFramebufferStatus()` — likewise
- Try to track some GL state
  - Intercept `glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo)`
  - Answer `glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING)` queries

Duck Mapping

```
glMapBufferRange(target, offset, length,
  GL_MAP_WRITE_BIT | GL_MAP_UNSYNCHRONIZED_BIT)
```

shouldn't sync, right?

- Give `data = malloc(length)` to the application
- Remember `(offset, length, data)` for `target`
- When application calls `glUnmapBuffer`:
  - `glBufferSubData(target, offset, length, data)`
  - `free(data)`

Only do it if `length` is small enough

Duck Mapping

```
glMapBufferRange(target, offset, length,
  GL_MAP_WRITE_BIT | GL_MAP_UNSYNCHRONIZED_BIT)
```

shouldn't sync, right?

- Give `data = malloc(length)` to the application
- Remember `(offset, length, data)` for `target`
- When application calls `glUnmapBuffer`:
  - `glBufferSubData(target, offset, length, data)`
  - `free(data)`

Only do it if `length` is small enough

Duck Mapping

```
glMapBufferRange(target, offset, length,
  GL_MAP_WRITE_BIT | GL_MAP_UNSYNCHRONIZED_BIT)
```

shouldn't sync, right?

- Give `data = malloc(length)` to the application
- Remember `(offset, length, data)` for target
- When application calls `glUnmapBuffer`:
  - `glBufferSubData(target, offset, length, data)`
  - `free(data)`

Only do it if `length` is small enough

Tangle and Mangle

Contradicting goals

- Threaded dispatch
  - Simple 1:1 call mapping
  - Low overhead
- Sync avoidance:
  - Do some tracking — not free
  - Call transformations — plenty of room for error

Completely separate in two libraries:

- `tangl` — pure threaded dispatch
  - Simple, correct, fast
  - Good enough for "well-behaved" applications
- `mangl` — call transformation
  - All kinds of questionable hacks to sync avoidance
  - Plenty of room for error
  - Ability to deviate from GL spec (should be configurable)
  - Adds overhead

Tangle and Mangle

Contradicting goals

- Threaded dispatch
  - Simple 1:1 call mapping
  - Low overhead
- Sync avoidance:
  - Do some tracking — not free
  - Call transformations — plenty of room for error

Completely separate in two libraries:

- `tangl` — pure threaded dispatch
  - Simple, correct, fast
  - Good enough for “well-behaved” applications
- `mangl` — call transformation
  - All kinds of questionable hacks to sync avoidance
  - Plenty of room for error
  - Ability to deviate from GL spec (should be configurable)
  - Adds overhead

Missing Pieces

Enabling asynchronous memory access in the driver
No way in core GL to say:

- *Here's a memory range in the application address space*
- *I promise I won't modify or unmap it*
- *Therefore the driver may access it asynchronously*

Example use case:

- `mmap` a resource file
- `glTexImage` from mmap'ed range
- `glFenceSync`
- do something else
- `glClientWaitSync`
- `munmap`

or `glReadPixels`/`glGetBufferSubData` into a prescribed buffer
Actually this was done as extensions:

- `GL_SGIX_async`, 1998
- `GL_NV_pixel_data_range`, 2002

Why not in main spec?

Missing Pieces

Enabling asynchronous memory access in the driver

No way in core GL to say:

- *Here's a memory range in the application address space*
- *I promise I won't modify or unmap it*
- *Therefore the driver may access it asynchronously*

Example use case:

- `mmap` a resource file
- `glTexImage` from mmap'ed range
- `glFenceSync`
- do something else
- `glClientWaitSync`
- `munmap`

or `glReadPixels`/`glGetBufferSubData` into a prescribed buffer

Actually this was done as extensions:

- GL_SGIX_async, 1998
- GL_NV_pixel_data_range, 2002

Why not in main spec?

Missing Pieces

Enabling asynchronous memory access in the driver

No way in core GL to say:

- *Here's a memory range in the application address space*
- *I promise I won't modify or unmap it*
- *Therefore the driver may access it asynchronously*

Example use case:

- `mmap` a resource file
- `glTexImage` from mmap'ed range
- `glFenceSync`
- do something else
- `glClientWaitSync`
- `munmap`

or `glReadPixels`/`glGetBufferSubData` into a prescribed buffer

Actually this was done as extensions:

- `GL_SGIX_async`, 1998
- `GL_NV_pixel_data_range`, 2002

Why not in main spec?

Missing Pieces II: Fence Callbacks

No way to register a user function for fence completion

- Callbacks are not a foreign concept in GL (debug output)
- Without callbacks, `glClientWaitSync` needs a complete synchronization stall in threaded dispatch

More oddity in GL fence objects:

- `glFenceSync` conflates object creation and GPU operation

Suitable for `GL_ARB_sync2`?

Thank you!

Backup/extra slides follow

Safety First

You might not want this in Mesa:

- `libpthread` is required to spawn worker threads
- loading `libpthread` switches all mutexes from no-op to real
- on FreeBSD `libpthread` cannot be dynamically loaded
- not necessarily a good idea to absorb everything

Higher Hanging Fruit

In-driver implementation can do a bit better:

- Skip one level of GL dispatch (direct/indirect) in workers
- Skip PLT for API calls in the worker
- Tune code layout for I-cache locality
- Do some state tracking up front (and reuse tracking code)

Pie in the Sky

Interesting potential developments based on fast threaded dispatch layer:

- Low-overhead GL tracing
- Out-of-process GL
- tee dispatch