Performance tuning applications for Intel GEN Graphics for Linux* and Google* Chrome OS

Ian Romanick <ian.d.romanick@intel.com>
July 23rd, 2013
A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice.

All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Intel processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Any code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user.

Intel product plans in this presentation do not constitute Intel plan of record product roadmaps. Please contact your Intel representative to obtain Intel's current plan of record product roadmaps.

Performance claims: Software and workloads used in performance tests may have been optimized for performance only on Intel® microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.Intel.com/performance

Intel, Intel Inside, the Intel logo, Centrino, Intel Core, Intel Atom, Pentium, and Ultrabook are trademarks of Intel Corporation in the United States and other countries.
Introduction

Agenda

- Overview of Intel GEN graphics architecture
- General optimization tips
- Tools to help along the way
Linux Supported APIs

OpenGL ES 3.0
- Fully supported on Sandybridge, Ivybridge, and Haswell
- One of only two certified desktop implementations!

OpenGL 3.1 + many extensions
- No GL_ARB_compatibility extension

OpenGL 3.2 coming later this year
- Core profile only
GEN Graphics Architecture

Unified memory architecture
- Most generations have shared CPU / GPU cache

Unified shader execution units
- More on this later

HiZ, fast depth clears, fast color clears, etc.
- Hardware optimizations you expect in a desktop part

NOT A TILED RENDERER!
Data Flow Inside the GPU
Data Flow Inside the GPU

Vertex fetcher pulls data from memory to URB.
Data Flow Inside the GPU

VS pulls from URB, writes results back to same location.
Data Flow Inside the GPU

GS pulls from URB, writes results back to new locations.
Data Flow Inside the GPU

Clipper pulls from URB, passed to SF via internal FIFO
Data Flow Inside the GPU

Strips-and-fans operates similarly.
Data Flow Inside the GPU

FS pulls from internal FIFO, writes to color calculator.
Data Flow Inside the GPU

Most data comes from the URB
- Non-UBO uniforms
- Vertex, geometry, and tessellation inputs
- “Pushed” to EU registers at shader start-up

Most data goes out to the URB
- Vertex, geometry, and tessellation outputs

Textures, TexBOs, and UBOs come from the sampler

*All in-flight primitives share the fixed-size URB*
Unified Shader Cores

Shader execution units are shared, but...

- Vertex and geometry shaders execute in AoS mode
  - Vertex shaders dispatch 2 vertices at a time
- Fragment shaders execute in SoA mode
  - Depending on register usage, fragment shaders dispatch either 8 or 16 fragments at a time
  - Fragment shader uses channel-masking for per-fragment divergent flow-control
More Information

Hardware documentation publicly available:
https://01.org/linuxgraphics/documentation

Driver source code publicly available:
http://cgit.freedesktop.org/mesa/mesa/
Introduction

Agenda

- Overview of Intel GEN graphics architecture
  - General optimization tips
  - Tools to help along the way
Memory Management

Put everything in buffer object

- `glBufferData` is `malloc` for everything going to the GPU
- `glMapBuffer` and `glMapBufferRange` give access to the data
Memory Management

Use `glMapBufferRange` for asynchronous subrange mappings

- Use multiple BOs to avoid buffer-wrap synchronization
- Or use `glBufferData` to orphan the old backing store

Avoid staging buffers

- Do not write data to one BO, then `glCopySubBufferSubData` to another

Avoid `glBufferSubData`
Channel Masked Execution

Fragment dispatches two 4x4 “quads” together

- Uniform flow-control within the 8 fragment group is cheap
- Non-uniform flow-control within the group is expensive
  - Both branches of an if-then-else are executed for all fragments, etc.

Two quads dispatched together as an 8 fragment group
Channel Masked Execution

16-wide dispatches two 8 fragment groups together

- Instructions are fetched and decoded once, but executed twice
- Divergent flow-control between groups isn't terribly expensive
  - Uniform flow-control within a group won't branch, but the instructions don't execute either
Use `textureOffset` and Friends

Post-process filters often need all pixels in a small neighborhood
Use `textureOffset` and Friends

Post-process filters often need all pixels in a small neighborhood

- **Do:**
  ```c
  void main()
  {
      color = (texture(s, tc) +
        textureOffset(s, tc, ivec2(0, 1)) +
        ...
        textureOffset(s, tc, ivec2(7, 7))) / N;
  }
  ```

  - Less register usage
  - Fewer instructions
  - Less URB usage

- **Do not:**
  ```c
  uniform vec2 pixel_offsets[N-1];
  void main()
  {
      vec4 c = texture(s, tc);
      for (int i = 0; i < pixel_offsets.length(); i++)
          c += texture(s, tc + pixel_offsets[i]);
      color = c / N;
  }
  ```

  ```c
  in vec2 pixel_centers[N];
  void main()
  {
      color = (texture(s, pixel_centers[0])
        ...
        texture(s, pixel_centers[N-1])) / N;
  }
  ```
Use textureOffset and Friends

Post-process filters often need all pixels in a small neighborhood

- Use textureOffset to get the neighboring pixels
- OpenGL 3.0+ or OpenGL ES 3.0

http://www.opengl.org/sdk/docs/manGLsl/xhtml/textureOffset.xml
http://www.opengl.org/sdk/docs/manGLsl/xhtml/texelFetchOffset.xml
Instruction Throughput

Generally, throughput is 1 instruction every 2 cycles

- Simple instructions have 14 cycle latency
- Complex math (rcp, exp2, log2, rsq, sqrt, sin, cos) have 16 cycle latency
- pow has 24 cycle latency
- Texture access has ~140 cycle latency for a cache hit, ~700 otherwise
  - Issue rate is ~18 cycles
  - UBO and TexBO accesses are approximately the same
  - Compiler tries really hard to eliminate redundant UBO accesses
Compact Varyings

Declare varyings to be only as large as necessary

- Do:
  ```
  out vec2 tc;
  void main()
  {
    tc = ...
  }
  ```
  - Fewer VS instructions
  - Fewer FS instructions
  - Less URB space

- Do not:
  ```
  out vec4 tc;
  void main()
  {
    tc = vec4(..., 0, 1);
  }
  ```
Flat Varyings

Declare per-primitive varyings as flat

- **Do:**

```cpp
out vec2 tc;
flat out vec2 other;
void main()
{
    tc = vec4(…);
    other = vec2(constant_a, constant_b);
}
```

- **Do not:**

```cpp
out vec4 tc;
void main()
{
    tc = vec4(…, constant_a, constant_b);
}
```

- Fewer FS instructions
Packing Varyings

The compiler will pack varyings, so organize them naturally

- **Do:**
  ```cpp
class vec3
  class float

  void main()
  {
    normal = ...;
    intensity = ...;
  }
  ```

- **Do not:**
  ```cpp
  out vec4 data;
  void main()
  {
    ...data = vec4(normal, intensity);
  }
  ```
Introduction

Agenda

✔ Overview of Intel GEN graphics architecture
✔ General optimization tips
  ▪ Tools to help along the way
Driver Debug Output

Environment variables control extra information from the driver:

```
INTEL_DEBUG=... ./my_application
```

- Some of this information will also be available via `GL_ARB_debug_output` later this year

Higher-level GLSL compiler output is available too:

```
MESA_GLSL=dump ./my_application
```
Performance Warnings

For performance warnings: `INTEL_DEBUG=perf`

- Synchronization on buffer mappings
- Synchronization on occlusion query results
- State-based shader recompiles
- Missing 16-wide execution for fragment shader
- Missing various driver fast-paths
- ...
- Currently about 70 different warnings
Shader Assembly Dumps

All shader targets can be dumped

- Vertex shader assembly: `INTEL_DEBUG=vs`
- Fragment shader assembly: `INTEL_DEBUG=fs`
- Can combine multiple options in comma-separated list
- Also dumps shaders generated by the driver
  - e.g., for `glClear`, `glBlitFramebuffer`, etc.
Shader Assembly Dumps

$ INTEL_DEBUG=vs ./my_application

GLSL IR for native vertex shader 3:

```glsl
(declare (shader_in) vec4 gl_Vertex)
(declare (shader_out) vec4 gl_Position)
(declare (shader_out) vec4 packed:texcoord)

function main

(parameters

(assign (xyzw) (var_ref gl_Position) (var_ref gl_Vertex))
(assign (xy) (var_ref packed:texcoord) (expression vec2 * (expression vec2 + (swiz xy (var_ref gl_Vertex)) {constant float (1.000000}) {constant float (0.500000)})}
)

vec4 estimated execution time: 44 cycles

Native code for vertex shader 3:

```asm
0x00000000: add(8) g5<1>.xyF g1<4,4,1>.xyyyF 1F {align16 WE_normal 1Q};
0x00000010: mul(8) g3<1>.xyF g5<4,4,1>.xyyyF 0.5F {align16 WE_normal 1Q};
0x00000020: mov(8) g114<1>D 0D {align16 WE_normal 1Q};
0x00000030: mov(8) g115<1>F g1<4,4,1>F {align16 WE_normal 1Q};
0x00000040: mov(8) g116<1>F g3<4,4,1>F {align16 WE_normal 1Q};
0x00000050: mov(8) g113<1>UD g0<4,4,1>UD {align16 WE_all 1Q};
0x00000060: or(1) g113.5<1>UD g0.5<0,1,0>UD 0x00000000UD {align1 WE_all};
0x00000070: send(8) null g113<4,4,1>F
urb 0 urb_write used complete mlen 5 rien 0 {align16 WE_normal 1Q EOT};
```
Shader Assembly Dumps

$ INTEL_DEBUG=vs ./my_application
GLSL IR for native vertex shader 3:
{
    (declare (shader_in ) vec4 gl_Vertex)
    (declare (shader_out ) vec4 gl_Position)
    (declare (shader_out ) vec4 packed:texcoord)
    (function main
        (signature void
            (parameters
                (assign (xyzw) (var_ref gl_Position) (var_ref gl_Vertex) )
                (assign (xy) (var_ref packed:texcoord) (expression vec2 * (expression vec2 + (swiz xy (var_ref gl_Vertex)) (constant float (1.000000)) ) (constant float (0.500000)) ) )
            )
        )
    )
}

vec4 estimated execution time: 44 cycles

Native code for vertex shader 3:

0x00000000: add(8)   g5<1>.xyF   g1<4,4,1>.xyyyF 1F  { align16 WE_normal 1Q };
0x00000010: mul(8)   g3<1>.xyF   g5<4,4,1>.xyyyF 0.5F  { align16 WE_normal 1Q };
0x00000020: mov(8)   g114<1>D   0D  { align16 WE_normal 1Q };
indices, point width, clip flags
0x00000030: mov(8)   g115<1>F   g1<4,4,1>F  { align16 WE_normal 1Q };
packed:texcoord
0x00000040: mov(8)   g116<1>F   g3<4,4,1>F  { align16 WE_normal 1Q };
URB write
0x00000050: mov(8)   g113<1>UD  g0<4,4,1>UD  { align16 WE_all 1Q };
0x00000060: or(1)    g113.5<1>UD  g0.5<0,1,0>UD  0x00000000UD  { align1 WE_all };
0x00000070: send(8)  null   g113<4,4,1>F  { align16 WE_normal 1Q EOT };
urb 0 urb_write used complete mlen 5 rien 0  { align16 WE_normal 1Q EOT };

Shader Assembly Dumps

$ INTEL_DEBUG=vs ./my_application
GLSL IR for native fragment shader 0:

(declare (shader_out ) vec4 gl_FragColor)
(declare (shader_in ) vec4 gl_Color)
(function main
  (signature void
    (parameters
      ))
  (assign (xyzw) (var_ref gl_FragColor) (var_ref gl_Color) )
))

fs8 estimated execution time: 22 cycles
fs16 estimated execution time: 30 cycles
Native code for fragment shader 0 (8-wide dispatch):
START B0
  (declare (shader_in ) vec4 gl_Color)
  0x00000000: pln.sat(8) g116<1>F g5.4<0,1,0>F g2<8,8,1>F { align1 WE_normal 1Q }
  0x00000010: pln.sat(8) g115<1>F g5<0,1,0>F g2<8,8,1>F { align1 WE_normal 1Q }
  0x00000020: pln.sat(8) g114<1>F g4.4<0,1,0>F g2<8,8,1>F { align1 WE_normal 1Q }
  0x00000030: pln.sat(8) g113<1>F g4<0,1,0>F g2<8,8,1>F { align1 WE_normal 1Q }
  FB write target 0
  0x00000040: sendc(8) null g113<8,8,1>F render ( RT write, 0, 4, 12) mlen 4 rlen 0 { align1 WE_normal 1Q EOT }
END B0

Native code for fragment shader 0 (16-wide dispatch):
START B0
  (declare (shader_in ) vec4 gl_Color)
  0x00000040: pln.sat(16) g119<1>F g7.4<0,1,0>F g2<8,8,1>F { align1 WE_normal 1H }
  0x00000050: pln.sat(16) g117<1>F g7<0,1,0>F g2<8,8,1>F { align1 WE_normal 1H }
  0x00000060: pln.sat(16) g115<1>F g6.4<0,1,0>F g2<8,8,1>F { align1 WE_normal 1H }
  0x00000070: pln.sat(16) g113<1>F g6<0,1,0>F g2<8,8,1>F { align1 WE_normal 1H }
  FB write target 0
  0x00000080: sendc(16) null g113<8,8,1>F render ( RT write, 0, 0, 12) mlen 8 rlen 0 { align1 WE_normal 1H EOT }
END B0
Shader Assembly Dumps

$ INTEL_DEBUG=vs ./my_application

GLSL IR for native fragment shader 0:

{  
  (declare (shader_out) vec4 gl_FragColor)
  (declare (shader_in) vec4 gl_Color)
  (function main
   (signature void
    (parameters
     )
    
    (assign (xyzw) (var_ref gl_FragColor) (var_ref gl_Color) )
  ))
}

fs8 estimated execution time: 22 cycles
fs16 estimated execution time: 30 cycles

Native code for fragment shader 0 (8-wide dispatch):
START B0
  (declare (shader_in) vec4 gl_Color)

0x00000000: pln.sat(8) g116<1>F g5.4<0,1,0>F g2<8,8,1>F ( align1 WE_normal 1Q );
0x00000010: pln.sat(8) g115<1>F g5<0,1,0>F g2<8,8,1,F> ( align1 WE_normal 1Q );
0x00000020: pln.sat(8) g114<1>F g4.4<0,1,0,F> g2<8,8,1,F> ( align1 WE_normal 1Q );
0x00000030: pln.sat(8) g113<1,F> g4<0,1,0,F> g2<8,8,1,F> ( align1 WE_normal 1Q );
  FB write target 0
0x00000040: sendc(8) null g113<8,8,1,F>
  render ( RT write, 0, 4, 12) mlen 4 rlen 0 ( align1 WE_normal 1Q EOT );
END B0

Native code for fragment shader 0 (16-wide dispatch):
START B0
  (declare (shader_in) vec4 gl_Color)

0x00000040: pln.sat(16) g119<1,F> g7.4<0,1,0,F> g2<8,8,1,F> ( align1 WE_normal 1H );
0x00000050: pln.sat(16) g117<1,F> g7<0,1,0,F> g2<8,8,1,F> ( align1 WE_normal 1H );
0x00000060: pln.sat(16) g115<1,F> g6.4<0,1,0,F> g2<8,8,1,F> ( align1 WE_normal 1H );
0x00000070: pln.sat(16) g113<1,F> g6<0,1,0,F> g2<8,8,1,F> ( align1 WE_normal 1H );
  FB write target 0
0x00000080: sendc(16) null g113<8,8,1,F>
  render ( RT write, 0, 0, 12) mlen 8 rlen 0 ( align1 WE_normal 1H EOT );
END B0
Execution times for shaders can be output: INTEL_DEBUG=shader_time

- Occasionally the times emitted are invalid
  - Some hardware conditions cause the counters to get reset
Shader Execution Timings

```bash
$ INTEL_DEBUG=shader_time ./my_application
No shader time collected yet

<table>
<thead>
<tr>
<th>type</th>
<th>ID</th>
<th>cycles spent</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs16</td>
<td>3:</td>
<td>291191730</td>
<td>0.29 Gcycles</td>
</tr>
<tr>
<td>fs8</td>
<td>3:</td>
<td>2265333642</td>
<td>2.27 Gcycles</td>
</tr>
<tr>
<td>vs</td>
<td>3:</td>
<td>10317260118</td>
<td>10.32 Gcycles</td>
</tr>
<tr>
<td>total fs16</td>
<td></td>
<td>2265333642</td>
<td>2.27 Gcycles</td>
</tr>
<tr>
<td>total fs8</td>
<td></td>
<td>10317260118</td>
<td>10.32 Gcycles</td>
</tr>
<tr>
<td>total vs</td>
<td></td>
<td>291191730</td>
<td>0.29 Gcycles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>ID</th>
<th>cycles spent</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs16</td>
<td>3:</td>
<td>619003966</td>
<td>0.62 Gcycles</td>
</tr>
<tr>
<td>fs8</td>
<td>3:</td>
<td>4786157361</td>
<td>4.79 Gcycles</td>
</tr>
<tr>
<td>vs</td>
<td>3:</td>
<td>22789663885</td>
<td>22.79 Gcycles</td>
</tr>
<tr>
<td>total vs</td>
<td></td>
<td>22789663885</td>
<td>22.79 Gcycles</td>
</tr>
<tr>
<td>total fs8</td>
<td></td>
<td>4786157361</td>
<td>4.79 Gcycles</td>
</tr>
<tr>
<td>total fs16</td>
<td></td>
<td>619003966</td>
<td>0.62 Gcycles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>ID</th>
<th>cycles spent</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs16</td>
<td>3:</td>
<td>934151049</td>
<td>0.93 Gcycles</td>
</tr>
<tr>
<td>fs8</td>
<td>3:</td>
<td>7312214004</td>
<td>7.31 Gcycles</td>
</tr>
<tr>
<td>vs</td>
<td>3:</td>
<td>36255971577</td>
<td>36.26 Gcycles</td>
</tr>
<tr>
<td>total vs</td>
<td></td>
<td>36255971577</td>
<td>36.26 Gcycles</td>
</tr>
<tr>
<td>total fs8</td>
<td></td>
<td>7312214004</td>
<td>7.31 Gcycles</td>
</tr>
<tr>
<td>total fs16</td>
<td></td>
<td>934151049</td>
<td>0.93 Gcycles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>ID</th>
<th>cycles spent</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs16</td>
<td>3:</td>
<td>1250458485</td>
<td>1.25 Gcycles</td>
</tr>
<tr>
<td>fs8</td>
<td>3:</td>
<td>9670458850</td>
<td>9.67 Gcycles</td>
</tr>
<tr>
<td>vs</td>
<td>3:</td>
<td>48742313487</td>
<td>48.74 Gcycles</td>
</tr>
<tr>
<td>total vs</td>
<td></td>
<td>48742313487</td>
<td>48.74 Gcycles</td>
</tr>
<tr>
<td>total fs8</td>
<td></td>
<td>9670458850</td>
<td>9.67 Gcycles</td>
</tr>
<tr>
<td>total fs16</td>
<td></td>
<td>1250458485</td>
<td>1.25 Gcycles</td>
</tr>
</tbody>
</table>
```
Nobody wants to sprinkle `glGetError` all over the place

- **Use GL_ARB_debug_output**
- **MESA_DEBUG=verbose** logs error information to console
- Set a breakpoint in the driver's error function!
  - Errors are emitted by `_mesa_error`
  - Framebuffer incompleteness is marked by `fbo_incomplete` and `att_incomplete`
  - Can also break at `mesa_Function` (e.g., `mesa_BindBuffer`) to break on a GL function
$ gdb ./my_application

Reading symbols from my_application...done.
(gdb) b _mesa_error
Function ".mesa_error" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (_mesa_error) pending.
(gdb) r
Starting program: my_application
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, _mesa_error (ctx=0x620500, error=1282, fmtString=0x7ffff4a9a930 "glUseProgramStages(program wasn't linked with the PROGRAM_SEPARABLE flag")
    at ../../../src/mesa/main/errors.c:935
935      debug_get_id(&error_msg_id);
Missing separate debuginfos, use: debuginfo-install expat-2.1.0-4.fc18.x86_64 glibc-2.16-31.fc18.x86_64 libgcc-4.7.2-8.fc18.x86_64 libstdc++-4.7.2-8.fc18.x86_64 libstdc++-4.7.2-8.fc18.x86_64 mesa-libGLU-9.0.0-1.fc18.x86_64 (gdb)
Debugging GL Errors

…
(gdb) bt
#0  _mesa_error (ctx=0x620500, error=1282, fmtString=0x7ffff4a997d0 "glUseProgramStages(program wasn't linked with the PROGRAM_SEPARABLE flag") at ../../../src/mesa/main/errors.c:935
#1  0x00007fff486d70c in _mesa_UseProgramStages (pipeline=1, stages=1, program=2) at ../../../src/mesa/main/pipelineobj.c:331
#2  0x00000000000400db8 in main (argc=1, argv=0x7fffffffdeb8) at /home/idr/src/my_application/main.cpp:35
(gdb) up
#1  0x00007fff486d70c in _mesa_UseProgramStages (pipeline=1, stages=1, program=2) at ../../../src/mesa/main/pipelineobj.c:331
331  _mesa_error(ctx, GL_INVALID_OPERATION,
(gdb) list
326          "glUseProgramStages(program not linked"));
327          return;
328      }
329  
330  if (!shProg->SeparateShader) {
331      _mesa_error(ctx, GL_INVALID_OPERATION,
332          "glUseProgramStages(program wasn't linked with the "
333             "PROGRAM_SEPARABLE flag"));
334      return;
335  }
336  …
apitrace

Collect and replay traces of GL calls from your application

- Inspect the parameters to every GL call made
- Inspect the GL state at any call during replay
- View textures, framebuffers, etc.
- Collect traces on one system replay on another
- Trim traces to minimal size, submit with bug reports

http://apitrace.github.io/
Your Favorite CPU Profiler

Profile all the way into the driver code

- Determine when application state changes trigger expensive validation in the driver
- Determine where the driver is spending time

*An open-source driver is not a black box!*
Your Favorite CPU Profiler
Contact us!

Our driver developers are accessible and can help!

- We're regularly on irc.freenode.net #intel-gfx
- We have a mailing list
  http://lists.freedesktop.org/mailman/listinfo/mesa-dev
- Our bug tracker is public
  https://01.org/linuxgraphics/documentation/how-report-bugs-0
Join us!

Tweet Us and Win

- Tweet about Intel Open Source with #Siggraph2013 and #IntelSoftware
- First 500 who can show their tweet win shirts or Linux Penguin chocolate

Visit us at Intel booth #201 and Room #201 for demos and more!

- We are hiring, too!

http://www.intel.com/jobs