Performance tuning applications for Intel GEN Graphics for Linux* and SteamOS*

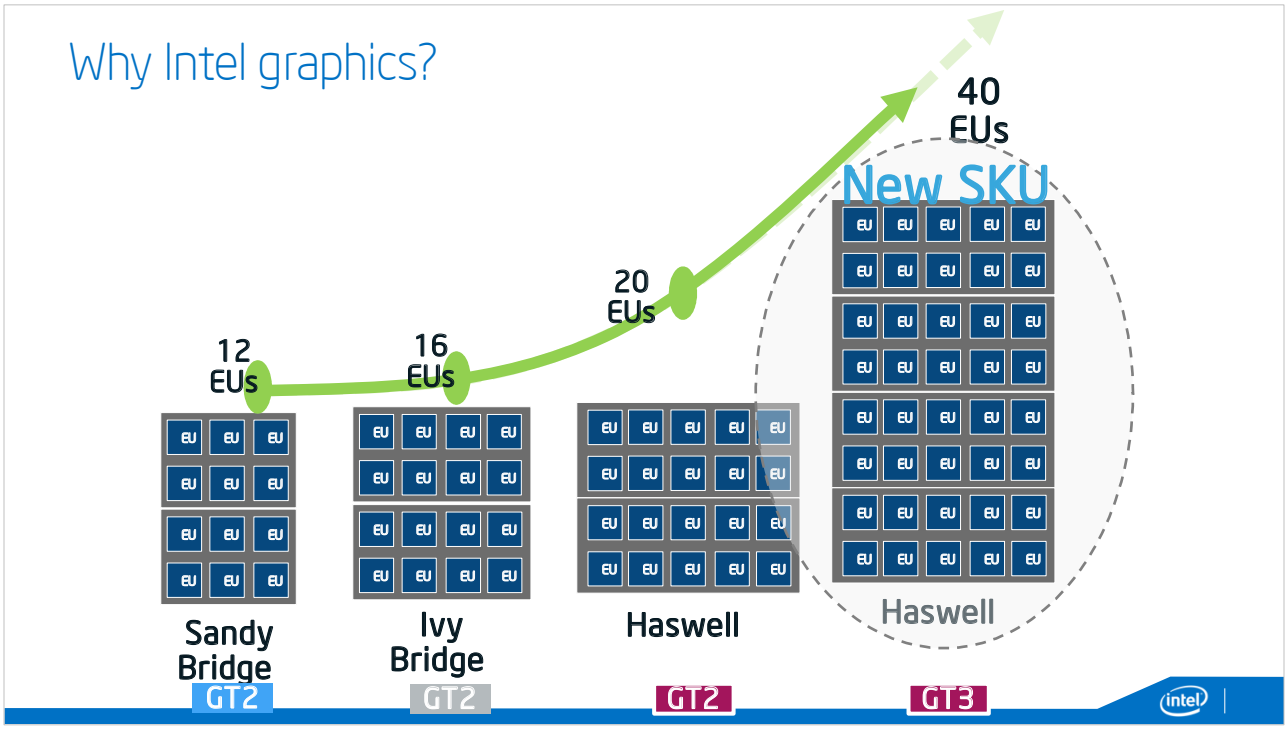
Ian Romanick <ian.d.romanick@intel.com>
January 15th, 2014

Legal

Introduction

Agenda

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


Why Intel graphics?
12 EUs
16 EUs
20 EUs
40 EUs
New SKU
Sandy Bridge
GT2
Ivy Bridge
GT2
Haswell
GT2
Haswell
GT3
intel

Gigabyte BRIX Pro

Component	Description
Chassis	Gigabyte* Brix Pro
Processor	Intel® CoreTM i7-4770R processor, 4 core, 3.2GHz running at 65W
Graphics	Intel® Iris(TM) Pro Graphics 5200 (codename: Haswell GT3e, with 128MB eDRAM)
Hard Disk Drive	Seagate* SSHD Hybrid 1TB + 8GB
Chipset	Intel® 8 Series Chipset Family
Memory	8GB (2X4GB) DDR3L 1600MHz

Gigabyte BRIX Pro

	Xbox One	PS4	Wii U	Brix "Pro"
				
Width (mm)	80	53	45	50
Depth (mm)	263	305	267	138
Height (mm)	343	275	172	115
Volume (l)	7.2	4.5	2.1	0.8
Performance (Tflops)	~1.3	~1.8	~0.4	~1.2

Open Source Driver Development

Actively supporting Linux since 2006

- OpenGL driver, X server, kernel, Wayland...

Large team spanning multiple contents

- North America, Europe, Asia, and Zealandia
- Not just a bunch of random Internet hackers

Linux Supported APIs

OpenGL ES 3.0

- Fully supported on Sandybridge, Ivybridge, and Haswell
- Support for next version of OpenGL ES planned for at least Haswell

OpenGL 3.3 + many extensions

- Core profile only

Linux Supported APIs

Texture compression

- S3TC - `GL_EXT_texture_compression_dxt1`, `GL_ANGLE_texture_compression_{dxt3,dxt5}`
  - Note that is `GL_EXT_texture_compression_s3tc` **not** on the list
- ETC1 - `GL_OES_compressed_ETC1_RGB8_texture`
  - Only on Baytrail for desktop OpenGL
  - Always enabled on ES, but textures may be decompressed
- ETC2 - OpenGL ES 3.0 or `GL_ARB_ES3_compatibility`
  - Same support as ETC1
- FXT1 (lol) - `GL_3DFX_texture_compression_FXT1`

The hardware can also do BPTC, but we haven't had developer requests for the feature yet.

ASTC is not supported in any shipping hardware.

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.

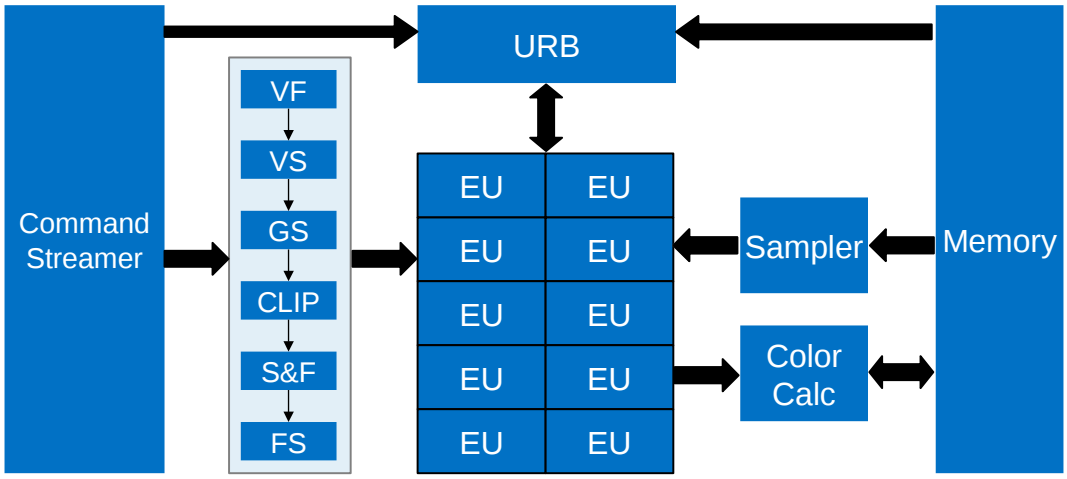
- Hardware optimizations you expect in a desktop part

NOT A TILED RENDERER!

Last-level cache (LLC) shared by CPU and GPU makes it efficient to generate data on the CPU to be consumed by the GPU.

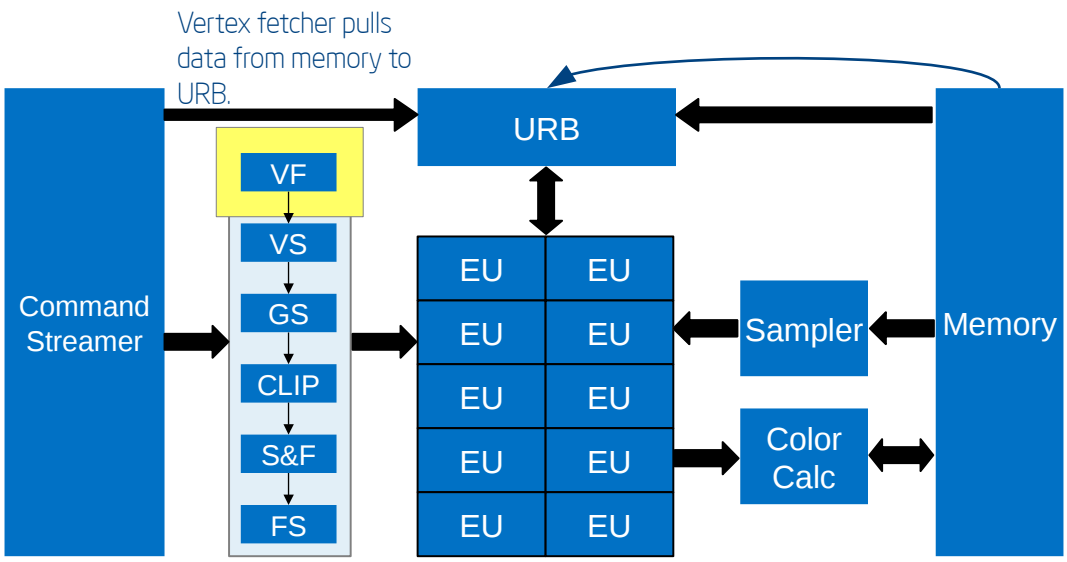
Mobile parts are generally tiled and UMA, and desktop parts are generally not tiled and not UMA. GEN is UMA but not tiled.

Data Flow Inside the GPU

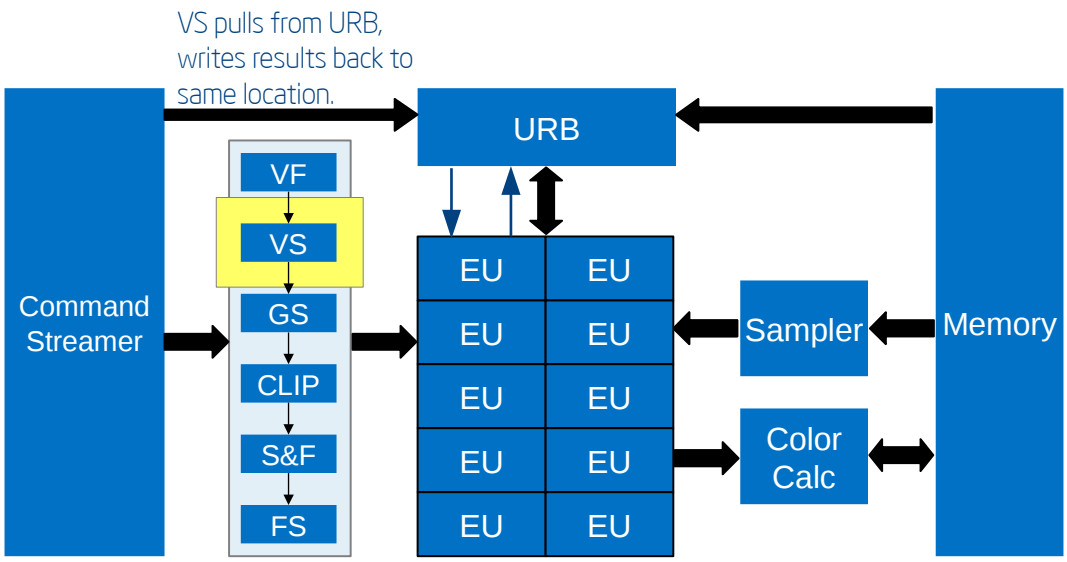


Unified return buffer (URB) is a region of on-chip memory used to store some kinds of data between stages.

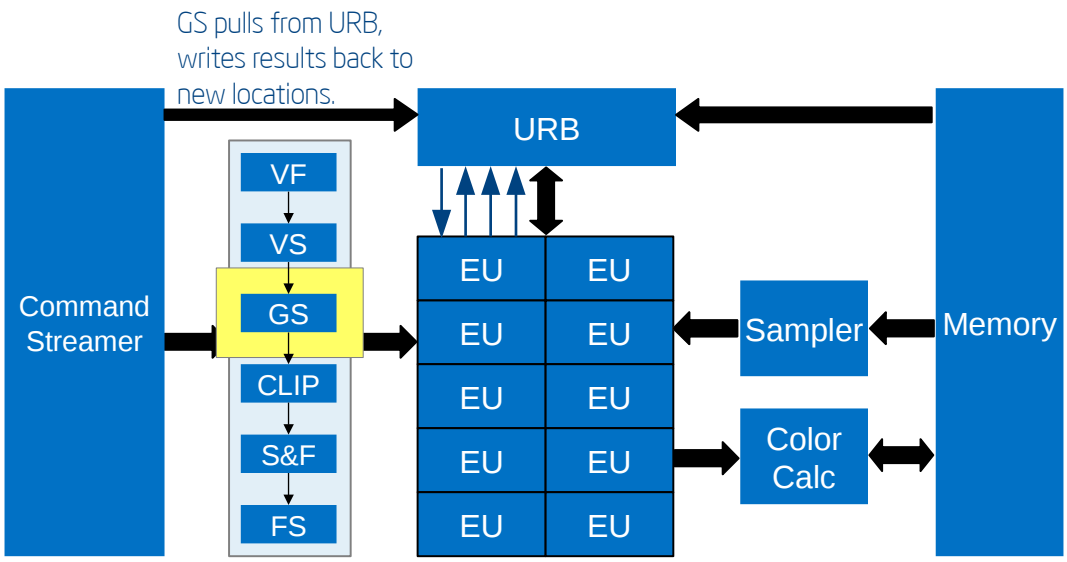
Data Flow Inside the GPU



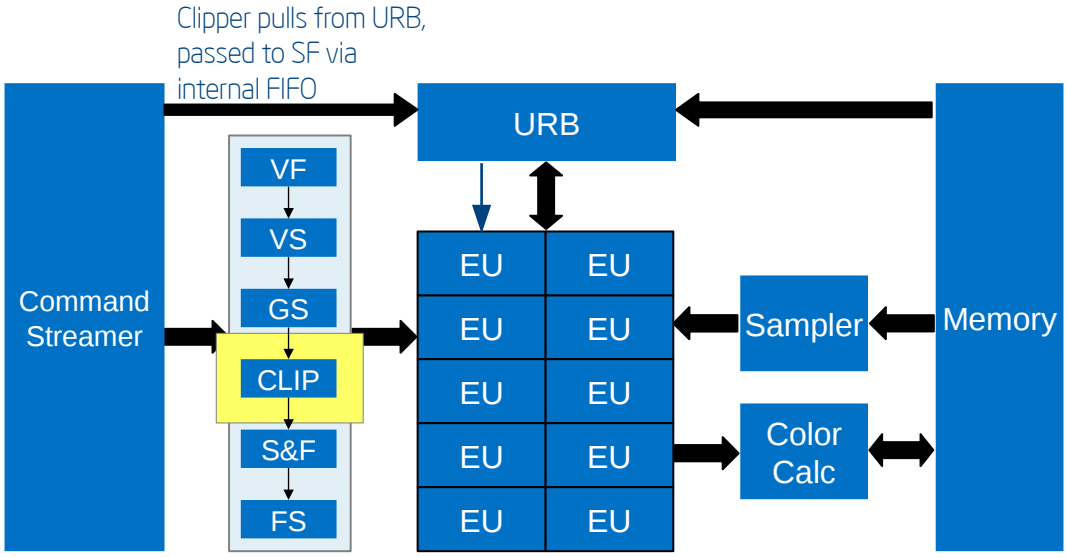
Data Flow Inside the GPU



Data Flow Inside the GPU

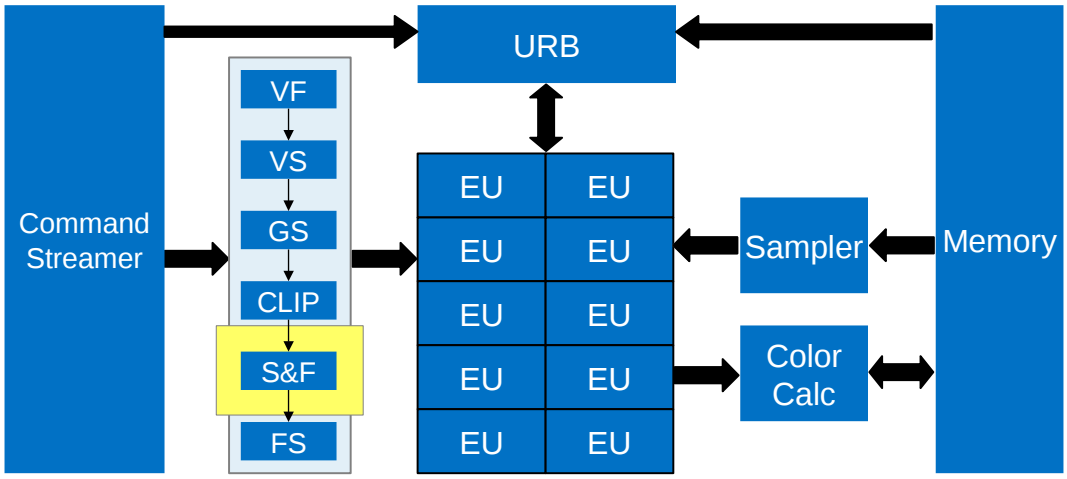


Data Flow Inside the GPU

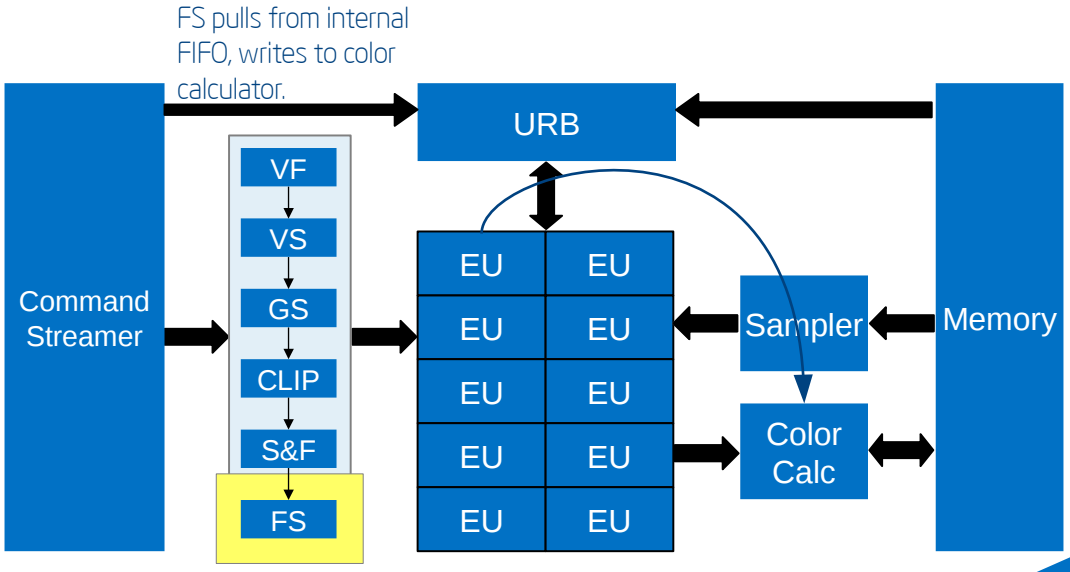


Data Flow Inside the GPU

Strips-and-fans operates similarly.



Data Flow Inside the GPU



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 tesselation outputs

Textures, TexBOs, and UBOs come from the sampler

All in-flight primitives share the fixed-size URB

Some non-UBO uniforms may spill into a UBO
- Arrays with non-constant indices
- Large data

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-documentation-prms

Driver source code publicly available:

https://01.org/linuxgraphics/documentation/source-code

https://01.org/linuxgraphics/documentation/build-guide-0

http://cgit.freedesktop.org/mesa/mesa/

Introduction

Agenda

- ✔ Why Intel graphics?
- ✔ 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 mapings

- 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

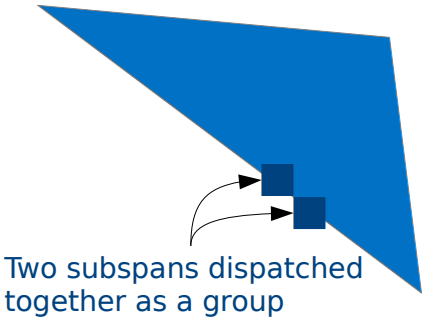
Avoid `glBufferSubData`

glBufferSubData is really just a malloc-allocated staging buffer.

Channel Masked Execution

Fragment dispatches two 2x2 subspans 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.



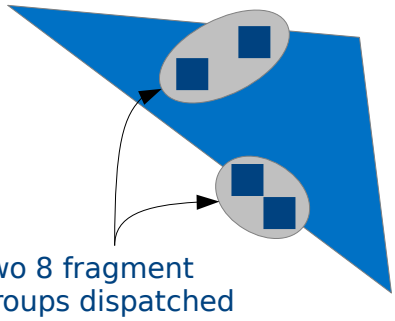
Two subspans dispatched together as a group

Hardware documentation and driver calls this SIMD8.

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



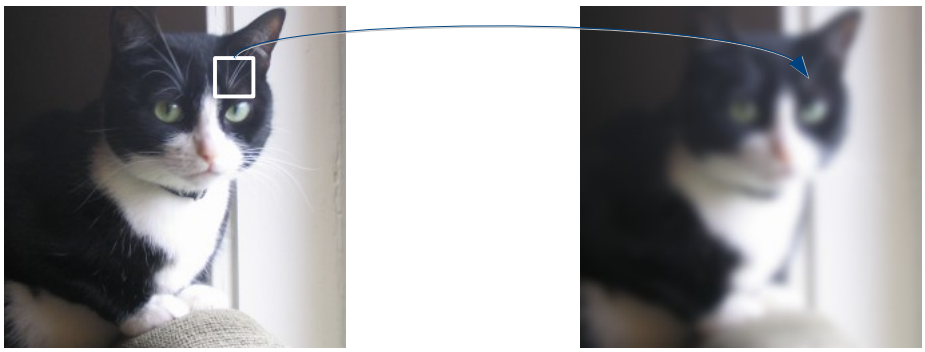
Two 8 fragment groups dispatched together

Hardware documentation and driver calls this SIMD16.

Each set of 8 fragments gets half of the registers.

Use `textureOffset` and Friends

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



This box filter averages all the pixels in the highlighted area to produce a single output pixel

Use `textureOffset` and Friends

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

- Do:

```
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:

```
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;
}
```

- Do not:

```
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
  - Same for complex math (rcp, exp2, log2, rsq, sqrt, sin, cos)
- pow has 22 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

These times are for HSW

Compact Varyings

Declare varyings to be only as large as necessary

- Do:

```
out vec2 tc;
void main()
{
    tc = …
}
```

- Do not:

```
out vec4 tc;
void main()
{
    tc = vec4(…, 0, 1);
}
```

- Fewer VS instructions
- Fewer FS instructions
- Less URB space

GEN doesn't use fixed-function hardware for interpolation. Instructions are inserted in the beginning of the fragment shader to perform the interpolation. Sending constant data results in unnecessary instructions.

If the z and w components are unused in the fragment shader, the extra instructions will likely be eliminated by the dead-code elimination pass.

Flat Varyings

Declare per-primitive varyings as flat

- Do:

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

- Do not:

```
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:

```
out vec3 normal;
out float intensity;
void main()
{
    normal = …;
    intensity = …;
}
```

- Do not:

```
out vec4 data;
void main()
{
    …
    data = vec4(normal, intensity);
}
```

SIMD “optimizations” in the vertex shader can trick our register allocator into having much larger live intervals than are necessary.

Introduction

Agenda

- ✔ Why Intel graphics?
- ✔ 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
```

- Much of this information is also available via `GL_ARB_debug_output` or `GL_KHR_debug`

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:
(
(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:
   (assign  (xy) (var_ref packed:texcoord)  (expression vec2 * (expression vec2 + (swiz xy (var_ref gl_Vertex) )(constant float (1.000000)) ) (constant float (0.500000)) ) )
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 };
   indices, point width, clip flags
0x00000020: mov(8)          g114<1>D        0D                              { align16 WE_normal 1Q };
   gl_Position
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   0x0000ff00UD    { align1 WE_all };
0x00000070: send(8)         null            g113<4,4,1>F
               urb 0 urb_write used complete mlen 5 rlen 0     { align16 WE_normal 1Q EOT };
```

Assembly language is documented in Volume 4, Part 3 of the PRM.

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:
   (assign  (xy) (var_ref packed:texcoord)  (expression vec2 * (expression vec2 + (swiz xy (var_ref gl_Vertex) )(constant float (1.000000)) ) (constant float (0.500000)) ) )
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 };
   indices, point width, clip flags
0x00000020: mov(8)          g114<1>D        0D                              { align16 WE_normal 1Q };
   gl_Position
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   0x0000ff00UD    { align1 WE_all };
0x00000070: send(8)         null            g113<4,4,1>F
               urb 0 urb_write used complete mlen 5 rlen 0     { align16 WE_normal 1Q EOT };
```

The time is a precomputed cost based on various metrics.

Flow-control, especially loops, and texture accesses make the estimate less accurate, but for straight-through code, the estimate is generally very good.

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
```

This is especially good for seeing how close your register usage is to being able to get 16-wide execution.

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 Execution Timings

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

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

type          ID      cycles spent                  % of total
fs16  glsl     3:       291191730 (   0.29 Gcycles)       2.3%
fs8   glsl     3:      2265333642 (   2.27 Gcycles)      17.6%
vs    glsl     3:     10317260118 (  10.32 Gcycles)      80.1%

total vs        :     10317260118 (  10.32 Gcycles)      80.1%
total fs8       :      2265333642 (   2.27 Gcycles)      17.6%
total fs16      :       291191730 (   0.29 Gcycles)       2.3%

type          ID      cycles spent                  % of total
fs16  glsl     3:       619003966 (   0.62 Gcycles)       2.2%
fs8   glsl     3:      4786157361 (   4.79 Gcycles)      17.0%
vs    glsl     3:     22789663885 (  22.79 Gcycles)      80.8%

total vs        :     22789663885 (  22.79 Gcycles)      80.8%
total fs8       :      4786157361 (   4.79 Gcycles)      17.0%
total fs16      :       619003966 (   0.62 Gcycles)       2.2%

type          ID      cycles spent                  % of total
fs16  glsl     3:       934151049 (   0.93 Gcycles)       2.1%
fs8   glsl     3:      7312214004 (   7.31 Gcycles)      16.4%
vs    glsl     3:     36255971577 (  36.26 Gcycles)      81.5%

total vs        :     36255971577 (  36.26 Gcycles)      81.5%
total fs8       :      7312214004 (   7.31 Gcycles)      16.4%
total fs16      :       934151049 (   0.93 Gcycles)       2.1%

type          ID      cycles spent                  % of total
fs16  glsl     3:      1250458485 (   1.25 Gcycles)       2.1%
fs8   glsl     3:      9670458850 (   9.67 Gcycles)      16.2%
vs    glsl     3:     48742313487 (  48.74 Gcycles)      81.7%

total vs        :     48742313487 (  48.74 Gcycles)      81.7%
total fs8       :      9670458850 (   9.67 Gcycles)      16.2%
total fs16      :      1250458485 (   1.25 Gcycles)       2.1%
```

Use INTEL_DEBUG=vs,fs to correlate the program numbers back to the actual shaders.

Debugging GL Errors

Nobody wants to sprinkle `glGetError` all over the place

- Use `GL_ARB_debug_output` or `GL_KHR_debug`
- `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

Debugging GL Errors

```
$ 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 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  0x00007ffff486d70c in _mesa_UseProgramStages (pipeline=1, stages=1, program=2) at ../../../src/mesa/main/pipelineobj.c:331
#2  0x0000000000400db8 in main (argc=1, argv=0x7fffffffdeb8) at /home/idr/src/my_application/main.cpp:35
(gdb) up
#1  0x00007ffff486d70c 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          }
```

Locking clocks for benchmarking

GPU clock control files live in sysfs

- Lock min and max clock by echoing values to `gt_min_freq_mhz` and `gt_max_freq_mhz`

```
$ ls /sys/class/drm/card0
card0-DP-1      card0-HDMI-A-2  dev             gt_min_freq_mhz  l3_parity
card0-DP-2      card0-HDMI-A-3  device          gt_RP0_freq_mhz  power
card0-DP-3      card0-LVDS-1    gt_cur_freq_mhz gt_RP1_freq_mhz  subsystem
card0-HDMI-A-1  card0-VGA-1     gt_max_freq_mhz gt_RPn_freq_mhz  uevent
$ cat gt_min_freq_mhz
350
$ cat gt_max_freq_mhz
1250
$ cat gt_cur_freq_mhz
350
```

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 and replay on another
- Trim traces to minimal size, submit with bug reports

http://apitrace.github.io/

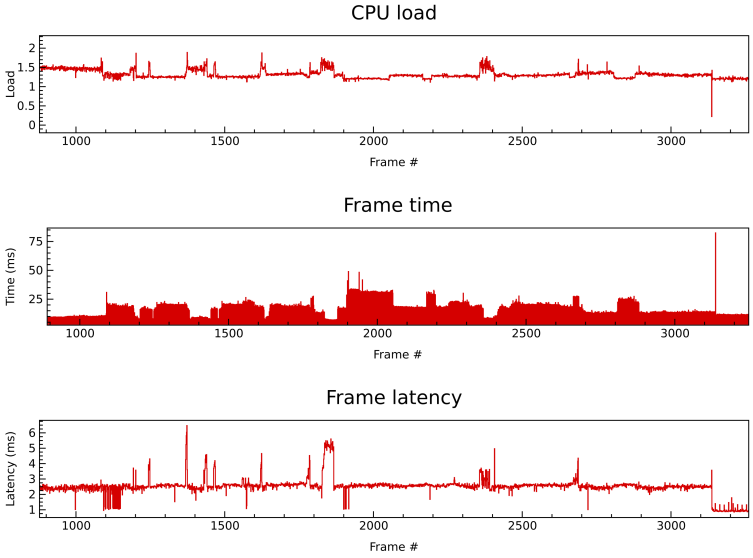
fips

Light-weight performance tracing of live applications

- Generates reports from hardware performance counters
- Generates reports of most frequently used / hot shaders
- Can be used with either live applications or trace replay tools

http://git.cworth.org/git/fips

fips


CPU load
Load
2
1.5
1
0.5
0
1000
1500
2000
2500
3000
Frame #
Frame time
Time (ms)
75
50
25
1000
1500
2000
2500
3000
Frame #
Frame latency
Latency (ms)
6
5
4
3
2
1
1000
1500
2000
2500
3000
Frame #

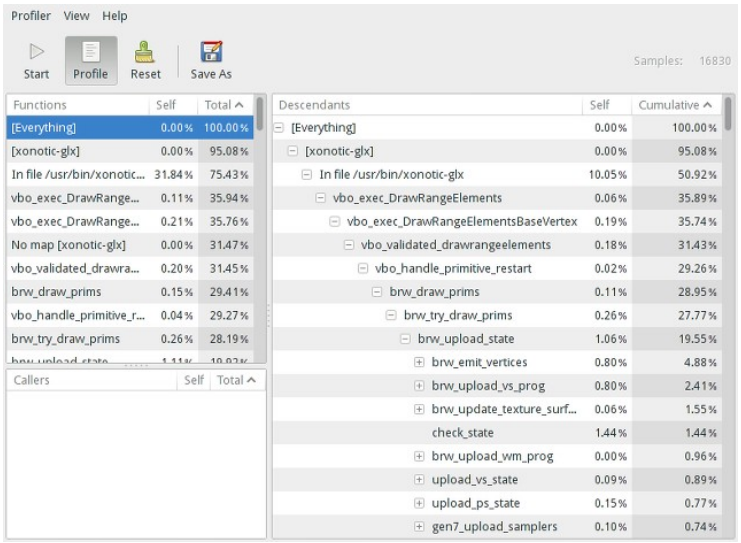
Your Favorite CPU Profiler

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!

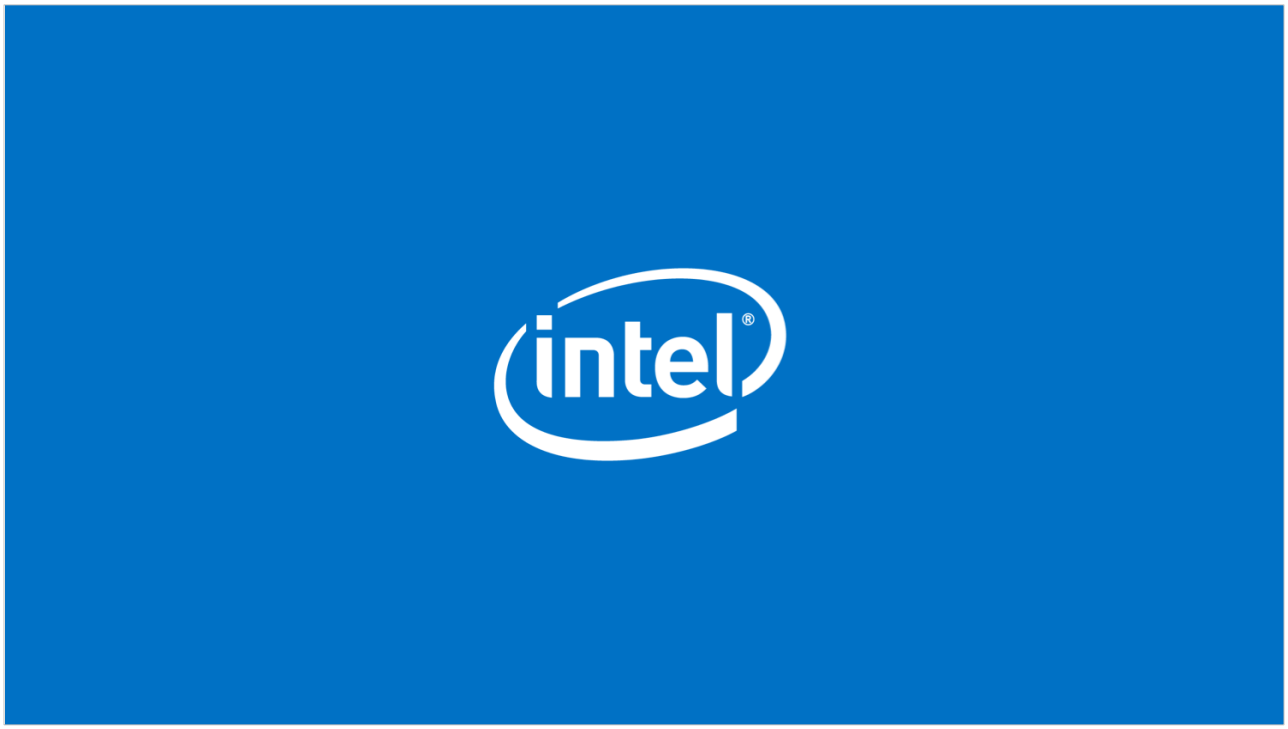
- We're regularly on irc.freenode.net #intel-gfx
- We have a mailing list

  http://lists.freedesktop.org/mailman/listinfo/mesa-dev
- Our bug tracker is public

  https://01.org/linuxgraphics/documentation/how-report-bugs-0

Join us... we are hiring

- ian.d.romanick@intel.com or kaveh.nasri@intel.com


intel