LLVM for OpenGL and other stuff

Chris Lattner
Apple Computer
clattner@apple.com
OpenGL JIT
OpenGL Vertex/Pixel Shaders
OpenGL Pixel/Vertex Shaders

• Small program, provided at run-time, to be run on each vertex/pixel:
  – Written in one of a few high-level graphics languages (e.g. GLSL)
  – Executed millions of times, extremely performance sensitive

• Ideally, these are executed on the graphics card:
  – What if hardware doesn’t support some feature? (e.g. laptop gfx)
  – Interpret or JIT on main CPU

```cpp
void main() {
  vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
  vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
  vec3 lightVec = normalize(LightPosition - ecPosition);
  vec3 reflectVec = reflect(-lightVec, tnorm);
  vec3 viewVec = normalize(-ecPosition);
  float diffuse = max(dot(lightVec, tnorm), 0.0);
  float spec = 0.0;
  if (diffuse > 0.0) {
    spec = max(dot(reflectVec, viewVec), 0.0);
    spec = pow(spec, 16.0);
  }
  LightIntensity = DiffuseContribution * diffuse + SpecularContribution * spec;
  MCposition = gl_Vertex.xy;
  gl_Position = ftransform();
}
```

GLSL Vertex Shader
MacOS OpenGL Before LLVM

• Custom JIT for X86-32 and PPC-32:
  – Very simple codegen: Glued chunks of Altivec or SSE code
  – Little optimization across operations (e.g. scheduling)
  – Very fragile, hard to understand and change (hex opcodes)

• OpenGL Interpreter:
  – JIT didn’t support all OpenGL features: fallback to interpreter
  – Interpreter was very slow, 100x or worse than JIT
OpenGL JIT built with LLVM Components

- At runtime, build LLVM IR for program, optimize, JIT:
  - Result supports any target LLVM supports (+ PPC64, X86-64 in MacOS 10.5)
  - Generated code is as good as an optimizing static compiler
- Other LLVM improvements to optimizer/codegen improves OpenGL
- Key question: How does the “OpenGL to LLVM” stage work?
Structure of an Interpreter

• Simple opcode-based dispatch loop:

```c
while (...) {
  ...
  switch (cur_opcode) {
    case dotproduct:  result = opengl_dot(lhs, rhs); break;
    case texturelookup: result = opengl_texlookup(lhs, rhs); break;
    case ...
  }
}
```

• One function per operation, written in C:

```c
double opengl_dot(vec3 LHS, vec3 RHS) {
  #ifdef ALTIVEC
    ... altivec intrinsics ...
  #elif SSE
    ... sse intrinsics ...
  #else
    ... generic c code ...
  #endif
}
```

• In a high-level language like GLSL, each op can be hundreds of LOC
At OpenGL build time, compile each opcode to LLVM bytecode:
- Same code used by the interpreter: easy to understand/change/optimize
OpenGL to LLVM: At runtime

1. Translate OpenGL AST into LLVM call instructions: one per operation
2. Use the LLVM inliner to inline opcodes from precompiled bytecode
3. Optimize/codegen as before

```glsl
vec3 viewVec = normalize(-ecPosition);
float diffuse = max(dot(lightVec, tnorm), 0.0);
...
```

```
%tmp1 = call opengl_negate(ecPosition)
%viewVec = call opengl_normalize(%tmp1);
%tmp2 = call opengl_dot(%lightVec, %tnorm)
%diffuse = call opengl_max(%tmp2, 0.0);
...
```

```
%tmp1 = sub <4 x float> <0,0,0,0>, %ecPosition
%tmp3 = shuffle <4 x float> %tmp1, ...;
%tmp4 = mul <4 x float> %tmp3, %tmp3
...
```

http://llvm.org/
Benefits of this approach

• Key features of this approach:
  – Each opcode is written/debugged for a simple interpreter, in standard C
  – Retains all advantages of an interpreter: debugability, understandability, etc
  – Easy to make algorithmic changes to opcodes
  – OpenGL runtime is independent of opcode implementation

• Primary contributions to Mac OS:
  – Support for PPC64/X86-64
  – Much better performance: optimizations, regalloc, scheduling, etc
  – No fallback to interpreter needed!
  – OpenGL group doesn’t maintain their own JIT!
Another Example: Colorspace Conversion

- Code to convert from one coordinate system to another:
  - e.g. BGRA 444R -> RGBA 8888
  - Hundreds of combinations, importance depends on input

```
for each pixel {
  switch (infmt) {
    case RGBA 5551:
      R = (*in >> 11) & C
      G = (*in >> 6) & C
      B = (*in >> 1) & C
      ... }
  switch (outfmt) {
    case RGB888:
      *outptr = R << 16 | G << 8 ...
  }
}
```

- Run Time Specialize
- Compiler optimizes shifts and masking
LLVM + Dynamic Languages
LLVM and Dynamic Languages

• Dynamic languages are very different than C:
  – Extremely polymorphic, reflective, dynamically extensible
  – Standard compiler optzns don’t help much if “+” is a dynamic method call

• Observation: in many important cases, dynamism is eliminable
  – Solution: Use dataflow and static analysis to infer types:

    ```
    'i' starts as an integer
    ++ on integer returns integer
    var i;
    for (i = 0; i < 10; ++i)
      ... A[i] ...
      i isn’t modified anywhere else
    ```

    – We proved “i” is always an integer: change its type to integer instead of object
    – Operations on “i” are now not dynamic
      – Faster, can be optimized by LLVM (e.g. loop unrolling)
Advantages and Limitations of Static Analysis

- Works on **unmodified programs** in scripting languages:
  - No need for user annotations, no need for sub-languages

- Many approaches for doing the analysis (with cost/benefit tradeoffs)

- Most of the analyses could work with many scripting languages:
  - Parameterize the model with info about the language operations

- Limitation: cannot find all types in general!
  - That’s ok though, the more we can prove, the faster it goes
Scripting Language Performance

• Ahead-of-Time Compilation provides:
  – Reduced memory footprint (no ASTs in memory)
  – Eliminate (if no ‘eval’) or reduce use of interpreter at runtime (save code size)
  – Much better performance if type inference is successful
• JIT compilation provides:
  – Full support for optimizing eval’d code (e.g. json objects in javascript)
  – Runtime “type profiling” for speculative optimizations

• LLVM provides:
  – Both of the above, with one language -> llvm translator
  – Install-time codegen
  – Continuously improving set of optimizations and targets
  – Ability to inline & optimize code from different languages
  – inline your runtime library into the client code?