# RSL - LLVM compiler project proposal

25 May, 2008

Syoyo Fujita

# Terminology

- RSL
  - RenderMan Shading Language
- LLVM http://www.llvm.org/
  - Compiler infrastructure
- AST
  - Abstract Syntax Tree
- JIT
  - Just In Time compiling

# Agenda

- Project goal
- Architecture overview
- Compiling RSL
- Runtime

# Project goal (1/2)

- Investigate a possibility of LLVM as a shader VM
  - JIT
  - Run-time specialization
  - Optimization
- For a global illumination setting, if possible.

# Project goal (2/2)

- No full RSL implementation
  - Do investigation with minimum equipment.
    - Explore LLVM's performance, functionality, etc. when it is applied to shader VM.
- Leave the experience into the document.

# Why LLVM?

- Has SIMD instruction and x86/SSE codegen support
- Has JIT support
- Has optimizer. It's optimization is as good as gcc does
- Actively developing

# Expected benefit by using LLVM for shader VM

- Faster execution of shader
  - JIT, partial evaluation, x86/SSE instruction
- Save out time
  - Production-quality compiler optimization for FREE!
  - Reusable shader VM code among aqsis, lucille, etc.

# How it would work?

shader code

RSL

compile

compiled shader code

LLVM bitcode

```
surface
matte( float Ka = 1;
       float Kd = 1;)
{
    normal Nf = faceforward(normalize(N), I);
    Oi = Os;
    Ci = Os * Cs * (Ka * ambient() + Kd * diffuse(Nf));
}
```

+

compiled builtin

LLVM bitcode

compile

builtin function source

C/C++, etc

```
faceforward()
normalize()
...
```

offline

runtime

JIT compile & specialization by LLVM

JIT-ed, specialized shader code

native

call

interact

Renderer

native

```
texture()
diffuse()
ambient()
trace()
...
```

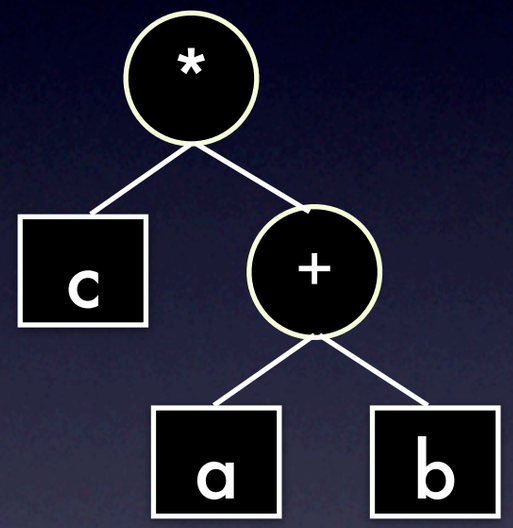
# What we have to develop

- RSL -> LLVM compiler
- Runtime support

# Compiling RSL

RSL → Parse → AST → Codegen → LLVM

(a + b) * c




```
%tmp0 = add  float %a, %b
%tmp1 = mul float %c,
%tmp0
```

(mul (id c) (add a b))

# slc.py

- I've wrote simple RSL to LLVM IR translator written in Python
- Directly emits LLVM IR assembly
- Construct AST internally
  - Easy to add codegen other than LLVM, if required.

```
$ cat input.sl
surface
myshader ()
{
    Cs = Ci;
}
$ slc.py input.sl
$ llvm-as input.ll -f
$ llvm-dis input.bc
...
define void @myshader() {
    %tmp0 = load <4 x float>* @Ci       ; <<4 x float>> [#uses=1]
    store <4 x float> %tmp0, <4 x float>* @Cs
    ret void
}
```

# Compiling builtin function

- **Self-contained functions**
  - normalize(), reflect(), ...
  - Just compile into LLVM bitcode by using you favorite frontend(C/C++, etc)
- **Functions which interacts renderer internal**
  - ambient(), trace(), illuminance(), ...
  - Need a careful treatment

Need a investigation

# Note

- External & built-in functions are unresolved at this time.
- How to handle DSO?
  - Also compile DSO into LLVM bitcode?

Need a investigation

# Runtime

# Runtime phase

- Read shader bitcode
- Setting up function parameter
- JIT compile the shader function
  - Specialize the shader function
- Call the shader function

# C/C++, renderer runtime, pseudo code.

```
setup_shader(
    const char *shaderfile,    // Compiled LLVM bitcode shader
    state_t    *state,         // shader env, shader param
    renderer_t *renderer)
{
  m    = LLVM_LoadModule(shaderfile);
  f    = LLVM_GetFunction(m);
  sig = LLVM_GetFunctionSignature(f);

  param = BindParameter(sig, state);

  // JIT compile the shader with optimization
  // (Include partial evaluation),
  // then get the C function pointer(i.e. on the memory).
  entrypoint = LLVM_JITCompileWithOpt(f, param);

  renderer->shader = entrypoint;
}
```

# C/C++, renderer runtime, pseudo code.

```
shade(
    state_t         *state,     // [in]
    shadingpoint_t *points,     // [inout]
    int              npoints,
    renderer_t      *renderer)
{
    for (i = 0; i < npoints; i++) {
        (*renderer->shader)(state, points[i]);
    }

}
```

# Shader specialization(1/2)

- Constant folding for uniform variable
  - Let LLVM do that.

## Input

```
surface(float Ka)
{
  Cs = sqrt(Ka) * Ci;
}
```

## Specialized

```
surface(4.0)
{
  Cs = 2.0 * Ci;
}
```

# Shader specialization(2/2)

- Caching a varying variable(e.g. texture())
  - G-buffer

Need a investigation

- Things will be difficult for global illumination
  - A lot of indirect references, buffers which makes impractical for caching

# Designing interface

# C/RSL/LLVM

- What is the good interface for these layer to access each other? e.g.
  - Pass vector type as pointer or as value?
  - Use opeque pointer?
- Requirement: portable, easy to specializable by LLVM

Need a investigation

# Polymorphic function

- RSL has polymorphic function
- How to define interface for polymorphic function?
- A proposal:
  - float noise(float) -> noise_ff
  - vector noise(float, float) -> noise_vff
  - vector noise(point) -> noise_vp

Need a investigation

# LLVM Code style

- SPMD
- short vector SIMD
- SPSD

# SPMD code style

- Pros
  - SIMD optimization in LLVM level
  - Fits reyes style renderer naturally
    - aqsis
- Cons
  - How to efficiently handle incoherency?
    - trace(), if/while
    - call a shader/DSO in the shader

Need a investigation

## Pseudo LLVM code

```
myshader(shadpts *sps, int n)
{
  for i = 0 to n:
      sps[i].Cs = sps[i].Ci
}
```

## RSL

```
surface myshade()
{
    Cs = Ci;
}
```

## Pseudo Renderer code

```
shade(primitive *prim)
{
  shadpts *pts; int n;

  dice(prim, pts /* out */, &n /*out */);

  // fill input
  for (i = 0; i < n; i++) {
    pts[i].Ci = ...
  }

  // call shader
  (*shader)(pts, n);
}
```

# SPSD code style

- Pros
  - Work in most case
  - Low overhead
  - Fits raytracing-based renderer
    - lucille
- Cons
  - To compute derivative, we have to execute 3 or 4 instance at a time anyway
    - short vector SIMD

- It’s possible to emit 3 pattern of shader code at compiling phase
  - SPMD(vector size: n) good for reyes
  - 4 SIMD(vector size: 4) good for raytracing
  - Scalar(vector size: 1) good for incoherency: DSO call, if/while

- And more, synthesize appropriate shader code place by place, by investigating shader program.
  - But needs sophisticated RSL compiler support.

# Note

- Keep in mind that LLVM bitcode can be modified/synthesizable at runtime phase.
- Tweaking parameter/ABI is easy
- No need to define ABI strictly.

# Current status

| component | progress |
| --- | --- |
| RSL to LLVM compiler | 50% |
| Self-contained<br>Builtin functions | 5% |
| complex builtin functions | 0% |
| Fake shader engine<br>(fake renderer) w/ FLTK GUI | 0% |
| ABI design | 0% |
| Document | 0% |
| Highlevel, whole-pipeline<br>optimization | if possible |

# Project period

- I’m seeing about a half year to finish this project since I have very limited vacant time after my day job I can spend for
  - And also I have many thing to do within this very limited & precious time: e.g. coding my renderer, writing SIMD language, consult global illumination rendering as volunteer and so on.
- Although we could collaborate/help each other the internet with enthusiasts(e.g. register, c4f), I wish someone could invest/support financially/employ me so that I could accelerate and do this project professionally.

# Resouce

- Discussion forum
  http://lucille.lefora.com/2008/05/06/rsl-llvm-compiler-project-started/
- SVN
  http://lucille.svn.sourceforge.net/svnroot/lucille/angelina/rsl2llvm

# Refernces


- Brian Guenter and Todd B. Knoblock and Erik Ruf, Specializing shaders, SIGGRAPH 1995.
- Jonathan Ragan-Kelley and Charlie Kilpatrick and Brian W. Smith and Doug Epps and Paul Green and Christophe Hery and Fr\'{e}do Durand, The lightspeed automatic interactive lighting preview system, SIGGRAPH 2007
- Conal Elliott, Programming Graphics Processors Functionally, Proceedings of the 2004 Haskell Workshop http://conal.net/Vertigo/
- Chad Austin, Renaissance : A Functional Shading Language, Master's Thesis http://aegisknight.org/articles