LLVM for Open Shading Language

Larry Gritz
& Solomon Boulos, Alejandro Conty, Chris Kulla, Cliff Stein

sony pictures
imageworks
Open Shading Language (OSL)

- Film VFX-oriented DSL
- Ray-trace/GI-friendly
- Modular, generic, renderer-agnostic
- Open source (BSD)
- Uses LLVM
Production Rendering

• Huge scenes: 10+ GB geom, 200+ GB texture
• Quality is everything
• 4-10 hours/frame, peak 20-40+ hr (CPU time)
• Many thousands of cores
• No graphics hardware for final frames
• Shaders
  • describe materials, lights, displacements
  • huge: 50,000+ ops per shader group
  • important programmable tool when all else fails
Shader

Inputs

 shader gamma ( 
  color Cin = 1,
  float gam = 1,
  output color Cout = 1
 )
{
  Cout = pow (Cin, 1/gam);
}

Outputs

Cout

Cin

gam
Shader Group
What’s wrong with shaders

- **C/C++ shaders:**
  - can crash the renderer or leak memory
  - too much exposure of renderer internals
  - versionitis madness
  - hard to globally optimize

- **Hardware dependence & limitations**

- **Black boxes:** can’t reason about them, sample, reorder, defer, etc.

- **Suboptimal for a modern ray tracer**
New Language Goals

• Similar to RSL/GSL, but evolved & easier
• Separate description vs implementation
  • End versionitis nightmare
  • Late-stage optimization
• Renderer control of rays / more physical shading
• Lazy running of layers
• Closures describe materials/lights
• Automatic differentiation

• No crashing, NaNs, etc.
• Allow multiple back ends
First: Bytecode Interpreter

```c
for (ip = beginop; ip < endop && beginpoint < endpoint; ++ip) {
    Opcode &op (code[ip]);
    op.implementation (op.nargs, args+op.firstarg);
}

OP_add (...) {
    Symbol &Result (exec->sym (args[0])), &A(...), &B(...);
    if (Result.is_uniform())
        Result[0] = A[0] + B[0];
    else {  // varying case
        for (int i = beginpoint; i < endpoint; ++i)
            if (runflags[i])
                Result[i] = A[i] + B[i];
    }
}
```

Gritz / OSL / LLVM dev conf 2010
Bytecode interpreter

- Extensive runtime specialization
- Interpreter performance pretty good
  - Interpreter overhead amortized over batches
  - Uniforms make up the difference
- Hinges on large enough batch sizes...
  - Hard for GI ray tracer to keep batches big enough
  - Lots of renderer-side overhead from batching
  - Performance about 1/5 what we needed for real-world batch sizes

- Next plan: LLVM
LLVM: first try

- Compile existing batch shadeops with llvm-gcc
- JIT code that makes the calls in succession
- Inline and hope for the best
- Not unlike the Apple OpenGL interpreter

Results:
- Tricky static initialization problems
- Huge memory bloat
- Not good performance
- Didn’t work correctly for all cases. Now what?
- But, inspiring proof of concept
LLVM: Plan B

- Bytecode $\rightarrow$ LLVM IR $\rightarrow$ JIT
  - LLVM IR for control flow and many ops
  - Some ops C++ compiled with llvm-gcc/clang
  - Some ops IR function calls to renderer internals
- Still do extensive folding, specialization, deriv analysis on the bytecode first
- Single point execution only
  - Greatly simplifies & optimizes renderer side
First “Real” Result

• About 40 man-hours to implement Plan B for a subset of the language necessary to run a small math benchmark

• 16x faster vs 1-point-at-a-time interpreter

• Next step: Plan B for full language
Final Results

• Full team worked ~3 months
• Greatly simplified internals & renderer
• Exceed perf of hand-coded C shaders
  • Our runtime specialization + LLVM optimization
  • Inter-layer optimization and pruning
  • No batching overhead on renderer side
• We recently removed the interpreter!
• Continuing to add features & optimize
• First shows going into production now
Issues we’re running into with LLVM
Issues: Optimization

- Full C++ optimizations not a good tradeoff
- Laborious process of picking LLVM passes
- Trial and error
- Still something to go back to
- Lingering bugs?
  - Lots of time fighting LLVM crashes with certain pass combinations
  - Still poorly understood
  - May be related to reusing EE
Issues: Parallelization

- We have many threads
- New shader groups to JIT at any time
- WBN: allow multiple threads to JIT without separate LLVM context per thread (for memory reasons)
- WBN: parallel JIT examples that share a global Module of shared code but can JIT independent code.
Issue: Memory

- Thousands of shader groups to JIT
- New ones coming along all the time
- Add to existing module/EE? (Bugs?)
- New module per group? (memory!)
- Once JITted, we only need machine code
- Planned: custom JITMemoryManager?
  - But would be great to be an LLVM feature
Issue: Hardware parallelism

- How to take advantage of SSE/AVX?
- PTX back end?
- LLVM vector support
  - avoid manually using intrinsics?
  - how to get better LLVM vector support?
LLVM wish list

- More working examples, especially for JIT (versus static compilation)
- Many static codepath tools useless for JIT
- No good metadata-for-debugging docs
- Better multithread JIT
- Return JIT code and free everything else
Future work

- Optimize performance
- Memory/speed issues as we scale to thousands of shaders
- Allow BSDF prims & integrators to be expressed in OSL itself
- Alternate back ends (PTX)
- SSE/AVX
- Conquer the world
Q&A and Reminders

• lg@imageworks.com
• http://code.google.com/p/openshadinglanguage

• Acknowledgements:
  • OSL Coathors: Cliff Stein, Chris Kulla, Alejandro Conty, Solomon Boulos
  • SPI: Adam Martinez, Jay Reynolds, John Monos, Rene Limberger, Rob Bredow
  • Images: Sony Pictures, Disney