LLVM and Clang: Next Generation Compiler Technology

LLVM: Low Level Virtual Machine

Chris Lattner
BSDCan 2008
May 17, 2008

Outline

- What is LLVM?
- llvm-gcc 4.2 Compiler
- OpenGL in Mac OS X 10.5
- clang Compiler Front-end

What is the LLVM Project?

Language Independent Optimizer and Code Generator

- Broad suite of compiler optimizations, both standard and advanced
- Many targets supported

llvm-gcc 4.2 front-end

- Provides drop in compatibility with GCC and existing makefiles
- GCC frontend provides support for C, C++, Objective-C, Ada, FORTRAN ...

clang front-end

- New "llvm-native" Front End for C based languages
- Designed for speed, reusability, compatibility with GCC quirks

Everything is Open Source, and BSD Licensed! (except GCC)

http://llvm.org/

Why new compilers?

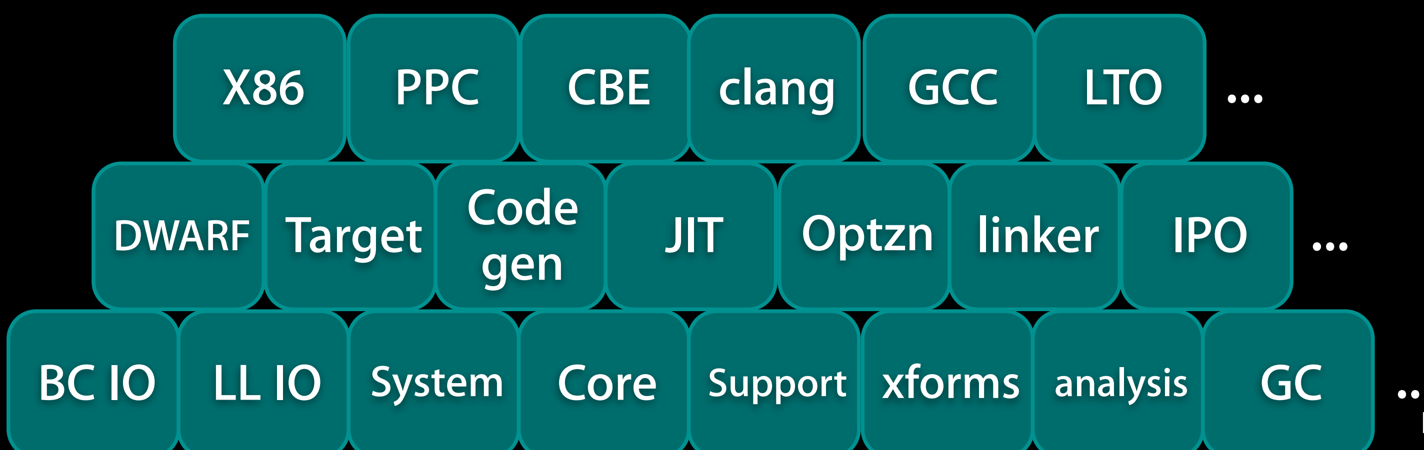
Existing Open Source C Compilers have Stagnated!

- Existing production-grade open source compilers:
  - Based on decades old code generation technology
  - No modern techniques like cross-file optimization and JIT codegen
  - Aging code bases: difficult to learn, hard to change substantially
  - Can't be reused in other applications
  - Keep getting slower with every release
- What I want:
  - A set of production-grade reusable libraries
  - ... which implement the best known techniques drawing from modern literature
  - ... which focus on compile time
  - ... and performance of the generated code
- Ideally support many different languages and applications!

LLVM Vision and Approach

- Primary mission: **build a set of modular compiler components**:
  - **Reduces the time & cost** to construct a particular compiler
    - A new compiler = glue code plus any components not yet available
  - Components are **shared across different compilers**
    - Improvements made for one compiler benefits the others
  - Allows choice of the **right component for the job**
    - Don't force "one true register allocator", scheduler, or optimization order

- **Secondary mission: Build compilers that use these components**
  - ... for example, a C compiler



http://llvm.org/

LLVM Optimizer/Codegen Highlights

Approachable code base, modern design, easy to learn

- Okay okay, assuming you are a compiler hacker
- Strong and friendly community, good documentation

Language and target independent code representation

- Very easy to generate from existing language front-ends
- Text form allows you to write your front-end in perl if you desire

Modern Code Generator:

- Supports both JIT and static code generation
- Targets: X86[-64], PPC[64], ARM/Thumb, Cell, MIPS, SPARC, IA64, Alpha, PIC16, C ...
- Much easier to retarget to new chips than GCC, for example

llvm-gcc 4.2

C/C++/ObjC/Ada/Fortran/...

GCC 4.x Design

- Standard compiler organization: front-end, optimizer, codegen
  - Parser and front-ends work with language-specific syntax trees
  - Optimizers improve code, mostly new since GCC 4.0
  - RTL code generator use antiquated compiler algorithms/data structures

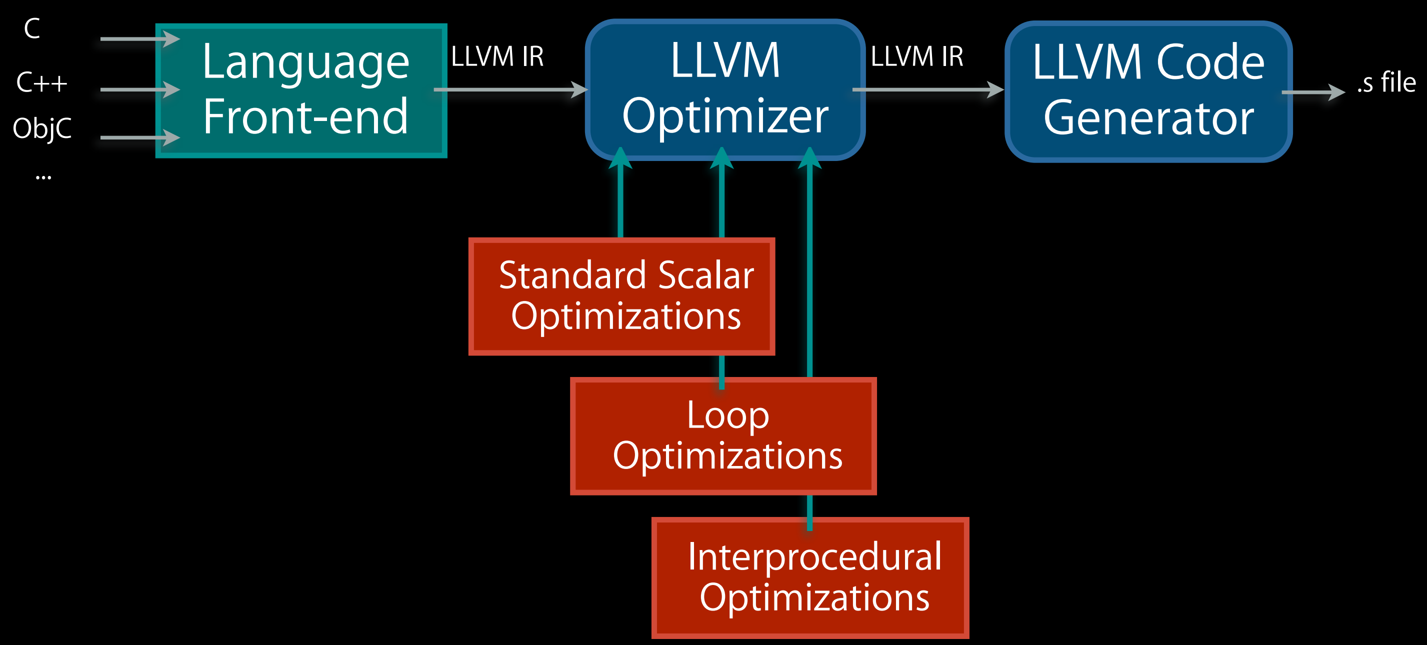
- Pros: Conformant front-ends, support for many processors, defacto standard
- Cons: Very slow, memory hungry, hard to retarget, no JIT, no cross-file optimizations, no aggressive optimizations, not a library...

llvm-gcc 4.2 Design

- Use GCC front-end with LLVM optimizer and code generator
  - Reuses parser, runtime libraries, and some GIMPLE lowering
  - Requires a new GCC "tree to llvm" converter

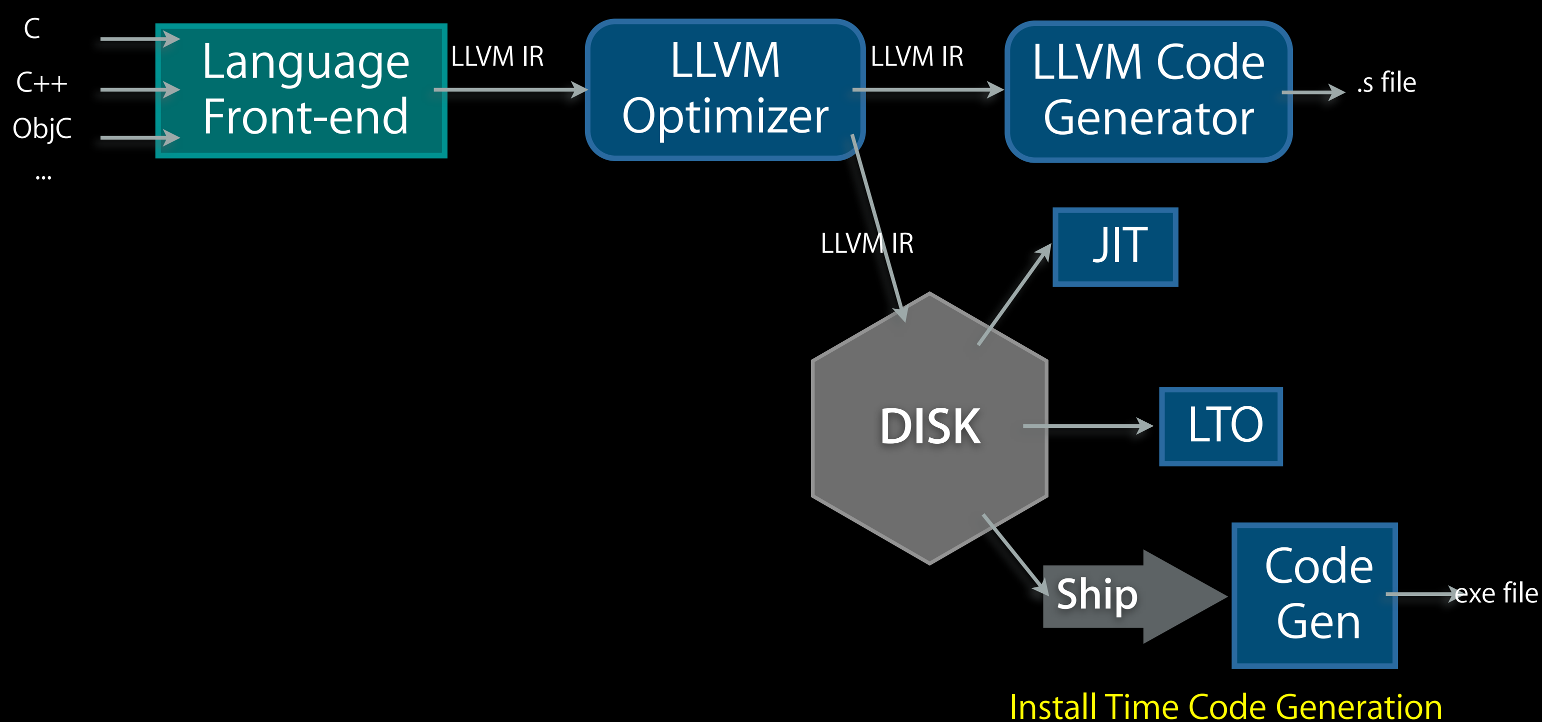
LLVM optimizer features used by llvm-gcc

- Aggressive and fast optimizer built on modern techniques
  - SSA-based optimizer for light-weight (fast) and aggressive transformations
  - Aggressive loop optimizations: unrolling, unswitching, mem promotion, ...
  - Inter-Procedural (cross function) optimizations: inlining, dead arg elimination, global variable optimization, IP constant prop, exception handling optimization, ...



Other LLVM features used by llvm-gcc

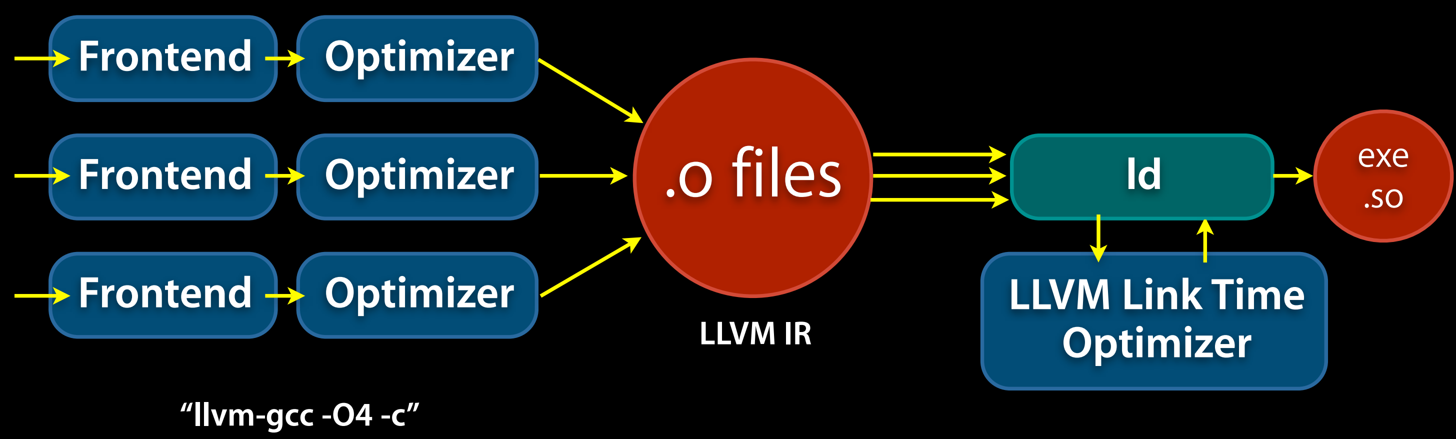
- Write LLVM IR to disk for codegen after compile time:
  - link-time, install-time, run-time



http://llvm.org/

LLVM Link Time Optimization

- Transparent LTO:
  - When compiling at -O4, emit LLVM IR to .o files
  - Provides drop in compatibility with existing makefiles and build systems
  - Works across languages: e.g. inline C++ code into C code



http://llvm.org/

llvm-gcc 4.2 Summary

- Drop in replacement for GCC 4.2
  - Compatible with GCC command line options
  - Works with existing makefiles (e.g. "make CC=llvm-gcc")
  - Supports GCC extensions and its languages (C, C++, Ada, FORTRAN, ...)
- Benefits of LLVM Optimizer and Code Generator
  - Optimizations across source files (e.g. inlining, constant propagation)
  - Faster optimizer (~30% at -O3 in most cases)
  - Slightly better codegen (usually 5-10% on x86/x86-64, depending on code)
- Allows interesting new applications
  - JIT compile/optimize C/C++ code
  - Generate code at install time

LLVM + OpenGL

Mac OS X 10.5: Colorspace Conversion

- Code to convert from one color format 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

```
for each pixel {
  R = (*in >> 11) & C;
  G = (*in >> 6) & C;
  B = (*in >> 1) & C;
  *outptr = R << 16 |
            G << 8 ...
}
```

Compiler optimizes shifts and masking

- Speedup depends on src/dest format:
  - 5.4x speedup on average, 19.3x max speedup: (13.3MB/s to 257.7MB/s)

OpenGL Pixel/Vertex Shaders

- Small program run on each vertex/pixel, provided at run-time:
  - 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**

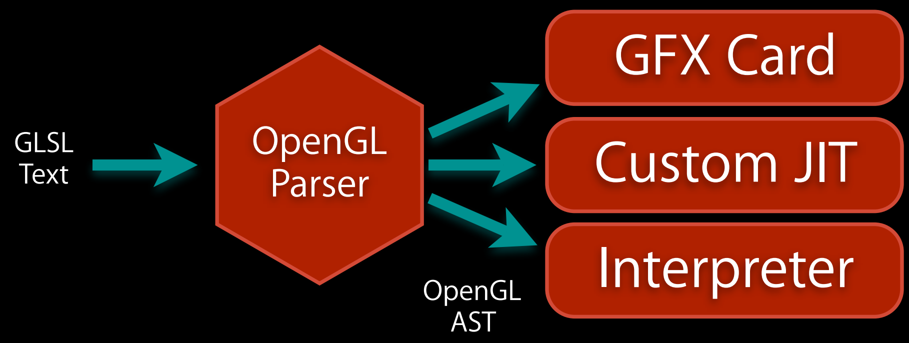
```
void main() {
 vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
 vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
 vec3 lightVec   = normalize(LightPosition - ecPosition);
 vec3 reflectVec = reflect(-lightVec, tnorm);
 vec3 viewVec    = normalize(-ecPosition);
 float diffuse   = max(dot(lightVec, 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

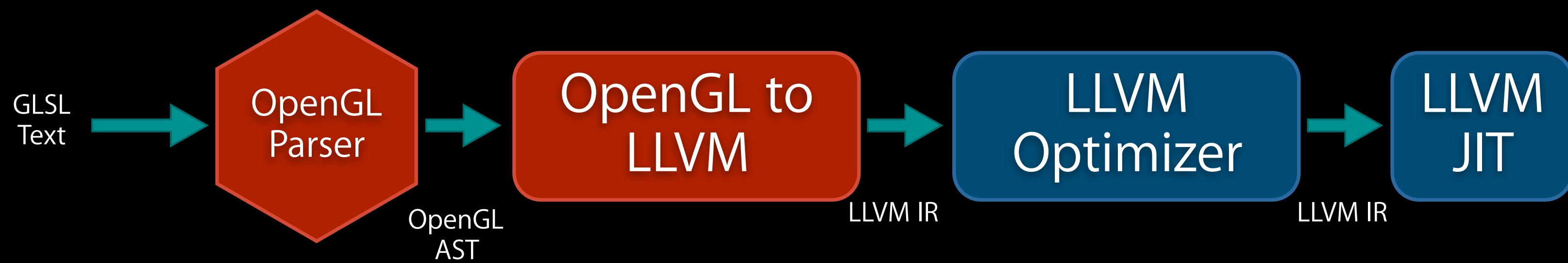
MacOS OpenGL Before LLVM

- 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?

http://llvm.org/

Structure of an Interpreter

- Simple opcode-based dispatch loop:

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

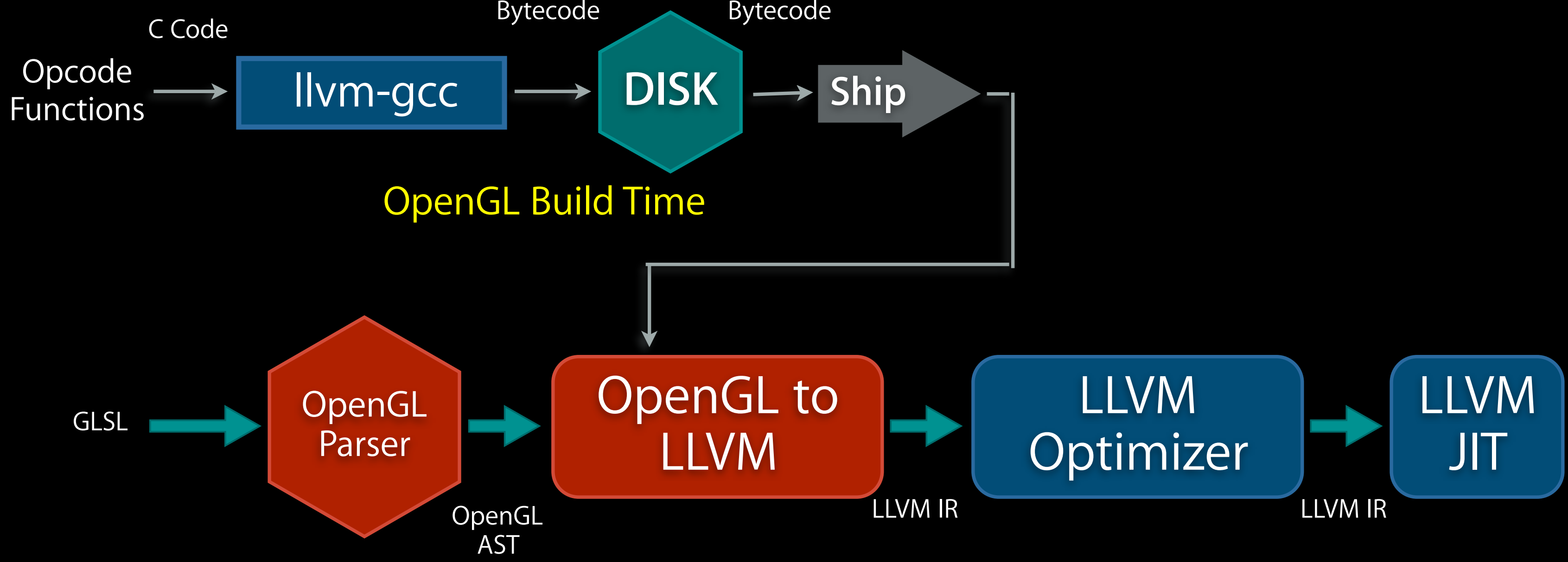
- One function per operation, written in C:

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

Key Advantage of an Interpreter:
Easy to understand and debug, easy to write each operation (each operation is just C code)

- In a high-level language like GLSL, each op can be hundreds of LOC

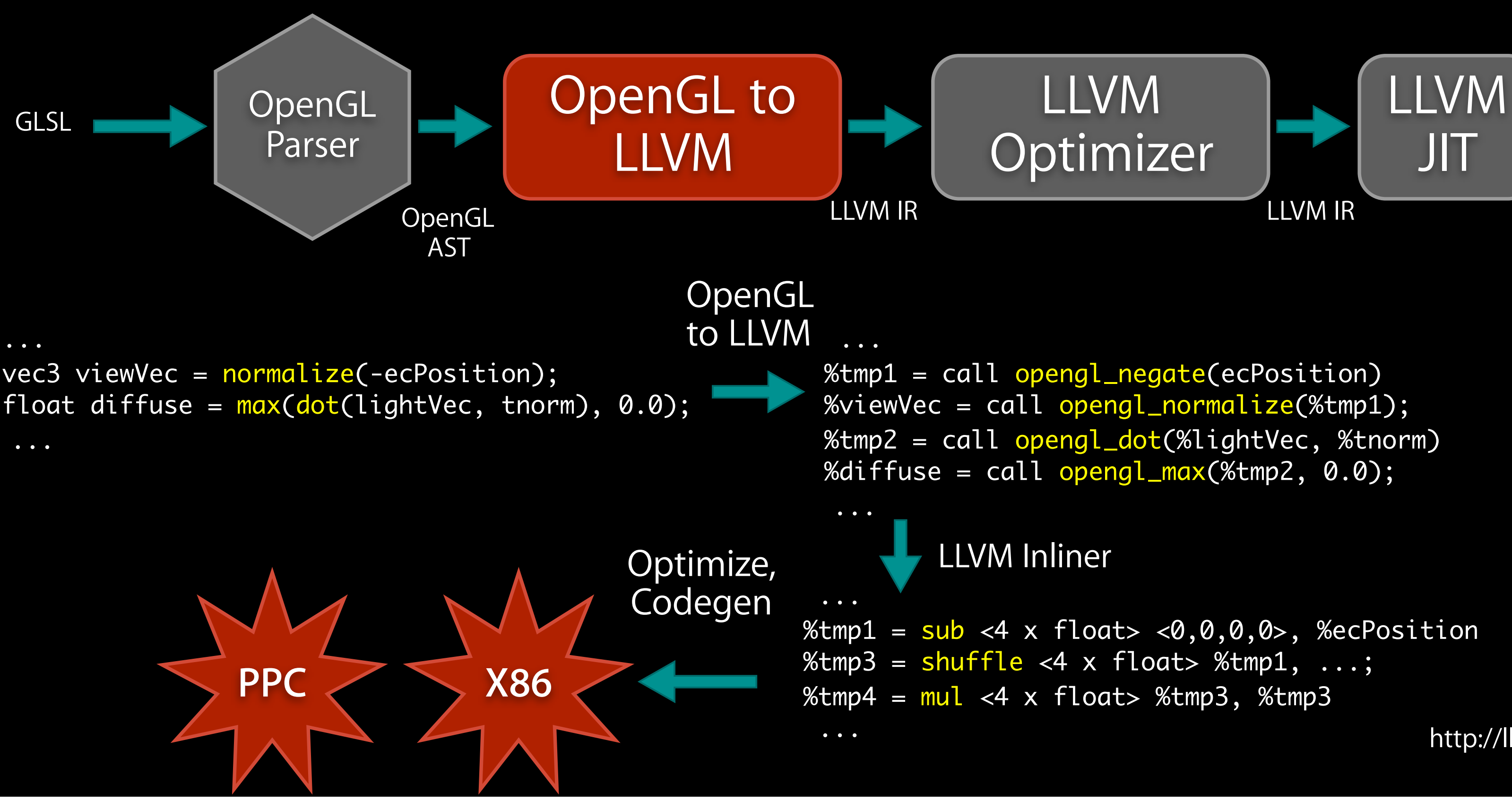
OpenGL to LLVM Implementation



- 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



http://llvm.org/

Benefits of this approach

- Key features:
  - 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
- Primary contributions to Mac OS X:
  - 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!
- You cannot get a polygon onto the screen in Mac OS X without LLVM!

clang Front-end

C C++ Objective-C

Motivation: Why a new front-end?

- GCC's front-end is slow and memory hungry (and getting worse over time)

- GCC doesn't service the diverse needs of an IDE
  - Indexing - scoped variable uses and defs: 'jump to definition' 'doxygen'
  - Static source analysis - 'automatic bug finding'
  - Refactoring - 'Rename variable' 'pull code into a new function'
  - Other source-to-source transformation tools, like 'smart editing'

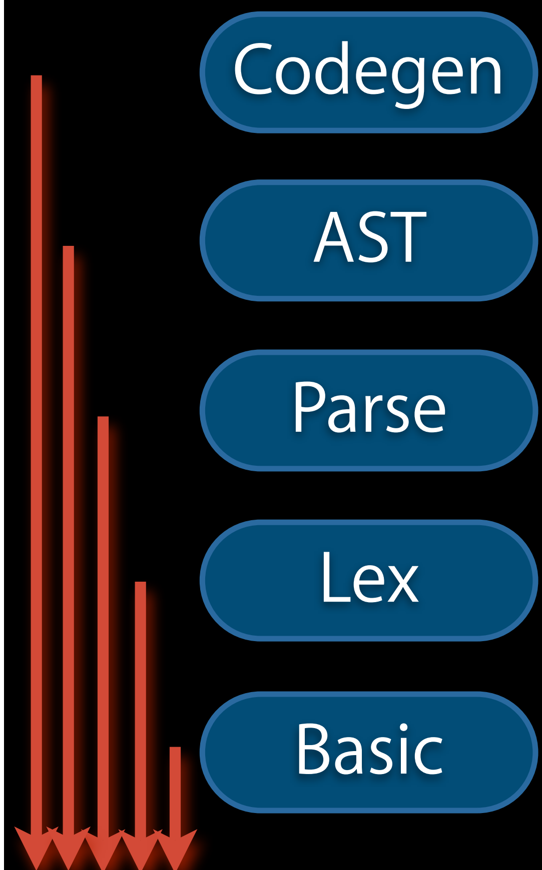
- GCC does not preserve enough source-level information
  - Source code information is lost as the parser runs (trees != source code)
  - Full column numbers, it implicitly folds/simplifies trees as it parses, etc

- GCC's front-end is difficult to work with:
  - Learning curve too steep for many developers
  - Implementation and politics limit innovation
  - GPL License restricts some applications of the front-end

Goals

- Unified parser for C-based languages
  - Language conformance (C, Objective C, C++) & GCC compatibility
  - Good error and warning messages
- Library based architecture with finely crafted C++ API's
  - Useable and extensible by mere mortals
  - Reentrant, composable, replaceable
- Multi-purpose
  - Indexing, static analysis, code generation
  - Source to source tools, refactoring
- High performance!
  - Low memory footprint, fast compiles
  - Support lazy evaluation, caching, multithreading

High Level Architecture



Convert AST to LLVM Code Representation
- Allows use of LLVM optimizer and code generator

Type Checking and Semantic Analysis
- Builds Abstract Syntax Trees (AST) for valid input

Parser for K&R, C90, C99, Objective-C. C++ in development
- Recursive descent parser, does not build syntax tree

Lexing, preprocessing, and pragma handling
- Identifier hash table, tokens, macros, literals

Diagnostics, target description, language dialect control
- Source locations, ranges, buffers, file caching

User Experience: Diagnostics

- Simple things:
  - Each diagnostic has Unique ID (allows fine-grain control)
  - Full column number information is always available and correct:

```
$ clang implicit-def.c -std=c89
implicit-def.c:6:10: warning: implicit declaration of function 'X'
  return X();
         ^
```

```
   struct A { int X; } someA;
   int func(int);

   int test1(int intArg) {
5:   intArg += *(someA.X);
6:   return intArg + func(intArg ? ((someA.X + 40) + someA) / 42 + someA.X : someA.X));
   }
```

```
% gcc t.c
t.c: In function 'test1':
t.c:5: error: invalid type argument of 'unary *'
t.c:6: error: invalid operands to binary +
```

http://clang.llvm.org/

“Expressive” Diagnostics

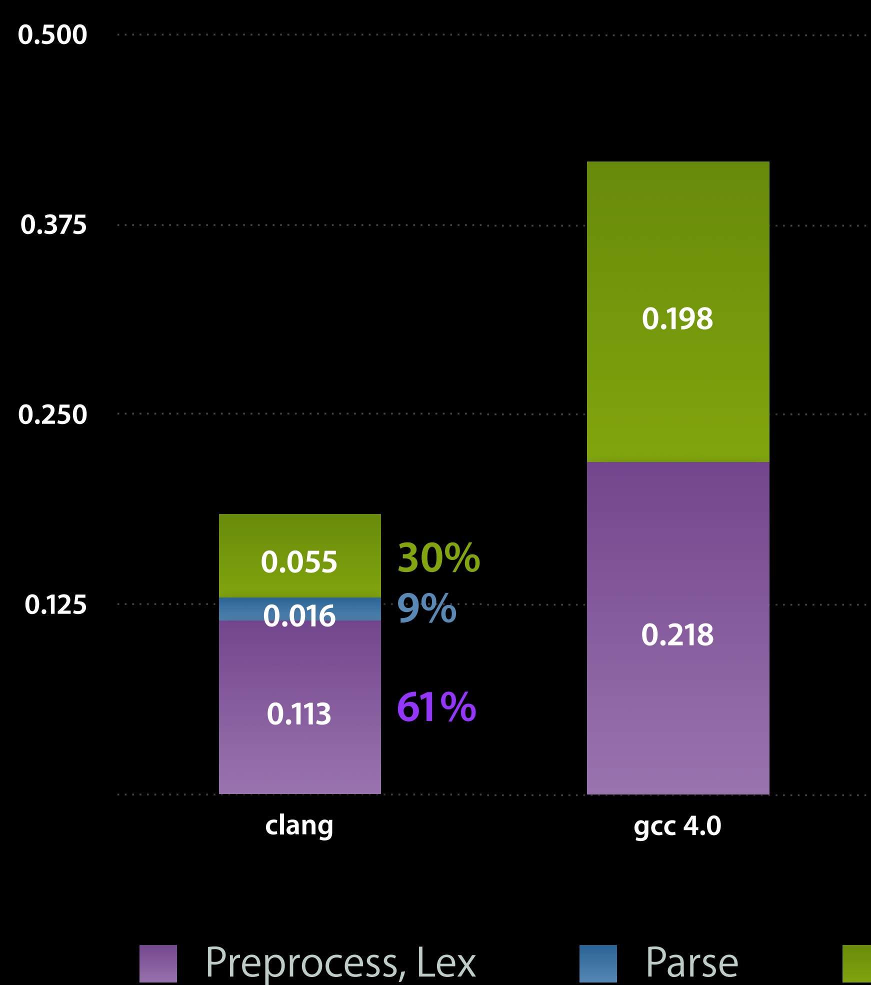
- Other Features:
  - Retains typedef info:
    - std::string instead of std::basic_string<char, std::char_traits<char>, std::allocator<char> >
    - __m128 instead of float __attribute__((__vector_size__(16)))
  - Fine grained location tracking (even through macro instantiations)

```
% clang test.c
t.c:5:13: error: indirection requires pointer operand ('int' invalid)
  intArg += *(someA.X);
            ^~~~~~~~~
t.c:6:49: error: invalid operands to binary expression ('int' and 'struct A')
  return intArg + func(intArg ? ((someA.X+40) + someA) / 42 + someA.X : someA.X));
                                 ~~~~~~~~~~~~ ^ ~~~~~
```

```
% gcc t.c
t.c: In function 'test1':
t.c:5: error: invalid type argument of 'unary *'
t.c:6: error: invalid operands to binary +
```

http://clang.llvm.org/

Carbon.h Parsing / Analysis Time



clang 2.3x faster
0.184s vs 0.416s

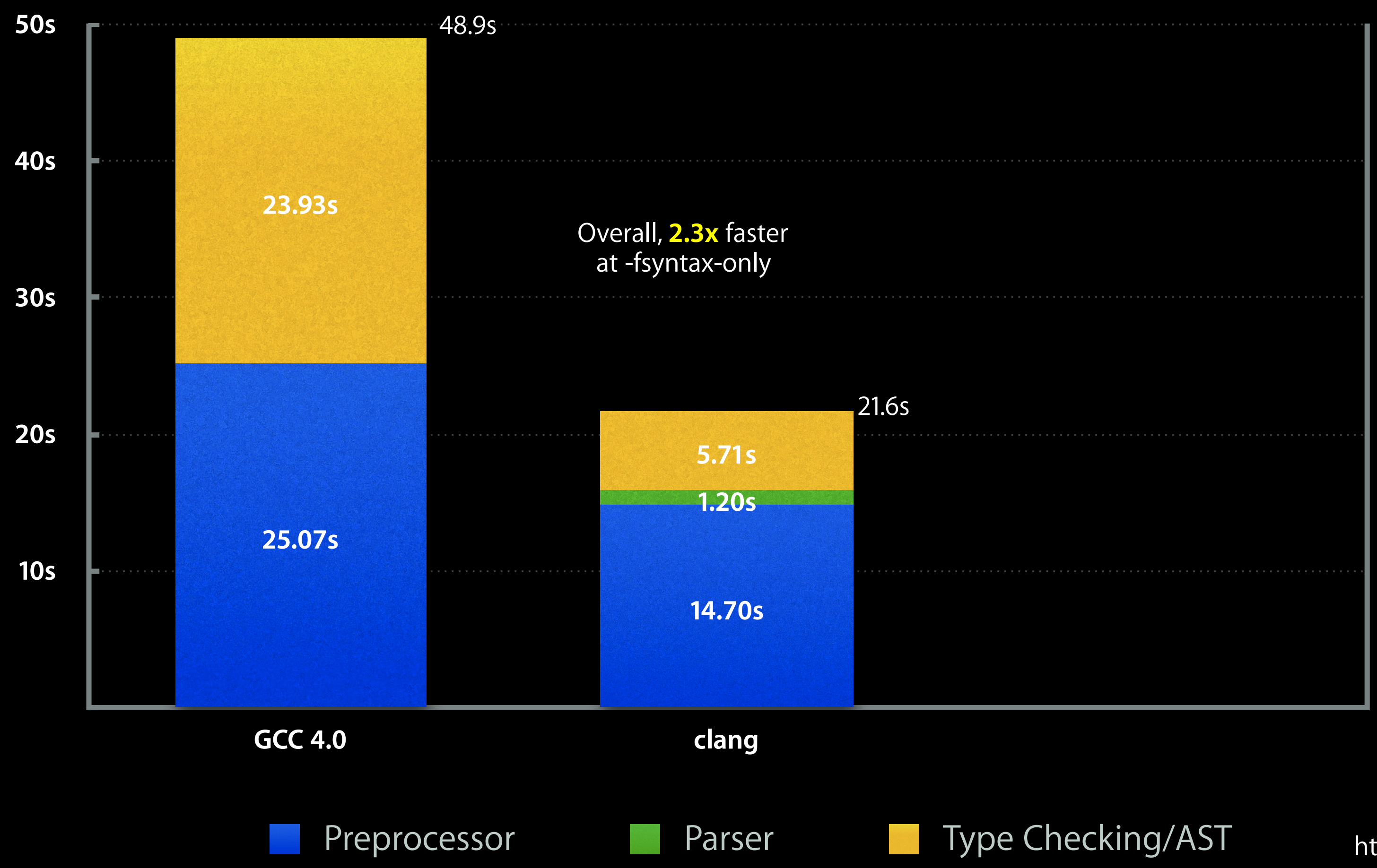
- How big is carbon.h?
  - 558 files
  - 12.3 megabytes!
  - 10,000 function decls
  - 2000 structs, 8000 fields
  - 3000 enums, 20000 enum consts
  - 5000 typedefs
  - 2000 file scoped variables
  - 6000 macros

2.66 Ghz Intel Core 2 Duo (Mac Pro)

http://clang.llvm.org/

PostgreSQL Front-end Times

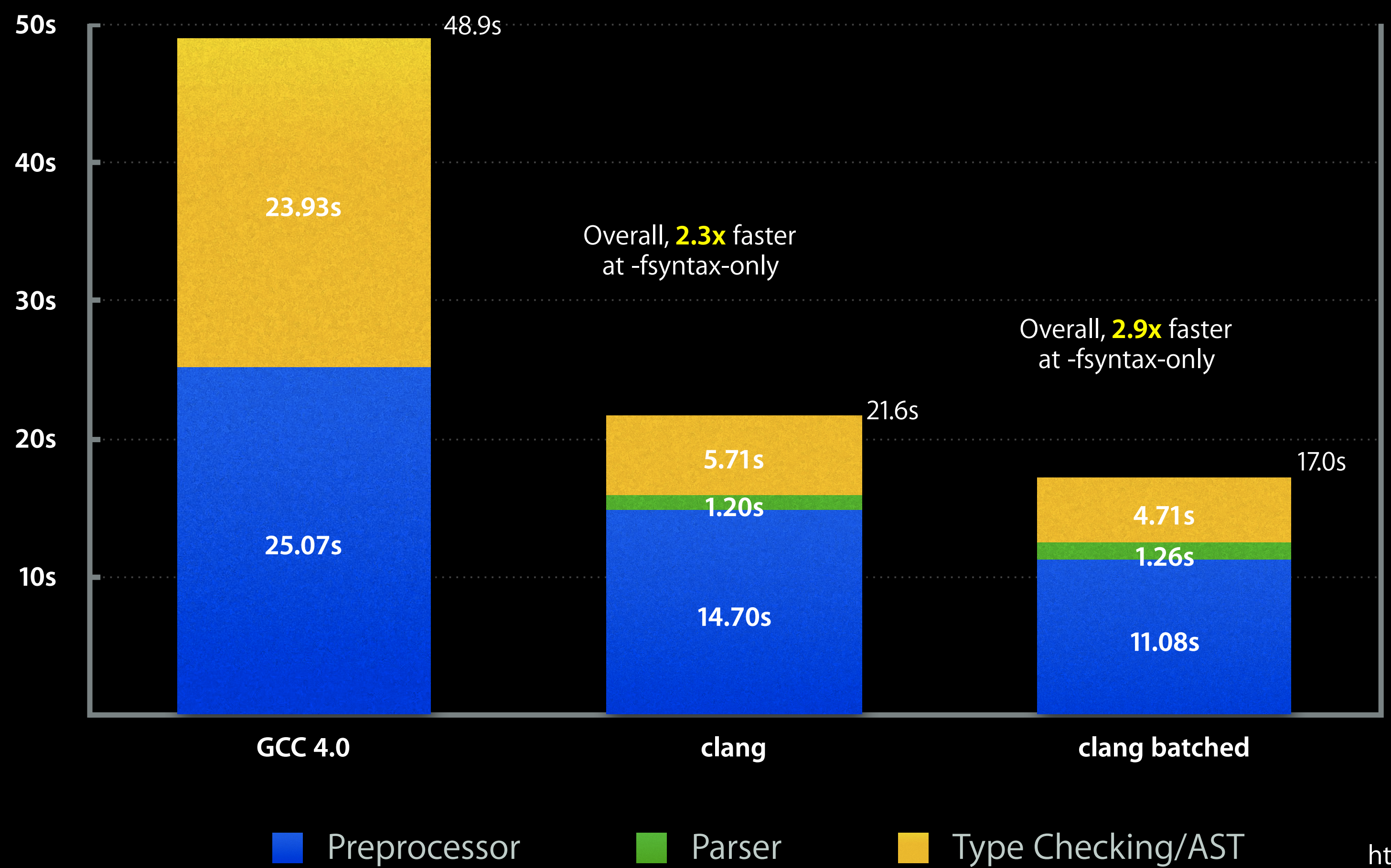
- Medium sized C project:
  - 619 C Files in 665K LOC, not counting headers
  - Timings on a fast 2.66Ghz machine, minimum over 5 runs



http://clang.llvm.org/

PostgreSQL Front-end Times

- Medium sized C project:
  - 619 C Files in 665K LOC, not counting headers
  - Timings on a fast 2.66Ghz machine, minimum over 5 runs



http://clang.llvm.org/

Other Applications of Clang

- Indexing e.g. lxr, doxygen, many IDEs
  - Match uses of variables/functions/etc to definitions
  - Code completion/typeahead, "intellisense"
  - Need to know language rules (scoping, templates, etc) to do correctly
- Refactoring e.g. Eclipse in Java
  - High level restructuring of programs for maintainability and extension
  - e.g. "rename global variable X to Y"
  - Requires language-sensitive analysis, dataflow analysis, for validity checks
- Static Code Analysis e.g. "Coverity Checker"
  - Use dataflow analysis to find obvious bugs in programs
  - Source-level representation allows accurate reporting to user
- All require high performance and accurate model of source code

LLVM Overview

- New compiler architecture built with reusable components
  - Retarget existing languages to JIT or static compilation
  - Many optimizations and supported targets
- llvm-gcc: drop in GCC-compatible compiler
  - Better compile speeds at -O
  - Better optimizer
  - New capabilities
  - Production quality
- Clang front-end: C/ObjC/C++ front-end
  - Several times faster than GCC, fully BSD licensed
  - Much better end-user features (warnings/errors)
  - Still in active development, but solid for C
- LLVM 2.3 release in early June '08!

Come join us at:
http://llvm.org
http://clang.llvm.org