# OpenCL and LLVM

Building An Efficient OpenCL Implementation

# Overview

- What is OpenCL?
- Building OpenCL
  - Front End
  - Runtime



What exactly is CL?
Want to share what we learned building an implementation with LLVM.

# OpenCL

- An API
  - Allocate, Copy, and Share Data
  - Declare Dependencies
  - Execute Programs on both CPU & GPU



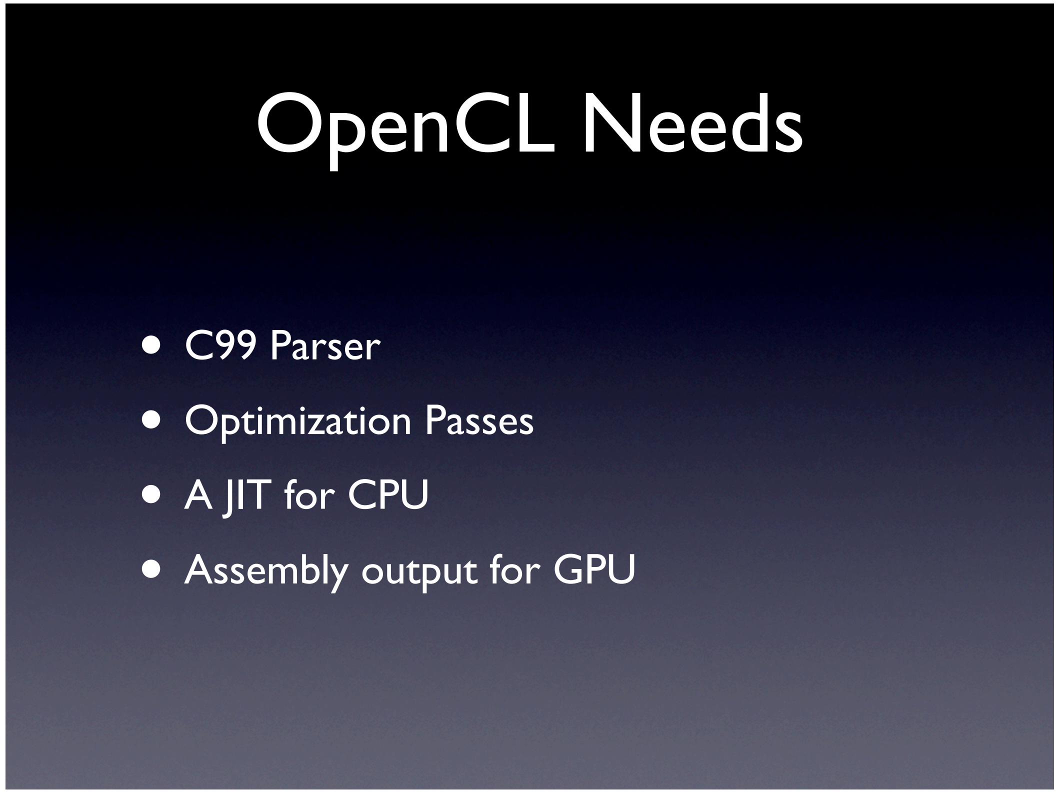
Open Compute Language

# OpenCL

- A Programming Language
  - C99 based
  - Vectors
  - Address Spaces
  - JIT or Binaries



Address spaces primarily for the different types of GPU memory.




5

These needs are for an OpenCL implementation compiling kernels “online”




But after this morning's talk on building a back end in 24 hours, how hard could GPU support have been anyway? :)

# Why Clang?

- C99 and useful extensions
- Easily extensible AST
- Performance focused
- Emits LLVM IR



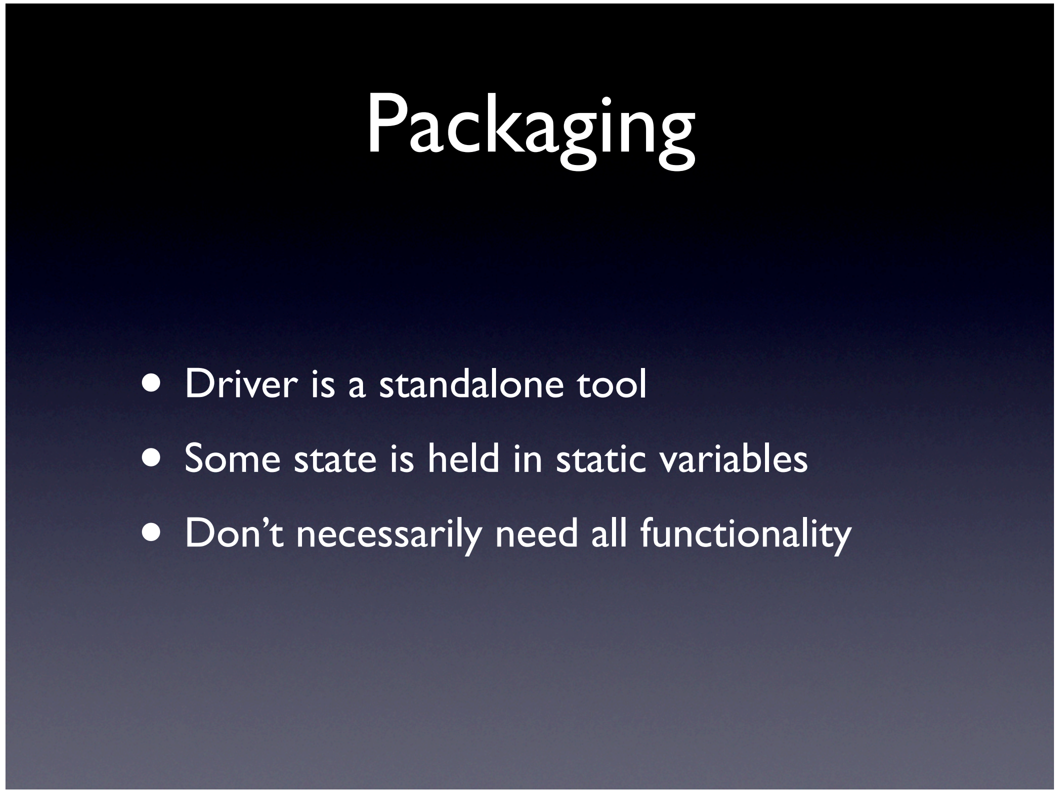
Find a place for 20/80 compile time split info

# Using Clang

- Packaging
- Compile Time
- Memory Footprint



How clang is built and deployed
How fast we can compile OpenCL programs
How much memory we consume while running the compiler






Clang is a set of libraries that are responsible for the various parts of a FE, but there is no “library driver” available.

# libClang

- Library entry point
  - Inputs: source, flags
  - Outputs: module, diagnostics
- No global state
- No use of *llvm::cl* command line parsing.



Flags – the things you would typically pass to a compiler driver (-I, -D, etc.)

# libClang Pseudocode

```
llvm::Module *clang(char* options, char *source, char **log) {
  // 1. Set up diagnostics

  // 2. Initialize language options for OpenCL.

  // 3. Create a Preprocessor, parsing -I and -D from 'options'

  // 4. Create a CodeGenerator and ASTContext.

  // 5. Invoke ParseAST() to run the CodeGen on 'source'.

  // 6. Return the Module and any Diagnostics in 'log'.
}
```



Reduce clang-cc's drive into a simple function that performs 6 basic steps.

# Compile Time

- OpenCL headers can be large
  - >4000 standard lib functions
  - ~350k on disk
- Initially fewer than 20 compiles/s



Places a large burden on clang before ever encountering user code.

# Predeclare Functions

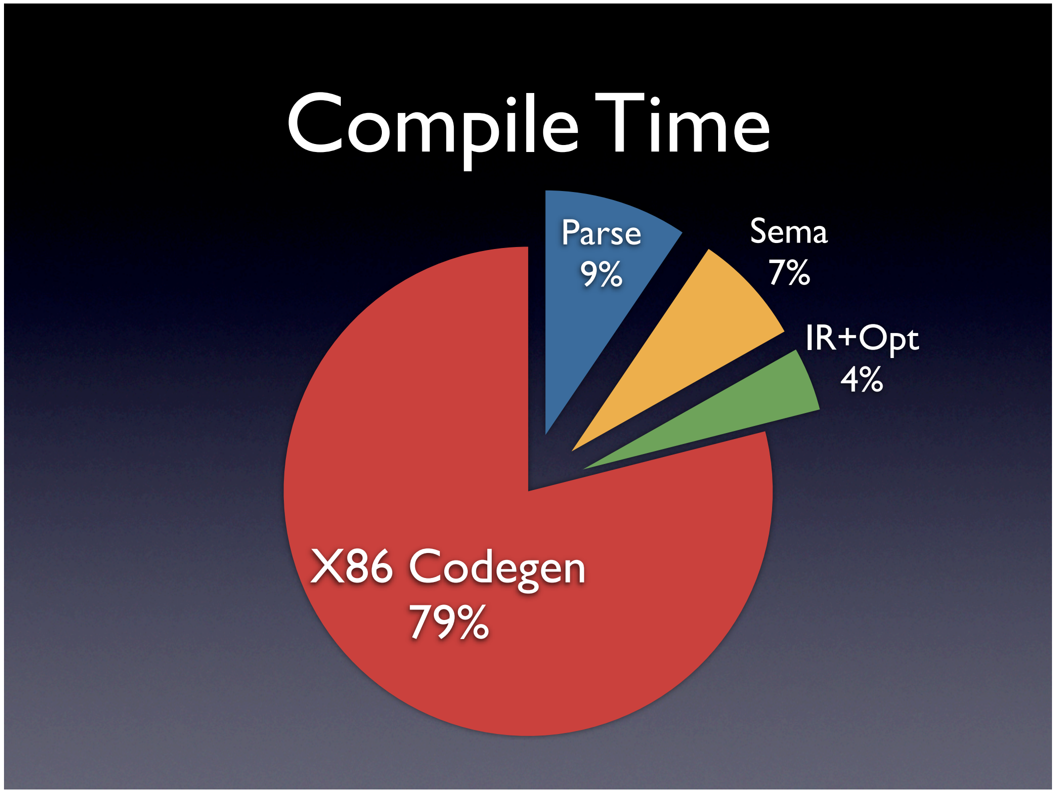
- Extend Builtins.def capabilities
  - Vector type with OpenCL semantics
  - Address Spaces on pointer types
- Fast, but not lazy.

# Example

```
// Example of ExtVector types specified in Builtins
// X16f = vector of 16 floats
// X16f = acosf(X16f)
BUILTIN(__acosf16, "X16fX16f", "fnc")

// Example of Address Spaces for Pointer Types
// X2f*1 = Pointer to vector of 2 floats in address space #1
// X2f = modf(X2f, X2f*1)
BUILTIN(__modfgf2,  "X2fX2fX2f*1", "fnc")
```

# Precompiled Headers

- Declarations are read lazily
- Substantially lower memory use
- Even faster than Builtins.def






Final compile-time breakdown of a typical large CL program

# Memory Footprint

- Leaks
- Heap thrashing
- Dirty page high water mark



Many kinds of memory footprint problems

# Allocation Strategy

- Allocate all AST nodes via ASTContext
- Switch to bump pointer allocator
- Free VM regions rather than individual allocations



All objects in a pool - no leaks when pool is destroyed
Bump ptr allocator - fast!
Free VM regions - high water mark irrelevant

# Allocation Strategy

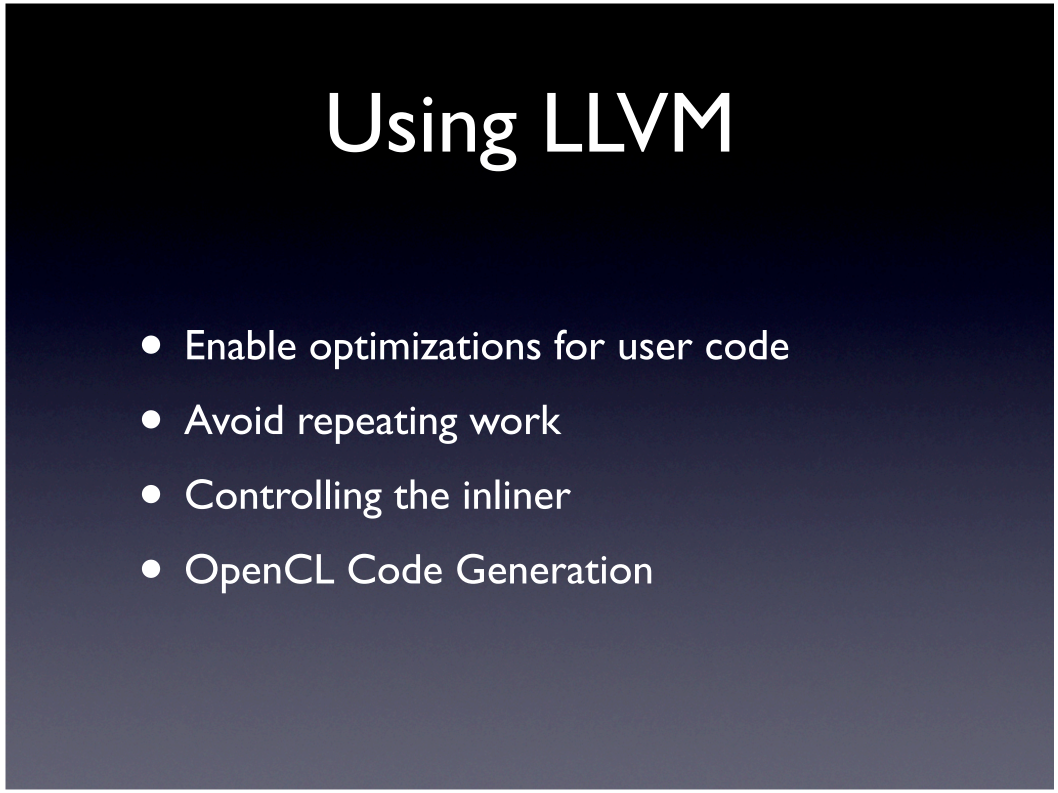
```
new ASTContext(PP->getLangOptions(),
               PP->getSourceManager(),
               PP->getTargetInfo(),
               PP->getIdentifierTable(),
               PP->getSelectorTable(),
DO THIS -----> /* FreeMemory = */ false,
               /* size_reserve = */0,
               /* InitializeBuiltins = */ !usepch);
```

# Clang Summary

- Easy to use as a library
- Fast compile times for small programs
- Memory usage low enough



Just a few dozen lines of code and a new makefile.
Performance is scalable from small shaders to large programs.
Won’t go so far as to say low memory use, but low enough for system use.






Optimize kernels at runtime --> bitcode libraries
ModuleProvider --> basis for bitcode libraries, how do we use it best?
Not all devices are capable of function calls, so the inliner is important.
Also, large module providers place additional demands on it.

# Bitcode Libraries

- Enable IPO w/ User Code
- Deserialization Cost
- Linker Performance
- Covered this last year, but worth a repeat!

# Lazy Deserializing

- Use LLVM *ModuleProvider* to lazily read file
- Link clang output into ModuleProvider
- Run IPO post-link

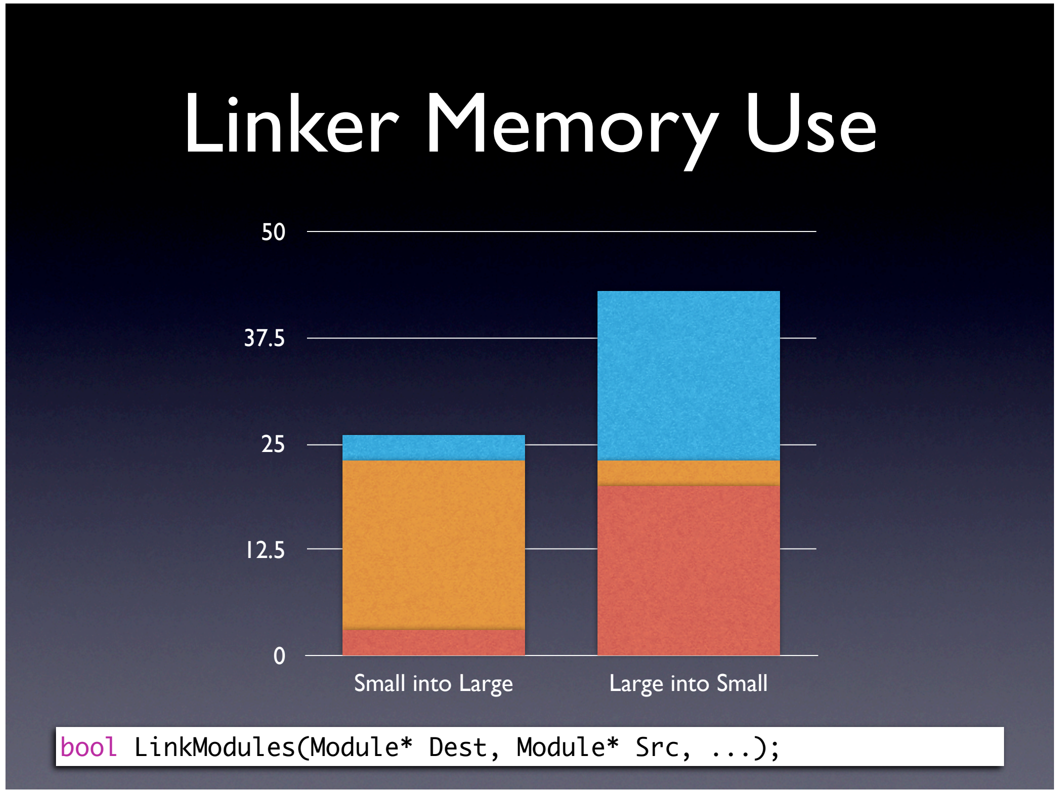
# Lazy Deserializing

```
MemoryBuffer *buffer = MemoryBuffer::getFile("stdlib.bc");
```

```
// Turn bitcode in MemoryBuffer into Module immediately.
Module *Library = ParseBitcodeFile(buffer);
```

↓

```
// Create the runtime library module provider, which will
// lazily stream functions out of the module.
ModuleProvider *MP = getBitcodeModuleProvider(buffer);
Module *Library = MP->getModule();
```






Linker is a smart bitcode copier. Here we have a small module and large module.

# Why Reuse Modules

- Creating a new ModuleProvider for each compile is easy
  - Must register & unregister w/ JIT
  - Must deserialize functions each time

# Module Reuse

- Clean up *ModuleProvider* when we're done
- Create a *GlobalVariable* to remember our *GlobalValues*
- Name lookup no good



What does it mean to reuse a module?

# Module Reuse Part 2

- JIT must have *Value* mappings invalidated
- Erase the “using GV” we made
- Erase each *Value* it referenced
- If you do that backwards, you will assert:

  cannot delete a Value whose #uses != 0



Dropping the mapping lets the JIT free any memory it allocated.

# Example Code

```
// For each GV we are tracking from original clang module,
// Tell JIT to forget about it.
for (i = 0; i != GVOperands.size(); ++i)
  JIT->updateGlobalMapping(GVOperands[i], 0);

// Delete the tracking GV
TrackingGV->eraseFromParent();

// Delete the tracking GV operands.
for (unsigned i = 0; i != GVOperands.size(); ++i)
  GVOperands[i]->eraseFromParent();
```



When the JIT's mapping is updated to null, that means free

# Inlining

- Inliner is a ModulePass
- Module has 1000's of decls
- User code unlikely to use them all



Final issue with re-using a module encompassing the entire bitcode library, inlining is slow.
Just looking at each decl and deciding not to inline it is slow. ~50ms for 5000 decls w/ ghost linkage.

# Inlining Faster

- We can reuse LLVM's Inliner
  - *InlineSimple, InlineCost* are ok, yay!
- Need a new CallGraph Analysis Pass
- Exposes only interesting parts of Module



Reusing LLVM's Inlining Infrastructure, yay!
Default CallGraph pass adds all functions in the module.

# Analysis Passes

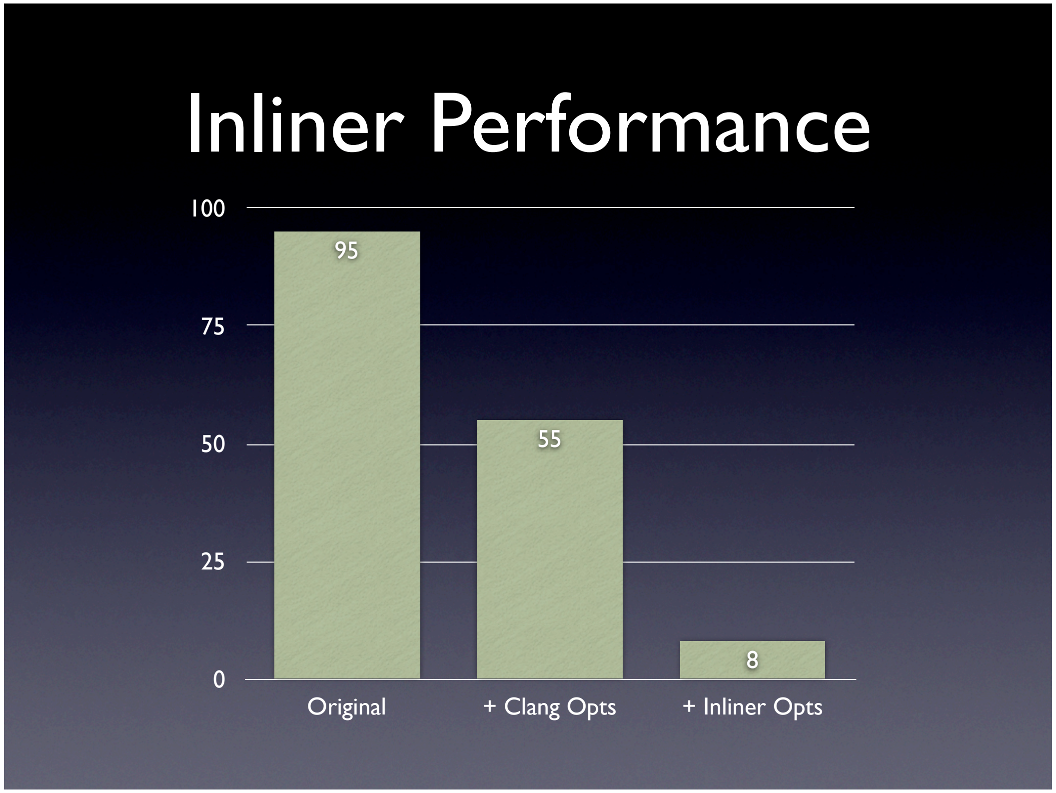
```
// Create a new call graph pass
CLCallGraph *CLCG = new CLCallGraph();
CLCG->initialize(*M);

// Add all interesting functions to the CallGraph
for (ki = Funcs.begin(), ke = Funcs.end(); ki != ke; ++ki)
  CLCG->addToCallGraph(*ki);

// Add our analysis & standard inliner to PassManager
PassManager InlinePM;
InlinePM.add(new TargetData(M));
InlinePM.add(CLCG);
InlinePM.add(createFunctionInliningPass());
InlinePM.run(*M);
```

# Code Example

```
class CLCallGraph : public CallGraph, public ModulePass {
public:

  // destroy - Release memory for the call graph
  virtual void destroy();

  // addToCallGraph - Add a function to the call graph, and
  // link the node to all of the functions that it calls.
  void addToCallGraph(Function *F);

  // runOnModule - do nothing, done by addToCallGraph()
  virtual bool runOnModule(Module &M) { return false; }
};
```



Somewhat simplified, but not much.





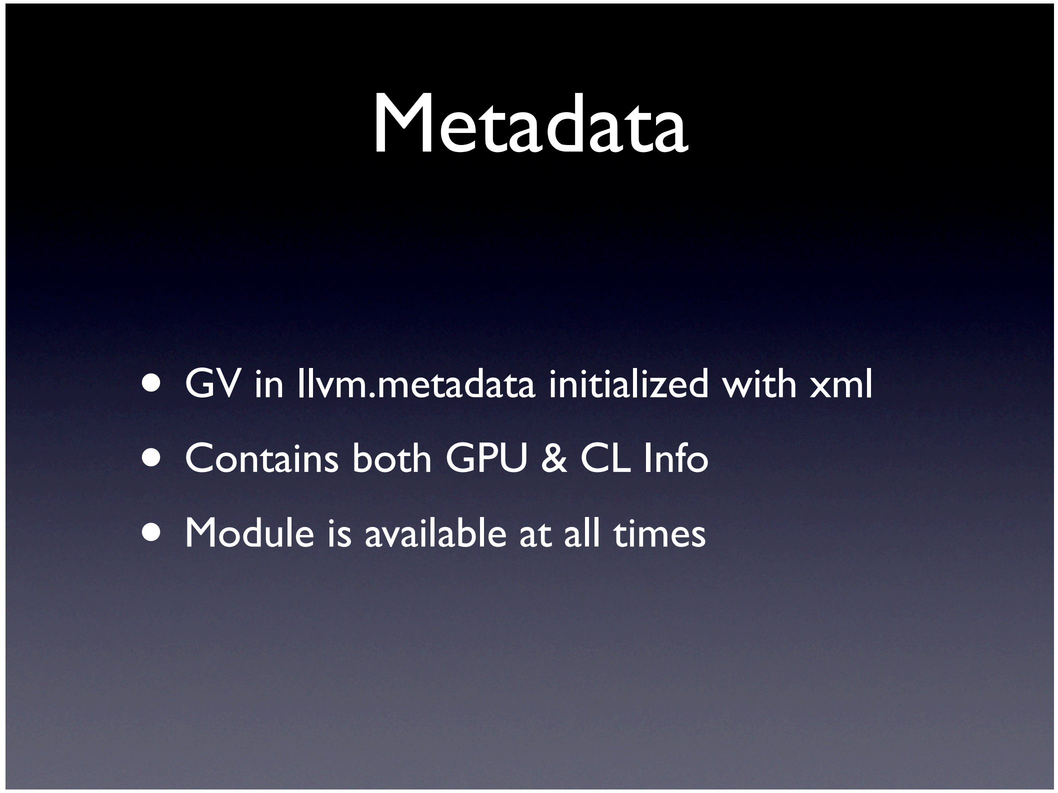

Time in milliseconds to compile and optimize a program

# OpenCL Requirements

- Map LLVM objects to physical resources
- CL Specification Requirements
- Optimization Opportunities



Why?  GPU's need help binding args to texture units, data banks (not virtualized)
Spec also has requirements, and also provides optimization opportunities






Official LLVM metadata didn’t exist yet!
XML is easily read and written by existing libraries, module is around during both clang and codegen.

# Helping the GPU

- Infinite register file
- Stack is a perf hazard
- No calls, merge allocas in inliner



Inlining has potential to bloat stack use.

# OpenCL Opportunities

- Images not readable and writable in CL
  - No aliasing with other inputs
- Spec mandates alignment for types
  - Different vector elements don't alias



Spec mandates alignments, image stuff, etc.  Opens up combiner-aa opportunities in codegen

# LLVM Usage Summary

- Bitcode libs have favorable cost/benefit
- Module reuse ensures efficiency
- Inlining and optimizing fast if controlled



Bitcode libraries enable the features we want, and can have costs minimized

When inputs can be limited to relevant code, inliner and optimizers extremely fast

# Findings

- Clang + LLVM = Efficient OpenCL
- Easily Extensible
- Leverage Community Effort

visit http://khronos.org/opencl/ for more on OpenCL the end.