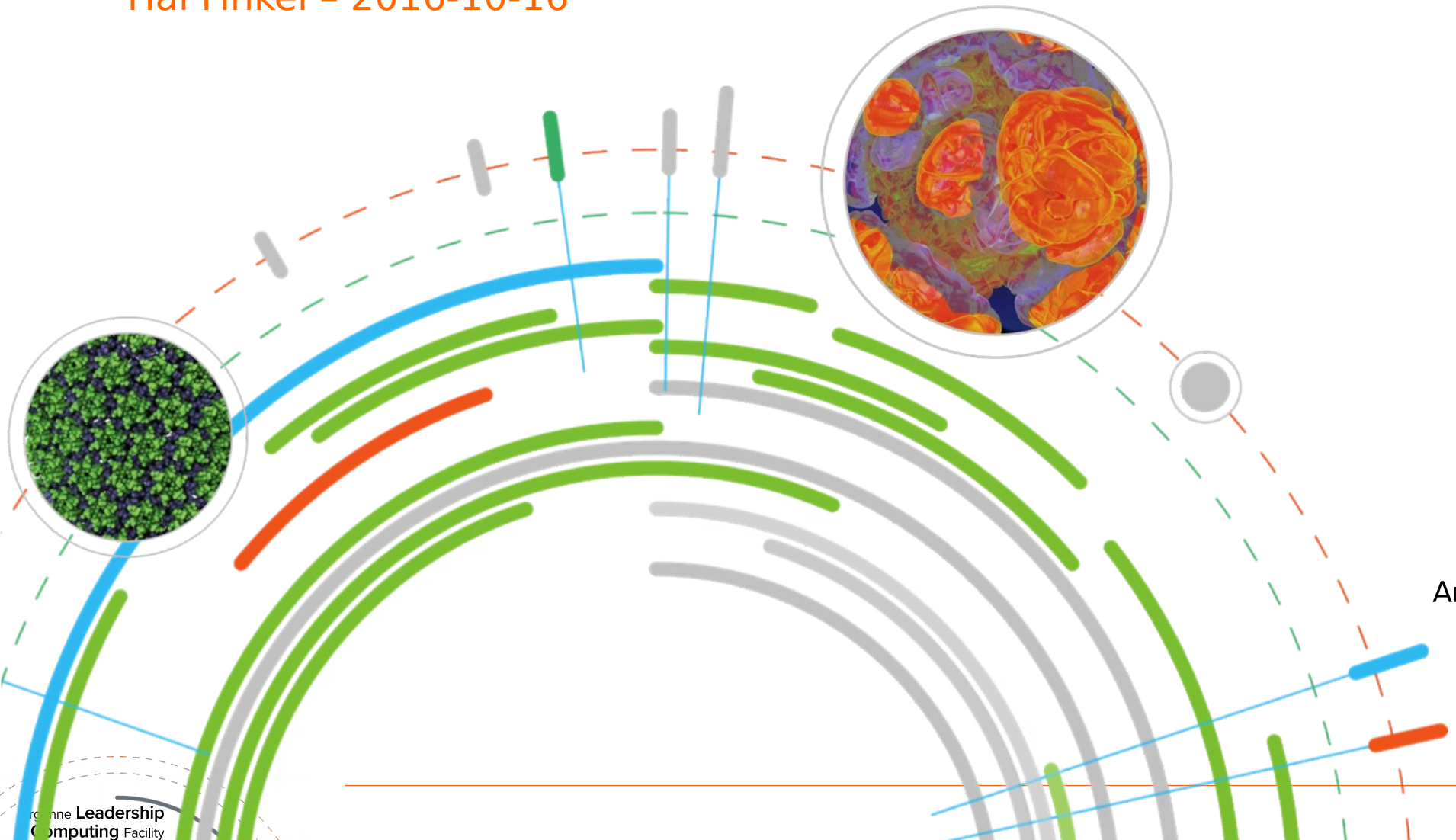
Clang/LLVM: A Modern C++ Compiler with an HPC Twist

Hal Finkel – 2016-10-16

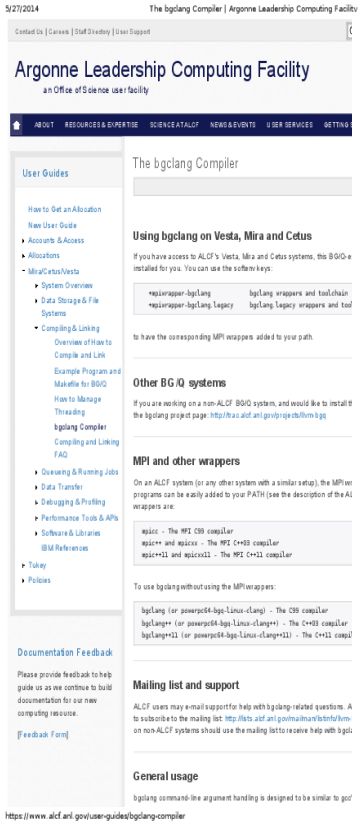


Argonne **Leadership Computing** Facility

Argonne NATIONAL LABORATORY

What we'll cover:

- What is LLVM and Clang?
- MPI-specific warnings and static analysis!
- The thread sanitizer (and other sanitizers)
- Optimization reports


5/27/2014 The bgclang Compiler | Argonne Leadership Computing Facility

Argonne Leadership Computing Facility
an Office of Science user facility

The bgclang Compiler

Using bgclang on Vesta, Mira and Cetus

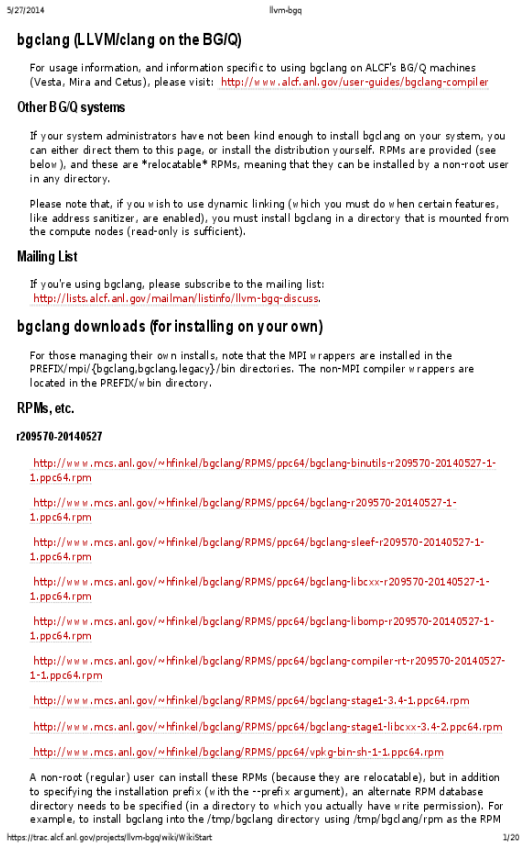
Other BG/Q systems

MPI and other wrappers

Mailing list and support

General usage

https://www.alcf.anl.gov/user-guides/bgclang-compiler


5/27/2014 llvm-bgq

bgclang (LLVM/clang on the BG/Q)

For usage information, and information specific to using bgclang on ALCF's BG/Q machines (Vesta, Mira and Cetus), please visit: http://www.alcf.anl.gov/user-guides/bgclang-compiler

Other BG/Q systems

If your system administrators have not been kind enough to install bgclang on your system, you can either direct them to this page, or install the distribution yourself. RPMs are provided (see below), and these are *relocatable* RPMs, meaning that they can be installed by a non-root user in any directory.

Please note that, if you wish to use dynamic linking (which you must do when certain features, like address sanitizer, are enabled), you must install bgclang in a directory that is mounted from the compute nodes (read-only is sufficient).

Mailing List

If you're using bgclang, please subscribe to the mailing list: http://lists.alcf.anl.gov/mailman/listinfo/llvm-bgq-discuss.

bgclang downloads (for installing on your own)

For those managing their own installs, note that the MPI wrappers are installed in the PREFIX/mpi/{bgclang,bgclang,legacy}/bin directories. The non-MPI compiler wrappers are located in the PREFIX/wbin directory.

RPMs, etc.

r209570-20140527

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-binutils-r209570-20140527-1-1.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-r209570-20140527-1-1.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-sleef-r209570-20140527-1-1.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-libcxx-r209570-20140527-1-1.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-libomp-r209570-20140527-1-1.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-compiler-rt-r209570-20140527-1-1.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-stage1-3.4-1.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-stage1-libcxx-3.4-2.ppc64.rpm

http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/vpkg-bin-sh-1-1.ppc64.rpm

A non-root (regular) user can install these RPMs (because they are relocatable), but in addition to specifying the installation prefix (with the --prefix argument), an alternate RPM database directory needs to be specified (in a directory to which you actually have write permission). For example, to install bgclang into the /tmp/bgclang directory using /tmp/bgclang/rpm as the RPM

https://trac.alcf.anl.gov/projects/llvm-bgq/wiki/WikiStart 1/20


5/27/2014 The LLVM Compiler Infrastructure Project

The LLVM Compiler Infrastructure

Site Map:
Overview
Features
Documentation
CommandGuide
FAQ
Publications
LLVM Projects
Open Projects
LLVM Users
Bug Database

LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a research project at the University of Illinois, with the goal of building a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "UIUC" BSD-style license.

The primary sub-projects of LLVM are:

The **LLVM Core** libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!) These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.

Clang is an "LLVM native" C/C++/Objective-C compiler, which

Latest LLVM Release!

Jan 6, 2014: LLVM 3.4 is now **available for download**! LLVM is publicly available under an open source License. Also, you might want to check out **the new features** in SVN that will appear in the next LLVM release. If you want them early, download LLVM through anonymous SVN.

ACM Software System Award!

LLVM has been awarded the **2012 ACM Software System Award**! This award is given by ACM to *one* software system worldwide every year. LLVM is in highly distinguished company! Click on any of the individual recipients' names on that page for the detailed citation describing the award.

Upcoming Releases

Onward to 3.5!

Developer Meetings

Proceedings from past meetings

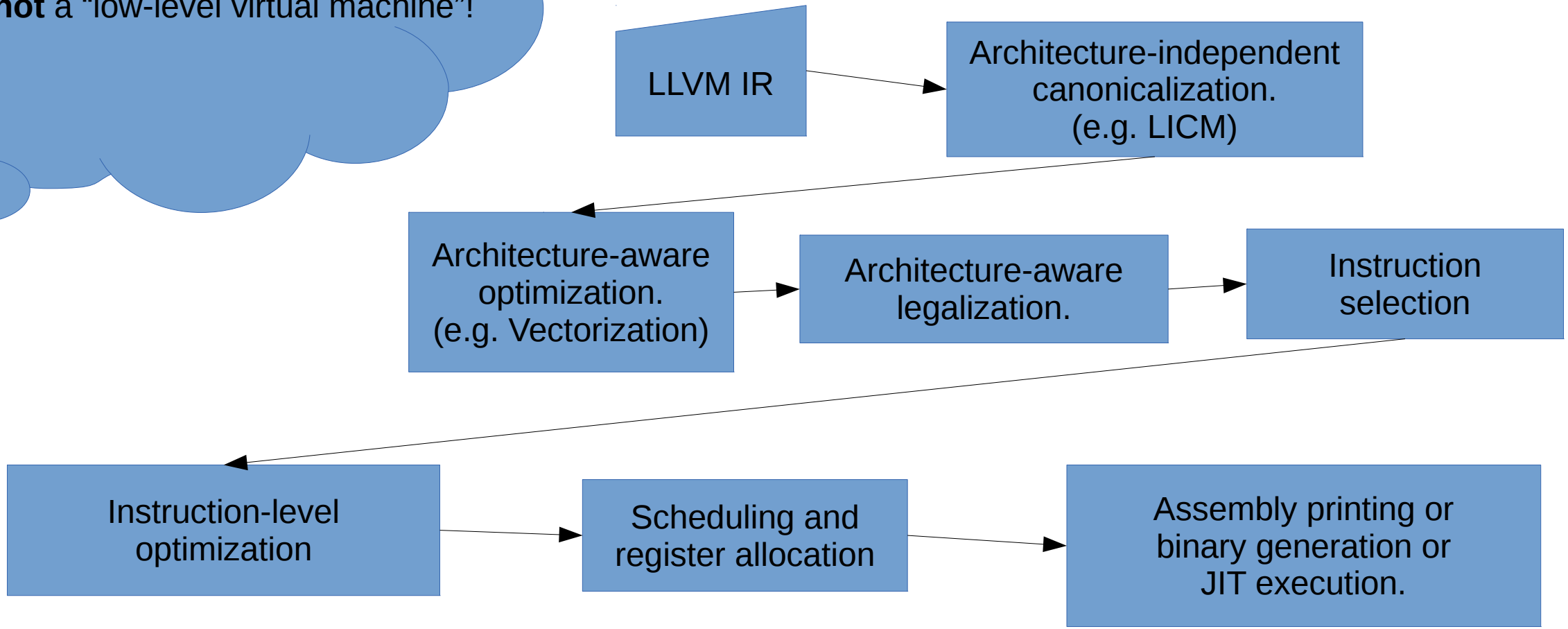
- April 7-8, 2014
- Nov 6-7, 2013
- April 29-30, 2013
- November 7-8, 2012
- April 12, 2012
- November 18, 2011
- September 2011

1/4

What is LLVM:

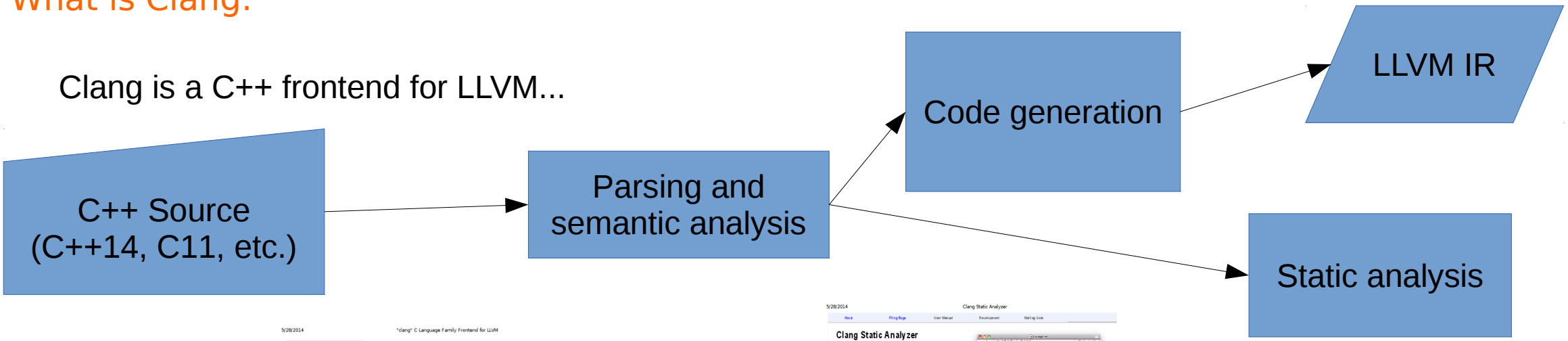
LLVM is **not** a “low-level virtual machine”!

LLVM is a multi-architecture infrastructure for constructing compilers and other toolchain components.



What is Clang:

Clang is a C++ frontend for LLVM...



5/28/2014 "clang": C Language Family Frontend for LLVM

LLVM Home

Clang Info
About
Features
Comparisons
Related Projects
User's Manual
Language Compatibility
Language Extensions
C++ Status

Clang Development
Get Started
Get Involved
Open Projects
Clang Internals
Hacking on Clang

Clang Tools
Automatic Bug Finding
Writing Clang Tools

Communication
cfe-users List
cfe-dev List
cfe-commits List
Bug Reports
Planet Clang
IRC: irc.oftc.net#llvm

The Code
Check Out SVN
Browse SVN
Browse ViewVC
doxygen

Quick Links
Testing Coverage

Clang Events
October 2009
November 2010
LLVM events

clang: a C language family frontend for LLVM

The goal of the Clang project is to create a new C, C++, Objective C and Objective C++ front-end for the LLVM compiler. You can get and build the source today.

Features and Goals

Some of the goals for the project include the following:

End-User Features:

- Fast compiles and low memory use
- Expressive diagnostics (examples)
- GCC compatibility

Utility and Applications:

- Modular library based architecture
- Support diverse clients (refactoring, static analysis, code generation, etc.)
- Allow tight integration with IDEs
- Use the LLVM 'BSD' License

Internal Design and Implementation:

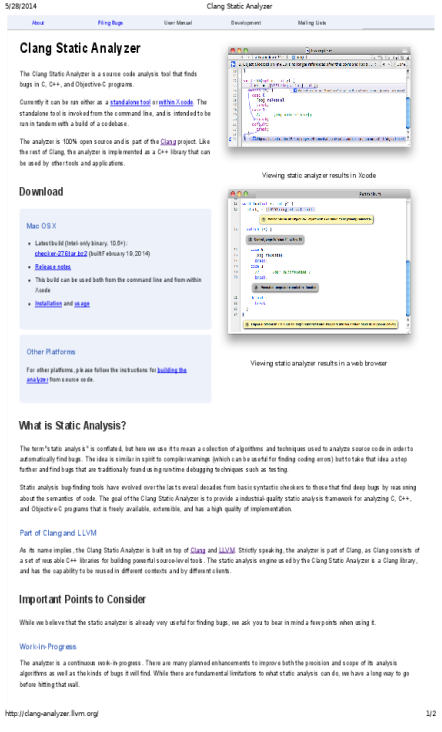
- A real-world, production quality compiler
- A simple and hackable code base
- A single unified parser for C, Objective C, C++, and Objective C++
- Conformance with C/C++/ObjC and their variants

Of course this is only a rough outline of the goals and features of Clang. To get a true sense of what it is all about, see the Features section, which breaks each of these down and explains them in more detail.

Why?

Development of the new front-end was started out of a need for a compiler that allows better diagnostics, better integration with IDEs, a license that is compatible with commercial products, and a nimble compiler that is easy to develop and maintain. All of these were motivations for starting work on a

http://clang.llvm.org/ 1/2



5/28/2014 Clang Static Analyzer

About | Filing Bugs | User Manual | Development | Mailing Lists

Clang Static Analyzer

The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs.

Currently it can be run either as a standalone tool or within Xcode. The standalone tool is invoked from the command line, and is intended to be run in tandem with a build of a codebase.

The analyzer is 100% open source and is part of the Clang project. Like the rest of Clang, the analyzer is implemented as a C++ library that can be used by other tools and applications.

Viewing static analyzer results in Xcode

Download

Mac OS X

- Latest build (Intel-only binary, 10.5+): checker-275.tar.bz2 (built February 19, 2014)
- Release notes
- This build can be used both from the command line and from within Xcode
- Installation and usage

Other Platforms

For other platforms, please follow the instructions for building the analyzer from source code.

Viewing static analyzer results in a web browser

What is Static Analysis?

The term "static analysis" is conflated, but here we use it to mean a collection of algorithms and techniques used to analyze source code in order to automatically find bugs. The idea is similar in spirit to compiler warnings (which can be useful for finding coding errors) but to take that idea a step further and find bugs that are traditionally found during runtime debugging techniques such as testing.

Static analysis bug-finding tools have evolved over the last several decades from basic syntactic checkers to those that find deep bugs by reasoning about the semantics of code. The goal of the Clang Static Analyzer is to provide a industrial-quality static analysis framework for analyzing C, C++, and Objective-C programs that is freely available, extensible, and has a high quality of implementation.

Part of Clang and LLVM

As its name implies, the Clang Static Analyzer is built on top of Clang and LLVM. Strictly speaking, the analyzer is part of Clang, as Clang consists of a set of reusable C++ libraries for building powerful source-level tools. The static analysis engine used by the Clang Static Analyzer is a Clang library, and has the capability to be reused in different contexts and by different clients.

Important Points to Consider

While we believe that the static analyzer is already very useful for finding bugs, we ask you to bear in mind a few points when using it.

Work-in-Progress

The analyzer is a continuous work-in-progress. There are many planned enhancements to improve both the precision and scope of its analysis algorithms as well as the kinds of bugs it will find. While there are fundamental limitations to what static analysis can do, we have a long way to go before hitting that wall.

http://clang-analyzer.llvm.org/ 1/2

MPI-specific warning messages

```
mpit2.c:18:17: warning: argument type 'char *' doesn't match specified 'MPI' type tag that requires 'double *' [-Wtype-safety]
  rc = MPI_Send(&outmsg, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
                ^~~~~~~     ~~~~~~~~~~
```

These are not really MPI specific, but uses the “type safety” attributes inspired by this use case:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype)
    __attribute__(( pointer_with_type_tag(mpi,1,3) ));
…
#define MPI_DATATYPE_NULL ((MPI_Datatype) 0xa0000000)
#define MPI_FLOAT         ((MPI_Datatype) 0xa0000001)
…
static const MPI_Datatype mpich_mpi_datatype_null __attribute__(( type_tag_for_datatype(mpi,void,must_be_null) )) = 0xa0000000;
static const MPI_Datatype mpich_mpi_float         __attribute__(( type_tag_for_datatype(mpi,float) ))            = 0xa0000001;
```

See Clang's test/Sema/warn-type-safety-mpi-hdf5.c, test/Sema/warn-type-safety.c and test/Sema/warn-type-safety.cpp for more examples, and: http://clang.llvm.org/docs/AttributeReference.html#type-safety-checking

MPI-specific static analysis

If you don't know how to run the static analyzer, see: http://clang-analyzer.llvm.org/scan-build.html
Also, this is useful: http://btorpey.github.io/blog/2015/04/27/static-analysis-with-clang/

Proper tools aside, if you have a recent Clang behind your mpicc, for example, you can run something like this:

```
mpicc --analyze -Xanalyzer -analyzer-output=html -Xanalyzer -analyzer-checker=optin.mpi.MPI-Checker \
  -o /tmp/somedir mpit4.c
```

and you might see some warnings like this:

```
mpit4.c:17:5: warning: Request 'sendReq1' has no matching wait.
    MPI_Irecv(&buf, 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &recvReq1);
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mpit4.c:31:5: warning: Double nonblocking on request 'sendReq1'.
    MPI_Irecv(&buf, 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &sendReq1);
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
```

and because we asked for HTML output, we'll also get this (in /tmp/somedir)...

MPI-specific static analysis

Currently handles:

- Missing waits
- Unmatched waits
- Double uses of requests

(plus the static analyzer has lots of non-MPI-specific check too)

http://dl.acm.org/citation.cfm?id=2833159

Annotated Source Code

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int main() {
5
6  }
7
8  void missingWait2() {
9    int rank = 0;
10   double buf = 0;
11   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12   if (rank == 0) {
       1 Assuming 'rank' is not equal to 0 →
       2 ← Taking false branch →
13   } else {
14     MPI_Request sendReq1, recvReq1;
15
16     MPI_Isend(&buf, 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, &sendReq1);
       3 ← Request is previously used by nonblocking call here. →
17     MPI_Irecv(&buf, 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &recvReq1);
       4 ← Request 'sendReq1' has no matching wait.
18     MPI_Wait(&recvReq1, MPI_STATUS_IGNORE);
19   }
20 }
```

Thread-safety warnings

```
class  __attribute__((lockable)) Mutex {
 public:
  void Lock() __attribute__((exclusive_lock_function));
  void ReaderLock() __attribute__((shared_lock_function));
  void Unlock() __attribute__((unlock_function));
  bool TryLock() __attribute__((exclusive_trylock_function(true)));
  bool ReaderTryLock() __attribute__((shared_trylock_function(true)));
  void LockWhen(const int &cond) __attribute__((exclusive_lock_function));
};

Mutex sls_mu;

Mutex sls_mu2 __attribute__((acquired_after(sls_mu)));
int sls_guard_var __attribute__((guarded_var)) = 0;
int sls_guardby_var __attribute__((guarded_by(sls_mu))) = 0;
```

Compile with -Wthread-safety

For more information, see: http://clang.llvm.org/docs/ThreadSafetyAnalysis.html and Clang's test/SemaCXX/warn-thread-safety-analysis.cpp

Thread-safety warnings

A simple example:

```
void sls_fun_bad_2() {
  sls_mu.Lock();
  sls_mu.Lock();
  sls_mu.Unlock();
}
```

```
/tmp/ts.cpp:19:10: warning: acquiring mutex 'sls_mu' that is already held [-Wthread-safety-analysis]
  sls_mu.Lock();
         ^
```

Also integrated with libc++, so you'll automatically get some of the benefit from using std::mutex, std::lock_guard, etc. (you need to pass define _LIBCPP_ENABLE_THREAD_SAFETY_ANNOTATIONS in order to enable this feature)

```
#define _LIBCPP_ENABLE_THREAD_SAFETY_ANNOTATIONS
#include <thread>

std::mutex sls_mu;

void sls_fun_bad_2() {
  sls_mu.lock();
  std::lock_guard<std::mutex> x(sls_mu);
  sls_mu.lock();
  sls_mu.unlock();
}
```

```
/tmp/ts2.cpp:8:31: warning: acquiring mutex 'sls_mu' that is already held [-Wthread-safety-analysis]
  std::lock_guard<std::mutex> x(sls_mu);
                              ^
/tmp/ts2.cpp:9:10: warning: acquiring mutex 'sls_mu' that is already held [-Wthread-safety-analysis]
  sls_mu.lock();
         ^
```

Sanitizers

The sanitizers (some now also supported by GCC) – Instrumentation-based debugging

- Checks get compiled in (and optimized along with the rest of the code) – Execution speed an order of magnitude or more faster than Valgrind
- You need to choose which checks to run at compile time:
  - Address sanitizer: -fsanitize=address – Checks for out-of-bounds memory access, use after free, etc.: http://clang.llvm.org/docs/AddressSanitizer.html
  - Leak sanitizer: Checks for memory leaks; really part of the address sanitizer, but can be enabled in a mode just to detect leaks with -fsanitize=leak: http://clang.llvm.org/docs/LeakSanitizer.html
  - Memory sanitizer: -fsanitize=memory – Checks for use of uninitialized memory: http://clang.llvm.org/docs/MemorySanitizer.html
  - Thread sanitizer: -fsanitize=thread – Checks for race conditions: http://clang.llvm.org/docs/ThreadSanitizer.html
  - Undefined-behavior sanitizer: -fsanitize=undefined – Checks for the execution of undefined behavior: http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
  - Efficiency sanitizer [Recent development]: -fsanitize=efficiency-cache-frag, -fsanitize=efficiency-working-set (-fsanitize=efficiency-all to get both)

And there's more, check out http://clang.llvm.org/docs/ and Clang's include/clang/Basic/Sanitizers.def for more information.

Thread Sanitizer

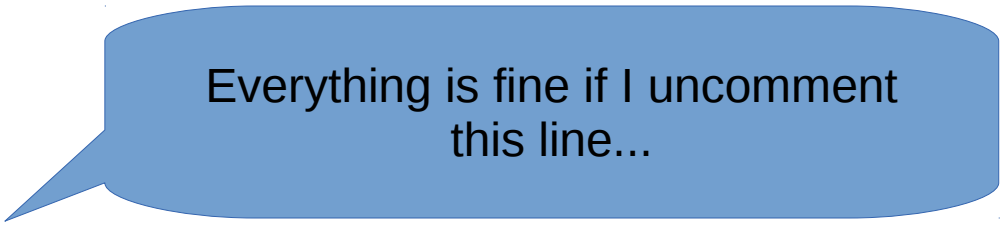
```
#include <thread>

int g_i = 0;
std::mutex g_i_mutex;  // protects g_i

void safe_increment()
{
    // std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;
}

int main()
{
    std::thread t1(safe_increment);
    std::thread t2(safe_increment);

    t1.join();
    t2.join();
}
```



Thread Sanitizer

```
$ clang++ -std=c++11 -stdlib=libc++ -fsanitize=thread -O1 -o /tmp/r1 /tmp/r1.cpp
$ /tmp/r1
```

```
==================
WARNING: ThreadSanitizer: data race (pid=486)
  Write of size 4 at 0x000001521cb8 by thread T2:
    #0 safe_increment() <null> (r1+0x00000049d2ac)
    #1 void* std::__1::__thread_proxy<std::__1::tuple<std::__1::unique_ptr<std::__1::__thread_struct, std::__1::default_delete<std::__1::__thread_stru
ct> >, void (*)()> >(void*) <null> (r1+0x00000049d455)

  Previous write of size 4 at 0x000001521cb8 by thread T1:
    #0 safe_increment() <null> (r1+0x00000049d2ac)
    #1 void* std::__1::__thread_proxy<std::__1::tuple<std::__1::unique_ptr<std::__1::__thread_struct, std::__1::default_delete<std::__1::__thread_stru
ct> >, void (*)()> >(void*) <null> (r1+0x00000049d455)

  Location is global '<null>' at 0x000000000000 (r1+0x000001521cb8)

  Thread T2 (tid=489, running) created by main thread at:
    #0 pthread_create /home/hfinkel/public/src/llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:902 (r1+0x000000420aa5)
    #1 std::__1::thread::thread<void (&)(), , void>(void (&)()) <null> (r1+0x00000049d3b6)
    #2 main <null> (r1+0x00000049d2ea)

  Thread T1 (tid=488, finished) created by main thread at:
    #0 pthread_create /home/hfinkel/public/src/llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:902 (r1+0x000000420aa5)
    #1 std::__1::thread::thread<void (&)(), , void>(void (&)()) <null> (r1+0x00000049d3b6)
    #2 main <null> (r1+0x00000049d2dd)

SUMMARY: ThreadSanitizer: data race (/tmp/r1+0x49d2ac) in safe_increment()
==================
ThreadSanitizer: reported 1 warnings
```

Optimization Reporting - Design Goals

To get information from the backend (LLVM) to the frontend (Clang, etc.)

- To enable the backend to generate diagnostics and informational messages for display to users.
- To enable these messages to carry additional “metadata” for use by knowledgeable frontends/tools
- To enable the programmatic use of these messages by tools (auto-tuners, etc.)
- To enable plugins to generate their own unique messages

```
sqlite3.c:60198:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
  if( sqlite3StrICmp(zLeft, "case_sensitive_like")==0 ){
      ^

sqlite3.c:60200:40: remark: getBoolean inlined into sqlite3Pragma [-Rpass=inline]
      sqlite3RegisterLikeFunctions(db, getBoolean(zRight));
                                       ^

sqlite3.c:60213:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
  if( sqlite3StrICmp(zLeft, "integrity_check")==0
      ^

sqlite3.c:60214:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
   || sqlite3StrICmp(zLeft, "quick_check")==0
      ^

sqlite3.c:44776:8: remark: sqlite3VdbeMemFinalize inlined into sqlite3VdbeExec [-Rpass=inline]
  rc = sqlite3VdbeMemFinalize(pMem, pOp->p4.pFunc);
       ^
```

See also: http://llvm.org/docs/Vectorizers.html#diagnostics

Optimization Reporting - Design Goals

There are two ways to get the diagnotics out:

- By using some fontend that installs the relevant callbacks and displays the associated messages
- By serializing the messages to a file (YAML is currently used as the format), and then processing them with an external tool

```
$ clang -O3 -c -o /tmp/or.o /tmp/or.c -fsave-optimization-record
$ llvm-opt-report /tmp/or.opt.yaml
```

Vectorized by a factor of 4, interleaved by a factor of 2.

Unrolled by a factor of 16

```
< /tmp/or.c
 1           | void bar();
 2           | void foo() { bar(); }
 3           |
 4           | void Test(int *res, int *c, int *d, int *p, int n) {
 5           |   int i;
 6           |
 7           | #pragma clang loop vectorize(assume_safety)
 8      V4,2 |   for (i = 0; i < 1600; i++) {
 9           |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
10           |   }
11           |
12  U16      |   for (i = 0; i < 16; i++) {
13           |     res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
14           |   }
15           |
16 I         |   foo();
17           |
18           |   foo(); bar(); foo();
   I         |   ^
   I         |                ^
19           | }
20           |
```

Optimization Reporting - Design Goals

But how code is optimized often depends on where it is inlined...

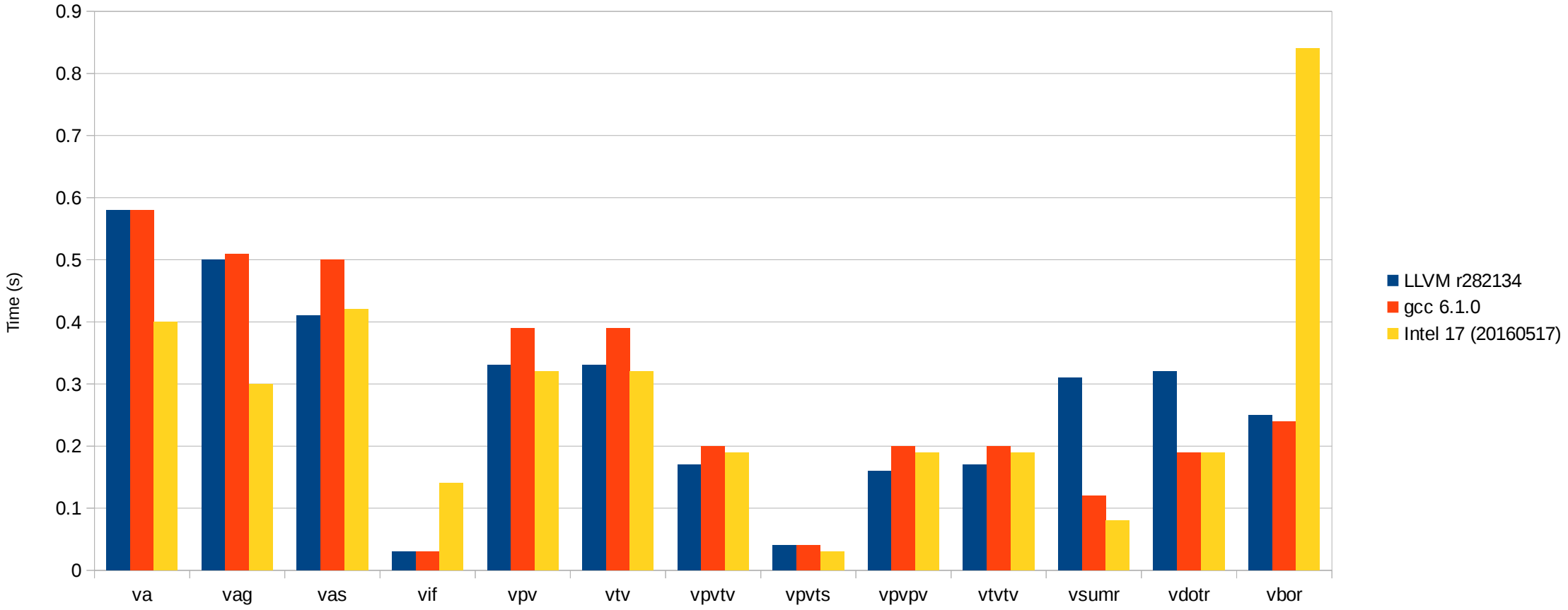
This loop is unrolled when inlined into quack() and quack2(), but not in foo itself.

```
< /tmp/q.cpp
 1      | void bar();
 2      | void foo(int n) {
 [[
  > foo(int):
 3      |   for (int i = 0; i < n; ++i)
  > quack(), quack2():
 3  U4  |   for (int i = 0; i < n; ++i)
 ]]
 4      |     bar();
 5      | }
 6      |
 7      | void quack() {
 8 I    |   foo(4);
 9      | }
10      |
11      | void quack2() {
12 I    |   foo(4);
13      | }
14      |
```

A viewer that creates HTML reports is also under development (the current prototype, called opt-viewer, is in LLVM's utils directory).

LLVM Supports AVX-512! (KNL Performance)

- llvm/projects/test-suite/MultiSource/Benchmarks/TSVC/ControlLoops-dbl



Clang Can Compile CUDA!

CUDA is the language used to compile code for NVIDIA GPUs.

```
$ clang++ axpy.cu -o axpy --cuda-gpu-arch=<GPU arch>
```

For example:
--cuda-gpu-arch=sm_35

When compiling, you may also need to pass --cuda-path=/path/to/cuda if you didn't install the CUDA SDK into /usr/local/cuda (or a few other "standard" locations).

For more information, see: http://llvm.org/docs/CompileCudaWithLLVM.html

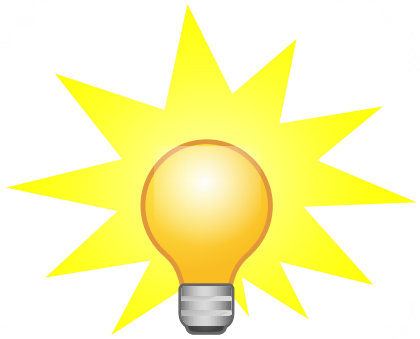
Clang's CUDA aims to provide better support for modern C++ than NVIDIA's nvcc.

Back to vectorization for a moment...

- OpenMP 4 SIMD directives are supported and often useful (as is putting 'restrict' on function arguments and using -ffast-math)

```
void foo(float * __restrict__ A, float * __restrict__ B, float * __restrict__ C) {
  #pragma omp simd
  for (int I = 0; …)
    C[i] = A[i] + B[i];
  }
```

- -fveclib=SVML will generate calls to Intel's vector math-functions library (to autovectorize calls to cos, etc.)



Acknowledgments

- The many contributors to the LLVM community
- ALCF is supported by DOE/SC under contract DE-AC02-06CH11357