The Many Faces of Instrumentation:
Debugging and Better Performance using LLVM in HPC

✔ What are LLVM, Clang, and Flang?
✔ How is LLVM Being Improved for HPC?
✔ What Facilities for Tooling Exist in LLVM?
✔ Opportunities for the Future!

Protocols 2019 @ SC19
2019-11-17

Hal Finkel
Leadership Computing Facility
Argonne National Laboratory
hfinkel@anl.gov
LLVM is a liberally-licensed(*) infrastructure for creating compilers, other toolchain components, and JIT compilation engines.

Clang is a modern C++ frontend for LLVM

LLVM and Clang will play significant roles in exascale computing systems!

(*) Now under the Apache 2 license with the LLVM Exception

LLVM/Clang is both a research platform and a production-quality compiler.
What is LLVM:

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

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

- **LLVM IR**
- **Architecture-independent simplification**
- **Architecture-aware optimization** (e.g. vectorization)
- **Backends** (Type legalization, instruction selection, register allocation, etc.)
- **Assembly printing, binary generation, or JIT execution**
What is Clang:

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

C++ Source (C++14, C11, etc.)

Parsing and semantic analysis

Code generation

Static analysis

LLVM IR

For basic compilation, Clang works just like gcc – using clang instead of gcc, or clang++ instead of g++, in your makefile will likely “just work.”

Clang has a scalable LTO, check out: https://clang.llvm.org/docs/ThinLTO.html
The core LLVM compiler-infrastructure components are one of the subprojects in the LLVM project. These components are also referred to as “LLVM.”
What About Flang?

- Started as a collaboration between DOE and NVIDIA/PGI. Now also involves ARM and other vendors.
- Flang (f18+runtimes) has been accepted to become a part of the LLVM project.
- Two development paths:
  - Flang based on PGI's existing frontend (in C). Production ready including OpenMP support.
  - f18 – A new frontend written in modern C++. Fortran runtime library and vectorized math-function library. Parsing, semantic analysis, etc. under active development.
What About MLIR?

- Started as a part of Google’s TensorFlow project.
- MLIR will become part of the LLVM project.
- MLIR is built around the simultaneous support of multiple dialects.
Clang Can Compile CUDA!

- CUDA is the language used to compile code for NVIDIA GPUs.
- Support now also developed by AMD as part of their HIP project.

$ 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](http://llvm.org/docs/CompileCudaWithLLVM.html)

Clang’s CUDA aims to provide better support for modern C++ than NVIDIA’s nvcc.
Existing LLVM Capabilities

- Clang Static Analysis (including now integration with the Z3 SMT solver)
- Clang Warnings and Provided-by-Default Analysis (e.g., MPI-specific warning messages)
- LLVM-based static analysis (using, e.g., optimization remarks)
- LLVM instrumentation-based checking (e.g., UBSan)
- LLVM instrumentation-based checking using Sanitizer libraries (e.g., AddressSanitizer)
- Lightweight instrumentation for performance collection (e.g., Xray)
- Low-level performance analysis (e.g., llvm-mca)
MPI-specific warning messages

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

```c
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,

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:60200:49: 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]
    if( sqlite3StrICmp(zLeft, "quick_check")==0

sqlite3.c:44776:8: remark: sqlite3VdbeMemFinalize inlined into sqlite3VdbeExec [-Rpass=inline]
    rc = sqlite3VdbeMemFinalize(pMem, p0->p4.pFunc);
```

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

```c
int main(int argc, char **argv) {
    int *array = new int[100];
    delete [] array;
    return array[argc]; } // BOOM
```

```bash
% clang++ -O1 -fsanitize=address a.cc && ./a.out
```

```
==30226== ERROR: AddressSanitizer: heap-use-after-free
READ of size 4 at 0x7fba07fca084 thread T0
    #0 0x40433c in main a.cc:4
0x7fba07fca084 is located 4 bytes inside of 400-byte region freed by thread T0 here:
    #0 0x4058fd in operator delete[](void*) _asan_rtl_
    #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
    #0 0x405579 in operator new[](unsigned long) _asan_rtl_
    #1 0x4042f3 in main a.cc:2
```

Address Sanitizer

ASan shadow memory

Virtual address space

Instrumentation

```
char *shadow = addr >> 3;
if (*shadow)
    ReportError(a);
*a = ...  
```

0xffffffff
0x20000000

0xffffffff
0x04000000

0x03ffffffff
0x00000000

Application

Shadow

mprotect-ed

#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();
}

Everything is fine if I uncomment this line...
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+0x0000049d2ac)
  #1 void* std::_1::_thread_proxy< std::_1::tuple< std::_1::unique_ptr< std::_1::_thread_struct, std::_1::default_delete< std::_1::_thread_struct* > >, void(*)() >>(void*) <null> (r1+0x0000049d455)

  Previous write of size 4 at 0x000001521cb8 by thread T1:
  #0 safe_increment() <null> (r1+0x0000049d2ac)
  #1 void* std::_1::_thread_proxy< std::_1::tuple< std::_1::unique_ptr< std::_1::_thread_struct, std::_1::default_delete< std::_1::_thread_struct* > >, void(*)() >>(void*) <null> (r1+0x0000049d455)

Location is global 'nullptr' at 0x000000000000 (r1+0x000001521cb8)

Thread T2 (tid=489, running) created by main thread at:
  #0 pthread_create /home/hfinkel/public/src/lvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:902 (r1+0x000000420aa5)
  #1 std::_1::thread::thread<void ()(), , void>(void ()()) <null> (r1+0x0000049d3b6)
  #2 main <null> (r1+0x0000049d2ea)

Thread T1 (tid=488, finished) created by main thread at:
  #0 pthread_create /home/hfinkel/public/src/lvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:902 (r1+0x000000420aa5)
  #1 std::_1::thread::thread<void ()(), , void>(void ()()) <null> (r1+0x0000049d3b6)
  #2 main <null> (r1+0x0000049d2dd)

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

Lightweight instrumentation library, add places to patch in instrumentation (generally to functions larger than some threshold):

```assembly
local_block_sled_0:
jmp . + 0x09
(9 bytes worth of nops)
... # function prologue starts, followed by the body.
... # function epilogue starts, just before ret...
local_block_sled_1:
retq
(10 bytes worth of nops)
```

Can be extended to do many things, but comes with an “Flight Data-Recorder” Mode:

```c
// Patch the sleds, if we haven't yet.
auto patch_status = __xray_patch();

// Maybe handle the patch status errors.

// When we want to flush the log, we need to finalize it first, to give
// threads a chance to return buffers to the queue.
auto finalize_status = xray_log_finalize();
if (finalize_status != XRAY_LOG_FINALIZED) {
  // maybe retry, or bail out.
}

// At this point, we are sure that the log is finalized, so we may try
// flushing the log.
auto flush_status = xray_log_flushLog();
if (flush_status != XRAY_LOG_FLUSHED) {
  // maybe retry, or bail out.
}
```

https://llvm.org/docs/XRay.html
LLVM MCA

Using LLVM’s instruction-scheduling infrastructure to analyze programs...

Below is an example of bottleneck analysis output generated by llvm-mca for 500 iterations of the dot-product example on btver2.

Cycles with backend pressure increase [48.07%]
Throughput Bottlenecks:
  Resource Pressure [47.77%]
    - JFPA [47.77%]
    - JFPUI [47.77%]
  Data Dependencies [0.30%]
    - Register Dependencies [0.30%]
    - Memory Dependencies [0.08%]

Critical sequence based on the simulation:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Dependency Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----&gt; 2. vhaddps %xmm3, %xmm3, %xmm4</td>
<td></td>
</tr>
<tr>
<td>&lt; loop carried &gt;</td>
<td></td>
</tr>
<tr>
<td>0. vmulps %xmm0, %xmm1, %xmm2</td>
<td></td>
</tr>
<tr>
<td>-----&gt; 1. vhaddps %xmm2, %xmm2, %xmm3</td>
<td># RESOURCE interference: JFPA [ probability: 74% ]</td>
</tr>
<tr>
<td>-----&gt; 2. vhaddps %xmm3, %xmm3, %xmm4</td>
<td># REGISTER dependency: %xmm3</td>
</tr>
<tr>
<td>&lt; loop carried &gt;</td>
<td></td>
</tr>
<tr>
<td>-----&gt; 1. vhaddps %xmm2, %xmm2, %xmm3</td>
<td># RESOURCE interference: JFPA [ probability: 74% ]</td>
</tr>
</tbody>
</table>

Profile-Guided Optimization

Instrumentation vs. Sampling PGO; for instrumentation:

C0

CompoundStmt

IfStmt

Stmt

Stmt

Stmt

C1

Instrumentation vs. Sampling PGO; for sampling:

Source Code → \(-O2\) +gline-tables-only → Base Optimized Binary

Peak Optimized Binary → \(-O2\) -fprofile-sample-use +gline-tables-only → Profile

Execute under profiler (low overhead)
early results

spec 2006 int (x86_64)

- llvm (-o2)
- gcc 4.8-google (-o2)
- llvm (-o2, pgo)
- gcc 4.8-google (-o2, pgo)

benchmark

90%
not 0-based!

define void @f(il %a) {
entry:
... 
  br il %a, label %t, label %f, !prof !0

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
Link-Time Optimization

LTO

Highly parallel frontend processing + initial optimizations

Link all bitcode in one single Module

Monolithic LTO Implementation

Single-threaded very boring usual optimizations

Potentially threaded CodeGen

LTO

Phase 1: Compile

Phase 2: Thin Link

Phase 3: Backends

Fully-parallel frontend processing + initial optimizations
Extra per-function summary information are generated “on the side”
Link only the summary info in a giant index: thin-link.
No need to parse the IR
Fully-parallel cross-module function
Importing based on summary.
Imported functions are dropped after inlining.
Fully-parallel (very boring) usual optimizations and CodeGen

Traditional Linking

LTO

Run-time Performance: SPEC cpu2006
Improvement over -O2 (all with PGO)

Monolithic LTO + PGO
ThinLTO + PGO

A role in exascale? Current/Future HPC vendors are already involved (plus many others)...

Apple + Google (Many millions invested annually) + many others (Qualcomm, Sony, Microsoft, Facebook, Ericsson, etc.)

ARM

IBM

NVIDIA (and PGI)

AMD

Intel

Cray

Academia, Labs, etc.
SOLLVE: OpenMP (WBS 2.3.1.13)

- Enhancing the implementation of OpenMP in LLVM:
  - Developing support for unified memory (e.g., from NVIDIA), kernel decomposition and pipelining, automated use of local memory, and other enhancements for accelerators.
  - Developing optimizations of OpenMP constructs to reduce overheads (e.g., from thread startup and barriers).
- Building on LLVM parallel-IR work in collaboration with Intel.
- Using LLVM, Clang, and Flang to prototype new OpenMP features for standardization.
- Developing an OpenMP test suite, and as a result, testing and improving the quality of OpenMP in LLVM, Clang, and Flang.

PROTEAS: Parallel IR & More (WBS 2.3.2.09)

- Developing extensions to LLVM’s intermediate representation (IR) to represent parallelism.
  - Strong collaboration with Intel and several academic groups.
  - Parallel IR can target OpenMP’s runtime library among others.
  - Parallel IR can be targeted by OpenMP, OpenACC, and other programming models in Clang, Flang, and other frontends.
  - Building optimizations on parallel IR to reduce overheads (e.g., merging parallel regions and removing redundant barriers).
- Developing support for OpenACC in Clang, prototyping non-volatile memory features, and integration with Tau performance tools.

Y-Tune: Autotuning (WBS 2.3.2.07)

- Enhancing LLVM to better interface with autotuning tools.
- Enhancing LLVM’s polyhedral loop optimizations and the ability to drive them using autotuning.
- Using Clang, and potentially Flang, for parsing and semantic analysis.

Kitsune: LANL ATDM Dev. Tools (WBS 2.3.2.02)

- Using parallel IR to replace template expansion in FleCSI, Kokkos, RAJA, etc.
- Enhanced parallel-IR optimizations and targeting of various runtimes/architectures.
- Flang evaluation, testing, and Legion integration, plus other programming-model enhancements.
- ByFi: Instrumentation-based performance counters using LLVM.

Flang: LLVM Fortran Frontend (WBS 2.3.5.06)

- Working with NVIDIA (PGI), ARM, and others to develop an open-source, production-quality LLVM Fortran frontend.
  - Can target parallel IR to support OpenMP (including OpenMP offloading) and OpenACC.

Note: The proxy-apps project (WBS 2.2.6.01) is also enhancing LLVM’s test suite.
Composition of Transformations
Order is Important

```c
#pragma omp unroll factor(2)
#pragma omp reverse
for (int i = 0; i < 128; i+=1)
    Stmt(i);

#pragma omp unroll factor(2)
for (int i = 127; i >= 0; i-=1)
    Stmt(i);

for (int i = 127; i >= 0; i-=1) {
    Stmt(i);
    Stmt(i-1);
}
```

```c
#pragma omp reverse
#pragma omp unroll factor(2)
for (int i = 0; i < 128; i+=1)
    Stmt(i);

#pragma omp reverse
for (int i = 0; i < 128; i+=2) {
    Stmt(i);
    Stmt(i+1);
}
```

```c
for (int i = 126; i >= 0; i-=2) {
    Stmt(i);
    Stmt(i+1);
}
```
Matrix-Matrix Multiplication

```c
void matmul(int M, int N, int K,
    double C[const restrict static M][N],
    double A[const restrict static M][K],
    double B[const restrict static K][N]) {

    #pragma clang loop(j2) pack array(A)
    #pragma clang loop(i1) pack array(B)
    #pragma clang loop(i1, j1, k1, i2, j2) interchange \
        permutation(j1, k1, i1, j2, i2)
    #pragma clang loop(i, j, k) tile sizes(96, 2048, 256) \
        pit_ids(i1, j1, k1) tile_ids(i2, j2, k2)

    #pragma clang loop id(i)
    for (int i = 0; i < M; i += 1)
        #pragma clang loop id(j)
        for (int j = 0; j < N; j += 1)
            #pragma clang loop id(k)
            for (int k = 0; k < K; k += 1)
                C[i][j] += A[i][k] * B[k][j];
}
```
Matrix-Matrix Multiplication

After Transformation

do double Packed_B[256][2048];
do double Packed_A[96][256];
if (runtime check) {
  if (N >= 1) {
    for (int c0 = 0; c0 <= floor(N - 1, 2048); c0 += 1) // Loop j1
      for (int c1 = 0; c1 <= floor(K - 1, 256); c1 += 1) { // Loop k1

// Copy-in: B -> Packed_B
  for (int c4 = 0; c4 <= min(2047, N - 2048 * c0 - 1); c4 += 1)
    for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1)
      Packed_B[c4][c5] = B[256 * c1 + c5][2048 * c0 + c4];

    for (int c2 = 0; c2 <= floor(N - 1, 96); c2 += 1) { // Loop j1

// Copy-in: A -> Packed_A
      for (int c6 = 0; c6 <= min(95, M - 96 * c2 - 1); c6 += 1)
        for (int c7 = 0; c7 <= min(255, K - 256 * c1 - 1); c7 += 1)
          Packed_A[c6][c7] = A[96 * c2 + c6][256 * c1 + c7];

        for (int c3 = 0; c3 <= min(2047, M - 2048 * c0 - 1); c3 += 1) // Loop j2
          for (int c4 = 0; c4 <= min(95, M - 96 * c2 - 1); c4 += 1) // Loop i2
            for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1) // Loop k2
              C[96 * c2 + c4][2048 * c0 + c3] = Packed_A[c4][c5] * Packed_B[c3][c5];
    }
  }
} else {
  /* original code */
}
Matrix-Matrix Multiplication

Execution Speed

- theoretical peak: 0.53s
- Intel MKL 2018.3: 0.59s (89%)
- OpenBLAS: 0.64s (83%)
- ATLAS: 0.9s (60%)
- Polly MatMul: 1.25s (42%)
- OpenBLAS*: 1.27s (42%)
- #pragma clang loop: 2.2s (24%)
- ATLAS*: 2.2s (24%)
- manual replication: 3.9s (14%)
- Netlib CBLAS*: 33.5s (1.6%)
- -O3 -march=native: 74.9s (0.7%)

* Pre-compiled from Ubuntu repository
What To Do With OpenACC Code?

Clacc: OpenACC Support for Clang and LLVM

Who
- Joel E. Denny (ORNL)
- Seyong Lee (ORNL)
- Jeffrey S. Vetter (ORNL)

Where
- https://ft.ornl.gov/research/clacc
- Clacc Poster (Wed at ECP AHM)

What
- Develop production-quality, standard-conforming traditional OpenACC compiler and runtime support by extending Clang and LLVM
- Enable research and development of source-level OpenACC tools
  - Design compiler to leverage Clang/LLVM ecosystem extensibility
  - E.g., Pretty printers, analyzers, lint tools, and debugger and editor extensions
- As matures, contribute OpenACC support to upstream Clang and LLVM
- Throughout development
  - Actively contribute upstream all mutually beneficial Clang and LLVM improvements
  - Actively contribute to the OpenACC specification
Optimizing Parallel Programs with LLVM

Penalty caused by (seq. execution of) an OpenMP parallel loop.

Performance is recovered by native compiler optimizations

Reuse "scalar" code
- constant propagation
- argument promotion
- attribute deduction
- ...

New "parallel" optimizations
- barrier elimination
- parallel region expansion
- parallelism aware code motion
- ...

See our IWOMP’18 & LCPC’18 papers, as well as the LLVMDev’18 talk/video!
Opportunities for the Future

- Race-Detection Tools and other Sanitizers in HPC
  - Scalable Data Collection
  - Integration with MPI or other inter-node communication frameworks
  - Support on GPUs and other accelerators
- More static analysis, both frontend and optimizer, for HPC
  - Support for MPI
  - Support for Fortran
  - Support for GPUs and other accelerators
  - Support for advanced loop optimizations and other user-directed optimizations
- FDR-like capabilities for large-scale HPC applications
  - Debugging crashes at scale is hard.
- Integrated dynamic and static performance analysis (e.g., using MCA-like capabilities)
  - Better understanding of performance counters
  - Understanding of working sets and cache populations
  - Support for GPUs and other accelerators
- Better support for LTO and PGO in HPC environments
  - Scalable data collection (for PGO)
  - Build-system integration, LTO-enabled libraries, etc.
  - Support for GPUs and other accelerators
Acknowledgments

Thanks to ALCF, ANL, ECP, DOE, and the LLVM community!

ALCF is supported by DOE/SC under contract DE-AC02-06CH11357.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.