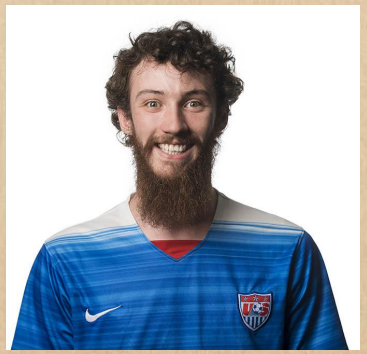
# LLVM Performance Tutorial

wmoses@mit.edu



jdoerfert@anl.gov

# Part 0

General Setup &
Recommendations

# Building LLVM yourself

Single command often suffices to configure:

```
cmake …/llvm-project/llvm -DLLVM_ENABLE_PROJECTS='clang;lld' -DLLVM_ENABLE_RUNTIMES=’openmp’
make -j
```

Useful options include:

```
CMAKE_BUILD_TYPE={Release,Asserts,…}
LLVM_ENABLE_ASSERTIONS={ON,OFF}
LLVM_CCACHE_BUILD={ON,OFF}
-G Ninja
```

May need debug build to debug certain compiler-based issues,

**release + assert** is often used as trade off

Various resources available online! Start here:

http://llvm.org/docs/GettingStarted.html

# General Recommendations

- Use a fast linker (`lld`), `ccache`, and `ninja`
- Consider LTO, either thin or full
- Use tooling (`clang-format`, `clang-tidy`, `clang-modernize`, …)
- Use `-O3`/`Ofast -march=native` as default
- Online documentation is not great but often not bad either
- Debug with sanitizers enabled
- A release + asserts build is best for every-day use

# Ask the LLVM Community

Many ways to interact:

- Discourse (forum/mailing list)
- Discord (persistent chat)
- IRC (non-persistent chat)
- Online Sync-Ups:
  - AA, MLIR, ML, RISC-V, …
- Office Hours ***NEW***
  - "AMA" with an "expert"
- Meetups (soon again!)

# Part 1

Locating the Problem

# Perf

Binary Instrumentation Tool

- Provides hardware performance counters
- Samples program at intervals to see where time is being spent
- Compiling with debug info (-g) can provide more source-level information

```
wmoses@beast:~LULESH $ perf record --call-graph=fp ./lulesh.exe -s 50
```

# Perf

- Can view the call trace of the program and which calls are taking the most time

```
wmoses@beast:~LULESH $ perf report
```

```
Samples: 262K of event 'cycles', Event count (approx.): 282455124599
  Children      Self  Command          Shared Object          Symbol
-   92.55%    78.10%  ser-single-forw  ser-single-forward.exe [.] LagrangeLeapFrog
   - 42.52% LagrangeLeapFrog
      - 13.52% page_fault
         + 11.93% do_page_fault
   + 18.73% 0
+   22.99%     0.00%  ser-single-forw  [unknown]              [.] 0000000000000000
+   13.58%     1.59%  ser-single-forw  [kernel.kallsyms]      [k] page_fault
+   11.98%     0.04%  ser-single-forw  [kernel.kallsyms]      [k] do_page_fault
+   11.91%     0.14%  ser-single-forw  [kernel.kallsyms]      [k] __do_page_fault
+   11.55%     0.23%  ser-single-forw  [kernel.kallsyms]      [k] handle_mm_fault
```

# Perf

- Can view the call trace of the program and which calls are taking the most time

```
wmoses@beast:~LULESH $ perf report
```

```
0.14          maxsd  %xmm4,%xmm7
0.05          mulsd  0x72ec(%rip),%xmm14        # 40df80 <_IO_stdin_used+0x60>
0.00          xorps  %xmm0,%xmm0
              sqrtsd %xmm7,%xmm0
1.22          divsd  %xmm0,%xmm14
0.76          mov    0x2f8(%rsp),%rbx
0.00          movsd  %xmm14,(%rbx,%r10,1)
0.07          movsd  (%r12,%rdx,8),%xmm0
0.00          movsd  (%r12,%r11,8),%xmm5
              movsd  %xmm0,0xf0(%rsp)
              subsd  %xmm5,%xmm0
```

# GDB/LLDB (Debugger)

Binary Instrumentation Tool

- Can either attach to currently running programs or execute a program from scratch
- Lets you interact with the program at any point (step through instructions, print out variables).
- Pausing execution at a point lets you see where (and why) a program is potentially hanging

```
wmoses@beast:~LULESH $ gdb --args ./lulesh.exe -s 50
```

# GDB/LLDB (Debugger)

Run the program

Pause execution

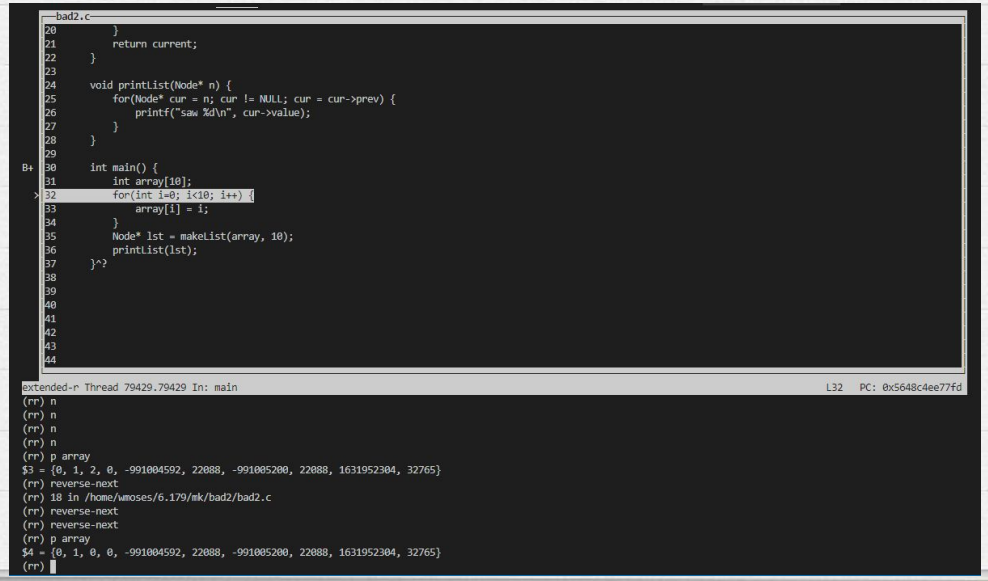
Print the stack trace

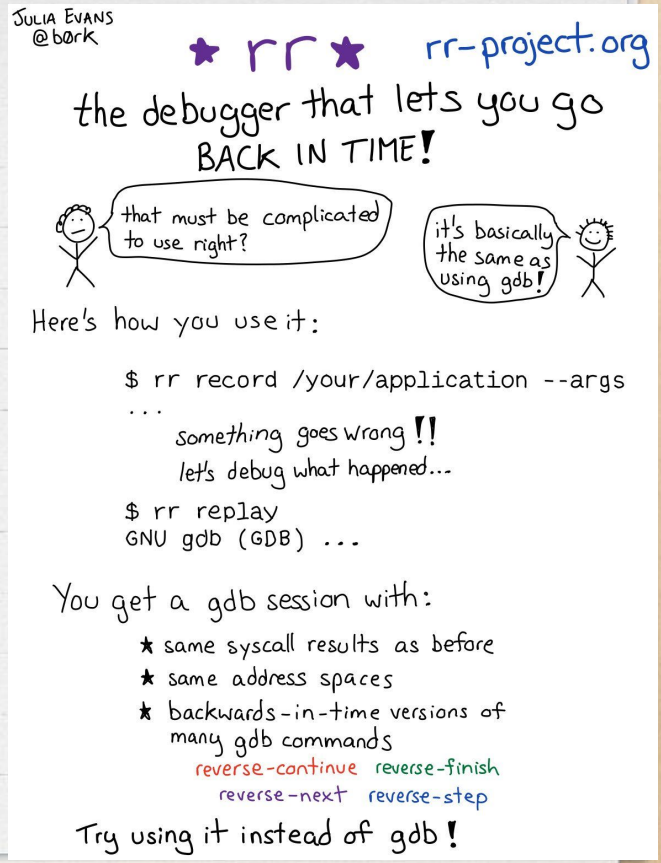
Print (and run) arbitrary code

```
(gdb) r
Starting program: /mnt/pci4/wmdata/Enzyme/enzyme/mpi/LULESH/ser-single-forward.exe -s 50
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Running problem size 50^3 per domain until completion
Num processors: 1
Total number of elements: 125000

To run other sizes, use -s <integer>.
To run a fixed number of iterations, use -i <integer>.
To run a more or less balanced region set, use -b <integer>.
To change the relative costs of regions, use -c <integer>.
To print out progress, use -p
To write an output file for VisIt, use -v
See help (-h) for more options

^C
Program received signal SIGINT, Interrupt.
0x0000000000407539 in CalcMonotonicQGradientsForElems (domain=...) at lulesh.cc:1713
1713          domain.delx_zeta(i) = vol / SQRT(ax*ax + ay*ay + az*az + ptiny) ;
(gdb) bt
#0  0x0000000000407539 in CalcMonotonicQGradientsForElems (domain=...) at lulesh.cc:1713
#1  CalcQForElems (domain=...) at lulesh.cc:1973
#2  LagrangeElements (domain=..., numElem=<optimized out>) at lulesh.cc:2483
#3  LagrangeLeapFrog (domain=...) at lulesh.cc:2663
#4  0x00000000004016da in main (argc=<optimized out>, argv=<optimized out>) at lulesh.cc:2799
(gdb) p i
$1 = 115674
(gdb)
```

# Reversible debugger (rr)

- Like gdb/lldb, but lets you execute the program backwards



# Part 2

Diagnosing the Problem

# Clang/LLVM-level Performance Diagnosis

- Now that we've diagnosed ***where*** the program is slow, we need to determine, ***why*** it is running slowly
- Already, some problems can be identified by looking at the source and fixing algorithmic/data structure problems.
- Much worse problems: your code should be fine, but an optimization isn't run?

# Optimization Remarks

Remarks (aka. optimization record) provides user-centric feedback.

Most-common use cases are determining why a program didn't vectorize

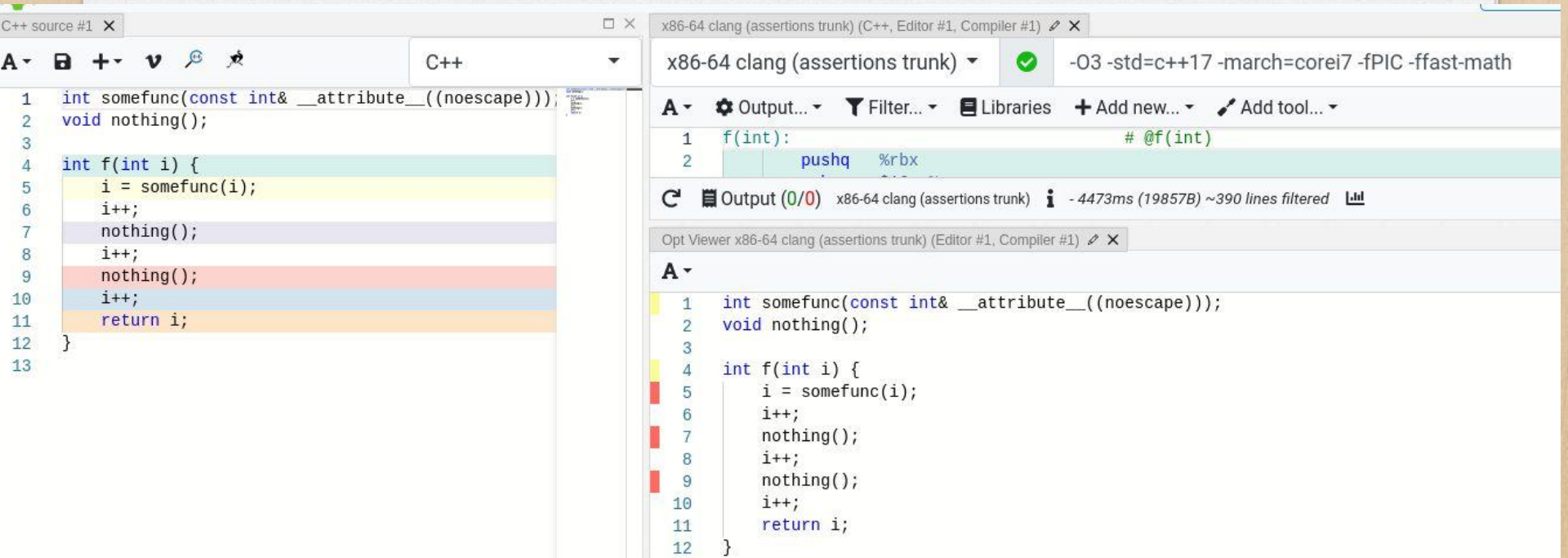
Lots of tooling (see LLVM Remarks page). Extensions available, e.g., FAROS[1]

https://clang.llvm.org/docs/UsersManual.html
https://www.llvm.org/docs/Remarks.html
[1] https://github.com/LLNL/FAROS

```
/data/benchmarks/llvm-test-suite/MultiSource/Applications/sqlite3/sqlite3.c:63421:17: remark: Stores SLP vectorized with cost -2 and with tree size 2 [-Rpass=slp-vectorizer]
      p->pLimit = pLimit;
                ^

/data/benchmarks/llvm-test-suite/MultiSource/Applications/sqlite3/sqlite3.c:63496:17: remark: Stores SLP vectorized with cost -1 and with tree size 2 [-Rpass=slp-vectorizer]
      p->pLimit = 0;
                ^

/data/benchmarks/llvm-test-suite/MultiSource/Applications/sqlite3/sqlite3.c:63503:17: remark: Stores SLP vectorized with cost -2 and with tree size 2 [-Rpass=slp-vectorizer]
      p->pLimit = pLimit;
                ^

/data/benchmarks/llvm-test-suite/MultiSource/Applications/sqlite3/sqlite3.c:9589:9: remark: Stores SLP vectorized with cost -3 and with tree size 2 [-Rpass=slp-vectorizer]
    x.Y = 2000;
        ^
```

# Compiler Explorer (Godbolt.org)

Interactively write code and see the impact of optimizations, final assembly, etc



# Inspecting LLVM IR

The compiler's internal intermediate representation (LLVM IR) can be instructive for why certain code is generated

- Consider: https://godbolt.org/z/Prxdo15KE

```
void compute(double* out, double* in, int N) {
    for (int i=0; i<N; i++) {
        out[i] = in[i] * in[i];
    }
}
```

```
9     %min.iters.check = icmp ult i32 %N, 4
10    br i1 %min.iters.check, label %for.body.preheader20, label %vecto
11
12  vector.memcheck:                                    ; preds = %for.bo
13    %scevgep = getelementptr double, double* %out, i64 %wide.trip.cou
14    %scevgep17 = getelementptr double, double* %in, i64 %wide.trip.co
15    %bound0 = icmp ugt double* %scevgep17, %out
16    %bound1 = icmp ugt double* %scevgep, %in
17    %found.conflict = and i1 %bound0, %bound1
18    br i1 %found.conflict, label %for.body.preheader20, label %vector
```

- LLVM had to insert a check whether in and out overlap

# Inspecting LLVM IR

Marking the variables as restrict (noalias in LLVM) informs the optimizer that the pointers don't overlap, getting rid of the check:

```
void compute(double* __restrict__ out,
             double* __restrict__ in, int N) {
    for (int i=0; i<N; i++) {
        out[i] = in[i] * in[i];
    }
}
```

```
for.body.preheader:                               ; preds
  %wide.trip.count = zext i32 %N to i64
  %min.iters.check = icmp ult i32 %N, 4
  br i1 %min.iters.check, label %for.body.preheader15, lab

vector.ph:                                        ; preds
  %n.vec = and i64 %wide.trip.count, 4294967292
  %0 = add nsw i64 %n.vec, -4
  %1 = lshr exact i64 %0, 2
  %2 = add nuw nsw i64 %1, 1
  %xtraiter = and i64 %2, 1
  %3 = icmp eq i64 %0, 0
  br i1 %3, label %middle.block.unr-lcssa, label %vector.p
```

# Inspecting LLVM IR

Inserting an assumption that the number of iterations is at least 4, gets rid of the minimum iteration check.

```
void compute(double* __restrict__ out,
             double* __restrict__ in, int N) {
    __builtin_assume(!(N < 4));
    for (int i=0; i<N; i++) {
        out[i] = in[i] * in[i];
    }
}
```

```
4     %cmp = icmp sgt i32 %N, 3
5     tail call void @llvm.assume(i1 %cmp)
6     %wide.trip.count = zext i32 %N to i64
7     %n.vec = and i64 %wide.trip.count, 2147483644
8     %0 = add nsw i64 %n.vec, -4
9     %1 = lshr exact i64 %0, 2
10    %2 = add nuw nsw i64 %1, 1
11    %xtraiter = and i64 %2, 1
12    %3 = icmp eq i64 %0, 0
13    br i1 %3, label %middle.block.unr-lcssa, label %for.body.pre
14
15  for.body.preheader.new:                          ; preds = %f
16    %unroll_iter = and i64 %2, 9223372036854775806
```

# Part 3

Random Thoughts

# LTO / PGO

Use link time optimization (LTO) to optimize across source files:

```
-flto          <- full/ monolithic LTO
-flto=thin     <- thin LTO
```

Use profile guided optimization (PGO):

```
-fprofile-generate
-fprofile-use
```

# -save-temps + llvm-extract

Get the "pristine" LLVM-IR from clang via

```
-save-temps
```

Use `llvm-extract` to get a subset of the functions:

```
llvm-extract --recursive --func=foo test.bc
```

llvm-extract, and other cool script are in llvm/tools

# -save-temps + run -OX multiple times

Running -O{1,2,3} multiple times help decide if optimizations are "possible".

For host only code, get an executable with

```
clang <myflags> -march=...  test.bc -o test.exe
```

Simple way to get a possible upper bound:

```
perf stat -r 11 ./test.exe
```

Also checkout the bisect scripts in llvm/utils!

# -save-temps + opt + bisect

Get the "pristine" LLVM-IR from clang via

```
-save-temps
```

Use `opt` to apply (a subset) of transformations:

```
opt -O3 test.bc
```

or

```
opt -O3 -opt-bisect-limit=50
```

Also checkout the bisect scripts in llvm/utils!

# LLVM-Core Flags

Most passes have an enable/disable flags:

```
-mllvm -enable-gvn-sink
```

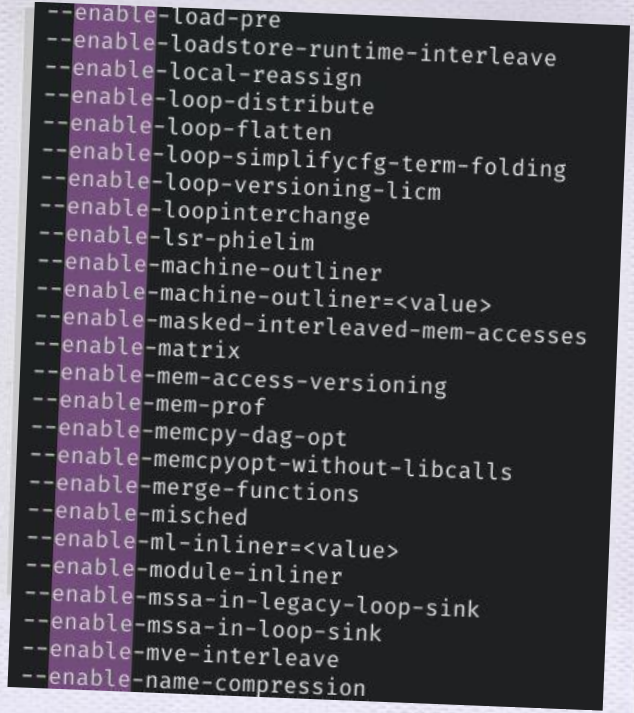
check

```
{opt, clang} -help
```

and

```
{opt, clang} -help-hidden
```

(and grep for enable/disable/gvn/...)



# Command Line Flag – Cheat Sheet

```
-O{1,2,3,fast}              <- enable optimization pipelines (-O0 is default)
-march={native,...}         <- enable CPU specific features, e.g., AVX512,
                                 and target specific choices

-ffast-math                 <- enable “unsafe” (=non standard) floating pointer
optimizations
   -fno-math-errno
   -freciprocal-math
   -fapprox-func

-fveclib={libmvec, Accelerate, MASSV, SVML, …}   <- use vectorized math functions

-save-temps                                      <- get the IR, assembly, … *before*
each step
-O0 -Xclang -disable-O0-optnone                  <- do not attach `optnone`, which is
                                                    default with -O0
```

# Command Line Flag - Cheat Sheet (cont't)

```
-ftime-passes            <- get a compile time breakdown (time per pass)

-mllvm -stats             <- get statistics, e.g., #vectorized loops,
                             from all the passes
     -save-stats          <- clang version to save the statistics to a file

-pass-remarks{-missed,-analysis}=<regex>       <- get optimization remarks
                                                  from opt
     -Rpass-remarks{-missed,analysis}=<regex>  <- clang versions
```

# C/C++ Source Annotations - Cheat Sheet

```
[__]restrict                          <- no pointer alias
__attribute__((noescape))             <- nocapture in IR, pointer is not “copied”
__attribute__((const))                <- will not access memory
__attribute__((pure))                 <- will at most read global memory
__attribute__((alloc_size(<i>)))      <- return at least <arg_i> bytes allocated memory
__attribute__((alloc_align(<i>)))     <- returned pointer is <arg_i> aligned
__attribute__((always_inline))        <- force inlining (even with -O0)
__attribute__((noinline))             <- do not inline the function
__attribute__((optnone))              <- do not optimize the function

Builtins:
      __builtin_assume(<bool>)
      __builtin_unreachable()
      __builtin_unpredicable(expr)
      __builtin_expect(expr, value)
      __builtin_expect_with_probability(expr, value, prob)
      __builtin_prefetch(addr, rw, locality)
```

https://clang.llvm.org/docs/LanguageExtensions.html https://clang.llvm.org/docs/AttributeReference.html

# Research for Performance GAP estimation

Embed "assumed knowledge" into a program, compile it, test it.

Determine knowledge that is probably correct and definitively helpful to improve performance.

Got up to 20% improvement for proxy apps with 3 minimal code changes!


PETOSPA (ISC'19): https://github.com/jdoerfert/PETOSPA
HTO (LLVMDev '19): https://www.youtube.com/watch?v=elmio6AoyKO
ORAQL (LLVMDev '21): https://www.youtube.com/watch?v=7UVB5AFJM1w

# OpenMP Offload

Additional Notes

# Optimization Remarks

## Example: OpenMP runtime call deduplication

```
double *A = malloc(size * omp_get_thread_limit());
double *B = malloc(size * omp_get_thread_limit());

#pragma omp parallel
do_work(A, B);
```

OpenMP runtime calls with same return values can be merged to a single call

$ clang -g -O2 deduplicate.c -fopenmp -Rpass=openmp-opt

```
deduplicate.c:12:29: remark: OpenMP runtime call omp_get_thread_limit moved to deduplicate.c:11:29: [-Rpass=openmp-opt]
   double *B = malloc(size*omp_get_thread_limit());
deduplicate.c:11:29: remark: OpenMP runtime call omp_get_thread_limit deduplicated [-Rpass=openmp-opt]
   double *A = malloc(size*omp_get_thread_limit());
```

# Optimization Remarks

## Example: OpenMP Target Scheduling

```
clang12 -Rpass=openmp-opt ...
```

```
void bar(void) {
  #pragma omp parallel
  { }
}
void foo(void) {
 #pragma omp target teams
 {
  #pragma omp parallel
  {}
  bar();
  #pragma omp parallel
  {}
 }
}
```

```
remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct
without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call
edges assumed by ptxas.
remark: Parallel region is not known to be called from a unique single target region, maybe the enclosing function has external linkage?; will not
attempt to rewrite the state machine use.
remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct
without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call
edges assumed by ptxas.
remark: Specialize parallel region that is only reached from a single target region to avoid spurious call edges and excessive register usage in
other target regions. (parallel region ID: __omp_outlined__1_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Target region containing the parallel region that is specialized. (parallel region ID: __omp_outlined__1_wrapper, kernel ID:
__omp_offloading_35_a1e179_foo_l7)
remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct
without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call
edges assumed by ptxas.
remark: Specialize parallel region that is only reached from a single target region to avoid spurious call edges and excessive register usage in
other target regions. (parallel region ID: __omp_outlined__3_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Target region containing the parallel region that is specialized. (parallel region ID: __omp_outlined__3_wrapper, kernel ID:
__omp_offloading_35_a1e179_foo_l7)
remark: OpenMP GPU kernel __omp_offloading_35_a1e179_foo_l7
```

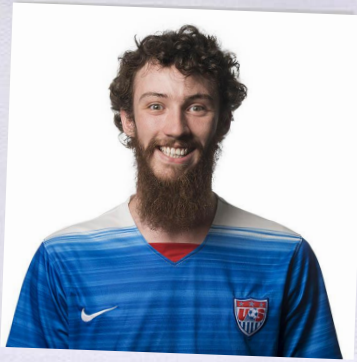
**Explained online!**

# OpenMP offload Recommendations

- Use a recent (e.g., nightly) compiler version.
- Enable compilation remarks https://openmp.llvm.org/remarks/OptimizationRemarks.html
- Use `LIBOMPTARGET_INFO(=16)` to learn about the GPU execution
  https://openmp.llvm.org/design/Runtimes.html#libomptarget-info
- Use `LIBOMPTARGET_PROFILE` for built in profiling support.
- Use `LIBOMPTARGET_DEBUG` (and `-fopenmp-target-debug`) for runtime assertions and other opt-in debug features https://openmp.llvm.org/design/Runtimes.html#debugging
- Consider assumptions for better performance:
  `LIBOMPTARGET_MAP_FORCE_ATOMIC=false` and `-fopenmp-assume-no-thread-state`
- Use the new driver `-fopenmp-new-driver` and device-side LTO `-foffload-lto`


Ask Us Anything

# Johannes Doerfert (he/him)




work with LLVM since ~2012

initial polyhedral optimization

nowadays
- OpenMP (runtime, openmp-opt, ...)
- interprocedural Optimization (Attributor)
- LLVM-IR

involved in various working groups:
- Alias Analysis, ML, OpenMP, Flang, ...