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

wmoses@beast:-LULESH $ perf report

<table>
<thead>
<tr>
<th>Children</th>
<th>Self</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>92.55%</td>
<td>78.10%</td>
<td>ser-single-forw</td>
<td>ser-single-forward.exe [.] LagrangeLeapFrog</td>
</tr>
<tr>
<td>-</td>
<td>42.52%</td>
<td></td>
<td>LagrangeLeapFrog</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>13.52%</td>
<td></td>
<td>page_fault</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>11.93%</td>
<td></td>
<td>do_page_fault</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>18.73%</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>22.99%</td>
<td>0.00%</td>
<td>ser-single-forw</td>
<td>[unknown] [.] 00000000000000000000000000000000</td>
</tr>
<tr>
<td>+</td>
<td>13.58%</td>
<td>1.59%</td>
<td>ser-single-forw</td>
<td>[kernel.kallsyms] [k] page_fault</td>
</tr>
<tr>
<td>+</td>
<td>11.98%</td>
<td>0.04%</td>
<td>ser-single-forw</td>
<td>[kernel.kallsyms] [k] do_page_fault</td>
</tr>
<tr>
<td>+</td>
<td>11.91%</td>
<td>0.14%</td>
<td>ser-single-forw</td>
<td>[kernel.kallsyms] [k] __do_page_fault</td>
</tr>
<tr>
<td>+</td>
<td>11.55%</td>
<td>0.23%</td>
<td>ser-single-forw</td>
<td>[kernel.kallsyms] [k] handle_mm_fault</td>
</tr>
</tbody>
</table>
- Can view the call trace of the program and which calls are taking the most time

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.14</td>
<td><code>maxsd %xmm4, %xmm7</code></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td><code>mulsd 0x72ec(%rip),%xmm14</code></td>
<td># 40df80 &lt;__I0_stdin_used+0x60&gt;</td>
</tr>
<tr>
<td>0.00</td>
<td><code>xorps %xmm0, %xmm0</code></td>
<td></td>
</tr>
<tr>
<td>1.22</td>
<td><code>sqrtsd %xmm7, %xmm0</code></td>
<td></td>
</tr>
<tr>
<td>0.76</td>
<td><code>divsd %xmm0, %xmm14</code></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td><code>mov 0x2f8(%rsp), %rbx</code></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td><code>movsd %xmm14, (%rbx, %r10, 1)</code></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td><code>movsd (%r12, %rdx, 8), %xmm0</code></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td><code>movsd (%r12, %r11, 8), %xmm5</code></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td><code>movsd %xmm0, 0x0f0(%rsp)</code></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td><code>subsd %xmm5, %xmm0</code></td>
<td></td>
</tr>
</tbody>
</table>
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/2c4/wndata/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

AC
Program received signal SIGINT, Interrupt.
0x00000000000487539 in CalcMonotonicQGradientsForElems (domain=...) at lulesh.cc:1713
1713  domain.delx_zeta[] = vol / SQRT(ax*ax + ay*ay + az*az + ptiny) ;
(gdb) bt
#0 0x00000000000487539 in CalcMonotonicQGradientsForElems (domain=...) at lulesh.cc:1713
#1 CalcQForElems (domain=...) at lulesh.cc:1973
#2 LagrangeElements (domain=..., numElems=optimized out) at lulesh.cc:2483
#3 LagrangeLeapFrog (domain=...) at lulesh.cc:2663
#4 0x000000000004016da in main (argc=optimized out>, argv=optimized out) at lulesh.cc:2799
(gdb) p i
i = 119674
(gdb)
```
Reversible debugger (rr)

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

```c
return current;
}

void printlist(node* node) {
    for(node* cur = node; cur != NULL; cur = cur->prev) {
        printf("%s from\n", cur->value);
    }
}

int main() {
    list* head = malloc(sizeof(list)), tail = NULL;
    node* list = malloc(sizeof(node), 8);    
    printlist(list);
}
```

Here's how you use it:

```bash
$ rr record /your/application --args ...

... something goes wrong!!
let's debug what happened...

$ rr replay
GNU gdb (GDB) ...
```

You get a gdb session with:

- same syscall results as before
- same address spaces
- backwards-in-time versions of many gdb commands
  - reverse-continue reverse-finish
  - reverse-next reverse-step

Try using it instead of gdb!
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., FAROS¹

https://www.llvm.org/docs/Remarks.html
¹https://github.com/LLNL/FAROS
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](https://godbolt.org/z/Prxdo15KE)

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

- 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:

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

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

```c
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];
  }
}
```
Part 3
Random Thoughts
LTO / PGO

Use link time optimization (LTO) to optimize across source files:
- \texttt{-flto} <- full/monolithic LTO
- \texttt{-flto=thin} <- thin LTO

Use profile guided optimization (PGO):
- \texttt{-fprofile-generate}
- \texttt{-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:

-mlllvm -enable-gvn-sink

cHECK

{opt, clang} -help

and

{opt, clang} -help-hidden

(and grep for enable/disable/gvn/...
Command Line Flag – Cheat Sheet

-0{1,2,3,fast}  <- enable optimization pipelines (-00 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 each step  <- get the IR, assembly, ... *before*
-00 -Xclang -disable-O0-optnone  <- do not attach `optnone`, which is default with -00
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

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>restrict</code></td>
<td>no pointer alias</td>
</tr>
<tr>
<td><code>__attribute__((noescape))</code></td>
<td>nocapture in IR, pointer is not “copied”</td>
</tr>
<tr>
<td><code>__attribute__((const))</code></td>
<td>will not access memory</td>
</tr>
<tr>
<td><code>__attribute__((pure))</code></td>
<td>will at most read global memory</td>
</tr>
<tr>
<td><code>__attribute__((alloc_size(&lt;i&gt;)))</code></td>
<td>return at least <code>&lt;arg_i&gt;</code> bytes allocated memory</td>
</tr>
<tr>
<td><code>__attribute__((alloc_align(&lt;i&gt;)))</code></td>
<td>returned pointer is <code>&lt;arg_i&gt;</code> aligned</td>
</tr>
<tr>
<td><code>__attribute__((always_inline))</code></td>
<td>force inlining (even with -O0)</td>
</tr>
<tr>
<td><code>__attribute__((noinline))</code></td>
<td>do not inline the function</td>
</tr>
<tr>
<td><code>__attribute__((optnone))</code></td>
<td>do not optimize the function</td>
</tr>
</tbody>
</table>

**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/LanguageExtensions.html)  [https://clang.llvm.org/docs/AttributeReference.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=elmio6AoyK0
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);

$ 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 surrounding 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](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](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](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, ...