Benchmarking compiler optimizations on OpenMP performance
The First Workshop on LLVM Compiler and Tools for HPC – ISC2020

Giorgis Georgakoudis, Ignacio Laguna, Tom Scogland, Lawrence Livermore National Laboratory
Johannes Doerfert, Argonne National Laboratory

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344, Lawrence Livermore National Security, LLC
Problem: Clang/LLVM optimizes differently OpenMP programs versus their serial elision

Serial elision

OpenMP

Compiler Optimizes

Sequential code

#pragma omp parallel
do_computation

Sequential code

Compiler Optimization?

Serial elision

OpenMP

Sequential code

#pragma omp parallel
do_computation

void omp_computed(
    type_1 *shared_var_1,…) {
    do_computation
}

__kmcp_fork_call( omp_outlined, &shared_var_1,…) )

Sequential code
Two examples of how compiler optimization using OpenMP impacts performance

- Executing OpenMP single-threaded
- hotspot is 25.48% faster
- srad is 135.23% slower

Why?
FAROS: A framework for analyzing OpenMP compiler optimization

**Functionality**
- Pinpoint compiler optimization differences
- Measure performance impact of OpenMP compilation

**Design goals**
- Extensible
- Informative
- Reproducible
The workflow of FAROS

- User provides a YAML configuration of benchmarks

**Output**

- Profiling report for different build optimizations
- Compiler optimizations reports based on LLVM remarks
The interface of the harness script


Harness for benchmarking a set of programs and compilation options

optional arguments:
  -h, --help                    show this help message and exit
  -i INPUT, --input INPUT      configuration YAML input file for programs
  -p PROGRAMS [PROGRAMS ...], --programs PROGRAMS [PROGRAMS ...] selected programs from the config
  -f, --fetch                  fetch program repos (without building)
  -b, --build                  build programs (will fetch too)
  -g, --generate               generate compilation reports
  -r RUN, --run RUN            run <repetitions>
  -d, --dry-run                enable dry run
The specification of the YAML configuration

LULESH:

`fetch: 'git clone -q https://github.com/LLNL/LULESH.git'`

`build_dir: 'LULESH'`

`build: {
seq: 'make CXX=clang++ CXXFLAGS="-g -O3 -march=native -I. -Wall -DUSE_MPI=0 -fsave-optimization-record -save-stats"',
omp: 'make CXX=clang++
    CXXFLAGS="-g -fopenmp -O3 -march=native -I. -Wall -DUSE_MPI=0 -fsave-optimization-record -save-stats"'
}

`copy: [ 'lulesh2.0' ]`

`run: 'env OMP_NUM_THREADS=1 OMP_PROC_BIND=true ./lulesh2.0'`

`input: '-i 500'`

`measure: 'Grind time.* (\d+\.\d+) .*overall'`

`clean: 'git clean -fx'`
FAROS generates compilation reports using LLVM remarks and opt-viewer tools

- LLVM per-source optimization remarks (Passed, Missed, Analysis)
- Combine and generate HTML reports
FAROS runs programs and collects profiling results in YAML

Run sequential and single-threaded OpenMP
Measure end-to-end execution time or regex of measure
FAROS integrates 39 OpenMP programs so far

<table>
<thead>
<tr>
<th>HPC proxy/mini/large</th>
<th>AMG, CoMD, CoSP2, Kripke, LULESH, Quicksilver, RSBench, SimpleMOC, XSBench, miniAMR, miniAero, miniFE, hpcg, gromacs-2019.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS</td>
<td>BT, CG, EP, FT, IS, LU, MG, SP</td>
</tr>
<tr>
<td>Rodinia</td>
<td>b+tree, backprop, bfs, cfd, heartwall, hotspot, hotspot3D, kmeans, lavaMD, leukocyte, lud, nn, nw, particlefilter, pathfinder, srad, streamcluster</td>
</tr>
</tbody>
</table>
Most programs slow down and few speedup from OpenMP compilation

- Programs with slowdown > 1.1× or slowdown < 0.95×
- 12/16 slow down, 4/16 speed up
OpenMP compilation of *srad* misses vectorization

**Diff** of passed optimizations for sequential (-) versus OpenMP(+)

<table>
<thead>
<tr>
<th>Line</th>
<th>Optimization</th>
<th>Source</th>
<th>Inline Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>#ifdef OPEN</td>
<td><code>omp_set_num_threads(nthreads);</code></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td></td>
<td><code>#pragma omp parallel for shared(J, dN, dS, d</code></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td><code>omp_set_num_threads(nthreads);</code></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td><code>#pragma omp parallel for shared(J, dN, dS, d</code></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td><code>for (int i = 0 ; i &lt; rows ; i++) {</code></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td></td>
<td><code>for (int j = 0; j &lt; cols; j++) {</code></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>-loop-vectorize</td>
<td><code>vectorized loop (vectorization width: 8, interleaved ...</code></td>
<td></td>
</tr>
</tbody>
</table>

*Details of optimizations and contexts are marked in the table.*
Analysis pinpoints the problem to fix for srad

Diff of analysis output for sequential (-) versus OpenMP (+)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>for (int j = 0; j &lt; cols; j++) {</td>
</tr>
<tr>
<td></td>
<td>+loop-vectorize</td>
</tr>
<tr>
<td></td>
<td>loop not vectorized: cannot identify array bounds</td>
</tr>
<tr>
<td>131</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>.omp_outl...</td>
</tr>
</tbody>
</table>

- Variable cols is shared, thus a pointer
  - Loop bound analysis fails
  - Pointer analysis cannot determine alias-free

- #pragma omp simd or
- IPO value propagation using experimental Attributor

---

OpenMP compilation of hotspot enables complete loop unrolling

**Diff** of passed optimizations sequential (-) versus OpenMP (+)

- OpenMP canonical loop constraint helps loop analysis
- `#pragma omp simd` dependence-free pointers \(\Rightarrow\) avoids runtime checks
Refactoring brings sequential \textit{hotspot} on par with OpenMP

- Refactor code to simplify loop analysis

  
  \begin{verbatim}
  for(cc = 0; cc < BLOCK_SIZE_C; cc++)
    c = c_start + cc;
  ...
  \end{verbatim}

- Declare pointers as \texttt{restrict}
Conclusion

Summary

- OpenMP compilation has complex effects on compiler optimization
- FAROS helps to benchmark and analyze

Future work

- Improve compilation reporting (LLVM IR, OpenMP remarks)
- Provide recommendations to maximize optimization
- Integrate more profiling information (TAU, HPCToolkit)
Thank you! Questions?

Disclaimer
This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.