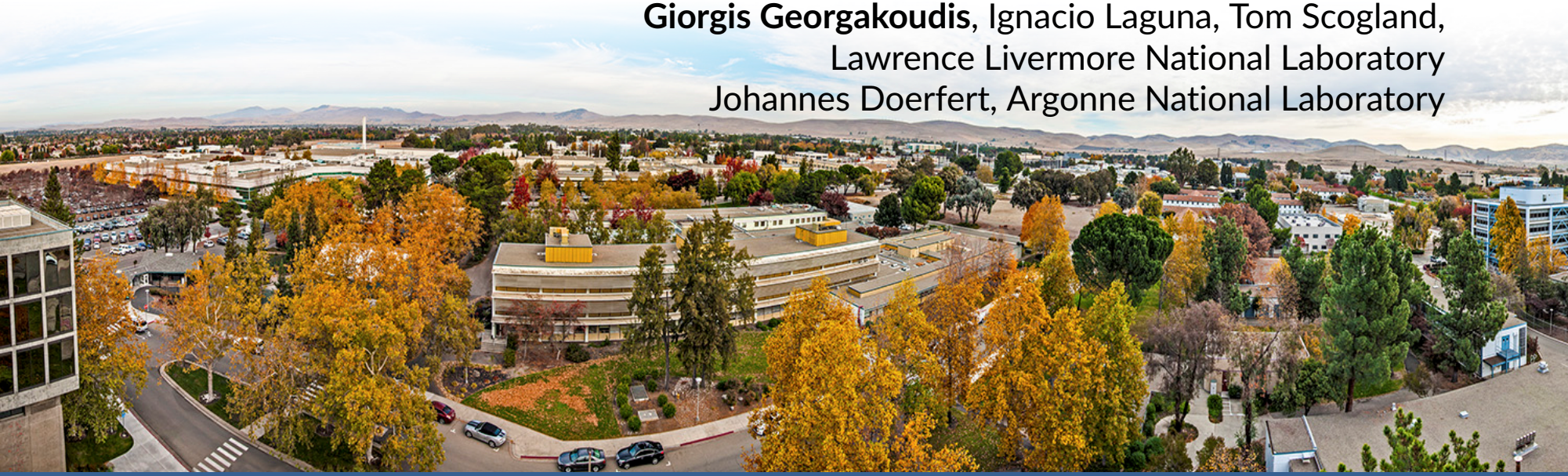
FAROS: A Framework To Analyze OpenMP Compilation Through Benchmarking and Compiler Optimization Analysis

16th International Workshop on OpenMP – IWOMP 2020

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



LLNL-PRES-811934

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

Lawrence Livermore National Laboratory

Problem: Clang/LLVM optimizes differently OpenMP programs versus their serial elision

```
sequential code

#pragma omp parallel
  do_computation

sequential code
```

Serial elision

Compiler Optimizes

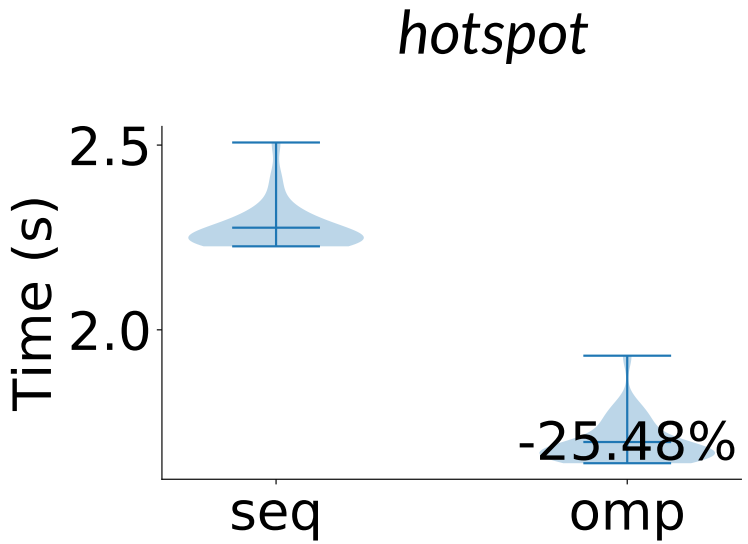
```
sequential code

#pragma omp parallel
  do_computation

sequential code
```

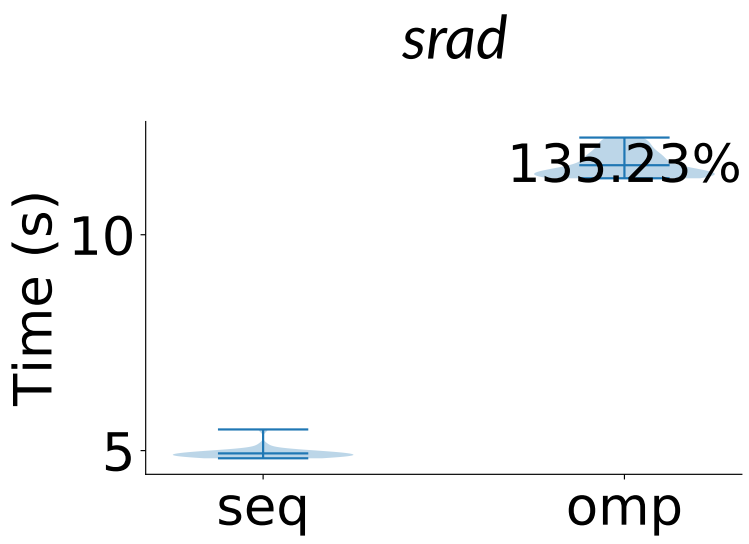
OpenMP

Compiler Optimization?

```
sequential code

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

__kmpc_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

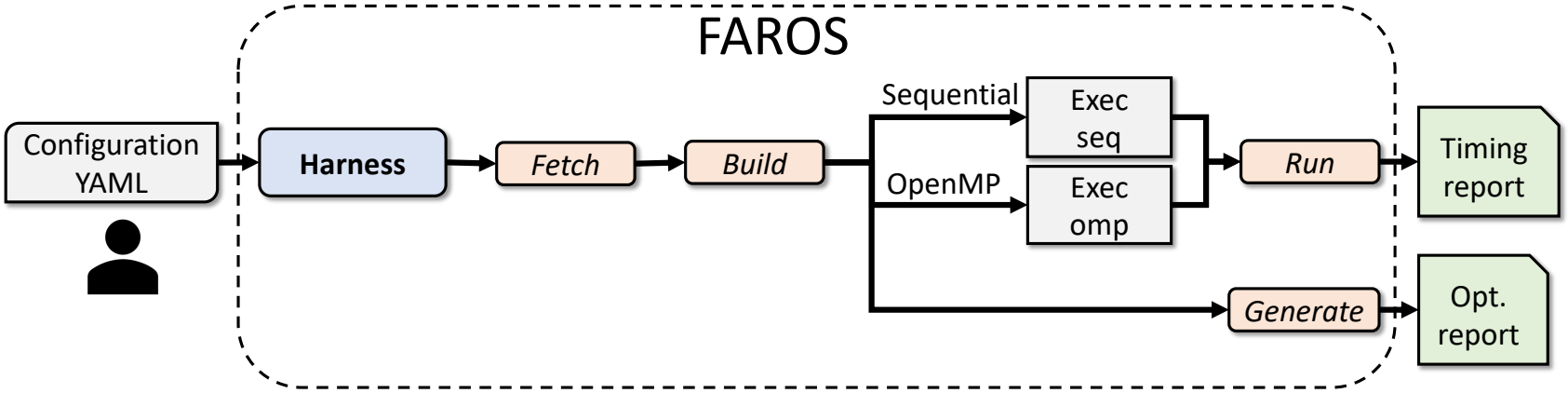
Functionality

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

Design goals

- Extensible
- Informative
- Reproducible

The workflow of FAROS



Input

- User provides a YAML configuration of programs

Output

- Timing report for different builds
- Compiler optimizations reports

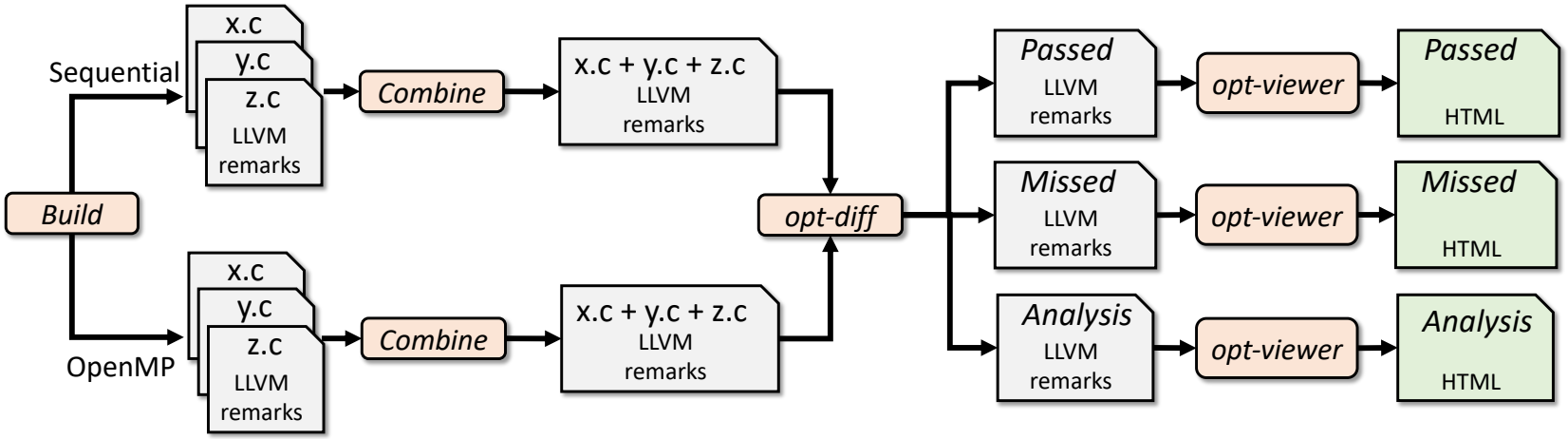
The interface of the harness script

```
usage: harness.py [-h] -i INPUT [-f] [-b] [-r RUN] [-g] -p PROGRAMS
                   [PROGRAMS ...] [-s] [-d]

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

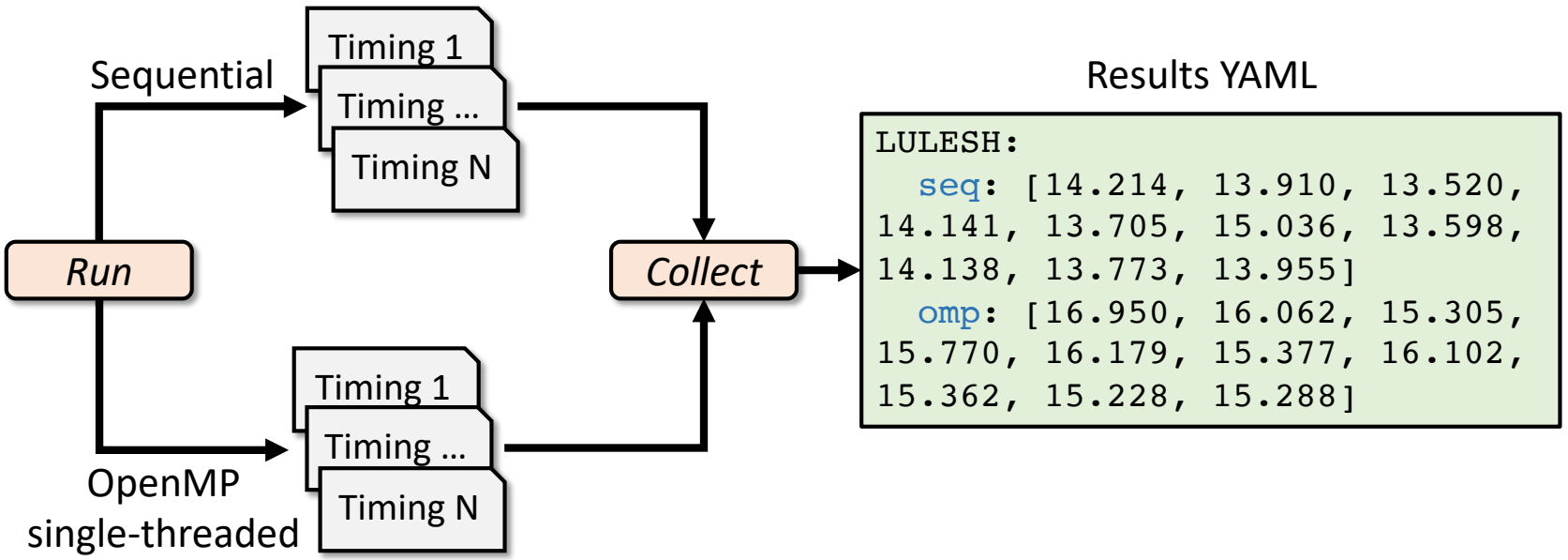
```
LULESH:
  fetch: 'git clone -q https://github.com/LLNL/LULESH.git'
  build_dir: 'LULESH'
  build: {
    seq: 'make CXX=clang++ CXXFLAGS="-g -O3 -march=native
        -I. -Wall -DUSE_MPI=0 -fsave-optimization-record
        -save-stats"',
    omp: 'make CXX=clang++
        CXXFLAGS="-g -fopenmp -O3 -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 timing results in YAML

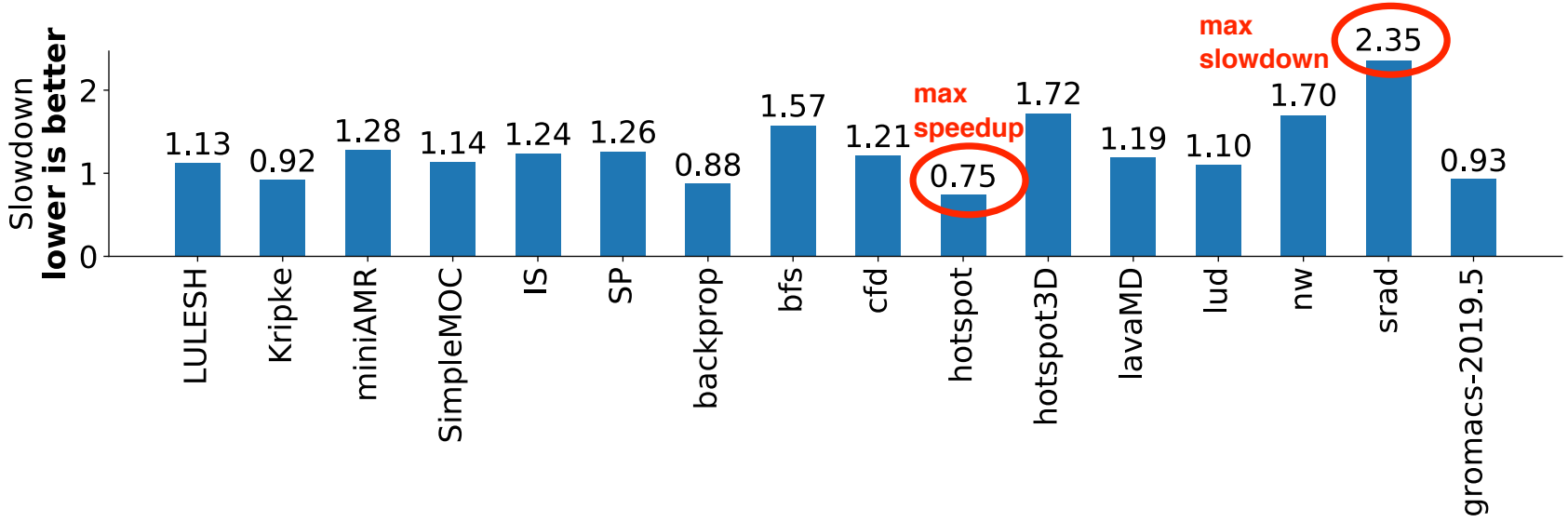


- Run sequential and single-threaded OpenMP
- Measure end-to-end execution time **or** regex of `measure`

FAROS integrates 39 OpenMP programs so far

HPC proxy/mini/large	AMG, CoMD, CoSP2, Kripke, LULESH, Quicksilver, RSBench, SimpleMOC, XSBench, miniAMR, miniAero, miniFE, hpcg, gromacs-2019.5
NAS	BT, CG, EP, FT, IS, LU, MG, SP
Rodinia	b+tree, backprop, bfs, cfd, heartwall, hotspot, hotspot3D, kmeans, lavaMD, leukocyte, lud, nn, nw, particlefilter, pathfinder, srad, streamcluster

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(+)

Line	Optimization	Source	Inline Contex
124			
125		`#ifdef OPEN`	
126		`omp_set_num_threads(nthreads);`	
127		`#pragma omp parallel for shared(J, dN, dS, d`	
	+licm	sinking getelementptr	main
	+inline	.omp_outlined.__debug__ inlined into .omp_ou…	.omp_outl…
	+licm	hoisting getelementptr	main
128		`#endif`	
129		`for (int i = 0 ; i < rows ; i++) {`	
	-licm	hoisting icmp	main
	-licm	hoisting zext	main
	-licm	sinking zext	main
130		`for (int j = 0; j < cols; j++) {`	
	-loop-vectorize	vectorized loop (vectorization width: 8, interleaved …	main
	-licm	sinking zext	main
	-licm	hoisting zext	main

Analysis pinpoints the problem to fix for *srad*

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

130		`for (int j = 0; j < cols; j++) {`	
	+loop-vectorize	loop not vectorized: cannot identify array bounds	.omp_outl…

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

SOLVED

- `#pragma omp simd` **or**
- IPO value propagation using experimental Attributor[1]

[1] Doerfert, Johannes, and Hal Finkel. "Compiler optimizations for OpenMP." In International Workshop on OpenMP, pp. 113-127. Springer, Cham, 2018.

OpenMP compilation of *hotspot* enables complete loop unrolling

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

```
137                 #pragma omp simd
      +loop-unroll  completely unrolled loop with 2 iterations                    .omp_ou…
      +loop-vectorize vectorized loop (vectorization width: 8, interleaved count: 1)  .omp_ou…
138                 for ( c = c_start; c < c_start + BLOCK_SIZE_C; ++c
                                                                                  16
            -licm   hoisting icmp                                                 single_it…
            -licm                          sinking trunc                          single_it…
      -loop-vectorize vectorized loop (vectorization width: 8, interleaved co…    single_it…
            +licm                                                          …      .omp_ou…
            -licm   hoisting shufflevector                                        single_it…
            -licm   hoisting insertelement                                        single_it…
            -licm   hoisting add                                                  single_it…
            -licm   hoisting and                                                  single_it…
139                 /* Update Temperatures */
140                     result[r*col+c] =temp[r*col+c]+
            +licm                          hoisting load                          .omp_ou…
            +licm       hoisting load                                             .omp_ou…
141                         ( Cap_1 * (power[r*col+c] +
            +licm                          hoisting load                          .omp_ou…
142                         (temp[(r+1)*col+c] + temp[(r-1)*col+c] - 2
143                         (temp[r*col+c+1] + temp[r*col+c-1] - 2.f*te
144                         (amb_temp - temp[r*col+c]) * Rz_1));
145                 }
```

- OpenMP canonical loop constraint helps loop analysis
- `simd` dependence-free pointers ⇒ avoids runtime checks

Refactoring brings sequential *hotspot* on par with OpenMP

SOLVED

- Refactor code to simplify loop analysis

```
for(cc = 0; cc < BLOCK_SIZE_C; cc++)
  c = c_start + cc;
  ...
```

- Declare pointers as `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 enable optimization
- Integrate 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.

Lawrence Livermore National Laboratory