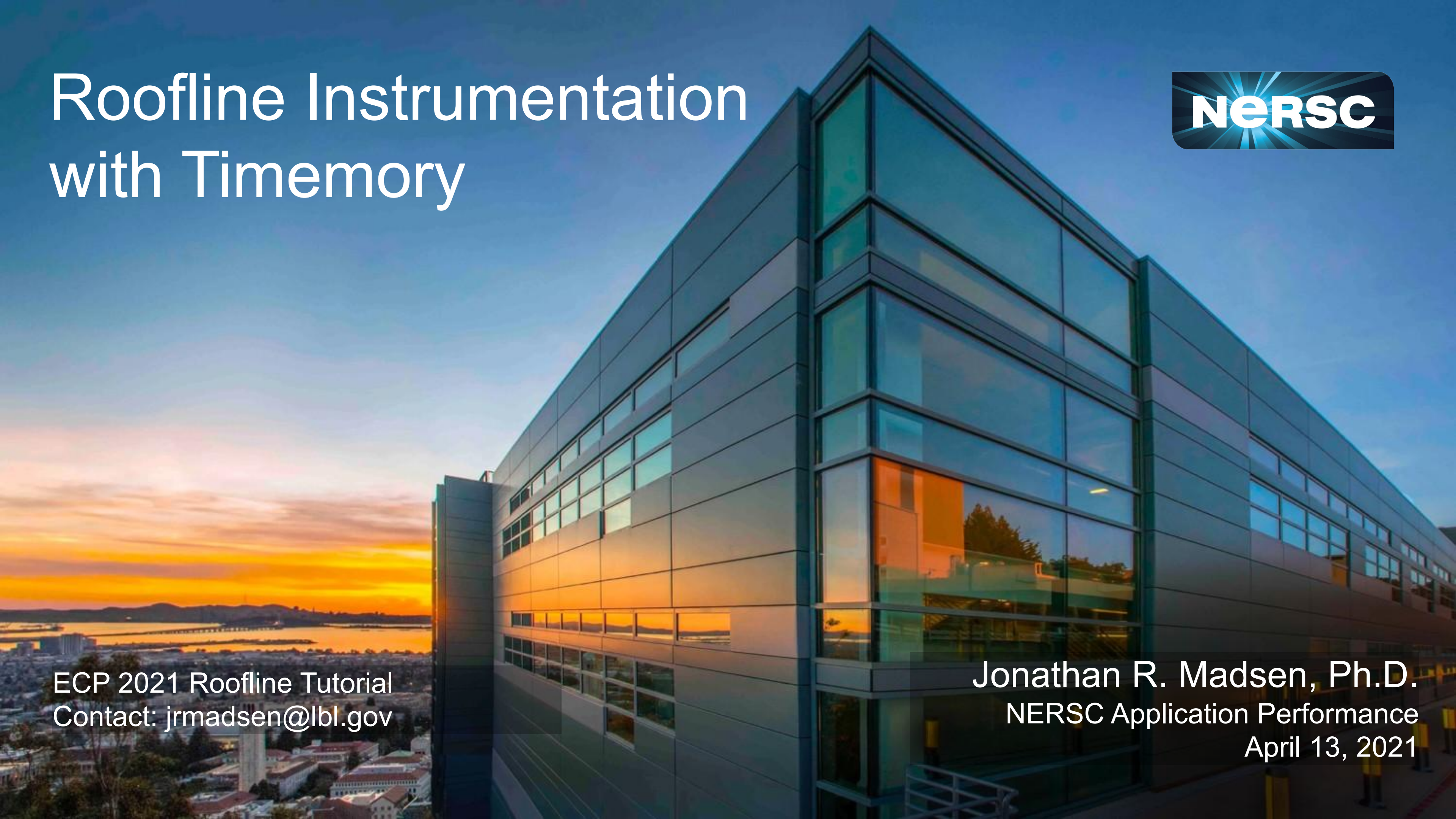

Roofline Instrumentation with Timemory

ECP 2021 Roofline Tutorial
Contact: jrmadsen@lbl.gov

Jonathan R. Madsen, Ph.D.
NERSC Application Performance
April 13, 2021

Scenario #1

- Software bug is found (e.g. wrong answer, seg-fault, etc.)
- Developer spends hours to days tracking it down and fixes it
- What *usually* happens now?
- Developer opens a merge-request with the bug fix and…
- … adds new test which verifies the bug has been fixed!
- Why?
- The test is not for the MR *specifically* → it is added to help ensure subsequent commits do not re-introduce the bug!

Scenario #2

- Developer(s) attend performance optimization hackathon
- Developer(s) spend days profiling code and improving the roofline
- What *usually* happens now?
- Developer opens a merge-request with the optimizations and…
- … assume subsequent commits will retain performance via comment:

```
// DO NOT MODIFY THIS BLOCK OF CODE!
// This [ugly] code is highly optimized for <insert architecture>
// Improvement was <blah> vs. <blah>
// <possibly insert detailed explanation>
//
// <name and date>
```

Scenario #2 (cont.)

- Why don't developers write performance tests with same regularity?
- Three part theory:

1. Easy-to-implement performance metrics (e.g. timers) are unreliable → too many variables with respect to hardware
2. Advanced metrics (e.g. HW counters) are far more difficult to collect and tailor the API configurations per-test (e.g. perf, CUpti, etc.)
3. Tools which have the capability to simplify #2 are rarely designed for in-situ programmatic data access

Pursuing a Solution to Scenario #2

- Performance analysis tools like Advisor, Nsight-Systems, Nsight-Compute, etc. were extremely useful to original optimization but provide no useful means for maintaining that performance via automation
- We need a toolkit which:
  - Provides advanced metrics as easy to use as timers
  - Fully configurable and customizable to testing needs
  - Recognizes it is auxiliary code, not fundamental code

Fundamental vs. Auxiliary Code

- Fundamental: code implementing the purpose of the application
- Auxiliary: important or useful code additions which are not critical to accomplishing the purpose of application
  - i.e. safety checks, log messages, performance analysis, check-pointing, etc.
- Developers usually have limits w.r.t. their willingness to maintain auxiliary code
  - Auxiliary performance-analysis code tends to strain these limits

Issues with Auxiliary Performance Analysis Code

- Difficult to resolve build or linking failures
  - Global install of 3rd-party dependency built with incompatible X
  - Tool uses X which interferes with my use of X
- Plethora of macros to handle minor API variations

```
#define    PROFILE_VAR(fname, vname)
#define    PROFILE_VAR_NS(fname, vname)
#define    PROFILE_VAR_START(vname)
#define    PROFILE_VAR_STOP(vname)
```

```
#define    PROFILE_VAR_NS_CUPTI(fname, vname)
#define    PROFILE_VAR_START_CUPTI(vname)
#define    PROFILE_VAR_STOP_CUPTI(vname)
#define    PROFILE_VAR_STOP_CUPTI_ID(vname, uintID)
```

```
#define    COMM_PROFILE(cft, size, pid, tag)
#define    COMM_PROFILE_BARRIER(message, bc)
#define    COMM_PROFILE_ALLREDUCE(cft, size, bc)
#define    COMM_PROFILE_REDUCE(cft, size, pid)
#define    COMM_PROFILE_WAIT(cft, reqs, status, bc)
#define    COMM_PROFILE_WAITSOME(cft, reqs, completed, status, bc)
```

Timemory Toolkit

- Timemory provides a way to manage auxiliary code
  - Trivially extendable and composable
  - Handles any valid C or C++ data-type
  - Provides modular and reusable performance components which can be customized for any given scenario, e.g. the “roofline test”
- github.com/NERSC/timemory
- timemory.readthedocs.io
  - Roofline components
  - Getting Started with Roofline

Timemory Design

- Strong focus on reusability and data locality

```
namespace tim::component {
struct inst_per_cycle : public base<inst_per_cycle> {
   tim::component::papi_tuple<PAPI_TOT_INS, PAPI_TOT_CYC> m_hw;

   void   start() { m_hw.start(); }
   void   stop() { m_hw.stop(); }
   double get() const { return m_hw.get()[0] / m_hw.get()[1]; }
};
} // namespace tim::component
```

Timemory Design (cont.)

- Too much to cover in this presentation
- The next two Mondays (April 19th and April 26th), there is a timemory ECP tutorial
  - www.exascaleproject.org/event/timemory/
  - First day will cover pre-built tools
    - Built-in roofline capabilities will be covered here
  - Second day will cover the toolkit design
    - Customizing roofline capabilities will be covered here
  - Tutorial content is at github.com/NERSC/timemory-tutorials
    - Following conclusion of tutorial, will be tagged as ecp2021

Roofline Instrumentation with Timemory Benefits

- Ability to collect roofline at scale
- Single tool for CPU and GPU roofline generation
  - Supports instruction roofline on the GPU
- No complicated script commands
- Empirical roofline peaks on GPU
  - Nsight uses theoretical peaks
- Multiple Instrumentation Options
  - Dynamic instrumentation, Source instrumentation
- Customize data output format

Roofline Capabilities

- HW counter components for PAPI and CUpti
  - Essentially as easy to use as timer components
  - More backends are planned (LIKWID, perf, etc.)
- Roofline components implement something quite similar to previous slide
  - CPU roofline components combine a PAPI and wall-clock timer components
  - GPU roofline components combine two CUpti APIs for HW counters and kernel runtimes
- Built-in empirical roofline toolkit (ERT)
  - Extensively customizable

Empirical Roofline Toolkit

- Configuration Customization
  - Minimum working size
  - Max data size
  - Number of threads
  - Number of streams (GPU)
  - Grid size (GPU)
  - Block size (GPU)
  - Data alignment
- Executor Customization
  - Labels
  - Target device
  - **Store function**
  - **Operation function**
  - Explicit vectorization unrolling
  - Bytes-per-element
  - Memory-accesses-per-element

ERT Executor Customization

```
// store function executed in peak calc
auto store = []
      (Tp& a, const Tp& b)
      { a = b; };

// operation function executed in peak calc
auto fma = []
      (Tp& a, const Tp& b, const Tp& c)
      { a = a * b + c; };

_counter.bytes_per_element = sizeof(Tp);
_counter.memory_accesses_per_element = 2;

return ops_main<Flops/2, Flops, ...>(
            _counter, fma, store);
```

- “store” and “fma” are the functions used to calculate the “roof” of the roofline
- Tp is the templated data type
  - E.g. Tp == float
- Flops is the vectorization width, e.g. 512 for AVX-512
- Counter holds the ERT results
- For testing purposes, you can customize these functions to target a simplified or idealized version of *<u>your algorithm</u>*

Roofline Instrumentation

- “timemory-run” provides dynamic instrumentation
- Dynamic function wrapping allows you build “plug-in” libraries which you can activate and deactivate

```
using roofline_t = tim::component_tuple<cpu_roofline, gpu_roofline>;
using roofline_bundle_t = tim::component::gotcha<2, roofline_t>;

double myfunc(const std::vector<double>&);

TIMEMORY_C_GOTCHA(roofline_bundle_t, 0, MPI_Allreduce)
TIMEMORY_CXX_GOTCHA(roofline_bundle_t, 1, myfunc)
```

Roofline Instrumentation (C/C++/Fortran)

- Sample using library API
- When HW counter capabilities are enabled, CPU roofline will always be collected

```
void spam()
{
    timemory_set_default("wall_clock, cpu_roofline");
    timemory_push_region("spam");

    foo();
    bar();

    timemory_pop_region("spam");
}
```

Roofline Instrumentation (C/C++/Fortran)

- Sample using library API
- When BENCHMARK is defined:
  - CPU roofline measurements in "benchmark" region
  - Wall-clock measurements in both "main" and "benchmark" regions
- Data access available but not demonstrated

```
void spam()
{
    timemory_set_default("wall_clock");
    timemory_push_region("spam");

#ifdef BENCHMARK
    timemory_add_components("cpu_roofline_flops");
    timemory_push_region("benchmark");
#endif

    foo();
    bar();

#ifdef BENCHMARK
    timemory_pop_region("benchmark");
    timemory_remove_components("cpu_roofline_flops");
#endif

    timemory_pop_region("spam");
}
```

Roofline Instrumentation (Python)

- Sample using Python API
- When HW counter capabilities are enabled, CPU roofline will always be collected

```python
import os
import timemory

@timemory.util.marker(["wall_clock", "cpu_roofline"])
def spam():
    foo()
    bar()
```

Roofline Instrumentation (Python)

- Sample using Python APIs
- Same metric collection as previous slide
- Push/pop create global storage entries
- “roof” variable provides direct access to the measurement data
- Partial ERT customization available; full customization pending JIT support

```python
import os
import timemory

@timemory.util.marker(["wall_clock"])
def spam():
    roof = None
    if os.environ.get("BENCHMARK", None):
        from timemory.component import CpuRooflineFlops
        roof = CpuRooflineFlops("benchmark")
        roof.push().start()

    foo()
    bar()

    if roof is not None:
        roof.stop().pop()
```

Roofline Instrumentation (C++ Template API)

- Sample using C++ API

```
TIMEMORY_DEFINE_API(benchmark)

#if !defined(BENCHMARK)
TIMEMORY_DEFINE_CONCRETE_TRAIT(
    is_available,
    api::benchmark,
    false_type)
#endif
```

- `benchmark_t` types will get "optimized" out of application when unavailable
    - Join Day 2 of the timemory ECP tutorial for a more detailed explanation

```
using benchmark_t = tim::component_bundle<
    tim::api::benchmark,
    tim::component::wall_clock,
    tim::component::cpu_roofline_flops,
    tim::quirk::auto_start>;

void spam()
{
    benchmark_t _bm{ "spam" };
    foo();
    bar();
}
```