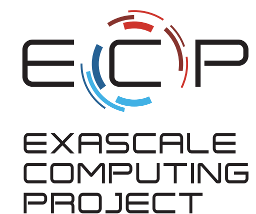
Caliper: A Performance Profiling Library

45th vi-hps Tuning Workshop



June 10, 2024

David Boehme
Computer Scientist

LLNL-PRES-821032
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

Caliper: A Performance Profiling Library

- Integrates a performance profiler into your program
  - Profiling is always available
  - Simplifies performance profiling for application end users

- Common instrumentation interface
  - Provides program context information for other tools

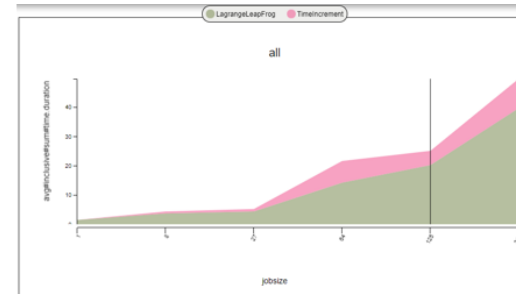
- Designed for HPC
  - MPI, OpenMP, CUDA, HIP, Kokkos support; call-stack sampling; hardware counters; memory profiling

Caliper Use Cases

- Lightweight always-on profiling
  - Performance summary report for each run
- Performance debugging
- Performance introspection
- Comparison studies across runs
  - Performance regression testing
  - Configuration and scaling studies
- Automated workflows

Performance reports

Path	Min time/rank	Max time/rank	Avg time/rank	Time %
main	0.000119	0.000119	0.000119	7.079120
mainloop	0.000067	0.000067	0.000067	3.985723
foo	0.000646	0.000646	0.000646	38.429506
init	0.000017	0.000017	0.000017	1.011303



Comparing runs

```
0.000 foo
├─ 5.000 bar
|  ├─ 5.000 baz
|  └─ 10.000 grault
├─ 0.000 qux
|  └─ 5.000 quux
|     └─ 10.000 corge
|        ├─ 5.000 bar
|        |  ├─ 5.000 baz
|        |  └─ 10.000 grault
|        ├─ 10.000 grault
|        └─ 15.000 garply
└─ 0.000 waldo
```

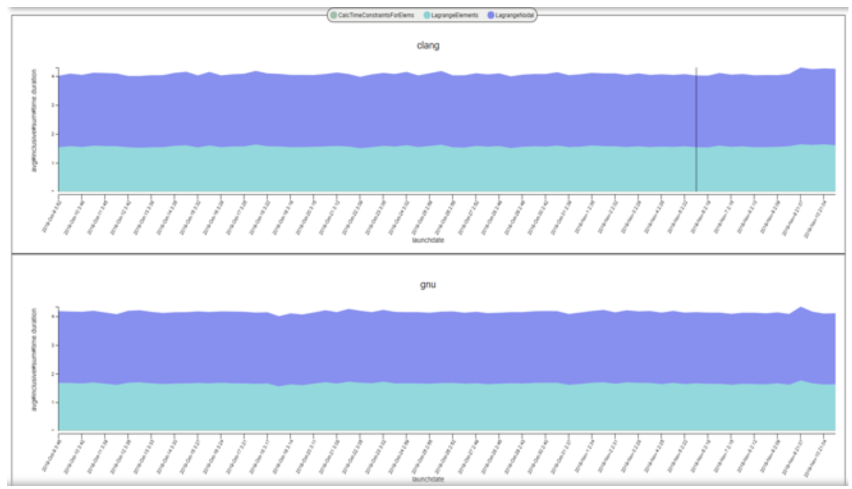
Debugging

Performance Analysis with Caliper, SPOT, Hatchet, and Thicket

“spot” config

```
#include <caliper/cali.h>

void LagrangeElements(Domain& domain,
Index_t numElem)
{
   CALI_CXX_MARK_FUNCTION;
// ...
```

Caliper:
Instrumentation and Profiling



SPOT and Thicket:
Analysis of
large collections of runs

hatchet-region-profile,
hatchet-sample-profile

Pre-populated Jupyter notebooks

```
0.000 foo
├─ 5.000 bar
|  ├─ 5.000 baz
|  └─ 10.000 grault
├─ 0.000 qux
|  └─ 5.000 quux
|     └─ 10.000 corge
|        ├─ 5.000 bar
|        |  ├─ 5.000 baz
|        |  └─ 10.000 grault
|        ├─ 10.000 grault
|        └─ 15.000 garply
└─ 0.000 waldo
```

Hatchet:
Call graph analysis in Python

Materials, Contact & Links

- Tutorial materials: https://github.com/daboehme/caliper-tutorial

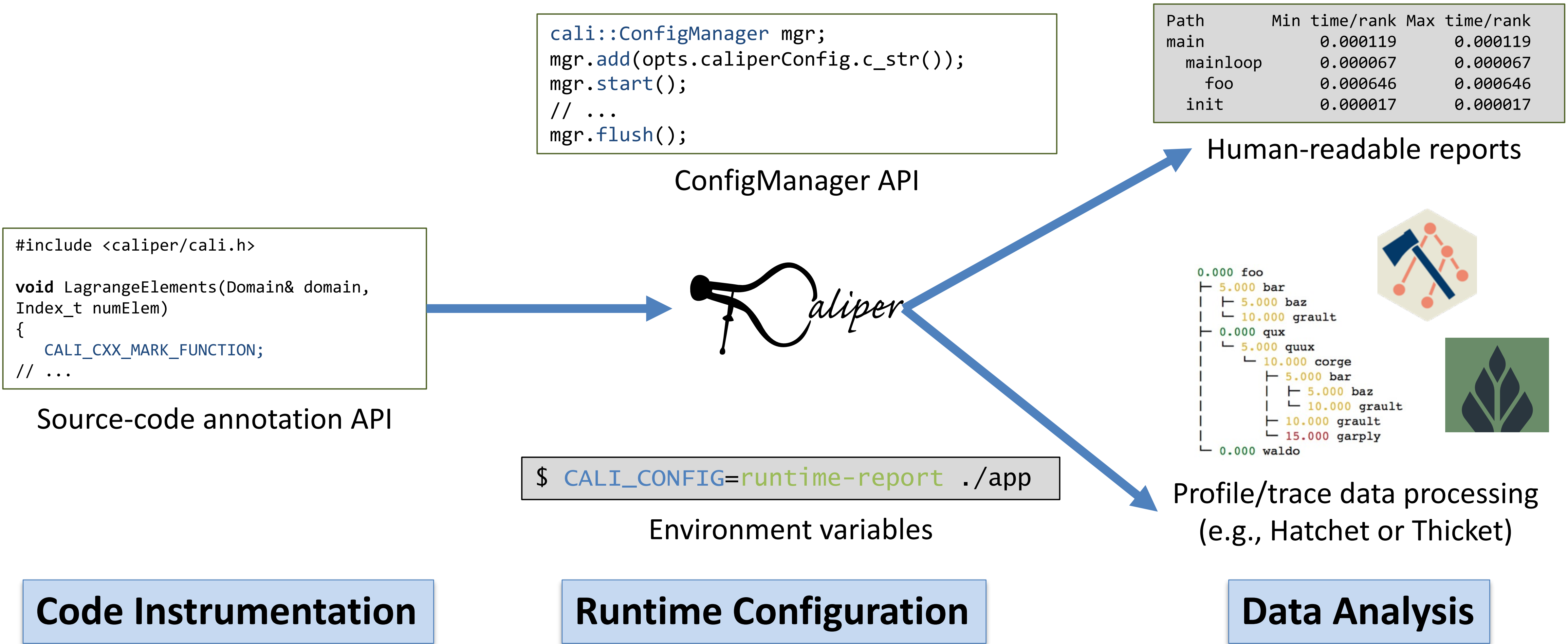
```
$ git clone --recursive https://github.com/daboehme/caliper-tutorial.git
$ . setup-env.sh
```

- GitHub repository: https://github.com/LLNL/Caliper
- Documentation: https://llnl.github.io/Caliper
- GitHub Discussions: https://github.com/LLNL/Caliper/discussions

- Contact: David Boehme (boehme3@llnl.gov)

Using Caliper

Using Caliper: Workflow



Region Profiling: Marking Code Regions

C/C++

```
#include <caliper/cali.h>

void main() {
  CALI_MARK_BEGIN(”init”);

  do_init();

  CALI_MARK_END(”init”);
}
```

Fortran

```
USE caliper_mod


CALL cali_begin_region(‘init’)

CALL do_init()

CALL cali_end_region(‘init’)
```

- Use annotation macros (C/C++) or functions to mark and name code regions

Region Profiling: Marking Phases (New in v2.11)

C/C++

```
#include <caliper/cali.h>

void main() {
  CALI_MARK_PHASE_BEGIN(”hydrodynamics”);

  do_init();

  CALI_MARK_PHASE_END(”hydrodynamics”);
}
```

- Phases mark high-level packages / program phases
- Lets users restrict expensive measurements to phases or produce compact profiles

Region Profiling: Best Practices

- Be selective: Instrument high-level program subdivisions (kernels, phases, ...)
- Be clear: Choose meaningful names
- Start small: Add instrumentation incrementally

```
RAJA::ReduceSum<RAJA::omp_reduce, double> ompdot(0.0);

CALI_MARK_BEGIN(”dotproduct”);

RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
  ompdot += a[i] * b[i];
});
dot = ompdot.get();

CALI_MARK_END(”dotproduct”);
```

Caliper annotations give meaningful names to high-level program constructs

Region Profiling: Printing a Runtime Report

```
$ cd Caliper/build
$ make cxx-example
$ CALI_CONFIG=runtime-report ./examples/apps/cxx-example
```

```
Path       Min time/rank Max time/rank Avg time/rank Time %
main            0.000119      0.000119      0.000119  7.079120
  mainloop      0.000067      0.000067      0.000067  3.985723
    foo         0.000646      0.000646      0.000646 38.429506
  init          0.000017      0.000017      0.000017  1.011303
```

- Set the CALI_CONFIG environment variable to access Caliper's built-in profiling configurations
- "runtime-report" measures, aggregates, and prints time in annotated code regions

List of Caliper's Built-in Profiling Recipes

Config name	Description
runtime-report	Print a time profile for annotated regions
loop-report	Print summary and time-series information for loops
mpi-report	Print time spent in MPI functions
sample-report	Print time spent in regions using call-path sampling
event-trace	Record a trace of region enter/exit events in .cali format
hatchet-region-profile	Record a region time profile for processing with hatchet or cali-query
hatchet-sample-profile	Record a sampling profile for processing with hatchet or cali-query
spot	Record a time profile for the SPOT web visualization framework or Thicket

Use `cali-query --help=configs` to list all built-in configs and their options

Built-In Profiling Recipes: Configuration String Syntax

Config name specifies the kind of performance measurement

Parameters enable additional features, metrics, or output options

```
$ CALI_CONFIG="runtime-report(mem.highwatermark,output=stdout)" ./examples/apps/cxx-example
```

```
Path         Min time/rank Max time/rank Avg time/rank   Time % Allocated MB
main              0.000179      0.000179      0.000179 2.054637     0.000047
  mainloop        0.000082      0.000082      0.000082 0.941230     0.000016
    foo           0.000778      0.000778      0.000778 8.930211     0.000016
  init            0.000020      0.000020      0.000020 0.229568     0.000000
```

- Most Caliper measurement recipes have optional parameters to enable additional features or configure output settings

Profiling Options: MPI Function Profiling

```
$ CALI_CONFIG=runtime-report,profile.mpi ./lulesh2.0
```

```
Path                                       Min time/rank Max time/rank Avg time/rank Time %
MPI_Comm_dup                                    0.000034      0.003876      0.001999  0.10089
main                                            0.009013      0.010797      0.010173  0.51335
  MPI_Reduce                                    0.000031      0.000049      0.000037  0.001886
  lulesh.cycle                                  0.002031      0.002258      0.002085  0.105220
    LagrangeLeapFrog                            0.002158      0.002511      0.002227  0.112366
      CalcTimeConstraintsForElems               0.015166      0.015443      0.015277  0.770922
      CalcQForElems                             0.058781      0.060196      0.059699  3.01254
          CalcMonotonicQForElems                0.035331      0.041057      0.038496  1.942601
          CommMonoQ                             0.005280      0.006152      0.005544  0.279781
            MPI_Wait                            0.004182      0.084533      0.035324  1.78249
          CommSend                              0.006893      0.009062      0.008071  0.407298
            MPI_Waitall                         0.000986      0.001778      0.001343  0.067789
            MPI_Isend                           0.004564      0.005785      0.004930  0.248765
          CommRecv                              0.002265      0.002616      0.002341  0.118144
[...]
```

The profile.mpi option measures time spent in MPI functions

Use cali-query to get Help for Profiling Recipes

```
$ cali-query --help runtime-report
```

```
runtime-report
 Print a time profile for annotated regions
  Options:
   aggregate_across_ranks    Aggregate results across MPI ranks
   calc.inclusive            Report inclusive instead of exclusive times
   cuda.gputime              Report GPU time in CUDA activities
   exclude_regions           Do not take snapshots for the given region names/patterns.
   include_branches          Only take snapshots for branches with the given region names.
   include_regions           Only take snapshots for the given region names/patterns.
   io.bytes                  Report I/O bytes written and read
   io.bytes.read             Report I/O bytes read
   io.bytes.written          Report I/O bytes written
   io.read.bandwidth         Report I/O read bandwidth
   io.write.bandwidth        Report I/O write bandwidth
   level                     Minimum region level that triggers snapshots
   main_thread_only          Only include measurements from the main thread in results.
   max_column_width          Maximum column width in the tree display
...
```

Profiling Options: CUDA Profiling

```
$ lrun -n 4 ./tea_leaf runtime-report,profile.cuda
```

```
Path                          Min time/rank Max time/rank Avg time/rank Time %
timestep_loop                      0.000175      0.000791      0.000345  0.002076
[...]
  total_solve                      0.000105      0.000689      0.000252  0.001516
    solve                          0.583837      0.617376      0.594771  3.581811
      dot_product                  0.000936      0.001015      0.000969  0.005837
      cudaMalloc                   0.000060      0.000066      0.000063  0.000382
      internal_halo_update         0.077627      0.079476      0.078697  0.473925
      halo_update                  0.158597      0.161853      0.160023  0.963685
      halo_exchange                1.502106      1.572522      1.532860  9.231136
        cudaMemcpy                11.840890     11.871018     11.860343 71.424929
        cudaLaunchKernel           1.177454      1.230816      1.211668  7.296865
      cudaMemcpy                   0.470123      0.471485      0.470596  2.834008
      cudaLaunchKernel             0.658269      0.682566      0.673030  4.053100
[...]
```

The `profile.cuda` option measures time in CUDA runtime API calls

Sample Profiling

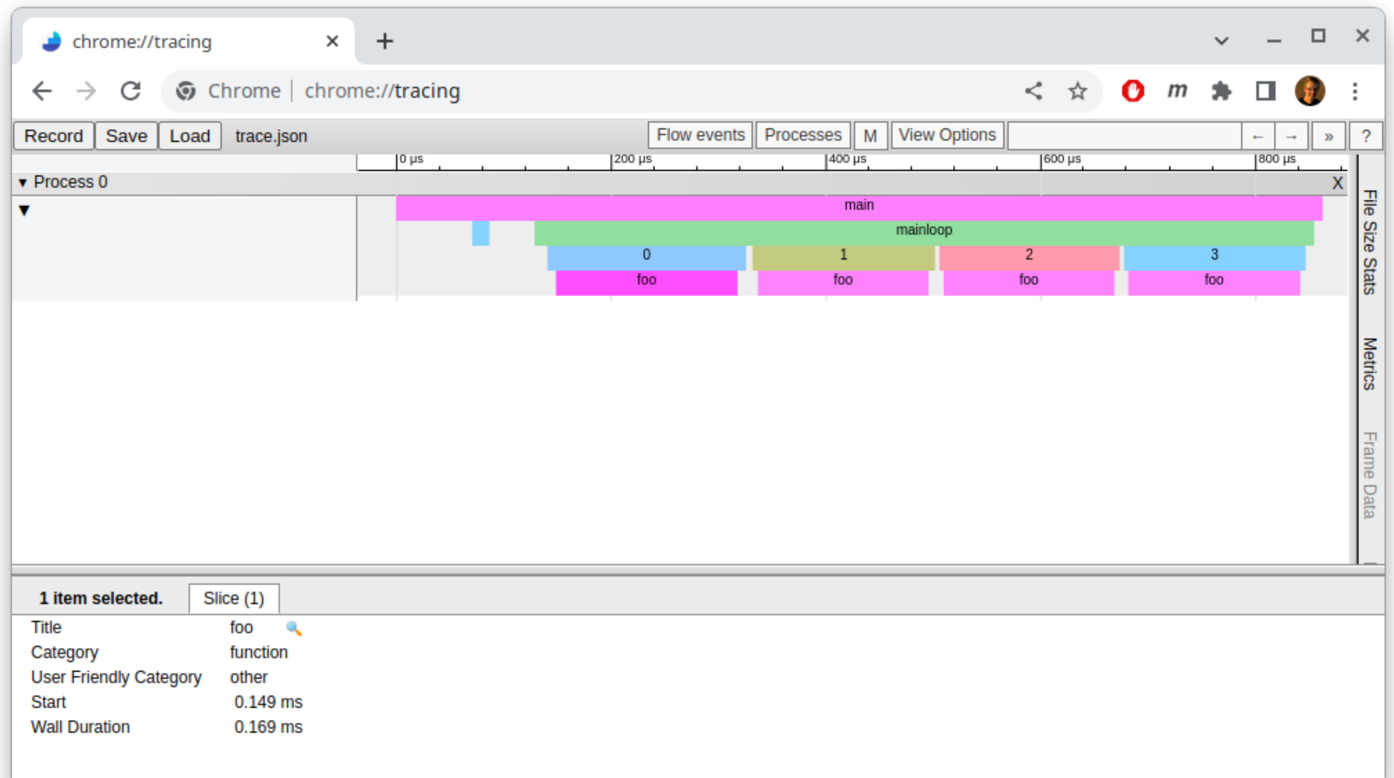
```
$ CALI_CONFIG=sample-report ./lulesh2.0
```

```
Path    Min time/rank Max time/rank Avg time/rank Total time   Time % Function
main
 |-          0.005000      0.005000      0.005000   0.035000 0.059691 Domain::AllocateElemPersistent
 |-          0.005000      0.005000      0.005000   0.035000 0.059691 Domain::SetupThreadSupportStru
 |-          0.005000      0.005000      0.005000   0.005000 0.008527 sysmalloc
 |-          0.005000      0.005000      0.005000   0.005000 0.008527 Domain::BuildMesh(int, int, in
lulesh.cycle
   TimeIncrement
     |-      0.075000      0.740000      0.355000   2.840000 4.843523 gomp_barrier_wait_end
     |-      0.005000      0.060000      0.027857   0.195000 0.332566 psm2_mq_ipeek2
     |-      0.005000      0.005000      0.005000   0.015000 0.025582 psm_no_lock
     |-      0.005000      0.060000      0.023571   0.165000 0.281402 psm_progress_wait
     |-      0.015000      0.030000      0.022143   0.155000 0.264347 mv2_shm_bcast
     |-      0.005000      0.025000      0.013750   0.055000 0.093801 amsh_poll
     |-      0.005000      0.010000      0.007500   0.030000 0.051164 psmi_poll_internal
...
```

The *sample-report* recipe samples source functions or file+line locations.

Event Tracing and Timeline Visualization

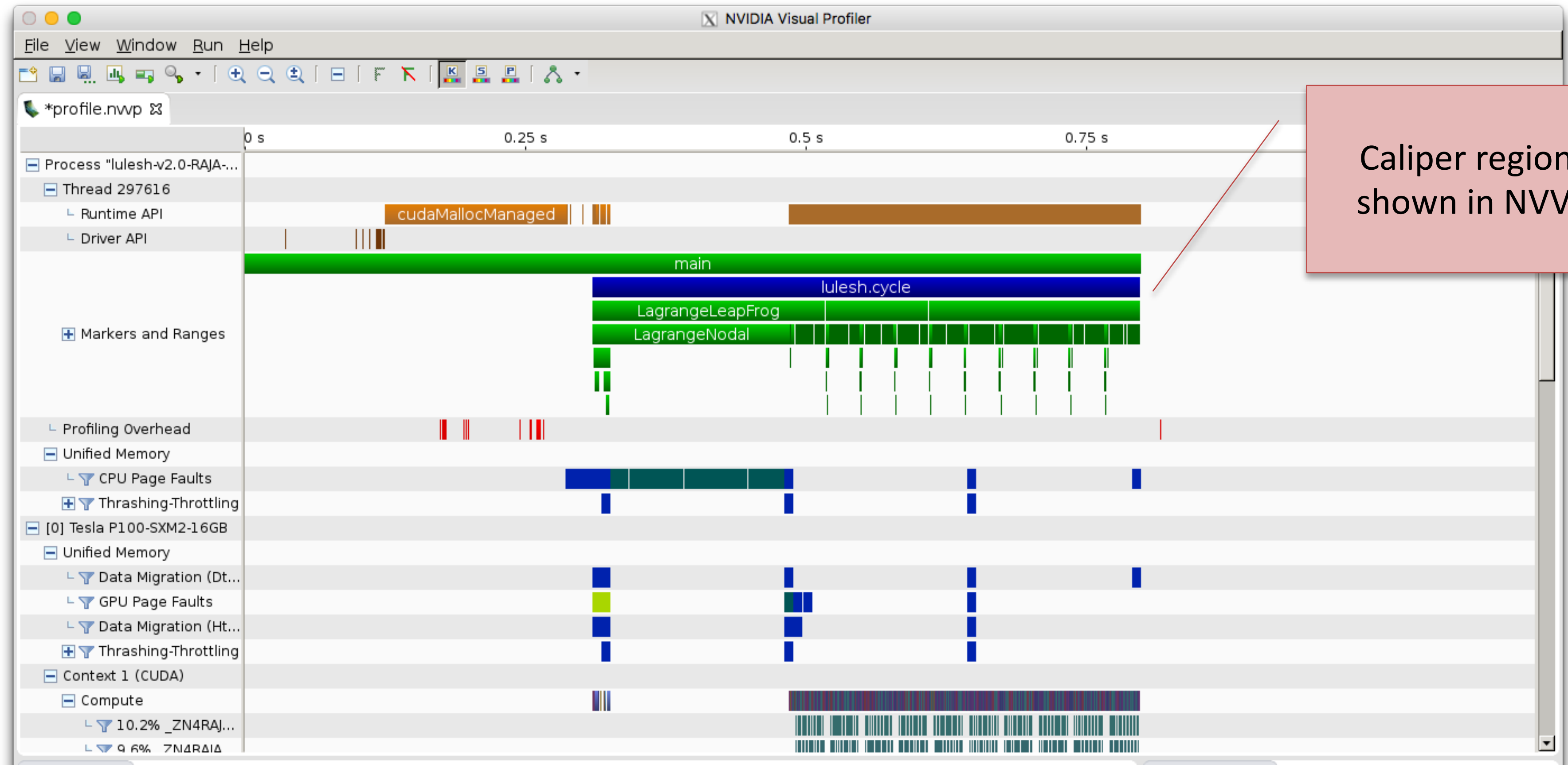
```
$ CALI_CONFIG=event-trace,output=trace.cali ./lulesh2.0
$ cali2traceevent.py trace.cali trace.json
```



Caliper event traces can be visualized in Chrome trace browser or Perfetto

Forwarding Annotations to Third-Party Tools

```
$ CALI_CONFIG=nvtx nvprof <nvprof-opts> ./app
```



Caliper regions shown in NVVP

The nvtx config forwards annotations to NVidia's NVTX API

Call Graph Analysis with the Hatchet Python Library

- Caliper records data for hatchet with `hatchet-region-profile` or `hatchet-sample-profile`

```
$ CALI_CONFIG=hatchet-region-profile srun -n 8 ./lulesh2.0
```

Hatchet allows manipulation, computation, comparison, and visualization of call graph data

```
>>> import hatchet
>>> gf = hatchet.GraphFrame.from_caliperreader("region_profile_mpi_8x4.cali")
>>> gf.drop_index_levels()
>>> gf.subgraph_sum(["time"])
>>> gff = gf.filter(lambda x: x["time"] > 0.05).squash()
>>> print(gff.tree())

v2024.1.2

0.836 main
└─ 0.833 lulesh.cycle
   └─ 0.830 LagrangeLeapFrog
      ├─ 0.319 LagrangeElements
      │  ├─ 0.198 ApplyMaterialPropertiesForElems
      │  │  └─ 0.193 EvalEOSForElems
      │  │     └─ 0.129 CalcEnergyForElems
      │  ├─ 0.061 CalcLagrangeElements
      │  │  └─ 0.058 CalcKinematicsForElems
      │  └─ 0.059 CalcQForElems
      └─ 0.506 LagrangeNodal
         └─ 0.468 CalcForceForNodes
            └─ 0.436 CalcVolumeForceForElems
               ├─ 0.330 CalcHourglassControlForElems
               │  └─ 0.157 CalcFBHourglassForceForElems
               └─ 0.102 IntegrateStressForElems
```

Generating Reports with cali-query

```
$ mpirun -n 8 lulesh2.0 -P hatchet-region-profile,profile.mpi
$ cali-query -q “select mpi.rank,sum#sum#time.duration where mpi.function=MPI_Allreduce format tree”
    region_profile.cali
```

```
Path                   mpi.rank time
main
  lulesh.cycle
    TimeIncrement
      MPI_Allreduce
        |-                    0 0.002174
        |-                    1 0.083150
        |-                    2 0.100689
        |-                    3 0.171060
        |-                    4 0.171347
        |-                    5 0.204563
        |-                    6 0.161819
        |-                    7 0.203488
```

- Run SQL-like queries on Caliper output with the cali-query tool

Control Profiling Programmatically: The ConfigManager API

```
#include <caliper/cali.h>
#include <caliper/cali-manager.h>

int main(int argc, char* argv[])
{
  cali::ConfigManager mgr;
  mgr.add(argv[1]);
  if (mgr.error())
    std::cerr << mgr.error_msg() << "\n";

  mgr.start();
  // ...
  mgr.flush();
}
```

- Use ConfigManager to access Caliper's built-in profiling configurations

```
$ ./app runtime-report
```

- Now we can use command-line arguments or other program inputs to enable profiling

Manual Configuration Allows Custom Analyses

```
cali-query -q "select alloc.label#cupti.fault.addr as Pool,
  cupti.uvm.kind as UVM\ Event,
  scale(cupti.uvm.bytes,1e-6) as MB,
  scale(cupti.activity.duration,1e-9) as Time
group by
  prop:nested,alloc.label#cupti.fault.addr,cupti.uvm.kind
where cupti.uvm.kind format tree” trace.cali
```

caliper.config

```
CALI_SERVICES_ENABLE=alloc,cupti,cuptitrace,mpi,trace,recorder
CALI_ALLOC_RESOLVE_ADDRESSES=true
CALI_CUPTI_CALLBACK_DOMAINS=sync
CALI_CUPTITRACE_ACTIVITIES=uvm
CALI_CUPTITRACE_CORRELATE_CONTEXT=false
CALI_CUPTITRACE_FLUSH_ON_SNAPSHOT=true
```

```
Path
main
  solve
    TIME_STEPPING
      enforceBC
        CURVI in EnforceBC
          CurviCartIC
            CurviCartIC::PART 3  Pool          UVM Event      MB          Time
              curvilinear4sgwind UM_pool       pagefaults.gpu             2.806946
              curvilinear4sgwind UM_pool       HtoD           7862.747136 0.232238
              curvilinear4sgwind UM_pool_temps pagefaults.gpu             0.130167
              curvilinear4sgwind UM_pool       DtoH           9986.441216 0.378583
              curvilinear4sgwind UM_pool       pagefaults.cpu
```

- Mapping CPU/GPU unified memory transfer events to Umpire memory pools in SW4

Recording Program Metadata with the Adiak Library

TeaLeaf_CUDA example [C++]

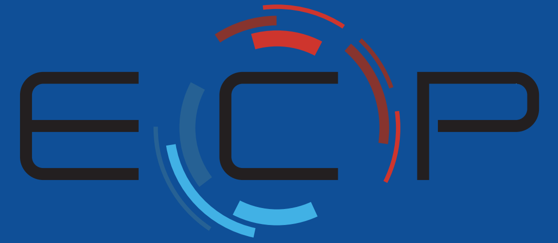
```
#include <adiak.hpp>

adiak::user();
adiak::launchdate();
adiak::jobsize();

adiak::value("end_step", readInt(input, "end_step"));
adiak::value(”halo_depth", readInt(input, ”halo_depth"));

if (tl_use_ppcg) {
  adiak::value(“solver”, “PPCG”);
// [...]
```

Use built-in Adiak functions to collect common metadata

Use key:value functions to collect program-specific data

- Use the Adiak C/C++ library to record program metadata
  - Environment info (user, launchdate, system name, ...)
  - Program configuration (input problem description, problem size, ...)
- Enables performance comparisons across runs. Required for SPOT and Thicket.

Live Demo



This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

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.