SAND2021-10556PE

kokkos

Kokkos Tools

EXASCALE COMPUTING PROJECT

The Kokkos Team

September 3, 2021

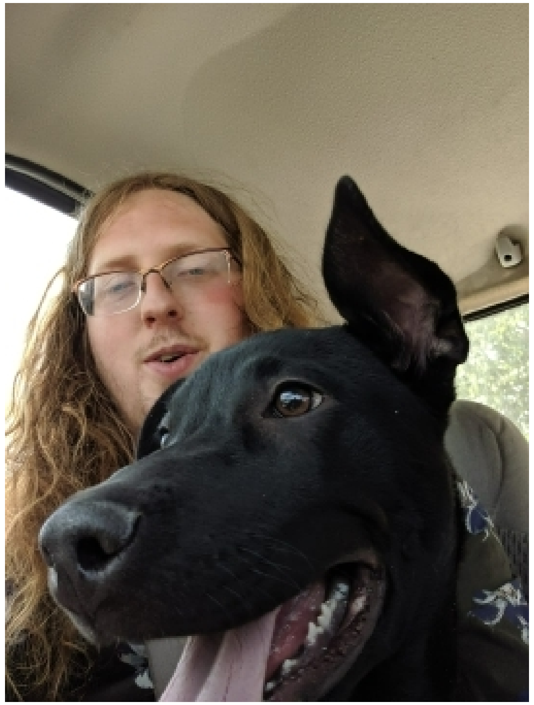
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Speaker Intro

- Work for Sandia
  - Application Performance Team
- Worked for LLNL for a while
- Help multiphysics code teams leverage research efforts
- Make prototypes to answer “how the heck will we _______ on this weird architecture?”

What is the Kokkos Tools effort?

- Kokkos aims to provide a unified interface to a variety of hardware and programming models
- Kokkos *Tools* does the same, but for tooling
- Current mature capability areas
  - Profiling
  - Autotuning
- Exploratory
  - Compilers
  - IDE integrations
  - Debuggers



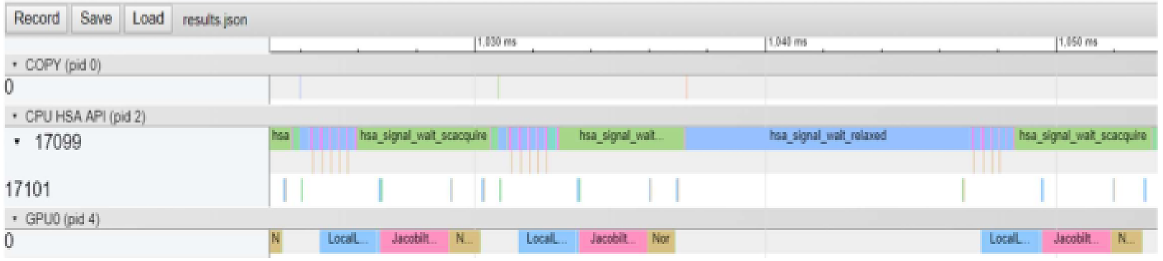
David Poliakoff:
Profiling tools,
Debuggers,
Autotuning,
IDEs,
Dog facts

Drew Lewis:
Compilers

Why Kokkos Tools?

- “Toolchain-per-architecture” undesirable

NVIDIA nsys



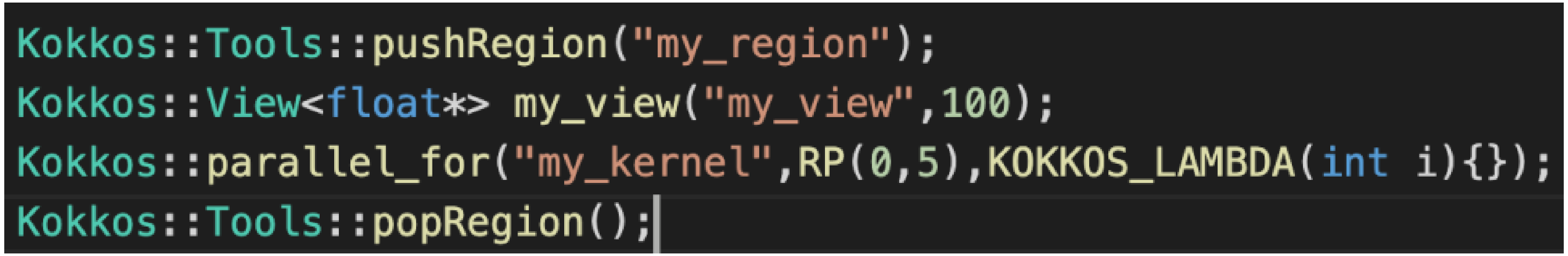
AMD Rocprof

- Tooling with Kokkos Semantics, not C++
- In C++: “void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::ParallelFor<_GLOBAL__N__49_tmpxft_00004d6b_00000000_7_integrator_nve_cpp1_ii_28abe736::InitialIntegrateFunctor, Kokkos::RangePolicy<>, Kokkos::Cuda>>(_GLOBAL__N__49_tmpxft_00004d6b_00000000_7_integrator_nve_cpp1_ii_28abe736::InitialIntegrateFunctor”
- In Kokkos: “IntegratorNVE::initial_integrate”

Design/Goals

- Don’t have “tool-enabled builds,” always enable tools. Turning them on or off should be a *runtime decision*
  - Necessitates *zero or very low overhead when not in use* (we achieve this)
- No versioned tool-side headers
- Function-pointer callback-based system. On Unix, we dlopen a tool library and fill out function pointers from it
  - Comparing those function pointers to nullptr is very fast
- Events we track
  - Kernels, Regions, Metadata, Memory alloc/free (including Views), DualView operations
- Events we will soon track
  - Using a View in a kernel

How do I integrate these into my Kokkos code?



```
Kokkos::Tools::pushRegion("my_region");
Kokkos::View<float*> my_view("my_view",100);
Kokkos::parallel_for("my_kernel",RP(0,5),KOKKOS_LAMBDA(int i){});
Kokkos::Tools::popRegion();
```

Instrumentation “built-in” to Kokkos Core

```
./my_application [run with no tool]

./my_application --kokkos-tools-library=/path/to/tool.so [run with a tool]

KOKKOS_PROFILE_LIBRARY=/path/to/tool.so ./my_application  [run with a tool]
```

No recompilation, just add a command
line argument!

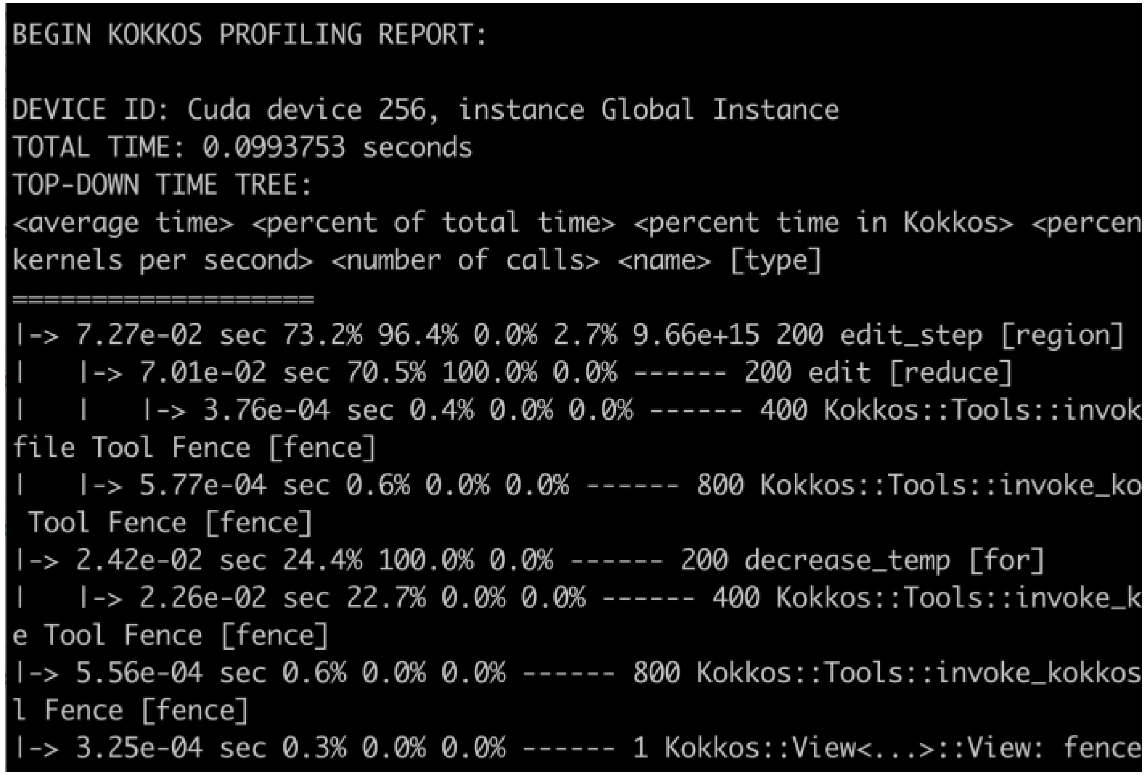
Where to get Tools that support this?

- Kokkos Tools repo
  - git@github.com:kokkos/kokkos-tools
  - Simple tools to do simple tasks, builds are *trivial* (just type “make”)
- Caliper
  - git@github.com:LLNL/caliper
  - More complicated, more powerful. I (David P) tend to prototype functionality here
  - UVM Profiling, SPOT performance tracking
- APEX
  - git@github.com:khuck/xpress-apex
  - Developed out of University of Oregon, popular with many ORNL users
  - Supports profiling a wide variety of programming models, *and* autotuning
  - Handles asynchronous tasks, unlike many other tools
  - Slices, dices, juliennes fries

Simple Tools

Why Simple Tools?

- *Suppose* DOE was purchasing new architectures with new toolchains at an incredible clip
  - I know, it’s inconceivable
- Do we *really* have to learn a toolchain per architecture for simple tasks?
  - No

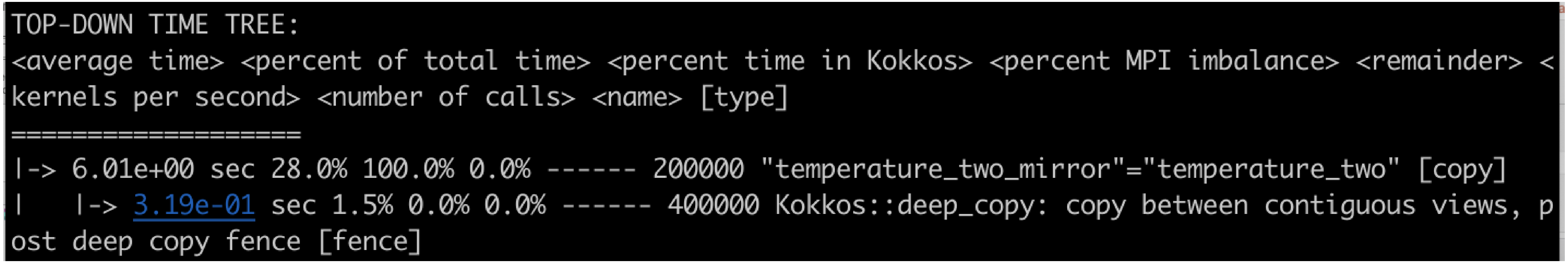


```
BEGIN KOKKOS PROFILING REPORT:

DEVICE ID: Cuda device 256, instance Global Instance
TOTAL TIME: 0.0993753 seconds
TOP-DOWN TIME TREE:
<average time> <percent of total time> <percent time in Kokkos> <percen
kernels per second> <number of calls> <name> [type]
===================
|-> 7.27e-02 sec 73.2% 96.4% 0.0% 2.7% 9.66e+15 200 edit_step [region]
|   |-> 7.01e-02 sec 70.5% 100.0% 0.0% ------ 200 edit [reduce]
|   |   |-> 3.76e-04 sec 0.4% 0.0% 0.0% ------ 400 Kokkos::Tools::invok
file Tool Fence [fence]
|   |-> 5.77e-04 sec 0.6% 0.0% 0.0% ------ 800 Kokkos::Tools::invoke_ko
 Tool Fence [fence]
|-> 2.42e-02 sec 24.4% 100.0% 0.0% ------ 200 decrease_temp [for]
|   |-> 2.26e-02 sec 22.7% 0.0% 0.0% ------ 400 Kokkos::Tools::invoke_k
e Tool Fence [fence]
|-> 5.56e-04 sec 0.6% 0.0% 0.0% ------ 800 Kokkos::Tools::invoke_kokkos
l Fence [fence]
|-> 3.25e-04 sec 0.3% 0.0% 0.0% ------ 1 Kokkos::View<...>::View: fence
```

Space-time Stack:
where am I spending
time and memory?

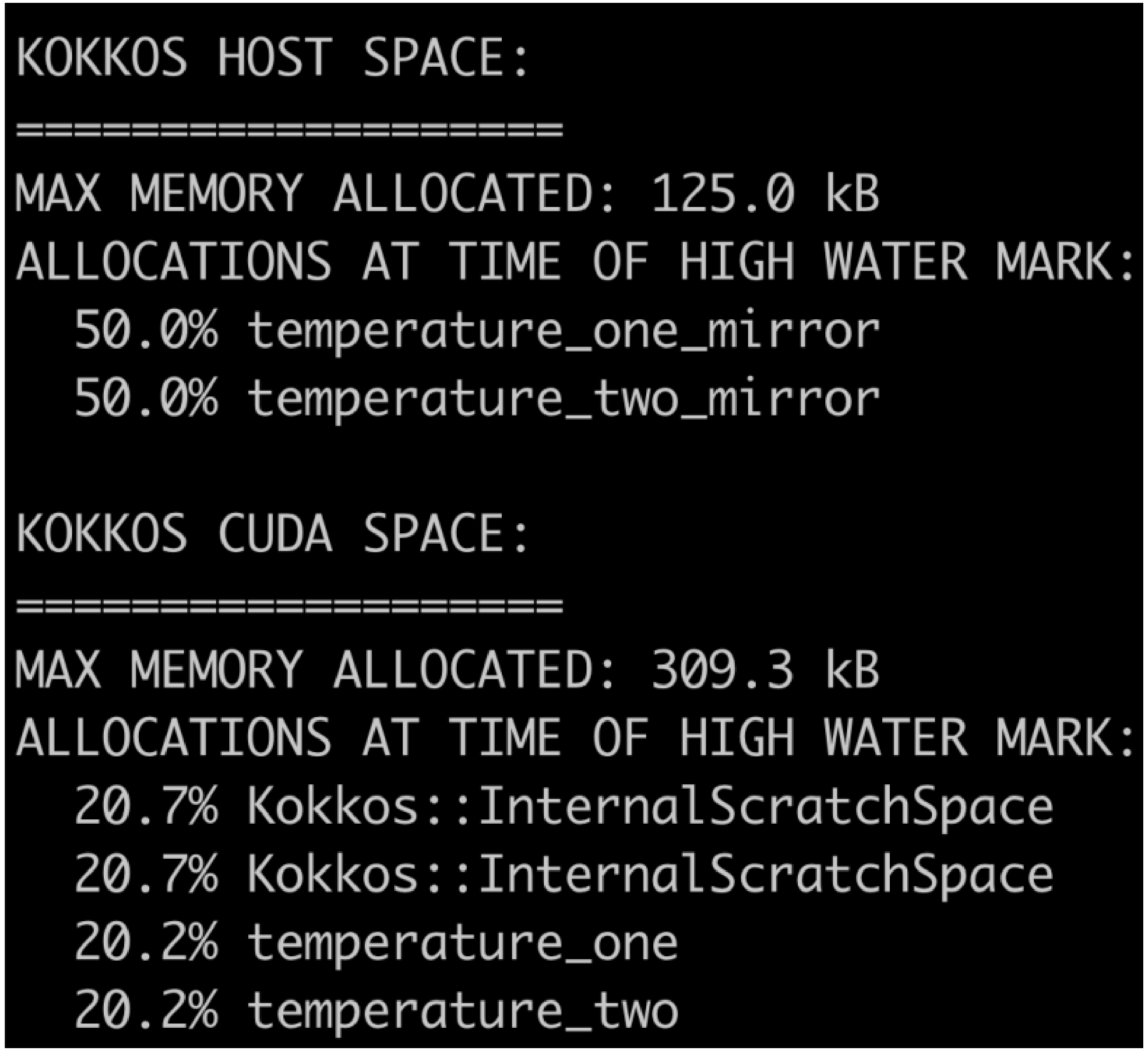
Space-Time Stack: Dead simple, highly useful tool

- For this part, I recommend using your own Kokkos code. If you don't have one, though, try the "instances" example in the examples repo
- Running is *extremely* complicated:
  - Set KOKKOS_PROFILE_LIBRARY to [examples install dir]/lib64/kp_space_time_stack.so
  - Run your program



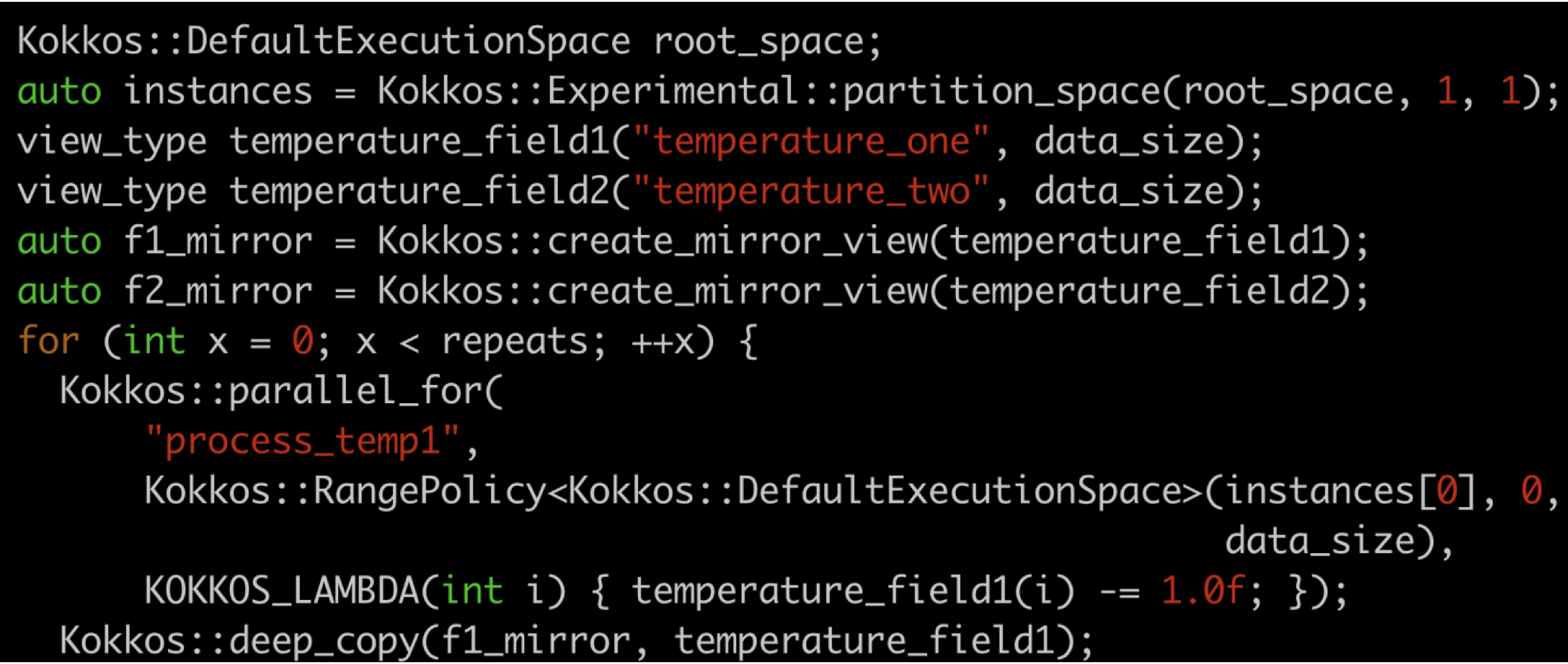
```
TOP-DOWN TIME TREE:
<average time> <percent of total time> <percent time in Kokkos> <percent MPI imbalance> <remainder> <
kernels per second> <number of calls> <name> [type]
===================
|-> 6.01e+00 sec 28.0% 100.0% 0.0% ------ 200000 "temperature_two_mirror"="temperature_two" [copy]
|   |-> 3.19e-01 sec 1.5% 0.0% 0.0% ------ 400000 Kokkos::deep_copy: copy between contiguous views, p
ost deep copy fence [fence]
```

Space-time-stack: continued



```
KOKKOS HOST SPACE:
===================
MAX MEMORY ALLOCATED: 125.0 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  50.0% temperature_one_mirror
  50.0% temperature_two_mirror

KOKKOS CUDA SPACE:
===================
MAX MEMORY ALLOCATED: 309.3 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  20.7% Kokkos::InternalScratchSpace
  20.7% Kokkos::InternalScratchSpace
  20.2% temperature_one
  20.2% temperature_two
```

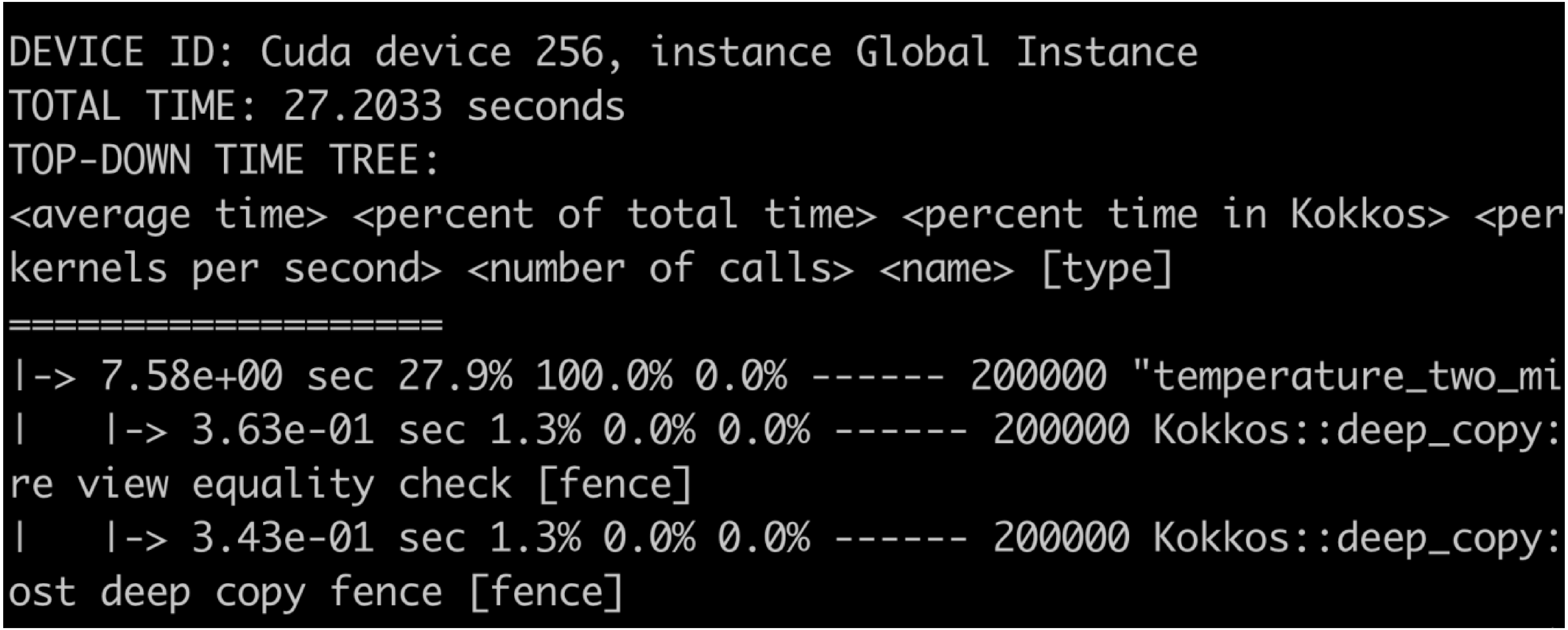
Simple Tools: Advanced Mode



```
Kokkos::DefaultExecutionSpace root_space;
auto instances = Kokkos::Experimental::partition_space(root_space, 1, 1);
view_type temperature_field1("temperature_one", data_size);
view_type temperature_field2("temperature_two", data_size);
auto f1_mirror = Kokkos::create_mirror_view(temperature_field1);
auto f2_mirror = Kokkos::create_mirror_view(temperature_field2);
for (int x = 0; x < repeats; ++x) {
  Kokkos::parallel_for(
      "process_temp1",
      Kokkos::RangePolicy<Kokkos::DefaultExecutionSpace>(instances[0], 0,
                                                         data_size),
      KOKKOS_LAMBDA(int i) { temperature_field1(i) -= 1.0f; });
  Kokkos::deep_copy(f1_mirror, temperature_field1);
```

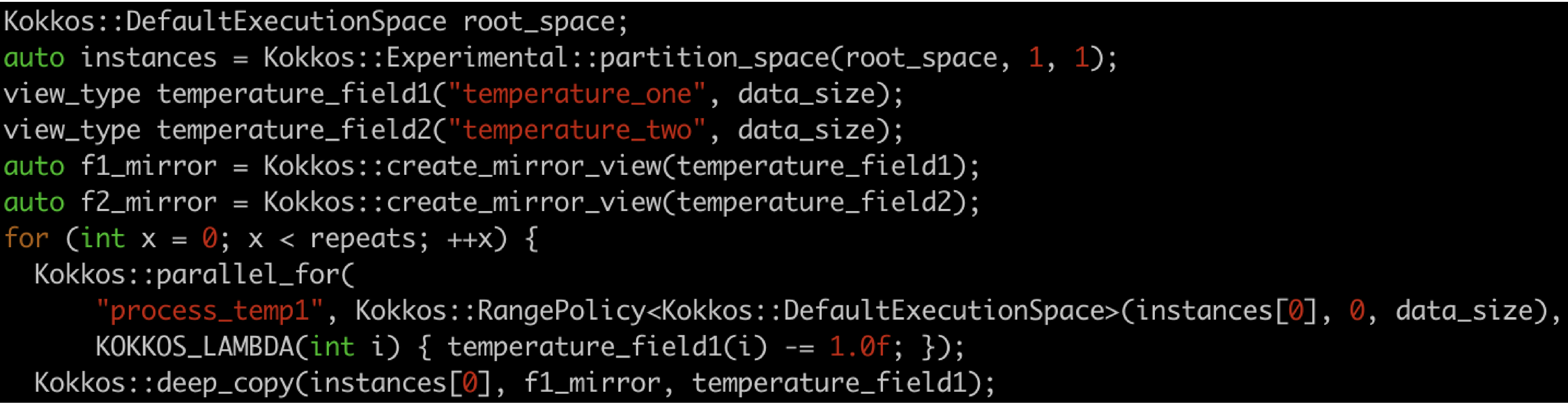
Finding fences

```
KOKKOS_PROFILE_LIBRARY=./lib64/kp_space_time_stack.so
./bin/instances_begin --kokkos-tools-args=--separate-devices
```



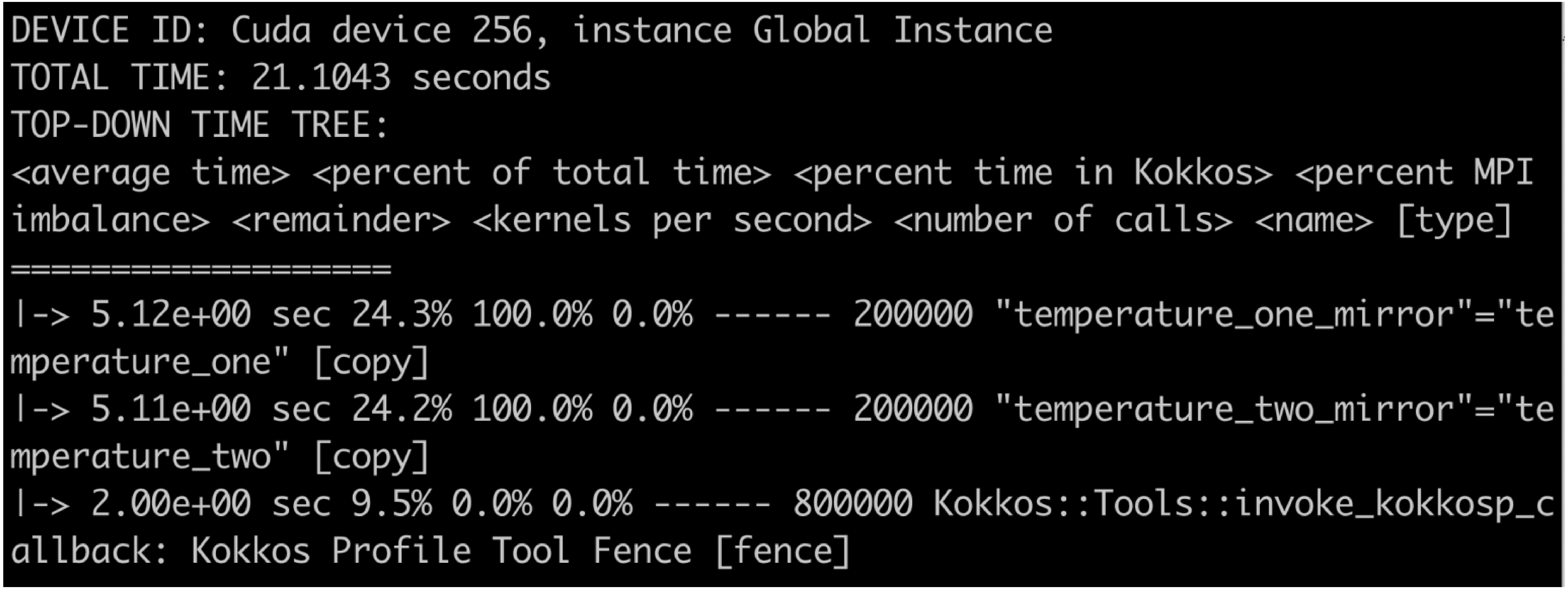
```
DEVICE ID: Cuda device 256, instance Global Instance
TOTAL TIME: 27.2033 seconds
TOP-DOWN TIME TREE:
<average time> <percent of total time> <percent time in Kokkos> <per
kernels per second> <number of calls> <name> [type]
===================
|-> 7.58e+00 sec 27.9% 100.0% 0.0% ------ 200000 "temperature_two_mi
|   |-> 3.63e-01 sec 1.3% 0.0% 0.0% ------ 200000 Kokkos::deep_copy:
re view equality check [fence]
|   |-> 3.43e-01 sec 1.3% 0.0% 0.0% ------ 200000 Kokkos::deep_copy:
ost deep copy fence [fence]
```

Fixed



```
Kokkos::DefaultExecutionSpace root_space;
auto instances = Kokkos::Experimental::partition_space(root_space, 1, 1);
view_type temperature_field1("temperature_one", data_size);
view_type temperature_field2("temperature_two", data_size);
auto f1_mirror = Kokkos::create_mirror_view(temperature_field1);
auto f2_mirror = Kokkos::create_mirror_view(temperature_field2);
for (int x = 0; x < repeats; ++x) {
  Kokkos::parallel_for(
      "process_temp1", Kokkos::RangePolicy<Kokkos::DefaultExecutionSpace>(instances[0], 0, data_size),
      KOKKOS_LAMBDA(int i) { temperature_field1(i) -= 1.0f; });
  Kokkos::deep_copy(instances[0], f1_mirror, temperature_field1);
```

Note: the only fences are Tool fences



```
DEVICE ID: Cuda device 256, instance Global Instance
TOTAL TIME: 21.1043 seconds
TOP-DOWN TIME TREE:
<average time> <percent of total time> <percent time in Kokkos> <percent MPI
imbalance> <remainder> <kernels per second> <number of calls> <name> [type]
===================
|-> 5.12e+00 sec 24.3% 100.0% 0.0% ------ 200000 "temperature_one_mirror"="te
mperature_one" [copy]
|-> 5.11e+00 sec 24.2% 100.0% 0.0% ------ 200000 "temperature_two_mirror"="te
mperature_two" [copy]
|-> 2.00e+00 sec 9.5% 0.0% 0.0% ------ 800000 Kokkos::Tools::invoke_kokkosp_c
allback: Kokkos Profile Tool Fence [fence]
```

Autotuning

Why autotune?

The last 10-15% of performance in a Kokkos app comes from setting a few tuning knobs. These need to be maintained per:

Hardware: Intel, AMD, and NVIDIA GPU

Programming models: Serial, OpenMP, OpenMPTarget, CUDA, HIP, SYCL, Threads, HPX

Compilers: NVCC, Clang, GCC, vendor clang variants

How do you feel about maintaining that many heuristics?

Heuristics aren’t feasible moving forward

Requirements: what can't we do?

- Recall from Profiling:
  - No recompilation
  - Applications can't *fail* if tools aren't available
  - No third-party dependencies in Kokkos
- Good news: there are many good tuning technologies in ECP we can use
- Bad news: there are *too many* good tuning technologies in ECP to pick one
- Answer: abstraction through a callback interface

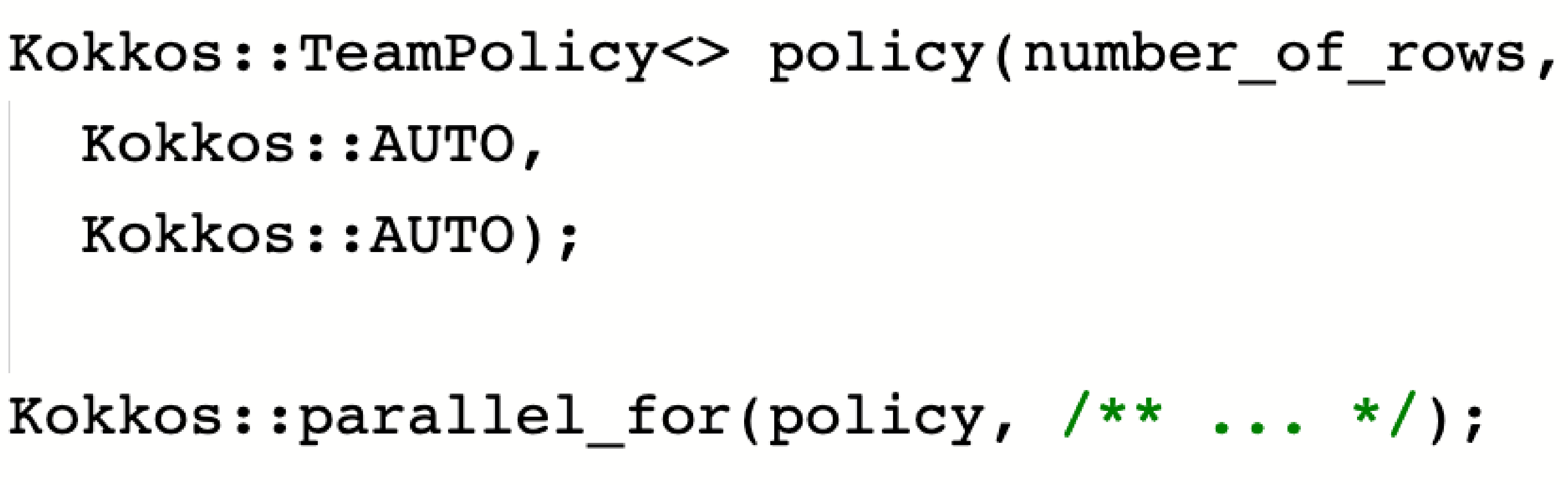
Need to support a variety of tools, ECP has *depth* in this area

Based on our original tuner: Christian trott

- How does the Trott Tuner think about things like sparse matrix vector product?
  - What do I need to tune?
    - An implementation
    - Team sizes for my team policies
  - What might affect my choice?
    - Number of rows in the matrix
    - Sparsity
    - Backend I’m using
  - What options are valid?
    - Maybe a set, maybe a range?
  - Our autotuners need the information Christian has, provided formally

Need to support a variety of tools, ECP has *depth* in this area

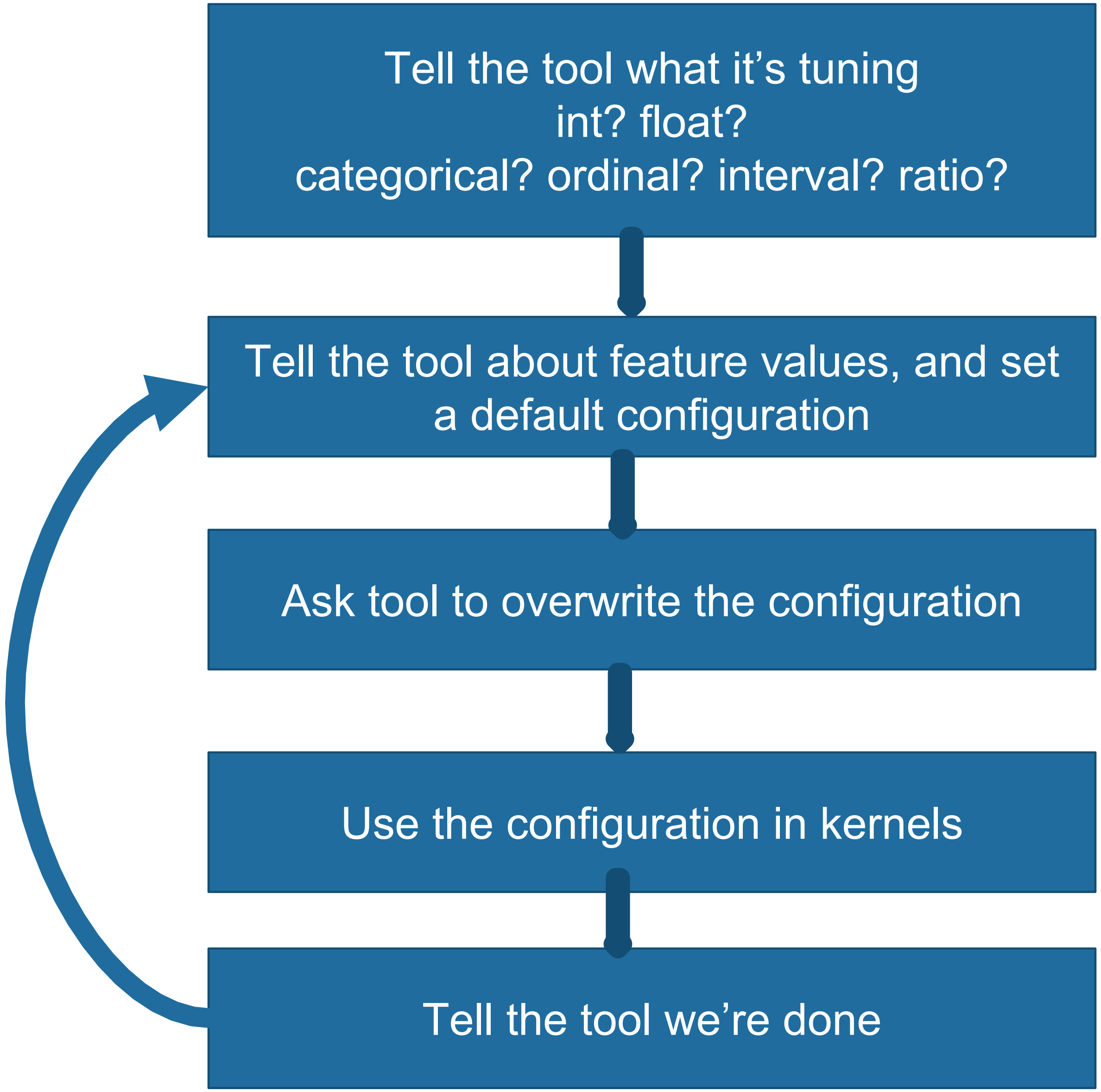
App workflow: typical



```
Kokkos::TeamPolicy<> policy(number_of_rows,
  Kokkos::AUTO,
  Kokkos::AUTO);


Kokkos::parallel_for(policy, /** ... */);
```

In most cases, code changes very little

App workflow: advanced



Tell the tool what it’s tuning
int? float?
categorical? ordinal? interval? ratio?

Tell the tool about feature values, and set a default configuration

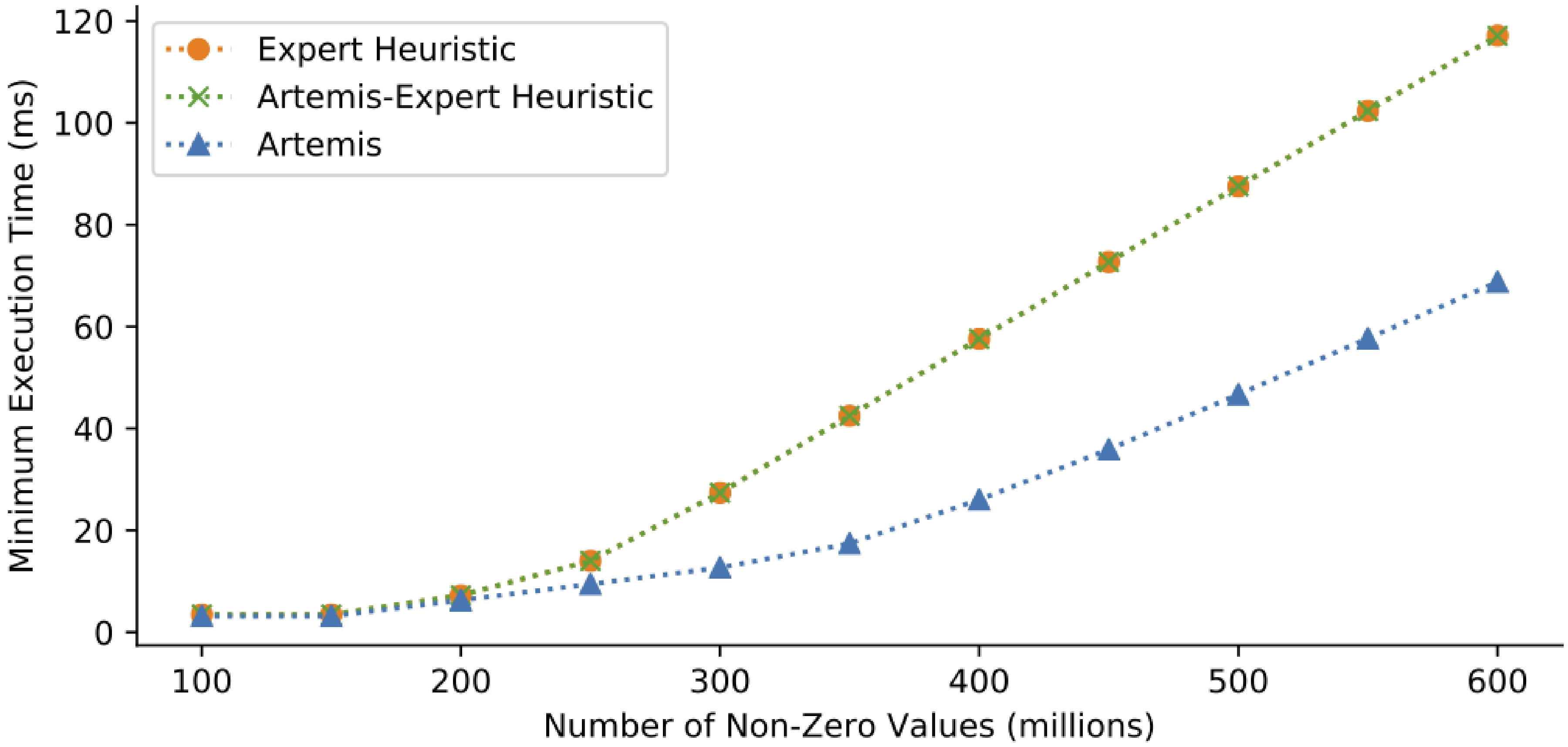
Ask tool to overwrite the configuration

Use the configuration in kernels

Tell the tool we’re done

While app not done

SPMV sees speedups



Graph courtesy of Artemis team from LLNL and UO

Speedups on hand-tuned kernels. Reproduced with multiple tools

EXASCALE COMPUTING PROJECT

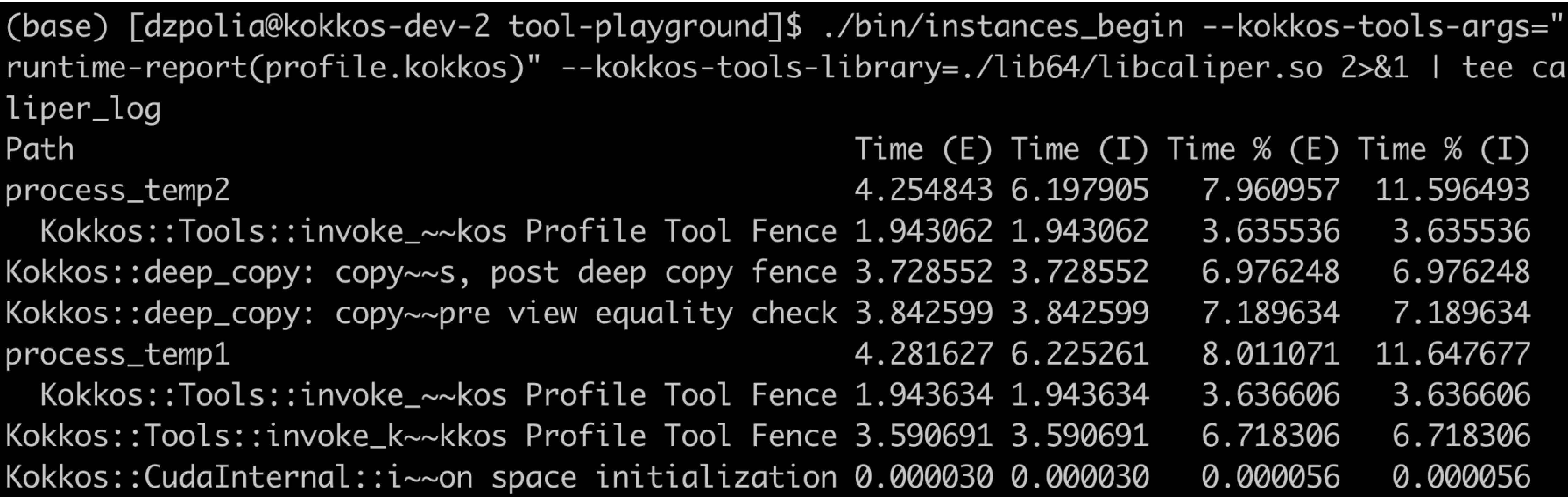
Caliper

Caliper: a Performance Analysis Toolbox

- Developed at Lawrence Livermore National Lab
- https://software.llnl.gov/Caliper/
- *Significantly* more than a Kokkos Tool, but a great Kokkos Tool
- KOKKOS_PROFILE_LIBRARY=/path/to/libcaliper.so
- Configuration
  - Set Caliper environment variables
  - Or use prebaked configs
  - "--kokkos-tools-args=config," or "CALI_CONFIG=config"
  - Generally, add "profile.kokkos" to a config to get Kokkos profiling

David Boehme: "the Caliper man"

Simple timing

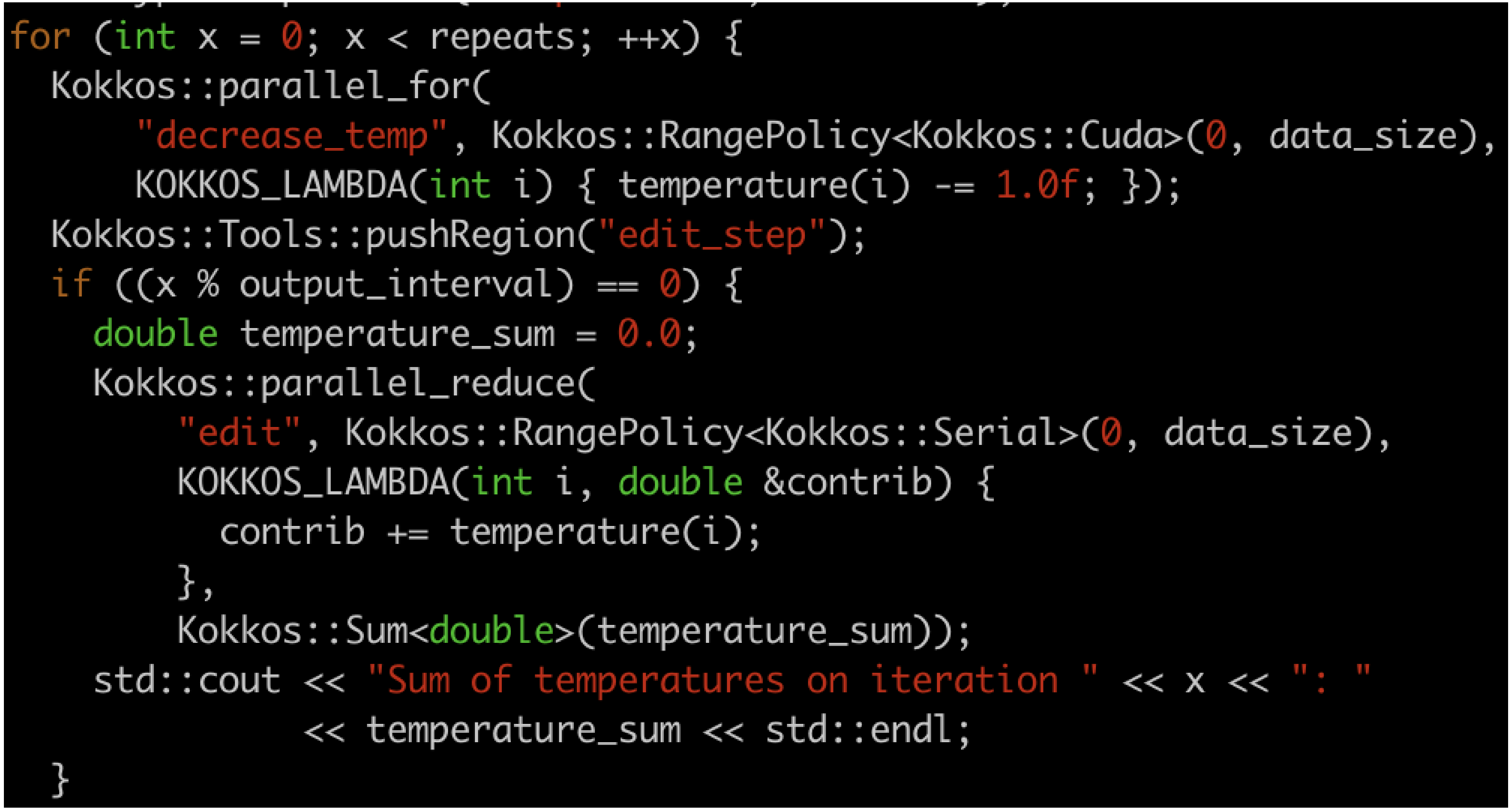


```
(base) [dzpolia@kokkos-dev-2 tool-playground]$ ./bin/instances_begin --kokkos-tools-args="
runtime-report(profile.kokkos)" --kokkos-tools-library=./lib64/libcaliper.so 2>&1 | tee ca
liper_log
Path                                            Time (E) Time (I) Time % (E) Time % (I)
process_temp2                                   4.254843 6.197905   7.960957  11.596493
  Kokkos::Tools::invoke_~~kos Profile Tool Fence 1.943062 1.943062   3.635536   3.635536
Kokkos::deep_copy: copy~~s, post deep copy fence 3.728552 3.728552   6.976248   6.976248
Kokkos::deep_copy: copy~~pre view equality check 3.842599 3.842599   7.189634   7.189634
process_temp1                                   4.281627 6.225261   8.011071  11.647677
  Kokkos::Tools::invoke_~~kos Profile Tool Fence 1.943634 1.943634   3.636606   3.636606
Kokkos::Tools::invoke_k~~kkos Profile Tool Fence 3.590691 3.590691   6.718306   6.718306
Kokkos::CudaInternal::i~~on space initialization 0.000030 0.000030   0.000056   0.000056
```

Okay, so it does the space-time-stack? Why Caliper?

- In addition to simple timings, Caliper supports an unbelievable array of profiling capabilities
  - Often the first place we prototype functionality
- Tech not discussed here
  - SPOT: performance tracking utility, see whether you’re helping or harming the performance of a codebase as you develop it
  - Hatchet: slice and dice your calltrees, calculate which parts of a program are speeding up or slowing down
  - CurIOs: IO profiling
- There are entire Caliper trainings available

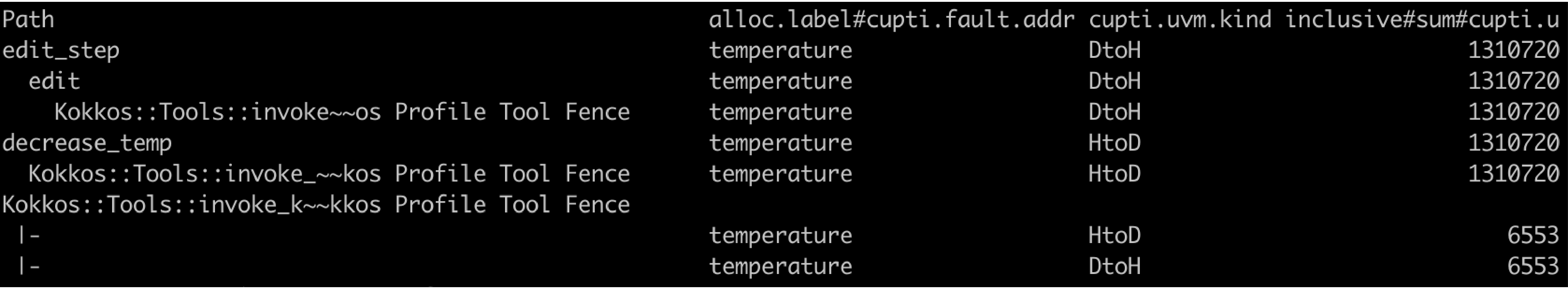
UVM Profiling: a Caliper case study



```
for (int x = 0; x < repeats; ++x) {
  Kokkos::parallel_for(
      "decrease_temp", Kokkos::RangePolicy<Kokkos::Cuda>(0, data_size),
      KOKKOS_LAMBDA(int i) { temperature(i) -= 1.0f; });
  Kokkos::Tools::pushRegion("edit_step");
  if ((x % output_interval) == 0) {
    double temperature_sum = 0.0;
    Kokkos::parallel_reduce(
        "edit", Kokkos::RangePolicy<Kokkos::Serial>(0, data_size),
        KOKKOS_LAMBDA(int i, double &contrib) {
          contrib += temperature(i);
        },
        Kokkos::Sum<double>(temperature_sum));
    std::cout << "Sum of temperatures on iteration " << x << ": "
              << temperature_sum << std::endl;
  }
```

What can we see?

- ./bin/uvm_caliper ./bin/uvm_begin
- “uvm_caliper” just sets environment variables



```
Path                                            alloc.label#cupti.fault.addr cupti.uvm.kind inclusive#sum#cupti.u
edit_step                                       temperature                  DtoH                  1310720
  edit                                          temperature                  DtoH                  1310720
    Kokkos::Tools::invoke~~os Profile Tool Fence temperature                 DtoH                  1310720
decrease_temp                                   temperature                  HtoD                  1310720
  Kokkos::Tools::invoke_~~kos Profile Tool Fence temperature                 HtoD                  1310720
Kokkos::Tools::invoke_k~~kkos Profile Tool Fence
 |-                                             temperature                  HtoD                     6553
 |-                                             temperature                  DtoH                     6553
```

Typical optimization path

- Understand Kokkos Utilization (SpaceTimeStack)
  - Check how much time in kernels
  - Identify HotSpot Kernels
- Run Memory Analysis (MemoryEvents)
  - Are there many allocations/deallocations - 5000/s is OK.
  - Identify temporary allocations which might be able to hoisted
- Identify Serial Code Regions (SpaceTimeStack)
  - Add Profiling Regions
  - Find Regions with low fraction of time spend in Kernels
- Dive into individual Kernels
  - Use connector tools to analyze kernels.
  - E.g. use roof line analysis to find underperforming code.

C++ Compilers ☹

Clang-Tidy with Kokkos Knowledge

What we did:

Augmented ClangTidy, a LLVM/Clang based static analysis tool, with knowledge of Kokkos semantics to detect bugs early in the development cycle.

Current Checks:

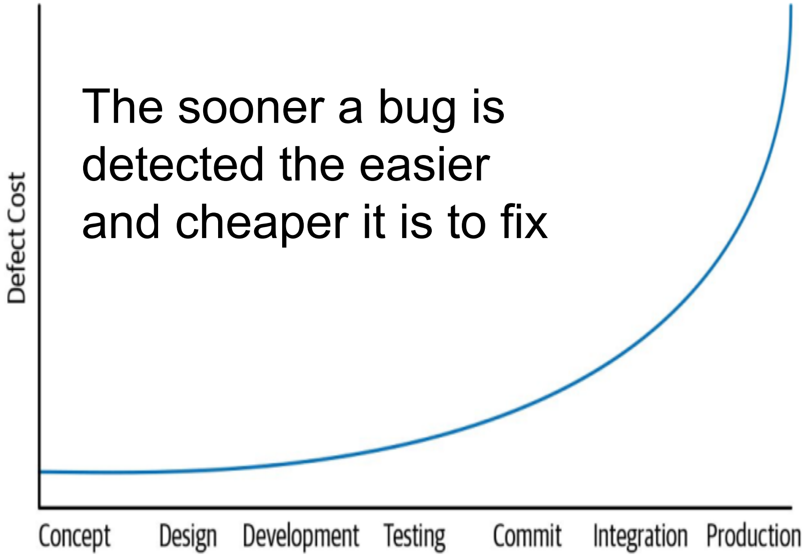
- Implicit this
- Ensure Kokkos function

- Code at: https://github.com/kokkos/llvm-project
- SIAM poster at: poster-link
- Tool tutorial at: youtube-link

Apps teams have been very positive about what compiler based tools can accomplish

EXASCALE COMPUTING PROJECT

Early Bug Detection Saves Time and Money

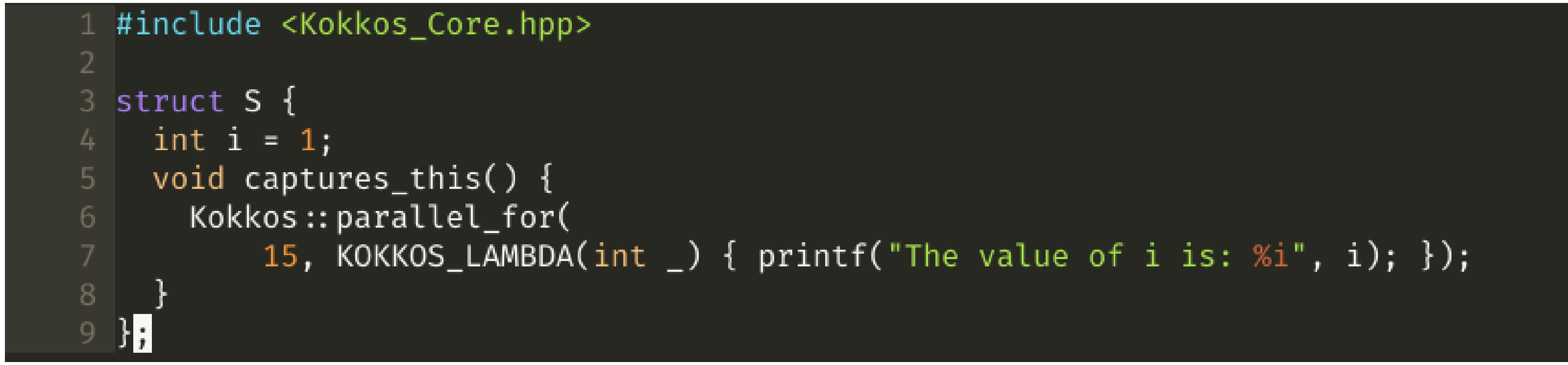


The sooner a bug is detected the easier and cheaper it is to fix

Figure 1-2. Timeline of the developer workflow

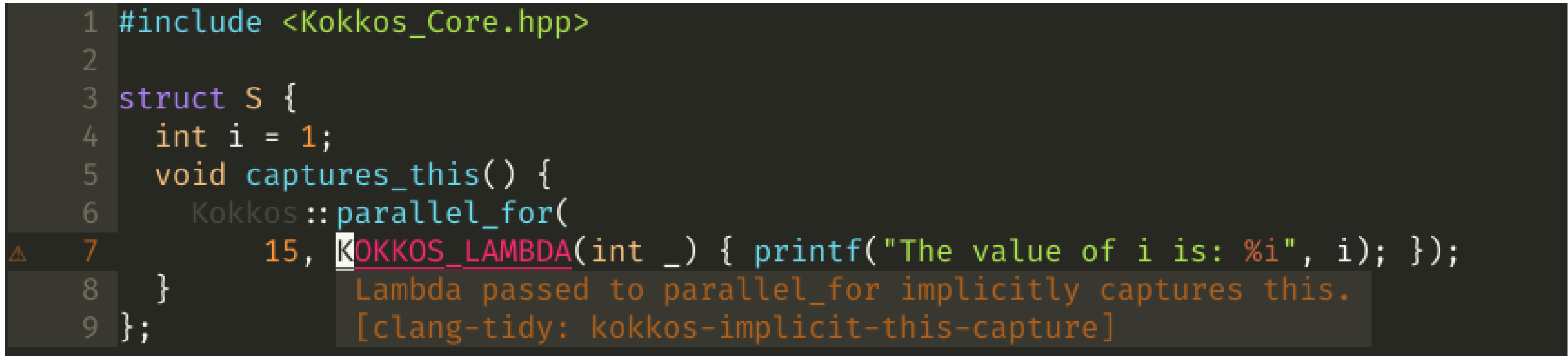
Credit: Software Engineering at Google (p. 35). O'Reilly Media.

Code with an easy to introduce bug



```
#include <Kokkos_Core.hpp>

struct S {
  int i = 1;
  void captures_this() {
    Kokkos::parallel_for(
          15, KOKKOS_LAMBDA(int _) { printf("The value of i is: %i", i); });
  }
};
```

Our compiler tool warning the user about the bug, literally as they type.

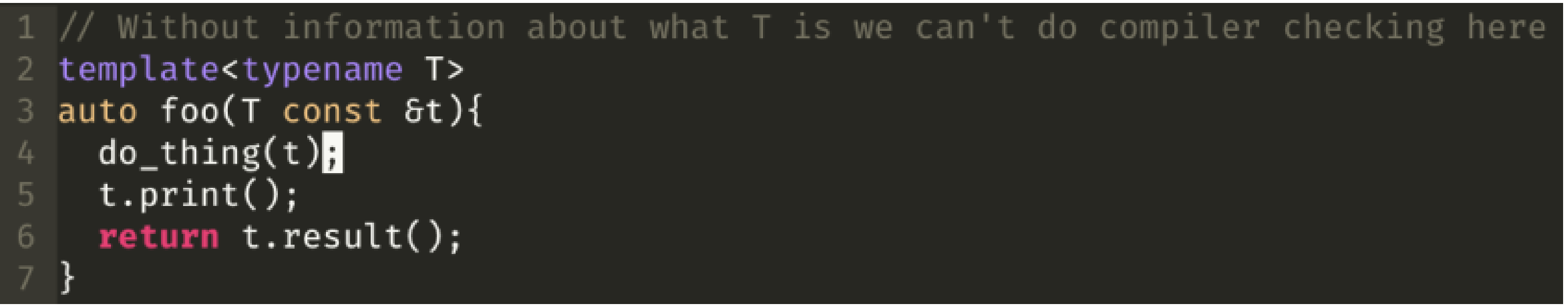


```
#include <Kokkos_Core.hpp>

struct S {
  int i = 1;
  void captures_this() {
    Kokkos::parallel_for(
          15, KOKKOS_LAMBDA(int _) { printf("The value of i is: %i", i); });
  }       Lambda passed to parallel_for implicitly captures this.
};        [clang-tidy: kokkos-implicit-this-capture]
```

We want to detect bugs as early as possible

But generic code is hard (not just for Kokkos)

C++ templates make it very hard to use the compiler to check the validity of generic code.



```
// Without information about what T is we can't do compiler checking here
template<typename T>
auto foo(T const &t){
  do_thing(t);
  t.print();
  return t.result();
}
```

We can check correctness for concrete types,
but generic types like View<T> are hard to impossible to check in real time

We have two possible ideas to solve this problem for Kokkos, which we will investigate in the future:

1. User provides a pragma or hint about what type we should use to check the code with
2. Use C++20 concepts to provide checking for known Kokkos types

Improve C++ tooling for our application teams

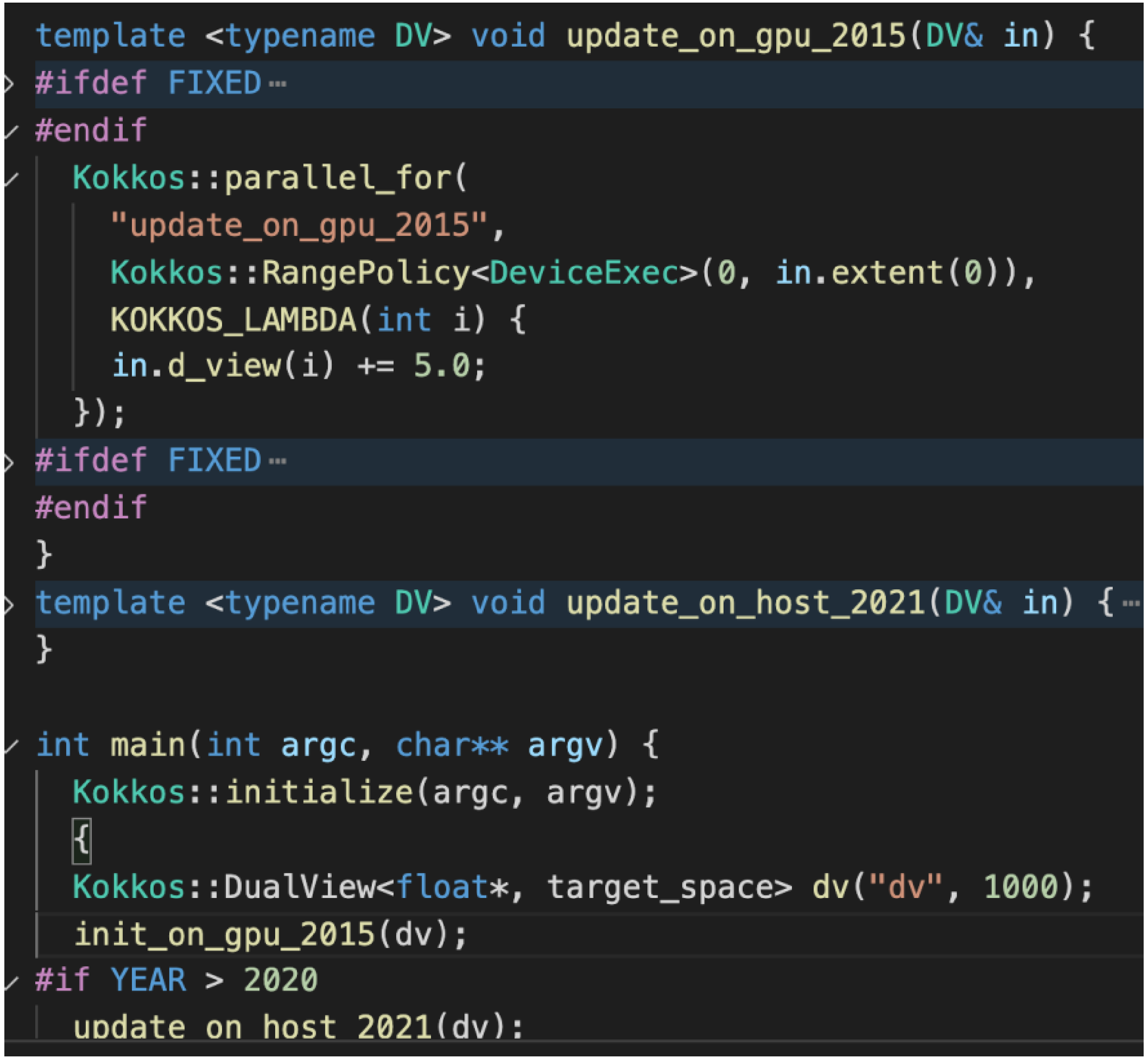
Skylos

EXASCALE COMPUTING PROJECT

“Kokkos Sanitizers” + IDE integration

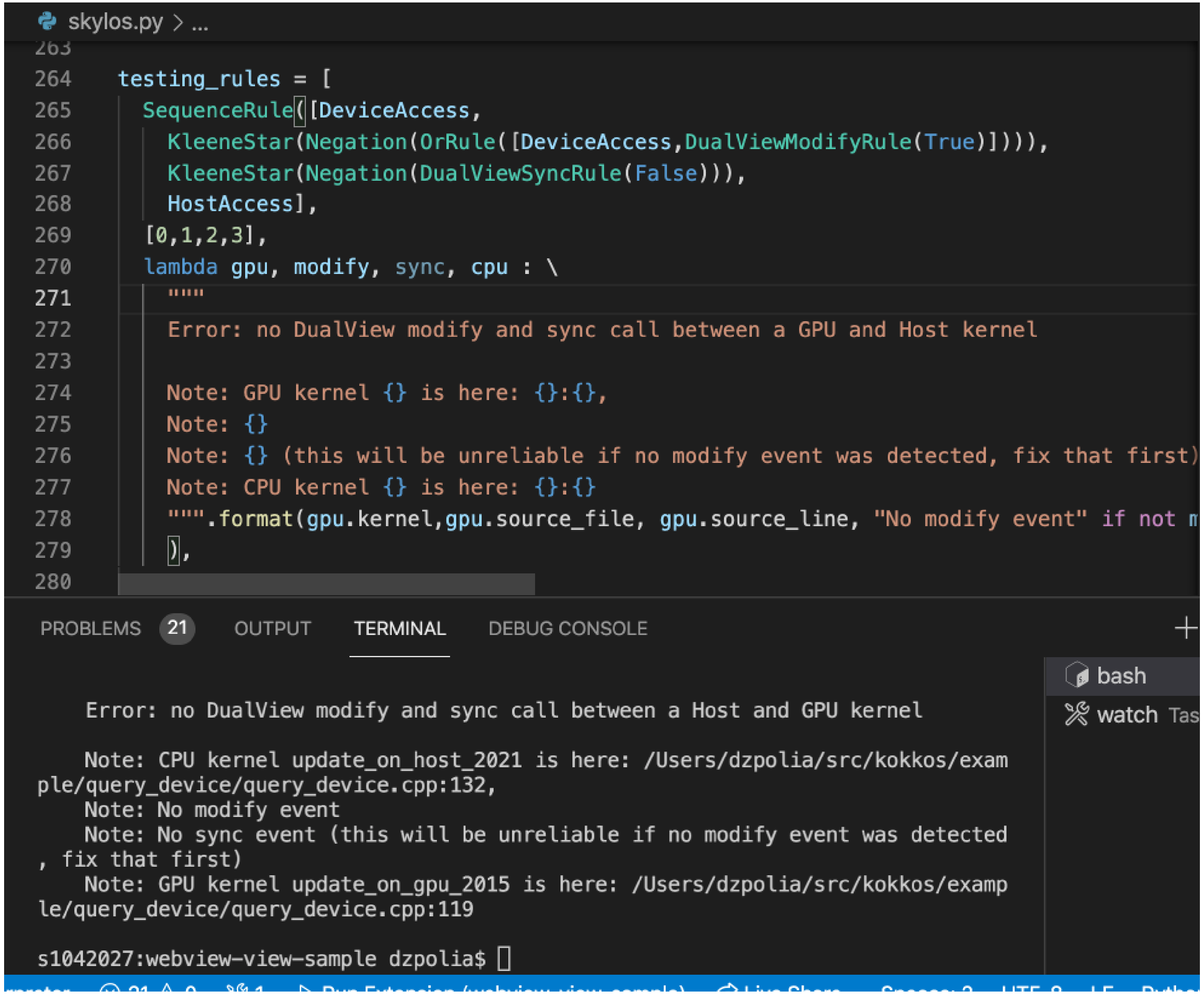
- Kokkos has semantics
  - Semantics that can be violated
- DualView

DualView: the bottomless bit of footguns



```
template <typename DV> void update_on_gpu_2015(DV& in) {
#ifdef FIXED⋯
#endif
  Kokkos::parallel_for(
    "update_on_gpu_2015",
    Kokkos::RangePolicy<DeviceExec>(0, in.extent(0)),
    KOKKOS_LAMBDA(int i) {
    in.d_view(i) += 5.0;
  });
#ifdef FIXED⋯
#endif
}
template <typename DV> void update_on_host_2021(DV& in) {⋯
}

int main(int argc, char** argv) {
  Kokkos::initialize(argc, argv);
  {
  Kokkos::DualView<float*, target_space> dv("dv", 1000);
  init_on_gpu_2015(dv);
#if YEAR > 2020
  update_on_host_2021(dv);
```

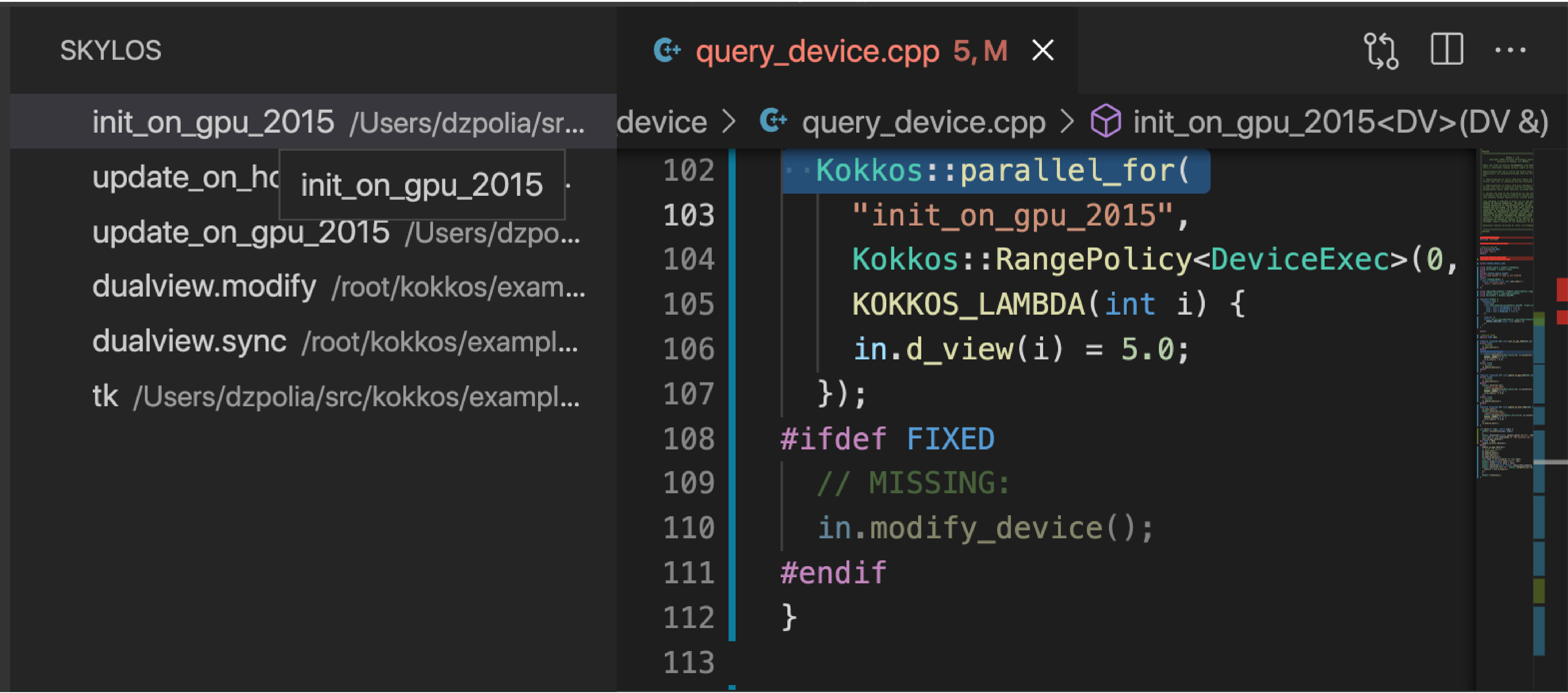
Debugging



```python
testing_rules = [
  SequenceRule([DeviceAccess,
    KleeneStar(Negation(OrRule([DeviceAccess,DualViewModifyRule(True)]))),
    KleeneStar(Negation(DualViewSyncRule(False))),
    HostAccess],
  [0,1,2,3],
  lambda gpu, modify, sync, cpu : \
    """
    Error: no DualView modify and sync call between a GPU and Host kernel

    Note: GPU kernel {} is here: {}:{},
    Note: {}
    Note: {} (this will be unreliable if no modify event was detected, fix that first)
    Note: CPU kernel {} is here: {}:{}
    """.format(gpu.kernel,gpu.source_file, gpu.source_line, "No modify event" if not m
    ),
```

```
    Error: no DualView modify and sync call between a Host and GPU kernel

    Note: CPU kernel update_on_host_2021 is here: /Users/dzpolia/src/kokkos/exam
ple/query_device/query_device.cpp:132,
    Note: No modify event
    Note: No sync event (this will be unreliable if no modify event was detected
, fix that first)
    Note: GPU kernel update_on_gpu_2015 is here: /Users/dzpolia/src/kokkos/examp
le/query_device/query_device.cpp:119

s1042027:webview-view-sample dzpolia$ 
```

IDE Integration



SKYLOS

- init_on_gpu_2015 /Users/dzpolia/sr...
- update_on_h init_on_gpu_2015
- update_on_gpu_2015 /Users/dzpo...
- dualview.modify /root/kokkos/exam...
- dualview.sync /root/kokkos/exampl...
- tk /Users/dzpolia/src/kokkos/exampl...

```
  Kokkos::parallel_for(
    "init_on_gpu_2015",
    Kokkos::RangePolicy<DeviceExec>(0,
    KOKKOS_LAMBDA(int i) {
    in.d_view(i) = 5.0;
  });
#ifdef FIXED
  // MISSING:
  in.modify_device();
#endif
}
```

Questions?

- kokkosteam.slack.com
- dzpolia@sandia.gov
- Kokkos more broadly: crtrott@sandia.gov