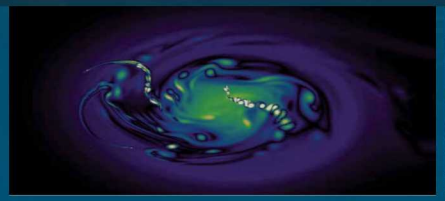
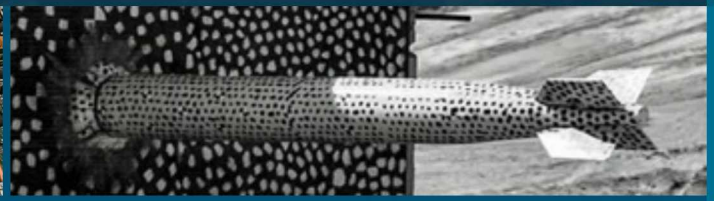
SAND2018-7004C

Sandia National Laboratories

Profiling and Debugging for the Kokkos Programming Model





PRESENTED BY

Si Hammond (sdhammo@sandia.gov)

Collaborators: Christian Trott, Dan Ibanez, Dan Sunderland, Nathan Ellingwood and Carter Edwards

U.S. DEPARTMENT OF ENERGY NNSA

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.

Images courtesy of Sandia National Laboratories and Oak Ridge National Laboratory

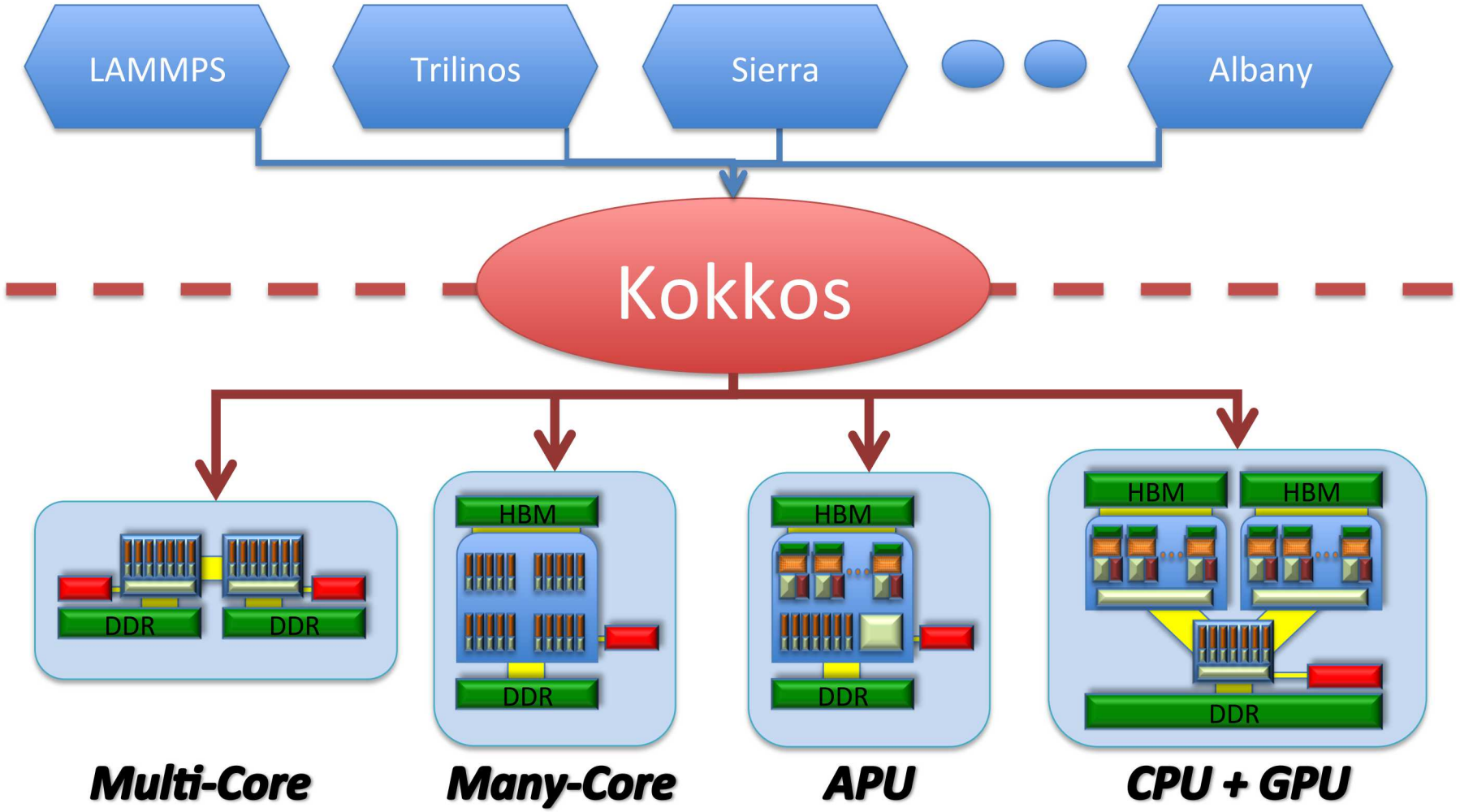
Introduction

Want to talk about how we are trying to bring uniform tools APIs and user experiences across our diverse platforms

- Quick overview of the Kokkos C++ Programming Model
  - Our vision for a future “parallel” C++ standard
- What are the problems with this approach in the tools environment?
- How can we solve that and make this easier/more uniform
- How can you use these interfaces today to help make better parallel programs?

Kokkos Overview

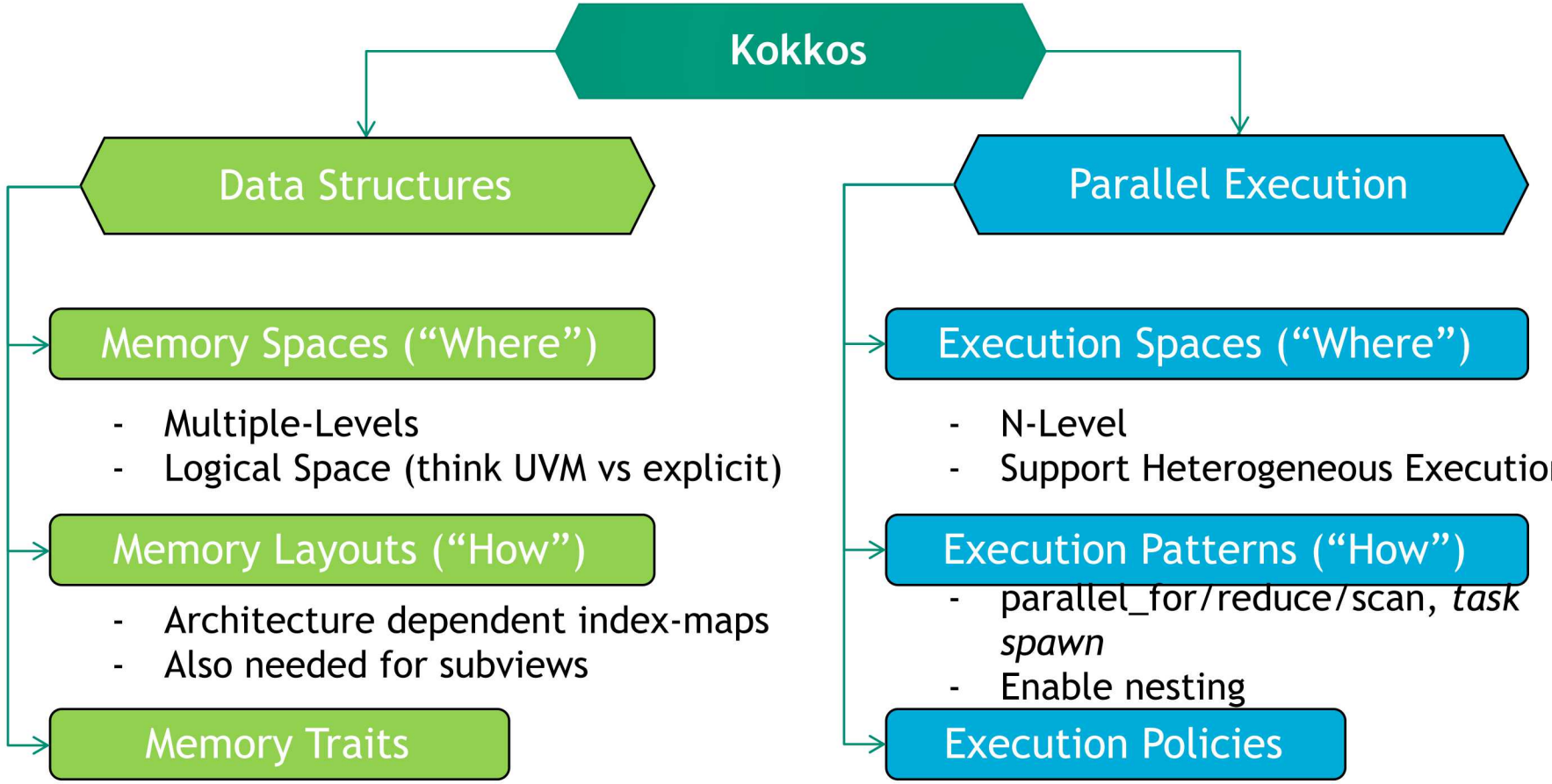
Overview of Kokkos



Kokkos Programming Model

http://github.com/kokkos/kokkos

Separating of Concerns for Future Systems…



Discussion

Kokkos is our vision for what future language based standards might want to expose

- Longer term plan is for Kokkos features to be in C++ standard
- Essentially pieces of Kokkos go away over time as the standard assumes more of a role in delivering these to programs

We want to give users the same experience across platforms/hardware

- A unified programming model
- Similar compilation and software build environments
  - Spend a lot of time thinking about how to make this easier, correct flags etc

Now we also want to have a unified tools and support environment

Experiences Profiling and Debugging when using C++ Parallelism Abstractions

Motivations – Why Profiling C++ Abstractions is Painful

Kokkos makes heavy use of C++ templates/meta-templates to deliver its abstractions

- Permits extremely complex optimization paths within the implementation
- Allows us to remove significant amount of code during compilation
  - Speeds up the final executable
  - Reduces code size (which is significant for large binaries developed at Sandia)

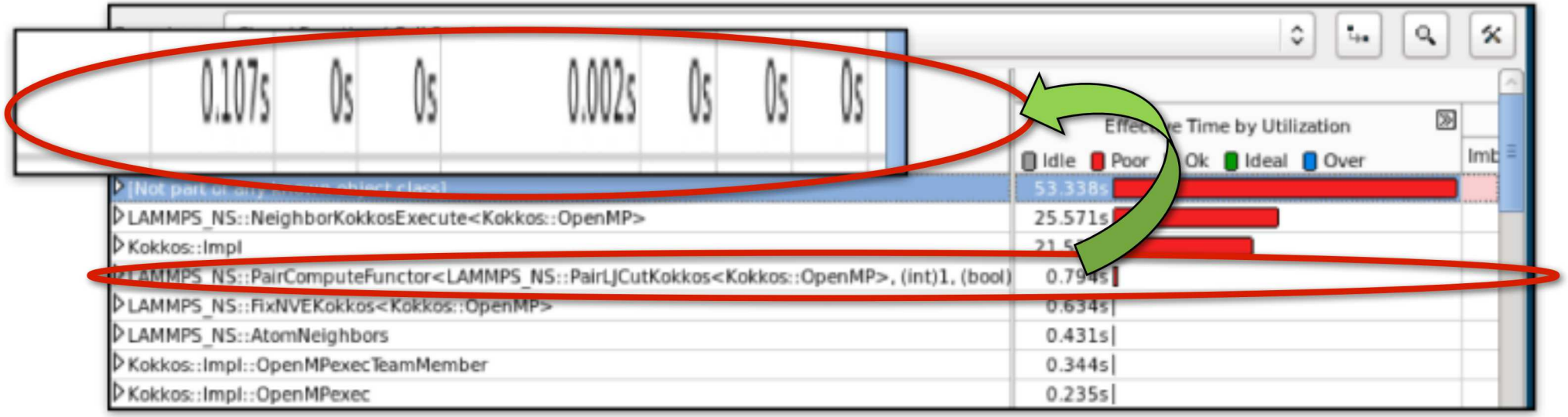
But templating causes some real headaches…

- Complex compilation paths, lots of header/function expansions
- (Extremely) long function names, these have been recorded at >8kB for a single function name
- Breaks expectations of debugging and profiling tools that routinely get used in our HPC ecosystem
- Longer compile times / larger binaries

Example…

See templates appearing in naming for Kokkos functions

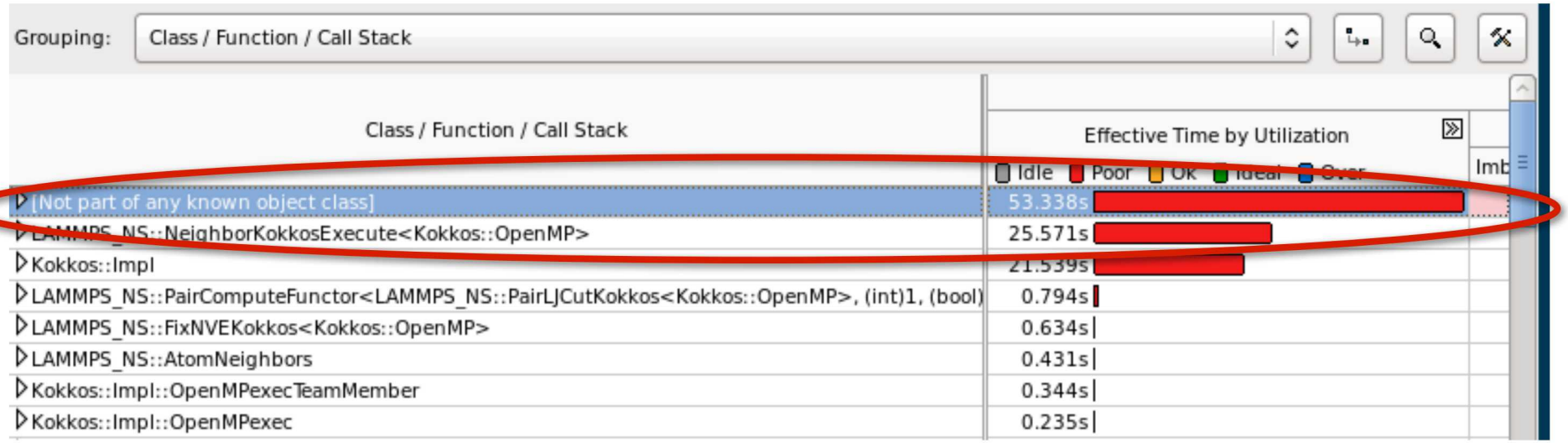
- In some cases this is just use of the final backends (e.g. OpenMP, CUDA etc)
- Other cases, includes use of constants, types etc which are used to decide *how* to compile the programming model for different devices
- Confusing for programmer, unable to relate properly to code, loops etc



Gets Worse with Some Tools…

Significant amount of time is attributed to be in unknown object classes, or in some cases all within a single OpenMP region

- Some tools struggle to handle the C++ lambda conventions used in C++ abstraction layers for loop bodies
- OpenMP expects pragmas to be directly above loop bodies – not above loop bodies which call lambdas
- Results in misattributed performance results



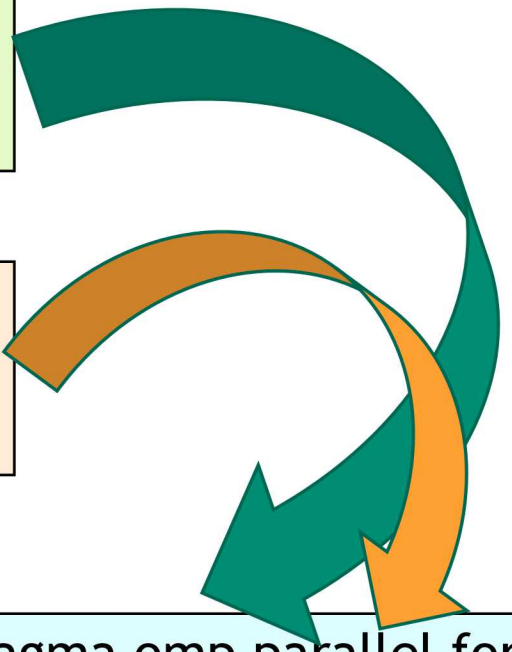
OpenMP in Particular…

User Defined Kernels (in Application)

```
Kokkos:parallel_for( 16, KOKKOS_LAMBDA(const int i) {
    printf("Hello from iteration i\n", i);
});
```

…

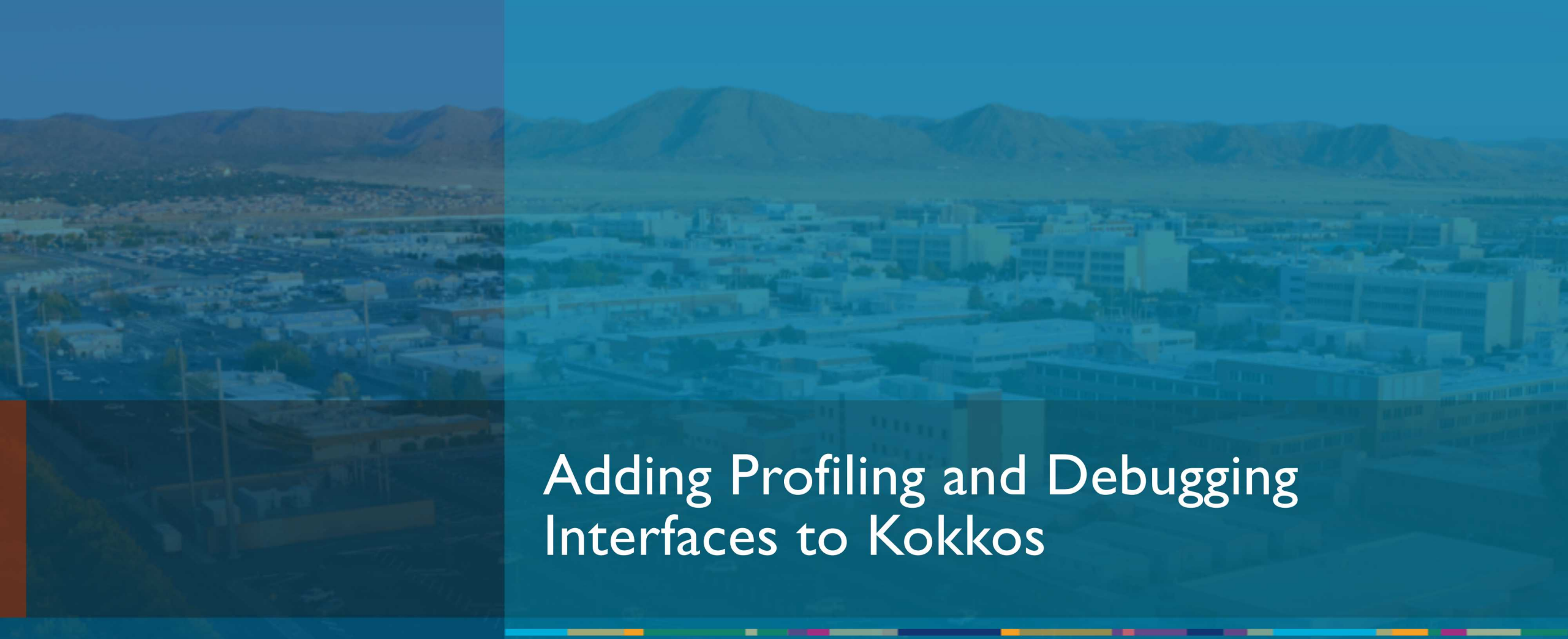
```
Kokkos:parallel_for( 32, KOKKOS_LAMBDA(const int i) {
    printf("Hello Again from iteration i\n", i);
});
```



```
#pragma omp parallel for
for(int kokkosI = 0; kokkosI < N; ++i) {
    // call my lambda (kokkosI);
}
```

OpenMP profilers think all the time is in the same region

Kokkos Runtime/Implementation



Adding Profiling and Debugging Interfaces to Kokkos

Kokkos Code Generation

User Application:

```
void mykernel() {
    ..

    Kokkos::parallel_for (N, KOKKOS_LAMBDA (const int i) {
      printf ("Hello from i = %i\n", i);
    });

    ..
}
```

C++ Compiler Magic Goes On Here

Kokkos Runtime:

```
Kokkos::parallel_for (..) {
    CALL_PROFILE_HOOK_START

    PARALLEL BODY

    CALL_PROFILE_HOOK_END
}
```

Backend Code Generation (i.e. OpenMP, CUDA etc is really done here)

In the usual cause of events, these hooks default to NULL and are not executed

Kokkos Profiling Hooks

In the default configurations, profiling hooks are always compiled into applications

- Means no need to recompile applications to get additional profiling information
  - This is a big deal for our applications, usually when we recompile we lose some of the artifacts we wanted to examine
- Can work without debug information because Kokkos supplies kernel naming

Dynamically load tools into the application at runtime

- Very flexible load mechanism
- Stack tools together – can choose how multiple tools running together interact

Extremely easy to decide what events you want to subscribe to as a tool writer

Kokkos Profiling Interface at Runtime

User Application:

```
void mykernel() {
    ..

    Kokkos::parallel_for (N, KOKKOS_LAMBDA (const int i) {
      printf ("Hello from i = %i\n", i);
    });

    ..
}
```

C++ Compiler Magic Goes On Here

Kokkos Runtime:

```
START_KERNEL_PROFILER
(e.g. START_TIMER)

END_KERNEL_PROFILER
(e.g. END_TIMER)
```

```
Kokkos::parallel_for (..) {
    CALL_PROFILE_HOOK_START

    PARALLEL BODY

    CALL_PROFILE_HOOK_END
}
```

Dynamically loaded/linked at runtime

Kokkos Profiling Interface for Vendor Tools

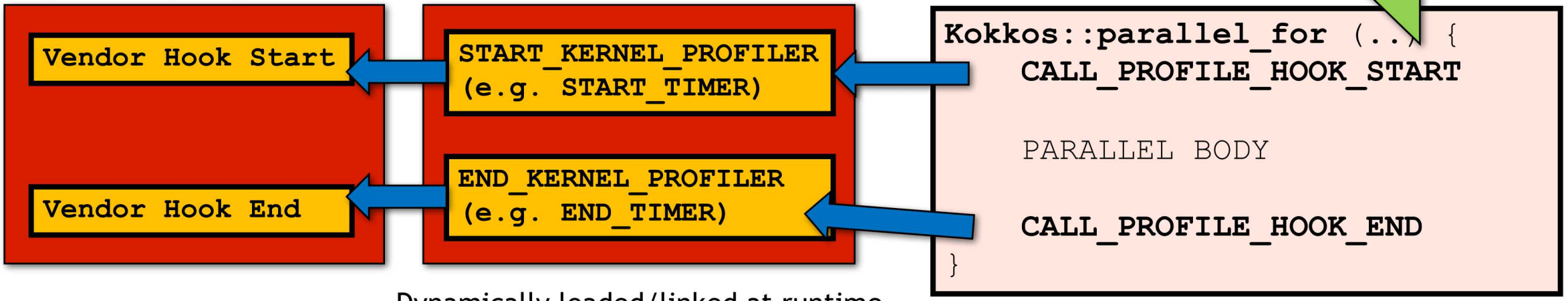
User Application:

```
void mykernel() {
    ..

    Kokkos::parallel_for (N, KOKKOS_LAMBDA (const int i) {
      printf ("Hello from i = %i\n", i);
    });

    ..
}
```

C++ Compiler Magic Goes On Here

e.g. VTune, NSight

Kokkos Runtime:

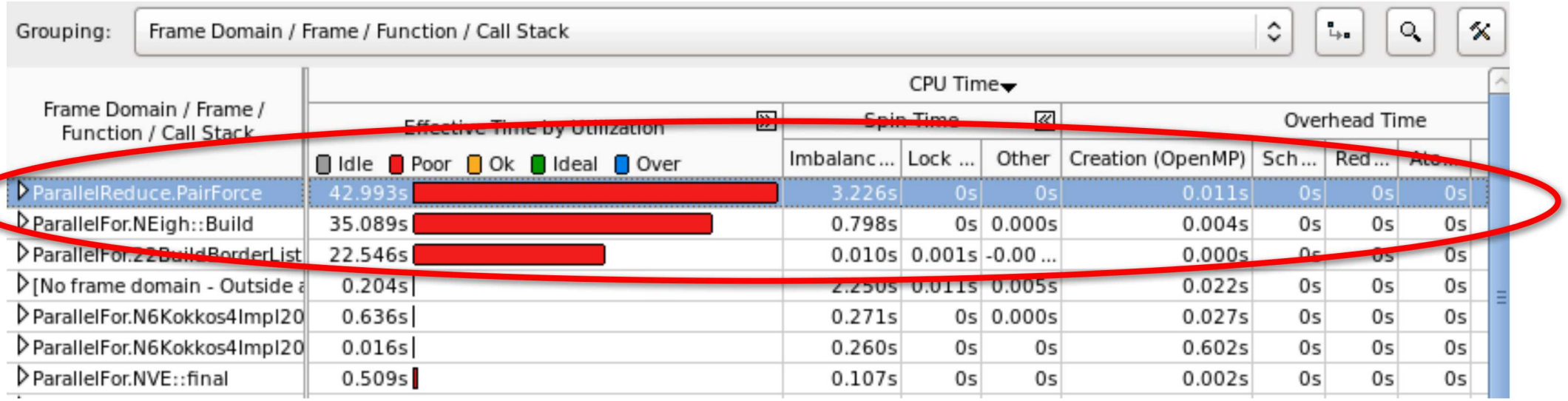


Dynamically loaded/linked at runtime

Example using Kokkos Connectors

Using the KokkosP-VTune Connector we can annotate parallel execution regions, code segments (“regions”), tasks etc

- Enhance output with kernel pattern types (e.g. parallel-for, reduce, scan etc)
- Provide user-defined kernel naming
- Enhance timeline output (specific kernel naming in timeline)
- Significantly improved user experience and much faster for analysis



More Than Parallel Kernels

In more recent versions of Kokkos we have added extra support (no longer just parallel kernel dispatch):

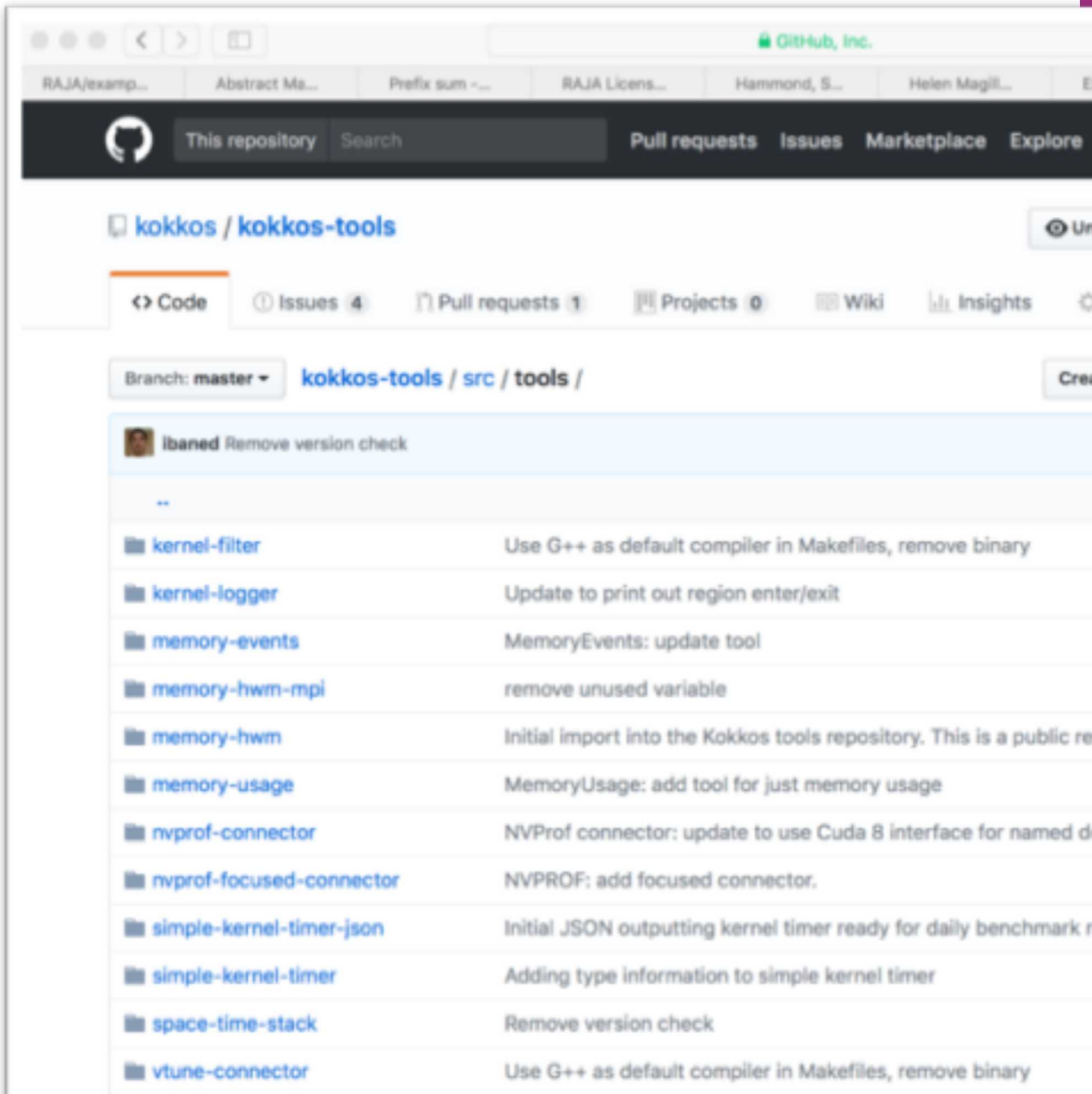
- User Defined Regions (stack-like behavior, push/pop), mainly for entry/exit into/out of libraries
- User Defined Sections (arbitrary start/stop calls, gives unstructured tracking of code bodies)
- Tracking of Kokkos allocated data structures (Kokkos malloc, Views, Containers)

Ability to track data structures has been particularly useful for tracking memory consumption prior/after kernel execution

- Useful for memory leaks in parallel programs
- Tracking data copying into/out of accelerators
- Understanding behavior of hardware assistance, e.g. NVLINK data movements
- Opens up additional tool connectivity (e.g. Intel Inspector, custom Valgrind..)

Kokkos Tools Suite

http://github.com/kokkos/kokkos-tools



Kokkos also has its own suite of lightweight tools which work across our supported platforms

- Not *every* platform we target Kokkos to has great performance tools
- Still want to provide a capability that has the same *feel* across environments

Important to demonstrate some of our concepts to third parties and provide basic guarantees to programmers across machines

Examples In Action … Data Structure Tracking

User Supplied Allocation Names

Data Structure Tracking

- Get allocation addresses
- Allocation sizes (bytes)
- Where data gets allocated
  - Host?
  - Device? Which Device?
- User supplies name for allocation/View

Allows us to develop data structure life time views

- Compare across MPI ranks

# Time	Ptr	Size	MemSpace	Op	Name
0.002958	0x7f5c9dd030a0	8998912	Host	Allocate	Vector
0.004280	0x7f5c9d46d0a0	8998912	Host	Allocate	Vector
0.005055	0x7f5c9cbd70a0	8998912	Host	Allocate	Vector
2.008489	0x2c60470	4499464	Host	Allocate	MatrixRowOffsets
2.009254	0x7f5c40b0f0a0	119164000	Host	Allocate	MatrixLocalIndicies
2.017489	0x7f5c327c50a0	238328000	Host	Allocate	MatrixValues
4.638244	0x7f5c7f76a0a0	8998912	Host	Allocate	no-label
4.832563	0x373b0d0	562440	Host	Allocate	MatrixRowOffsets
4.832609	0x7f5c7c72e0a0	14609056	Host	Allocate	MatrixLocalIndicies
4.833970	0x7f5c6a4220a0	29218112	Host	Allocate	MatrixValues
5.154044	0x3a12ed0	1124864	Host	Allocate	Vector
5.154236	0x3b25970	1124864	Host	Allocate	Vector
5.154358	0x3c38410	8998912	Host	Allocate	Vector
5.155568	0x44cd4b0	1124864	Host	Allocate	no-label
5.178790	0x4fa7bf0	70312	Host	Allocate	MatrixRowOffsets
5.178849	0x4fba010	1755904	Host	Allocate	MatrixLocalIndicies
5.179045	0x5166bb0	3511808	Host	Allocate	MatrixValues
5.217818	0x4646f40	140608	Host	Allocate	Vector
5.218140	0x4669520	140608	Host	Allocate	Vector
5.218422	0x468bb00	1124864	Host	Allocate	Vector
5.220563	0x479e5a0	140608	Host	Allocate	no-label
5.223459	0x4903620	8792	Host	Allocate	MatrixRowOffsets
5.223489	0x49069f0	202616	Host	Allocate	MatrixLocalIndicies

Examples In Action … Kernel Breakdown

Track timing and call counts for each kernel

- Reporting using user-supplied kernel names
- Analyze how much time is spent inside/outside of Kokkos
- Regions, Sections, Kernels

```
                                                  Reference: ComputeSPMV (Region)   0.83942   205   0.00409   7.219   2.422
                                          Main: Validation Testing Phase (Region)   0.73603     1   0.73603   6.330   2.123
                                                Main: Reference SpMV+MG (Region)   0.31914     1   0.31914   2.744   0.921
                                           Optimized: ComputeDotProduct (Region)   0.09877   474   0.00021   0.849   0.285
                                               Optimized: ComputeWAXPBY (Region)   0.08671   468   0.00019   0.746   0.250
                                           Reference: ComputeDotProduct (Region)   0.04568   151   0.00030   0.393   0.132
                                      Main: Reference CG::OptimizeProblem (Region)   0.03189     1   0.03189   0.274   0.092
                                           Reference: ComputeWAXPBY (Region)   0.03050   150   0.00020   0.262   0.088
                                                      Main: Clean up (Region)   0.01020     1   0.01020   0.088   0.029
                                                Main: Report Results (Region)   0.00265     1   0.00265   0.023   0.008
                                                     Main: TestNorms (Region)   0.00000     1   0.00000   0.000   0.000
```

```
  N11KokkosGraph19GraphColoringHandleIKiS1_S1_N6Kokkos6OpenMPENS2_9HostSpaceES4_E16ReduceMaxFunctorE (ParRed)   0.00013   4   0.00003   0.001   0.000
                                                  KokkosKernels::Impl::StridedCopy (ParFor)   0.00002   4   0.00000   0.000   0.000

----------------------------------------------------------------------
Summary:

Total Execution Time (incl. Kokkos + Non-Kokkos:          34.66302 seconds
Total Time in Kokkos kernels:                             11.62839 seconds
   -> Time outside Kokkos kernels:                        23.03464 seconds
   -> Percentage in Kokkos kernels:                          33.55 %
Total Calls to Kokkos Kernels:                               31732
```

Conclusions and Discussion

Conclusions

Kokkos is our vision for a future parallel-enabled C++

- We recognize this is part of the future of HPC, needs to integrate into the broader community and work with other ideas (RAJA, Thrust, Agency, C++-AMP, etc)

Use of heavily templated C++ and some targets make this particularly difficult to profile and debug

- Templating interferes with traditional tool expectations
- Reduces insight for application developers at a time when they need *more* information

Bringing a unified collection of tool APIs and interfaces to our programming model and runtime

- Provide the same user experience across platforms

Where We Want to Use This Every Day…

Jenkins Job

User Test/Benchmark:

```
void mykernel() {
    ..

    Kokkos::parallel_for (N, KOKKOS_LAMBDA (const int i) {
      printf ("Hello from i = %i\n", i);
    });

    ..
}
```

C++ Compiler Magic Goes On Here

Kokkos Runtime:

```
Kokkos::parallel_for (..) {
    CALL_PROFILE_HOOK_START

    PARALLEL BODY

    CALL_PROFILE_HOOK_END
}
```

START_KERNEL_PROFILER (e.g. START_TIMER)

END_KERNEL_PROFILER (e.g. END_TIMER)

Dynamically loaded/linked at runtime

LDMS

Site-Wide Database


Sandia
National
Laboratories