The Kokkos Lectures

Module 7: Kokkos Tools

June 17, 2024

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series:
  - Slides, recording and Q&A for the Lectures
- https://kokkos.github.io/kokkos-core-wiki:
  - Wiki including API reference
- https://github.com/kokkos/kokkos-tools/wiki:
  - Kokkos Tools Wiki
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

Lecture Series Outline

- 07/17 Module 1: Introduction, Building and Parallel Dispatch
- 07/24 Module 2: Views and Spaces
- 07/31 Module 3: Data Structures + MultiDimensional Loops
- 08/07 Module 4: Hierarchical Parallelism
- 08/14 Module 5: Tasking, Streams and SIMD
- 08/21 Module 6: Internode: MPI and PGAS
- **08/28 Module 7: Tools: Profiling, Tuning and Debugging**
- 09/04 Module 8: Kernels: Sparse and Dense Linear Algebra
- 09/11 Reserve Day

Module 6 Summary

Simple MPI and Kokkos Interaction is easy!

- Simply pass `data()` of a View to MPI functions plus its size.
  - But it better be a contiguous View!
- Initialize Kokkos after MPI, and finalize it before MPI

Overlapping communication and computation possible

- Use Execution Space instances to overlap packing/unpacking with other computation.
- Order operations to maximize overlapping potential.

Fortran Language Compatibility Layer

- Initialize Kokkos from Fortran via `kokkos_initialize` and `kokkos_finalize`
- `nd_array_t` is a representation of a `Kokkos::View`
- Create `nd_array_t` from a Fortran array via `to_nd_array`
- Allocate `Kokkos::DualView` in Fortran with `kokkos_allocate_dualview`

The Python Interop

- Initialize and Finalize Kokkos from Python
- Create Views from Python
- Alias Kokkos Views with NumPy arrays
- **This is in pre-release: ask us for access.**

Module 7: Learning objectives

Simple Tools Usage

- How to dynamically load a Kokkos Tool.
- Simple Profiling and Debugging.
- Leveraging the KokkosP instrumentation for third party tools.

Kokkos Tuning

- Learn to auto-tune runtime parameters.

Build Your Own Tool

- Learn how to build your own tools.

Leveraging Static Analysis

- How to use Kokkos' LLVM tools for static analysis.

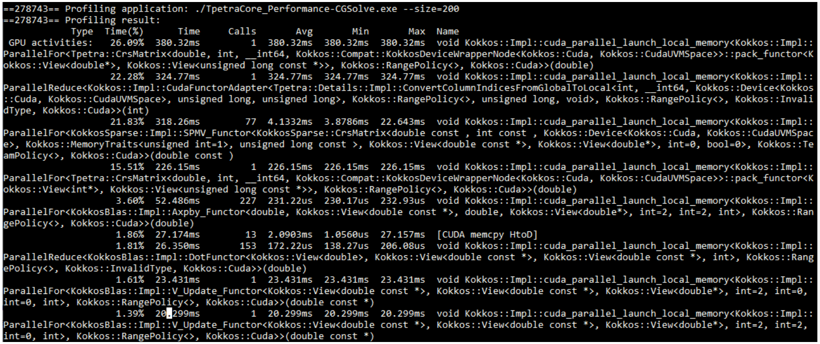
Kokkos Tools

Leveraging Kokkos' built-in instrumentation.

Learning objectives:

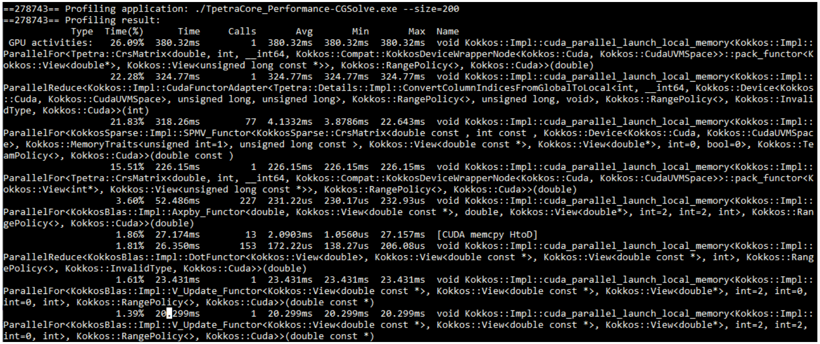
- The need for Kokkos-aware tools.
- How instrumentation helps.
- Simple profiling tools.
- Simple debugging tools.

Output from NVIDIA NVProf for Trilinos Tpetra



```
==278743== Profiling application: ./TpetraCore_Performance-CGSolve.exe --size=200
==278743== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   26.09%  380.32ms         1  380.32ms  380.32ms  380.32ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<double*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                   22.28%  324.77ms         1  324.77ms  324.77ms  324.77ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal<int, __int64, Kokkos::Device<Kokkos
::Cuda, Kokkos::CudaUVMSpace>, unsigned long, unsigned long>, Kokkos::RangePolicy<>, unsigned long, void>, Kokkos::RangePolicy<>, Kokkos::Invali
dType, Kokkos::Cuda>>(int)
                   21.83%  318.26ms        77  4.1332ms  3.8786ms  22.643ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosSparse::Impl::SPMV_Functor<KokkosSparse::CrsMatrix<double const , int const , Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpac
e>, Kokkos::MemoryTraits<unsigned int=1>, unsigned long const >, Kokkos::View<double const *>, Kokkos::View<double*>, int=0, bool=0>, Kokkos::Te
amPolicy<>, Kokkos::Cuda>>(double const )
                   15.51%  226.15ms         1  226.15ms  226.15ms  226.15ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<int*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                    3.60%  52.486ms       227  231.22us  230.17us  232.93us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::Axpby_Functor<double, Kokkos::View<double const *>, double, Kokkos::View<double*>, int=2, int=2, int>, Kokkos::Ran
gePolicy<>, Kokkos::Cuda>>(double)
                    1.86%  27.174ms        13  2.0903ms  1.0560us  27.157ms  [CUDA memcpy HtoD]
                    1.81%  26.350ms       153  172.22us  138.27us  206.08us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<KokkosBlas::Impl::DotFunctor<Kokkos::View<double>, Kokkos::View<double const *>, Kokkos::View<double const *>, int>, Kokkos::Rang
ePolicy<>, Kokkos::InvalidType, Kokkos::Cuda>>(double)
                    1.61%  23.431ms         1  23.431ms  23.431ms  23.431ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=0,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
                    1.39%  20.299ms         1  20.299ms  20.299ms  20.299ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=2,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
```

Output from NVIDIA NVProf for Trilinos Tpetra



```
==278743== Profiling application: ./TpetraCore_Performance-CGSolve.exe --size=200
==278743== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   26.09%  380.32ms         1  380.32ms  380.32ms  380.32ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<double*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                   22.28%  324.77ms         1  324.77ms  324.77ms  324.77ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal<int, __int64, Kokkos::Device<Kokkos
::Cuda, Kokkos::CudaUVMSpace>, unsigned long, unsigned long>, Kokkos::RangePolicy<>, unsigned long, void>, Kokkos::RangePolicy<>, Kokkos::Invali
dType, Kokkos::Cuda>>(int)
                   21.83%  318.26ms        77  4.1332ms  3.8786ms  22.643ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosSparse::Impl::SPMV_Functor<KokkosSparse::CrsMatrix<double const , int const , Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpac
e>, Kokkos::MemoryTraits<unsigned int=1>, unsigned long const >, Kokkos::View<double const *>, Kokkos::View<double*>, int=0, bool=0>, Kokkos::Te
amPolicy<>, Kokkos::Cuda>>(double const )
                   15.51%  226.15ms         1  226.15ms  226.15ms  226.15ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrsMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>>::pack_functor<K
okkos::View<int*>, Kokkos::View<unsigned long const *>>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
                    3.60%  52.486ms       227  231.22us  230.17us  232.93us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::Axpby_Functor<double, Kokkos::View<double const *>, double, Kokkos::View<double*>, int=2, int=2, int>, Kokkos::Ran
gePolicy<>, Kokkos::Cuda>>(double)
                    1.86%  27.174ms        13  2.0903ms  1.0560us  27.157ms  [CUDA memcpy HtoD]
                    1.81%  26.350ms       153  172.22us  138.27us  206.08us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<KokkosBlas::Impl::DotFunctor<Kokkos::View<double>, Kokkos::View<double const *>, Kokkos::View<double const *>, int>, Kokkos::Rang
ePolicy<>, Kokkos::InvalidType, Kokkos::Cuda>>(double)
                    1.61%  23.431ms         1  23.431ms  23.431ms  23.431ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=0,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
                    1.39%  20.299ms         1  20.299ms  20.299ms  20.299ms  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double const *>, Kokkos::View<double const *>, Kokkos::View<double*>, int=2, int=2,
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
```

What are those Kernels doing?

Generic code obscures what is happening from the tools
Historically a lot of profiling tools are coming from a Fortran and C world:

- Focused on functions and variables
- C++ has a lot of other concepts:
  - Classes with member functions
  - Inheritance
  - Template Metaprogramming
- Abstraction Models (Generic Programming) obscure things
  - From a profiler perspective interesting stuff happens in the abstraction layer (e.g. `#pragma omp parallel`)
  - Symbol names get really complex due to deep template layers

Instrumentation enables context information to reach tools.

Most profiling tools have an instrumentation interface

- E.g. nvtx for NVIDIA, ITT for Intel.
- Allows to name regions
- Sometimes can mark up memory operations.

Instrumentation enables context information to reach tools.

Most profiling tools have an instrumentation interface

- E.g. nvtx for NVIDIA, ITT for Intel.
- Allows to name regions
- Sometimes can mark up memory operations.

KokkosP

Kokkos has its own instrumentation interface KokkosP, which can be used to write tools.

- Knows about parallel dispatch
- Knows about allocations, deallocations and deep_copy
- Provides region markers
- Leverages naming information (kernels, Views)

There are two components to Kokkos Tools: the KokkosP instrumentation interface and the actual Tools.

KokkosP Interface

- The internal instrumentation layer of Kokkos.
- Always available even in release builds.
- Zero overhead if no tool is loaded.

Kokkos Tools

- Tools leveraging the KokkosP instrumentation layer.
- Are loaded at runtime by Kokkos.
  - Set `KOKKOS_TOOLS_LIBS` environment variable to load a shared library.
  - Compile tools into the executable and use the API callback setting mechanism.

How does it Work

Download tools from
https://github.com/kokkos/kokkos-tools

- Tools are largely independent of the Kokkos configuration
  - May need to use the same C++ standard library.
  - Use the same tool for CUDA and OpenMP code for example.
- We recommend you build the tools with CMake

```
cd kokkos-tools && cmake -B build
cmake --build build --parallel 4
cmake --install build --prefix /where/to/install/the/tools
```

Loading Tools:

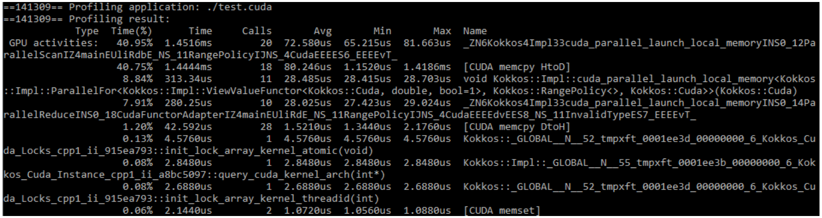
- Set `KOKKOS_TOOLS_LIBS` environment variable to the full path to the shared library of the tool.
- Kokkos dynamically loads symbols from the library during `initialize` and fills function pointers.
- If no tool is loaded the overhead is a function pointer comparison to `nullptr`.

An Example Code

```
View<double*> a("A",N);
View<double*, HostSpace> h_a = create_mirror_view(a);

Profiling::pushRegion("Setup");
parallel_for("Init_A",RangePolicy<h_exec_t>(0,N),
  KOKKOS_LAMBDA(int i) { h_a(i) = i; });
deep_copy(a,h_a);
Profiling::popRegion();

Profiling::pushRegion("Iterate");
for(int r=0; r<10; r++) {
  View<double*> tmp("Tmp",N);
  parallel_scan("K_1",RangePolicy<exec_t>(0,N),
    KOKKOS_LAMBDA(int i, double& lsum, bool f) {
      if(f) tmp(i) = lsum;
      lsum += a(i);
  });
  double sum;
  parallel_reduce("K_2",N, KOKKOS_LAMBDA(int i, double& lsum) {
    lsum += tmp(i);
  },sum);
}
Profiling::popRegion();
```

An Example Code: Nvprof

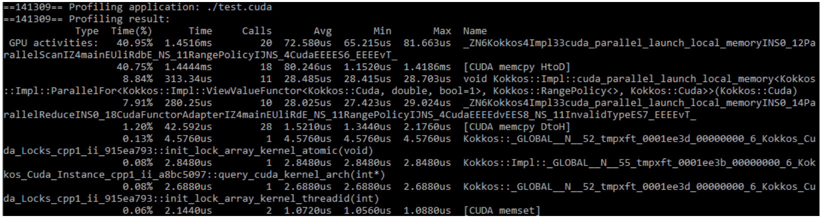
Output of: `nvprof ./test.cuda`



```
==141309== Profiling application: ./test.cuda
==141309== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   40.95%  1.4516ms        20  72.580us  65.215us  81.663us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_12Pa
rallelScanIZ4mainEUliRdbE_NS_11RangePolicyIJNS_4CudaEEEES6_EEEEvT_
                   40.75%  1.4444ms        18  80.246us  1.1520us  1.4186ms  [CUDA memcpy HtoD]
                    8.84%  313.34us        11  28.485us  28.415us  28.703us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos
::Impl::ParallelFor<Kokkos::Impl::ViewValueFunctor<Kokkos::Cuda, double, bool=1>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(Kokkos::Cuda)
                    7.91%  280.25us        10  28.025us  27.423us  29.024us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_14Pa
rallelReduceINS0_18CudaFunctorAdapterIZ4mainEUliRdE_NS_11RangePolicyIJNS_4CudaEEEEdvEES8_NS_11InvalidTypeES7_EEEEvT_
                    1.20%  42.592us        28  1.5210us  1.3440us  2.1760us  [CUDA memcpy DtoH]
                    0.13%  4.5760us         1  4.5760us  4.5760us  4.5760us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_atomic(void)
                    0.08%  2.8480us         1  2.8480us  2.8480us  2.8480us  Kokkos::Impl::_GLOBAL__N__55_tmpxft_0001ee3b_00000000_6_Kok
kos_Cuda_Instance_cpp1_ii_a8bc5097::query_cuda_kernel_arch(int*)
                    0.08%  2.6880us         1  2.6880us  2.6880us  2.6880us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_threadid(int)
                    0.06%  2.1440us         2  1.0720us  1.0560us  1.0880us  [CUDA memset]
```

An Example Code: Nvprof

Output of: `nvprof ./test.cuda`



```
==141309== Profiling application: ./test.cuda
==141309== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   40.95%  1.4516ms        20  72.580us  65.215us  81.663us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_12Pa
rallelScanIZ4mainEUliRdbE_NS_11RangePolicyIJNS_4CudaEEEES6_EEEEvT_
                   40.75%  1.4444ms        18  80.246us  1.1520us  1.4186ms  [CUDA memcpy HtoD]
                    8.84%  313.34us        11  28.485us  28.415us  28.703us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos
::Impl::ParallelFor<Kokkos::Impl::ViewValueFunctor<Kokkos::Cuda, double, bool=1>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(Kokkos::Cuda)
                    7.91%  280.25us        10  28.025us  27.423us  29.024us  _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_14Pa
rallelReduceINS0_18CudaFunctorAdapterIZ4mainEUliRdE_NS_11RangePolicyIJNS_4CudaEEEEdvEES8_NS_11InvalidTypeES7_EEEEvT_
                    1.20%  42.592us        28  1.5210us  1.3440us  2.1760us  [CUDA memcpy DtoH]
                    0.13%  4.5760us         1  4.5760us  4.5760us  4.5760us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_atomic(void)
                    0.08%  2.8480us         1  2.8480us  2.8480us  2.8480us  Kokkos::Impl::_GLOBAL__N__55_tmpxft_0001ee3b_00000000_6_Kok
kos_Cuda_Instance_cpp1_ii_a8bc5097::query_cuda_kernel_arch(int*)
                    0.08%  2.6880us         1  2.6880us  2.6880us  2.6880us  Kokkos::_GLOBAL__N__52_tmpxft_0001ee3d_00000000_6_Kokkos_Cu
da_Locks_cpp1_ii_915ea793::init_lock_array_kernel_threadid(int)
                    0.06%  2.1440us         2  1.0720us  1.0560us  1.0880us  [CUDA memset]
```

Let us make one larger:

```
_ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0
_14ParallelReduceINS0_18CudaFunctorAdapterIZ4mainEUliRdE
_NS_11RangePolicyIJNS_4CudaEEEEdvEES8_NS_11InvalidTypeES7_EEEEvT_
```

And demangled:

```
void Kokkos::Impl::cuda_parallel_launch_local_memory
<Kokkos::Impl::ParallelReduce<Kokkos::Impl::CudaFunctorAdapter
<main::{lambda(int, double&)#1}, Kokkos::RangePolicy<Kokkos::Cuda>
double, void>, Kokkos::Cuda, Kokkos::InvalidType, Kokkos::RangePol
(Kokkos::Impl::ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<
main::{lambda(int, double&)#1}, Kokkos::RangePolicy<Kokkos::Cuda>,
double, void>, Kokkos::Cuda, Kokkos::InvalidType, Kokkos::RangePol
```

Aaa this is horrifying can't we do better??

Aaa this is horrifying can't we do better??

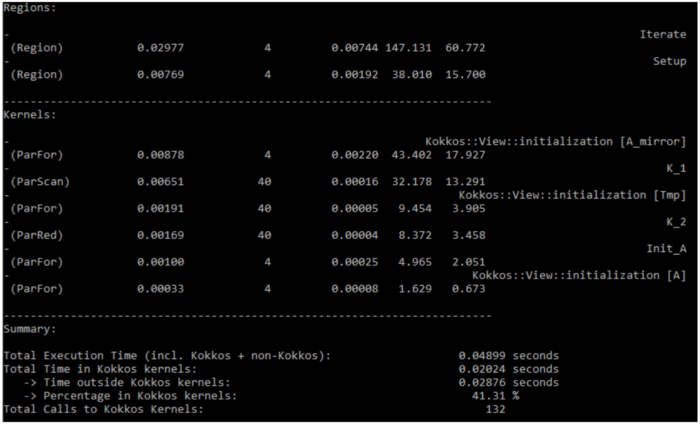
Lets use SimpleKernelTimer from Kokkos Tools:

- Simple tool producing a summary similar to nvprof
- Good way to get a rough overview of whats going on
- Writes a file `HOSTNAME-PROCESSID.dat` per process
- Use the reader accompanying the tool to read the data

Usage:

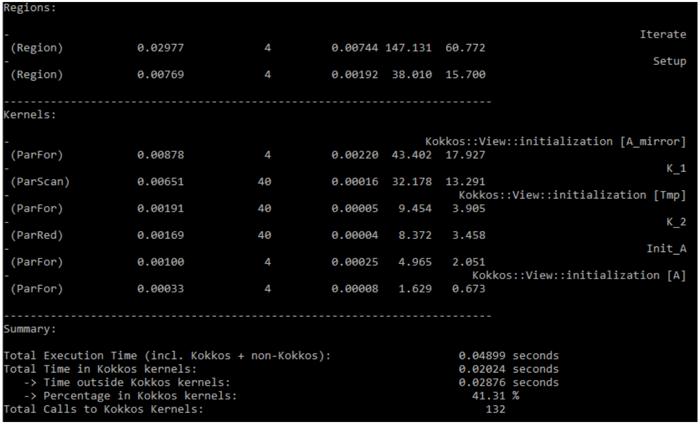
```
git clone git@github.com:kokkos/kokkos-tools
cd kokkos-tools/profiling/simple_kernel_timer
make
export KOKKOS_TOOLS_LIBS=${PWD}/kp_kernel_timer.so
export PATH=${PATH}:${PWD}
cd ${WORKDIR}
./text.cuda
kp_reader *.dat
```

Output from SimpleKernelTimer:



```
Regions:

-                                                                                    Iterate
 (Region)          0.02977           4         0.00744 147.131  60.772
-                                                                                      Setup
 (Region)          0.00769           4         0.00192  38.010  15.700

-------------------------------------------------------------------------
Kernels:

-                                                              Kokkos::View::initialization [A_mirror]
 (ParFor)          0.00878           4         0.00220  43.402  17.927
-                                                                                        K_1
 (ParScan)         0.00651          40         0.00016  32.178  13.291
-                                                                   Kokkos::View::initialization [Tmp]
 (ParFor)          0.00191          40         0.00005   9.454   3.905
-                                                                                        K_2
 (ParRed)          0.00169          40         0.00004   8.372   3.458
-                                                                                     Init_A
 (ParFor)          0.00100           4         0.00025   4.965   2.051
-                                                                     Kokkos::View::initialization [A]
 (ParFor)          0.00033           4         0.00008   1.629   0.673

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

Total Execution Time (incl. Kokkos + non-Kokkos):                    0.04899 seconds
Total Time in Kokkos kernels:                                        0.02024 seconds
   -> Time outside Kokkos kernels:                                   0.02876 seconds
   -> Percentage in Kokkos kernels:                                    41.31 %
Total Calls to Kokkos Kernels:                                           132
```

Output from SimpleKernelTimer:



```
Regions:

-                                                                                 Iterate
 (Region)         0.02977          4         0.00744 147.131  60.772
-                                                                                   Setup
 (Region)         0.00769          4         0.00192  38.010  15.700

-------------------------------------------------------------------------
Kernels:

-                                                Kokkos::View::initialization [A_mirror]
 (ParFor)         0.00878          4         0.00220  43.402  17.927
-                                                                                     K_1
 (ParScan)        0.00651         40         0.00016  32.178  13.291
-                                                     Kokkos::View::initialization [Tmp]
 (ParFor)         0.00191         40         0.00005   9.454   3.905
-                                                                                     K_2
 (ParRed)         0.00169         40         0.00004   8.372   3.458
-                                                                                  Init_A
 (ParFor)         0.00100          4         0.00025   4.965   2.051
-                                                       Kokkos::View::initialization [A]
 (ParFor)         0.00033          4         0.00008   1.629   0.673

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

Total Execution Time (incl. Kokkos + non-Kokkos):                    0.04899 seconds
Total Time in Kokkos kernels:                                        0.02024 seconds
   -> Time outside Kokkos kernels:                                   0.02876 seconds
   -> Percentage in Kokkos kernels:                                    41.31 %
Total Calls to Kokkos Kernels:                                           132
```

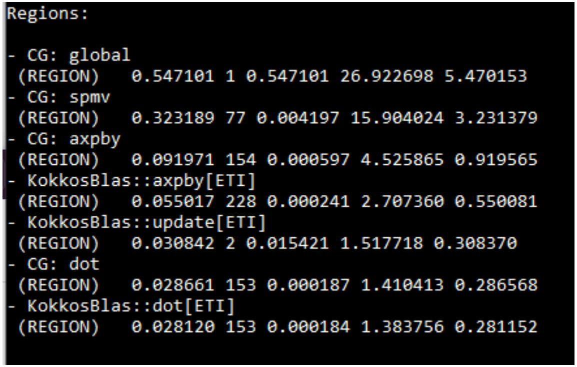
Will introduce *Regions* later.

Kernel Naming

Naming Kernels avoid seeing confusing Profiler output!

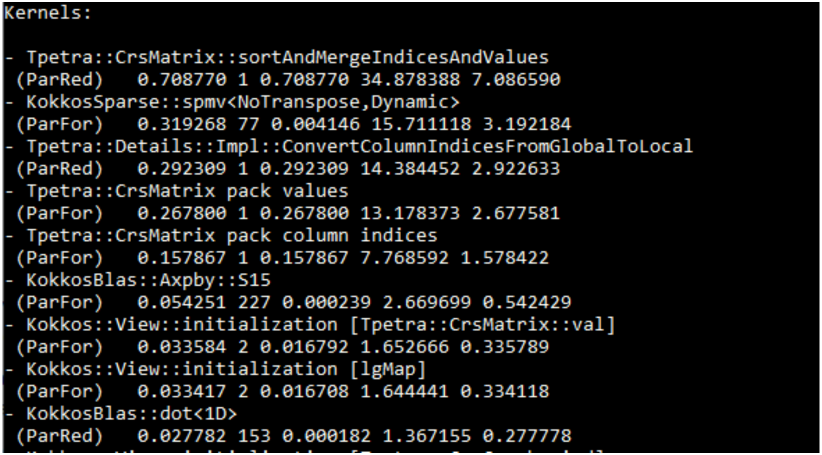
Lets look at Tpetra again with the Simple Kernel Timer Loaded:

At the top we get Region output:



```
Regions:

- CG: global
 (REGION)   0.547101 1 0.547101 26.922698 5.470153
- CG: spmv
 (REGION)   0.323189 77 0.004197 15.904024 3.231379
- CG: axpby
 (REGION)   0.091971 154 0.000597 4.525865 0.919565
- KokkosBlas::axpby[ETI]
 (REGION)   0.055017 228 0.000241 2.707360 0.550081
- KokkosBlas::update[ETI]
 (REGION)   0.030842 2 0.015421 1.517718 0.308370
- CG: dot
 (REGION)   0.028661 153 0.000187 1.410413 0.286568
- KokkosBlas::dot[ETI]
 (REGION)   0.028120 153 0.000184 1.383756 0.281152
```

Then we get kernel output:

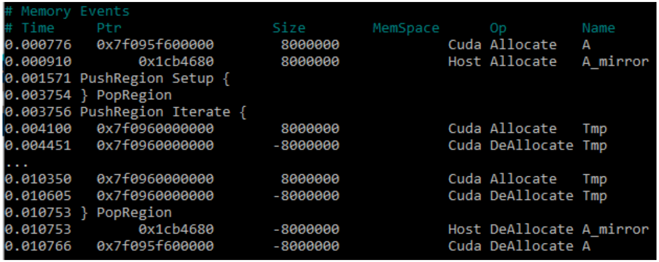


```
Kernels:

- Tpetra::CrsMatrix::sortAndMergeIndicesAndValues
 (ParRed)   0.708770 1 0.708770 34.878388 7.086590
- KokkosSparse::spmv<NoTranspose,Dynamic>
 (ParFor)   0.319268 77 0.004146 15.711118 3.192184
- Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal
 (ParRed)   0.292309 1 0.292309 14.384452 2.922633
- Tpetra::CrsMatrix pack values
 (ParFor)   0.267800 1 0.267800 13.178373 2.677581
- Tpetra::CrsMatrix pack column indices
 (ParFor)   0.157867 1 0.157867 7.768592 1.578422
- KokkosBlas::Axpby::S15
 (ParFor)   0.054251 227 0.000239 2.669699 0.542429
- Kokkos::View::initialization [Tpetra::CrsMatrix::val]
 (ParFor)   0.033584 2 0.016792 1.652666 0.335789
- Kokkos::View::initialization [lgMap]
 (ParFor)   0.033417 2 0.016708 1.644441 0.334118
- KokkosBlas::dot<1D>
 (ParRed)   0.027782 153 0.000182 1.367155 0.277778
```

Understanding MemorySpace Utilization is critical

Three simple tools for understanding memory utilization:

- MemoryHighWaterMark: just the maximum utilization for each memory space.
- MemoryUsage: Timeline of memory usage.
- MemoryEvents: allocation, deallocation and deep_copy.
  - Name, Memory Space, Pointer, Size



```
# Memory Events
# Time      Ptr                   Size        MemSpace      Op         Name
0.000776   0x7f095f600000        8000000              Cuda Allocate   A
0.000910        0x1cb4680        8000000              Host Allocate   A_mirror
0.001571 PushRegion Setup {
0.003754 } PopRegion
0.003756 PushRegion Iterate {
0.004100   0x7f0960000000        8000000              Cuda Allocate   Tmp
0.004451   0x7f0960000000       -8000000              Cuda DeAllocate Tmp
...
0.010350   0x7f0960000000        8000000              Cuda Allocate   Tmp
0.010605   0x7f0960000000       -8000000              Cuda DeAllocate Tmp
0.010753 } PopRegion
0.010753        0x1cb4680       -8000000              Host DeAllocate A_mirror
0.010766   0x7f095f600000       -8000000              Cuda DeAllocate A
```

Adding region markers to capture more code structure

Region Markers are helpful to:

- Find where time is spent outside of kernels.
- Group Kernels which belong together.
- Structure code profiles.
    - For example bracket *setup* or *solve* phase.

Adding region markers to capture more code structure
Region Markers are helpful to:

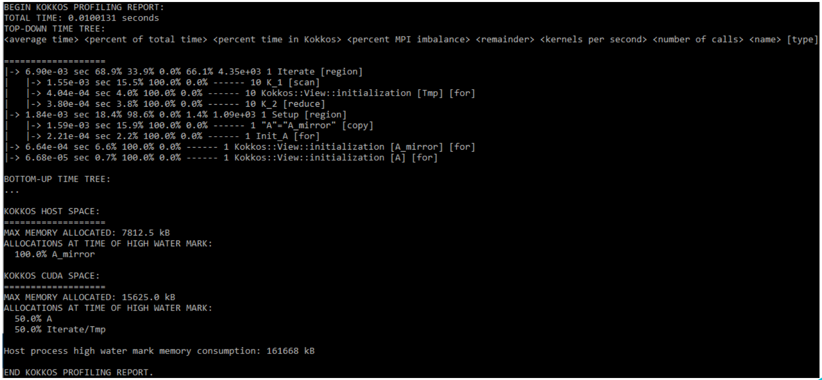
- Find where time is spent outside of kernels.
- Group Kernels which belong together.
- Structure code profiles.
  - For example bracket *setup* or *solve* phase.

Simple Push/Pop interface:

```
Kokkos::Profiling::pushRegion("Label");
...
Kokkos::Profiling::popRegion();
```

The simplest tool to leverage regions is the **Space Time Stack**:

- **Bottom Up** and **Top Down** data representation
- Can do MPI aggregation if compiled with MPI support
- Also aggregates memory utilization info.



```
BEGIN KOKKOS PROFILING REPORT:
TOTAL TIME: 0.0100131 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]

===================
|-> 6.90e-03 sec 68.9% 33.9% 0.0% 66.1% 4.35e+03 1 Iterate [region]
|   |-> 1.55e-03 sec 15.5% 100.0% 0.0% ------ 10 K_1 [scan]
|   |-> 4.04e-04 sec 4.0% 100.0% 0.0% ------ 10 Kokkos::View::initialization [Tmp] [for]
|   |-> 3.80e-04 sec 3.8% 100.0% 0.0% ------ 10 K_2 [reduce]
|-> 1.84e-03 sec 18.4% 98.6% 0.0% 1.4% 1.09e+03 1 Setup [region]
|   |-> 1.59e-03 sec 15.9% 100.0% 0.0% ------ 1 "A"="A_mirror" [copy]
|   |-> 2.21e-04 sec 2.2% 100.0% 0.0% ------ 1 Init_A [for]
|-> 6.64e-04 sec 6.6% 100.0% 0.0% ------ 1 Kokkos::View::initialization [A_mirror] [for]
|-> 6.68e-05 sec 0.7% 100.0% 0.0% ------ 1 Kokkos::View::initialization [A] [for]

BOTTOM-UP TIME TREE:
...

KOKKOS HOST SPACE:
===================
MAX MEMORY ALLOCATED: 7812.5 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  100.0% A_mirror

KOKKOS CUDA SPACE:
===================
MAX MEMORY ALLOCATED: 15625.0 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  50.0% A
  50.0% Iterate/Tmp

Host process high water mark memory consumption: 161668 kB

END KOKKOS PROFILING REPORT.
```

Non-Blocking Dispatch implies asynchronous error reporting!

```
Profiling::pushRegion("Iterate");
for(int r=0; r<10; r++) {
  parallel_for("K_1",2*N, KOKKOS_LAMBDA(int i) {a(i) = i;});
  printf("Passed␣point␣A\n");
  double sum;
  parallel_reduce("K_2",N, KOKKOS_LAMBDA(int i, double& lsum) {
    lsum += a(i); },sum);
}
Profiling::popRegion();
```

Output of the run:

```
./test.cuda
Passed point A
terminate called after throwing an instance of 'std::runtime_error
  what(): cudaStreamSynchronize(m_stream) error( cudaErrorIllegal
  an illegal memory access was encountered
    Kokkos/kokkos/core/src/Cuda/Kokkos_Cuda_Instance.cpp:312
Traceback functionality not available
Aborted (core dumped)
```

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

The KernelLogger is a tool to localize errors and check the actual runtime flow of a code.

- As other tools it inserts fences - which check for errors.
- Prints out Kokkos operations as they happen.

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

The KernelLogger is a tool to localize errors and check the actual runtime flow of a code.

- As other tools it inserts fences - which check for errors.
- Prints out Kokkos operations as they happen.

Output from the above test case with KernelLogger:

```
KokkosP: Allocate<Cuda> name: A pointer: 0x7f598b800000 size: 8000
KokkosP: Executing parallel-for kernel on device 0 with unique exe
KokkosP: Kokkos::View::initialization [A]
KokkosP: Execution of kernel 0 is completed.
KokkosP: Entering profiling region: Iterate
KokkosP: Executing parallel-for kernel on device 0 with unique exe
KokkosP: Iterate
KokkosP:   K_1
terminate called after throwing an instance of 'std::runtime_error
  what():  cudaDeviceSynchronize() error( cudaErrorIllegalAddress)
Traceback functionality not available
```

The standard Kokkos profiling approach

Understand Kokkos Utilization (SimpleKernelTimer)

- 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 could be 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 (next subsection) to analyze kernels.
- E.g. use roof line analysis to find underperforming code.

Exercise - Terrible MiniMD

Analyse a MiniMD variant with a serious performance issue.

Details:

- Location: `Exercises/tools_minimd/`
- Use standard Profiling Approach.
- Find the code location which causes the performance issue.
- Run with `miniMD.exe -s 20`

What should happen:

- Performance should be
- About 50% of time in a Force compute kernel
- About 25% in neighbor list creation

Basic Tool Summary

- Kokkos Tools provide an instrumentation interface **KokkosP** and **Tools** to leverage it.
- The interface is **always available** - even in release builds.
- Zero overhead if no tool is loaded during the run.
- Dynamically load a tool via setting `KOKKOS_TOOLS_LIBS` environment variable.
- Set callbacks directly in code for tools compiled into the executable.

Vendor and Independent Profiling GUIs

Connector tools translating Kokkos instrumentation.

Learning objectives:

- Understand what connectors provide
- Understand what tools are available

Using Third Party Tools

Kokkos Tools can also be used to interface and augment existing profiling tools.

- Provide context information like Kernel names
- Turn data collection on and off in a tool independent way

There are two ways this happens:

- Load a specific connector tool like `nvtx-connector`
  - For example for Nsight Compute and VTune
- Tools themselves know about Kokkos instrumentation
  - For example Tau

Connecting to Tools - Nsight Compute

Use the `nvtx-connector` to interact with NVIDIA tools

Translates KokkosP hooks into NVTX instrumentation

- Works with all NVIDIA tools which understand NVTX
- Translates Regions and Kernel Dispatches

Connecting to Tools - Nsight Compute

Use the `nvtx-connector` to interact with NVIDIA tools

Translates KokkosP hooks into NVTX instrumentation

- Works with all NVIDIA tools which understand NVTX
- Translates Regions and Kernel Dispatches

Initially wasn't very useful since regions are shown independently of kernels

Connecting to Tools - Nsight Compute

Use the `nvtx-connector` to interact with NVIDIA tools

Translates KokkosP hooks into NVTX instrumentation

- Works with all NVIDIA tools which understand NVTX
- Translates Regions and Kernel Dispatches

Initially wasn't very useful since regions are shown independently of kernels

But CUDA 11 added renaming of Kernels based on Kokkos User feedback!

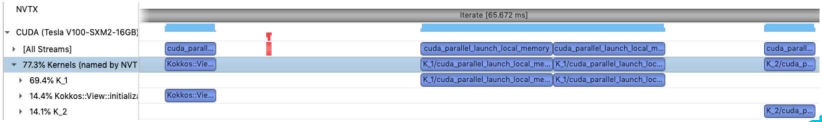
Connecting to Tools - Nsight Compute

To enable kernel renaming you need to:

- Load the nvprof-connector via setting `KOKKOS_TOOLS_LIBS` in the run configuration.
- Go to `Tools > Preferences > Rename CUDA Kernels by NVTX` and set it on.

This does a few things:

- User Labels are now used as the primary name.
- You can still expand the row to see which actual kernels are grouped under it.
  - For example if multiple kernels have the same label
- The bars are now named `Label/GLOBAL_FUNCTION_NAME`.


NVTX
Iterate [65.672 ms]
CUDA (Tesla V100-SXM2-16GB)
[All Streams]
77.3% Kernels (named by NVT
69.4% K_1
14.4% Kokkos::View::initializ
14.1% K_2
cuda_parall...
cuda_parallel_launch_local_memory
cuda_parallel_launch_local_m...
Kokkos::Vie...
K_1/cuda_parallel_launch_local_me...
K_1/cuda_parallel_launch_loc...
K_2/cuda_p...

Connecting to Tools - Vtune

To enable kernel renaming you need to:

- Load the vtune-connector via setting `KOKKOS_TOOLS_LIBS` in the run configuration.
- Choose the `Frame Domain / Frame / Function / Call Stack` grouping in the bottom up panel.

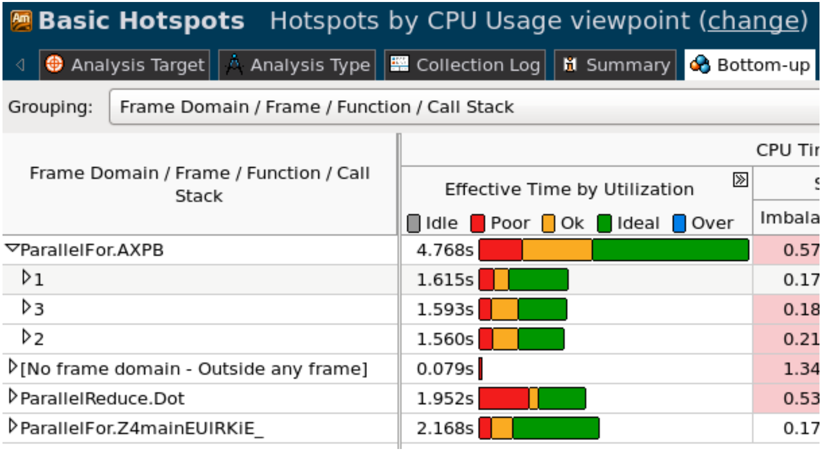
This does a few things:

- User Labels are now used as the primary name.
- You can expand to see individual kernel invocations
- You can dive further into an individual kernel invocation to see function calls within.
- Focus in on a kernel or individual invocation and do more detailed analysis.

Also available: vtune-focused-connector:

- Used in conjunction with kernel-filter tool.
- Restricts profiling to a subset of kernels.

Connecting to Tools - Vtune



Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Grouping: Frame Domain / Frame / Function / Call Stack

Frame Domain / Frame / Function / Call Stack	Effective Time by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Imbala
ParallelFor.AXPB	4.768s	0.57
1	1.615s	0.17
3	1.593s	0.18
2	1.560s	0.21
[No frame domain - Outside any frame]	0.079s	1.34
ParallelReduce.Dot	1.952s	0.53
ParallelFor.Z4mainEUIRKiE_	2.168s	0.17

TAU is a widely used Profiling Tool supporting most platforms.

Tau supports:

- profiling
- sampling
- tracing

You do not need a connector tool for Tau!

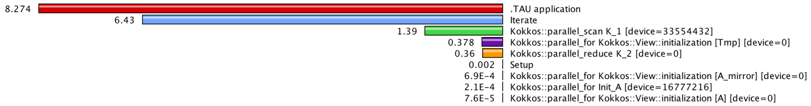
To enable TAU's Kokkos integration simply

- Download and install TAU
- Launch your program with tau_exec (which will set `KOKKOS_TOOLS_LIBS` for you)

For questions contact tau-users@cs.uoregon.edu

Connecting to Tools - Tau

Tau will use Kokkos instrumentation to display names and regions as defined by Kokkos:

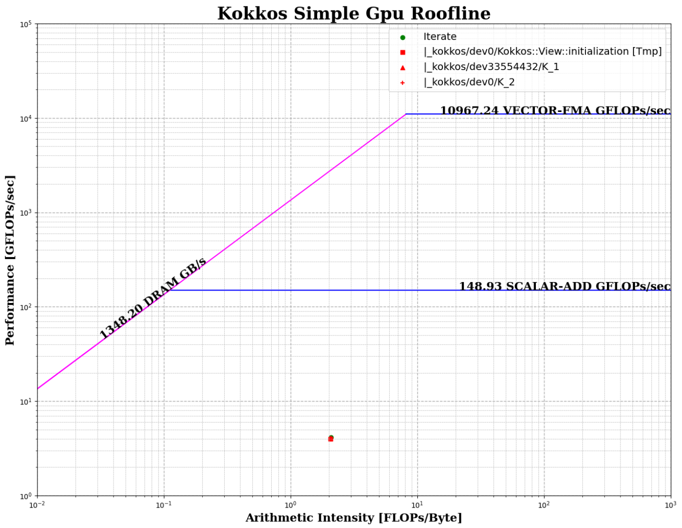


Timemory is a modular toolkit provided by NERSC that aims to simplify the creation of performance analysis tools by providing a common design pattern of classes which encapsulate how to perform a start+stop/sample/entry of "something". Each of these components (from timers to HW counters to other profilers) can be used individually with zero overhead from the library. It also provides wrappers and utilities for handling multiple components generically, data storage, writing JSON, comparing outputs, etc.

As a by-product this design, the library provides an large set of individual profiling libraries whose usage comes with the same ease as using the simple-timer tool: setting `KOKKOS_TOOLS_LIBS`.

Timemory

- It also provides novel capabilities other tools don't, like simultaneous CPU/GPU roofline modeling.
- The usage here is simple:
  - `spack install timemory +kokkos_tools +kokkos_build_config [+mpi +cuda +cupti +papi +caliper ...]`
  - Wait 3 months while spack builds every software package ever from scratch
  - In `<PREFIX>/lib/timemory/kokkos_tools/` there will be 5 to 30+ Kokkos profiling libraries
- Roofline modeling requires one additional setup
  - `timemory-roofline -T "TITLE"-t gpu_roofline -- <CMD>`
  - Where everything after -- is just running your application
- For more information: https://github.com/NERSC/timemory


Kokkos Simple Gpu Roofline
Iterate
|_kokkos/dev0/Kokkos::View::initialization [Tmp]
|_kokkos/dev33554432/K_1
|_kokkos/dev0/K_2
10967.24 VECTOR-FMA GFLOPs/sec
148.93 SCALAR-ADD GFLOPs/sec
1348.20 DRAM GB/s
Performance [GFLOPs/sec]
Arithmetic Intensity [FLOPs/Byte]
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{-2}
10^{-1}

- Caliper - Broad program analysis capabilities. UVM Profiling.
- HPCToolkit - Not a connector, but a sampling tool with great Kokkos support

Connector Summary

- Connectors inject Kokkos specific information into vendor and academic tools.
- Helps readability of profiles.
- Removes your need to put vendor specific instrumentation in your code.
- Growing list of tools support Kokkos natively.

Tuning

Using Kokkos' autotuning hooks.

Learning objectives:

- Why do we need tuning?
- What are Input and Output Variables?
- How to register parameters for tuning.
- Using the Apollo Tuner.

Lets look at the canonical implementation for SPMV in Kokkos:

```
int rows_per_team = ...;
parallel_for("SPMV",TeamPolicy<>(nrows/rows_per_team,
  team_size,vector_length),
  KOKKOS_LAMBDA(auto const team_t& team) {
  int start_row = team.league_rank()*rows_per_team;
  parallel_for(
    TeamThreadRange(team,start_row,start_row+rows_per_team),
    [&](int row) {
      int idx_begin = a.offsets(row);
      int idx_end = a.offsets(row+1);
      parallel_reduce(ThreadVectorRange(team,idx_begin,idx_end),
      [&](int i, double& lsum) {
        lsum += A.value(i) * x(A.idx(i));
      },y(row));
    });
  });
```

Why Tuning?

There are three free parameters which determine performance:

rows_per_team **team_size** **vector_length**

These parameters depend most on three factors:

- Which architecture are you on?
- How many rows are in A?
- How many non-zeros are in A?

Finding the right parameters is a daunting task.
Heuristics are possible, but they have to change all the time

- KokkosKernels' heuristic for NVIDIA K80 failed on V100
- Now AMD GPUs and Intel GPUs are coming.

Finding the right parameters is a daunting task.
Heuristics are possible, but they have to change all the time

- KokkosKernels' heuristic for NVIDIA K80 failed on V100
- Now AMD GPUs and Intel GPUs are coming.

What if you could auto tune these parameters instead?

What information would you need to provide and what comes out?
Need:

Finding the right parameters is a daunting task.
Heuristics are possible, but they have to change all the time

- KokkosKernels' heuristic for NVIDIA K80 failed on V100
- Now AMD GPUs and Intel GPUs are coming.

What if you could auto tune these parameters instead?

What information would you need to provide and what comes out?
Need:

- Context information, such as problem sizes.
- To be able to provide multiple inputs of different types.
- To tune multiple correlated parameters.
- Different tuning strategies in different areas.

Finding the right parameters is a daunting task.
Heuristics are possible, but they have to change all the time

- KokkosKernels' heuristic for NVIDIA K80 failed on V100
- Now AMD GPUs and Intel GPUs are coming.

What if you could auto tune these parameters instead?

What information would you need to provide and what comes out?
Need:

- Context information, such as problem sizes.
- To be able to provide multiple inputs of different types.
- To tune multiple correlated parameters.
- Different tuning strategies in different areas.

Kokkos Tuning

Kokkos Tuning provides a flexible runtime auto tuning interface.

Kokkos' Tuning Infrastructure is very flexible.

Kokkos' Tuning Infrastructure is very flexible.

Which makes it right now more complex than is desirable.
We will glance over some aspects here and give you the most important info for simple tuning tasks.

Kokkos' Tuning Infrastructure is very flexible.

Which makes it right now more complex than is desirable.
We will glance over some aspects here and give you the most important info for simple tuning tasks.

Kokkos Tuning has four fundamental concepts:

- `Input-Types`: Descriptors for the type of input information for tuning tasks
- `Output-Types`: Descriptors of output variables for tuning tasks
- `Variable-Values`: Instances of `Input-Types` or `Output-Types`
- `Contexts:` Marker for tuning scopes.

The types for input variables and output variables describe what makes sense to do with a variable

- Not types in the C++ sense
- These types can contain runtime information such as candidate sets.

The types for input variables and output variables describe what makes sense to do with a variable

- Not types in the C++ sense
- These types can contain runtime information such as candidate sets.

Huh? Where is this coming from?

Think about the different optimization spaces of variables:

- Discrete sets, only specific values make sense: e.g. vector length 2, 4, 8, 16
- Continuous ranges, all values in a range $0 - N$ are valid.
- Statistical semantics, is the search space logarithmic or linear?

Often you'll have a simple case, for which we will provide helper functions.

The types for input variables and output variables describe what makes sense to do with a variable

- Not types in the C++ sense
- These types can contain runtime information such as candidate sets.

Huh? Where is this coming from?

Think about the different optimization spaces of variables:

- Discrete sets, only specific values make sense: e.g. vector length 2, 4, 8, 16
- Continuous ranges, all values in a range $0 - N$ are valid.
- Statistical semantics, is the search space logarithmic or linear?

Often you'll have a simple case, for which we will provide helper functions.

Tuning Variables (both input and output) need to accomodate these situations.

We will discuss this later

Kokkos Provided API will be highlighted, and is in the namespace *Kokkos::Tools::Experimental*

Start by creating the types (helper functions discussed later):

```
std::vector<int64_t> candidates = {0, 3, 7, 11};
size_t tuning_candidate_type_id =
  create_tuning_output_type("values",candidates);
size_t tuning_input_type_id =
  create_tuning_input_type("kernels");
```

Next create variables for the inputs:

```
VariableValue input_A =
  make_variable_value(tuning_input_type_id,"A");
VariableValue input_B =
  make_variable_value(tuning_input_type_id,"B");
```

The actual tuning region is scoped through a context:

```
size_t context_1 = get_new_context_id();
begin_context(context_1);
// This is the tuned region
end_context(context_1);
```

Tuning a single variable

The context scope defines both the timing for the tuning operation and the scope in which to set input variables and obtain output (tuned) variables:

```
size_t context_1 = get_new_context_id();
begin_context(context_1);
set_input_values(context_1, 1, &input_value_A);
request_output_values(context_1, 1, &tuned_value);
end_context(context_1);
```

Tuning a single variable

The context scope defines both the timing for the tuning operation and the scope in which to set input variables and obtain output (tuned) variables:

```
size_t context_1 = get_new_context_id();
begin_context(context_1);
set_input_values(context_1, 1, &input_value_A);
request_output_values(context_1, 1, &tuned_value);
end_context(context_1);
```

In this case we used a `Categorical` input value

- Essentially just marks a code path as used here.
- But for SPMV optimal vector length depends on row lengths!
  - If there is only 1 matrix: categorical works
  - Else need numerical input value, where output mapping depends on input potentially not just as a lookup.

Tuning a single variable

The context scope defines both the timing for the tuning operation and the scope in which to set input variables and obtain output (tuned) variables:

```
size_t context_1 = get_new_context_id();
begin_context(context_1);
set_input_values(context_1, 1, &input_value_A);
request_output_values(context_1, 1, &tuned_value);
end_context(context_1);
```

In this case we used a `Categorical` input value

- Essentially just marks a code path as used here.
- But for SPMV optimal vector length depends on row lengths!
  - If there is only 1 matrix: categorical works
  - Else need numerical input value, where output mapping depends on input potentially not just as a lookup.

We also only used one input and one output value:

- Interface takes pointers to arrays for multiple `VariableValue`!

Helper Functions

The code we demonstrated before used helper functions. We'll show their implementation to help demonstrate some details.

```
template <class T>
size_t create_tuning_output_type(
    const char* name,
    std::vector<T>& candidate_values) {
  using Kokkos::Tools::Experimental;
  VariableInfo tuningVariableInfo;
  tuningVariableInfo.category =
      StatisticalCategory::kokkos_value_categorical;
  tuningVariableInfo.type = std::is_integral<T>::value ?
      ValueType::kokkos_value_int64 :
      ValueType::kokkos_value_double;
  tuningVariableInfo.valueQuantity =
      CandidateValueType::kokkos_value_set;
  tuningVariableInfo.candidates = make_candidate_set(
      candidate_values.size(),
      candidate_values.data());
  return declare_output_type(name, tuningVariableInfo);
}
```

Helper Functions Continued

```
size_t create_tuning_input_type(const char* name) {
  using Kokkos::Tools::Experimental;
  VariableInfo info;
  info.category = StatisticalCategory::kokkos_value_categorical;
  info.type = ValueType::kokkos_value_string;
  info.valueQuantity = kokkos_value_unbounded;
  return declare_input_type(name, info);
}
```

Apollo: a Prototype Tuning Tool

Apollo: A model driven auto tuning tool

- Most feature-rich Tuning tool currently targeting this interface.
- Builds decision tree based models.
- Can retrain models if observed and expected performance deviate.
- Can save models for subsequent runs.

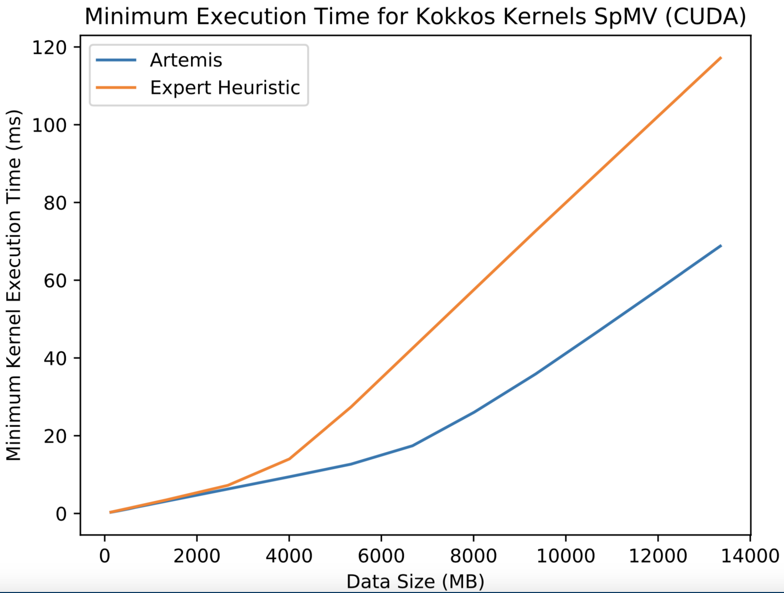
Apollo: A model driven auto tuning tool

- Most feature-rich Tuning tool currently targeting this interface.
- Builds decision tree based models.
- Can retrain models if observed and expected performance deviate.
- Can save models for subsequent runs.

How to use Apollo:

```
export KOKKOS_TOOLS_LIBS=${APOLLO_PATH}/libapollo-tuner.so
./Executable.exe ARGS
```

Some Results from the KokkosKernels Test Suite



Rules for Tuning

- Load the output value array you pass to request_output_values with sane defaults. If the tool doesn't overwrite them, your program shouldn't crash. This protects you from a tool-free situation
- No choice from your set/range of candidates should crash your program. Options can be slow, but must all be functional
- Call set_input_values and request_output_values only once per context.

Future: Built-in Tuning

For the future we plan on allowing automatic internal tuning of things like:

- Team Size and Vector Length for `TeamPolicy`
- Tile Sizes for `MDRangePolicy`
- CUDA block size of `RangePolicy`
- Occupancy of kernels.

```
parallel_for("A",TeamPolicy<>(N,AUTO,AUTO), ... );
parallel_for("B",MDRangePolicy<>({0,0},{N0,N1},{AUTO,AUTO}), ... )
```

Future: Built-in Tuning

For the future we plan on allowing automatic internal tuning of things like:

- Team Size and Vector Length for `TeamPolicy`
- Tile Sizes for `MDRangePolicy`
- CUDA block size of `RangePolicy`
- Occupancy of kernels.

```
parallel_for("A",TeamPolicy<>(N,AUTO,AUTO), ... );
parallel_for("B",MDRangePolicy<>({0,0},{N0,N1},{AUTO,AUTO}), ... )
```

But often more context is needed:

- Kokkos on its own has limited information: Label, Iteration Range, and Kernel Type ID.
- SPMV: can't distinguish two matrices with same row count but vastly different row lengths.
- Stencil: can't distinguish runtime stencil depth.

Kokkos Tuning Hooks enable more performance portability

- Avoid figuring out the right heuristic for every platform.
- Will be more valuable when targeting Intel, NVIDIA and AMD GPUs as well as ARM, Intel, IBM and AMD CPUs!

The app provides input variables to describe the context

- Input variables are descriptors of the problem scope.
- Categorical, Ranges, Sets are possible.
- Describe scaling for Ranges such as logarithmic or linear for categorizing problems.

The app requests output variables

- Same type system as input variables.
- Enables the description of the search space for tools.

Custom Tools

How to write your own tools for the KokkosP interface.

Learning objectives:

- The KokkosP hooks
- Callback registration inside the application
- Throwaway debugging tools

KokkosTools also allow you to write your own tools!

- Implement a simple C interface.
- Only implement what you want to use!
- Full access to the entire instrumentation.

But why would I want to do that?

- Profiling tools which know about your code structure and properly categorize information.
- Add in situ analysis hooked into your CI system.
- Write debugging tools specific for your framework.
- Write throwaway debugging tools for larger apps, instead of recompiling.

KokkosTools also allow you to write your own tools!

- Implement a simple C interface.
- Only implement what you want to use!
- Full access to the entire instrumentation.

But why would I want to do that?

- Profiling tools which know about your code structure and properly categorize information.
- Add in situ analysis hooked into your CI system.
- Write debugging tools specific for your framework.
- Write throwaway debugging tools for larger apps, instead of recompiling.

We will first walk through the hooks and then illustate with an example.

Some Helper Classes

```
// Contains a unique device identifier.
struct KokkosPDeviceInfo { uint32_t deviceID; };

// Unique name of execution and memory spaces.
struct SpaceHandle { char name[64]; };
```

Some Helper Classes

```
// Contains a unique device identifier.
struct KokkosPDeviceInfo { uint32_t deviceID; };

// Unique name of execution and memory spaces.
struct SpaceHandle { char name[64]; };
```

Initialization and Finalization hooks

```
extern "C" void kokkosp_init_library(
  int loadseq, uint64_t version, uint32_t num_devinfos,
  KokkosPDeviceInfo* devinfos);
```

- Called during `Kokkos::initialize`
- Provides device ids used subsequently.
- Use this call to setup tool infrastructure.

```
extern "C" void kokkosp_finalize_library();
```

- Called during `Kokkos::finalize`
- Usually used to output results.

Parallelism Hooks

```
extern "C" {
  void kokkosp_begin_parallel_for(const char* name,
                                  uint32_t devid,
                                  uint64_t* kernid);
  void kokkosp_begin_parallel_reduce(const char* name,
                                     uint32_t devid,
                                     uint64_t* kernid);
  void kokkosp_begin_parallel_scan(const char* name,
                                   uint32_t devid,
                                   uint64_t* kernid);
};
```

- Called when a parallel dispatch is initiated.
- `name` is the user provided string or a typeid.
- `kernid` is set by the tool to match up with the end call.

```
extern "C" void kokkosp_end_parallel_for(uint64_t kernid);
extern "C" void kokkosp_end_parallel_reduce(uint64_t kernid);
extern "C" void kokkosp_end_parallel_scan(uint64_t kernid);
```

- Called when a parallel dispatch is done.
- `kernid` is the value the begin call set.

Memory Hooks

```
extern "C" void kokkosp_begin_deep_copy(
  SpaceHandle dst_hndl, const char* dst_name, const void* dst_ptr,
  SpaceHandle src_hndl, const char* src_name, const void* src_ptr,
  uint64_t size);
```

- Called when a deep_copy is started.
- Provides space handles, names, ptrs and size of allocations.

```
extern "C" void kokkosp_end_deep_copy();
```

- Called when a deep_copy is done.

```
extern "C" void kokkosp_allocate_data(SpaceHandle hndl,
  const char* name, void* ptr, uint64_t size);
extern "C" void kokkosp_deallocate_data(SpaceHandle hndl,
  const char* name, void* ptr, uint64_t size);
```

- Called when allocating or deallocating data.

Callback Registration

Sometimes it is useful to build a tool into an executable.

Callback Registration

Kokkos Tools provide a callback setting system to set tool callbacks from within the application.

Takes the form of:

```
void set_HOOK_callback(HOOK_FUNCTION_PTR callback);
```

Where `HOOK` is one of

```
init finalize push_region pop_region begin_parallel_for
end_parallel_for begin_parallel_reduce end_parallel_reduce
begin_parallel_scan end_parallel_scan begin_fence end_fence
allocate_data deallocate_data begin_deep_copy end_deep_copy
```

One can also store a callback set, reload it and pause tool calls

```
EventSet get_callbacks();  void set_callbacks(EventSet);
void pause_tools();        void resume_tools();
```

Example:

```
#include <Kokkos_Core.hpp>
using Kokkos::Profiling;
using Kokkos::Tools::Experimental;
using Kokkos;

void kokkosp_allocate_data(SpaceHandle space,
  const char* label, const void* const ptr, uint64_t size) {
  printf("Allocate:␣[%s]␣%lu\n",label,size);
}
void kokkosp_deallocate_data(SpaceHandle space,
  const char* label, const void* const ptr, uint64_t size) {
  printf("Deallocate:␣[%s]␣%lu\n",label,size);
}

int main(int argc, char* argv[]) {
  initialize(argc, argv);
  set_allocate_data_callback(kokkosp_allocate_data);
  set_deallocate_data_callback(kokkosp_deallocate_data);
  ...
  finalize();
}
```

Example: Throwaway Debugging Tool

Sometimes you just need to know what is in a View before and after entering a kernel for the 5th time:

- The view is on the GPU and its on some rank of a large run.
- Recompiling the app takes hours.

Example: Throwaway Debugging Tool

Sometimes you just need to know what is in a View before and after entering a kernel for the 5th time:

- The view is on the GPU and its on some rank of a large run.
- Recompiling the app takes hours.

Simple Kokkos tool could do it!
What we need:

- Store the pointer and size of the view with a specific label when it gets allocated.
- Print the View when entering a kernel and before exiting it.
- Make sure the view didn't get deallocated in the mean time.

Example: Throwaway Debugging Tool

Store the pointer:

```
int* data; uint64_t N; int count;
extern "C" void kokkosp_allocate_data(SpaceHandle handle,
  const char* name, void* ptr, uint64_t size) {
  if(strcmp(name,"PuppyWeights")==0) {
    data = (int*)ptr+32; N = size; count = 0;
}}
```

Print the View:

```
void print_data() {
  std::vector<int> hcpy(N);
  cudaMemcpy(hcpy.data(),data,N*sizeof(int));
  for(int i=0;i<N;++i) printf("(%d␣%d)",i,hcpy[i]); printf("\n");
}
extern "C" void kokkosp_begin_parallel_for(const char* name,
  uint32_t, uint64_t* kernid) {
  if(strcmp(name,"PuppyOnCouch")==0) {
    count++; if(count==5) print_data(); *kernid=1;
  } else { *kernid = 0; }
}
extern "C" void kokkosp_end_parallel_for(uint64_t kernid) {
  if(kernid == 1 && count==5) print_data();
}
```

```
#include <Kokkos_Core.hpp>
#include <cmath>

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  {
    int N = argc > 1 ? atoi(argv[1]) : 12;
    int R = argc > 2 ? atoi(argv[2]) : 10;
    Kokkos::View<double*> a("PuppyWeights",N);

    for(int r=0; r<R; r++) {
      Kokkos::parallel_for("PuppyOnCouch",N,KOKKOS_LAMBDA(int i)
                           { a(i) = i*r; });
    }
  }
  Kokkos::finalize();
}
```

Output:

```
(0 0) (1 4) (2 8) (3 12)
(0 0) (1 5) (2 10) (3 15)
```

Hooks Summary

Implementing your own tools is easy!

- Simply implement the needed C callback functions.
- Only implement what you need.
- Goal is to make it simple enough so that one-off tools are a viable debugging help.

Callback registration for applications

- The callback registration system allows to embed tools in applications.
- Store callback sets and restore them.

Clang Based Static Analysis

Goals of this section

- Introduce The Possibility Of Kokkos Specific Warnings
- Show The Three Classes Of Errors We Can Detect
- Show You How To Use Them
- List Current/Planned Warnings

Can We Have Kokkos Specific Warnings even if the current configuration compiles?

```
void fooOOPS(int i) { printf("%i\n", i); }

int main(int argc, char **argv) {
  // Initialize  ...
  Kokkos::parallel_for(15, KOKKOS_LAMBDA(int i) {
       fooOOPS(i);
      });
  }
  // Finalize ...
}
```

Can We Have Kokkos Specific Warnings even if the current configuration compiles?

```
void fooOOPS(int i) { printf("%i\n", i); }

int main(int argc, char **argv) {
  // Initialize  ...
  Kokkos::parallel_for(15, KOKKOS_LAMBDA(int i) {
       fooOOPS(i);
      });
  }
  // Finalize ...
}
```

Answer: Yes, now we can.

```
void fooOOPS(int i) { printf("%i\n", i); }

int main(int argc, char **argv) {
  // Initialize ...
  Kokkos::parallel_for(15, KOKKOS_LAMBDA(int i) {
       fooOOPS(i);
      });
  }
  // Finalize ...
}
```

Using clang-tidy

```
> clang-tidy -checks=-*,kokkos-* file.cpp
    <file:line:col> warning: Function 'fooOOPS' called in
                         a lambda was missing
                         KOKKOS_X_FUNCTION annotation.
         fooOOPS(i);
         ^
<file:line:col> note: Function 'fooOOPS' was delcared here
void fooOOPS(int i) { printf("%i\n", i); }
```

Could become compiler errors

```
void fooOOPS(int i) { printf("%i\n", i);}
KOKKOS_FUNCTION void foo(){fooOOPS(1);}
```

Could become runtime crashes

```
struct bar {
  int baz;
  void foo(){parallel_for(15, KOKKOS_LAMBDA(int){baz;});}
};
```

Will produce incorrect results

```
double foo(){
  double d;
  auto func = KOKKOS_LAMBDA(int i, double sum){sum += i;};
  parallel_reduce(15, func, d);
  return d;
}
```

How to use

- **Code:** kokkos/llvm-project
- **Build:** llvm build instructions
- **Run:** The same way you would normally use clang-tidy, except with kokkos checks enabled.

Usage Examples: With Cmake

```
#! /bin/bash

cmake \
  /path/to/kokkos/code/you/want/to/build \
  -DKokkos_ROOT="/path/to/installed/kokkos" \
  -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
  -DCMAKE_CXX_CLANG_TIDY="clang-tidy;-checks=kokkos-*"
```

The above will:

- make a compile_commands.json file that clang-tidy and clangd can use
- invoke clang-tidy on all of the files compiled by the CXX compiler

If the kokkos clang-tidy is not in the path you will need to put the full path to it.

Usage Examples: Invoke clang-tidy directly

```
> clang-tidy -checks=-*,kokkos-* file.cpp
    <file:line:col> warning: Function 'fooOOPS' called in
                             a lambda was missing
                             KOKKOS_X_FUNCTION annotation.
         fooOOPS(i);
         ^
<file:line:col> note: Function 'fooOOPS' was delcared here
void fooOOPS(int i) { printf("%i\n", i); }
```

Assumes that we have the compile_commands.json file from the previous slide either in the current directory or in a parent directory.

Usage Examples: As part of clangd

```
void fooOOPS(int i) { printf("%i\n", i); }

int main(int argc, char **argv) {
  // Initialize  ...
  Kokkos::parallel_for(15, KOKKOS_LAMBDA(int i) {
       fooOOPS(i); Function 'fooOOPS' called in lambda...
      });
  }
  // Finalize ...
}
```

clangd is a language server that can work with many editors via a plugin.

Video Demo Of Clang Tools

State of The Tool

Current Checks

- Ensure KOKKOS_FUNCTION (the one you saw here)
- KOKKOS_LAMBDA captures implicit this

Beta and planned checks

- parallel_reduce functor takes argument by reference
- Nested reference lambda capture const behavior
- Unallowed types like std::vector in Kokkos contexts
- Force users to provide names for kernels

Your Issue?

- **Send us your requests:** kokkos/llvm-project

Module 7: Summary

Kokkos Tools:

- Kokkos Tools provide an instrumentation interface **KokkosP** and **Tools** to leverage it.
- The interface is **always available** - even in release builds.
- Zero overhead if no tool is loaded during the run.
- Dynamically load a tool via setting `KOKKOS_TOOLS_LIBS` environment variable.
- Set callbacks in code for tools compiled into the executable.

Kokkos Connector Tools:

- Connectors inject Kokkos specific information into vendor and academic tools.
- Helps readability of profiles.
- Removes need to put vendor specific instrumentation in codes.
- Growing list of tools support Kokkos natively.

Module 7: Summary

Kokkos Tuning Hooks enable more performance portability

- Avoid figuring out the right heuristic for every platform.
- Input variables descripte the problem scope.
- Output variables descripe the search space.

Implementing your own tools is easy!

- Simply implement the needed C callback functions.
- Only implement what you need.
- The callback registration system allows to embed tools in applications.

Static Analysis

- Have semantic checks going beyond C++ errors.
- Integrates into your editors.

KokkosKernels Dense Linear Algebra
KokkosKernels Sparse Linear Algebra
KokkosKernels Sparse Solvers
KokkosKernels Graph Kernels

Don't Forget: Join our Slack Channel and drop into our office hours on Tuesday.

Updates at: kokkos.link/the-lectures-updates

Recordings/Slides: kokkos.link/the-lectures