Kokkos: An Introduction

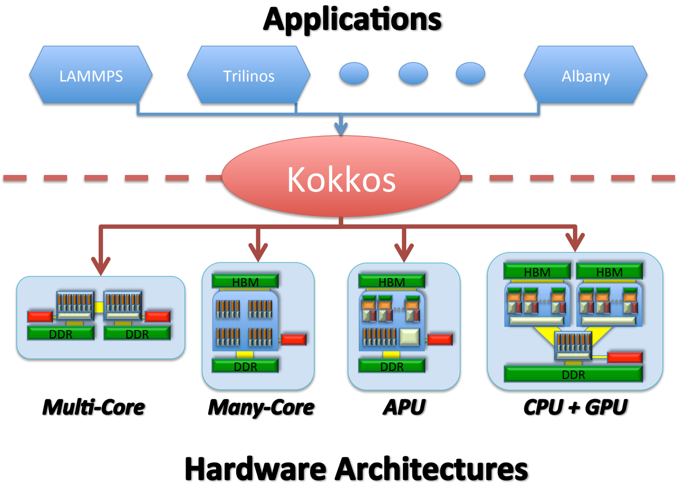
Christian R. Trott [1], H. Carter Edwards [1]

[1]Sandia National Laboratories

Kokkos Short Tutorial: Version 1.0

SAND2015-6737 PE

Motivation: Increasing Node Complexity



Kokkos the Programming Model

- **Machine model**
  - N execution spaces $\times$ M memory spaces
  - $N \times M$ matrix for memory access performance/possibility
  - Asynchronous execution allowed
- **Implementation Approach**
  - A C++ template library
  - Application focused: each feature is requested by application and used right now
  - Performance focused: very high bar for acceptance if a feature impeders performance
  - C++11 required
  - Target different back-ends for different hardware architectures
  - Provide abstraction layers for execution and memory
- **Distribution**
  - Open Source library
  - Available on Github: github.com/kokkos/kokkos

Abstraction Concepts

Execution Pattern: parallel_for, parallel_reduce, parallel_scan, task, ...
Execution Policy: how (and where) a user function is executed

- E.g., data parallel range : concurrently call function(i) for i = [0..N)
- User's function is a C++ functor or C++11 lambda

Execution Space: where functions execute

- Encapsulates hardware resources; e.g., cores, GPU, vector units, ...

Memory Space: where data resides

- AND what execution space can access that data
- Also differentiated by access performance; e.g., latency & bandwidth

Memory Layout: how data structures are ordered in memory

- provide mapping from logical to physical index space

Memory Traits: how data shall be accessed

- allow specialisation for different usage scenarios (read only, random, atomic, ...)

Execution Pattern

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

int main(int argc, char* argv[]) {
  // Initialize Kokkos analogous to MPI_Init()
  // Takes arguments which set hardware resources (number of threads, GPU Id)
  Kokkos::initialize(argc, argv);

  // A parallel_for executes the body in parallel over the index space, here a simple range 0<=i<10
  // It takes an execution policy (here an implicit range as an int) and a functor or lambda
  // The lambda operator has one argument, and index_type (here a simple int for a range)
  Kokkos::parallel_for(10,[=](int i){
    printf("Hello %i\n",i);
  });

  // A parallel_reduce executes the body in parallel over the index space,
  // and performs a reduction over the values given to the second argument
  // It takes an execution policy; a functor or lambda; and a return value
  double sum = 0;
  Kokkos::parallel_reduce(10,[=](int i, int& lsum) {
    lsum += i;
  },sum);
  printf("Result %lf\n",sum);

  // A parallel_scan executes the body in parallel over the index space, and
  // performs a scan operation over the values given to the second argument
  // If final == true lsum contains the prefix sum.
  double sum = 0;
  Kokkos::parallel_scan(10,[=](int i, int& lsum, bool final) {
    if(final) printf("ScanValue %i\n",lsum);
    lsum += i;
  });

  Kokkos::finalize();
}
```

Execution Pattern

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

int ma
  // I
  // T
  Kokk
```

Parallel Loop: `parallel_for(count, lambda)`

- equivalent to a #pragma omp for

```
  // A parallel_for executes the body in parallel over the index space, here a simple range 0<=i<10
  // It takes an execution policy (here an implicit range as an int) and a functor or lambda
  // The lambda operator has one argument, and index_type (here a simple int for a range)
  Kokkos::parallel_for(10,[=](int i){
    printf("Hello␣%i\n",i);
  });

  // A parallel_reduce executes the body in parallel over the index space,
  // and performs a reduction over the values given to the second argument
  // It takes an execution policy; a functor or lambda; and a return value
  double sum = 0;
  Kokkos::parallel_reduce(10,[=](int i, int& lsum) {
    lsum += i;
  },sum);
  printf("Result␣%lf\n",sum);

  // A parallel_scan executes the body in parallel over the index space, and
  // performs a scan operation over the values given to the second argument
  // If final == true lsum contains the prefix sum.
  double sum = 0;
  Kokkos::parallel_scan(10,[=](int i, int& lsum, bool final) {
    if(final) printf("ScanValue␣%i\n",lsum);
    lsum += i;
  });

  Kokkos::finalize();
}
```

Execution Pattern

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

int ma
  // I
  // T
  Kokk

  // A parallel_for executes the body in parallel over the index space, here a simple range 0<=i<10
  // It takes an execution policy (here an implicit range as an int) and a functor or lambda
  // The lambda operator has one argument, and index_type (here a simple int for a range)
  Kokk
    pr
  });

  // A
  // a
  // I
  double sum = 0;
  Kokkos::parallel_reduce(10,[=](int i, int& lsum) {
    lsum += i;
  },sum);
  printf("Result %lf\n",sum);

  // A parallel_scan executes the body in parallel over the index space, and
  // performs a scan operation over the values given to the second argument
  // If final == true lsum contains the prefix sum.
  double sum = 0;
  Kokkos::parallel_scan(10,[=](int i, int& lsum, bool final) {
    if(final) printf("ScanValue %i\n",lsum);
    lsum += i;
  });

  Kokkos::finalize();
}
```

Parallel Loop: `parallel_for(count, lambda)`

- equivalent to a #pragma omp for

Parallel Reduction: `parallel_reduce(count, lambda, result)`

- equivalent to a #pragma omp for reduction(+:lsum)
- custom reduction operators through functors with join function

Execution Pattern

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

int ma
  // I
  // T
  Kokk

  // A parallel_for executes the body in parallel over the index space, here a simple range 0<=i<10
  // It takes an execution policy (here an implicit range as an int) and a functor or lambda
  // The lambda operator has one argument, and index_type (here a simple int for a range)
  Kokk
    pr
  });

  // A
  // a
  // I
  double sum = 0;
  Kokkos::parallel_reduce(10,[=](int i, int& lsum) {
    lsum += i;
  },su
  prin

  // A
  // p
  // I
  doub
  Kokk
    if
    ls
  });

  Kokkos::finalize();
}
```

Parallel Loop: `parallel_for(count, lambda)`

- equivalent to a #pragma omp for

Parallel Reduction: `parallel_reduce(count, lambda, result)`

- equivalent to a #pragma omp for reduction(+:lsum)
- custom reduction operators through functors with join function

Parallel Scan: `parallel_scan(count, lambda)`

- no direct equivalence in OpenMP
- custom reduction operators through functors with join function
- prefix or postfix scan
- depending on architecture has to perform the loop twice

Execution Policy

```cpp
#include <Kokkos_Core.hpp>
#include <cstdio>

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);

  // The first argument for any parallel pattern function is an execution policy.
  // Kokkos has currently two pre existing Execution Policies: RangePolicy and TeamPolicy
  // A range policy executes the body end-start times; On CPUs the range gets chunked.
  Kokkos::parallel_for(Kokkos::RangePolicy<>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello %i\n",i);
  });

  // The TeamPolicy allows for hierarchical parallelism. One can use it to do nested parallelism.
  // The nested levels can be any parallel pattern, but only have special RangePolicies:
  //    - TeamThreadLoop splits the range over threads in a team
  // Note that the whole lambda body is a parallel region!
  Kokkos::parallel_for(Kokkos::TeamPolicy<>(10,8), KOKKOS_LAMBDA(Kokkos::TeamPolicy::member_type thread){
    Kokkos::parallel_reduce(Kokkos::TeamThreadRange(thread, thread.league_rank()), [=](int i, int& lsum){
      lsum += i;
    },sum);
    if (thread.team_rank() == 0)
      printf("Result %i %lf\n", thread.league_rank(), sum);
  });

  // The TeamPolicy can actually have three levels: team, thread, vector
  // On GPUs the Vector level is guaranteed to be threads within a warp
  Kokkos::parallel_for(Kokkos::TeamPolicy<>(10,8,4),KOKKOS_LAMBDA(Kokkos::TeamPolicy::member_type thread){
    Kokkos::parallel_for(Kokkos::TeamThreadRange(thread, thread.league_rank()) , [=](int i) {
      Kokkos::parallel_for(Kokkos::ThreadVectorRange(thread, 17) , [=](int j) {
        printf("Hello index %i %i %i: with thread %i\n", thread.league_rank(),i,j, thread.team_rank());
      });
    });
  });

  Kokkos::finalize();
}
```

Execution Policy

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

int main(int argc, char* argv[]) {
 Kokko

 // Th
 // Ko
 // A
 Kokko
   pri
 });

 // The TeamPolicy allows for hierarchical parallelism. One can use it to do nested parallelism.
 // The nested levels can be any parallel pattern, but only have special RangePolicies:
 //    - TeamThreadLoop splits the range over threads in a team
 // Note that the whole lambda body is a parallel region!
 Kokkos::parallel_for(Kokkos::TeamPolicy<>(10,8), KOKKOS_LAMBDA(Kokkos::TeamPolicy::member_type thread){
   Kokkos::parallel_reduce(Kokkos::TeamThreadRange(thread, thread.league_rank()), [=](int i, int& lsum){
     lsum += i;
   },sum);
   if (thread.team_rank() == 0)
     printf("Result %i %lf\n", thread.league_rank(), sum);
 });

 // The TeamPolicy can actually have three levels: team, thread, vector
 // On GPUs the Vector level is guaranteed to be threads within a warp
 Kokkos::parallel_for(Kokkos::TeamPolicy<>(10,8,4),KOKKOS_LAMBDA(Kokkos::TeamPolicy::member_type thread){
   Kokkos::parallel_for(Kokkos::TeamThreadRange(thread, thread.league_rank()) , [=](int i) {
     Kokkos::parallel_for(Kokkos::ThreadVectorRange(thread, 17) , [=](int j) {
       printf("Hello index %i %i %i: with thread %i\n", thread.league_rank(),i,j, thread.team_rank());
     });
   });
 });

 Kokkos::finalize();
}
```

RangePolicy

- split a range over execution unit
- mapping architecture dependent (chunks vs. interleaved)

Execution Policy

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

int main(int argc, char* argv[]) {
 Kokko

 // Th
 // Ko
 // A
 Kokko
   pri
 });

 // The TeamPolicy allows for hierarchical parallelism. One can use it to do nested parallelism.
 // The nested levels can be any parallel pattern, but only have special RangePolicies:
 //
 // No
 Kokko                                                                                         read){
   Kok                                                                                         lsum){
     l
   },s
   if
     p
 });

 // Th
 // On
 Kokko                                                                                         hread){
   Kok
     Kokkos::parallel_for(Kokkos::ThreadVectorRange(thread, 17) , [=](int j) {
       printf("Hello index %i %i %i: with thread %i\n", thread.league_rank(),i,j, thread.team_rank());
     });
   });
 });

 Kokkos::finalize();
}
```

RangePolicy

- split a range over execution unit
- mapping architecture dependent (chunks vs. interleaved)

TeamPolicy

- split a 2D or 3D index space over execution unit (team, thread, vector)
- 1st index is logical (e.g. number of worksets)
- 2nd and 3rd index are hardware restricted (e.g. number of hyperthreads on a core, threads in a Cuda warp)
- Nested parallelism to write generic algorithms

Kokkos View

```cpp
#include <Kokkos_Core.hpp>
#include <cstdio>

// A simple 2D array (rank==2) with one compile time dimension
// By default a view using this type will be reference counted.
typedef Kokkos::View<double*[3]> view_type;

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  // Allocate a view with the runtime dimension set to 10 and a label "A"
  // The label is used in debug output and error messages
  view_type a("A",10);

  // The view a is passed on via copy to the parallel dispatch which is
  // important if the execution space can not access the default HostSpace
  // directly (or if it is slow) as e.g. on GPUs.
  // Note: the underlying allocation is not moved, only meta_data such as
  // pointers and shape information is copied.
  Kokkos::parallel_for(10,KOKKOS_LAMBDA(int i){
    // Read and write access to data comes via operator()
    a(i,0) = 1.0*i;  a(i,1) = 1.0*i*i;  a(i,2) = 1.0*i*i*i;
  });

  double sum = 0;
  Kokkos::parallel_reduce(10,KOKKOS_LAMBDA(int i, double& lsum) {
    lsum+= a(i,0)*a(i,1)/(a(i,2)+0.1);
  },sum);

  printf("Result_%lf\n",sum);
  Kokkos::finalize();
}
```

Kokkos View

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

// A simple 2D array (rank==2) with one compile time dimension
// By default a view using this type will be reference counted.
typedef Kokkos::View<double*[3]> view_type;

int ma
  Kokk
  // A
  // T
  view

  // T
  // i
  // c
  // N
  // p
  Kokk
    //
    a(
  });

  double sum = 0;
  Kokkos::parallel_reduce(10,KOKKOS_LAMBDA(int i, double& lsum) {
    lsum+= a(i,0)*a(i,1)/(a(i,2)+0.1);
  },sum);

  printf("Result %lf\n",sum);
  Kokkos::finalize();
}
```

Kokkos View

- 0-8 dimensional array
- reference counted
- compile and runtime dimensions
- bounds checking in debug mode
- optional template parameters for

Memory Spaces

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

typedef Kokkos::View<double*[3], Kokkos::CudaSpace> view_type;

// HostMirror is a view with the same layout / padding as its parent type but in the host memory space.
// This memory space can be the same as the device memory space for example when running on CPUs
typedef view_type::HostMirror host_view_type;

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);

  view_type a("A",10);

  // Create an allocation with the same dimensions as a in the host memory space
  // If the memory space of view_type and its HostMirror are the same, no allocation
  //   will be created, but both views will see the same data
  host_view_type h_a = Kokkos::create_mirror_view(a);

  for(int i = 0; i < 10; i++) for(int j = 0; j < 3; j++)  h_a(i,j) = i*10 + j;

  // Transfer data from h_a to a. This is a no-op when both views are referencing the same data
  Kokkos::deep_copy(a,h_a);

  int sum = 0;
  Kokkos::parallel_reduce(10, KOKKOS_LAMBDA (int i, int &lsum) {
    lsum += a(i,0)-a(i,1)+a(i,2);
  },sum);

  printf("Result is %i\n",sum);
  Kokkos::finalize();
}
```

Memory Spaces

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

typedef Kokkos::View<double*[3], Kokkos::CudaSpace> view_type;

// HostMirror is a view with the same layout / padding as its parent type but in the host memory space.
// This
typede

int ma
  Kokk

  view

  // C
  // I
  //
  host

  for(
```

MemorySpace Support

- currently: HostSpace, CudaSpace, CudaUVMSpace, CudaHostPinnedSpace
- easily extensible as soon as we have hardware for that
- HostMirror: bit wise copyable version of a view in HostSpace
- if MemorySpace is HostSpace: points to same data

```
  // Transfer data from h_a to a. This is a no op when both views are referencing the same data
  Kokkos::deep_copy(a,h_a);

  int sum = 0;
  Kokkos::parallel_reduce(10, KOKKOS_LAMBDA (int i, int &lsum) {
    lsum += a(i,0)-a(i,1)+a(i,2);
  },sum);

  printf("Result is %i\n",sum);
  Kokkos::finalize();
}
```

Memory Spaces

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

typedef Kokkos::View<double*[3], Kokkos::CudaSpace> view_type;

// HostMirror is a view with the same layout / padding as its parent type but in the host memory space.
// Thi
typede

int ma
  Kokk

  view

  // C
  // I
  //
  host

  for(

  // Transfer data from h_a to a. This is a no op when both views are referencing the same data
  Kokkos::deep_copy(a,h_a);

  int sum = 0;
  Kokkos::parallel_reduce(10, KOKKOS_LAMBDA (int i, int &lsum) {
    lsum += a(i,0)-a(i,1)+a(i,2);
  },sum);

  prin
  Kokk
}
```

MemorySpace Support

- currently: HostSpace, CudaSpace, CudaUVMSpace, CudaHostPinnedSpace
- easily extensible as soon as we have hardware for that
- HostMirror: bit wise copyable version of a view in HostSpace
- if MemorySpace is HostSpace: points to same data

DeepCopy

- always explicit
- no-op if pointing to same data

```
#include <Kokkos_Core.hpp>
#include <Kokkos_Random.hpp>
#include <cstdio>

// The Layout is an optional template parameter which describes the mapping form logical indicies to
// the memory offset. Kokkos has 4 build in Layouts: LayoutLeft, LayoutRight, LayoutStride, LayoutTile
// Custom Layouts require minimal about 50-100 lines of code
typedef Kokkos::View<double**, Kokkos::LayoutLeft > view_left;
typedef Kokkos::View<double**, Kokkos::LayoutRight > view_right;

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  view_left l("L",10000,10000);

  Kokkos::View<double*> vector("V",10000);

  Kokkos::Random_XorShift64_Pool<> rand_pool(1313);
  Kokkos::fill_random(vector,rand_pool,100);
  Kokkos::fill_random(l,rand_pool,100);
  Kokkos::fill_random(r,rand_pool,100);

  // A Dense MatVec (GEMV). On GPUs LayoutLeft is better, on CPUs LayoutRight
  Kokkos::parallel_for(Kokkos::TeamPolicy<>(l.dimension_0(),16),
                      KOKKOS_LAMBDA (Kokkos::TeamPolicy::member_type thread) {
     double sum = 0;
     Kokkos::parallel_reduce(Kokkos::TeamThreadRange(thread, l.dimension_1()) , [=](int i, double& lsum)
      lsum += l(thread.league_rank(),i) * vector(i);
    },sum);
    if (thread.team_rank() == 0)
      result(thread.league_rank() = sum);
  });

  Kokkos::finalize();
}
```

Memory Layout

```
#include <Kokkos_Core.hpp>
#include <Kokkos_Random.hpp>
#include <cstdio>

// The Layout is an optional template parameter which describes the mapping form logical indicies to
// the memory offset. Kokkos has 4 build in Layouts: LayoutLeft, LayoutRight, LayoutStride, LayoutTile
// Custom Layouts require minimal about 50-100 lines of code
typedef Kokkos::View<double**, Kokkos::LayoutLeft > view_left;
typedef

int ma
  Kokk
  view

  Kokk

  Kokk
  Kokk
  Kokk
  Kokk

  // A
  Kokk

    d
    Kokkos::parallel_reduce(Kokkos::TeamThreadRange(thread, A.dimension_1()), [=](int i, double& lsum)
     lsum += l(thread.league_rank(),i) * vector(i);
    },sum);
    if (thread.team_rank() == 0)
      result(thread.league_rank() = sum);
  });

  Kokkos::finalize();
}
```

Memory Layout

- mapping of logical indices to memory offset
- use typedefs depending on architecture
- change access pattern without changing kernel code
- custom layouts about 50 lines of code
- default Layout depends on MemorySpace: assume first array index is loop index of parallel_for

Memory Traits

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

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);

  // A Default 3D-View with 2 runtime dimensions.
  // This View will be reference counted
  Kokkos::View<double**[8]> A(?A-Default?,1000,256);

  // An atomic view of A. Every access (=,+=,=-,?) will be atomic.
  // This View will be reference counted (i.e. it has a shared reference count with A
  Kokkos::View<double**[8], Kokkos::MemoryTraits<Kokkos::Atomic> A-Atomic = A;

  // A const view of the data. RandomAccess is a hint that the view will
  // be used with non contiguous accesses.
  // On GPUs using this view will utilize Texture Fetches.
  // This View will be reference counted (i.e. it has a shared reference count with A
  Kokkos::View<const double**[8], Kokkos::MemoryTraits<Kokkos::RandomAccess> A-Rand = A;

  // An unmanaged View of A. This View is not reference counted. It is invalid to access it
  // after the allocation is gone away.
  Kokkos::View<double**[8], Kokkos::MemoryTraits<Kokkos::Unmanaged> A-Unmanaged = A;

  Kokkos::finalize();
}
```

Memory Traits

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

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);

  // A
  // T
  Kokk

  // A
  // T
  Kokk
```

Memory Traits

- have views of data for different access scenarios (atomic, random access, non temporal, ...)
- map trait to hardware specific load paths/intrinsics

```
  // A const view of the data. RandomAccess is a hint that the view will
  // be used with non contiguous accesses.
  // On GPUs using this view will utilize Texture Fetches.
  // This View will be reference counted (i.e. it has a shared reference count with A
  Kokkos::View<const double**[8], Kokkos::MemoryTraits<Kokkos::RandomAccess> A_Rand = A;

  // An unmanaged View of A. This View is not reference counted. It is invalid to access it
  // after the allocation is gone away.
  Kokkos::View<double**[8], Kokkos::MemoryTraits<Kokkos::Unmanaged> A_Unmanaged = A;

  Kokkos::finalize();
}
```

Performance Impact: Layout and Traits

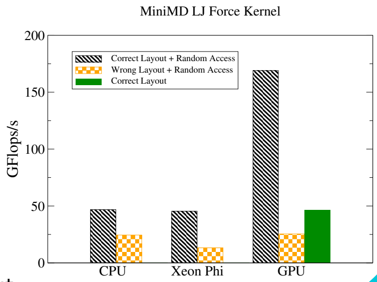
- Molecular dynamics computational kernel in miniMD
- Simple Lennard Jones force model $F_i = \sum_{j, r_{ij} < r_c} 6\epsilon(\frac{\sigma}{r_{ij}})^7 - (\frac{\sigma}{r_{ij}})^{13}$
- Atom neighbor list to avoid N^2 computations

```
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj); // 2D access: layout matters
  r_ij = pos_i ? pos(j); // random read 3 floats
  if (|r_ij| < r_cut) f_i += 6*e*((s/r_ij)^7 ? 2*(s/r_ij)^13)
}
f(i) = f_i;
```

Test Problem

- 864k atoms, 77 neighbors
- 2D neighbor array
- Different layouts CPU vs GPU
- Random read 'pos' through
- GPU texture cache

Large performance loss with wrong array layout



Subviews

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

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);

  // A Default 3D-View with 2 runtime dimensions.
  // This View will be reference counted
  Kokkos::View<double**[8]> A(?A-Default?,1000,256);

  // Generate a 2D subview of A. Equivalent to fortran A(5,1:256,1:8).
  // This View will be reference counted (i.e. it has a shared reference count with A
  auto A_5 = Kokkos::subview(A,5,Kokkos::ALL(),Kokkos::ALL());

  // Generate a 1D subview from A_5. Equivalent to fortran A_5(3:10,1);
  auto A_5_3t10_1 = Kokkos::subview(A_5,std::pair(3,10),1);

  Kokkos::finalize();
}
```

BETA FEATURE

- in-build profiling hooks (right now requires macro)
- overhead: check a function pointer
- at runtime set environment variable:
  `KOKKOS_PROFILE_LIBRARY=library_a.so,library_b.so`
- inserts fences before and after kernels
- give Kernels names: `parallel_for("Hello",N,LAMBDA)`

```
KokkosP: Finalization of Profiling Library
KokkosP: Executed a total of 3126 kernels
KokkosP: Kernel                                            Calls   s/Total   %/Ko   %/Tot   s/Call  Type
KokkosP: compute_force(OscSystem)::$_5                     1000    0.06850   96.98   91.16  0.00007 PFOR
KokkosP: update_velocity(OscSystem const&)::$_4            1000    0.00103    1.45    1.36  0.00000 PFOR
KokkosP: update_position(OscSystem const&)::$_3            1000    0.00095    1.35    1.26  0.00000 PFOR
KokkosP: compute_kinetic_energy(OscSystem const&)::$_6      100    0.00012    0.16    0.15  0.00000 RDCE
KokkosP: init_type(OscSystem const&)::$_1                     1    0.00001    0.02    0.01  0.00001 PFOR
KokkosP: init_potential(OscSystem const&)::$_2               16    0.00001    0.01    0.01  0.00000 PFOR
KokkosP: Total Execution Time:                  0.075137 seconds.
KokkosP: Time in Kokkos Kernels:                0.070629 seconds.
KokkosP: Time spent outside Kokkos:             0.004508 seconds.
KokkosP: Runtime in Kokkos Kernels:            94.000260
KokkosP: Unique kernels:                                     10
KokkosP: Parallel For Calls:                               3126
```

Wrap Up

Features which were not discussed:

- Algorithms: Sort and Random Numbers
- Containers: DualView, std::vector replacement, unordered map
- Linear Algebra: (now in Tpetra): sparse (and dense) linear algebra
- ExecutionTags: have classes act as functors with multiple tagged operators
- Custom Reductions/Scans: use functors with join, init and final functions

Whats next (next couple of years and subject to finding people):

- Kernels package in Trilinos: BLAS, Sparse LA, Graph algorithms
- Task support: under development, prototype on CPUs
- Remote memory spaces: incorporate shmem like capabilities
- Profiling support: simple inbuild capabilities + hooks for third party tools
- More debugging features: e.g. runtime identification of potential write conflicts
- Push more features into C++ standard (so far: Atomics, Views with Layouts)