Exceptional service in the national interest

Sandia National Laboratories

Kokkos: An Introduction

The Kokkos Team at Sandia:
Carter Edwards
Christian Trott

SAND2015-1465 C

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

U.S. DEPARTMENT OF ENERGY

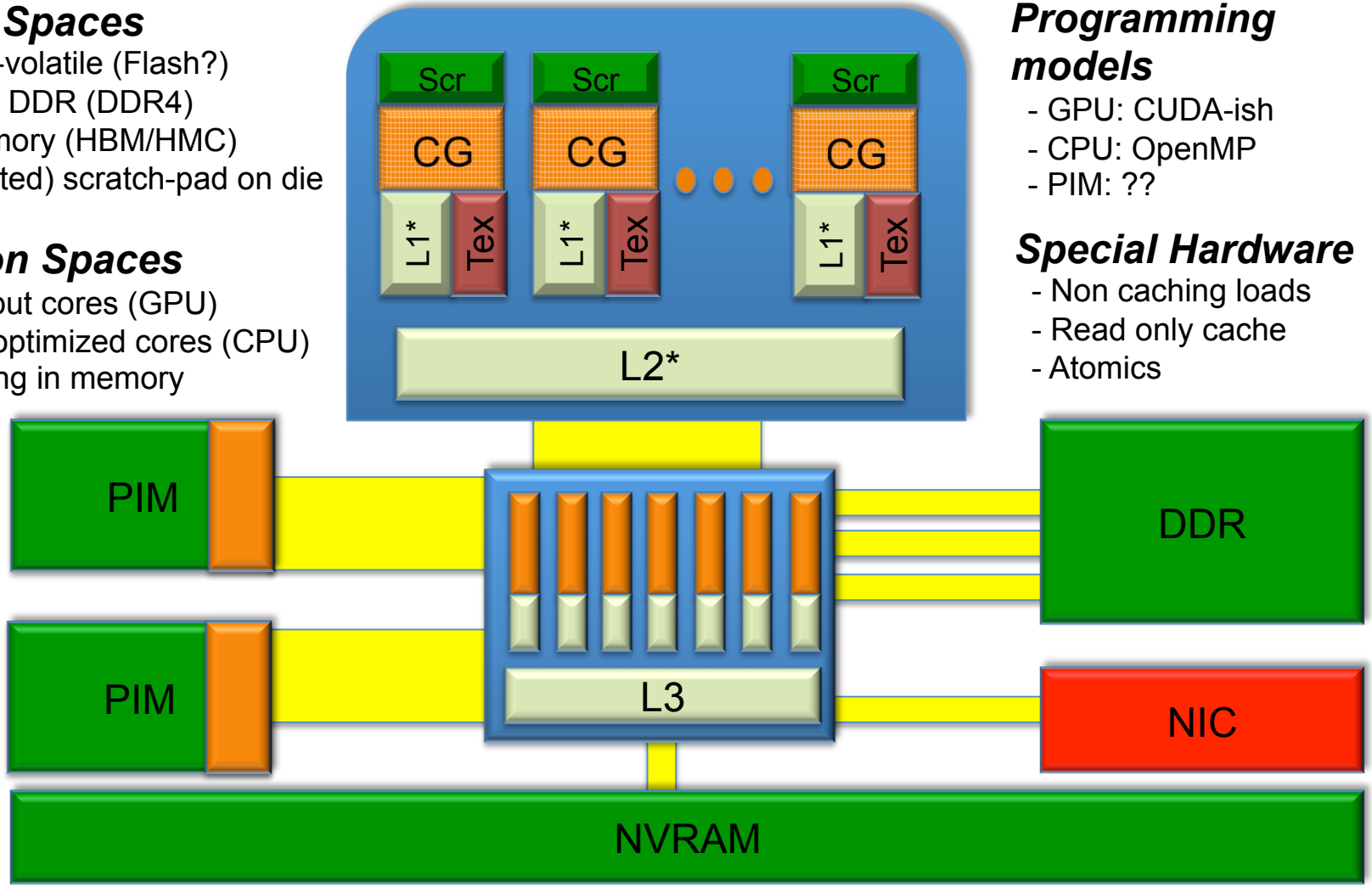
NNSA National Nuclear Security Administration

Motivation: Increasing Node Complexity

Sandia National Laboratories

Memory Spaces
- Bulk non-volatile (Flash?)
- Standard DDR (DDR4)
- Fast memory (HBM/HMC)
- (Segmented) scratch-pad on die

Execution Spaces
- Throughput cores (GPU)
- Latency optimized cores (CPU)
- Processing in memory



Programming models
- GPU: CUDA-ish
- CPU: OpenMP
- PIM: ??

Special Hardware
- Non caching loads
- Read only cache
- Atomics

Kokkos the Programming Model

Sandia National Laboratories

Goal: One code achieves good performance on every platform

- **Machine model:**
  - *N* execution spaces + *M* memory spaces
  - *NxM* 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 impedes performance
  - C++11 now required
  - Target different back-ends for different hardware architecture
  - Provide abstraction layers for execution and memory which can map to

- **Distribution**
  - Open Source library
  - Soon (i.e. in the next few weeks) available on GitHub

- **Long Term Vision**
  - Move features into the C++ standard (Carter is voting committee member)

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, …)

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

En

Our thesis: an application expressed using these abstraction concepts can be fundamentally performance portable.

Mem

- 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, …)

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, here a simple range 0<=i<10 and
  // performs a reduction over the values given to the second argument
  // It takes an execution policy (here an implicit range as an int); 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, here a simple range 0<=i<10 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 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 parall
  // performs a reduction over the values given to the second argument
  // It takes an execution policy (here an implicit range as an int); 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, here a simple range 0<=i<10 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`

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, here a simple range 0<=i<10 and
  // performs a reduction over the values given to the second argument
  // It takes an execution policy (here an implicit range as an int); 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, here a simple range 0<=i<10 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 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, here a simple range 0<=i<10 and
  // performs a reduction over the values given to the second argument
  // It takes an execution policy (here an implicit range as an int); 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_sc
  // Performs a sc
  // If final == t
  double sum = 0;
  Kokkos::parallel
    if(final) printf("ScanValue %i\n",lsum);
    lsum += i;
  });

  Kokkos::finalize();
}
```

Parallel Reduction: `parallel_reduce(count, lambda, reduction_result);`

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

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, here a simple range 0<=i<10 and
  // performs a reduction over the values given to the second argument
  // It takes an execution policy (here an implicit range as an int); 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, here a simple range 0<=i<10 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 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_redu                                                                    and
  // performs a redu
  // It takes an exe                                                                    value
  double sum = 0;
  Kokkos::parallel_r
    lsum += i;
  },sum);
  printf("Result %lf


  // A parallel_scan executes the body in parallel over the index space, here a simple range 0<=i<10 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 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 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, here a simple range 0<=i<10 and
  // performs a reduction over the values given to the second argument
  // It takes an execution policy (here an implicit range as an int); 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, here a simple range 0<=i<10 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 Policy

```
#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[]) {
  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 hav
  // On GPUs the Vector level is gua
  Kokkos::parallel_for(Kokkos::TeamP
    Kokkos::parallel_for(Kokkos::Tea
      Kokkos::parallel_for(Kokkos::T
        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[]) {
  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[]) {
  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 all
  // The nested levels
  //     - TeamThreadLoo
  // Note that the whol
  Kokkos::parallel_for(
    Kokkos::parallel_re
      lsum += i;
    },sum);
    if (thread.team_rar
      printf("Result %i
  });

  // 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();
}
```

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

Execution Policy

```
#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 Space

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

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

  // An execution policy can take an ExecutionSpace as a template parameter
  // What ExecutionSpaces are available depends on your compilation target
  // There is always a default ExecutionSpace defined – typically the 'highest' enabled one
  // Cuda execution with Lambdas from the host is preview feature of CUDA 7.5 (special Early Access for Sandia)
  Kokkos::parallel_for(Kokkos::RangePolicy<>(5,51),KOKKOS_LAMBDA (int i){
    printf("Hello from Default %i\n",i);
  });

  #ifdef KOKKOS_HAVE_SERIAL
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Serial>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from Serial %i\n",i);
  });
  #endif

  #ifdef KOKKOS_HAVE_OPENMP
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::OpenMP>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from OpenMP %i\n",i);
  });
  #endif

  #ifdef KOKKOS_HAVE_CUDA
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from Cuda %i\n",i);
  });
  #endif

  Kokkos::finalize();
}
```

Execution Space

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

int main(int argc, char* a
  Kokkos::initialize(argc,

  // An execution policy c
  // What ExecutionSpaces
  // There is always a def
  // Cuda execution with L                                                    ndia)
  Kokkos::parallel_for(Kok
    printf("Hello from Def
  });

  #ifdef KOKKOS_HAVE_SERIAL
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Serial>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from Serial %i\n",i);
  });
  #endif

  #ifdef KOKKOS_HAVE_OPENMP
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::OpenMP>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from OpenMP %i\n",i);
  });
  #endif

  #ifdef KOKKOS_HAVE_CUDA
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from Cuda %i\n",i);
  });
  #endif

  Kokkos::finalize();
}
```

ExecutionSpace:
- given as argument to ExecutionPolicy or as typedef in functor
- multiple ExecutionSpaces can be active at the same time (e.g. use OpenMP on CPUs + CUDA on GPU concurrently)
- If omitted use DefaultExecutionSpace

Execution Space

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

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

  // An execution policy can take an ExecutionSpace as a template parameter
  // What ExecutionSpaces are available depends on your compilation target
  // There is always a default ExecutionSpace defined – typically the 'highest' enabled one
  // Cuda execution with Lambdas from the host is preview feature of CUDA 7.5 (special Early Access for Sandia)
  Kokkos::parallel_for(Kokkos::RangePolicy<>(5,51),KOKKOS_LAMBDA (int i){
    printf("Hello from Default %i\n",i);
  });

  #ifdef KOKKOS_HAVE_SERIAL
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Serial>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from Serial %i\n",i);
  });
  #endif

  #ifdef KOKKOS_HAVE_OPENMP
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::OpenMP>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from OpenMP %i\n",i);
  });
  #endif

  #ifdef KOKKOS_HAVE_CUDA
  Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(5,51), KOKKOS_LAMBDA (int i){
    printf("Hello from Cuda %i\n",i);
  });
  #endif

  Kokkos::finalize();
}
```

The 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 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();
}
```

The 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 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 t                                  space can not
  //   access the default HostSpace direct
  // Note: the underlying allocation is no                                  mation is copied
  Kokkos::parallel_for(10,KOKKOS_LAMBDA(in
    // Read and write access to data comes
    a(i,0) = 1.0*i;  a(i,1) = 1.0*i*i;  a(
  });

  double sum = 0;
  Kokkos::parallel_reduce(10,KOKKOS_LAMBDA
    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 Layout, MemorySpace and MemoryTraits

The 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 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();
}
```

Memory Space

```
#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 Space

```
#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

  // Create
  // If the
  //   will
  host_view

  for(int i

  // 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();
}
```

MemorySpaces 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

Memory Space

```
#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 Space

```
#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, KO
    lsum += a(i,0)-a(i,1)+a(i,2)
  },sum);

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

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

Memory Space

```
#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 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 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 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_po
  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 Layout
  Kokkos::parallel_for(Kokkos::TeamPolicy<
                       KOKKOS_LAMBDA (Kokk
     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();
}
```

MemoryLayout:
- 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 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 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 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 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();
}
```

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

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();
}
```

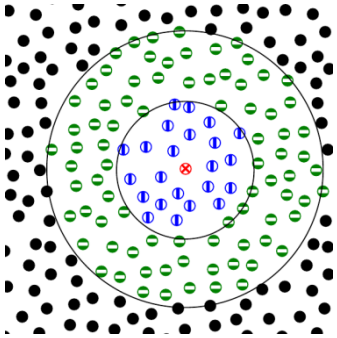
Performance Impact: Layout and Memory Traits

- Molecular dynamics computational kernel in miniMD
- Simple Lennard Jones force model:

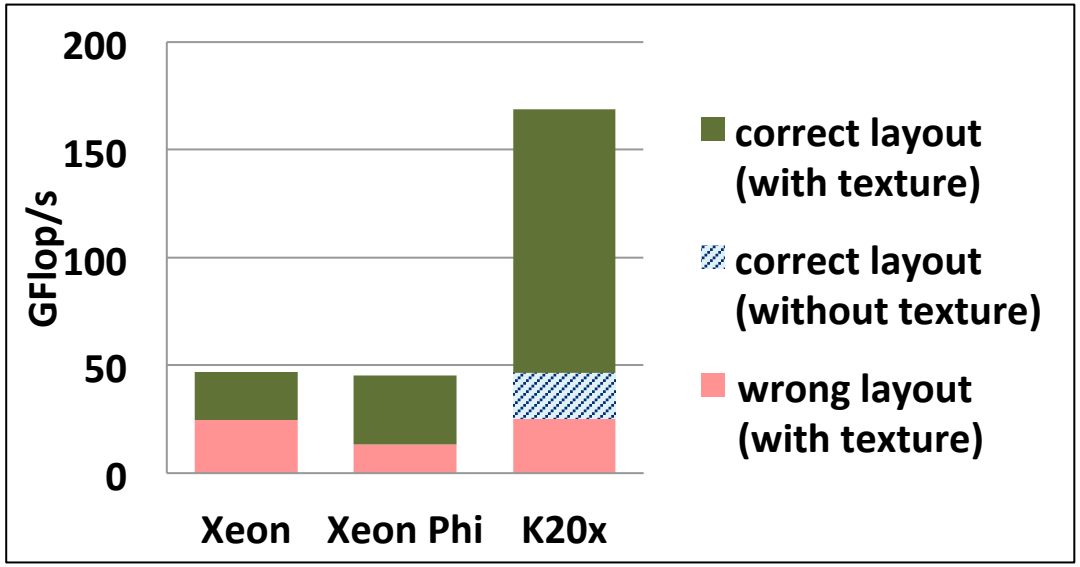
$$F_i = \sum_{j,\, r_{ij} < r_{cut}} 6\varepsilon \left[\left(\frac{\varsigma}{r_{ij}}\right)^7 - 2\left(\frac{\varsigma}{r_{ij}}\right)^{13} \right]$$

- Atom neighbor list to avoid N^2 computations

```
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj);
  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**



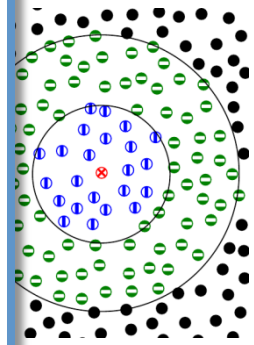
Performance Impact: Layout and Memory Traits

- Molecular dynamics computational kernel in miniMD
- Simple Lennard Jones force model:

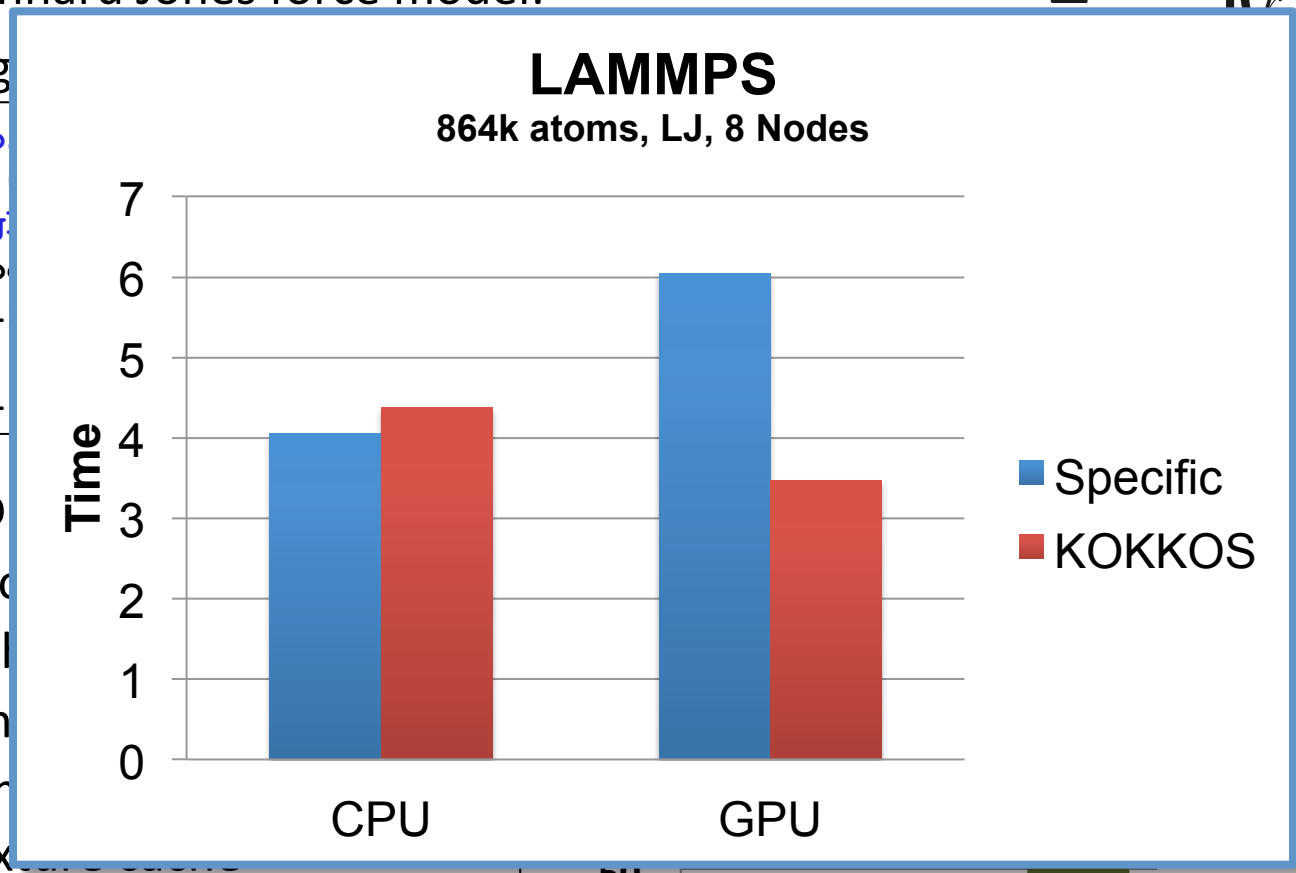
$$F_i = \sum 6\varepsilon\left[\left(\frac{\varsigma}{r_{ij}}\right)^{7} - 2\left(\frac{\varsigma}{r_{ij}}\right)^{13}\right]$$

- Atom neig

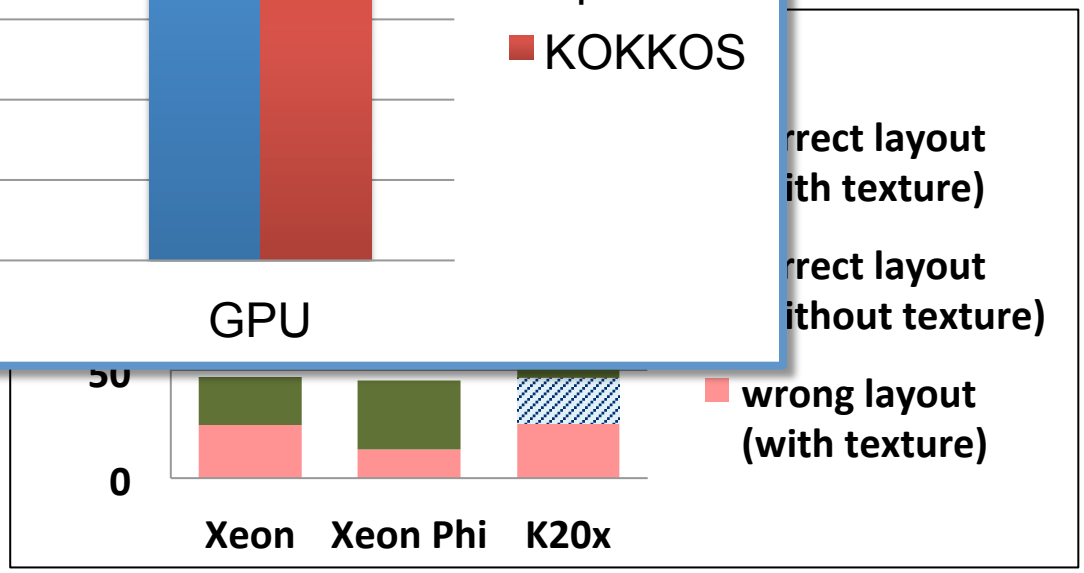
```
pos_i = po
for( jj =
  j = neig
  r_ij = p
  if (|r_i
}
f(i) = f_i
```



- **Test Pro**
  - 864k ato
  - 2D neigl
  - Differen
  - Random
    GPU tex



- **Large performance loss with wrong array layout**

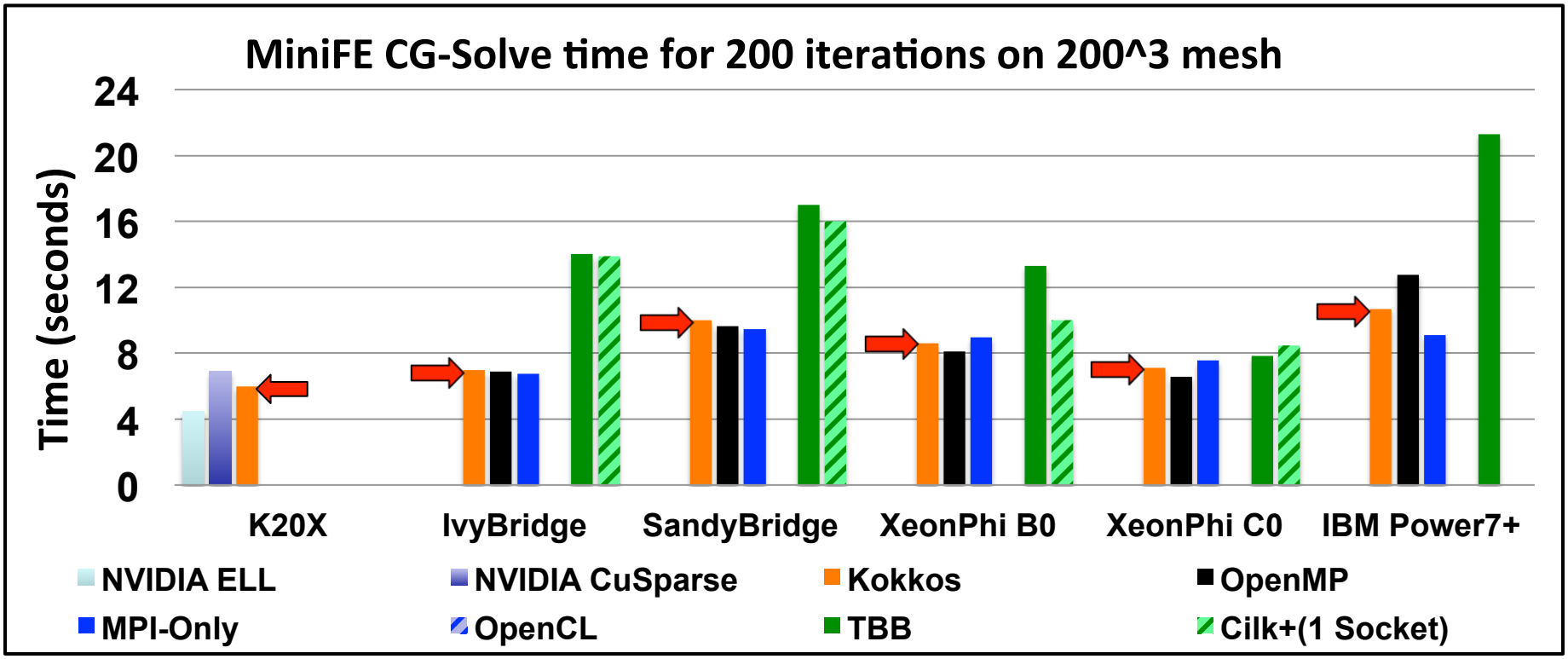


Comparison with other Programing Models

MiniFE: finite element linear system iterative solver mini-app

Compare to versions specialized for programming models

Running on hardware testbeds



Wrap Up

Features which were not discussed:

- Subviews: get range and index based subviews of views
- 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):

- Move to GitHub (in March)
- 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)

Some Projects leveraging Kokkos

Trilinos / Tpetra : foundational data structures and kernels for sparse linear algebra; Mark Hoemmen, Christian Trott

Trilinos / Stokhos : accelerating embedded UQ ; Eric Phipps

LAMMPS : molecular dynamics; Christian Trott, Stan Moore

Albany : finite elements applied to ice sheets, atmosphere, mechanics, quantum devices; Andy Salinger, Irina Demeshko

ASCR Multiphysics MHD; Roger Pawlowski and Eric Cyr

FASTMath SciDAC / Graph Algorithms : Siva Rajamanickam

Zoltan / Graph Coloring : fast threaded graph coloring to identify independent sets of work for task parallelism; Erik Boman, Siva Rajamanickam

EMPRESS and miniPIC : particle-in-cell ; Matt Bettencourt

miniAero : CFD finite element mini-application ; Ken Franco

miniContact : contact detection for solid mechanics ; Glen Hansen

SHIFT @ ORNL; Steve Hamilton

Kokkos SNL/LDRD : directed acyclic graph of internally data parallel tasks

Uintah : A Unified Heterogeneous Task Scheduling and Runtime System, switch to Kokkos for local data parallelism

Sandia National Laboratories

Exceptional service in the national interest

Questions and further discussion: crtrott@sandia.gov