SAND2014-15145PE

LAMMPS-Kokkos: The Tutorial *alpha*

Christian Trott

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.

an introduction to most API features of Kokkos

- code based with examples being valid Kokkos programs
- incrementally increasing complexity for a low entry barrier

What this tutorial is NOT:

- an introduction to parallel programming
- a presentation of Kokkos features
- a performance comparison of Kokkos with other approaches

What you should know:

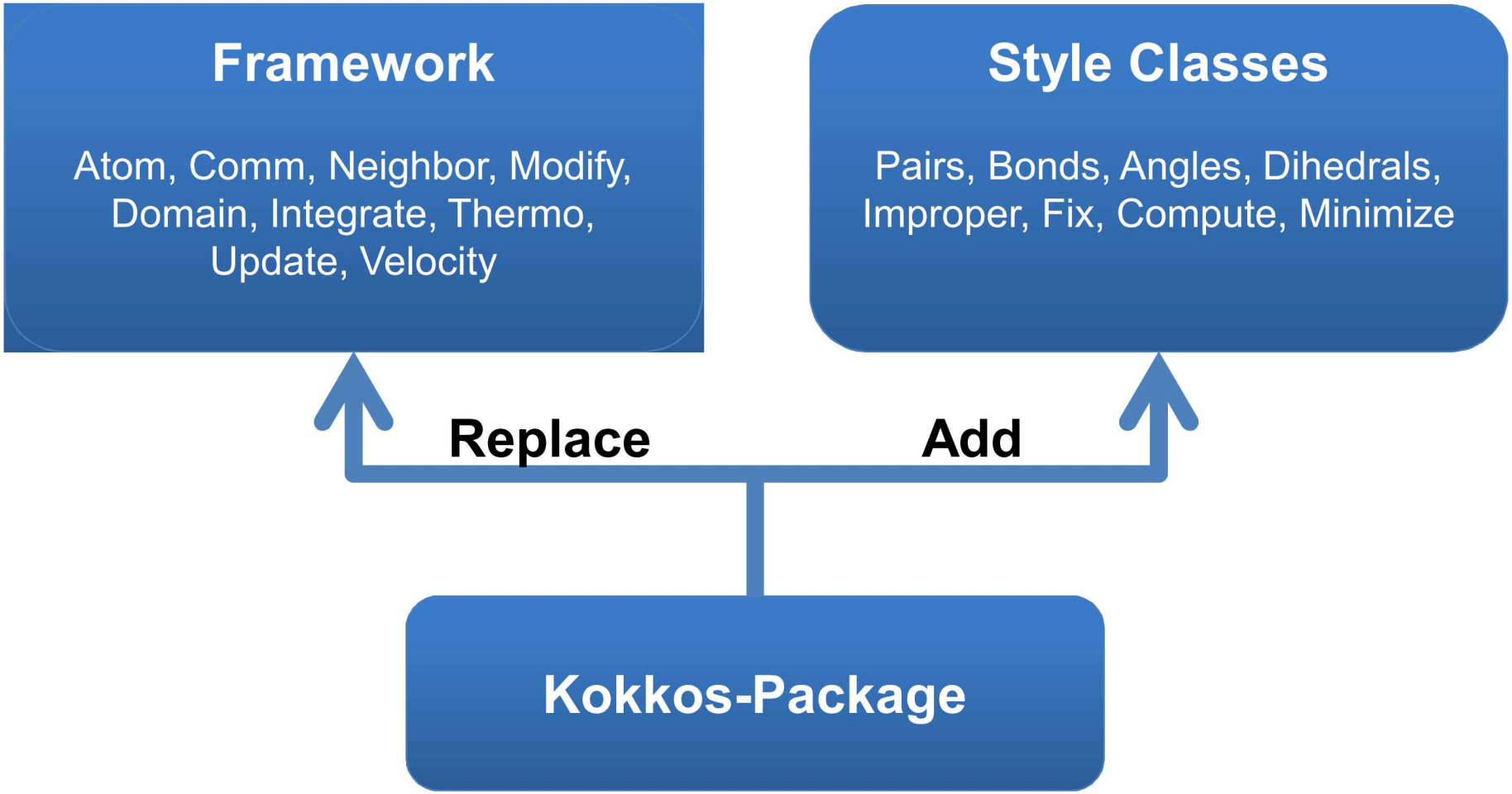
- some C++
- general parallel programming concepts

Where the code can be found:

- Trilinos/packages/kokkos/example/tutorial

Sandia National Laboratories

LAMMPS



Kokkos Styles are compiled for both ExecutionSpaces

- fix 1 all nve/kk/device
- fix 1 all nve/kk/host

Sandia National Laboratories

Device Managemnt

Communication can run on Host or Device

- package kokkos comm/forward device comm/exchange host

There is a enum corresponding to the DeviceTypes:

- `enum ExecutionSpace{Host,Device};`

Data Management - Types

Sandia National Laboratories

Type directory: kokkos_type.h

- Provides typedefs for all array types for host and device

Use different scalars for different arrays to allow mixed precisi

- Provides typedefs for memory traits variants

```
template <class DeviceType>
struct ArrayTypes;

template <>
struct ArrayTypes<LMPDeviceType> {

//2d F_FLOAT array n*m
typedef Kokkos::DualView<F_FLOAT**, Kokkos::LayoutRight, LMPDeviceType> tdual_ffloat_2d;
typedef tdual_ffloat_2d::t_dev t_ffloat_2d;
typedef tdual_ffloat_2d::t_dev_const t_ffloat_2d_const;
typedef tdual_ffloat_2d::t_dev_um t_ffloat_2d_um;
typedef tdual_ffloat_2d::t_dev_const_um t_ffloat_2d_const_um;
typedef tdual_ffloat_2d::t_dev_const_randomread t_ffloat_2d_randomread;

//2d F_FLOAT array n*3
typedef Kokkos::DualView<F_FLOAT*[3], Kokkos::LayoutRight, LMPDeviceType> tdual_f_array;
typedef tdual_f_array::t_dev t_f_array;
typedef tdual_f_array::t_dev_const t_f_array_const;
typedef tdual_f_array::t_dev_um t_f_array_um;
typedef tdual_f_array::t_dev_const_um t_f_array_const_um;
typedef tdual_f_array::t_dev_const_randomread t_f_array_randomread;

}
```

Data Management - Instances

sistent global data structures (members of Atom class) are Dual

- Wrap classic data structures around Kokkos-Allocation
  - Legacy data structure views same data as Host View
  - *Limitation:* Kokkos needs to use legacy data-layout (LayoutRight)
- *Convention:* prefix DualView with: '**k_**', Device View with: '**d_**', Host View with: '**h_**'

AtomKokkos class (atomKK):

```
class AtomKokkos : public Atom {
 public:
  // DAT = ArrayTypes<LMPDeviceType>
  DAT::tdual_int_1d k_tag, k_type, k_mask, k_molecule; // Various n of int
  DAT::tdual_tagint_1d k_image; // Image n of tagint
  DAT::tdual_x_array k_x; // Positions n*3 of X_FLOAT
  DAT::tdual_v_array k_v; // Velocities n*3 of V_FLOAT
  DAT::tdual_f_array k_f; // Forces n*3 of F_FLOAT

  DAT::tdual_float_1d k_mass; // Mass m of LMP_FLOAT
}
```

Data Management - Allocation

Allocation:

```
memory->grow_kokkos(atomKK->k_x,atomKK->x,nmax,3,"atom:x");
```

Internals in memory_kokkos.h:

```
template <typename TYPE>
TYPE create_kokkos(TYPE &data, typename TYPE::value_type **&array,
                   int n1, const char *name)
{
  data = TYPE(std::string(name),n1);
  bigint nbytes = ((bigint) sizeof(typename TYPE::value_type *)) * n1;
  array = (typename TYPE::value_type **) smalloc(nbytes,name);

  for (int i = 0; i < n1; i++)
    array[i] = &data.h_view(i,0);

  return data;
}
```

Data Management - Synchronization

Sandia National Laboratories

Styles have data masks to identify read and write access

The Framework will call Sync and Modify of DualViews

Modify:

```
for (int i = 0; i < nfix; i++) {
  atomKK->sync(fix[i]->execution_space,fix[i]->datamask_read);
  atomKK->modified(fix[i]->execution_space,fix[i]->datamask_modify);
  fix[i]->setup(vflag);
}
```

Verlet_Kokkos:

```
if (pair_compute_flag) {
  atomKK->sync(force->pair->execution_space,force->pair->datamask_read);
  atomKK->modified(force->pair->execution_space,force->pair->datamask_modify);
  force->pair->compute(eflag,vflag);
  timer->stamp(TIME_PAIR);
}
```

If a style class has multiple parallel functions optimize by leavin datamasks empty and call atomKK->sync explicitly.

Parallel Dispatch – Two Approaches

Wrapper Functor

- easier to implement
  - less code change
  - used for fix/nve

Porting Steps:

1. move loop body into “item” functions
2. create functor with a copy of the class as member
3. call “item” function of class from operator()
4. use Kokkos Views for internal and global data
5. add modify/sync calls where needed
6. add cleanup_copy function

Stand alone Functor

potentially more performance
used for neighbor list construction

Porting steps

1. use Kokkos Views for internal and global data
2. create functor
3. make members in functor for all variables accessed in function
4. copy loop body into operator()
5. copy all functions called in loop body to functor

Parallel Dispatch – Fix NVE

```
template<class DeviceType>
class FixNVEKokkos : public FixNVE {
 public:
  void cleanup_copy(); // Eliminate all direct allocations (e.g. label)
  void initial_integrate(int); // One of the original compute functions

  KOKKOS_INLINE_FUNCTION
  void initial_integrate_item(int) const; // Loop body of former initial_integrate
}

template <class DeviceType, int RMass>
struct FixNVEKokkosInitialIntegrateFunctor  {
  typedef DeviceType  device_type ;
  FixNVEKokkos<DeviceType> c;

  // Get a copy of the original class
  FixNVEKokkosInitialIntegrateFunctor(FixNVEKokkos<DeviceType>* c_ptr):
  c(*c_ptr) {c.cleanup_copy();};

  // Call the correct item function. This is templated on Rmass,
  // so that the conditional is compile time and does not hinder vectorization
  KOKKOS_INLINE_FUNCTION
  void operator()(const int i) const {
    if (RMass) c.initial_integrate_rmass_item(i);
    else c.initial_integrate_item(i);
  }
};
```

Parallel Dispatch – Fix NVE continued

```
template<class DeviceType>
FixNVEKokkos<DeviceType>::FixNVEKokkos(LAMMPS *lmp, int narg, char **arg) :
  FixNVE(lmp, narg, arg)
{
  atomKK = (AtomKokkos *) atom;
  // Set runtime execution_space identifier
  execution_space = ExecutionSpaceFromDevice<DeviceType>::space;

  // Set data masks
  datamask_read = X_MASK | V_MASK | F_MASK | MASK_MASK | RMASS_MASK | TYPE_MASK;
  datamask_modify = X_MASK | V_MASK ;
}

template<class DeviceType>
void FixNVEKokkos<DeviceType>::initial_integrate_item(int i) const
{
  if (mask[i] & groupbit) {
    const double dtfm = dtf / mass[type[i]];
    v(i,0) += dtfm * f(i,0);
    v(i,1) += dtfm * f(i,1);
    v(i,2) += dtfm * f(i,2);
    x(i,0) += dtv * v(i,0);
    x(i,1) += dtv * v(i,1);
    x(i,2) += dtv * v(i,2);
  }
}
```

Parallel Dispatch – Fix NVE continued (ii)

```
template<class DeviceType>
void FixNVEKokkos<DeviceType>::initial_integrate(int vflag)
{
  // Syncs are performed by modify and are not needed here
  // atomKK->sync(execution_space,datamask_read);
  // atomKK->modified(execution_space,datamask_modify);

  // Update data references
  x = atomKK->k_x.view<DeviceType>();
  v = atomKK->k_v.view<DeviceType>();
  f = atomKK->k_f.view<DeviceType>();
  rmass = atomKK->rmass;
  mass = atomKK->k_mass.view<DeviceType>();
  type = atomKK->k_type.view<DeviceType>();
  mask = atomKK->k_mask.view<DeviceType>();
  int nlocal = atomKK->nlocal;
  if (igroup == atomKK->firstgroup) nlocal = atomKK->nfirst;

  if (rmass) {  // Call functor, template expand rmass conditional
    FixNVEKokkosInitialIntegrateFunctor<DeviceType,1> functor(this);
    Kokkos::parallel_for(nlocal,functor);
  } else {
    FixNVEKokkosInitialIntegrateFunctor<DeviceType,0> functor(this);
    Kokkos::parallel_for(nlocal,functor);
  }
  DeviceType::fence();
}
```

Pair Forces

- Call pair_compute function templated on the pair force style
- Pair_compute provides iteration schemes for different neighbor
  - much better extensibility: add new neighbor styles / iteration schemes once
  - Pair force provides functions for actual physics code
  - take distance and atom types, return force or energy
- Two options to handle force parameters: on stack or in heap
  - significant performance difference in particular on GPU
  - stack can only handle limited amount: compile time setting (Default 12 types)

```
template<class DeviceType>
template<bool STACKPARAMS, class Specialisation>
KOKKOS_INLINE_FUNCTION
F_FLOAT PairLJCutKokkos<DeviceType>::
compute_fpair(const F_FLOAT& rsq, const int& i, const int&j,
              const int& itype, const int& jtype) const {
  const F_FLOAT r2inv = 1.0/rsq;
  const F_FLOAT r6inv = r2inv*r2inv*r2inv;

  const F_FLOAT forcelj = r6inv *
    ((STACKPARAMS?m_params[itype][jtype].lj1:params(itype,jtype).lj1)*r6inv -
     (STACKPARAMS?m_params[itype][jtype].lj2:params(itype,jtype).lj2));
  return forcelj*r2inv;
}
```

```
friend class PairComputeFunctor<PairLJCutKokkos,HALFTHREAD,true>;
friend class PairComputeFunctor<PairLJCutKokkos,N2,true>;
friend class PairComputeFunctor<PairLJCutKokkos,FULLCLUSTER,true >;
friend class PairComputeFunctor<PairLJCutKokkos,FULL,false>;
friend class PairComputeFunctor<PairLJCutKokkos,HALF,false>;
friend class PairComputeFunctor<PairLJCutKokkos,HALFTHREAD,false>;
friend class PairComputeFunctor<PairLJCutKokkos,N2,false>;
friend class PairComputeFunctor<PairLJCutKokkos,FULLCLUSTER,false >;
friend EV_FLOAT pair_compute<PairLJCutKokkos,void>
      (PairLJCutKokkos*,NeighListKokkos<DeviceType>*);
```

Pair Force Dispatch

Pair styles don't need to support all of them

Pair Force Dispatch

```
atomKK->sync(execution_space,datamask_read);
k_cutsq.template sync<DeviceType>();
k_params.template sync<DeviceType>();
if (eflag || vflag) atomKK->modified(execution_space,datamask_modify);
else atomKK->modified(execution_space,F_MASK);

x = atomKK->k_x.view<DeviceType>();
c_x = atomKK->k_x.view<DeviceType>();
f = atomKK->k_f.view<DeviceType>();
type = atomKK->k_type.view<DeviceType>();
nlocal = atom->nlocal;
nall = atom->nlocal + atom->nghost;
special_lj[0] = force->special_lj[0];
special_lj[1] = force->special_lj[1];
special_lj[2] = force->special_lj[2];
special_lj[3] = force->special_lj[3];
newton_pair = force->newton_pair;

// loop over neighbors of my atoms
EV_FLOAT ev = pair_compute<PairLJCutKokkos<DeviceType>,void >(this,(NeighListKokkos<DeviceType>*)list);

DeviceType::fence();

if (eflag) eng_vdwl += ev.evdwl;
if (vflag_global) {
  virial[0] += ev.v[0];
  virial[1] += ev.v[1];
  virial[2] += ev.v[2];
  virial[3] += ev.v[3];
  virial[4] += ev.v[4];
  virial[5] += ev.v[5];
}
```

Sandia National Laboratories

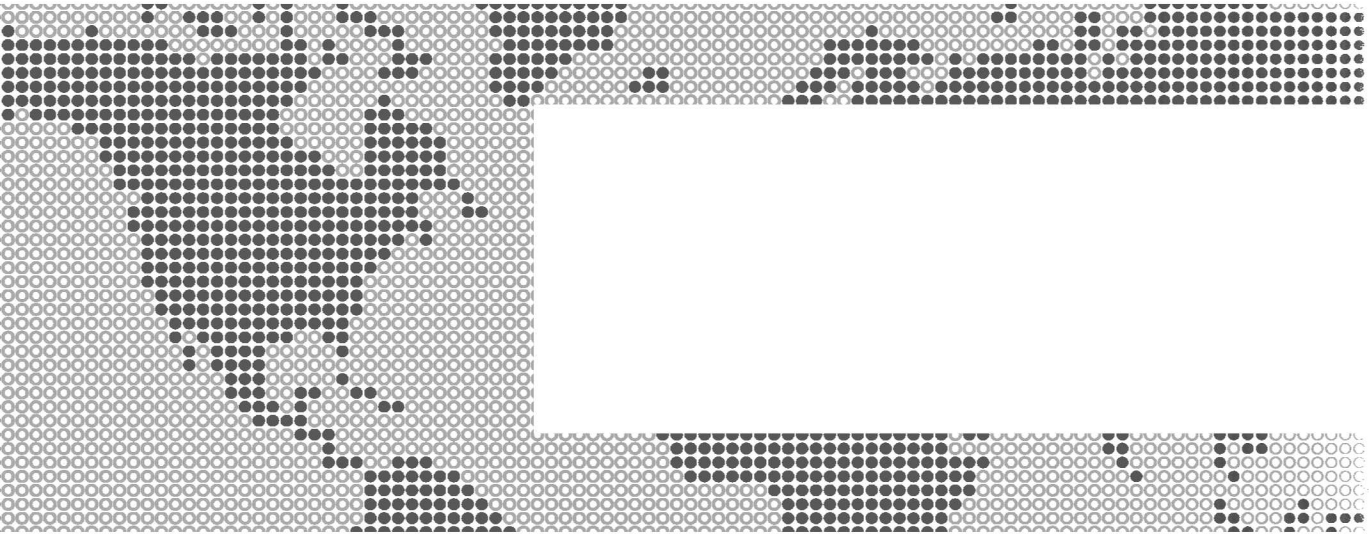
Pair Force Dispatch

```
template<class PairStyle, class Specialisation>
EV_FLOAT pair_compute (PairStyle* fpair,
                       NeighListKokkos<typename PairStyle::device_type>* list) {
  EV_FLOAT ev;
  if(fpair->atom->ntypes > MAX_TYPES_STACKPARAMS) {

    if (fpair->neighflag == FULL) {
      PairComputeFunctor<PairStyle,FULL,false,Specialisation >
        ff(fpair, list);
      if (fpair->eflag || fpair->vflag) Kokkos::parallel_reduce(list->inum,ff,ev);
      else Kokkos::parallel_for(list->inum,ff);

    } else if (fpair->neighflag == HALFTHREAD) {
      PairComputeFunctor<PairStyle,HALFTHREAD,false,Specialisation >
        ff(fpair, list);
      if (fpair->eflag || fpair->vflag) Kokkos::parallel_reduce(list->inum,ff,ev);
      else Kokkos::parallel_for(list->inum,ff);

    } else if (fpair->neighflag == HALF) {
      PairComputeFunctor<PairStyle,HALF,false,Specialisation >
        ff(fpair, list);
      if (fpair->eflag || fpair->vflag) Kokkos::parallel_reduce(list->inum,ff,ev);
      else Kokkos::parallel_for(list->inum,ff);
```



Questions and further discussion: crtrott@sandia.gov