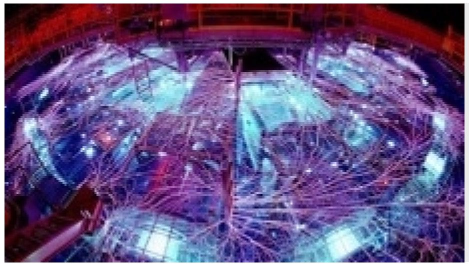
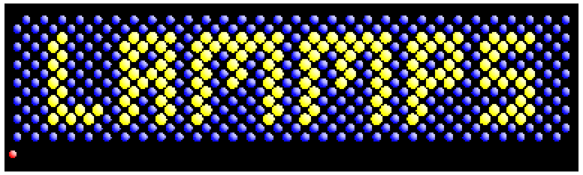
Exceptional service in the national interest

Sandia National Laboratories



Center for Computing Research

Kokkos in LAMMPS

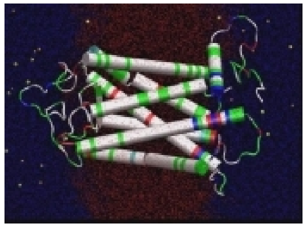
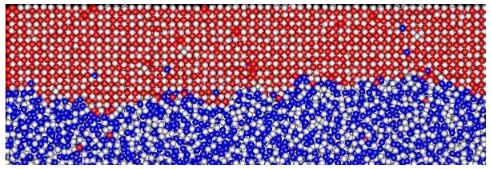
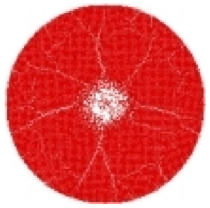


Stan Moore

2024 HPC LAMMPS Master Class Workshop

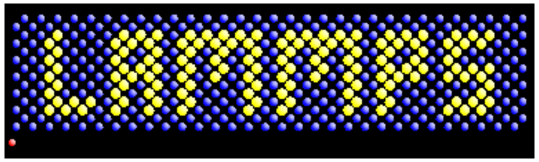
LAMMPS

- Large-scale Atomic/Molecular Massively Parallel Simulator
- https://lammps.org
  - Open source, C++ molecular dynamics code
  - Bio, materials, mesoscale

  - Particle simulator at varying length and time scales
    - Electrons → atomistic → coarse-grained → continuum
  - Spatial-decomposition of simulation domain for parallelism
  - Energy minimization, dynamics, non-equilibrium MD
  - GPU and OpenMP enhanced
  - Can be coupled to other scales: QM, kMC, FE, CFD, …

Accelerator Packages in LAMMPS



- **Vanilla C++ version**

<u>**Accelerator packages:**</u>

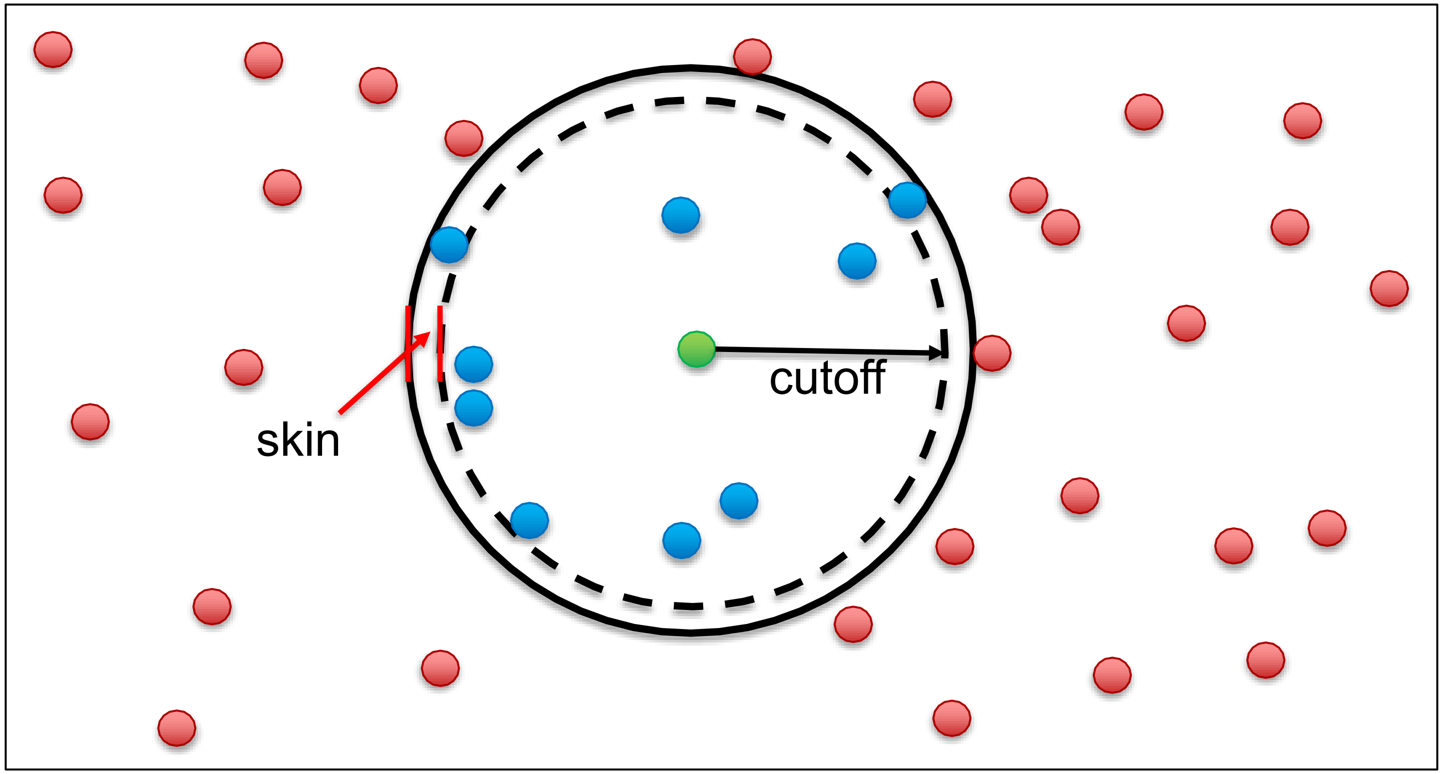
- **OpenMP Package:** native OpenMP threading
- **INTEL Package**: native OpenMP threading, enhanced SIMD vectorization, uses hardware intrinsics, fast on CPUs but very complex code
- **GPU Package**: native CUDA and OpenCL support, only runs a few kernels (e.g. *pair* force calculation) on GPU, needs multiple MPI ranks per GPU to parallelize CPU calculations
- **KOKKOS Package**: implements Kokkos library abstractions, tries to run everything on device, supports CUDA (NVIDIA GPUs), HIP (AMD GPUs), SYCL (INTEL GPUs), and OpenMP (CPU) threading backends

KOKKOS Package Options

- Many options that affect KOKKOS package performance can be controlled via the *package* command in the input script
- See https://docs.lammps.org/package.html
- Package commands can also be invoked on the command line (my preference): `–pk kokkos newton on neigh half`
- Default values often (but not always) optimal
- Many of these options are commonly seen in KOKKOS package code

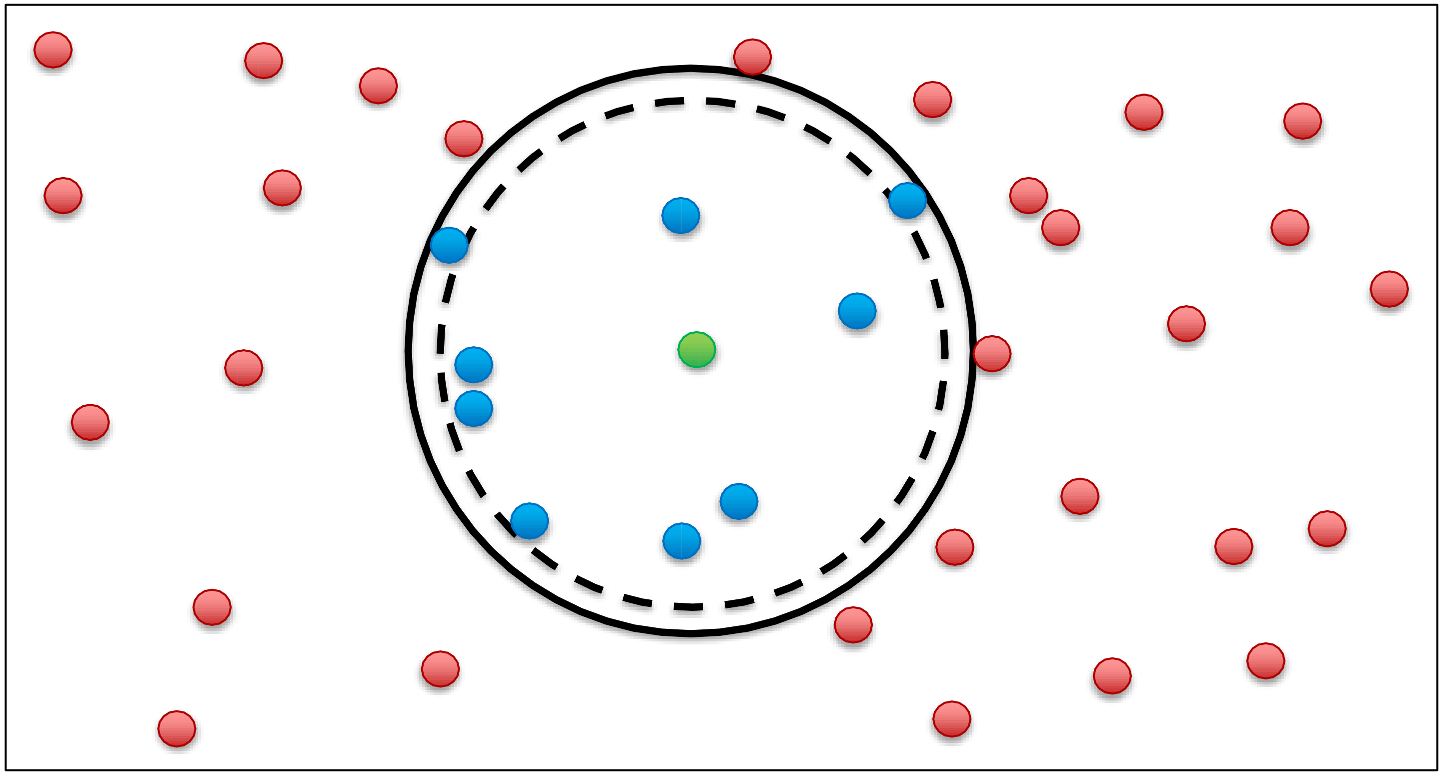
Neighbor Lists

- Neighbor lists are a list of neighboring atoms within the interaction cutoff + skin for each central atom
- Extra skin allows lists to be built less often



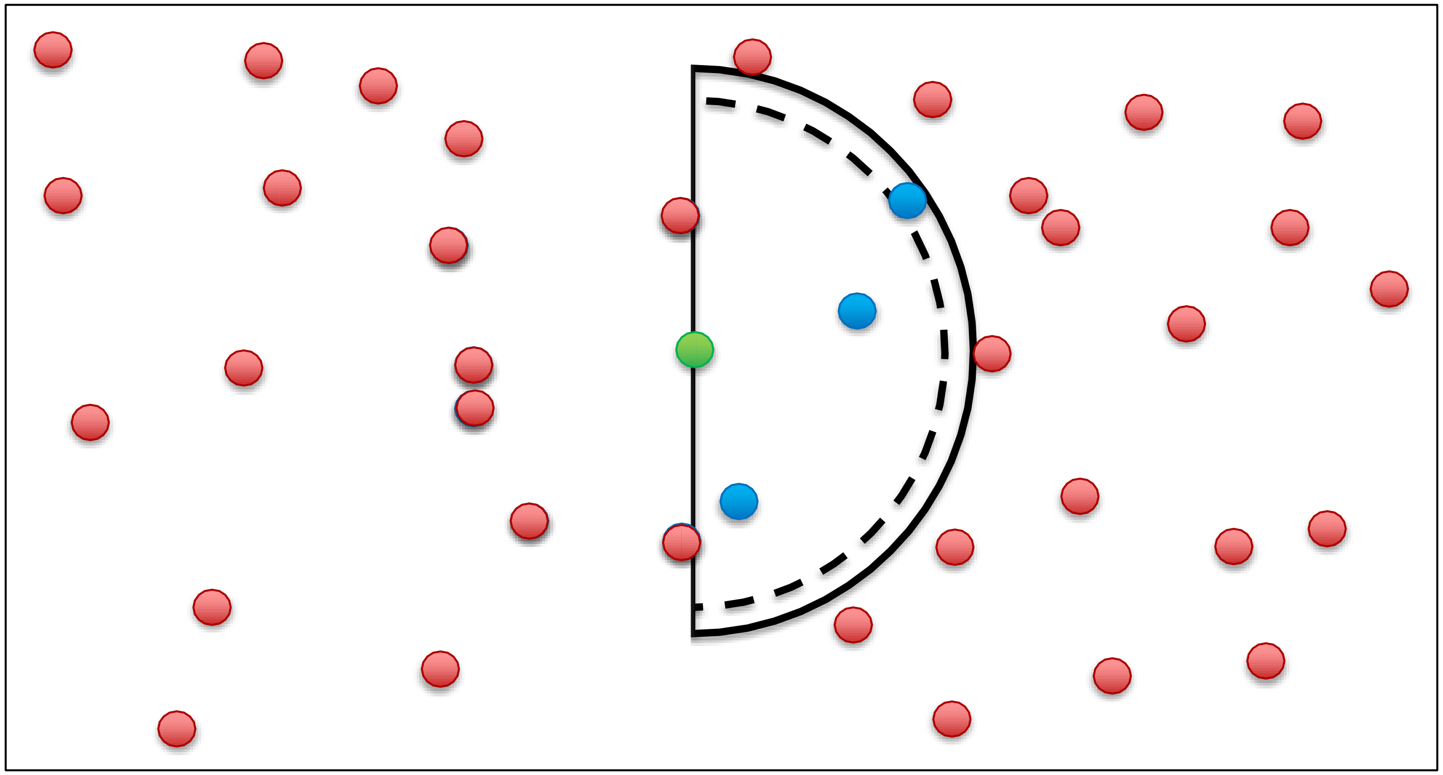
Full Neighbor List

- Each pair stored twice which doubles computation for pairwise potentials but reduces communication, doesn't require atomic operations for thread safety (may be faster on GPUs for pairwise potentials, but often not for manybody)



Half Neighbor List

- With newton flag on, each pair is stored only once (usually better for CPUs), requires atomic operations for thread-safety
- Many newer GPUs have fast FP64 hardware atomics, half list can be better for manybody potentials

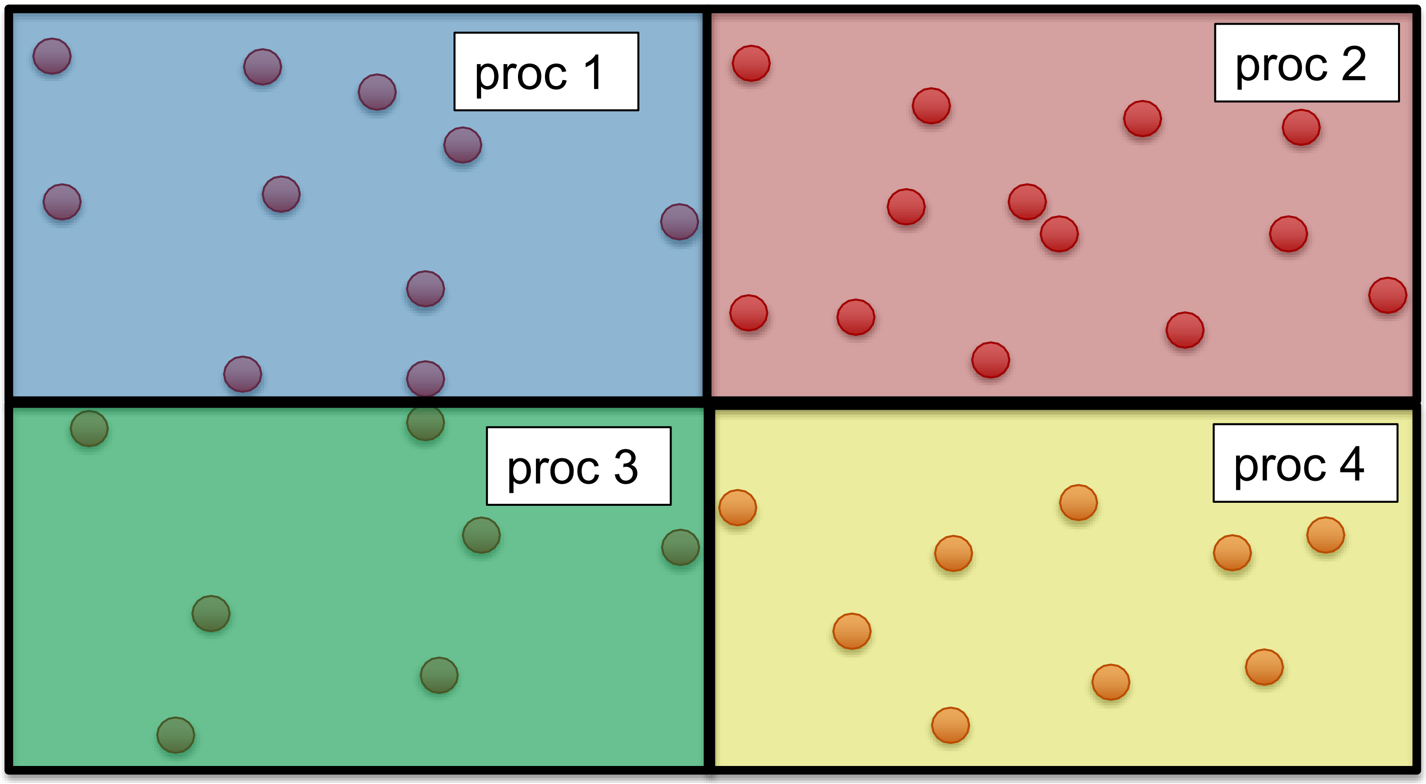


Newton Option

- Newton flag to *off* means that if two interacting atoms are on different processors, **both processors compute their interaction** and the resulting force information is not communicated
- Setting the newton flag to *on* saves computation but increases communication
- Performance depends on problem size, force cutoff lengths, a machine's compute/communication ratio, and how many processors are being used
- Newton off almost always better for GPUs, newton on almost always better for CPUs

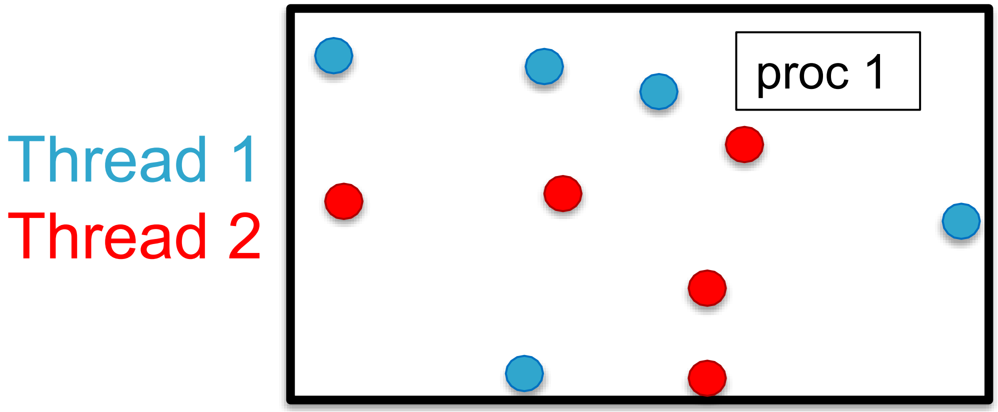
MPI Parallelization Approach

- Domain decomposition: each processor owns a portion of the simulation domain and atoms therein



Multithreading (e.g. OpenMP, CUDA)

- Used on top of MPI domain decomposition
- Each thread processes a subset atoms in a processor's subdomain
- Threads run concurrently, **no guarantee of order**



Threading in LAMMPS

- Typically thread over owned atoms (`atom->nlocal`)
- Additionally can thread over neighbors to expose more parallelism, but can have overheads. Default for 16k or less atoms with simple pair-wise potentials
- Threading over neighbors also used for expensive machine learning potentials with low atom counts/GPU. SNAP threads over atoms, neighbors, and bispectrum
- Can also manually flatten multiple loops into a single loop to expose more parallelism
- Typically run Kokkos LAMMPS with 1 MPI rank per GPU (leaves many CPU cores idle, but often most of the FLOPs are on the GPUs)
- If significant kernels are not running on the GPU, then using multiple MPI ranks per GPU can help parallelize host calculations (need to enable *CUDA MPS* on NVIDIA GPUs)

Execution Spaces

- With GPUs, *LMPHostType* execution space = CPU backed (Serial or OpenMP), *LMPDeviceType* = GPU backend
- In LAMMPS, all Kokkos pair styles are templated on *DeviceType*: can be either host or device
- Compiler creates two different versions of the code, one for CPU backend and one for GPU backend
- Users can choose at runtime which version to use via the LAMMPS *suffix* command (see https://docs.lammps.org/suffix.html)
- Generally must not use *LMPDeviceType* directly, use *DeviceType* template parameter instead, since user could be running a style on host

KOKKOS View Structs

- `ArrayTypes` struct defined in src/KOKKOS/kokkos_type.h, contains nearly all needed view types
- `DAT::` refers to `ArrayTypes<LMPDeviceType>`
  `HAT::` refers to `ArrayTypes<LMPHostType>`
- `AT::` refers to `ArrayTypes<DeviceType>`, must be defined in each class or functor
- `DAT::tdual_*` refers to DualView
- `AT::t_*` refers to device view, `HAT::t_*` refers to host
  Must not use `DAT::t_*` because it hard codes device space (some rare exceptions, usually in core code)
- Can define new view types if needed, see https://github.com/lammps/lammps/blob/develop/src/KOKKOS/pair_eam_kokkos.h#L171-L177

KOKKOS View Names in LAMMPS

- k_[view_name] means DualView
- d_ [view_name] means device view
- h_ [view_name] means host view
- l_ [view_name] means local copy of device view (for lambdas)

Allocating Memory

- Not all styles in LAMMPS are ported to Kokkos, sometimes need to maintain compatibility with legacy data structures
- Alias Kokkos DualViews to LAMMPS legacy data using `memoryKK->create_kokkos()`
- Already done for all atom data (position, force, velocity, etc.)
- Constrains `LayoutRight`, may not give optimal performance for GPUs, so do not alias unless necessary
- For class variable not used by legacy code, use `memKK::realloc_kokkos()` instead, avoids initialization overhead and reduces memory use, does not preserve existing values
- Use `memoryKK->grow_kokkos()` or `view.resize()` to preserve values

Data Transfer

- `DualView` data transfer controlled by atom masks : `X_MASK` (positions), `F_MASK` (forces), `V_MASK` (velocities), `ALL_MASK`, `EMPTY_MASK`, etc.
- Always use `atomKK->sync` and `modified` if possible (unless custom class view), e.g. `atomKK->sync(space,X_MASK|F_MASK)`
- All pair styles define two variables: `datamask_read` and `datamask_modify`
- If only one parallel kernel, fine to set `datamask_read` and `datamask_modify`, e.g. `atomkk->sync(space,datamask_read)`
- Otherwise better to define these as EMPTY_MASK and use `atomKK->sync(space,X_MASK|F_MASK)` instead

Memory

- If a LAMMPS style is not ported to Kokkos it will run on CPU in serial and require data transfer every time it is invoked: consider porting to Kokkos to improve performance
- In LAMMPS often use `Kokkos::DualView sync` and `modify` on Device and Host to transfer data (not for atomKK)
- Also see `Kokkos::deep_copy` and `Kokkos::create_mirror_view` in some cases instead (not for atomKK)

Copymode

- When using tags in class or KOKKOS_CLASS_LAMBDA, we pass `*this` as the functor, creates a copy of the class
- When the class copy goes out of scope at the end of the parallel region, the class destructors gets called, including the parent destructor
- This deallocates class memory prematurely, can lead to a segmentation fault or other memory issues
- Need to protect all destructors (Kokkos class and all parent classes) with `if (copymode) return;`
- Need to set `copymode = 1` before Kokkos parallel region, `copymode = 0` at end

Templating for Performance

- Templating parallel regions can reduce runtime overhead of conditional statements
- `if (TEMPLATE_PARAM)`: compiler should move conditional check to compile time and create two separate code branches
- If not, can use C++17 `if constexpr(TEMPLATE_PARAM)` as hammer on the compiler to force it
- Can template functors and tags, not sure how to template lambdas

Initialization and Overheads

- Typically assume that a user is going to run a simulation for an hour or longer
- Therefore, don't typically try to port initialization and setup kernels to use Kokkos (likely won't run efficiently on GPUs anyway)
- Same for infrequent operations on CPU like load balancing
- Keep in mind that every thermo output step requires copying significant amount of data from GPU to CPU (all atom data goes to host since we don't know what will be used)
- Keep thermo output to a minimum (e.g. only 1000 steps or less frequent)
- Similar overhead for fixes that collect data like `fix ave/atom`, better to call every 10 timesteps than every step

Polymorphism

- Nearly all KOKKOS package classes are derived from non-Kokkos parent classes using polymorphism (overriding virtual functions)
- Greatly reduces code duplication: initialization/setup code is run on host and often identical for Kokkos styles vs parent
- When using tags, all scalars in class stack memory get copied and can be used in device code. Pointers in class need to be changed to views
- Any scalar references to host memory (e.g. `force->boltz)` need to be copied into class variables
- Typically cannot use virtual multiple inheritance for Kokkos device code on GPUs, must duplicate code (e.g. see pair eam/kk vs eam/alloy/kk)

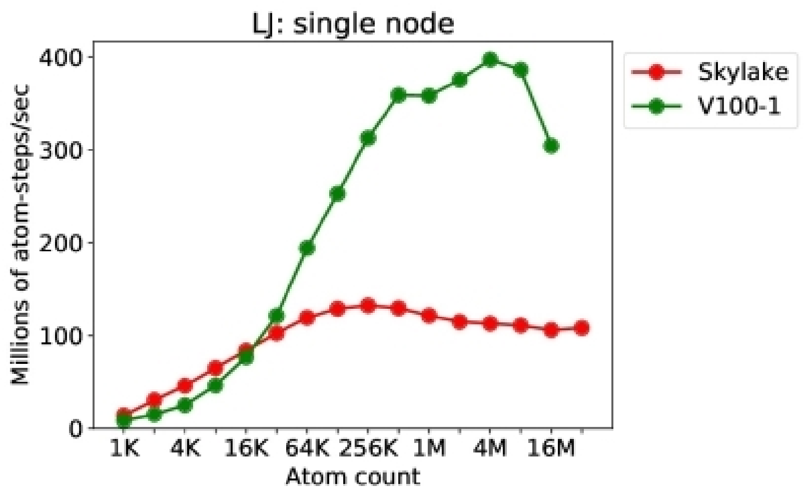
Parallel Reductions

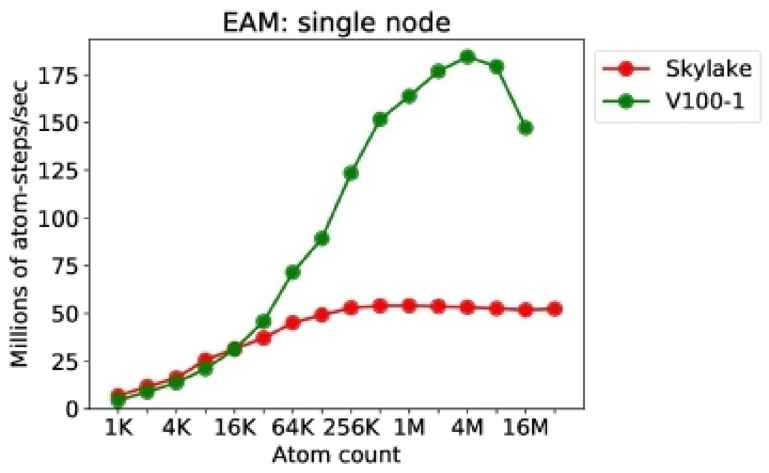
- For multiple values, must create custom struct with multiple members. See https://github.com/lammps/lammps/blob/develop/src/KOKKOS/compute_temp_kokkos.h#L41-L57
- Some built-in Kokkos reducer types exist (e.g. compute min or max of values instead of sum), see https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/Custom-Reductions-Built-In-Reducers.html
- Can also have custom `init()` and `join()` functions
- With templated tags in a class, can only have a single reduction type. Need to use functors if have multiple `parallel_reduce` with different reduction types (e.g. integer and double)

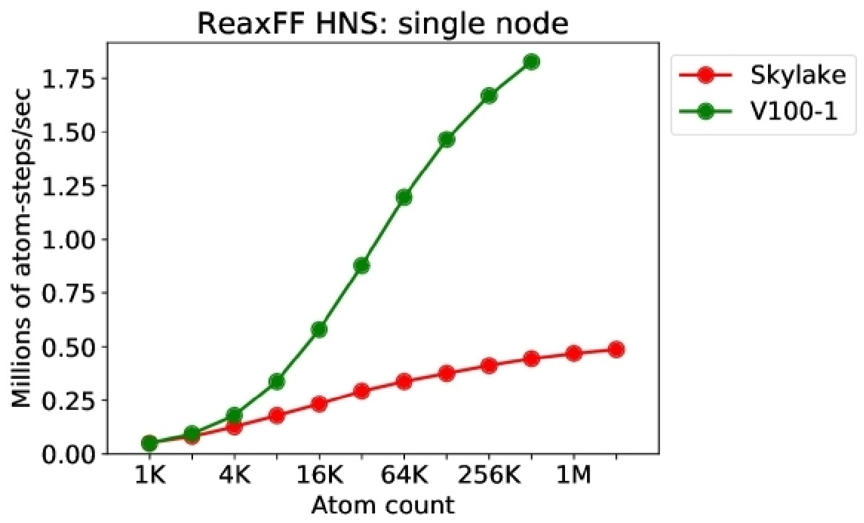
How to Optimize GPU Performance

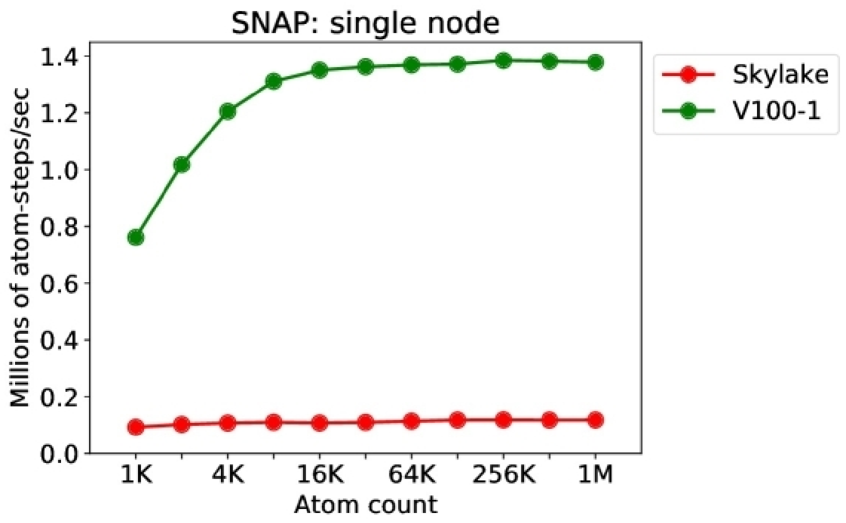
- Saturate GPU threads (increase number of atoms or expose more parallelism)
- Use memory efficiently (improve memory access patterns and data locality, be mindful of view *LayoutLeft* or *LayoutRight*)
- Keep atom data in GPU memory (avoid moving data as much as possible, port all kernels to Kokkos)
- Avoid launch latency overhead for small systems (fuse kernels if possible), use subview array instead of multiple scalar views for data transfer
- Avoid allocating memory every timestep (overallocate views and only grow if size is exceeded, don’t shrink)
- Watch out for Kokkos view initialization overheads (on by default but can turn off)
- Actually profile to see what is important; don’t spend time optimizing a kernel that is <1% of runtime

Performance of Real Potentials









Port a (Simple) Pairwise Potential

- Abstractions in src/KOKKOS/pair_kokkos.h greatly simplify the process
- Copy *.h and *.cpp files from existing style into src/KOKKOS
- Find/replace old class name with new
- Change the code in the energy and force functions
- Minor changes to parent styles, e.g. add virtual keyword, `copymode`
- Add new styles to src/KOKKOS/Install.sh
- Add new styles to documentation. Don’t forget to update doc/src/Commands_*.rst

Port a Manybody or ML Potential

- Copy *.h and *.cpp files from existing style into src/KOKKOS
- Find/replace old class name with new
- Port loops over atoms to use Kokkos `parallel_for` or `parallel_reduce`
- Port arrays to use Kokkos views
- Minor changes to parent styles, e.g. add virtual keyword, `copymode`
- Add new styles to src/KOKKOS/Install.sh
- Add new styles to documentation. Don’t forget to update doc/src/Commands_*.rst

New KOKKOS Package Option

- Add a new class variable to `kokkos.h` and initialize to default value in `kokkos.cpp`
- Add a new parsing line that sets the value of the variable
- In command names, avoid underscores "`_`", prefer "`/`" instead, e.g. `atom/map` instead of `atom_map`
- Update documentation (lammps/doc/src/package.rst)

Code Examples

- Examine simple compute code: `compute ave/sphere/atom/kk`
- Port simple pairwise potential: `pair_style soft`
- Examine manybody potential code: `pair_style eam/kk`

Porting Advanced Features

- New atom style, styles with complicated forward/reverse comm, core features that need to be ported, etc.
- Copy from closest existing Kokkos file
- Contact developers on LAMMPS developer Slack channel (https://lammps.slack.com) or send an email (see https://www.lammps.org/authors.html) for advice

More Resources

- KOKKOS package documentation: https://docs.lammps.org/Speed_kokkos.html
- MatSci LAMMPS forum archives: join and post new questions, https://matsci.org/lammps
- Github: submit a bug report or draft pull request, https://github.com/lammps/lammps
- LAMMPS reference paper: gives an overview of the code including its parallel algorithms, design features, performance, and brief highlights of many of its materials modeling capabilities https://doi.org/10.1016/j.cpc.2021.108171

Thank You

Questions?