Exceptional service in the national interest

Sandia National Laboratories

A next generation LAMMPS: preparing for the many-core future with Kokkos

Christian Trott

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration

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.

Sandia National Laboratories

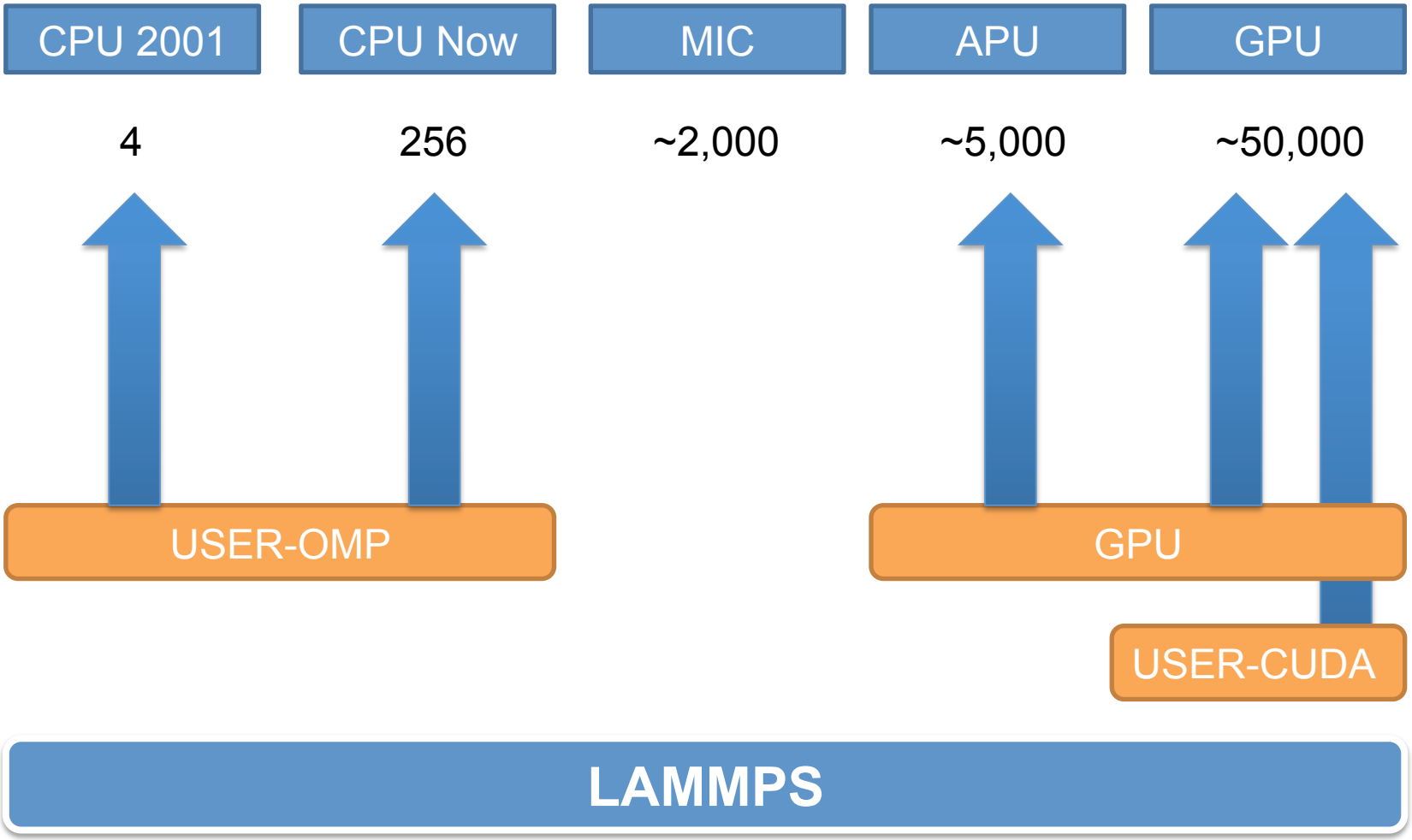
Results presented in this presentation are for preproduction Intel Xeon Phi co-processors (codenamed Knights Corner) and pre-production versions of Intel's Xeon Phi software stack. Performance and configuration of the co-processors may be different in final production releases.

The challenge – Node parallelism

CPU 2001	CPU Now	MIC	APU	GPU
4	256	~2,000	~5,000	~50,000

MPI-Only will not work anymore
! Domains get to small !
We need threading.

The challenge – Node parallelism



Sandia National Laboratories

LAMMPS Current threading support

Packages for specific Hardware

- USER-OMP -> MultiCore CPUs
- USER-CUDA -> NVIDIA GPUs (via CUDA)
- GPU -> general GPUs (via CUDA or OpenCL)
- Smaller non-official packages for GPUs

Advantages

- Very flexible
- No one else has to care

Disadvantages

- No one else has to care -> breaks stuff frequently
- Lot of code repetition
- Divergent code

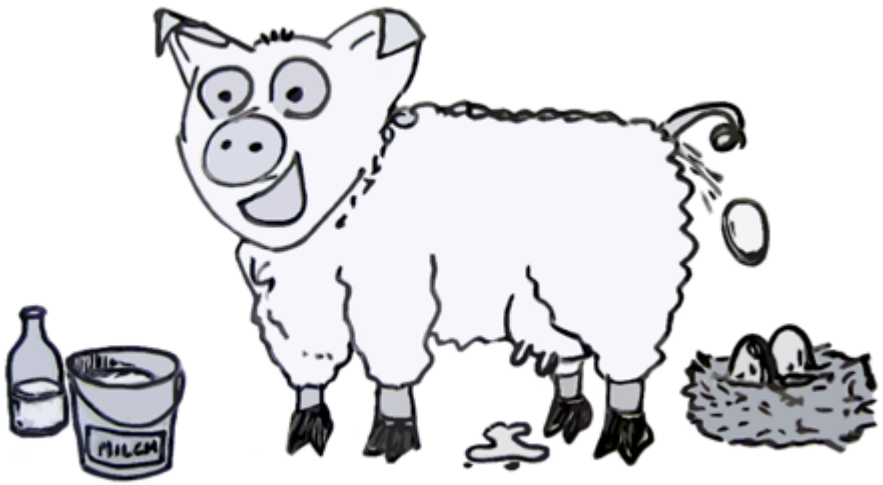
What do we want?

- Single code base
- Support for all current (and future) hardware

- Flexible run configurations
  - MPI-Only
  - MPI + Threads
  - MPI + GPU
  - MPI + GPU + Threads

- Close to optimal performance (i.e. performance of a specialized code)
- Possibility for code specialisation
- Use vendor compilers

- Simple code



Eierlegende Wollmilchsau
(egg-laying wool-milk-sow)

Kokkos as a solution

A programming model with two major components:

Data access abstraction

- Change data layout at compile time without changing access syntax
  => Optimal access pattern for each device
- No pointer chasing
- Data padding and alignment is transparent

Parallel dispatch

- Express algorithms with **parallel_for**, **parallel_reduc**e etc.
- Transparently mapped onto back end languages (e.g. OpenMP, CUDA)

Goal: Separate physics code from hardware details

What is Kokkos?

- C++ template library => almost everything is headers
- Developed as node level parallelism layer for Trilinos

  Trilinos is a Open-Source solver library, development led by Sandia

- Open-Source
- Standalone (no required dependencies)
- Lead developer: Carter Edwards, SNL
- First stable release in September
- Will be integrated into Trilinos during 2014

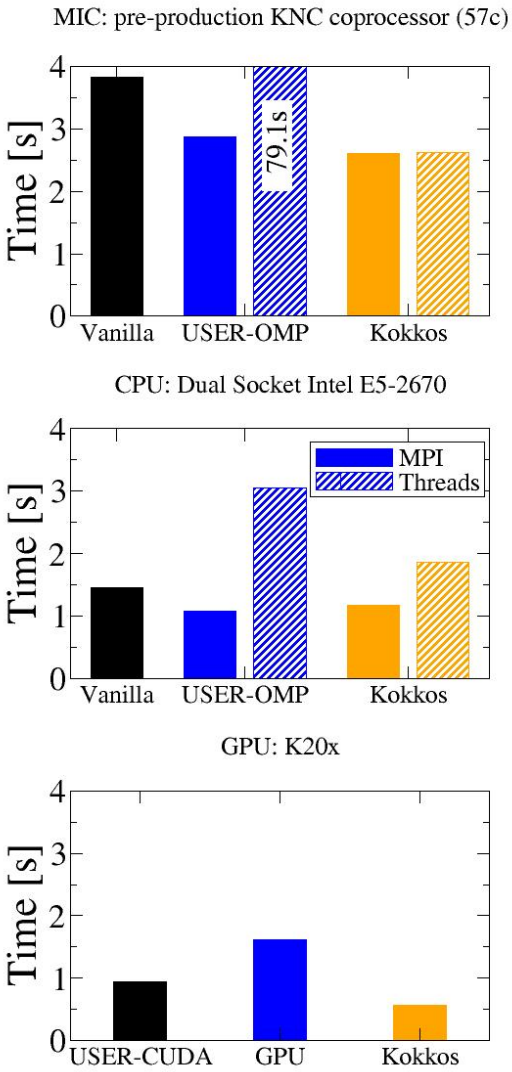
Pre print: *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*
H. Carter Edwards, Christian R. Trott; submitted to JPDC

LAMMPS-Kokkos Prototype

Lennard Jones + NVE

- Kokkos parts
  - Force Kernel
  - Neighborlist construction
  - Integration
  - Forward communication
- Matches performance of LAMMPS and its packages, but better thread-scaling
- CPU: MPI – only and MPI + OpenMP
- GPU: USER-CUDA package and GPU package

On the right: force + neigh time
1000 timesteps, 32,000 atoms LJ



Benefits for Users of LAMMPS

- Capabilities of packages will be part of main LAMMPS
- Less frequent breakdowns / incompatibilities
- No performance bottlenecks due to:

  "I didn't get around to include that in the package"

- Better performance in baseline code
- Maybe coalescing of efforts for many core support?

Hopefully: Future proof LAMMPS

How does it work – datatypes

```
// The View constructor allocates an array
// in 'Device' memory space with dimensions
// N*M*8*3, where each '*' token denotes a
// dimension to be supplied at runtime.
// The label "A" is used in error messages
// which may occur in regard to this array.
View<double**[8][3],Device> a("A",N,M);

// The parentheses operator implements the
// layout map.
a(i,j,k,l) = value ;
```

How does it work - Parallel dispatch

```
// old code
void f(double k, int N) {
  for(int i = 0; i<N; i++) {
    // loop body
  }
}

// new code
template<class Device>
struct f_functor {
  typedef Device device_type;
  double k;
  f_functor(double ktmp): k(ktmp) {};
  operator()(int i) const {
    // loop body
  }
};

void f(double k, int N) {
  f_functor<DefaultDevice> func(k);
  parallel_for(N,func);
}
```

(i) Create a functor with
- operator ()
- members for all accessed data

(ii) Copy loop body into functor

(iii) Call functor with parallel_for

Parallel_for is mapped to parallel iterations using back-end threading model (OpenMP, Pthreads, CUDA).

*There is **no** automatic data transfer!*

Sandia National Laboratories

Current State

Kokkos: Research stable in September (start to keep backward compatibility)

LAMMPS:

- Prototyping on branch
- Framework basics (data synchronization, runtime device choice)
- ***Migrating without breaking current packages***
- Trying to find elegant coding style/guidelines

Hopefully soon: Decision on whether to go forward and start the transition.

If and when decision is made: Help would be very much appreciated to get chunks of LAMMPS migrated.

Sandia National Laboratories

Exceptional service in the national interest

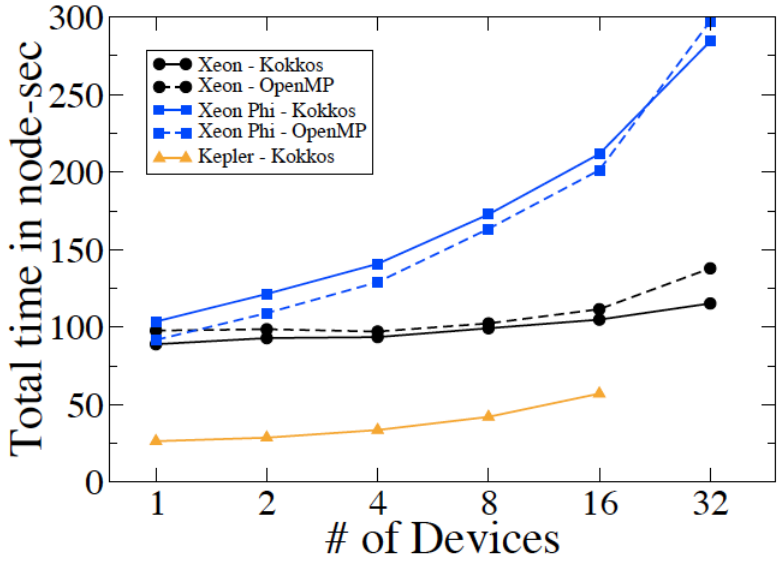
Questions and further discussion: crtrott@sandia.gov

Proof of concept

Using miniApps to explore portability and performance tradeoffs: *mantevo.org*

MiniMD

- All kernels within 10% of "native code"
- Shown on: CPUs, MIC, GPUs
- Significantly higher performance then vanilla LAMMPs on CPUs and MIC
- Only device specific kernel is neighborlist construction



MiniFE

- Math kernels beat MKL and CUSPARSE

Code (soon) available on mantevo.org

More Performance Results – MPI vs. Threads

CPU: 2x E5-2670

	Ref/32x1	Opt/32x1	OMP/32x1	OMP/8x4	OMP/2x16	OMP/1x32	Kokkos/32x1	Kokkos/8x4	Kokkos/2x16	Kokkos/1x32
Total	1.72	1.31	1.34	1.52	2.12	4.24	1.54	1.97	2.16	2.17
Force	1.30	0.90	0.93	1.04	1.32	2.81	1.01	1.42	1.48	1.51
Neigh	0.15	0.15	0.15	0.15	0.17	0.24	0.17	0.33	0.34	0.35
Comm	0.24	0.23	0.23	0.27	0.50	0.51	0.31	0.17	0.27	0.16
Other	0.03	0.03	0.04	0.06	0.13	0.67	0.05	0.05	0.07	0.16

MIC: pre-production Intel KNC coprocessor

	Ref/224x1	Opt/224x1	OMP/224x1	OMP/56x4	OMP/4x56	OMP/1x224	Kokkos/224x1	Kokkos/56x4	Kokkos/4x56	Kokkos/1x224
Total	6.94	5.05	5.55	5.68	29.39	89.70	5.04	3.82	5.27	4.60
Force	3.40	2.27	2.41	3.48	23.23	77.74	2.01	1.83	2.20	2.00
Neigh	0.43	0.43	0.45	0.43	0.85	1.37	0.60	0.56	0.66	0.63
Comm	2.90	2.15	2.35	1.44	3.40	4.43	2.08	1.19	2.04	1.36
Other	0.15	0.15	0.27	0.32	1.90	6.13	0.29	0.22	0.36	0.60

OPT: - best performance on CPU

USER-OMP: - for small thread-counts faster than Kokkos on CPUs

- breaks down at thread counts larger than 16

Kokkos: - best performance on MIC, also thread-scalable

Sandia National Laboratories

Advanced capabilities enabled

- Mixed Precision (maybe extended precision)
- Hybrid execution

- Hardware specific configuration files: choose optimal data layouts for each platform
- Easy way to write specialised versions of kernels for specific hardware in native back-end language

- Debugging help: relatively cheap out-of-bounds checks
- Potentially use some subpackages based on Kokkos (i.e. parallel HashTable, math kernels)

How does it work – datatypes (ii)

```
typedef Kokkos::Host DefaultDevice;

// Precision for position, velocity, and force
typedef double X_Float;
typedef double V_Float;
typedef double F_Float;

// Particle positions always use right layout
// to improve cache line usage with random access
typedef View<X_Float*[3],LayoutRight,
             DefaultDevice>  t_x_array ;
typedef View<const X_Float*[3],LayoutRight,
             DefaultDevice,ReadRandom> t_x_array_rnd;
typedef t_x_array::HostMirror t_x_array_host ;

// Particle velocities use default layout for
// optimal contiguous access pattern
typedef View<V_Float*[3],DefaultDevice> t_v_array ;
typedfe t_v_array::HostMirror t_v_array_host ;

// Neighborlist uses default layout for
// optimal contiguous access pattern
typedef View<int** ,DefaultDevice> t_neighs;
typedef View<const int** ,
             DefaultDevice> t_neighs_const;
```

Migration Strategy – data wrapping

```
// Original array variables
double **x, **v, **f;

// Kokkos array variables
t_x_array d_x; t_x_array_host h_x;
t_v_array d_v; t_v_array_host h_v;
t_f_array d_f; t_f_array_host h_f;

// Allocate on the device
t_x_array d_x = t_x_array("X",natoms);

// View or allocate a host copy
t_x_array_host h_x = create_mirror_view(d_x);

// Temporarily wrap old data structure:
double **x = new double*[natoms];
for(int i = 0; i<natoms; i++)
  x[i] = & h_x(i,0);
```

- Allocate Kokkos Views
- Wrap old structures around it

Problem: Padding and Layout changes problematic/impossible

Sandia National Laboratories

Migration Strategy

- Change data structures
- Develop functors
- Enable dispatch (offload model) for GPU execution
- Optimize algorithms for threading
- Specialize kernels for specific architectures.

How does it work – parallel dispatch (ii)

```
// Original class member function
class C {
public:
  void f(double k, int N) {
     for(int i = 0; i<N; i++) {
       // loop body
     }
  }
};

// Modified class with a functor-wrapper
class C {
public:
  double k ;
  void f(double ktmp, int N) {
    k = ktmp;
    f_functor<DefaultDevice> func(*this);
    parallel_for(N,func);
  }

  void f_item(int i) const {
    // loop body
  }
};

template<class Device>
struct f_functor {
  typedef Device device_type
  C c;
  f_functor(C &c_in): c(c_in) {};
  operator()(int i) const {
    c.f_item(i);
  }
};
```

- Modify original function to copy all arguments into class members
- Create a new class member function **C::f_item(int i)** containing the loop body and the loop index as its argument.
- Replace loop body with call to the new **C::f_item** function to test modified code.
- Create **f_functor** with an instance of the class **C** as a member, and its parentheses operator calls **C ::f_item(int i)**.
- Change the original class member function create and dispatch **f_functor** via **parallel_for** or **parallel_reduce**.