IMPERIAL

Granular GPU LAMMPS Simulations with KOKKOS

Float Precision, Nondimensionalization, Compiling and Running LAMMPS with KOKKOS

Dariel Hernández - Delfin

Belfast

25/03/2026

Outline

Table of Contents

Motivation

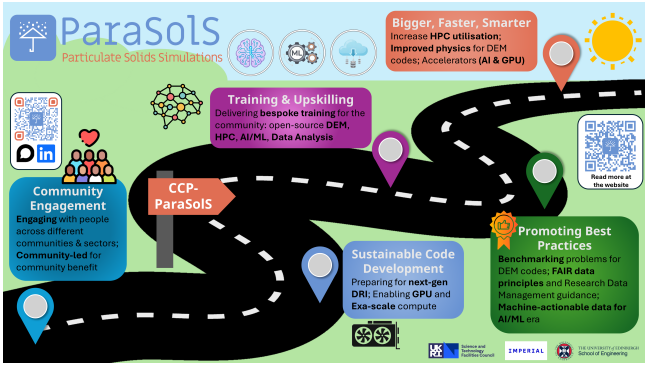


Figure: ParaSolS roadmap

Motivation

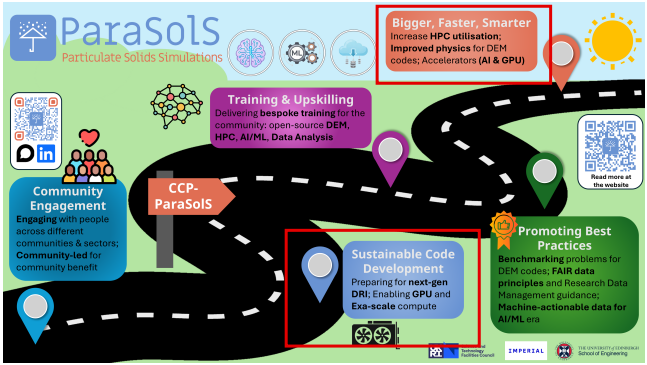


Figure: ParaSolS roadmap

Enabling GPU for DEM simulations is the Present and Future of ParaSols Community.

Table of Contents

Why Kokkos for Granular Media?

The Challenge:

- Granular simulations are computationally expensive ($O(N)$ neighbour lists, force calculations at short time steps, particle shape complexity).
- Hardware heterogeneity (CPUs, GPUs [Nvidia, AMD, and Intel]).
- Memory bandwidth limitations (moving data, irregular access).

The Solution: Kokkos

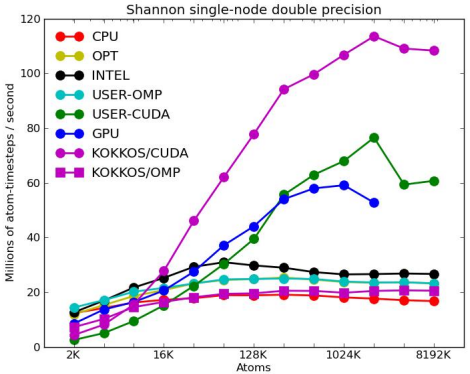
- Performance portability ecosystem.
- Single source code for multiple architectures – The code can be compiled with CUDA/HIP/SYCL (GPUs) and OpenMP (CPUs), preventing the use of "special code".
- Seamless integration with LAMMPS (KOKKOS package).

Why Kokkos for Granular Media?

Previous benchmarks showed good results for other kinds of pair interactions.

Performance Metrics

$$MATS/s = N_{atoms} N_{steps} / (WallTime * 10^6)$$



Top 10 HPC Systems

9/10 relies on GPUs

Rank	System	GPU	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	El Capitan	AMD Instinct MI300A	11,340,000	1809.00	2821.10	29,685
2	Frontier	AMD Instinct MI250X	9,066,176	1353.00	2055.72	24,607
3	Aurora	Intel Data Center GPU Max	9,264,128	1012.00	1980.01	38,698
4	JUPITER Booster	NVIDIA GH200 Superchip	4,801,344	1000.00	1226.28	15,794
5	Eagle	NVIDIA H100	2,073,600	561.20	846.84	–
6	HPC6	AMD Instinct MI250X	3,143,520	477.90	606.97	8,461
7	Fugaku	–	7,630,848	442.01	537.21	29,899
8	Alps	NVIDIA GH200 Superchip	2,121,600	434.90	574.84	7,124
9	LUMI	AMD Instinct MI250X	2,752,704	379.70	531.51	7,107
10	Leonardo	NVIDIA A100 SXM4 64 GB	1,824,768	241.20	306.31	7,494

Source: TOP500.org

Top 5 HPC Systems in the UK

Rank	System	GPU	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
11	Isambard-AI phase 2	NVIDIA GH200 Superchip	1,028,160	216.50	278.58	–
31	Njoerd	NVIDIA H100 SXM5 80GB	273,280	78.20	106.28	–
71	Mary Coombs	NVIDIA H100 SXM5 80GB	99,456	24.41	31.18	–
88	ARCHER2	–	716,800	19.54	25.80	–
92	Dawn	Intel Data Center GPU Max	156,864	19.46	53.85	–

Source: TOP500.org

Table of Contents

Why discuss floating-point precision?

The ideal mathematical picture

In granular mechanics we usually think in terms of continuous variables:

$$\mathbf{x},\ \mathbf{v},\ \delta,\ \mathbf{F}_n,\ \mathbf{F}_t \in \mathbb{R}$$

That is, positions, velocities, overlaps, and contact forces are treated as real-valued quantities.

What the computer actually sees

A computer does not work on the continuum of real numbers, but on a finite set of representable values:

$$\mathbf{x},\ \mathbf{v},\ \delta,\ \mathbf{F}_n,\ \mathbf{F}_t \in \mathcal{F}$$

where $\mathcal{F}$ is the set of floating-point numbers defined by the IEEE standard.

Core idea

A numerical simulation evolves continuum mechanics through a discrete arithmetic lattice.

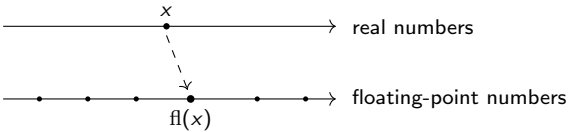
An intuitive picture: computation on a grid

Analogy

Instead of living on the full real line, the computer lives on a grid of admissible numbers.

Every time a number is stored, or an arithmetic operation is performed, the exact result is generally replaced by a nearby floating-point number:

$$\mathrm{fl}(x) \approx x$$



- some numbers are represented exactly,
- many others are rounded,
- each operation may introduce a small perturbation.

Consequence

Floating-point error is not only a measurement limitation: it is part of the arithmetic itself.

Why this matters in granular simulations

Granular simulations can be especially sensitive to small numerical perturbations because they involve:

- contact detection based on small gaps or overlaps,
- force laws that depend nonlinearly on overlap,
- history-dependent tangential interactions,
- long sequences of time integration steps,
- large numbers of particles and contacts.

Key observation

A tiny rounding error in position or overlap may alter:

$$\delta \;\rightarrow\; F_n(\delta), \qquad \Delta s_t \;\rightarrow\; F_t$$

and these differences can accumulate over many contact updates.

Floating-Point Numbers in IEEE 754

General idea

A floating-point number is represented in the form

$$x = (-1)^s \, (1.f)_2 \, 2^{e-\text{bias}}$$

$$(1.f)_2 = 1 + \Sigma_{i=1}^{p-1} f_i 2^{-i}$$

for normalised numbers, where:

- s is the sign bit,
- f is the fraction (mantissa field),
- e is the stored exponent,
- bias is a fixed offset used for the exponent.

IEEE 754 common formats

- **Single precision (32 bits) – FP32:**

  $$1 \text{ sign} + 8 \text{ exponent} + 23 \text{ fraction}$$

  $$\text{bias} = 127$$

- **Double precision (64 bits) – FP64:**

  $$1 \text{ sign} + 11 \text{ exponent} + 52 \text{ fraction}$$

  $$\text{bias} = 1023$$

IEEE 754: range and precision

Key quantities

For a binary floating-point format with precision p and exponent bias:

$$x_{\min}^{\mathrm{norm}} = 2^{1-\mathrm{bias}}, \qquad x_{\max}^{\mathrm{norm}} = \left(2 - 2^{-(p-1)}\right) 2^{\mathrm{bias}}, \qquad u = 2^{-p}$$

Format	Smallest positive normalized	Largest positive normalized	Unit roundoff u
Single precision	$2^{-126} \approx 1.18 \times 10^{-38}$	$(2 - 2^{-23})2^{127} \approx 3.40 \times 10^{38}$	$2^{-24} \approx 5.96 \times 10^{-8}$
Double precision	$2^{-1022} \approx 2.23 \times 10^{-308}$	$(2 - 2^{-52})2^{1023} \approx 1.80 \times 10^{308}$	$2^{-53} \approx 1.11 \times 10^{-16}$

Interpretation

The smallest and largest values determine the *range*; the unit roundoff determines the *relative precision*.

Shear history: single precision example

Unit roundoff

Single precision: $u \approx 5.96 \times 10^{-8}$
Smallest resolvable increment: $\Delta\delta_{t,\min} \sim u\,|\delta_t|$

Numerical example (colab example single precision)

$\delta_t = 10^{-3}$ m, $\Delta\delta_t = 10^{-11}$ m
Threshold: $u\,\delta_t \approx 5.96 \times 10^{-11}$ m
$\Delta\delta_t < u\,\delta_t \Rightarrow$ update may be lost

Consequence

Lost increments distort $\boldsymbol{\delta}_t$ and $\mathbf{F}_t$, affecting long-term stress and energy, especially for long contacts or tiny time steps.

Special Cases

$$x = (-1)^s \, (1.f)_2 \, 2^{e-\text{bias}},$$

Exponent field all zeros

- fraction $= 0 \;\rightarrow\; \pm 0$
- fraction $\neq 0 \;\rightarrow\;$ subnormal numbers

Exponent field all ones

- fraction $= 0 \;\rightarrow\; \pm\infty$
- fraction $\neq 0 \;\rightarrow\;$ NaN

Subnormal numbers–Provide gradual underflow near zero

$$x = (-1)^s (0.f)_2 \, 2^{1-\text{bias}}$$

Modern GPU cores: floating-point perspective

1. Scalar / CUDA cores

- General-purpose ALUs capable of FP32, FP64 (and sometimes FP16 / BF16 / INT)
- Primary workhorse for irregular kernels, reductions, particle loops
- Used in most DEM kernels: neighbour lists, contact forces, per-particle updates

2. Tensor / Matrix cores

- Specialised units for dense linear algebra (matrix-matrix, GEMM)
- Accelerate FP32, FP64 (limited), FP16/BF16, and lower-precision AI formats
- Peak TFLOPS often reported here; very beneficial for ML, batched linear algebra
- Only useful for DEM if kernels are batched into matrix-style operations

Modern GPU cores: floating-point perspective

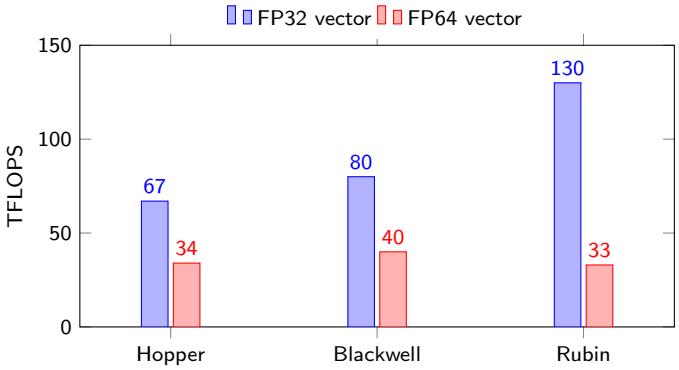
3. Optional specialised cores (AI / Ray Tracing)

- RT cores: geometry acceleration, ray tracing (mostly graphics)
- Sparse / AI cores: FP4 / FP8 or INT4-8; mainly AI inference
- Minimal relevance for classical DEM unless ML-assisted pruning or surrogate models are used

Takeaway for HPC / DEM

The cores that actually matter for irregular particle simulations are usually scalar / CUDA cores (FP32 and FP64 paths). Tensor/Matrix cores matter only if the workload is explicitly expressed as dense batched arithmetic.

Vector floating-point throughput across recent NVIDIA architectures



Observation

FP32 throughput grows strongly from Hopper to Rubin, but FP64 does not follow the same trend.

Implication

A newer GPU does not automatically provide a proportional gain for FP64-heavy scientific workloads.

Key distinction

These are **vector** floating-point figures. They should not be confused with much larger Tensor/matrix throughput numbers.

Why FP32/FP64 charts matter more than headline AI numbers

For simulation, the useful question is not

"Which GPU has the biggest advertised FLOP number?"

The useful question is

"Which precision path does my code actually use?"

FP32? FP64? Tensor/matrix?

- DEM and contact dynamics are often dominated by irregular kernels, memory traffic, and reductions.
- Those kernels usually do not benefit from Tensor-core headline throughput in the same way as dense matrix workloads.
- Therefore, vector FP32 / FP64 performance is often a more honest first comparison.

Takeaway

Architectural progress is becoming increasingly **precision-dependent**: large gains in FP32 or AI-oriented formats do not imply equally large gains in FP64-oriented simulation.

Is the same trend observed in AMD GPUs?

Not to the same extent

Recent AMD Instinct GPUs retain a more HPC-oriented balance between FP32 and FP64 vector throughput.

GPU	FP32 vector	FP64 vector
AMD Instinct MI300X	163.4 TFLOPS	81.7 TFLOPS
AMD Instinct MI350X	144.2 TFLOPS	72.1 TFLOPS

Interpretation

For these AMD Instinct parts, FP64 remains a major performance path rather than a secondary capability.

Takeaway

The recent AMD Instinct narrative is still more aligned with

$$\text{AI} + \text{HPC}$$

whereas the recent NVIDIA roadmap appears more strongly tilted toward

AI-first growth, especially in low precision and FP32-oriented paths.

Recent NVIDIA vs AMD trend in vector FP32 and FP64

Architecture / GPU	FP32 vector	FP64 vector	Comment
NVIDIA Hopper H100	67	34	balanced AI/HPC generation
NVIDIA Blackwell	80	40	moderate FP64 growth
NVIDIA Rubin	130	33	strong FP32 growth, FP64 not proportional
AMD MI300X	163.4	81.7	strong HPC-style FP64 support
AMD MI350X	144.2	72.1	FP64 still preserved at high level

Main message

The current trend is not universal across vendors: NVIDIA's newest public roadmap emphasises stronger FP32/FP16/FP8 / AI expansion, while AMD Instinct still presents a more balanced FP32–FP64 HPC profile.

Table of Contents

Why discuss scaling in DEM?

- DEM simulations of granular materials often involve:
  - small particles,
  - large stiffness,
  - different levels of confining stresses,
  - long quasi-static shear stages.
- This combination usually leads to:
  - very small stable timesteps,
  - very large number of steps,
  - possible round-off issues in single precision.
- The natural question is:

  Can nondimensionalisation or scaling improve performance without changing the physics?

Main questions addressed

- What changes after nondimensionalisation?
- What does not change?
- How does pressure scaling work?
- How do the timestep and the number of steps scale?
- What is the role of the inertial number I?
- What are the differences between quasi-static and inertial regimes?
- When does single precision become risky?

Reference physical system

Typical system considered:

$$d = 2 \times 10^{-3}\ \mathrm{m},$$
$$\mu = 0.5,$$
$$G = 29\ \mathrm{GPa} = 2.9 \times 10^{10}\ \mathrm{Pa},$$
$$p \in [50\ \mathrm{kPa},\ 1\ \mathrm{MPa}], I \sim 10^{-4}$$

Representative density:

$$\rho \approx 2650\ \mathrm{kg/m^3}$$

A useful stiffness ratio is:

$$\frac{G}{p}$$

For example, at $p = 100$ kPa:

$$\frac{G}{p} = \frac{2.9 \times 10^{10}}{10^5} = 2.9 \times 10^5$$

General nondimensionalisation

Choose reference scales:

$$L_0 = d, \qquad \sigma_0 = \text{reference stress}, \qquad \rho_0 = \rho$$

Then:

$$x^* = \frac{x}{d}, \qquad p^* = \frac{p}{\sigma_0}, \qquad G^* = \frac{G}{\sigma_0}$$

Natural time scale:

$$t_0 = d\sqrt{\frac{\rho}{\sigma_0}}$$

Dimensionless time, force, and shear rate:

$$t^* = \frac{t}{t_0}, \qquad F^* = \frac{F}{\sigma_0 d^2}, \qquad \dot{\gamma}^* = \dot{\gamma}\, t_0$$

Example at $p = 100$ kPa

Using:

$$G = 2.9 \times 10^{10}\ \mathrm{Pa}, \qquad p = 10^{5}\ \mathrm{Pa}$$

Scaling	G^*	p^*
Pressure scaling ($\sigma_0 = p$)	2.9×10^5	1

Interpretation:

- pressure scaling keeps stresses $\mathcal{O}(1)$,

Hertz normal force law

For elastic contact:

$$F_n \sim G\sqrt{d}\,\delta^{3/2}$$

In a quasi-static assembly, a typical contact force is set by confinement:

$$F_n \sim pd^2$$

Equating both gives the characteristic overlap:

$$pd^2 \sim G\sqrt{d}\,\delta^{3/2}$$

Hence:

$$\delta \sim d\left(\frac{p}{G}\right)^{2/3}$$

Linearised contact stiffness

Local Hertz stiffness:

$$k_n = \frac{dF_n}{d\delta} \sim G\sqrt{d\delta}$$

Using

$$\delta \sim d\left(\frac{p}{G}\right)^{2/3}$$

we obtain:

$$k_n \sim G\sqrt{d \cdot d\left(\frac{p}{G}\right)^{2/3}}$$

Thus:

$$k_n \sim d\, G^{2/3} p^{1/3}$$

Dimensionless timestep in quasi-static DEM

Using

$$t_0 = d\sqrt{\frac{\rho}{\sigma_0}}$$

the dimensionless stable timestep scales as:

$$\Delta t^* \sim (G^*)^{-1/6}$$

This is not an exact equality, but a very useful scaling estimate.
Consequences:

- larger G^* gives smaller admissible Δt^*,
- softer dimensionless systems allow larger Δt^*,
- the gain is only algebraic, since the exponent is $-1/6$.

Scaled shear rate

From

$$t_0 = d\sqrt{\frac{\rho}{\sigma_0}}$$

the dimensionless shear rate is:

$$\boxed{\dot{\gamma}^* = \dot{\gamma}\, d\sqrt{\frac{\rho}{\sigma_0}}}$$

This quantity depends on the chosen reference stress σ_0.

$$\boxed{I = \frac{\dot{\gamma} d}{\sqrt{p/\rho}} \text{ is invariant under nondimensionalisation}}$$

How does the number of steps change?

The number of steps is:

$$N = \frac{T}{\Delta t}$$

In dimensionless form:

$$N = \frac{T^*}{\Delta t^*}$$

Therefore:

N does not change under nondimensionalisation alone

This is crucial:

- changing units does not reduce computational cost by itself,
- cost changes only if the admissible timestep changes,
- or if the physical loading path is modified.

Presentation-ready magnitude table (Pressure scaling vs Dimensional)

Quantity	Dimensional	Pressure scaling	Safety comment
Position x	$\mathcal{O}(10^{-3})$–$\mathcal{O}(10^{-1})$ m	$\mathcal{O}(1)$–$\mathcal{O}(10^{2})$	Safe; positions well resolved
Velocity v	6.14×10^{-3} m/s	1.00×10^{-3}	Safe; fully within single-precision range
Normal overlap δ_n	4.56×10^{-7} m	2.28×10^{-4}	Safe; no resolution issues
Normal force F_n	4.0×10^{-1} N	1	Good; centered nicely for computation
Tangential force $F_t \sim \mu F_n$	2.0×10^{-1} N	5.0×10^{-1}	Safe; same as F_n
Normal stiffness k_n	8.8×10^{5} N/m	4.39×10^{3}	Large but still numerically fine
Tangential stiffness k_t	$\sim k_n$	$\sim 4.39 \times 10^{3}$	Same as k_n
Scaled timestep Δt^*	10^{-7} s	3.07×10^{-4}	Comfortably resolved
Shear increment Δs_t	6.14×10^{-10} m	3.07×10^{-7}	Slightly small; still safe
Shear force increment ΔF_t	$\sim 5.4 \times 10^{-4}$ N	1.35×10^{-3}	Critical; monitor history updates carefully

Take-home message from the table

All variables in the reference problem lie safely inside the *dynamic range* of single precision. Therefore, the real question is not:

Do the numbers fit in float?

but rather:

Are the smallest updates still resolvable when added to stored values?

For the present estimates:

single precision is technically defensible under pressure scaling

and often still acceptable under moderate hybrid scaling, but much less convincing under pure G-scaling.

Table of Contents

Kokkos Essentials: Views & Parallelism

1. **Kokkos Views – Data Abstraction**
   - Multi-dimensional arrays for host/device memory.
   - Example:

     ```
     Kokkos::View<double*> x("x", N);
     ```

   - Portable, memory-safe, resides on CPU or GPU.
2. **Parallel Execution –** `parallel_for` **&** `parallel_reduce`
   - `parallel_for` – loop parallelization:

     ```
     Kokkos::parallel_for(N, KOKKOS_LAMBDA(int i){x(i) = i*i;});
     ```

   - `parallel_reduce` – sum or custom reduction:

     ```
     double sum;
     Kokkos::parallel_reduce(N, KOKKOS_LAMBDA(int i, double& local_sum){
     local_sum += x(i);}, sum);
     ```

   - Portable across threads, GPU, and other backends.

Key Takeaways: Views decouple memory computation; parallel patterns run anywhere.

LAMMPS + Kokkos: CUDA (NVIDIA GPU)

- Set these CMake options:

  ```
  -D Kokkos_ARCH_HOSTARCH=yes
  -D Kokkos_ARCH_GPUARCH=yes
  -D Kokkos_ENABLE_CUDA=yes
  -D Kokkos_ENABLE_OPENMP=yes
  ```

- Optional: set precision

  ```
  -D KOKKOS_PREC=<double|mixed|single>
  ```

- Other recommended flags:

  ```
  -D Kokkos_ENABLE_LIBRT=ON
  ```

LAMMPS + Kokkos: HIP (AMD/NVIDIA GPU)

- Set these CMake options:

  ```
  -D Kokkos_ARCH_HOSTARCH=yes
  -D Kokkos_ARCH_GPUARCH=yes
  -D Kokkos_ENABLE_HIP=yes
  -D Kokkos_ENABLE_OPENMP=yes
  ```

- Optional: set precision

  ```
  -D KOKKOS_PREC=<double|mixed|single>
  ```

- Other recommended flags:

  ```
  -D Kokkos_ENABLE_LIBRT=ON
  ```

LAMMPS + Kokkos: SYCL (Intel GPU)

- Set these CMake options:

```
-D Kokkos_ARCH_HOSTARCH=yes
-D Kokkos_ARCH_GPUARCH=yes
-D Kokkos_ENABLE_SYCL=yes
-D Kokkos_ENABLE_OPENMP=yes
-D FFT_KOKKOS=MKL_GPU
```

- Optional: set precision

```
-D KOKKOS_PREC=<double|mixed|single>
```

- Other recommended flags:

```
-D Kokkos_ENABLE_LIBRT=ON
```

Precision Options in Kokkos

- **double**: FP64 everywhere (default)
- **mixed**: FP64 for accumulation (forces, energy, virial), FP32 otherwise
- **single**: FP32 everywhere
- Warning: Reduced precision may affect stability and energy conservation. Always check simulations.

LAMMPS + Kokkos: Overview

- Goal: Compile LAMMPS with Kokkos for GPU/CPU portability.
- Targets: Local computer (WSL/Ubuntu) and HPC system (Imperial HPC example).
- Steps:
  1. Install prerequisites
  2. Download LAMMPS source
  3. Configure CMake build
  4. Compile and optionally install

Prerequisites: Local Machine (WSL/Ubuntu)

- Install Ubuntu (example: 24.04) and enable WSL:

```
wsl --install Ubuntu-24.04 --name lammpskokkos
```

- System update dependencies:

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y cmake build-essential ccache gfortran \
    openmpi-bin libopenmpi-dev libfftw3-dev libjpeg-dev \
    libpng-dev python3-dev python3-pip python3-virtualenv \
    libblas-dev liblapack-dev libhdf5-serial-dev hdf5-tools \
    libgtk2.0-dev
```

- GPU setup (NVIDIA):

```
# check GPU
nvidia-smi
# install CUDA 13.2 following official guide
export PATH=$PATH:/usr/local/cuda-13.2/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-13.2/lib64
```

Prerequisites: HPC System (Imperial HPC)

- Connect via SSH:

```
ssh user@login.cx3.hpc.imperial.ac.uk
```

- Load modules:

```
module load CMake
module load intel/2025b
module load CUDA
```

- Many HPC systems already include precompiled LAMMPS + Kokkos modules:

```
module spider Kokkos
# Examples:
# LAMMPS/2Aug2023_update2-foss-2023a-kokkos-CUDA-12.1.1
```

Download LAMMPS Source

- Clone the release branch:

  ```
  git clone -b release https://github.com/lammps/lammps.git lammps_kokkos
  ```

- Enter source directory:

  ```
  cd lammps_kokkos
  ```

Build with Kokkos, Granular, Misc Packages

- Example for Intel + CUDA GPU:

```
cmake -S cmake -B build \
  -D PKG_KOKKOS=yes \
  -D PKG_GRANULAR=yes \
  -D PKG_MISC=yes \
  -D Kokkos_ARCH_ICX=yes \
  -D Kokkos_ARCH_ADA89=yes \
  -D Kokkos_ENABLE_CUDA=yes \
  -D Kokkos_ENABLE_OPENMP=yes
```

- Adjust ‘Kokkos_ARCH_*‘ and backend flags for your system:
  - CUDA / HIP / SYCL
  - Host architecture

Running LAMMPS on GPUs with Kokkos

- Use `-k` to specify GPUs per node.
- Set MPI tasks/node (`-np`) equal to the number of physical GPUs.
- Multiple MPI tasks can share a GPU (may improve speed if parts of the script are not Kokkos-enabled).
- CUDA MPS is recommended when sharing GPUs.
- GPU-aware MPI is highly recommended.

Kokkos GPU Examples: Single Multi-node

- 1 node, 2 MPI tasks/node, 2 GPUs/node:

```
mpirun -np 2 lmp_kokkos_cuda_openmpi \
       -k on g 2 -sf kk -in in.lj
```

- 16 nodes, 2 MPI tasks/node, 2 GPUs/node (32 GPUs total):

```
mpirun -np 32 -ppn 2 lmp_kokkos_cuda_openmpi \
       -k on g 2 -sf kk -in in.lj
```

Kokkos Package Defaults for GPUs

- Default:
  - Full neighbor lists
  - Newton off for pairwise bonded interactions
  - Threaded communication
- Maxwell/Kepler GPUs: defaults are usually best
- Pascal GPUs+: consider
  - Half neighbor lists
  - Newton on
  - Neighbor binsize $= 2 \times$ CPU default

Custom Kokkos Options Example

```
# Newton on, half neighbor list, binsize = neighbor ghost cutoff
mpirun -np 2 lmp_kokkos_cuda_openmpi -k on g 2 -sf kk \
       -pk kokkos newton on neigh half binsize 2.8 -in in.lj
```

- Adjust options to match GPU architecture and potential style
- Experimenting can improve performance significantly

Performance Tips with Kokkos on GPUs

- Avoid fix or compute styles not yet Kokkos-enabled.
- Minimize host-GPU data transfers:
  - Non-Kokkos fixes/computes
  - Thermo or dump output
- Keep data on GPU across multiple timesteps for best performance.

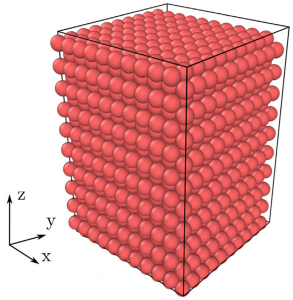
Benchmarking CPU/GPU on Local Computer

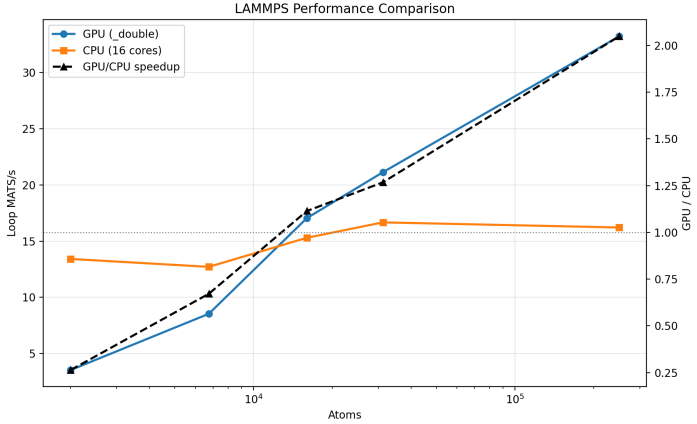
Modified LAMMPS (Kokkos-enabled commands)

```
compute pressure/kk
fix press/berendsen/kk
```

Benchmarking CPU/GPU on local computer (just Nsteps = 20000)

$$MATS/s = N_{atoms} N_{steps} / (WallTime * 10^6)$$





YGES2026 announcement

www.imperial.ac.uk/geotechnics/events/yges2026/

Questions?

d.hernandez-delfin@imperial.ac.uk