Characterizing Molecular Dynamics Simulation on Commodity Platforms

Francesco Peverelli[1] Davide Conficconi[1] Davide Basilio Bartolini[2]
Alberto Scolari[2] Marco Domenico Santambrogio[1]

[1]*DEIB, Politecnico di Milano, Italy*
[2]*Systems Laboratory, Zurich Research Center, Huawei Technologies, Switzerland*

{francesco.peverelli, davide.conficconi, marco.santambrogio}@polimi.it
{davide.basilio.bartolini, alberto.scolari}@huawei.com

Abstract

Molecular Dynamics (MD) simulation is an essential tool driving innovation in key scientific domains such as physics, materials science, biochemistry, and drug discovery. Enabling larger, longer, and more accurate MD simulations can directly impact scientific discovery and innovation. While domain-specific architectures for MD exist, they are not widely accessible, and MD performance on commodity platforms (i.e., CPUs and GPUs) remains critical for supporting broad and agile scientific progress.

This paper aims at characterizing MD simulation on commodity platforms with a benchmark campaign on modern systems available in public cloud offerings. We focus on LAMMPS, one of the prevalent MD frameworks, and characterize several representative and diverse MD experiments. We find that the benchmarked CPU instance provides good scalability to many cores, while the reference LAMMPS GPU implementation struggles with scaling to multiple devices. Additionally, we evaluate the performance impact of application-specific parameters such as error threshold and arithmetic precision.

Our study indicates that key drivers for further improvement of LAMMPS performance on commodity systems are: 1) improving scalability and offload efficiency in multi-accelerator systems and 2) reducing work imbalance in the CPU parallelization.

1. Introduction

Molecular Dynamics (MD) simulation is a crucial computational tool driving scientific discovery and innovation in several domains such as physics, material science, biochemistry, and drug discovery [20]. The computational demands of MD simulation restrict the duration, size, and accuracy for which the evolution of a system can be simulated, even on High-Performance Computing (HPC) systems. Cutting-edge Domain-Specific Architectures (DSAs) designed for MD simulation achieve up to hundreds of microseconds of simulated time per day of real-time on a 512-node system when simulating one million atoms [35]. Real applications, however, are in strong demand for high performance. For example, drug discovery would greatly benefit from reducing the turnaround time of experiments simulating several milliseconds of molecular system evolution [14].

Even DSAs fall short of modern application demands and, additionally, this specialized hardware is not very accessible: despite public institutions offering access [8], only limited compute time is available and requires submitting a detailed proposal. For this reason, broad and agile scientific progress hinges upon MD simulations being run on general-purpose commodity platforms (i.e., CPUs and GPUs), which are readily available in HPC-grade configurations from cloud computing providers. However, commodity platforms are currently up to 1000× slower than DSAs [35, 37].

While the performance gap with specialized hardware can hardly be closed [15, 30, 35, 41, 42], detailed workload analysis and characterization can still provide valuable insights to drive performance improvement for MD on commodity systems, thus expanding the experiments that do not require a DSA for a reasonable turnaround time [34]. While previous work [3, 25, 28] focused on proving good weak scaling properties and performance portability, the goal of this paper is to provide a detailed characterization of several representative and diverse MD experiments on modern high-performance commodity platforms. We construct our workloads with the widely used LAMMPS simulator and benchmark them on modern HPC-grade CPUs and GPUs, and we focus our analysis on performance and scalability within a single compute node (i.e., a multi-socket CPU and a multi-GPU server).

In summary, this paper makes the following contributions:

- We provide a concise illustration of the common structure and computational steps of MD experiments (Section 2).
- We construct a representative and diverse benchmark suite of MD experiments using LAMMPS (Section 3).

. DOI: 10.1109/IISWC55918.2022.00016

- Based on a rigorous methodology (Section 4) [32] , we characterize our benchmark suite on HPC-grade CPU-based (Section 5) and GPU-based (Section 6) systems.
- We perform a sensitivity study to experiment-specific input parameters (Section 7 and Section 8).
- We highlight directions for improving MD performance on next-generation commodity platforms (Section 10).

2. Molecular Dynamics Simulation

A Molecular Dynamics (MD) simulation is a computational tool for analyzing the temporal evolution of a system of particles [21]. This paper focuses on LAMMPS, an open-source MD framework [3, 39], as a representative platform for characterizing the workload. Before delving into our analysis, we provide the relevant background information on typical MD experiments and LAMMPS.

An MD experiment models a portion of space containing the system of particles under study as a ***simulation box***. Within it the experiment simulates the evolution of the system by iteratively *computing* interaction *forces* between particles and consequently *updating* their velocities and positions. We call one iteration of this computation ***timestep***. Regardless of the timestep granularity (e.g., microseconds or nanoseconds), which is specific to each experiment, we evaluate performance in terms of *timesteps/s* or *TS/s*. This standard metric reflects the execution time of the main algorithmic steps, allowing us to compare experiments with different timestep granularity.

Interaction forces between particles can be ***bonded***, i.e., among few neighboring atoms linked by covalent bonds, and ***non-bonded*** or long-range, i.e., affecting every particle in the system. These interactions are computed based on molecular mechanics ***force fields*** or inter-atomic potentials [22]. Examples of force fields used in our following experiments are *Lennard-Jones (LJ)* [24], *CHARMM* [40] pairwise potentials, and *EAM* many-body potential [13]. Additionally, one of our experiments (*Chute*) uses a Hookean-style formula for the friction between two granular particles [11].

Particle velocity and position are determined by numerically solving Newton's equations of motion, usually through the Velocity Verlet algorithm [38] and, depending on which quantities are assumed to remain constant during the simulation, different ***integration strategies*** can be used. All the experiments we analyze except one (*Rhodopsin*) use the *NVE LAMMPS* command, which performs plain time integration under the assumptions that 1) the number of atoms (N), volume (V), and energy of the system (E) remain constant, and 2) the simulation box has ***periodic boundary conditions*** [23]. Conversely, Rhodopsin uses the *NPT command*, which performs time integration via Nose-Hoover style non-Hamiltonian equations of motion.

Computing pairwise interactions scales as $O(N^2)$, where N is the number of particles, and becomes impractical for large systems. One way to reduce this complexity is to *ignore the interaction forces* caused by particles farther away than a given ***cutoff threshold***. However, introducing a distance cutoff requires keeping track of which particles are within the cutoff distance of each other particle at each timestep. Particularly for non-bonded forces, which apply across all space, interactions of particles within the cutoff threshold are considered ***short-range***, in contrast to ***long-range*** interactions of particles beyond the cutoff threshold.

LAMMPS supports a cutoff threshold with ***neighbor lists***, thus storing for each particle in the system an array with its neighbors, i.e., all particles that are within the cutoff distance. To avoid frequently updating the neighbor list for particles moving across the cutoff distance, LAMMPS defines a ***skin distance*** and additionally stores in the neighbor list all particle pairs within the cutoff plus the skin distance: a larger skin distance requires checking more particles for possible interactions at each timestep, but allows rebuilding neighbor lists less often.

In the case of non-bonded forces, e.g., with Coulomb and dipolar potentials, truncating the contribution of particles outside the cutoff range effectively disregards all long-range interactions and may result in significant simulation errors. For this reason, several methods have been developed to approximate the long-range contribution of these forces. In particular, LAMMPS implements the *Ewald summation* [16] and *Particle-particle Particle Mesh (PPPM)* [19, 27] methods. The *Ewald summation* method splits the summation of the potentials into two separate terms for short- and long-range contributions. Short-range contributions are computed directly with pairwise interactions, while long-range contributions are approximated with a more efficient computation in the frequency domain: potentials and 3D charge density are first transformed into the frequency space, then multiplied pairwise, and, finally, transformed back to the real domain. Transforming the original convolution into a pointwise multiplication in the frequency domain reduces the complexity of the long-range computation from $O(N^2)$ to $O(Nlog(N))$. LAMMPS implements this step via a 3D Fast Fourier Transform (FFT). The PPPM method additionally interpolates the charge density on a fixed grid when computing the long-range interactions.

2.1. Reference MD Experiment Structure

Building on top of the terminology just defined, Figure 1 illustrates the structure of an MD simulation experiment First, an initialization step sets the values for initial particle positions and velocities; then, timesteps are iteratively computed. For a system of N atoms and npa_{avg} *neighbors per atom* on average, each timestep goes through the following steps:

- **Initial integration** ($O(N)$): computes positions and velocities at the next simulation step.
- **Apply boundary conditions** ($O(N)$): takes into account boundary conditions at the edge of the simulation box.
- **Update neighbor list** ($O(N)$): bookkeeping step to keep track of neighboring particles for cutoff optimization.
- **Compute forces**: interaction forces between particles are computed according to the experiment-specific

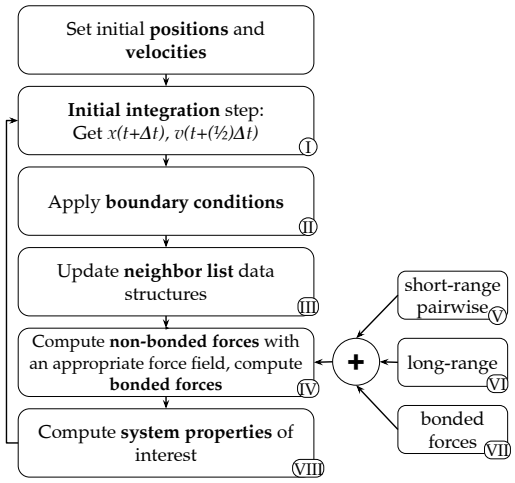

Figure 1: High-level structure of an MD simulation timestep

TABLE 1: Steps of a LAMMPS simulation, corresponding to Figure 1

Computational Tasks	
Bond (VII)	Computation of bonded forces
Comm (IV)	Inter-processor communication of atoms and their properties
Kspace (VI)	Computation of long-range interaction forces
Modify (II)	Fixes† and computes invoked by fixes
Neigh (III)	Neighbor list construction
Output (VIII)	Output of thermodynamic info and dump files
Pair (V)	Computation of pairwise potential
Other	All other tasks

†*Fixes* are operations applied to groups of atoms, e.g., applying constraint forces (e.g., SHAKE bond, and angle constraints [10]), controlling temperature, and enforcing boundary conditions.

force field and approximation strategy, $O(N)$ (bonded), $O(N*npa_{avg})$ (pairwise non-bonded) and $O(N*log(N))$ (long-range).

- **Compute properties** ($O(N)$): computes output thermodynamic properties of interest, such as temperature, volume, and others.

Table 1 breaks down the main computational steps of a LAMMPS simulation and maps them to the general structure of Figure 1.

2.2. LAMMPS Parallelization Strategy

The most effective way to parallelize MD simulations is to ***divide the simulation box into sub-domains***. Each sub-domain computes a *timestep independently*, although it must receive the *updated positions* of atoms near the sub-domain boundaries from its neighbors and *send back* the force contributions acting on said atoms. Additionally, computing *long-range forces* requires *several data exchanges* to compute a 3D FFT on the whole simulation domain.

We leverage LAMMPS's INTEL package [5] to run all experiments on the commodity CPUs, enabling two levels of parallelization. The *MPI level* leverages the spatial decomposition of the problem domain into smaller sub-tasks, and the *OpenMP* parallelization attempts to parallelize the code within a single task. This implementation [5] is generic and parallelizes on both multi- and single-node scenarios. Given our focus on performance and scalability within a single node, we pick the single-node MPI reference for MPI processes parallelization across multiple cores. Each MPI process is assigned to a different core. We experimented with OpenMP and observed that, for our experiments, the OpenMP parallelization (or a combination of the two) was less performing than the MPI-based one in all cases. Similarly, the reference LAMMPS GPU package [4] we leverage also uses MPI parallelization on our GPU instances.

3. MD Experiments Benchmark Suite

We select five MD experiments as representative benchmarks for our suite; this choice depends on their diverse characteristics and their previous use in literature [3]:

Rhodopsin simulates an all-atom rhodopsin protein in solvated lipid bilayer [31] by using a CHARMM force field and NPT time integration with added SHAKE constraints [10]; contrary to all other experiments, it computes long-range non-bonded forces. PPPM is used for long-range force calculation, with a relative error threshold in forces of $1.0 \cdot e^{-4}$.

LJ simulates a 3D Lennard-Jones melt [24]. The force field is LJ with cutoff, hence, no long-range forces are computed.

Chain simulates a bread-spring polymer melt with 100-mer chains. It uses a FENE (Finite Extensible Nonlinear Elastic) bonded potential [26], NVE time integration, and applies a Langevin thermostat [12] to all atoms.

EAM simulates a copper metallic solid with EAM (Embedded Atom Method) [13] potential. The force cutoff and *Neighbor skin* are expressed in Angstroms and are specific to the metal considered in the experiment.

Chute simulates a chute flow of packed granular particles with frictional history potential [11]. The history variant accounts for the tangential displacement between the particles for the duration of the time they are in contact. Unlike all previous benchmarks, this experiment does not leverage Newton's third law to reduce the number of pairwise interactions to compute.

Table 2 summarizes the main features, as explained in Section 2, of these five benchmarks:

- *Cutoff* reports the cutoff between short- and long-range interactions. Based on the measurement units, we report the distance in Angstrom or σ, i.e., the distance at which the particle-particle potential energy is zero.
- *Neighbor skin* reports the neighbor list skin distance through which we decide which particles we consider in our timestep updates.
- *Neighbors/atom* reports the number of atoms within the cutoff range for an atom in the system.
- *pair_modify* allows modifying the parameters of the pairwise interactions in the chosen force field, e.g., interaction coefficients between atoms of different

TABLE 2: Main characteristics of our benchmark suite.

Experiments Taxonomy					
Benchmark	Rhodopsin	LJ	Chain	EAM	Chute
Min atoms	32k	32k	32k	32k	32k
Force field	CHARMM	lj	lj	EAM	gran/hooke/ history
Cutoff	8.0-10.0 Å	2.5 σ	1.12 σ	4.95 Å	1.0 σ
Neighbor skin	2.0 Å	0.3 σ	0.4 σ	1.0 Å	0.1 σ
Neighbors/atom	440	55	5	45	7
pair_modify†	mix arithmetic	-	-	-	-
kspace_style‡	pppm	-	-	-	-
Kspace error‡	1.0 e-4	-	-	-	-
Integration	NPT	NVE	NVE	NVE	NVE

†Only applicable to the force field used in Rhodopsin.
‡Only for Rhodopsin, the only one computing long-range interactions.

types, adding energy offsets or tail corrections. The *mix* keyword determines which formula is used to derive mixed-types coefficients from same-types coefficients. Possible styles are *arithmetic*, *geometric* and *sixthpower* [6].

- *kspace_style* shows the algorithm used to compute the long-range interactions, where applicable.

4. Characterization Methodology

We focus on characterizing the performance of LAMMPS for single compute nodes, evaluating all experiments for different numbers of CPU cores and GPUs. Our goal is to give insights into the architecture behaviors and directions to improve next-generation commodity platforms.

4.1. Importance of single-node performance

Previous work studies multi-node strong scaling for MD and shows that it rapidly becomes inefficient (e.g., 33% parallel efficiency for *LJ* on Haswell with 64 nodes [1]). Scaling out to multiple nodes is also inefficient in terms of memory consumption, considering that an MD simulation has limited memory requirements (2.9 GB for the biggest experiment we run). A single node does not limit the simulation box size for reasonably large experiments, as modern high-performance, general-purpose servers are normally configured with hundreds of GB to TB of DRAM, to support memory-intensive workloads.

While scaling out may still be required in order to meet requirements on simulation turnaround, improving single-node performance translates to a smaller number of nodes required for the same performance target, also increasing per-node main memory utilization (important because DRAM is a major driving cost in modern servers).

We believe that characterizing MD performance on a single node (CPU- or GPU-based) is a significant missing piece in literature and orthogonal to studying scale-out behavior. Our study is a first contribution towards making MD more efficient on future general-purpose systems.

4.2. Platforms and Experimental Flow

We use two systems for our evaluation: 1) the ***CPU instance***, equipped with a dual-socket Intel Xeon Platinum 8358; 2) the ***GPU instance***, equipped with 8 Nvidia V100 GPUs. Table 3 reports a detailed description of the instances.

TABLE 3: CPU and GPU Instances Description

CPU Specs.	CPU Inst.	GPU Inst.
CPU	Intel Xeon Platinum 8358	Intel Xeon Platinum 8167M
Cores	32	26
Threads	64	52
Freq. (turbo)	2.6 GHz (3.4 GHz)	2 GHz (2.4 GHz)
L1 Cache	64 KB per core	32 KB per core
L2 Cache	1 MB per core	
L3 Cache	48 MB shared	35.75 MB shared
Tech. Node	10nm	14 nm
TDP	250W	165W
GPU Specs.	CPU Inst.	GPU Inst.
GPU	-	NVIDIA V100
SM	-	84
Global Mem.	-	16 GB HBM
L2 Cache	-	6 MB shared
L1 Cache	-	128 KB per SM
Frequency	-	1.35 GHz
Tech. Node	-	12nm
TDP	-	300W
Instance Specs.	CPU Inst.	GPU Inst.
Sockets	2 Intel Socket 4189	2 Intel Socket 3647
Memory	1024 GB DDR4	768 GB DDR4
OS	Ubuntu 20.04.4 LTS	
Kernel	Linux 5.13.0-1033-oracle	

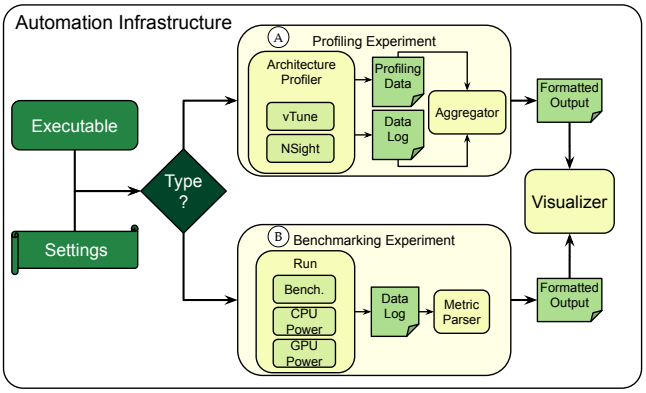

Figure 2: Framework structure to automate the experiments.

Our characterization methodology builds on top of an open-source ***experimental framework*** [32] we developed to ensure repeatability and correctness of evaluations. Figure 2 outlines the overall structure of our framework. For each experiment, the user can define the mode of operation, namely Ⓐ ***profiling*** or Ⓑ ***benchmarking***, and define the parameters space, e.g., number of MPI processes, system sizes, and input of the benchmark. The ***benchmarking*** mode records performance and energy efficiency statistics, while the ***profiling*** mode extracts and stores profiling information from each run. The framework uses *powerstat* for CPUs and *nvidia-smi* for GPUs, running at a fixed sampling rate of 0.5 *seconds* to measure the power consumption. In all experiments, we set each benchmark to run enough timesteps to reach a run time of at least ten seconds, providing enough time to collect power measurements at the given sampling rate. The framework exploits architecture-

specific profiling utilities and includes an aggregator that parses and processes data to produce plots for visualization. We rely on Intel VTune Profiler [2] version 2022.2 and NSightSystems 2022.1 [7] profile on the CPU and GPU instances, respectively.

4.3. Compiler and System Settings

To execute all experiments, we compiled LAMMPS via `Makefile` with the following settings. We used *mpiicpc* for the Intel® MPI Library 2021.6 for Linux; *icpc* version 2021.6.0 (compatible with *gcc* 9.4.0). The compiler flags included are the standard flags in the `Makefile.intel_cpu_intelmpi` makefile. We used these LAMMPS flags for FFT: `-DFFT_MKL -DFFT_SINGLE`. The *mkl* version used was 2022.1.0. The GPU library was compiled with the mpi Makefile (`-m` mpi flag) and with the `-a sm_70` arch flag. We compiled the GPU code with `nvcc` 11.4.120, CUDA version 11.4, and driver version 470.57.02.

5. CPU Experiments Characterization

We first analyze our benchmark suite on the CPU instance (see Table 3). As a first analysis, we *break down execution time* in terms of the *fundamental tasks* we identified in subsection 2.1. In all experiments, we map each MPI process to a separate physical core using Intel's *KMP_AFFINITY*; we do not use hyperthreads and fully fill one socket before using the second one (i.e., only experiments with 64 MPI processes use two sockets). Typical sizes of the simulation box of MD experiments range from some hundred thousand of atoms to a few millions [14, 17, 33]. To cover this space, we report four different experiment sizes ranging between 32*k* and 2048*k* particles. Figure 3 reports the execution time breakdown across our benchmarks, highlighting different behaviors across the suite.

We observe that the number of neighbors per atom, rather than the specific pairwise force field employed, is responsible for increasing the relative runtime of the short-range pairwise forces computation ("Pair" tasks of Figure 3). Although the *Chain* and *LJ* experiments use the same Lennard-Jones pairwise potential, the *LJ* experiment spends over 75% of its runtime computing pairwise interactions if not parallelized, i.e., with one MPI process. Conversely, the *Chain* and *Chute* experiments, having 5 and 7 neighbors per atom, respectively, spend significantly less time in that portion of the timestep. This result is consistent with the fact that the pairwise computation scales as the product of the number of atoms times the average number of neighbors per atom.

A second trend we identify is that, although parallelization generally reduces the time spent in short-range forces computation, this effect is less noticeable for larger experiment sizes. This effect can be explained by considering that for the same cutoff range for a given experiment, in a larger system, we have subdomains with more atoms internal to the subdomain than the surrounding ghost atoms that need to be exchanged among processes. The

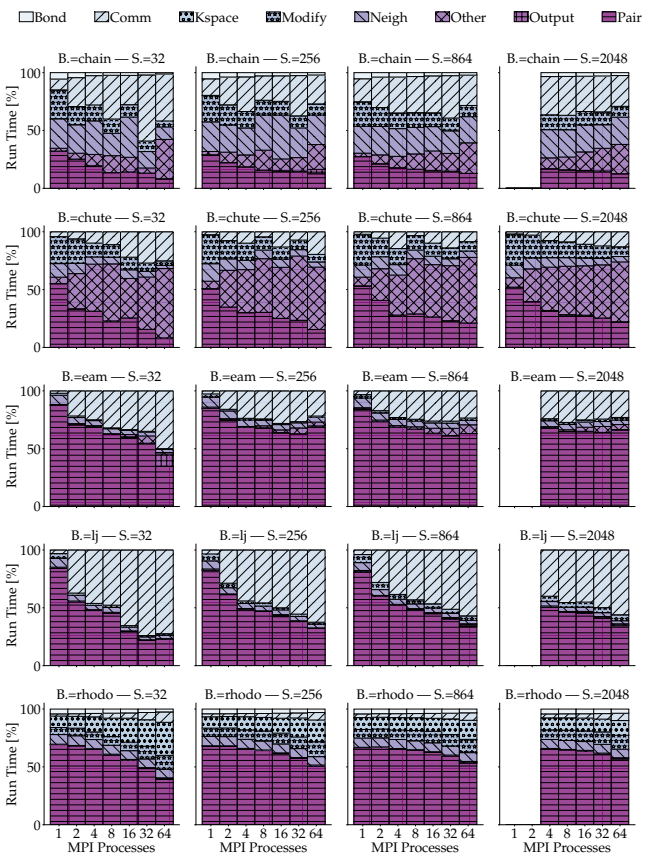

Figure 3: Breakdown of execution time by task for all benchmarks (one per row); problem size increases left-to-right from 32*k* to 2048*k* particles; the number of MPI processes corresponds to the number of CPU cores used.

communication portion starts to dominate for smaller systems with a high degree of parallelization.

Across the suite, only Rhodopsin and Chain include the computation of bonded forces. For these two benchmarks, the time spent computing them is marginal and scales well. This is in line with the findings for Anton 3 [35], in that bonded forces are only worth optimizing once non-bonded forces have been optimized as much as possible.

5.1. MPI Parallelization Scaling Overhead

We leverage our profiling methodology from Section 4 to investigate the *MPI parallelization overhead* on the execution time of all experiments for different problem sizes and number of MPI processes (a.k.a. *MPI ranks*). Figure 4 (top - distribution over all MPI ranks of the percentage of time spent in MPI functions) and Figure 5 (breakdown per MPI function of the time in Figure 4, averaged over all MPI ranks) illustrate the MPI overheads for simulation running for 10*k* timesteps. Figure 5 shows that the *MPI_Init()* function, responsible for initializing the MPI context, takes a considerable portion of the overall time spent in MPI functions. Even for relatively long-running experiments, its impact remains very relevant, despite the function being

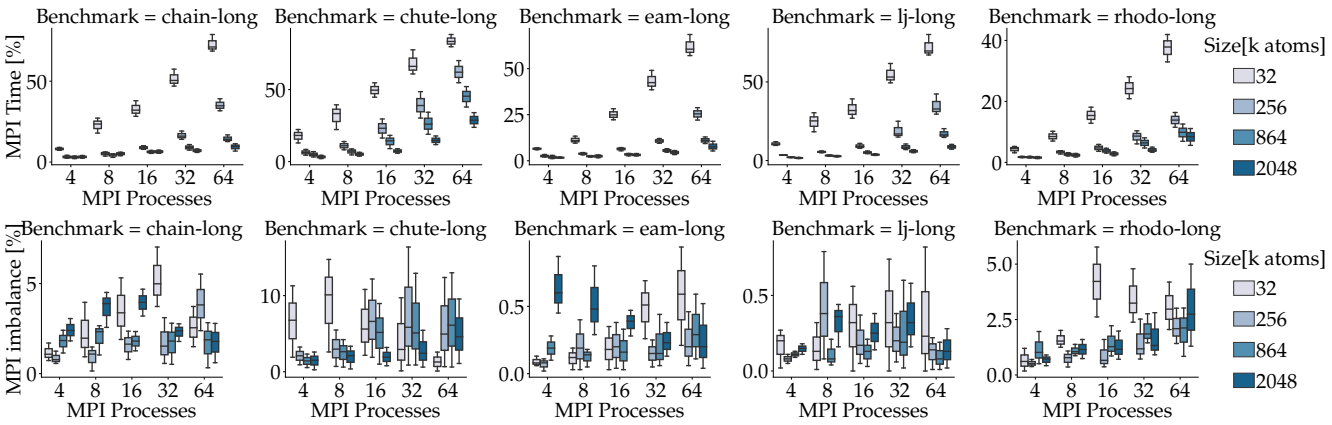

Figure 4: Total MPI overhead and MPI imbalance percentage, averaged for all ranks over total time.

called only once per MPI rank for each run. In order to confirm that the long time spent on *MPI_Init()* is not an artifact due to short simulations, we increased the number of timesteps in our experiment by two orders of magnitude and observed that the time spent in *MPI_Init()* scales with the total execution time. We observe that the time spent executing *MPI_Init*, averaged per MPI rank, increases with the number of MPI processes. This overhead likely depends on the specific MPI library implementation. Investigating performance bottlenecks specific to the Intel MPI library implementation is out of the scope of this paper and left for future work.

Looking at the absolute overhead of MPI on the simulation in Figure 4, we can observe that the overhead decreases if we increase the system size. We also observe that most of the overhead for smaller systems is not due to actual data transfers between MPI processes but rather a consequence of initialization and synchronization overheads. Indeed, the MPI function breakdown of Figure 5 shows that most of the execution time is spent in *MPI_Init()*, as already discussed, and *MPI_Wait()*. *MPI_Send()*, *MPI_Sendrecv()*, and *MPI_Allreduce()* become more prominent for bigger systems but generally decrease with the number of MPI processes. The decrease of the overall impact of MPI calls for systems with more atoms despite the increased data transfer size is an expected behavior. Indeed, the increasing number of atoms in each subdomain (with fixed density d) increases the computation more than the communication between subdomains: assuming each subdomain is a cube of side L, we can approximate the number of data to transfer on each timestep as $O(6(L^2) * cutoff\_range * d)$, whereas the number of iterations of the pairwise potential kernel is $O((L^3) * npa_{avg} * d)$, where npa_{avg} is the number of neighbors per atom on average.

We analyzed the **MPI imbalance** of our workload, which represents the time spent in the MPI library calls when a process waits for data. Figure 4 (bottom) reports the percentage of MPI time spent due to MPI imbalance. We observe that the LJ, Rhodospin, and EAM experiments have a much lower imbalance (EAM and LJ in particular) than Chain and Chute. Indeed, these three benchmarks spend more time in the "Pair" computation, and Rhodospin, the most imbalanced of the three, performs long-range forces calculation. Despite a well-balanced computation, this trend indicates that the addition of more tasks (e.g., fixes and long-range forces) or the more frequent neighbor list construction may result in more imbalance in the the pairwise potential central step Ⓥ.

5.2. CPU Strong Scaling Analysis

Finally, we analyze the workload scalability along timesteps, energy and parallelization efficiency. We define **parallel efficiency** as: $P_n/(P_1 \times n)$, where P_n is performance (*timestep/s*, *TS/s*) with n resources (e.g., MPI processes).

Figure 6 (top) reports indeed the performance, showing how the total efficiency of a timestep depends on many factors. For example, the Rhodopsin experiment has by far the lowest performance (10.7 *TS/s* on a 2 million atoms system), due to having one order of magnitude more neighbors per atom than the EAM and LJ experiments, as well as having to compute the long-range forces contributions. Rather unexpectedly, Rhodopsin scales well with bigger system sizes, meaning that the 3D FFT for the computation of the long-range forces scales up similarly to the other timestep components.

Conversely, the Chute experiment has the best performance for small systems (10697 *TS/s*) but cannot sustain it for larger systems, where LJ and Chain demonstrate superior scalability. Moreover, Chute also does not scale well with MPI parallelization, exhibiting the worse parallel efficiency (as low as 48%) for systems with more than $32k$ atoms (Figure 6, bottom). Looking at the profiling results, we noticed that Chute exhibits the lowest average physical core utilization at 24%, compared to the 48% of LJ, 56% of Chain, 63% of EAM, and 83% of Rhodopsin. This lends support to the idea that parallelization improvements may be possible for this experiment. The low core utilization also explains the difference in power efficiency between Chute and the other benchmarks in Figure 6 (middle).

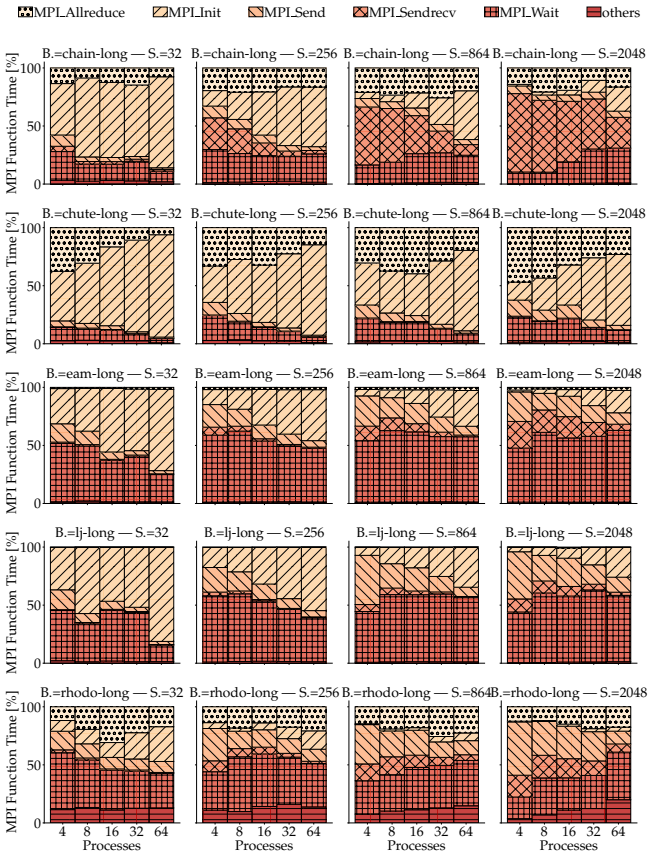

Figure 5: Breakdown of the MPI overhead in terms of the most relevant MPI functions for the time spent by each benchmark (one per row); problem size increases left-to-right.

6. GPU Experiments Characterization

We now characterize the benchmarks of Section 3 execution through the LAMMPS standard GPU package on the GPU instance. Differently from the CPU counterpart, the standard GPU package does not support the *Chute* experiment because of unimplemented features, (gran/hooke pair style). Since implementing this support is beyond the scope of our work, we exclude *Chute* from the GPU analysis. As for the CPU analysis, we discuss the runtime breakdown and the strong scaling across multiple GPUs.

6.1. Task Optimization vs Flexibility on GPUs

Figure 7 reports the tasks *runtime breakdown for the GPU instance*, drawing a different picture than the CPU instance counterpart. First, we can see that the relative runtime of the pairwise forces computation changes drastically: in the Rhodopsin experiment the GPU accelerated version performs this task significantly faster, spanning less than 25% of the total runtime. On the other hand, Figure 8 strongly suggests that not all kernels are equally optimized. The EAM experiment, for example, still spends most of its runtime in pairwise forces computation. The reason

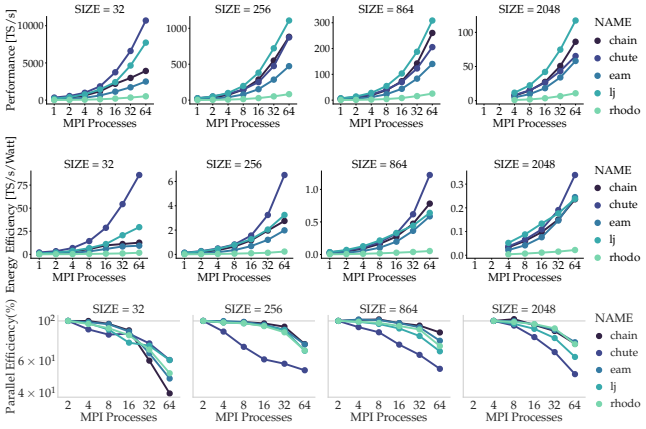

Figure 6: Performance (top), Energy Efficiency (mid), Parallel Efficiency (bottom) of the target benchmarks on the CPU instance for increasing problem sizes (left to right)

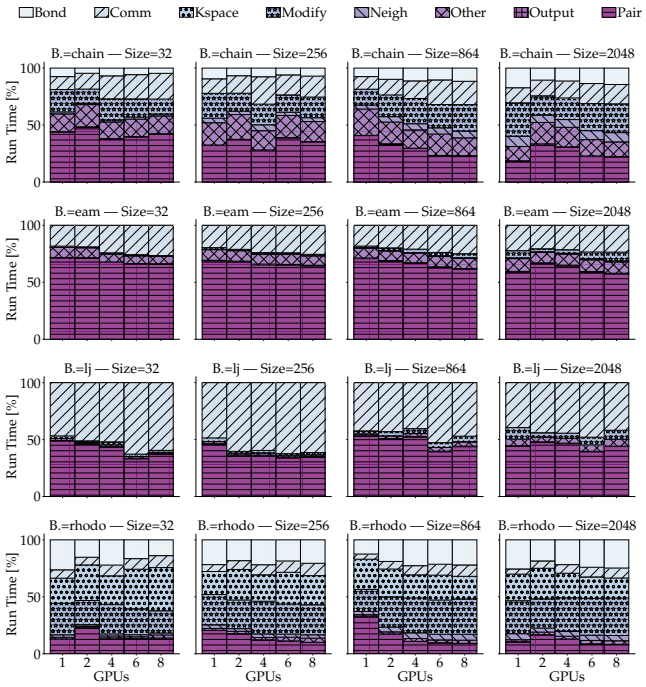

Figure 7: Breakdown of GPU execution time by task for all benchmarks (one per row), problem size increases left-to-right.

stands partially in the additional EAM overhead of splitting the pairwise forces computation into the *k_eam_fast* and *k_energy_fast* kernels. However, their individual runtime on the device is greater than the one of the Rhodopsin counterpart, *k_charmm_long*, as shown in Figure 8. This result suggests that additional optimization of the EAM potential pairwise GPU kernel may be worth investigating.

Another important takeaway for the Rhodopsin benchmark is that the neighbor list construction seems to hit a breaking point for systems larger than 864*k* atoms: while

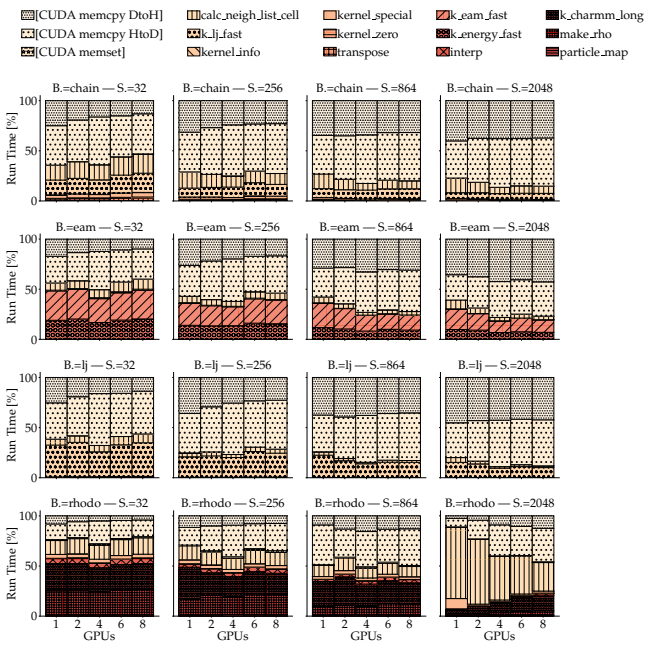

Figure 8: GPU kernels and data movement functions for all benchmarks (one per row), problem size increases left-to-right.

for all systems up to that size the longest-running GPU kernels are related to the computation of the long-range forces (*make_rho* and *particle_map*), for the 2 million atoms simulation *calc_neigh_list_cell* becomes prevalent. Moreover, we observe that the *Modify* section for the Rhodopsin experiment has become more relevant, although we see no counterpart in the GPU kernels. The reason is the lack of a GPU implementation for the SHAKE constraints; hence the CPU is in charge in the current LAMMPS version. Considering that it has now become a more relevant portion of the timestep runtime, it seems that accelerating this computation on the GPU may be a viable next step in optimizing the GPU package performance.

Lastly, we note how the majority of the time actively spent by the GPU is involved in memory movement primitives, specifically *CUDA memcpy* from host to device and from device to host. This clearly shows that from the perspective of GPU utilization, the amount of computation per communication is sub-optimal, and the first instinct would be to optimize this by porting more computation steps on the GPU. However, this is not always feasible while maintaining the flexibility required to support all computational variations and fixes, as indicated by the example of the SHAKE constraints. Moreover, the GPU acceleration needs to be balanced with the MPI parallelization on the host side. In fact, as stated by the optimization guide for the GPU package in the LAMMPS documentation [4], it is often optimal to assign multiple MPI processes to the same device, increasing utilization by allowing multiple subdomains to time-multiplex the use of the device while still parallelizing the computation on the host side. In this case, the optimal

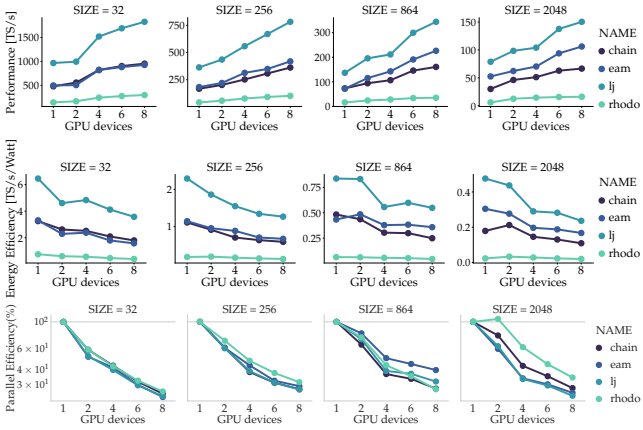

Figure 9: Performance (top), Energy Efficiency (mid), Parallel Efficiency (bottom) of the target benchmarks on the GPU instance for increasing problem sizes (left to right).

balance has to be manually tuned according the capabilities of the host CPU and the GPUs.

6.2. Strong Scaling Analysis: the Data Movement Bottleneck

Similarly to the results reported for the CPU instance, we analyze the strong scaling behavior of the experiments in the GPU-powered instance (Figure 9). Firstly, we observe that the strong scaling on multiple GPU devices is considerably worse than the CPU-only MPI parallelization scaling. In fact, the parallel efficiency (Figure 9, bottom) becomes as low as 23.28%. We empirically tested different numbers of MPI processes per device for different system sizes, and in any case no more than 48 total MPI processes were beneficial, despite having 52 available hardware cores on the CPU. The low parallel efficiency of the GPU stems from the fact that the reference CUDA implementation suffers from low device utilization: therefore, using multiple devices achieves little improvement. This is likely an implementation issue stemming from the fact that LAMMPS supports many different combinations of kernels that are not co-optimized to the fullest extent (this is far from a trivial task). The problem is exacerbated by fractioning the computation domain assigned to each process; we observe that data movement through PCIe occupies most of the runtime, but the the PCIe bandwidth is under-utilized. This analysis indicates that more aggressive software optimizations for the GPU implementation are a very promising direction towards significantly improving single-node MD performance. We do not see fundamental limitations preventing designers from realizing MD algorithms that optimize data movement and better leverage the available bandwidth. One example would be porting more of the code to run on the GPU (instead of relying on the host), such as some of the 'fixes' and integration strategies (e.g., the SHAKE integration strategy for Rhodopsin).

Secondly, we can observe that some benchmarks benefit more than others from GPU parallelization. For example,

EAM outperforms Chain on the GPU instance, contrary to what we observe on the CPU instance. This is likely because the computation of pairwise forces, where EAM spends most of its runtime in both instances, is more suitable for acceleration than the construction of the neighbor lists and the computation of bonded forces, which are more time-consuming for Chain.

Comparing the performance of the CPU and GPU code is not trivial, considering that the CPU of the CPU-only instance is relatively new (released in 2021), while the GPU instance uses both the V100 GPUs and the its own CPU, which has though lower performance and core count than the CPU-only instance. Nevertheless, moving towards a GPU implementation that spends less time on data transfers and allows to fully utilize both the host and the devices is likely the most sensible optimization trajectory to improve the overall performance.

7. Lowering of the Error Threshold

After characterizing the performance of LAMMPS benchmarks with a baseline configuration, we further our analysis by focusing on specific input parameters and on their impact on application behavior. This analysis is not meant to discuss the optimality of the choice of the parameters themselves, which highly depends on the individual experiments; instead, it aims to show how the scaling properties of a sub-section of the algorithm can impact performance.

Firstly, we evaluate the impact of the maximum relative error threshold on forces for the long-range interaction component. We consider the Rhodopsin experiment being the only experiment among the proposed benchmarks using PPPM to estimate long-range forces contribution. We progressively lower the error threshold and run the benchmark for the previously considered system sizes and computational resources. As expected, the runtime spent on long-range forces computations increases by lowering the error threshold, as Figure 11 shows.

Figure 10 showcases how the required precision has a massive impact on performance. Considering a system of 2 millions atoms, an error threshold of $1.0 \cdot e^{-4}$ on 64 MPI processes achieves 10.77 *TS/s*, while lowering the threshold $1.0 \cdot e^{-7}$ causes a performance drop to 3.54 *TS/s*. The parallel efficiency is also negatively affected, dropping from 74.29% to 56.54%, highlighting how the long-range portion of the timestep exhibits worse strong scaling properties, most likely due to the global communication steps required by the 3D FFT.

However, the MPI overhead on the total execution time is reduced, as shown in Figure 14. The threshold of $1.0 \cdot e^{-5}$ and $1.0 \cdot e^{-6}$ exhibits a similar behavior, hence, we exclude the former from the figure. The MPI functions breakdown (Figure 12) shows how for bigger system sizes, the prevalence of *MPI_Send* increases over all other functions, indicating that less time is spent on synchronization between tasks and more time is spent on actual data exchange.

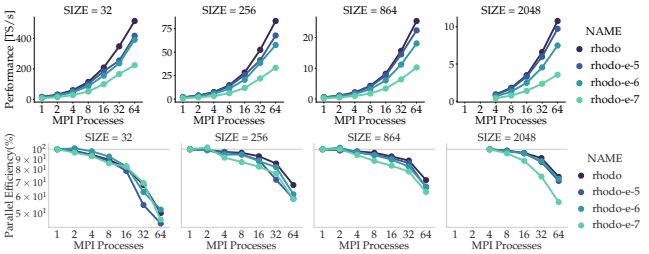

Figure 10: Performance and parallel efficiency of the rhodopsin benchmark on the CPU instance for decreasing problem kspace relative error threshold on forces.

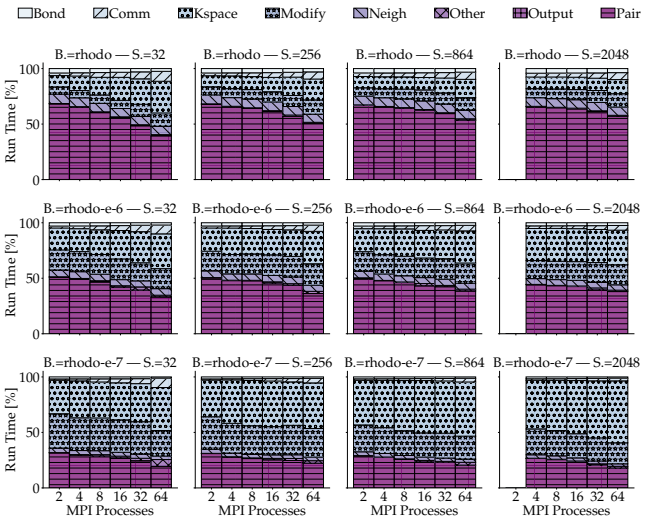

Figure 11: Breakdown of execution time by task for the rhodopsin benchmark on the CPU instance for decreasing problem kspace relative error threshold on forces, indicated as "rhodo-*e*-*" in each plot.

Concerning the GPU instance, the negative impact on performance is even more noticeable in Figure 13. For the same 2 million system we go from 16.09 *TS/s* on 8 GPUs, to 0.46 *TS/s* for an error threshold of $1.0 \cdot e^{-7}$. A lower error threshold implies *CUDA memcpy* from host to device to grow substantially, shadowing all other CUDA API and kernel calls.

8. Scaling Performance to Double Precision

Lastly, we evaluate the impact of floating-point precision on the pairwise non-bonded forces calculations on both CPU and GPU. Usually, the computations exploit single precision for the pairwise forces while accumulating the contributions in double precision. We tune this mixed strategy to use only single or double precision for the whole computation. A command-line parameter in the INTEL package allows choosing between the floating-point precisions, while the GPU requires re-compilation with a dedicated flag. Figure 15 and 16 report the performance for the LJ and Rhodopsin

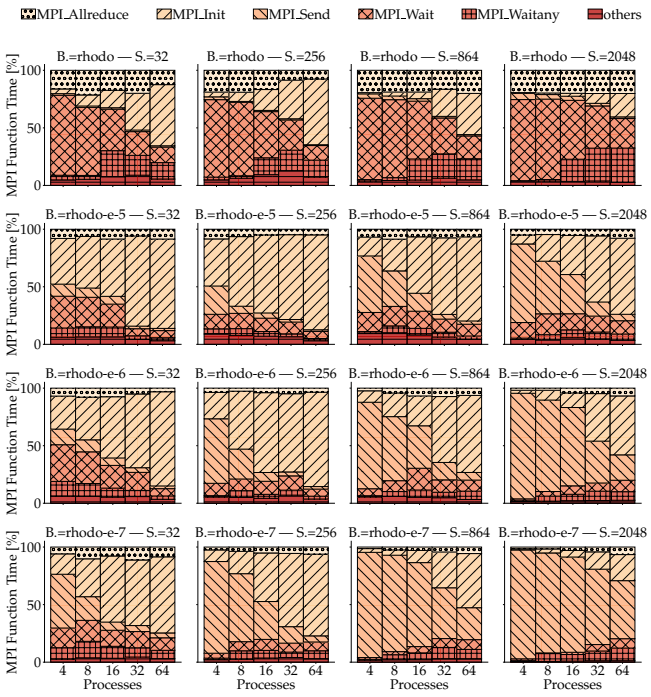

Figure 12: Breakdown of the MPI overhead in terms of the most relevant MPI functions.

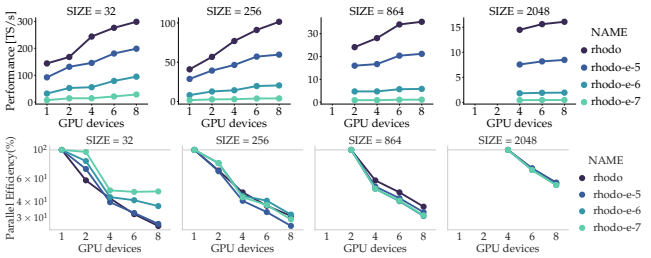

Figure 13: Performance and parallel efficiency of the rhodopsin benchmark on the GPU instance for decreasing problem kspace relative error threshold on forces.

benchmarks with single, double, and mixed precision on CPU and GPU instances.

LJ shows a slight decrease in performance for the double precision version for the CPU instance, while the performance for single and mixed-precision is comparable. For a system of 2048 thousand atoms with 64 MPI processes, the performance drops from 115.2 for the single-precision version to 98.9 *TS/s* for double precision. The GPU versionvshows a more significant change, going from 170.0 *TS/s* on 8 GPUs with single precision to 121.6 *TS/ss* for the double-precision version for the same number of atoms, likely due to the hardware implementation of double precision arithmetics.

Considering the Rhodopsin benchmark, the performance difference is larger for the CPU instance, dropping from 11.5 for single to 8.4 *TS/s* for double. The GPU version shows only a slight decrease in performance for the larger

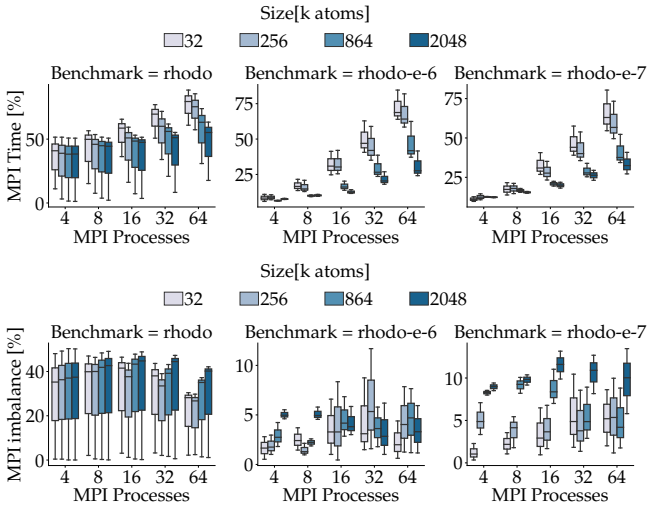

Figure 14: Total MPI overhead (top) and imbalance (bottom) percentage, averaged for all ranks over total time.

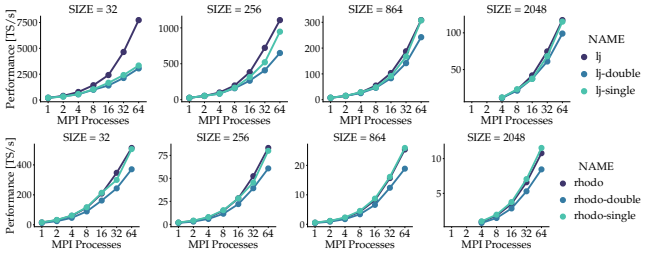

Figure 15: Performance for the LJ and rhodopsin benchmarks with different floating point precision on the CPU instance.

system tested (from 17.1 to 16.5 *TS/s*); the performance (in *TS/s*) further increases when smaller systems and fewer devices are used.

In summary, it is clear that a higher precision negatively affects performance. However, the relative impact of this parameter on the overall runtime is highly dependent on system size and the number and type of devices chosen, with smaller systems being the more affected, and the LJ benchmark on GPU being the most sensitive to this

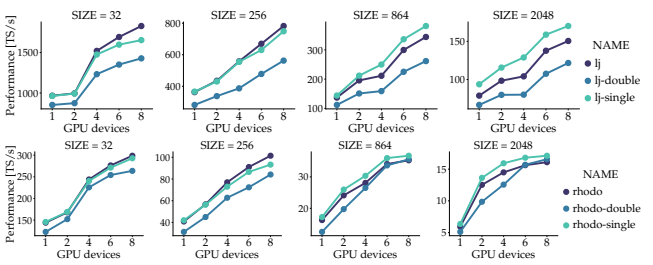

Figure 16: Performance for the LJ and rhodopsin benchmarks with different floating point precision on the GPU instance.

parameter. We also collected the same results for the other four benchmarks, with EAM showing similar behavior to the LJ experiment and Chain behaving similarly to Rhodopsin.

9. Related Work

Several works have investigated LAMMPS scalability and performance over the years. However, none of them presented a comprehensive, fine-grained characterization of single-node commodity general-purpose systems as this work.

A sizable collection of benchmarking results is collected on the LAMMPS website's benchmarking directory [3]. Several weak scaling results are reported for all benchmarks studied in this work on different systems, both desktop machines, CPU-based and GPU-based clusters. Differently, our characterization focuses on single-node performance and scalability, as well as a more fine-grained analysis of application behavior through profiling. We also include energy efficiency measures.

The work by Morinigo et al. [29] investigates the effect of task co-location on MPI performance on the TACC, Helios, and Eagle supercomputers. Instead, our work focuses on the MPI parallelization overhead on a single node, investigating its impact on different benchmarks, required kspace precision, and floating-point precision.

The work of Kondratyuk et al. [25] investigates the performance portability of LAMMPS and GROMACS between NVIDIA's Titan V and AMD's Radeon VII. Unlike this work, we focus on characterizing the CUDA kernel performance and multi-GPU capability of LAMMPS.

The work by Hagerty et al. [18] investigates the portability of LAMMPS performance on several HPC GPU node clusters. The authors also investigate the impact of several options in the Kokkos package. With respect to this work, we choose to characterize the standard GPU package, focusing on multi-GPU strong scaling and characterizing the energy efficiency of the experiments under study.

The work by David Shaw et al. on Anton 2 and Anton 3 [35, 36] represents the most notable example of Domain-Specific Architecture for molecular dynamics simulation. Although this work is pivotal in setting the bar for achievable performance in the post-Moore era, the limited availability of this specialized hardware means that commodity hardware still has a very relevant role in this field. Therefore, we choose commodity hardware as our primary focus in this analysis.

10. Takeaways and Conclusion

We presented an in-depth characterization of relevant MD workloads on general-purpose commodity systems through a benchmark suite of MD experiments exploiting LAMMPS for different interaction potentials. We built a rigorous methodology and automated framework to extensively characterize our benchmarks on single-node commodity CPU and GPU instances, scaling the system under evaluation size from 32k to 2048k atoms. Our analysis provides insights to drive next-generation commodity platforms towards MD optimizations.

Our findings showcase the good scalability of the considered CPU instance across all experiments, especially for big system sizes. Indeed, for Rhodopsin, EAM and LJ with 2 million atoms the parallel efficiency loss normalized per number of MPI processes is roughly constant. Rhodopsin incurs in greater parallel efficiency loss from 32 to 64 MPI processes. Moreover, we characterized the MPI overhead across benchmarks and instance resources. Instead, the evaluated GPU package struggles to scale on multiple devices per single node (up to 8 NVIDIA V100 GPUs), dropping under 30% parallel efficiency for some benchmarks. The reason is the modular style of acceleration for relevant portions of the tasks in the standard GPU package, involving multiple data serialization and transmission steps between CPU and GPU. These scaling results correspond to reduced energy efficiency for the GPU instance, whereas the CPU-only instance yields higher efficiency.

We also perform a sensitivity analysis on experiment-specific parameters. Both lowering the long-forces error threshold and changing the precision to compute pairwise non-bonded forces decrease performance on both instances. Moreover, lowering the error threshold exhibits an MPI overhead reduction and a growth of the CUDA *memcpy* for the CPU and the GPU instance, respectively. On one hand the CPU instance benefits from an increased data exchange volume and reduced tasks synchronization. On the other, the GPU instance suffers from the increased data movement overhead, lowering the accelerator utilization.

Our results show that an experiment like Rhodopsin with 2 million atoms on a single CPU node runs at 2 ns/Day on current commodity hardware. Our GPU node with eight devices reached 2.8 ns/Day. These numbers imply that we are still very far from being able to carry out milliseconds-scale experiments on commodity hardware. The results presented in this breakdown suggest that better utilization of GPUs as accelerators would help bring commodity hardware closer to DSA in terms of achievable performance. In fact, on 2 million atoms systems, the average utilization per GPU reaches only 30%. A more effective parallelization is critical in achieving the full potential of these instances. An promising direction to maintain flexibility while achieving optimal performance is adopting Domain-Specific Languages (DSLs), allowing to write efficient MD code more productively. Examples exist in the literature, but have not seen widespread adoption [9].

Acknowledgments

This work was supported by the Computing Systems Lab, part of the Huawei Zurich Research Center.

This work was supported in part by Oracle Cloud credits and related resources provided by the Oracle for Research program.

References

[1] "Accelerating lammps performance," https://www.lammps.org/workshops/Aug17/pdf/moore.pdf, accessed: 2022-06-23.

[2] "Intel vtune profiler," https://www.intel.com/content/www/us/en/\developer/tools/oneapi/vtune-profiler.html.

[3] "Lammps benchmarks," https://www.lammps.org/bench.html, accessed: 2022-03-23.

[4] "Lammps gpu package," https://docs.lammps.org/Speed_gpu.html, accessed: 2022-06-23.

[5] "Lammps intel package," https://docs.lammps.org/Speed_intel.html, accessed: 2022-06-23.

[6] "Lammps pair_modify command," https://docs.lammps.org/pair_modify.html, accessed: 2022-06-23.

[7] "Nsight systems," https://developer.nvidia.com/blog/nvidia-nsight-systems-2022-1-introduces-vulkan-1-3-and-linux-backtrace-sampling-and-profiling-improvements/, accessed: 2022-03-22.

[8] "Request for proposals for biomolecular simulation time on anton at pittsburgh supercomputing center," https://www.psc.edu/resources/anton/anton-rfp/, accessed: 2022-07-07.

[9] B. Acun, D. J. Hardy, L. V. Kale, K. Li, J. C. Phillips, and J. E. Stone, "Scalable molecular dynamics with namd on the summit system," *IBM journal of research and development*, vol. 62, no. 6, pp. 4–1, 2018.

[10] H. C. Andersen, "Rattle: A "velocity" version of the shake algorithm for molecular dynamics calculations," *Journal of computational Physics*, vol. 52, no. 1, pp. 24–34, 1983.

[11] N. V. Brilliantov, F. Spahn, J.-M. Hertzsch, and T. Pöschel, "Model for collisions in granular gases," *Physical review E*, vol. 53, no. 5, p. 5382, 1996.

[12] R. L. Davidchack, R. Handel, and M. Tretyakov, "Langevin thermostat for rigid body dynamics," *The Journal of chemical physics*, vol. 130, no. 23, p. 234101, 2009.

[13] M. S. Daw and M. I. Baskes, "Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals," *Physical Review B*, vol. 29, no. 12, p. 6443, 1984.

[14] M. De Vivo, M. Masetti, G. Bottegoni, and A. Cavalli, "Role of molecular dynamics and related methods in drug discovery," *Journal of medicinal chemistry*, vol. 59, no. 9, pp. 4035–4061, 2016.

[15] E. D'Arnese, D. Conficconi, M. D. Santambrogio, and D. Sciuto, "Reconfigurable architectures: The shift from general systems to domain specific solutions," in *Emerging Computing: From Devices to Systems*. Springer, 2023, pp. 435–456.

[16] P. P. Ewald, "Die berechnung optischer und elektrostatischer gitterpotentiale," *Annalen der physik*, vol. 369, no. 3, pp. 253–287, 1921.

[17] H. Guterres and W. Im, "Improving protein-ligand docking results with high-throughput molecular dynamics simulations," *Journal of Chemical Information and Modeling*, vol. 60, no. 4, pp. 2189–2198, 2020.

[18] N. Hagerty, V. Melesse Vergara, and A. Tharrington, "Studying performance portability of lammps across diverse gpu-based platforms," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2022.

[19] R. W. Hockney and J. W. Eastwood, *Computer simulation using particles*. crc Press, 2021.

[20] S. A. Hollingsworth and R. O. Dror, "Molecular dynamics simulation for all," *Neuron*, vol. 99, no. 6, pp. 1129–1143, 2018.

[21] A. Hospital, J. R. Goñi, M. Orozco, and J. L. Gelpí, "Molecular dynamics simulations: advances and applications," *Advances and applications in bioinformatics and chemistry: AABC*, vol. 8, p. 37, 2015.

[22] G. Jaeger, "What in the (quantum) world is macroscopic?" *American Journal of Physics*, vol. 82, no. 9, pp. 896–905, 2014.

[23] V. Jansoone, "Dielectric properties of a model fluid with the monte carlo method," *Chemical Physics*, vol. 3, no. 1, pp. 78–86, 1974.

[24] J. E. Jones, "On the determination of molecular fields.—i. from the variation of the viscosity of a gas with temperature," *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, vol. 106, no. 738, pp. 441–462, 1924.

[25] N. Kondratyuk, V. Nikolskiy, D. Pavlov, and V. Stegailov, "Gpu-accelerated molecular dynamics: State-of-art software performance and porting from nvidia cuda to amd hip," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 312–324, 2021.

[26] K. Kremer and G. S. Grest, "Dynamics of entangled linear polymer melts: A molecular-dynamics simulation," *The Journal of Chemical Physics*, vol. 92, no. 8, pp. 5057–5086, 1990.

[27] B. A. Luty, M. E. Davis, I. G. Tironi, and W. F. Van Gunsteren, "A comparison of particle-particle, particle-mesh and ewald methods for calculating electrostatic interactions in periodic molecular systems," *Molecular Simulation*, vol. 14, no. 1, pp. 11–20, 1994.

[28] G. McKenna, "Performance analysis and optimisation of lammps on xcmaster, hpcx and bluegene," *MSc, University of Edinburgh, EPCC*, 2007.

[29] J. A. Moríñigo, P. García-Muller, A. J. Rubio-Montero, A. Gómez-Iglesias, N. Meyer, and R. Mayo-García, "Benchmarking lammps: sensitivity to task location under cpu-based weak-scaling," in *Latin American High Performance Computing Conference*. Springer, 2018, pp. 224–238.

[30] I. Ohmura, G. Morimoto, Y. Ohno, A. Hasegawa, and M. Taiji, "Mdgrape-4: a special-purpose computer system for molecular dynamics simulations," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2021, p. 20130387, 2014.

[31] K. Palczewski, T. Kumasaka, T. Hori, C. A. Behnke, H. Motoshima, B. A. Fox, I. L. Trong, D. C. Teller, T. Okada, R. E. Stenkamp *et al.*, "Crystal structure of rhodopsin: Ag protein-coupled receptor," *science*, vol. 289, no. 5480, pp. 739–745, 2000.

[32] F. Peverelli, D. Conficconi, D. B. Bartolini, A. Scolari, and M. D. Santambrogio, "Lammps benchmarking repository," https://github.com/necst/lammps-benchmarks, 2022.

[33] O. M. Salo-Ahen, I. Alanko, R. Bhadane, A. M. Bonvin, R. V. Honorato, S. Hossain, A. H. Juffer, A. Kabedev, M. Lahtela-Kakkonen, A. S. Larsen *et al.*, "Molecular dynamics simulations in drug discovery and pharmaceutical development," *Processes*, vol. 9, no. 1, p. 71, 2020.

[34] M. Schaffner and L. Benini, "On the feasibility of fpga acceleration of molecular dynamics simulations," *arXiv preprint arXiv:1808.04201*, 2018.

[35] D. E. Shaw, P. J. Adams, A. Azaria, J. A. Bank, B. Batson, A. Bell, M. Bergdorf, J. Bhatt, J. A. Butts, T. Correia *et al.*, "Anton 3: twenty microseconds of molecular dynamics simulation before lunch," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–11.

[36] D. E. Shaw, J. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton *et al.*, "Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 41–53.

[37] K. S. Shim, B. Greskamp, B. Towles, B. Edwards, J. Grossman, and D. E. Shaw, "The specialized high-performance network on anton 3," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 1211–1223.

[38] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson, "A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters," *The Journal of chemical physics*, vol. 76, no. 1, pp. 637–649, 1982.

[39] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen *et al.*, "Lammps-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022.

[40] K. Vanommeslaeghe, E. Hatcher, C. Acharya, S. Kundu, S. Zhong, J. Shim, E. Darian, O. Guvench, P. Lopes, I. Vorobyov *et al.*, "Charmm general force field: A force field for drug-like molecules compatible with the charmm all-atom additive biological force fields," *Journal of computational chemistry*, vol. 31, no. 4, pp. 671–690, 2010.

[41] T. Wang, T. Geng, X. Jin, and M. Herbordt, "Accelerating ap3m-based computational astrophysics simulations with reconfigurable clusters," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 181–184.

[42] C. Wu, T. Geng, S. Bandara, C. Yang, V. Sachdeva, W. Sherman, and M. Herbordt, "Upgrade of fpga range-limited molecular dynamics to handle hundreds of processors," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 142–151.

Appendix

The code is available at [32] for the Artifact available with the latest updates, with the DOI: https://doi.org/10.5281/zenodo.7153144.

1. Abstract

This brief appendix describes how to download and test our LAMMPS benchmarking framework.

2. Artifact check-list (meta-information)

- **Algorithm:** Molecular Dynamics simulation
- **Program:** LAMMPS
- **Compilation:** C++, CUDA, Makefile, CMake (LAMMPS)
- **Binary:** LAMMPS binaries are required
- **Data set:** available in the LAMMPS repository under 'lammps/bench'
- **Run-time environment:** Linux
- **Hardware:** Intel CPU, NVIDIA GPU
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **Publicly available?: yes**
- **Code licenses (if publicly available)?:** MIT License

3. Description

3.1. How to access. Code publicly available at https://github.com/francesco-peverelli/lammps-benchmarks.git .

3.2. Hardware dependencies.

- requires an Intel CPU to run VTune profiling
- requires a CUDA-capable GPU to run and profline GPU experiments

3.3. Software dependencies.

- Requires CUDA >= 11.4 to run GPU experiments
- Requires the powerstat CPU profiling tool
- Requires the nvidia-smi and NVIDIA Nsight Systems utilities
- Requires python 3, and the pandas and seaborn python packages

3.4. Data sets.

- dataset available in the LAMMPS repository under 'lammps/bench'

4. Installation and Benchmarking Flow

- verify the installation of the dependencies provided in this appendix
- clone the lammps repository; This tool assumes that your LAMMPS software is located at '../lammps' relative to this repository. If your LAMMPS/ input files are in a different location, modify the run and profiling scripts accordingly.
- configure, compile and install LAMMPS with the packages that you wish to benchmark/profile
- clone https://github.com/francesco-peverelli/lammps-benchmarks.git
- no further build steps are required, modify the run and profiling scripts indicated below according to the experiments you want to run

In the repo we collect results of different benchmarks and software on a range of architectures. A collection of 'run_<RUN TYPE>.sh' and 'run_<RUN TYPE>.py' scripts are available to run different experiment setups on CPU and GPU. A collection of 'profile_<RUN TYPE>.sh' and 'profile_<RUN TYPE>.py' are available to manage profiling runs. These scripts are intended to be modified by the user to set up the desired experiments,

The 'lammps' and 'lammps_gpu' directories will contain the run results of the performance measurement experiments. If the content of the 'runs.csv' file within this directory does not match the repo state, other utilities such as graph generation may not work properely.

'run_wrapper.py' is used internally to run the benchmarks. More information on the options for these scripts is available in the repo's README

5. Evaluation and expected results

The benchmarking results heavily depend on the architecture under test. After succesfully running a benchmarking script, the runs are added under the benchmark's directory in a 'run.csv' file (e.g., lammps/runs.csv'). To visualize the results, edit the script in the benchmark's directory called 'generate_all_charts4paper.py', specifying the experiments configurations you ran for which you want to produce the graphs. Examples are already provided, however you will need to comment out the graph rendering for all the experiments you have not run. Similarly. the results for the profiling runs will be available under '<bench_name>/prof'. Additional scripts are available to post-process the profiling results, enabling graph generation for them as well. Run 'aggregate_mpi_data.py' and 'parse_task_breakdown.py' for MPI profiling data, and 'generate_report.sh' for VTune profiling post-processing. Run 'aggregate_gpu_data.py', 'parse_task_breakdown.py' and 'generate_report.sh' for GPU experiments.

6. Experiment customization

To customize the experiments run, edit the run and profiling scripts. The lists at the top of each script control the number of devices used, the benchmarks run and the number of iterations. The corresponding input data must be present in your local LAMMPS directory.

7. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html