Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS

Szilárd Páll ⓘ ; Artem Zhmurov; Paul Bauer ⓘ ; Mark Abraham ⓘ ; Magnus Lundborg ⓘ ; Alan Gray;
Berk Hess ✉ ⓘ ; Erik Lindahl ✉ ⓘ

Check for updates

CrossMark

View Online

Export Citation

Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS

View Online   Export Citation   CrossMark

Szilárd Páll,[1] (iD) Artem Zhmurov,[1] Paul Bauer,[2] (iD) Mark Abraham,[2] (iD) Magnus Lundborg,[3] (iD) Alan Gray,[4] Berk Hess,[2,a)] (iD) and Erik Lindahl[2,5,a)] (iD)

AFFILIATIONS

[1] Swedish e-Science Research Center, PDC Center for High Performance Computing, KTH Royal Institute of Technology, 100 44 Stockholm, Sweden
[2] Science for Life Laboratory, Department of Applied Physics, Swedish e-Science Research Center, KTH Royal Institute of Technology, Box 1031, 171 21 Solna, Sweden
[3] ERCO Pharma AB, Stockholm, Sweden
[4] NVIDIA Corporation, Reading, United Kingdom
[5] Science for Life Laboratory, Department of Biochemistry and Biophysics, Stockholm University, Box 1031, 171 21 Solna, Sweden

Note: This paper is part of the JCP Special Topic on Classical Molecular Dynamics (MD) Simulations: Codes, Algorithms, Force Fields, and Applications.
[a)] Authors to whom correspondence should be addressed: hess@kth.se and erik.lindahl@scilifelab.se

ABSTRACT

The introduction of accelerator devices such as graphics processing units (GPUs) has had profound impact on molecular dynamics simulations and has enabled order-of-magnitude performance advances using commodity hardware. To fully reap these benefits, it has been necessary to reformulate some of the most fundamental algorithms, including the Verlet list, pair searching, and cutoffs. Here, we present the heterogeneous parallelization and acceleration design of molecular dynamics implemented in the GROMACS codebase over the last decade. The setup involves a general cluster-based approach to pair lists and non-bonded pair interactions that utilizes both GPU and central processing unit (CPU) single instruction, multiple data acceleration efficiently, including the ability to load-balance tasks between CPUs and GPUs. The algorithm work efficiency is tuned for each type of hardware, and to use accelerators more efficiently, we introduce dual pair lists with rolling pruning updates. Combined with new direct GPU–GPU communication and GPU integration, this enables excellent performance from single GPU simulations through strong scaling across multiple GPUs and efficient multi-node parallelization.

I. INTRODUCTION

Molecular dynamics (MD) simulation has had tremendous success in a number of application areas in the past two decades, in part due to hardware improvements that have enabled studies of systems and timescales that were previously not feasible. These advances have also made it possible to introduce better algorithms, and longer simulations have enabled more accurate calibration of force fields against experimental data, all of which have contributed to increasing trust in computational studies. However, the high computational cost of evaluating forces between all particles combined with integrating over short time steps (~2 fs) has led to fundamental challenges for the field as the speed of individual processor cores is no longer increasing. Without algorithms that can better exploit

new parallel hardware, the timescales accessible in simulations will hit a brick wall. Unlike some other fields, improving resolution by increasing the model detail, e.g., with quantum effects or increasing the size of the system, cannot replace reaching longer timescales. In most cases, molecular dynamics targeting a specific application depends critically on achieving faster simulations by reducing the time each MD step takes.

One successful recent approach has been the introduction of enhanced sampling based on ensembles of simulations. When combined with parallelization of individual runs, this makes it possible to use the largest high performance computing (HPC) resources in the world to study even small systems. However, even for HPC systems, a high rate of producing trajectories is imperative to sample dynamics covering adequate timescales, which means cost-efficiency and throughput are of paramount importance.[1]

The design choices in GROMACS are guided by a bottom-up approach to parallelization and optimization, partly due to the code's roots of high performance on cost-efficient hardware. This is not without challenges; good arguments can be made for focusing either top-down on scaling or just sticking to single-graphics processing unit (GPU) simulations. However, by employing state-of-the-art algorithms and efficient parallel implementations, the code is able to target hardware and efficiently parallelize from the lowest level of SIMD (single instruction, multiple data) vector units to multiple cores and caches, accelerators, and distributed-memory HPC resources.

We believe that this approach makes great use of limited compute resources to improve research productivity,[2,3] and it is increasingly enabling higher absolute performance on any given resource. Exploiting low-level parallelism can be tedious and has often been avoided in favor of using more hardware to achieve the desired time-to-solution. However, the evolution of hardware is making this trade-off increasingly difficult. The end of microprocessor frequency scaling and the consequent increase in hardware parallelism means that targeting all levels of parallelism is a necessity rather than option. The MD community has been at the forefront of investing in this direction,[4–7] and our early work on scalable algorithms,[8] fine-grained parallelism,[9] and low-level portable parallelization abstractions[10] has been previous steps on this path.

Accelerators such as GPUs are expected to provide the majority of raw floating point operations per second (FLOPS) in upcoming exascale machines. However, the impact of GPUs can also be seen in low- to mid-range capacity computing, especially in fields such as MD that have been able to utilize the high instruction throughput as well as single precision capabilities; this has had particularly high impact in making consumer GPU hardware available for scientific computing.

While algorithms with large amounts of fine-grained parallelism are well-suited to GPUs, tasks with little parallelism or irregular data access are better suited to central processing unit (CPU) architectures. Accelerators have become increasingly flexible but still require host systems equipped with a CPU. While there has been some convergence of architectures, the difference between latency- and throughput-optimized functional units is fundamental, and utilizing each of them for the tasks at which they are best suited requires *heterogeneous parallelization*. This typically employs the CPU also for scheduling work, transferring data, and launching computation on the accelerator, as well as inter- and intra-node communication.

Accelerator tasks are launched asynchronously using APIs such as CUDA, OpenCL, and SYCL to allow concurrent CPU–GPU execution. The heterogeneous parallelization model adds complexity that comes at a cost, both in terms of hardware (latency, complex topology) and programmability, but it provides flexibility (every single algorithm does not need to be implemented on the accelerator) and opportunities for higher performance. Heterogeneous systems are evolving fast with very tightly coupled compute units, but the heterogeneity in HPC will remain and is likely best addressed explicitly.

Our first GPU support was introduced in GROMACS 4.5[11] and relied on a homogeneous acceleration by implementing the entire MD calculation on the GPU. The same approach has been used by several codes (e.g., ACEMD,[12] AMBER,[13,14] HOOMD-blue,[15] FENZI,[16] and DESMOND-GPU[17]) and has the benefit that it keeps the GPU busy, avoiding communication as long as scaling is not a concern. However, this first approach also had shortcomings: only algorithms ported to the GPU can be used in simulations, which limits applicability in large community codes. Implementing the full set of MD algorithms on all accelerator frameworks is not practical from porting and maintenance concerns. In addition, our experience showed that outperforming the highly optimized CPU code in GROMACS by only relying on GPUs was difficult, especially in parallel runs where the CPU-accelerated code excels. To make use of GPUs without giving up feature support while providing speedup to as many simulation use-cases as possible, utilizing both CPU and GPU in heterogeneous parallelization was necessary.

Heterogeneous offload is employed by several MD codes (NAMD,[18] LAMMPS,[19] CHARMM,[20] or GENESIS[21]). However, here too, the GROMACS CPU performance provided a challenge: since the tuned CPU SIMD kernels are already capable of achieving iteration rates around 1 ms per step without GPU acceleration, the *relative* speedup of adding an accelerator was less impressive at the time.

To address this, we started from scratch by recasting algorithms into a future-proof form to exploit both GPUs and CPUs (including multiple devices) to provide very high absolute performance while supporting virtually all features no matter what hardware is available. The pair-interaction calculation was redesigned with a cluster pair algorithm[9] to fit modern architectures, which replaces the traditional Verlet list based on particles. Clusters are optimized to fit the hardware, and the classical cut-off setup has evolved into accuracy-based approaches for simulation settings to allow multi-level load balancing and on-the-fly tuning based on system properties. Together with CPU SIMD parallelization and multi-threading,[22] this has allowed efficient offloading of short-range non-bonded calculations to GPUs and brought major gains in performance.[23]

New algorithms and the heterogeneous acceleration framework have made it possible to track the shift toward dense heterogeneous machines and balancing CPU/GPU utilization by offloading more work to powerful accelerators.[1] The most recent release has almost come full circle to allow offloading full MD steps, but this version also supports most features of the MD engine by utilizing both CPUs and GPUs, it targets multiple accelerator architectures, and provides scaling both across multiple accelerator devices and multiple nodes. This bottom-up heterogeneous acceleration approach provides

flexibility, portability, and performance for a wide range of target architectures, ranging from laptops to supercomputers and from CPU-only machines to dense multi-GPU clusters. In the following, we present the algorithms and implementations that have enabled it. These are relatively complex concepts, so we will first discuss the general ideas in Secs. II–IV, after which we dedicate Sec. V to details of core algorithms, returning to performance benchmarks and discussions in the Secs. VI and VII.

II. COMPUTATIONAL CHALLENGES IN MD SIMULATIONS

The core of classical MD is the time-evolution of particle systems by numerically integrating Newton's equations of motion. This requires calculating forces for every time step, which is the main computational cost. While this can be parallelized, the integration step is inherently iterative.

The total force on each particle involves multiple terms: non-bonded pair interactions (typically Lennard-Jones and electrostatics), bonded interactions (e.g., bonds, angles, and torsions), and possibly terms such as restraints or external forces. Given particle coordinates, each term can be computed independently (Fig. 1).

While there are good examples of MD applications with gigantic systems,[24] the most common approach is to keep the simulation size fixed and small to improve absolute performance. Improving the time-to-solution thus requires strong scaling. Historically, virtually all time was spent computing forces, which made it straightforward to parallelize, but well-optimized MD engines now routinely achieve step iteration rates at or below a millisecond.[8,10,11,25] Thus, the MD problem is increasingly becoming latency-sensitive where synchronization, bandwidth, and latency both within nodes and over the network are becoming major challenges for homogeneous as well as heterogeneous parallelization due to Amdahl's law. This has partly been compensated for by increasing density of HPC resources where jobs rely more on intra-node communication than on the inter-node network. This, in turn, has enabled a shift from coarse parallelism using Message Passing Interface (MPI) and domain decomposition (DD) to finer-grained concurrency within force calculation tasks, which is better suited for multicore and accelerator architectures.

Dense multi-GPU servers often make it possible for simulations to remain on a single node, but as the performance has improved, the previous fast intra-node communication has become the new bottleneck compared to the fast synchronization within a single accelerator or CPU. As a side-effect, simulations that a decade ago required extreme HPC resources are now routinely performed on single nodes (often with amazingly cost-efficient consumer hardware). This has commoditized MD simulations, but to explore biological events, we depend on advancing absolute performance such that individual simulations cover dynamics in the hundreds of microseconds, which can then be combined with ensemble-level parallelism to sample multi-millisecond processes.

In the mid 2000s, processors hit a frequency wall and the increases in transistor count were instead used for more functional units, leading to multi- and many-core designs. Specialization has enabled improvements such as wider and more feature-rich SIMD units as well as application-specific instructions, and recent GPUs have also brought compute-oriented application-specific acceleration features. Compute units, however, need to be fed data, but memory subsystems and interconnects have not showed similar improvement. Instead, there has been an increasing discrepancy between the speed of computation and data movement. The arithmetic intensity per memory bandwidth required to fully utilize compute units has increased threefold for CPUs in the last decade and nearly tenfold for GPUs.[26] In particular, most accelerators still rely on communication over the slowly evolving PCIe bus, while their peak FLOP rate has increased five times and GROMACS compute kernel performance grew by up to an order of magnitude.[1] This imbalance has made heterogeneous acceleration and overlapping compute and communication increasingly difficult. This is partly being addressed through tighter host–accelerator integration with NVIDIA NVLink among the first (other technologies include Intel CXL or AMD Infinity Fabric), and this trend is likely to continue.

MD as a field was established in an era where the all-important goal was to save arithmetic operations, which is even reflected in functional forms such as the Lennard-Jones potential (the power-12 repulsion is used as the square of the power-6 dispersion instead of an expensive exponential). However, algorithm design and parallelization is shifting from saving FLOPS to efficient data layout, reducing and optimizing data movement, overlapping communication and computation, or simply recomputing data instead of storing and reloading. This shifts burden of extracting performance to the software, including authors of compilers, libraries, and applications, but it pays off with significant performance improvements, and the resulting surplus of FLOPS suddenly available will enable the introduction of more accurate functional forms without excessive cost.
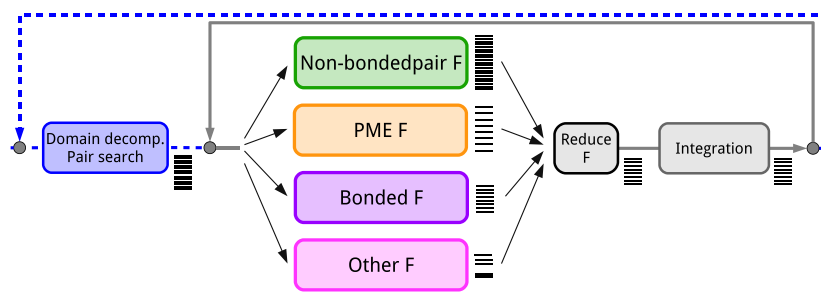
FIG. 1. Structure of an MD step. There is concurrency available for parallelization both for different force terms and within tasks (horizontal bars). The inner (gray) loop only computes forces and integrates, while the less frequent outer loop (blue dashed) involves tasks to decompose the problem.

III. PARALLELIZATION OF MD IN GROMACS

A. The structure of the MD algorithm

The force terms computed in MD are independent and expose task parallelism within each MD step. Force tasks typically only depend on positions from the previous step and other constant data, although domain decomposition (DD) introduces an additional dependency on the communication of particle coordinates from other nodes. The per-step concurrency in computing forces (Fig. 1) is a central aspect in offload-based parallel implementations. The reduction to sum forces for integration acts as a barrier; only when new coordinates become available can the next iteration start. The domain decomposition algorithm on the other hand exposes coarse-grained data parallelism through a spatial decomposition of the particle system. Within each domain, finer-grained data-parallelism is also available (in particular, in non-bonded pair interactions), but to improve absolute performance, it is the total step iteration rate in this high-level flowchart that has to be reduced to the order of 100 μs.

B. Multi-level parallelism

Modern hardware exposes multiple levels of parallelism characterized by the type and speed of data access and communication between compute units. Hierarchical memory and intra- and inter-node interconnects facilitate handling data close to compute units. Targeting each level of parallelism has been increasingly important on petascale architectures, and GROMACS does so to improve performance.[10,22]

On the lowest level, SIMD units of CPUs offer fine-grained data-parallel execution. Similarly, modern GPUs rely on SIMD-like execution called SIMT (single instruction, multiple thread) where a group of threads execute in lockstep (width typically 32–64). CPUs have multiple cores, frequently with multiple hardware threads per core. Similarly, GPUs contain groups of execution units (multiprocessors/compute units), but unlike on CPUs, distributing work across these cannot be controlled directly, which poses load balancing challenges. On the node level, multiple CPUs communicate through the system bus. Accelerators are attached to the host CPU or other GPUs using a dedicated bus. These CPU–GPU and GPU–GPU buses add complexity in heterogeneous systems and, together with the separate global memory, represent some of the main challenges in a heterogeneous setup. Finally, the network is essentially a third-level bus for inter-node communication. An important concern is not only fast access on each level but also the non-uniform memory access (NUMA): moving data between compute units has non-uniform cost. This also applies to intra-node buses as each accelerator is typically connected only to one NUMA domain, not to mention inter-node interconnects where the topology can have large impact on communication latency.

The original GROMACS approach was largely focused on efficient use of low- to medium-scale resources, particularly commodity hardware, through highly tuned assembly (and later SIMD) kernels. The original MPI- (also PVM) based scaling was less impressive, but in version 4.0,[8] this was replaced with a state-of-the-art neutral-territory domain-decomposition[27] combined with fully flexible 3D dynamic load balancing (DLB) of triclinic domains. This is combined with a high-level task decomposition that dedicates

a subset of MPI ranks to long-range Particle Mesh Ewald (PME) electrostatics to reduce the cost of collective communication required by the 3D FFTs, which means multiple-program, multiple-data (MPMD) parallelization. Domain decomposition was initially also used for intra-node parallelism using MPI as well as our own thread-based MPI library implementation.[11] Since the DD algorithm ensures data locality, this has been a surprisingly good fit to NUMA architectures, but it comes with challenges related to exposing finer-grain parallelism across cores and limits the ability to make use of efficient data-exchange with shared caches. Algorithmic limitations (minimum domain size) also restrict the amount of parallelism that can be exposed in this manner. While the design had served well, significant extensions were required in order to target manycore and heterogeneous GPU-accelerated architectures.

The multilevel heterogeneous parallelization was born from a redesign that extended the parallelization to separately target each level of hardware parallelism, first introduced in version 4.6.[10] New algorithms and programming models have been adopted to expose parallelism with finer granularity. Our first major change was to redesign the pair-interaction calculation to provide a flexible and future-proof algorithm with accuracy-based settings and load balancing capabilities, which can target either wide SIMD or GPU architectures. On the CPU front, SIMD parallelism is used for most major time-consuming parts of the code. This was necessitated by Amdahl's law: as the performance of non-bonded kernels and PME improved, previously insignificant components such as integration turned into new bottlenecks. This was made fully portable by the introduction of the GROMACS SIMD abstraction layer, which started as the replacement of raw assembly with intrinsics and now supports a range of CPU architectures using 14 different SIMD instruction sets,[28] with additional ones in development.

To utilize both CPUs and GPUs, intra-node parallelization was extended with an accelerator offload layer and multithreading. The offload layer schedules GPU tasks and data movement to ensure concurrent CPU–GPU execution and it has evolved as more offload abilities were added. OpenMP multithreading was first introduced to improve PME scaling[29] and later extended to the entire MD engine. To allow assembling larger units of computation for GPUs, we increased the size of MPI tasks and have them run across multiple cores instead of dispatching work from a large number of MPI ranks per node. This avoids bottlenecks in scheduling and execution of small GPU tasks. Multithreading algorithms have also gone through several generations with a focus on data-locality, cache-optimizations, and load balancing, improving scalability to larger thread counts. Hardware topology detection based on the hwloc[30] library is used to guide automated thread affinity setting, and NUMA considerations are taken into account when placing threads.

For single-node CPU-only parallelism, execution is still done with sequentially dependent tasks [Fig. 2(a)], which allows relying on implicit dependencies and sharing output across force calculations. Expressing concurrency (Fig. 1) to allow parallel execution over multiple cores and GPU has required explicitly expressing dependencies. The new design uses multi-threading and heterogeneous extensions for handling force accumulation and reduction. On the CPU, per-thread force accumulation buffers are used with
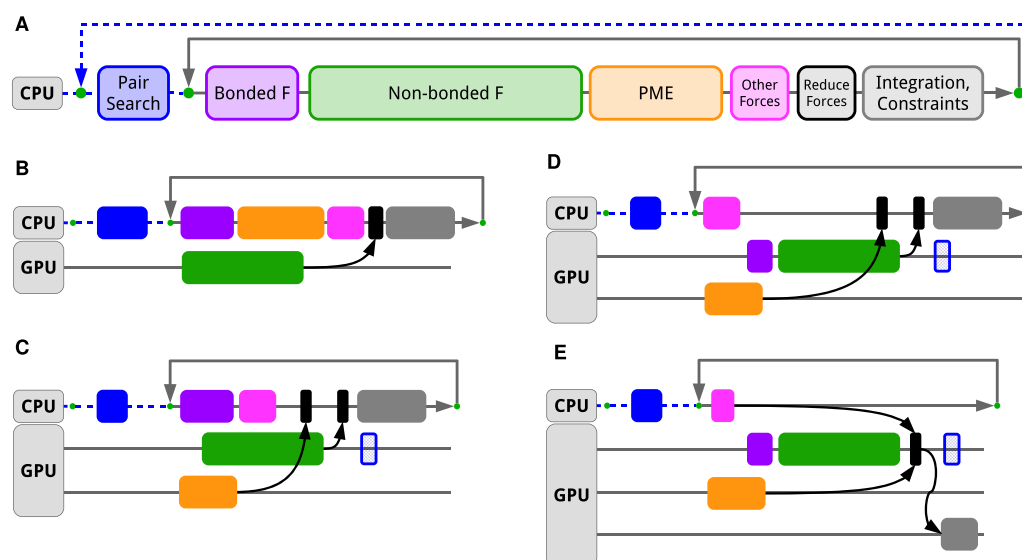
FIG. 2. Execution flows of (a) single-CPU and flavors of the CPU–GPU heterogeneous offload designs. Incremental task offloading is shown for (b) short-range non-bonded interactions (green), (c) PME (orange) and dynamic list pruning (blue cross-hatch), (d) bonded interactions (purple), and (e) integration/constraints (gray). With asynchronous offload, force reduction requires resolving dependencies. The explicit ones are enforced through synchronization (CPU) or asynchronous events (GPU) illustrated by black arrows. Implicit synchronization is fulfilled by the sequential dependencies between tasks on the CPU timeline, as well as those on the same GPU timeline (corresponding to in-order streams).

cache-efficient sparse reduction instead of atomic operations. This is important for bonded interactions where a thread typically only contributes forces to a small fraction of particles in a domain. When combined with accelerators, force tasks can be assigned to either CPU or GPU, with additional remote force contributions received over MPI. With forces distributed in CPU and GPU memories, we use a new reduction tree to combine all contributions. Explicit dependencies of this reduction for the single GPU case are indicated by black arrows in Fig. 2. Fulfilling dependencies may require CPU–GPU transfers to the compute unit that does the reduction, and the heterogeneous schedule is optimized to ensure that these overlap with computation [panels (c) and (d) of Fig. 2].

## IV. HETEROGENEOUS PARALLELIZATION

Asynchronous offloading in GROMACS is implemented using either CUDA or OpenCL APIs and has two main functionalities: explicit control of CPU–GPU data movement and asynchronous scheduling of concurrent task execution and synchronization. This design aims to maximize CPU–GPU execution overlap, reduce the number of transfers by moving data early, keeping data on the accelerator as long as possible, ensuring that transfer is overlapping with computation, and optimize task scheduling for the critical path to reduce the time per step.

A. Offloading force computation

GROMACS initially chose to offload the non-bonded pair interactions to the GPU, while overlapping with PME and bonded

interactions being evaluated on the CPU [Fig. 2(b)]. While this approach requires CPU resources, it has the advantage of supporting domain decomposition and all functionalities, since any special algorithm can be executed on the CPU.[22,31] When combined with DD, interactions with particles not local to the domain depend on halo exchange. This is handled by splitting non-bonded work into two kernels: one for local-only interactions and the other for interactions that involve non-local particles. Based on co-design with NVIDIA, stream priority bits were introduced in the GPU hardware and exposed in CUDA. This made it possible to launch non-local work in a high priority stream to preempt the local kernel and return remote forces early, while the local kernel execution can overlap with communication. Currently, only a single priority bit is available, but increasing this should facilitate additional offloading; this is a less complex solution than a persistent kernel with dynamic workload-switching. The intra-node load balancing together with control and data flow of the heterogeneous setup with short-range force offload (non-bonded and bonded) is illustrated in Fig. 3.

The gradual shift in CPU–GPU performance balance in heterogeneous systems[1] brought the need for offloading further force tasks to avoid the CPU becoming a bottleneck or, from a cost perspective, not needing expensive CPUs. Consequently, we added offload of PME long-range electrostatics both in CUDA and OpenCL. PME is offloaded to a separate stream [Fig. 2(c)] to allow overlap with short-range interactions. The main challenge arises from the two 3D FFTs required. Simulations rely on small grids (typical dimensions 32–256), which has not been an optimization target in GPU FFT libraries, so scaling FFT to multiple GPUs would often not provide meaningful benefits. However, we can still allow multi-GPU scaling
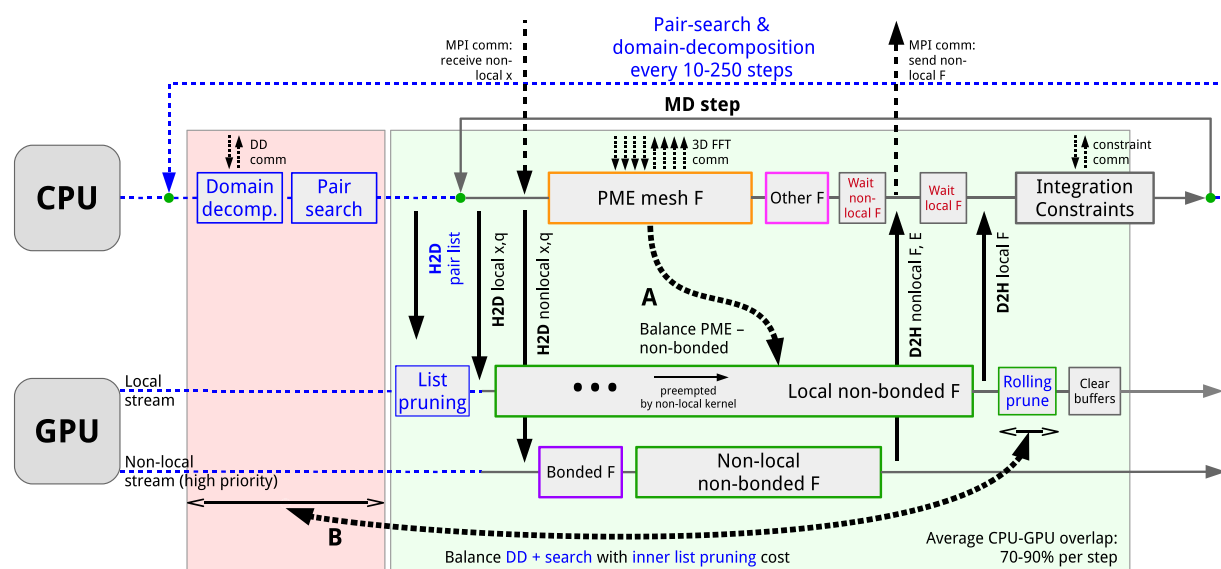
FIG. 3. Multi-domain heterogeneous control and data flow with short-range non-bonded and bonded tasks offloaded. Horizontal lines indicate CPU/GPU timelines with inner MD steps (gray) and pair-search/DD (blue dashed). Data transfers are indicated by vertical arrows (solid ones for CPU–GPU and dashed ones for MPI; H2D is host-to-device and D2H is device-to-host). The area enclosed in green is concurrent CPU–GPU execution, while the red one indicates no overlap (pair search and DD). Task load balancing is used to increase CPU–GPU overlap (dotted arrows) by shifting work between PME and short-range non-bonded tasks (a) and balancing CPU-based pair search/DD with list pruning (b).

by reusing our MPMD approach and placing the entire PME execution on a specific GPU. We have also developed a hybrid PME offload to allow back-offloading the FFTs to the CPU, while the rest of the work is done on the GPU. This is particularly useful for legacy GPU architectures where FFT performance can be low. Additionally, it could be beneficial for strong scaling on next-generation machines with high-bandwidth CPU–GPU interconnects to allow grid transfer overlap and exploiting well-optimized parallel CPU 3D FFT implementations. The last force task to be offloaded was the bonded interactions. Without DD, this is executed on the same stream as the short-range non-bonded task [Fig. 2(d)]. Both tasks consume the same non-bonded layout-optimized coordinates and share force output buffer. With DD, the bonded task is scheduled on the nonlocal stream (Fig. 3) as it is often small and not split by locality.

The force offload design inherently requires data transfer to/from the GPU every step, copying coordinates to, and forces from, the GPU prior to force reduction [black boxes in Figs. 2(b)–2(e)], followed by integration on the CPU. With accelerator-heavy systems, this can render an offload-based setup CPU-limited. In addition, GPU compute to PCIe bandwidth is also imbalanced. High performance interconnects are not common and typically used only for GPU–GPU communication. This disadvantages the offload design as it leaves the GPU idle for part of each step, although this can partly be compensated for with pair list pruning described in Sec. V. Pipelining force computation, transfer, and integration or using intra-domain force decomposition can reduce the CPU bottlenecks.[32] However, slow CPU–GPU transfers are harder to address by overlapping since computation is faster than data movement. For this reason, our recent efforts have aimed to increasingly eliminate

CPU–GPU data movement and rely on direct GPU communication for scaling.

B. Offloading complete MD iterations

To avoid the data transfer overhead, GROMACS now supports executing the entire innermost iteration, including integration, on accelerators [Fig. 2(e)]. This can fully remove the CPU from the critical path and reduces the number of synchronization events. At the same time, the CPU is employed for pair search and domain decomposition (done infrequently), and special algorithms can use the now free CPU resources during the GPU step. In addition to the force tasks performed on the GPU, the data ownership for the particle coordinates, velocities, and forces is now also moved to the GPU. This allows shifting the previous parallelization trade-off and minimize GPU idle time. The inner MD loop, however, still supports forces that are computed on the CPU, and often, the cycle of copying the data from/to the GPU and evaluating these forces on the CPU takes less time than the force tasks assigned to the GPU [Fig. 2(e)]. The CPU can now be considered a supporting device to evaluate forces not implemented on GPU (e.g., CMAP corrections) or those not well-suited for GPU evaluation (e.g., pulling forces acting on a single atom). This keeps the GPU code base to a minimum and balances the load by assigning a different set of tasks to the GPU depending on the simulation setup and hardware configuration. This is highly beneficial both for high-throughput and multi-simulation experiments on GPU-dense compute resources or upgrading old systems with a high-end GPU. At the same time, it also allows making efficient use of communication directly between GPUs, including dedicated

high-bandwidth/low-latency interconnects where available. In our most recent implementation, data movement can be automatically routed directly between GPUs instead of staging communication through CPU memory. When a CUDA-aware MPI library is used, communication operates directly on GPU memory spaces. Our own thread-MPI implementation relies on direct CUDA copies. Additionally, by exchanging CUDA events, it can use stream synchronization across devices, which allows fully asynchronous communication offload leaving the CPU free from both compute and coordination/wait. The external MPI implementation requires additional CPU–GPU synchronization prior to communication but allows the new functionality to be used across multiple nodes. Much of the GPU–GPU communication, either between short-range tasks or between short-range and PME tasks, is of a halo exchange nature where non-contiguous coordinates and forces are exchanged, which requires GPU buffer packing and unpacking operations. Particularly for this, keeping the outer loop of domain decomposition and pair search on the CPU turns out to be a clear advantage, since the index map building is a somewhat complex random access operation, but once complete, the data are moved to the accelerator and reused across multiple simulation time steps.

V. ALGORITHM DETAILS

A. The cluster pair algorithm

The Verlet list[33] and linked cell list[34] algorithms for finding particles in spatial proximity and constructing lists of short-range neighbors were some of the first algorithms in the field and are cornerstones of MD. However, while the Verlet list exposes a high degree of parallelism, its traditional formulation expresses this in an irregular form, which is ill-suited for SIMD-like architectures. To reduce the execution imbalance due to varying list lengths, padding[35] or binning[18] has been used. However, the community has largely converged on reformulating the problem by grouping interactions into fixed size work units instead.[9,13,21,32,36,37]

A common approach is to assign different i-particles to each SIMT thread requiring a separate j particle loaded for each pair interaction. This leads to memory access divergence, which becomes a bottleneck in SIMD-style implementations, even with arithmetically intensive interactions.[38] Sorting particles to increase spatial locality for better caching improves performance,[15,39–41] but the inherently scattered access is still inefficient. Early efforts used GPU textures[18,42] to improve data reuse, but this is hard to control as the effectiveness depends on the size of the j-lists relative to cache.

Given the need for increasing the arithmetic-to-memory operation ratio, we formulated an algorithm that regularizes the problem and increases data reuse. The goal is to load j-particle data as efficiently and rarely as possible and reuse it for multiple i-particles, roughly similar to blocking algorithms in matrix–matrix multiplication. Our cluster pair algorithm uses a fixed number of particles per cluster, and pairs of such clusters rather than individual particles are the unit of computing short-range interactions. Hence, we compute interactions between i-clusters of N particles and a list of j-clusters each of M particles. M is adjusted to the SIMD width, while N allows balancing data reuse with register usage. In addition to a data layout that allows efficient access and that $N \times M$ interactions are calculated for every load/store, the algorithm is easy to adapt to new architectures or SIMD widths. Since the algorithmic efficiency will be higher for smaller clusters, we can also place two sets of the N i-cluster particles in a wide SIMD register of length $2M$, which our SIMD layer supports on all hardware where sub-register load/store operations are efficient. The clusters and pair list are obtained during search using a regular grid in x and y dimensions but binning the resulting z columns into cells with fixed number of particles (in contrast to fixed-size cells) that define our clusters (Fig. 4, left). The irregular 3D grid is then used to build the cluster pair list.[9]

Following the approach used for CPU SIMD, choosing M to match the GPU execution width may seem suitable. However, as GPUs typically have large execution width, the resulting cluster size
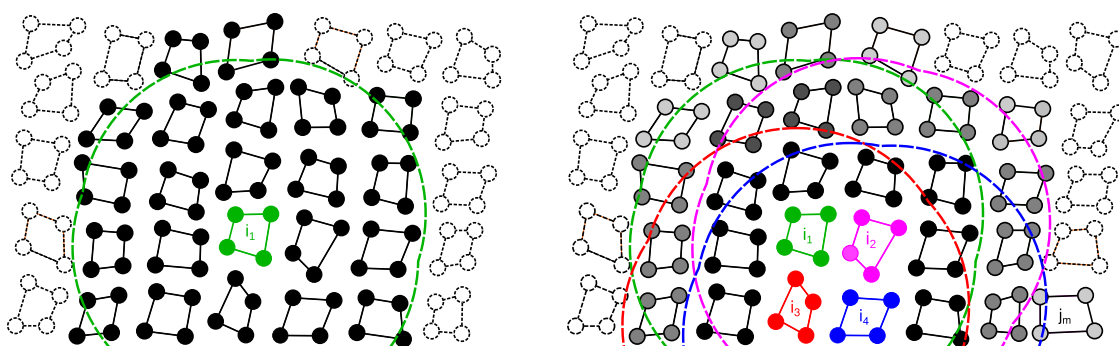
FIG. 4. Cluster pair setups with four particles ($N = 4$ and $M = 4$). Left panel: CPU/SIMD-centric setup. All clusters with solid lines are included in the pair list of cluster i_1 (green). Clusters with filled circles have interactions within the buffered cutoff (green dashed line) of at least one particle in i_1, while particles in clusters intersected by the buffered cutoff that fall outside of it represent an extra implicit buffer. Right panel: hierarchical super-clusters on GPUs. Clusters i_1–i_4 (green, magenta, red, and blue) are grouped into a super-cluster. Dashed lines represent buffered cutoffs of each i-cluster. Clusters with any particle in any region will be included in the common pair list. Particles of j-clusters in the joint list are illustrated by discs filled in black to gray; black indicates clusters that interact with all four i-clusters, while lighter gray shading indicates that a cluster only interacts with 1–3 i-cluster(s), e.g., j_m only with i_4.

would greatly increase the fraction of zero interactions evaluated. The raw FLOP-rate would be high, but of efficiency low. To avoid this, our GPU algorithm calculates interactions between all pairs of an $i - j$ cluster pair instead of assigning the same i- and different j-particles to each GPU hardware thread. Hence, we adjust $N \times M$ to the GPU execution width. However, evaluating all $N \times M$ interactions of a cluster pair in parallel would eliminate the i-particle data reuse. The arithmetic intensity required to saturate modern processors is quite similar across the board,[26,43] so restoring data reuse is imperative. We achieve this by introducing a *super-cluster* grouping (Fig. 4). A *joint pair list* is built for the super-cluster as the union of j-clusters that fall in the interaction sphere of any i-cluster. This improves arithmetic saturation at the cost of some pairs in the list not containing interacting particles, since all j-clusters are not shared. To minimize this overhead, the super-clusters are kept as compact as possible, and the search is optimized to obtain close to cubic cluster geometry—we use an eight-way grouping obtained from a $2 \times 2 \times 2$ cell grouping on the search grid. Even so, the joint j-list would lead to substantial overhead if interactions were computed with all clusters, similar to the challenge with large regular tiling. We avoid this elegantly by skipping cluster pairs with no interacting particles based on an interaction bitmask stored in the pair list. This is illustrated in Fig. 4 where lighter-shaded j-clusters do not interact with some of the i-clusters; for example, j_m can be skipped for $i_1 - i_3$. As $M \times N$ is adjusted to match the execution width, the interaction masks allow efficient divergence-free skipping of entire cluster pairs. Our organization of pair-interaction calculation is similar to that used by others,[14,37] with the key difference that those approaches rely on larger fixed size tiles and use other techniques to reduce the impact of large grouping.

The interaction mask describes a $j - i$ relationship, swapping the order of the standard $i - j$ formulation. Consequently, the loop order is also swapped and our GPU implementation uses an outer loop over the joint j-cluster list and inner loop over the eight i-clusters. This has two main benefits: First, 8 bits per j-cluster is sufficient to encode the interaction mask, instead of needing a variable-length structure. Second, the force reduction becomes more efficient. Since we utilize Newton's third law to only calculate interactions once, we need to reduce forces both for i- and j-particles. At the end of an outer j iteration, all interactions of the j-particles loaded will have been computed and the results can be reduced and stored. At the same time, accumulating the i-particle partial forces requires little memory (8×8 forces) and can be done in registers. Self-exclusions are handled in the interaction kernels, while force-field exclusions are stored in the list with j-clusters as bitmasks and enforced simultaneously with the interaction cutoff, just as the interaction masks.[9] In our typical target systems, on average, approximately four of the eight i-clusters contain interactions with j particles. Hence, about half of the inner loop checks result in skips, and we have observed these to cost 8%–12%, which is a rough estimate of the super-cluster overhead. In comparison, the normal cut-off check has at most 5%–10% cost in the CUDA implementation.

For PME simulations, we calculate the real-space Ewald correction term in the kernel. On early GPU architectures (also CPUs with low FMA FLOPS), tabulated $F \cdot r$ is most efficient. On all recent architectures, we have instead developed an analytical function approximation of the correction force. This yields better performance as it relies on FMA arithmetics despite the >15% increase in kernel instruction count.

Multiprocessor-level parallelism is provided by assigning each thread block a list element that computes interactions of all particles in a super-cluster. To avoid conditionals, separate kernels are used for different combinations of electrostatics and Lennard-Jones interactions and cutoffs, whether energy is required or not. The cluster algorithm has been implemented both in CUDA and OpenCL and tuned for multiple GPU architectures. On NVIDIA and recent AMD GPUs with 32-wide execution, we use an 8×4 cluster setup for 64-wide execution on AMD 8×8 and on Intel hardware with 8-wide execution a 4×2 setup, all with 8-way super-clustering.

B. Algorithmic work efficiency and pair-list buffers

The cluster algorithm trades computing interactions known to evaluate to zero for improved execution efficiency on SIMD-style architectures. To quantify the amount of additional work, we calculate the parallel work efficiency of the algorithm as a fraction of non-zero interactions evaluated. It is worth noting that this metric is ≤1 even for the standard Verlet algorithm as any finite buffer contains non-interacting particles (Fig. 5). In the cluster algorithm, this is augmented with particles outside the buffered sphere, but where another particle in the cluster falls inside it (Fig. 4). The work efficiency depends on the cutoff, buffer size, and geometry/size of the clusters that is optimized during search. The cost of this search is the reason why absolute performance still benefits from larger buffers, just as kernel execution efficiency benefits from the cluster size.

The improved computational efficiency offsets the algorithmic work-efficiency trade-off for all modern architectures.[9] Moreover, particles in the pair list that fall outside the buffered cut-off can be made use of: these provide an extra *implicit buffer* that allows using a shorter explicit buffer when evaluating pair interactions while maintaining the accuracy of the algorithm. One example of a cluster contributing to the implicit buffer is illustrated in Fig. 4. Making use
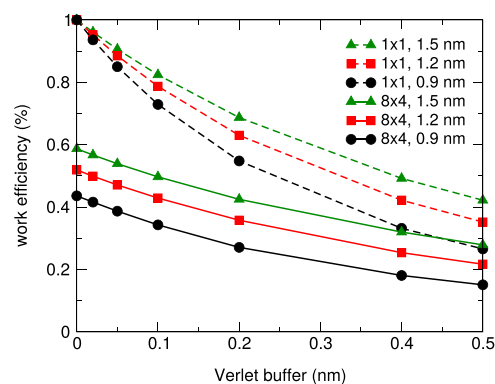
FIG. 5. Algorithmic work efficiency of the particle (1×1) and cluster (8×4) Verlet approaches defined as the fraction of interactions calculated that are within the actual cutoff, shown as a function of buffer size for three common cut-off distances. The trade-off is that larger cluster sizes enable greater *computational* efficiency, and increased buffers enable longer reuse of the pair list.

10 July 2023 07:24:24

of this implicit buffer increases the practical parallel efficiency relative to the standard particle-based algorithm. With a box of SPC/E water where a 0.9 nm cut-off pair list is reconstructed every 40 steps, the 1×1 setup requires a buffer of 0.218 nm to reach the same error tolerance as the 8×4 cluster setup achieves with a 0.105 nm buffer.

We believe that this is a striking example of the importance of moving to tolerance-based settings instead of rule-of-thumb or heuristics to control accuracy. All algorithms in a simulation affect the accuracy of the final results, and while the acceptable error varies greatly between problems, it will be dominated by the worst part of the algorithm—there is little benefit from evaluating only some parts more accurately. To control the effect of missing pair interactions close to the cutoff, our implementation estimates the Verlet buffer for a given upper bound for the error in the energy. The estimate is based on the particle masses, temperature, pair interaction functions, and constraints, also taking into account the cluster setup and its implicit buffering.[9] Since first introduced, we have refined the buffer estimate to account for constrained atoms rotating around the atom they are constrained to rather than moving linearly, which allows tighter estimates for long list lifetimes. The upper bound for the maximum drift can be provided by the user as a tolerance setting in the simulation input. We use 0.005 kJ/mol/ps per atom as default, but the tolerance and hence drift can be arbitrarily small. For the default setting, the implicit buffer turns out to be sufficient for a water system or solvated biomolecule with PME electrostatics and 20 fs pair-list update intervals, and no extra explicit buffer is thus required in this case. The *actual* energy drift caused by these settings is 0.0001 kJ/mol/ps per atom, a factor of 5 smaller than the upper bound. For comparison, typical constraint algorithms result in drifts around 0.0002 kJ/mol/ps per atom, so being significantly more conservative than this will usually not improve the overall error in a simulation.

We see several advantages to this approach and would argue that the field, in general, should move to requested tolerances instead of heuristic settings. First, the user can set a single parameter that is easier to reason about and that will be valid across systems and temperatures. Second, it will enable innovation in new algorithms that maintain accuracy (instead of performance improvements becoming a race toward the least accurate implementation). Finally, since other parameters can be optimized freely for the input and run conditions, we can make use of this for advanced load balancing to safely deviate from classical setups by relying on maintaining accuracy rather than arbitrary settings as described below.

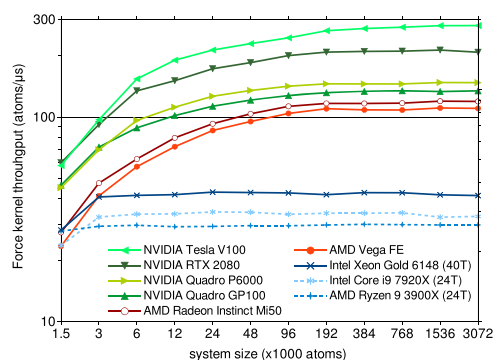### C. Non-bonded pair interaction kernel throughput

The throughput of the cluster-based pair interaction algorithm depends on the number of interactions per particle and hence particle density. This varies across applications from coarse-grained to all-atom bio-molecular systems or liquid-crystal simulations. Hence, we investigate the performance of the pair interaction kernels as a function of particle density. We measure the pair throughput of the nonbonded kernel using a Lennard-Jones system consisting of argon atoms to facilitate comparison across a wide range of application areas from physical to biological systems. The *effective* pair throughput (counting only non-zero interactions) is also influenced by the buffer length and the conditionally enforced cutoff in the GPU

kernels. When comparing CUDA GPU and AVX512 CPU kernels with same-size clusters (identical work efficiency), the raw throughput reaches peak performance already from ~150 pairs per particle on the CPU, while the GPU does not saturate until ~1000 pairs (Fig. 6). This is explained by the increasing *i*-particle data reuse with longer *j*-lists. The effective pair throughput shows a monotonic increase, since more pairs will be inside the cutoff with more particles in the interaction range, as expected from Fig. 5. For typical all-atom simulations, the effective GPU kernel throughput gets close to 100 Ginteractions/s, while the corresponding throughput on a 20-core CPU is an order-of-magnitude lower.

To provide good performance for small systems and enable strong scaling, it is important to achieve high kernel efficiency already at limited particle counts. To illustrate performance as a function of system size for all-atom systems, Fig. 7 shows actual pair throughput for SPC/E water systems with 1 nm cut-off, Ewald electrostatics, 40 step search frequency, and default tolerances. Water typically represents up to 90% of biomolecular systems (hence, it is the particle density of interest), and therefore, nonbonded performance is critical for water. Historically, GROMACS and other codes used special kernels for water, but this no longer works well with SIMD-style architectures. On the other hand, when some particles have only one type of interaction (hydrogens in water typically only have Coulomb interactions), this opens up the possibility for additional optimizations useful on some architectures. The GROMACS CPU kernels achieve peak performance already around 3000 atoms (using up to 40 threads), and apart from the largest devices, within 10% of peak GPU pair throughput is reached around 48k atoms. GPU throughput is up to sevenfold higher at peak than on CPUs, and although it suffers significantly with smaller inputs (up to fivefold lower than peak), for all but the very smallest systems, the GPU kernels reach higher absolute throughput. The challenge for small systems is the overhead incurred from kernel invocation and other fixed-cost operations. Pair list balancing also comes at a slight
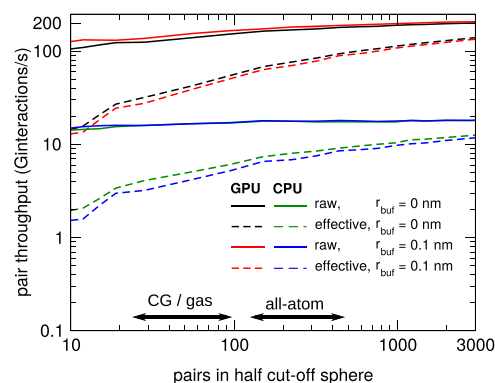
FIG. 6. Non-bonded pair interaction throughput of CUDA GPU and AVX512 CPU kernels as a function of particles in the half cut-off sphere. The raw throughput includes zero interactions, while effective throughput only counts non-zero interactions. Pair ranges typical to all-atom and coarse-grained/gas simulations are indicated. Measurements were done using a 157k particle Lennard-Jones system to minimize input size effects (density ρ = 26 nm^{-3} and σ = 0.3345 nm). Hardware: NVIDIA V100 PCIe GPU and Intel Xeon Gold 6148 CPU.

FIG. 7. Force-only pair interaction kernel performance as a function of input size. The throughput indicates how CPU kernels have less overhead for small systems, while GPU kernels achieve significantly higher throughput from moderate inputs sizes. Multiple generations of consumer and professional CPU and GPU hardware are shown; CUDA kernels are used on NVIDIA GPUs, OpenCL on AMD, and AVX2 and AVX512 on the AMD and Intel CPUs, respectively. All cores and threads were used for CPUs.

cost, and while it is effective when there are enough lists to split, it is limited by the amount of work relative to the size of the GPU. The sub-10k atom systems simply do not have enough parallelism to execute in a balanced manner on the largest GPUs. In contrast, CPU kernels exhibit a slight decrease for large input due to cache effects.

D. The pair list generation algorithm

GROMACS uses a fixed pair list lifetime instead of heuristic updates based on particle displacement, since the Maxwell–Boltzmann distribution of velocities means that the likelihood of requesting a pair list update at a step approaches unity as the size of the system increases. In addition, the accuracy-based buffer estimate allows the pair search frequency to be picked freely (it will only influence the buffer size), unlike the classical approach that requires it to be carefully chosen considering simulation settings and run conditions.

We early decided to keep the pair search on the CPU. Given the complex algorithms involved, our main reason was to ensure parallelization, portability, and ease of maintenance while getting good performance (reducing GPU idle time) through algorithmic improvements. The search uses a SIMD-optimized implementation and, to further reduce its cost, the hierarchical GPU list is initially built using cluster bounding-box distances avoiding expensive all-to-all particle distance checks.[9] A particle-pair distance based list pruning is carried out on the GPU, which eliminates non-interacting cluster pairs. This can also further adapt the setup to the hardware execution width by splitting *j*-clusters; for example, for current Intel GPUs, the search produces a 4 × 4 cluster setup (Fig. 4), which is pruned into 4 × 2 for eight-wide execution. Initial pruning is done the first time the list is processed by a special version of the kernel (using warp collectives for low extra cost), and the pruned list is reused for consecutive MD steps. Depending on the cutoff and buffer length, pruning reduces the list size by 50%–75%.

E. Dual pair list with dynamic pruning

Domain decomposition and pair list generation rely on irregular data access, and their performance has not improved at the same rate as compute kernels. Trading their cost for more pair interaction work through increasing the search frequency has drawbacks. First, as the buffer increases, the overhead becomes large (Fig. 5). The trade-off is also sensitive to inputs and runtime conditions, with a small optimal window. In order to address this, we have developed an extension to the cluster algorithm with a dual pair list setup that uses a longer outer and a short inner list cutoff. The outer list is built very infrequently, while frequent pruning steps produce a pair list based on the inner cutoff, typically with close to zero explicit buffer. As pruning operates on regularized particle data layout produced by the pair search, it comes at a much lower cost (typically <1% of the total runtime) than using a long buffer-based pair list in evaluating pair interactions. This avoids the previous trade-off and reduces the cost of search and DD without force computation overhead. With GPUs, pruning is done in a rolling fashion scheduled in chunks between force computations of consecutive MD steps, which allows it to overlap other work such as CPU integration (Fig. 3).

F. Multi-level load balancing

Both data and task decomposition contribute to load imbalance on multiple levels of parallelism. Sources of data-parallel imbalance include inherently irregular pair interaction data (varying list sizes) and non-uniform particle density (e.g., membrane protein simulations using united-atom lipids) resulting in non-bonded imbalance across domains and non-homogeneously distributed bonded interactions (solvent does not have as many bonds as a protein). With task-parallelism when using MPMD or GPU offload, task load imbalance is also a source of imbalance between MPI ranks or CPU and GPU. Certain algorithmic choices such as pair search frequency or electrostatics settings can shift load between parts of the computation, whether running in parallel or not.

In particular, for small systems, it is a challenge to balance work between the compute units of high-performance GPUs. Especially with irregular work, there will be a kernel "tail" where only some compute units are active, which leads to inefficient execution. In addition, with small domains per GPU with DD, there may be too few cluster lists to process in a balanced manner. To reduce this imbalance and the kernel tail, it would be desirable to control block scheduling, but this is presently not possible on GPUs.[44] Instead, we tune scheduling indirectly through indexing order. The GPU pair interaction work is reshaped by sorting to avoid long lists being scheduled late. In addition, when the number of pair lists is too low for efficient execution for given hardware, we heuristically split lists to increase the available parallelism.

Kernel tail effects can be also be mitigated by overlapping compute kernels. This requires enough concurrent work available to fill idle GPU cores and that it is expressed such that parallel execution is possible. GPU APIs do not allow fine-grained control of kernel execution, and instead, the hardware scheduler decides on an execution strategy. By using multiple streams/queues with asynchronous event-based dependencies, our GPU schedule is optimized to maximize the opportunity for kernel overlap. This reduces the amount of idle GPU resources due to kernel tails as well as those caused

by scheduling gaps during a sequence of short dependent kernels (e.g., the 3D-FFT kernels used in PME).

With PME electrostatics, the split into real and reciprocal space provides opportunities to rebalance work at constant tolerance by scaling the cutoff together with the PME grid spacing.[45] This was first introduced as part of our MPMD approach with an external tool.[11] This load balancing was later automated and implemented as an online load balancer,[22] originally to allow shifting work from the CPU to the GPU. This approach now works remarkably well also with dedicated PME ranks both in CPU-only runs and when using multiple GPUs. The load balancer is automatically triggered at startup and scans through cutoff and PME grid combinations using CPU cycle counters to find the highest-performance alternative as illustrated in Fig. 8 using a biomolecular system typical for applications that aim to reduce the system size simulated.

The load balancer typically converges in a few thousand steps apart from noisy environments such as multi-node runs with network contention. Significant efforts have been made to ensure the robustness of the algorithm. It accounts for measurement noise, avoids cache effects, and mitigates interference of CPU/GPU frequency scaling, and it reduces undesirable interaction with the DD load balancer (which could change the domain size while the cutoff is scaled).

Nevertheless, load balancing comes with a trade-off in terms of increased communication volume. In addition, linearithmic (FFT) or linear (kernel) time-complexity reciprocal-space work is traded for quadratic time complexity real-space work (Fig. 8). To mitigate waste of energy, we impose a cut-off scaling threshold to avoid increasing GPU load in heavily CPU-bound runs. The performance gain from PME load balancing depends on the hardware; with balanced CPU–GPU setups, it is up to 25%, but in highly imbalanced cases, much larger speedups can be observed.[31]

The dual pair list algorithm allows us to avoid most of the drawbacks of shifting work to direct space, since the list pruning

is significantly cheaper than evaluating interactions. This makes the balancing less sensitive and easier to use, and where the previous approach saturated around pair list update intervals of 50–100 steps, the dual list with pruning can allow hundreds of steps, which is particularly useful in reducing CPU load to maximize GPU utilization in runs that offload the entire inner iteration (Fig. 3).

Finally, the domain decomposition achieves load balancing by resizing spatial domains, thereby redistributing particles between domains and shifting work between MPI ranks. With force offload, the use of GPUs is largely transparent to the code, but extensions to the DLB algorithm were necessary. Support for timing concurrent GPU tasks is limited, particularly in CUDA. We account for GPU work in DLB through the wall-time the CPU spends waiting for results, labeled accordingly on the CPU timeline in Fig. 3. However, this can introduce jitter when a GPU is assigned to multiple MPI ranks. GPUs are not partitioned across MPI ranks, but work is scheduled in an undefined order and executed until completion (unless preempted). Hence, the CPU wait can only be systematically measured on some of the ranks sharing a GPU while not on others. This leads to spurious imbalance, and to avoid it, we redistribute the CPU wall-time spent waiting for the GPU evenly across the MPI ranks assigned to the same GPU to reflect the average GPU load and eliminate execution order bias.

VI. PERFORMANCE BENCHMARKS

A. Benchmark systems

To illustrate the real-world performance of the GROMACS heterogeneous parallelization, we use a set of benchmark systems representative of typical biomolecular workloads both in term of size and force-fields. For single GPU benchmarks, we evaluate performance using a small (RNAse) and a medium-sized (GluCl) biomolecular system, both using the AMBER force field. To show multi-GPU ensemble simulation throughput, we use a medium-sized aquaporin membrane protein with coupled simulations that employ the Accelerated Weight Histogram (AWH) enhanced sampling algorithm,[46] while strong benchmarks use a larger $\sim 1 \times 10^6$ atom satellite tobacco mosaic virus (STMV) system, still representative of common workloads and challenging for strong scaling. The latter two benchmark systems use the CHARMM force field and its characteristic settings that notably yield a different short- to long-range nonbonded workload balance and hence different performance behavior compared to AMBER-based simulations that use shorter cutoffs. Further details of the benchmark systems, including input files, can be found in the supplementary material.

Faster hardware has been a blessing for simulations, but as shown in Fig. 9, the improvements in algorithms and software described here have at least doubled performance *for the same hardware* even for older cost-efficient GPU hardware, and with latest-generation consumer cards, the improvement is almost fourfold over the past five years. Given the low-end CPU and high-end GPU combination, new offload modes bring significant performance improvements when offloading either only PME or the entire inner iteration to the accelerator.

Figure 10 shows the impact of different offload setups for single-GPU runs. As expected, the CPU-only run scales with the number of cores. When the non-bonded task is offloaded, the
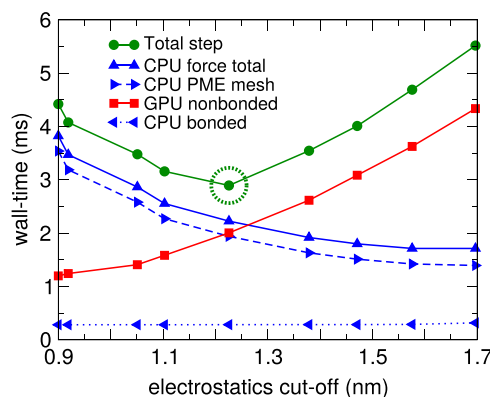
FIG. 8. CPU–GPU load balancing between short- and long-range non-bonded force tasks used. The PME load balancing seeks to minimize total wall-time, here at 1.226 nm (green dashed circle), by increasing the electrostatics direct-space cutoff while also scaling PME grid spacing. This shifts load from the CPU PME task (blue dashed) to the non-bonded GPU task (red). System: alcohol dehydrogenase (95k atoms, 0.9 nm cutoff, default buffer tolerance). Hardware: Intel Core i7-5960X CPU and NVIDIA GTX TITAN GPU.

10 July 2023 07:24:24
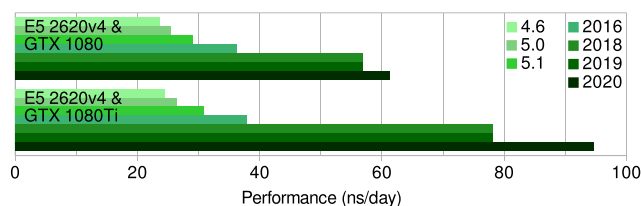
FIG. 9. Performance evolution of GROMACS from the first version with heterogeneous parallelism support on identical hardware. Performance is shown for the 142k atom GluCl benchmark on two hardware configurations with varying CPU–GPU balance using one Intel Xeon E5 2620v4 CPU and NVIDIA GeForce GTX 1080/GTX 1080Ti GPUs.

performance increases significantly for both systems, but it does not saturate even when using all cores—indicating that the CPU is oversubscribed. This is confirmed by the large jump in performance when the PME task too is offloaded. The GPU now becomes the bottleneck for computations, and the curves saturate when enough CPU cores are used—adding more will not aid performance. Consequently, when the bonded forces are also offloaded, there is a performance regression particularly for the membrane protein system with lots of torsions [Fig. 10(b)]. With GPU force tasks taking longer than the CPU force tasks, the data transfers between host and device are no longer effectively overlapping with compute tasks. This is solved by offloading the entire innermost MD loop, including coordinate constraining and updating. This leads to another significant jump in performance, despite the CPU now being mostly idle. To make use of this idle resource, one can move the bonded force evaluation from the GPU back to the CPU. This is beneficial when the entire cycle is faster than the evaluation of non-bonded and PME forces on the GPU. For all results, the crossover points will depend on the system, but it is generally faster to evaluate bonded forces on the CPU when apart from very dense systems where only a single CPU core is available per GPU.
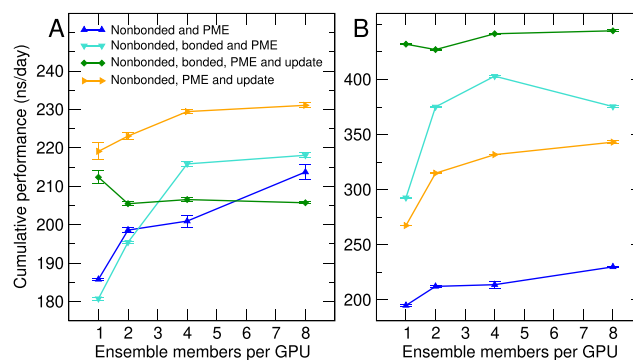
Although it is common to run a single simulation per GPU, it is often not the best way of maximizing cumulative throughput since
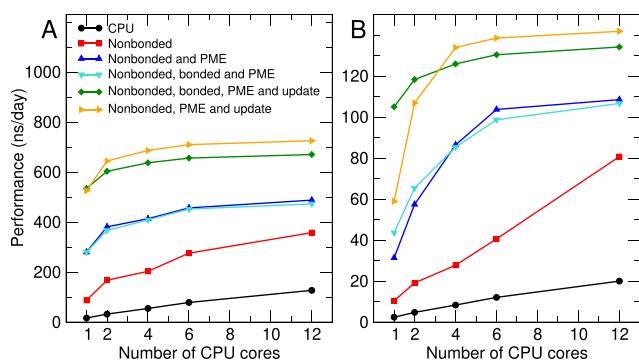
the options to overlap data transfers between CPU and GPU with computational tasks are limited. One way to increase the efficiency even further is to run many uncoupled or loosely coupled trajectories simultaneously on a single GPU. In this case, compute tasks from one trajectory can overlap with the data transfers in another. Figure 11 shows benchmarks for the case of ensemble simulations using the AWH method[46] on a single node equipped with two CPUs and four GPUs. With medium-performance GPUs, the best performance is achieved when a CPU is used for computing the bonded forces, with everything else evaluated on the GPU [Fig. 11(a)], and the most efficient throughput is obtained with four ensemble members running on each GPU. Using the GPU for the full MD loop is still the second best case when only a single run is executed on the GPU, but for many runs per GPU, there are more options for overlapping transfer and compute tasks when using the CPU for updates (integration) and/or bonded forces. However, for the somewhat older GPUs, the difference between the worst- and best-performing cases is only about 25%. When pairing the same older CPUs with recent GPUs ([Fig. 11(b)], the balance changes appreciably, and it is no longer justified to use the CPUs even for the lightest compute tasks. Performing all tasks on the GPU more than doubles performance compared to leaving updates and bonded forces on the CPU. Another advantage is that the throughput does not change significantly with more ensemble members per GPU, which allows for greater flexibility, not to mention that the absolute performance will always be highest when only a single ensemble member is assigned to each GPU.

Finally, the work on direct GPU communication now also enables quite efficient multi-GPU scaling combined with outstanding absolute performance. Figure 12 illustrates the effect of the direct GPU communication optimizations on performance through results from running the Satellite tobacco mosaic virus (STMV) benchmark (1M atoms, 2 fs steps) on up to four compute nodes, each equipped with four NVIDIA Tesla V100 GPUs per node. Intra-node communication uses NVLink and inter-node communication MPI over Infiniband. We believe that this configuration is a good match to

FIG. 10. Single-GPU performance for (a) RNAse (24k atoms) and (b) GluCl ion channel (142k atoms) systems, both with the AMBER99 force-field. Different offload setups are illustrated, with the tasks assigned to the GPU listed in the legend. Hardware: AMD Ryzen 3900X and NVIDIA GeForce RTX 2080 Super.

FIG. 11. Ensemble run cumulative performance as a function of number of accelerated weight histogram walkers executed simultaneously for different offload scenarios. The benchmark system is aquaporin (109 992 atoms per ensemble member, CHARMM36 force-field). The performance was measured on a dual Intel Xeon E5-2620 v4 CPU server with four NVIDIA GeForce GTX 1080 GPUs (a) and four NVIDIA GeForce RTX 2080 GPUs (b).

10 July 2023 07:24:24
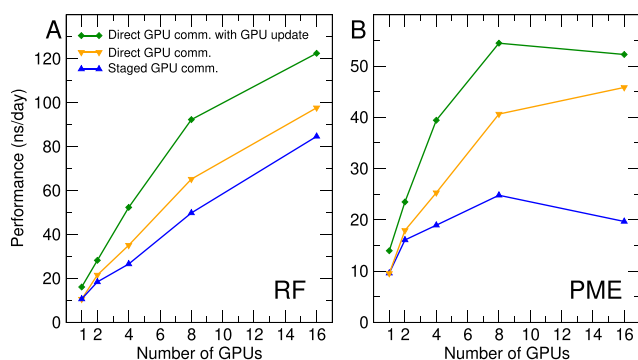
FIG. 12. Multi-GPU and multi-node scaling performance STMV benchmark (1M atoms). Performance when using staged communication through the CPU, direct GPU communication, and the additional benefit of GPU integration are shown. Left panel: when using reaction-field to avoid lattice summation, the scaling is excellent and we achieve iteration rates around 1 ms of wall clock time per step over 4 nodes with 4 GPUs each. Right panel: with PME, the scaling currently becomes limited by the use a single PME GPUs for offloading, but the absolute performance is high (despite the different scales). All runs use 1 MPI task per GPU, except the 2-GPU PME runs that use four MPI ranks to improve task balance.

emerging next-generation HPC systems, which have a focus on good balance for mixed workloads. When using reaction-field instead of PME, the scaling is excellent all the way to 16 GPUs. While this is a less common choice for electrostatics, it highlights the efficiency and benefits of the GPU halo exchange combined with GPU update and shows the performance and scaling possible when avoiding the challenges with small 3D FFTs, extra communication between direct- and reciprocal-space GPUs, and task imbalance inherent to PME. When using PME electrostatics, the relative scaling is good up to 8 GPUs (50% efficiency) when the GPU halo exchange is combined with the direct GPU PME task communication, and there are again clear benefits from combination with the GPU update path. Beyond this, we are currently limited by the restriction of a single PME GPU when offloading lattice summation, which becomes a bottleneck both in terms of communication and imbalance in computation. However, we believe that the absolute performance of 55 ns/day is excellent.

VII. DISCUSSION

Microsecond-scale simulations have not only become routine but eminently approachable with commodity hardware. However, it is only the barrier of entry that has been lowered. Bridging the time scale gap from hundreds of microseconds to millisecond in single-trajectories still requires special-purpose hardware.[47,48] Nevertheless, general-purpose codes have unique advantages in terms of flexibility, adaptability, and portability to new hardware—not to mention that it is relatively straightforward to introduce new special algorithms for including experimental constraints in simulations. There are also great opportunities with using massive-scale resources for efficient ensemble simulations where the main challenge is cost-efficient generation of trajectories. Hence, we believe

that a combination of new algorithms, efficient heterogeneous parallelization, and large-scale ensemble algorithms will characterize MD in the exascale era, and performance advances in the core MD codes will always multiply the advances obtained from new cleaver sampling algorithms.

The flexibility of the GROMACS engine comes with challenges for both developers and users. There is a range of options to tune from algorithmic parameters to parallelization settings, and many of these are related to fundamental shifts toward much more diverse hardware that the entire MD community has to adapt to. Our approach has been to provide a broad range of heuristics-based defaults to ensure good performance out of the box, but by increasingly moving to tolerance-based settings, we aim to both improve quality of simulations and make life easier for users. Still, efficiently using a multitude of different hardware combinations for either throughput or single long simulations is challenging. As described here, many of the steps have been automated, but to dynamically decide the algorithm to resource mapping in a complex compute node (or what code flavor to use) requires a fully dynamic auto-tuning approach.[44] Developing a robust auto-tuning framework and integrating it with the multi-level load balancers is especially demanding due to the complex feature set and broad use-cases of codes such as GROMACS, but it is something we are working actively on.

To reach performance for which ASICs are needed today, MD engines need to be capable of <100 μs iterations. Such iteration rates are possible with GROMACS for small systems, such as the villin headpice (~5000 atoms).[49] Reaching this performance for larger systems such as membrane proteins with hundreds of thousands of atoms will require a range of improvements. In terms of parallelization in GROMACS, improving the efficiency of GPU task scheduling, CPU tasking, and better overlap of communication are necessary. When it comes to algorithms, we expect PME to remain the long-range interaction method of choice at low scale, but the limitations of the 3D FFT many-to-many communication for strong scaling require a new approach. With recent extensions to the fast multipole method,[50] we expect it to become the algorithm of choice for the largest parallel runs. Future technological improvements including faster interconnects and closer on-chip integration and advances in both traditional[3,51] and coarse-grained reconfigurable architectures[52] could allow getting closer to this performance target.

We expect to see several of these advancements in the future. In the meantime, we believe that the present GROMACS implementation provides a major step forward in terms of absolute performance as well as relative scaling by being able to use almost arbitrary-balance combinations of CPU and GPU functional units. It is our hope that this will help enable a wide range of scientific applications on everything from cost-efficient consumer GPU hardware to large HPC resources.

SUPPLEMENTARY MATERIAL

See the supplementary material (heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS) for performance benchmark methodology, topologies, input data, and parameters.

DATA AVAILABILITY

The data that support the findings of this study are available in the supplementary material and in the following repositories: performance benchmarks inputs and methodology are published at https://doi.org/10.5281/zenodo.3893789 (supplementary material) and the source code for multi-GPU scaling is available at https://doi.org/10.5281/zenodo.3890246 (Ref. 53).

REFERENCES

[1] C. Kutzner, S. Páll, M. Fechner, A. Esztermann, B. L. Groot, and H. Grubmüller, "More bang for your buck: Improved use of GPU nodes for GROMACS 2018," J. Comput. Chem. **40**, 2418–2431 (2019).

[2] H. H. Loeffler and M. D. Winn, "Large biomolecular simulation on HPC platforms II. DL POLY, GROMACS, LAMMPS and NAMD," Technical report, STFC, 2012.

[3] M. Schaffner and L. Benini, "On the feasibility of FPGA acceleration of molecular dynamics simulations," Technical report, ETH Zurich, Integrated Systems Lab IIS, 2018; arXiv:1808.04201.

[4] N. Yoshii, Y. Andoh, K. Fujimoto, H. Kojima, A. Yamada, and S. Okazaki, "MODYLAS: A highly parallelized general-purpose molecular dynamics simulation program," Int. J. Quantum Chem. **115**, 342–348 (2014).

[5] W. M. Brown, A. Semin, M. Hebenstreit, S. Khvostov, K. Raman, and S. J. Plimpton, "Increasing molecular dynamics simulation rates with an 8-fold increase in electrical power efficiency," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (IEEE Press, 2016), pp. 82–95.

[6] W. McDoniel, M. Höhnerbach, R. Canales, A. E. Ismail, and P. Bientinesi, "LAMMPS' PPPM long-range solver for the second generation Xeon Phi," in *High Performance Computing, ISC 2017*, Lecture Notes in Computer Science, Vol. 10266, edited by J. Kunkel, R. Yokota, P. Balaji, and D. Keyes (Springer, Cham, 2017).

[7] 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 J. Res. Dev. **62**, 4:1–4:9 (2018).

[8] B. Hess, C. Kutzner, D. Van Der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," J. Chem. Theory Comput. **4**, 435–447 (2008).

[9] S. Páll and B. Hess, "A flexible algorithm for calculating pair interactions on SIMD architectures," Comput. Phys. Commun. **184**, 2641–2650 (2013); arXiv:1306.1737.

[10] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," SoftwareX **1-7**, 19 (2015).

[11] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl, "GROMACS

4.5: A high-throughput and highly parallel open source molecular simulation toolkit," Bioinformatics **29**, 845–854 (2013).

[12] M. J. Harvey, G. Giupponi, and G. D. Fabritiis, "ACEMD: Accelerating biomolecular dynamics in the microsecond time scale," J. Chem. Theory Comput. **5**, 1632–1639 (2009).

[13] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. Generalized born," J. Chem. Theory Comput. **8**, 1542–1555 (2012).

[14] R. Salomon-Ferrer, A. W. Goetz, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald," J. Chem. Theory Comput. **9**, 3878 (2013).

[15] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," J. Comput. Phys. **227**, 5342–5359 (2008).

[16] N. Ganesan, M. Taufer, B. Bauer, and S. Patel, "FENZI: GPU-enabled molecular dynamics simulations of large membrane regions based on the CHARMM force field and PME," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.D. Forum* (IEEE, 2011), pp. 472–480.

[17] M. Bergdorf, S. Baxter, C. A. Rendleman, and D. E. Shaw, "Desmond/GPU performance as of November 2016," Technical Report No. ĿDESRES/TR–2016-01, D. E. Shaw Research, 2016.

[18] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," J. Comput. Chem. **28**, 2618–2640 (2007).

[19] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers—Particle–particle particle-mesh," Comput. Phys. Commun. **183**, 449–459 (2012).

[20] A.-P. Hynninen and M. F. Crowley, "New faster CHARMM molecular dynamics engine," J. Comput. Chem. **35**, 406–413 (2013).

[21] C. Kobayashi, J. Jung, Y. Matsunaga, T. Mori, T. Ando, K. Tamura, M. Kamiya, and Y. Sugita, "GENESIS 1.1: A hybrid-parallel molecular dynamics simulator with enhanced sampling algorithms on multiple computational platforms," J. Comput. Chem. **38**, 2193–2206 (2017).

[22] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl, "Tackling exascale software challenges in molecular dynamics simulations with GROMACS," in *Solving Software Challenges for Exascale*, Lecture Notes in Computer Science Vol. 8759, edited by S. Markidis and E. Laure (Springer, Cham, 2015), pp. 3–27.

[23] C. Kutzner, R. Apostolov, and B. Hess, "Scaling of the GROMACS 4.6 molecular dynamics code on SuperMUC," in International Conference on Parallel Computing-ParCo2013, 2013.

[24] J. R. Perilla and K. Schulten, "Physical properties of the HIV-1 capsid from all-atom molecular dynamics simulations," Nat. Commun. **8**, 15959 (2017).

[25] T.-S. Lee, D. S. Cerutti, D. Mermelstein, C. Lin, S. Legrand, T. J. Giese, A. Roitberg, D. A. Case, R. C. Walker, and D. M. York, "GPU-accelerated molecular dynamics and free energy methods in Amber18: Performance enhancements and new features," J. Chem. Inf. Model. **58**, 2043–2050 (2018).

[26] K. Rupp, "CPU GPU and MIC hardware characteristics over time (2019)," https://github.com/karlrupp/cpu-gpu-mic-comparison (Last viewed May 12, 2020).

[27] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. C. Berendsen, "GROMACS: Fast, flexible, and free," J. Comput. Chem. **26**, 1701–1718 (2005).

[28] E. Lindahl, M. Abraham, B. Hess, and D. van der Spoel, "GROMACS 2020.2 source code," Zenodo. V.2020.2. Dataset. http://doi.org/10.5281/zenodo.3773801

[29] R. Schulz, B. Lindner, L. Petridis, and J. C. Smith, "Scaling of multimillion-atom biological molecular dynamics simulation on a petascale supercomputer," J. Chem. Theory Comput. **5**, 2798–2808 (2009).

[30] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing* (IEEE, 2010), pp. 180–186.

[31] C. Kutzner, S. Páll, M. Fechner, A. Esztermann, B. L. de Groot, and H. Grubmüller, "Best bang for your buck: GPU nodes for GROMACS biomolecular simulations," J. Comput. Chem. **36**, 1990–2008 (2015); arXiv:1507.00898.

10 July 2023 07:24:24

[32] J. E. Stone, A.-P. Hynninen, J. C. Phillips, and K. Schulten, "Early experiences porting the NAMD and VMD molecular simulation and analysis software to GPU-accelerated OpenPOWER platforms," in *High Performance Computing. ISC High Performance 2016*, Lecture Notes in Computer Science, Vol. 9945, edited by M. Taufer, B. Mohr, and J. Kunkel (Springer, Cham, 2016).

[33] L. Verlet, "Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules," Phys. Rev. **159**, 98–103 (1967).

[34] R. W. Hockney, S. P. Goel, and J. W. Eastwood, "Quiet high-resolution computer models of a plasma," J. Comput. Phys. **14**, 148–158 (1974).

[35] J. Yang, Y. Wang, and Y. Chen, "GPU accelerated molecular dynamics simulation of thermal conductivities," J. Comput. Phys. **221**, 799–804 (2007).

[36] J. A. van Meel, A. Arnold, D. Frenkel, S. F. Portegies Zwart, and R. G. Belleman, "Harvesting graphics power for MD simulations," Mol. Simul. **34**, 259–266 (2008).

[37] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," J. Comput. Phys. **30**, 864–872 (2009).

[38] S. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis, "Exploring SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors," in *2013 IEEE 27th International Symposium on Parallel & Distributed Processing* (IEEE, 2013).

[39] P. Gonnet, "A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations," J. Comput. Chem. **28**, 570–573 (2007).

[40] P. Eastman and V. S. Pande, "Efficient nonbonded interactions for molecular dynamics on a graphics processing unit," J. Comput. Chem. **31**, 1268–1272 (2010).

[41] U. Welling and G. Germano, "Efficiency of linked cell algorithms," Comput. Phys. Commun. **182**, 611–615 (2011).

[42] N. Bailey, T. Ingebrigtsen, J. S. Hansen, A. Veldhorst, L. Bøhling, C. Lemarchand, A. Olsen, A. Bacher, L. Costigliola, U. Pedersen, H. Larsen, J. Dyre, and T. Schrøder, "RUMD: A general purpose molecular dynamics package optimized to utilize GPU hardware down to a few thousand particles," SciPost Phys. **3**, 038 (2017); arXiv:1506.05094.

[43] L. Barba and R. Yokota, "How will the fast multipole method fare in the exascale era?," SIAM News (2013), https://sinews.siam.org/Details-Page/how-will-the-fast-multipole-method-fare-in-the-exascale-era (Last viewed May 12, 2020).

[44] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer, "Strong scaling of general-purpose molecular dynamics simulations on GPUs," Comput. Phys. Commun. **192**, 97–107 (2015); arXiv:1412.3387.

[45] M. J. Abraham and J. E. Gready, J. Comput. Chem. **32**, 2031–2040 (2011).

[46] V. Lindahl, J. Lidmar, and B. Hess, "Accelerated weight histogram method for exploring free energy landscapes," J. Chem. Phys. **141**, 044110 (2014).

[47] I. Ohmura, G. Morimoto, Y. Ohno, A. Hasegawa, and M. Taiji, "MDGRAPE-4: A special-purpose computer system for molecular dynamics simulations," Philos. Trans. R. Soc., A **372**, 20130387 (2014).

[48] J. Grossman, B. Towles, B. Greskamp, and D. E. Shaw, "Filtering, reductions and synchronization in the Anton 2 network," in *2015 IEEE International Parallel and Distributed Processing Symposium* (IEEE, 2015), pp. 860–870.

[49] 1.95 μs/day (<88 μs/step) on an AMD R9-3900X CPU and NVIDIA RTX 2080 SUPER GPU using 2 fs time step and 0.9 nm cutoff.

[50] D. S. Shamshirgar, R. Yokota, A. K. Tornberg, and B. Hess, "Regularizing the fast multipole method for use in molecular simulation," J. Chem. Phys. **151**, 234113 (2019).

[51] C. Yang, T. Geng, T. Wang, R. Patel, Q. Xiong, A. Sanaullah, C. Wu, J. Sheng, C. Lin, V. Sachdeva, W. Sherman, and M. Herbordt, "Fully integrated FPGA molecular dynamics simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, New York, NY, USA, 2019), pp. 1–31.

[52] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. Herr, C. Hughes, T. Mattson, and P. Dubey, "T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (IEEE, 2019), pp. 181–189.

[53] A. Gray and G. Garg, "GROMACS with CUDA-aware MPI direct GPU communication support," Zenodo. V.2021-GPUcomm-JCP, http://doi.org/10.5281/zenodo.3890247

10 July 2023 07:24:24