JCTC Journal of Chemical Theory and Computation

pubs.acs.org/JCTC

Article

Tinker-HP: Accelerating Molecular Dynamics Simulations of Large Complex Systems with Advanced Point Dipole Polarizable Force Fields Using GPUs and Multi-GPU Systems

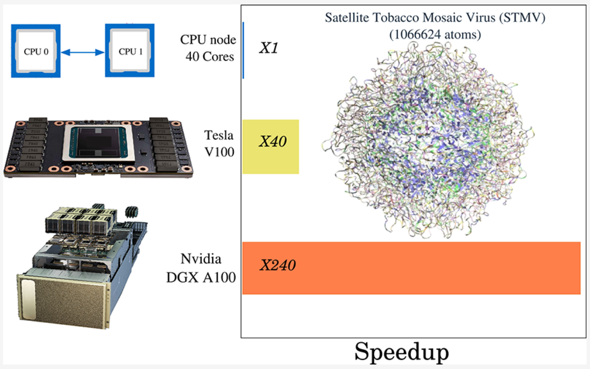
Olivier Adjoua, Louis Lagardère,* Luc-Henri Jolly, Arnaud Durocher, Thibaut Very, Isabelle Dupays, Zhi Wang, Théo Jaffrelot Inizan, Frédéric Célerse, Pengyu Ren, Jay W. Ponder, and Jean-Philip Piquemal*

Cite This: *J. Chem. Theory Comput.* 2021, 17, 2034−2053

Read Online

ACCESS | Metrics & More | Article Recommendations | Supporting Information

ABSTRACT: We present the extension of the Tinker-HP package (Lagardère, et al. *Chem. Sci.* **2018**, *9*, 956−972) to the use of Graphics Processing Unit (GPU) cards to accelerate molecular dynamics simulations using polarizable many-body force fields. The new high-performance module allows for an efficient use of single- and multiple-GPU architectures ranging from research laboratories to modern supercomputer centers. After detailing an analysis of our general scalable strategy that relies on OpenACC and CUDA, we discuss the various capabilities of the package. Among them, the multiprecision possibilities of the code are discussed. If an efficient double precision implementation is provided to preserve the possibility of fast reference computations, we show that a lower precision arithmetic is preferred providing a similar accuracy for molecular dynamics while exhibiting superior performances. As Tinker-HP is mainly dedicated to accelerate simulations using new generation point dipole polarizable force field, we focus our study on the implementation of the AMOEBA model. Testing various NVIDIA platforms including 2080Ti, 3090, V100, and A100 cards, we provide illustrative benchmarks of the code for single- and multicards simulations on large biosystems encompassing up to millions of atoms. The new code strongly reduces time to solution and offers the best performances to date obtained using the AMOEBA polarizable force field. Perspectives toward the strong-scaling performance of our multinode massive parallelization strategy, unsupervised adaptive sampling and large scale applicability of the Tinker-HP code in biophysics are discussed. The present software has been released in phase advance on GitHub in link with the High Performance Computing community COVID-19 research efforts and is free for Academics (see https://github.com/TinkerTools/tinker-hp).



INTRODUCTION

Molecular dynamics (MD) is a very active research field that is continuously progressing.[1,2] Among various evolutions, the definition of force fields themselves grows more complex. Indeed, beyond the popular pairwise additive models[3−7] that remain extensively used, polarizable force field (PFF) approaches are becoming increasingly mainstream and start to be more widely adopted,[8−11] mainly because accounting for polarizability is often crucial for complex applications and adding new physics to the model through the use of many-body potentials can lead to significant accuracy enhancements.[10] Numerous approaches are currently under development but a few methodologies such as the Drude[12−14] or the AMOEBA[15−17] models emerge. These models are more and more employed because of the alleviation of their main bottleneck: their larger computational cost compared to classical pairwise models. Indeed, the availability of High Performance Computing (HPC) implementations of such models within popular packages such as NAMD[18] or GROMACS[19] for Drude or Tinker-HP[20] for AMOEBA fosters the diffusion of these new generation techniques within the research community. This paper is dedicated to the evolution of the Tinker-HP package.[20] The software, which is part of the Tinker distribution,[21] was initially introduced as a double precision massively parallel message passing interface (MPI) addition to Tinker dedicated to the acceleration of the various PFFs and nonpolarizable force fields (n-PFFs) present within

Received: November 6, 2020
Published: March 23, 2021

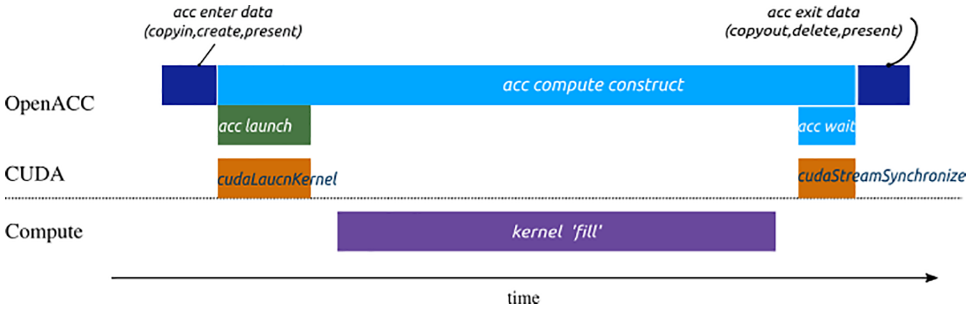


Figure 1. OpenACC synchronous execution model on test kernel `<fill>`.

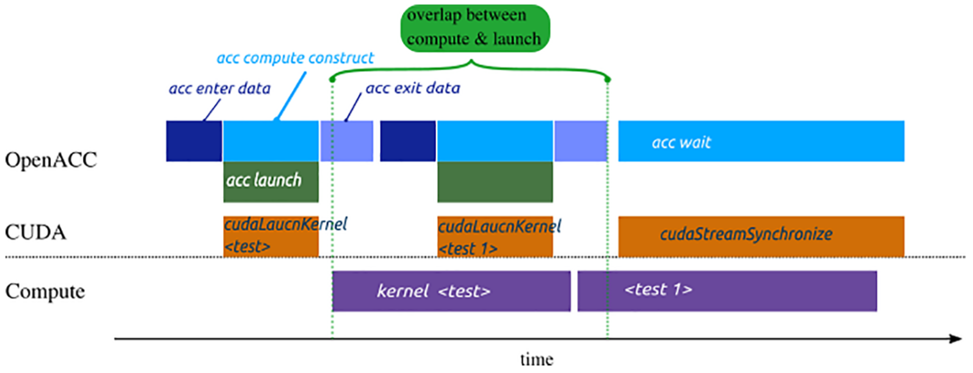


Figure 2. OpenACC asynchronous execution on both kernels `<test>` and `<test 1>`.

the Tinker package. The code was shown to be really efficient, being able to scale on up to tens of thousand cores on modern petascale supercomputers.[20,22] Recently, it has been optimized on various platforms taking advantage of vectorization and of the evolution of the recent CPUs (Central Processing Units).[22] However, in the last 15 years, the field has been increasingly using GPUs (Graphic Processor Units)[23−25] taking advantage of low precision arithmetic. Indeed, such platforms offer important computing capabilities at both low cost and high energy efficiency allowing for reaching routine microsecond simulations on standard GPU cards with pair potentials.[24,26] Regarding the AMOEBA polarizable force field, the OpenMM package[27] was the first to propose an AMOEBA-GPU library that was extensively used within Tinker through the Tinker-OpenMM GPU interface.[28] The present contribution aims to address two goals: (i) the design of an efficient native Tinker-HP GPU implementation; (ii) the HPC optimization in a massively parallel context to address both the use of research laboratories clusters and modern multi-GPU pre-exascale supercomputer systems. The paper is organized as follows. First, we will describe our OpenACC port and its efficiency in double precision. After observing the limitations of this implementation regarding the use of single precision, we introduce a new CUDA approach and detail the various parts of the code it concerns after a careful study of the precision. In both cases, we present benchmarks of the new code on illustrative large biosystems of increasing size on various NVIDIA platforms (including RTX 2080Ti, 3090, Tesla V100 and A100 cards). Then, we explore how to run on even larger systems and optimize memory management by making use of latest tools such as NVSHMEM.[29]

■ OPENACC APPROACH

Global Overview and Definitions. Tinker-HP is a molecular dynamics application with a MPI layer allowing a significant acceleration on CPUs. The core of the application is based on the resolution of the classical newton equations[20,30] given an interaction potential (force field) between atoms. In practice, a molecular dynamic simulation consists of the repetition of the call to an integrator routine defining the changes of the positions and the velocities of all the atoms of the simulated system between two consecutive time steps. The same process is repeated as many times as needed until the simulation duration is reached (see Figure 3). To distribute computations over the processes, a traditional three-dimensional domain decomposition is performed on the simulation box ($\boldsymbol{\Omega}$), which means that it is divided in subdomains ($\boldsymbol{\psi}$), each of which is associated with a MPI process. Then, within each time step, positions of the atoms and forces are exchanged between processes before and after the computation of the forces. Additionally, small communications are required after the update of the positions to deal with the fact that an atom can change the subdomain during a time step. This workflow is described in detail in ref 20.

In recent years a new paradigm has emerged to facilitate computation and programming on GPU devices. In the rest of the text, we will denote as *kernel* the smallest piece of code made of instructions designed for a unique purpose. Thus, a succession of kernels might constitute a *routine* and a *program* can be seen as a collection of routines designed for a specific purpose. There are two types of kernels

- *Serial* kernels, mostly used for variable configuration
- *Loops* kernels, operating on multiple data sets

This programming style, named OpenACC,[31,32] is a directive-based language similar to the multithreading OpenMP paradigm with an additional complexity level. Since a target kernel is destined to be executed on GPUs, it becomes crucial to manage data between both GPU and CPU platforms. At the most elementary level, OpenACC compiler interacts on

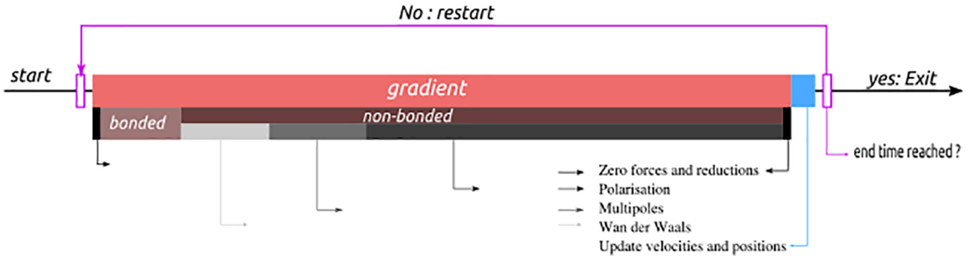


Figure 3. illustration of a MD time step.

a standard host (CPU) kernel and generates a device (GPU) kernel using directives implemented to describe its parallelism along with clauses to manage global data behavior at both entry and exit point and/or kernel launch configuration (Figure 1). This method offers two major benefits. Unlike the low-level CUDA programming language,[33] it takes only a few directives to generate a device kernel. Second, the same kernel is compatible with both platforms, CPUs and GPUs. The portability along with all the associated benefits such as host debug is therefore ensured. However, there are some immediate drawbacks mainly because CPUs and GPUs do not share the same architecture, specifications, and features. Individual CPU cores benefit from a significant optimization for serial tasks, a high clock frequency and integrated vectorization instructions to increase processing speed. GPUs on the other hand were developed and optimized from the beginning for parallel tasks with numerous aggregations of low clock cores holding multiple threads. This means that it may be necessary to reshape kernels to fit device architecture in order to get appropriate acceleration. Once we clearly exhibit a kernel parallelism and associate OpenACC directives to offload it on a device, it should perform almost as well as if it had been directly written in native CUDA. Still, in addition to kernel launch instruction (performed by both OpenACC and CUDA) before the appropriate execution, there is a global data checking operation overhead that might slow down execution (Figure 1). However, it is possible to overlap this operation using asynchronous device streams in the kernel configuration (Figure 2). Under proper conditions and with directly parallel kernels, OpenACC can already lead to an efficient acceleration close to the one reachable with CUDA.

In the following, we will say that a kernel is *semiparallel* if one can find a partition inside the instructions sequence that does not share any dependency at all. A semiparallel kernel is consequently defined *parallel* if all instructions in the partition do not induce a race condition within its throughput.

Once a kernel is device compiled, its execution requires a configuration defining the associated resources provided by the device. With OpenACC, these resources are respectively the total number of threads and the assignment stream. We can access the first one through the *gang* and *vector* clauses attached to a device compute region directive. A `gang` is a collection of vectors inside of which every thread can share cache memory. All gangs run separately on device streaming multiprocessors (SM) to process kernel instructions inside a stream where many other kernels are sequentially queued. OpenACC offers an intermediate parallelism level between gang and vector called `worker`. This level can be seen as a gang subdivision.

It is commonly known that GPUs are inefficient for sequential execution due to their latency. To cover up latency, each SM comes with a huge register file and cache memory in order to hold and run as many vectors as possible at the same time. Instructions from different gangs are therefore pipe-lined and injected in the compute unit.[33,34] From this emerges the kernel occupancy's concept, which is defined as the ratio between the gang's number concurrently running on one SM and the maximum gang number that can actually be held by this SM.

Global Scheme. The *parallel* computing power of GPUs is in constant evolution and the number of streaming multiprocessors (SM) is almost doubling with every generation. Considering their impressive compute potential in comparison to CPUs, one can assume that the only way to entirely benefit from this power is to offload the entire application on device. Any substantial part of the workflow of Tinker-HP should not be performed on the CPU platform. It will otherwise represent a bottleneck to performance in addition to requiring several data transfers. As for all MD applications, most of the computation lies in the evaluation of the forces. For the AMOEBA polarizable model, it takes around 97% of a time step to evaluate those forces when running sequentially on CPU platform. Of these 97%, around 10% concern bonded forces and 90% the nonbonded ones, namely, polarization, (multipolar) permanent electrostatics, and van der Waals. The polarization, which includes the iterative resolution of induced dipoles, largely dominates this part (see Figure 3). Nonbonded forces and polarization in particular will thus be our main focus regarding the porting and optimization. We will then benefit from the already present Tinker-HP MPI layer[20,22] to operate on several GPUs. The communications can then be made directly between GPUs by using a CUDA aware MPI implementation.[35] The Smooth Particle Mesh Ewald method[20,36,37] is at the heart of both the permanent electrostatics and polarization nonbonded forces used in Tinker-HP, first through the iterative solution of the induced dipoles and then through the final force evaluation. It consists in separating the electrostatic energy in two independent pieces: real space and reciprocal space contributions. Let us describe our OpenACC strategy regarding those two terms.

Real Space Scheme. Because the real space part of the total PME energy and forces has the same structure as the van der Waals one, the associated OpenACC strategy is the same. Evaluating real space energy and forces is made through the computation of pairwise interactions. Considering n atoms, a total of $n(n - 1)$ pairwise interactions need to be computed. This number is reduced by half because of the symmetry of the interactions. Besides, because we use a cutoff distance after which we neglect these interactions, we can reduce their number to being proportional to n in homogeneous systems by using neighbor lists. The up-bound constant is naturally

Chart 1. OpenACC Real Space Offload Scheme[a]

```
c$acc parallel loop gang default(present) async
c$acc&         private(scaling_data)
do i = 1,numLocalAtoms
   iglob = glob(i)  ! Get Atom i global id
   !Get Atom iglob parameter and positions
   ...
   !Gather Atoms iglob scaling interactions in 'scaling_data'
   ...
c$acc loop vector
   do k = 1, numNeig(i)
      kglob = glob( list(k,i) )
      ! Get Atom kglob parameter and positions
      ! Compute distance (d) between iglob & kglob
      if (d < dcut) then
         call resolve_scaling_factor(scaling_data)
         ...
         call Compute_interaction !inlined
         ...
         call Update_(energy,forces,virial)
      end if
   end do
end do
```

[a] The kernel is offloaded onto a device using two of the three parallelism levels offered by OpenACC. The first loop is broken down over gangs and gathers all data related to atom `iglob` using gang's shared memory through the private clause. OpenACC vectors are responsible of the evaluation and the addition of forces and energy after resolving scaling factor if necessary. Regarding data management we make sure with the present clause that everything is available on device before the execution of the kernel.

reduced to a maximum neighbors for every atoms noted as $\mathrm{Neig}_{\mathrm{max}}$.

The number of interactions is up-bounded by n $^*\mathrm{Neig}_{\mathrm{max}}$. In terms of implementation, we have written the compute algorithm into a single loop kernel. As all the interactions are independent, the kernel is semiparallel regarding each iteration. By making sure that energy and forces are added one at a time, the kernel becomes parallel. To do that, we can use atomic operations on GPUs, which allow us to make this operation in parallel and solve any race condition issue without substantially impacting parallel performance. By doing so, real space kernels looks like Chart 1.

At first, our approach was designed to directly offload the CPU vectorized real space compute kernels that use small arrays to compute pairwise interactions in hopes of aligning the memory access pattern at the vector level and therefore accelerate the code.[22] This requires each gang to privatize every temporary array and results in a significant overhead with memory reservation associated with a superior bound on the gang's number. Making interactions computation scalar helps us remove those constraints and double the kernel performance. The explanation behind this increase arises from the use of GPU scalar registers. Still, one has to resolve the scaling factors of every interactions. As it happens inside gang shared memory, the performance is slightly affected. However, we would benefit from a complete removal of this inside search. There are two potential drawbacks to this approach:

- Scaling interactions between neighboring atoms of the same molecule can become very complex. This is particularly true with large proteins. Storage workspace can potentially affect shared memory and also kernel's occupancy.
- Depending on the interactions, there is more than one kind of scaling factor. For example, every AMOEBA polarization interaction needs three different scaling factors.

The best approach is then to compute scaling interactions separately in a second kernel. Because they only involve connected atoms, their number is small compared to the total number of nonbonded interactions. We first compute unscaled nonbonded interactions and then apply scaling correction in a second part. An additional issue is to make this approach compatible with the 3d domain decomposition. Our previous kernel then reads as in Chart 2.

Reciprocal Space Scheme. The calculation of Reciprocal space PME interactions essentially consists in five steps:

1. interpolating the (multipolar) density of charge at stake on a 3D grid with flexible b-spline order (still, the

Chart 2. Final OpenACC Real Space Offload Scheme[a]

```
c$acc parallel loop gang default(present) async
do i = 1,numLocalAtoms
   iglob = glob(i)  ! Get Atom i global id
   !Get Atom iglob parameter and positions
   ...
c$acc loop vector
   do k = 1, numNeig(i)
      kglob = glob( list(k,i) )
      ! Get Atom kglob parameter and positions
      ! Compute distance (d) between iglob & kglob
      if (d < dcut) then
         call Compute_interaction !inlined
         ...
         call Update_(energy,forces,virial)
      end if
   end do
end do

call correct_scaling
```

[a] This kernel is more balanced and exposes a much more computational load over vectors. A "correct_scaling" routine applies the correction of the scaling factors. This procedure appears to be much more suitable to device execution.

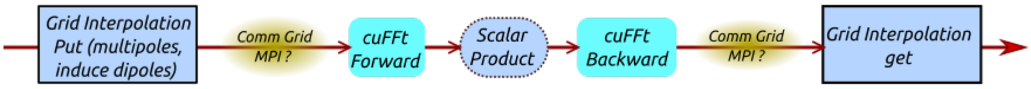


Figure 4. Reciprocal space offload scheme. Charge interpolation and Force interpolation are both written in a single kernel. They are naturally parallel except for the atomic contributions to the grid in the first one. The approach remains the same for data management between host and device as for real space: all data are by default device resident to prevent any sort of useless transfer. Regarding MPI communications, exchanges take place directly between GPUs through interconnection.

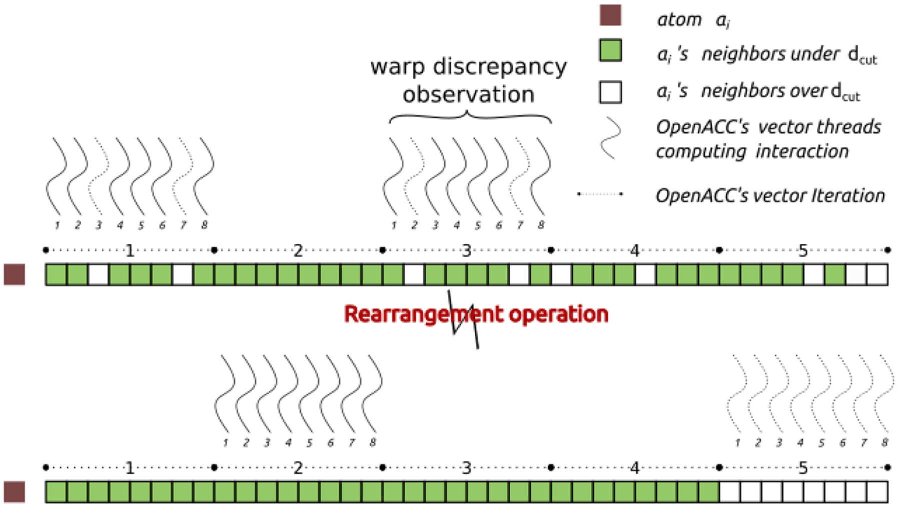

atom a_i
a_i 's neighbors under d_{cut}
a_i 's neighbors over d_{cut}
OpenACC's vector threads computing interaction
OpenACC's vector Iteration
warp discrepancy observation
1 2 3 4 5 6 7 8
1
2
3
4
5
Rearrangement operation

Figure 5. Illustration of compute balance due to the list reordering. Unbalanced computation in the first image induces an issue called warp discrepancy: a situation where all threads belonging to the same vector do not follow the same instructions. Minimizing that can increase kernel performance significantly since we ensure load balancing among each thread inside the vector.

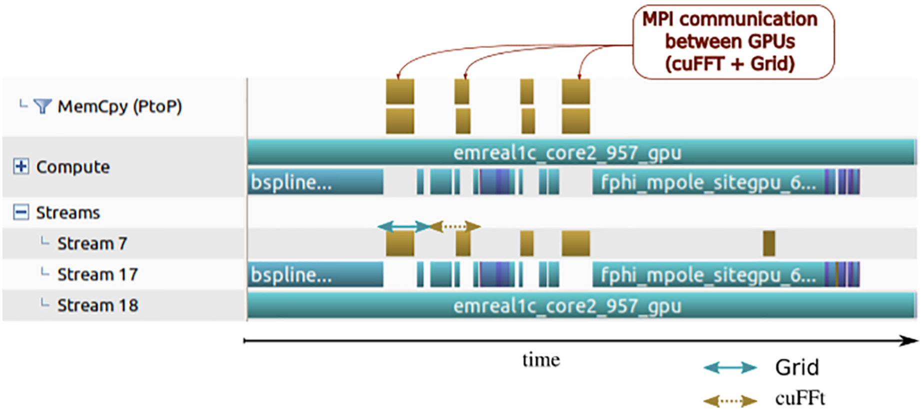


Figure 6. Representation of cuFFt's communication/computation overlap using different streams for direct and reciprocal space. Real space computation kernels are assigned to asynchronous stream 18. Reciprocal ones go into high priority asynchronous stream 17. The real space kernel therefore recovers FFt grid exchanges. This profile was retrieved on 2 GPUs.

implementation is optimized to use an order of 5 as it is the default and the one typically used with AMOEBA).

2. switching to Fourier space by using a forward fast Fourier transform (FFt)
3. performing a trivial scalar product in reciprocal space
4. performing a backward FFt to switch back to real space
5. performing a final multiplication by b-splines to interpolate the reciprocal forces

Regarding parallelism, Tinker-HP uses a two-dimensional decomposition of the associated 3d grid based on successive 1D FFts. Here, we use the cuFFt library.[38] The OpenACC offload scheme for reciprocal space is described in Figure 4.

We just reviewed our offload strategy of the nonbonded forces kernels with OpenACC, but the bonded ones remain to be treated. Also, the MPI layer has to be dealt with. The way bonded forces are computed is very similar to the real space ones, albeit simpler, which makes their offloading relatively straightforward. MPI layer kernels require, on the other hand, a slight rewriting as communications are made after a packing pretreatment. In parallel, one does not control the throughput order of this packing operation. This is why it becomes necessary to also communicate the atom list of each process to their neighbors. Now that we presented the main offload strategies, we can focus on some global optimizations regarding the implementation and execution for single and multiple GPUs. Some of them lead to very different results depending on the device architecture.

Optimizations Opportunities.

>A first optimization is to impose an optimal bound on the vector size when computing pair interactions. In a typical setup, for symmetry reasons, the number of neighbors for real space interactions varies between zero and a few hundred. Because of that second loop in Chart 2, the smallest vector length (32) is appropriate to balance computation among the threads it contains. Another optimization concerns the construction of the neighbor lists. Let us recall that it consists of storing, for every atom, the neighbors that are closer than a cut distance (d_{cut}) plus a buffer d_{buff}. This buffer is related to the frequency at which the list has to be updated. To balance computation at the vector level and at the same time reduce warp discrepancy (as illustrated in Figure 5), we have implemented a reordering kernel: we reorder the neighbor list for each atom so that the firsts are the ones under d_{cut} distance.

>A second optimization concerns the iterative resolution of the induced dipoles. Among the algorithms presented in refs 37 and 39, the first method we offloaded is the preconditioned conjugated gradient (PCG). It involves a (polarization) matrix-vector product at each iteration. Here, the idea is to reduce the computation and favor coalesce memory access by precomputing and storing the elements of (the real space part) of the matrix before the iterations. As the matrix-vector product is being repeated, we see a performance gain starting from the second iteration. This improves performance but implies a memory cost that could be an issue on large systems or on GPUs with small memory capabilities. This overhead will be reduced at a high level of multidevice parallelism.

>An additional improvement concerns the two-dimensional domain decomposition of the reciprocal space 3D grid involved with FFt. The parallel scheme for FFt used in Tinker-HP is the following for a forward transform:

$$\text{FFT}(s)\ 1\text{d}\ \dim(x) + x\ \text{Transpose}\ y + \text{FFt}(s)\ 1\text{d}\ \dim(y) + y\ \text{Transpose}\ z + \text{FFT}(s)\ 1\text{d}\ \dim(z)$$

Each transposition represents an all-to-all MPI communication, which is the major bottleneck preventing most MD applications using PME to scale across nodes.[20,22,40] Given the GPUs' huge computing power, this communication is even more problematic in that context. On the device, we use the cuFFt[38] library. Using many cuFFt 1d batches is not as efficient as using fewer batches in a higher dimension. Indeed, devices are known to underperform with low saturation kernels. In order to reduce MPI exchanges and increase load on device, we adopted a simple 3d dimensional cuFFt batch when running on a single device. On multiple GPUs, we use the following scheme based on a 1d domain decomposition along the z axis:

$$\text{cuFFt}(s)\ 2\text{d}\ \dim(x, y) + y\ \text{Transpose}\ z + \text{cuFFt}(s)\ 1\text{d}\ \dim(z)$$

which gives a 25% improvement compared to the initial approach.

>Profiling the application on a single device, we observed that real space computation is on average 5 times slower than reciprocal space computation. This trend reverses using multiple GPUs because of the communications mentioned above. This motivated the assignment of these two parts in two different priority streams. Reciprocal space kernels along with MPI communications are queued inside the higher priority

stream, and real space kernels, devoid of communications, can operate at the same time on the lower priority stream to recover communications. This is illustrated in Figure 6.

Simulation Validation and Benchmarks. Here, we use the same bench systems as in refs 20 and 22: the solvated DHFR protein, the solvated COX protein, and the STMV virus, all with the AMOEBA force field, respectively made of 23558, 171219, and 1066600 atoms. The molecular dynamics simulations were run in the *NVT* ensemble at 300 K for 5 ps simulation using a (bonded/nonbonded) RESPA integrator with a 2 fs outer time step (and a 1 fs inner time step)[41] and the Bussi thermostat.[42] The performance was averaged over the complete runs. For validation purposes, we compared the potential energy, temperature, and pressure of the systems during the first 50 time steps with values obtained with Tinker-HP v1.2. Furthermore, these results were compared to Tinker-OpenMM in the same exact setup.[28]

We can directly observe the technical superiority of the Quadro architecture compared to the Geforce one. Double precision (DP) compute units of the V100 allow us to vastly outperform the Geforce. In addition, by comparing the performance of the Geforce RTX to the one of the Quadro GV100, we see that Quadro devices are much less sensitive to warp discrepancy and noncoalesced data accessing pattern. It is almost as if the architecture of the V100 card overcomes traditional optimizations techniques related to parallel device implementation. However, we see that our pure OpenACC implementation manages to deliver more performance than usual device MD application with PFF in DP. The V100 results were obtained on the Jean-Zay HPE SGI 8600 cluster of the IDRIS supercomputer Center (GENCI-CNRS, Orsay, France) whose converged partitions are respectively made of 261 and 351 nodes. Each one is made of 2 Intel Cascade Lake 6248 processors (20 cores at 2.5 GHz) accelerated with 4 NVIDIA Tesla V100 SXM2 GPUs, interconnected through NVIDIA NVLink, and coming respectively with 32 GB of memory on the first partition and 16 GB on the second. Here as in all the tests presented in this paper, all the MPI communications were made with a CUDA aware MPI implementation.[35] This result is very satisfactory as a single V100 card is at least 10 times faster than an entire node of this supercomputer using only CPUs.

Multidevice benchmark results compared with host-platform execution are presented in Figure 7. In practice, the DHFR protein is too small to scale out. MPI communications overcome the computations even with optimizations. On the other hand, COX and STMV systems show good multi-GPU performances. Adding our latest MPI optimizations (FFt reshaping and asynchronous computing between the direct and reciprocal part of PME) allows for a substantial gain in performances. We see that on a Jean-Zay node we can only benefit from the maximum communication bandwidth when running on the entire node; hence the relative inflection point on the STMV performances on the 2 GPU setup. Indeed, all devices are interconnected inside one node in such a way that they all share the interconnection bandwidth. More precisely, running on 2 GPUs reduces the bandwidth by three and therefore affects the scalability. It is almost certain that results would get better on an interconnected node made exclusively of 2 GPUs. Those results are more than encouraging considering the fact that we manage to achieve them with a full OpenACC implementation of Tinker-HP (direct portage of the reference CPU code) in addition to some adjustments.

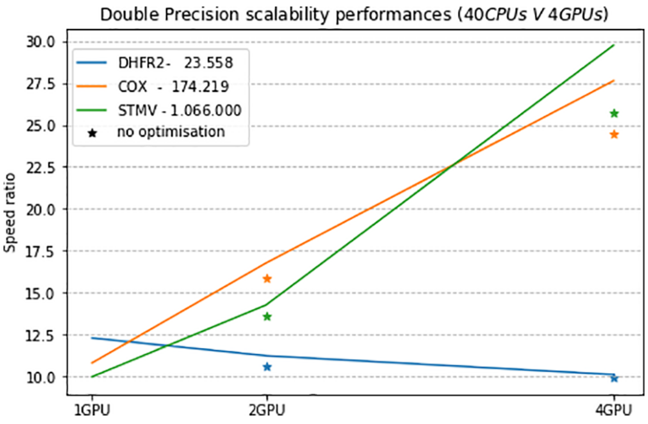


Figure 7. Performance ratio between single node GPU and single node CPU performance. Reference values can be found in Table 1.

Table 1. Single Device Benchmark: MD Production per Day (ns/day)[a]

systems/ devices (ns/day)	CPUs - one node	RTX 2080Ti	RTX 2080Ti + optim	V100	V100 + optim	Tinker-OpenMM V100
DHFR	0.754	2.364	3.903	8.900	9.260	6.300
COX	0.103	0.341	0.563	1.051	1.120	0.957
STMV	0.013	n/a	n/a	0.111	0.126	0.130

[a] All simulations were run using a RESPA/2 fs setup.

In summary, our DP implementation is already satisfactory compared to other applications such as Tinker-OpenMM. Our next section concerns the porting of Tinker-HP in a downgraded precision.

CUDA APPROACH

Even though we already have a robust OpenACC implementation of Tinker-HP in double precision, the gain in terms of computational speed when switching directly to single precision (SP) is modest, as shown in Table 2, which is inconsistent with the GPUs' computational capabilities.

Table 2. Single Precision MD Production (ns/day) within the OpenACC Implementation

	DHFR	COX	STMV
V100	11.69	1.72	0.15
RTX-2080 Ti	11.72	1.51	n/a

This is more obvious for Geforce architecture devices since those cards do not possess DP physical compute units and therefore emulate DP Instructions. According to Table 3, theoretical ratios of 2 and 31 are respectively expected from V100 and RTX-2080 Ti performances when switching from DP to SP, which makes an efficient SP implementation mandatory.

In practice, instead of doubling the speed on V100 cards, we ended up noticing a 1.25 increase factor on V100 and 3 on RTX on DHFR in SP compared to DP with the same setup. All tests have been done under the assumption that our simulations are valid in this precision mode. More results are shown in Table 2. Furthermore, a deep profile conducted on the kernels representing Tinker-HP's bottleneck (real space

Table 3. Device Hardware Specifications

	performances (Tflop/s)			compute/access	
GPU	DP	SP	memory bandwidth (GB/s)	DP (Ops/8B)	SP (Ops/4B)
Quadro GV100	7.40	14.80	870.0	68.04	68.04
Tesla V100 SXM2	7.80	15.70	900.0	69.33	69.77
Geforce RTX-2080 Ti	0.42	13.45	616.0	5.45	87.33
Geforce RTX-3090	0.556	35.58	936.2	4.75	152.01

nonbonded interactions) in the current state reveals an insufficient exploitation of the GPU SP compute power. Figure 8 suggests that there is still room for improvements in order to take full advantage of the card's computing power and memory bandwidth both in SP and DP. In order to exploit device SP computational power and get rid of the bottleneck exposed by Figure 8, it becomes necessary to reshape our implementation method and consider some technical aspects beyond OpenACC's scope.

Global Overview and Definitions. As mentioned in the previous section, GPUs are most efficient with parallel computations and coalesce memory access patterns. The execution model combines and handles effectively two nested levels of parallelism. The high level concerns multithreading and the low level the SIMD execution model for vectorization.[22,43] This model stands for single instruction multiple threads (SIMT).[44] When it comes to GPU programming, SIMT also includes control-flow instructions along with subroutine calls within the SIMD level. This provides additional freedom of approach during implementation. To improve the results presented in the last paragraph (Table 2) and increase peak performance on computation and throughput, it is crucial to expose more computations in real space kernels and to minimize global memory accesses in order to benefit from cache and shared memory accesses as well as registers. Considering OpenACC paradigm limitations in terms of kernel description as well as the required low-level features, we decided to rewrite those specific kernels using the standard approach of low-level device programming in addition to CUDA built-in intrinsics. In a following section, we will describe our corresponding strategy after a thorough review on precision.

Precision study and Validation.

Definition i. We shall call ϵ_p the machine precision (in SP or DP), the smallest floating point value such that $1 + \epsilon_p > 1$. They are respectively 1.2×10^{-7} and 2.2×10^{-16} in SP and DP.

Definition ii. Considering a positive floating point variable a, the machine precision ϵ_a attached to a is

$$1 + \epsilon > 1 \Leftrightarrow a + \epsilon_p \cdot a > a \Leftrightarrow \epsilon_a = \epsilon_p \cdot a$$

Therefore an error made for a floating point operation between a and b can be expressed as

$$a \tilde{\oplus} b = (a \oplus b)(1 + \epsilon_p) \tag{1}$$

where $\tilde{\oplus}$ designates the numerical operation between a and b.

Property i. Numerical error resulting from sequential reduction operations are linear while those resulting from parallel reduction are logarithmic. Thus, parallel reductions are entirely suitable to GPU implementation as they benefit from both parallelism and accuracy.

Before looking further into the matter of downgrading precision, we have to make sure that Tinker-HP is able to work in this mode. Although it has been proven in the literature[25,27,45] that many MD applications are able to provide correct results with simple precision, extensive precision studies with polarizable force fields are lacking.

When it comes to standard IEEE floating point arithmetic, regular 32 bit storage offers no more than 7 significant digits due to the mantissa. In comparison, we benefit from 16 significant digits with DP 64 storage bits. Without any consideration on the floating number's sign, it is safe to assume that any application working with absolute values outside the $[10^{-7}, 10^{7}]$ scope will fail to deliver sufficient accuracy when operating in complete SP mode. This is called the floating point overflow. To overcome this, the common solution is to use a mixed precision mode (MP) that encompasses both standard SP and a superior precision container to store variables subject to SP overflowing. In practice, most MD applications adopt SP for computation and a higher precision for accumulation. Moreover, applications like Amber or OpenMM propose another accumulation method which rely on a different type of variable.[45]

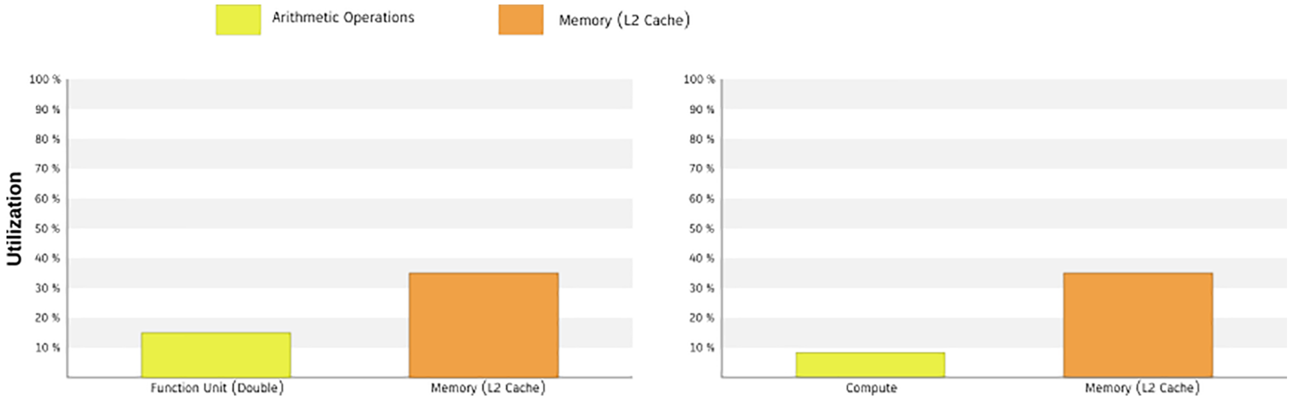


Figure 8. Profile of the matrix-vector compute kernel on the DHFR system. The left picure is obtained with the double precision and the right one with simple precision. In both modes, results indicate an obvious latency issue coming from memory accessing pattern that prevents the device from reaching its peak performance.

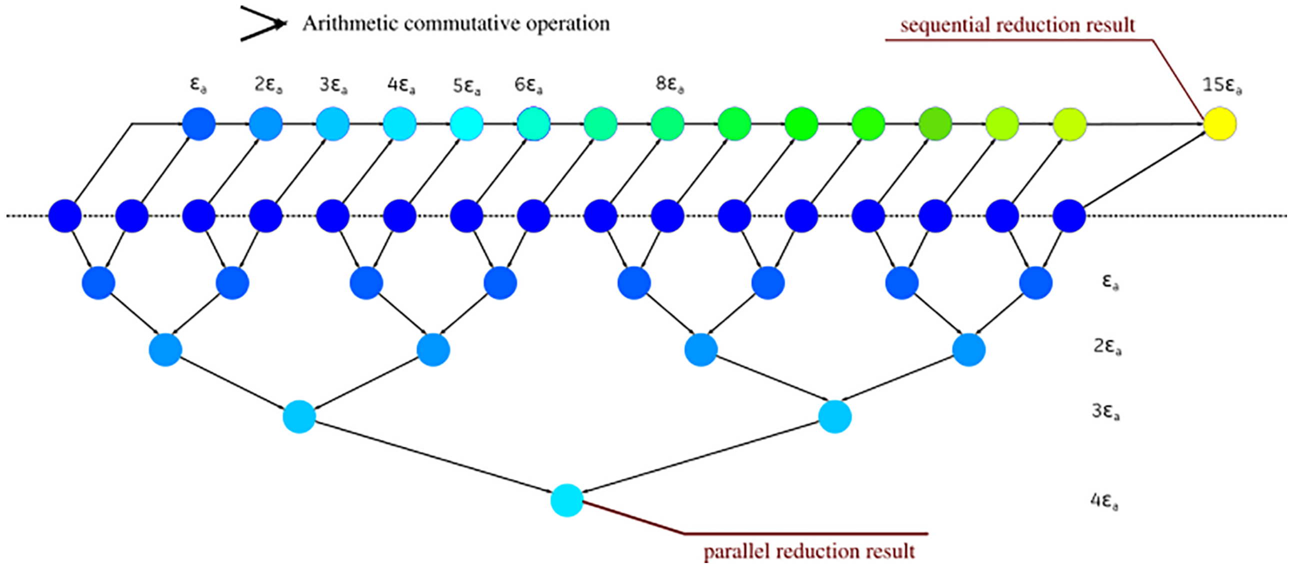

Arithmetic commutative operation
sequential reduction result
ε$_a$
2ε$_a$
3ε$_a$
4ε$_a$
5ε$_a$
6ε$_a$
8ε$_a$
15ε$_a$
ε$_a$
2ε$_a$
3ε$_a$
4ε$_a$
parallel reduction result

Figure 9. Illustration of the reduction operation on a 16 variables set. Each arithmetic operation generates an error ϵ_a that is accumulated during the sequential operation. On the other hand, parallel reduction uses intermediate variables to significantly reduce the error.

The description made in the previous section shows that energy and the virial evaluation are linear-dependent with the system's size. Depending on the complexity of the interaction in addition to the number of operations it requires, we can associate a constant error value ϵ_i to it. Thus, we can bound the error made on the computation of a potential energy with $N_{\text{int}}\epsilon_i < n\text{Neig}_{\max}\epsilon_i$, where N_{int} represents the number of interactions contributing to this energy and $\text{Neig}_{\max}$ the maximum number of neighbor it involves per atom. As it is linear with respect to the system size we have to evaluate this entity with a DP storage container. Furthermore, to reduce even more the accumulation of error due to large summation, we shall employ buffered parallel reduction instead of a sequential one (Figure 9). On the other hand, we have to deal with the forces that remain the principal quantities that drive a MD simulation. The error made for each atom on the nonbonded forces is bound by $\text{Neig}_{\max}\cdot\epsilon_i$ depending on the cutoff. However, each potential comes with a different ϵ_i. In practice, the corresponding highest values are the ones of both van der Waals and bonded potentials. The large number of pairwise interactions induced by the larger van der Waals cutoff in addition to the functional form that includes a power of 14 (for AMOEBA) causes SP overflowing for distances greater than 3 Å. By reshaping the variable encompassing the pairwise distance, we get a result much closer to DP since intermediate calculations do not overflow. Regarding the bonded potentials, ϵ_i^{bond} depends more on the conformation of the system.

Parameters involved in bond pairwise evaluation (spring stiffness, ...) cause a SP numerical error (ϵ_i^{bond}) standing between 1×10^{-3} and 1×10^{-2}, which frequently reach 1×10^{-1} (following (eq 1)) during the summation process, and this affects forces more than total energy. In order to minimize ϵ_i^{bond}, we evaluate the distances in DP before casting the result to SP. In the end, ϵ_i^{bond} is reduced on the scope of $[1 \times 10^{-4}, 1 \times 10^{-3}]$, which represents the smallest error we can expect from SP.

Furthermore, unlike the energy, a sequential reduction using atomic operations is applied to the forces. The resulting numerical error is therefore linear with the total number of summation operations. This is why we adopt a 64 bit container for those variables despite the fact they can be held in a 32 bits container.

Regarding the type of the 64 bit container, we analyze two different choices. First, we have the immediate choice of a floating point. The classical mixed precision uses FP64 for accumulation and integration. Every MD applications running on GPU integrates this mode. It presents the advantage of being straightforward to implement. Second, we can use an integer container for accumulation: this is the concept of fixed point arithmetic introduced by Yates.[46] To be able to hold a floating point inside an integer requires us to define a certain number of bits to hold the decimal part. It is called the fixed point fractional bits. The left bits are dedicated to the integer part. Unlike the floating point, freezing the decimal position constrains the approximation precision but offers a correct accuracy in addition to deterministic operations. Considering a floating point value x and an integer one a and a fractional bits value (**fB**), the relations establishing the transition back and forth between them, as $a = f(x)$ and $x = f^{-1}(a)$, are defined as follows:

$$a = \texttt{int}\left(\texttt{round}\left(x \times 2^{\mathbf{fB}}\right)\right) \tag{2}$$

$$x = \frac{\texttt{real}(a)}{2^{\mathbf{fB}}} \tag{3}$$

with `int` and `real` the converting functions and `round` the truncation function that extracts the integer part. When it comes to MD, fixed point arithmetic is an excellent tool: each SP pairwise contribution is small enough to be efficiently captured by a 64 bit fixed point. For instance, it takes only 27 bits to capture 8 digits after the decimal point with a large place left for the integer part. For typical values observed with different system sizes, we are far from the limit imposed by the integer part of the container. Inspired by the work of Walker, Götz, et al.,[45] we have implemented this feature inside Tinker-HP with the following configuration: 34 fractional bits has been selected for forces accumulation, which leaves 30 bits for the integer part, thus setting the absolute limit value to 2^{29} (kcal/mol)·Å. For the energy, we only allocated 30 fractional bits given the fact that it grows linearly with the system size. Besides, using an integer container for accumulation avoids dealing with DP instructions, which significantly affects performance on Geforce cards unlike Tesla ones. In summary, we should expect at least a performance or precision improvement from FP.

A practical verification is shown in Figure 10. In all cases, both MP and FP behave similarly. Forces being the driving components of MD, the trajectories generated by our mixed precision implementation are accurate. However, as one can see, if errors remain very low for forces even for large systems, a larger error exists for energies, a phenomenon observed in all previous MD GPU implementations. Some specific post-treatment computations, like in a BAR free energy computation or *NPT* simulations with a Monte Carlo barostat, require accurate energies. In such a situation, one could use the DP capabilities of the code for this postprocessing step as Tinker-HP remains exceptionally efficient in DP even for large systems. A further validation simulation in the *NVE* ensemble can be found in Figure 15, confirming the overall excellent stability of the code.

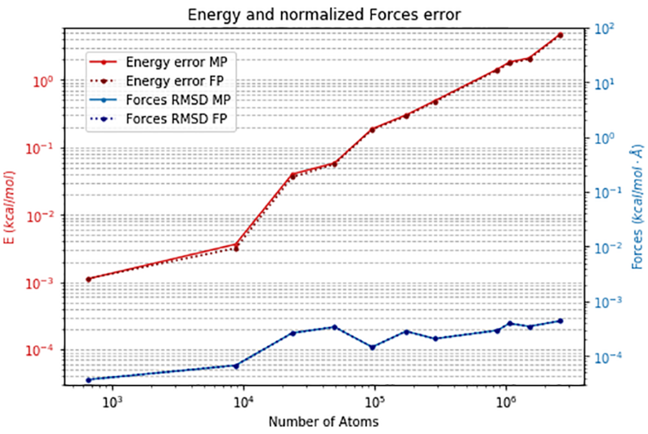


Figure 10. Absolute error between DP implementation and both FP and MP implementations on total potential energy and Forces. Forces root-mean-square deviation between DP and MP for systems from 648 up to 2592000 atoms. As expected, both absolute errors in SP and FP are almost identical on the energy and they grow linearly with the system size. Logarithmic regression gives 0.99 value for the curve slope and up to 5 kcal/mol for the largest system. However, the relative error for all systems is located under 7×10^{-7} in comparison to DP. One can also see that the error on the forces is independent of the system size.

Neighbor List. We want to expose the maximum of computation inside a kernel using the device shared memory. To do so, we consider the approach where a specific group of atoms interacts with another one in a block-matrix pattern (see Figure 11). We need to load the parameters of the group of atoms and the output structures needed for computation directly inside cache memory and/or registers. On top of that, CUDA built-in intrinsics can be used to read data from neighbor threads and if possible compute cross term interactions. Ideally, we can expose $B_{\text{comp}} = B_{\text{size}}{}^{2}$ computations without a single access to global memory, with B_{size} representing the number of atoms within the group. With this approach, the kernel should reach its peak in terms of computational load.

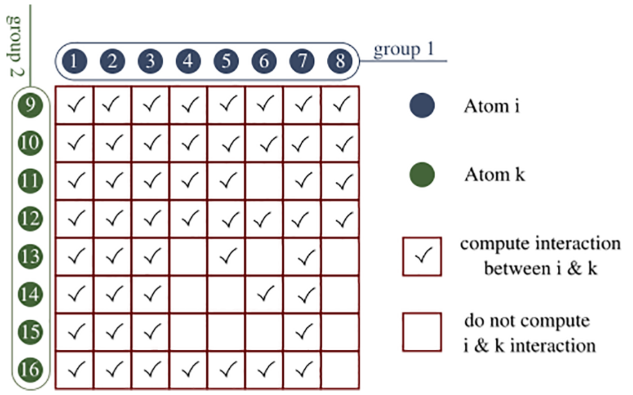

group 1
group 2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Atom i
Atom k
compute interaction between i & k
do not compute i & k interaction

Figure 11. Representation of interactions between two groups of atoms within Tinker-HP. $B_{\text{size}} = 8$ for the illustration.

A new approach of the neighbor list algorithm is necessary to follow the logic presented above. This method will be close to standard blocking techniques used in many MD applications.[25,27] Let us present the structure of the algorithm in a sequential and parallel, MPI, context.

Box Partitioning. Lets us recall that given a simulation box $\boldsymbol{\Omega}$, a set of $\boldsymbol{\omega}_c$ with $c \in [0, ..., N_c]$ forms a $\boldsymbol{\Omega}$ partition if and only if

$$\begin{cases} \boldsymbol{\omega}_1 \cup ... \cup \boldsymbol{\omega}_{N_c} = \boldsymbol{\Omega} \\ \boldsymbol{\omega}_1 \cap ... \cap \boldsymbol{\omega}_{N_c} = \varnothing \end{cases}$$

We consider in the following that each group deals with interactions involving atoms within a region of space. In order to maximize B_{comp} between every pair of groups, we must then ensure their spatial compactness. Moreover, all these regions need to define a partition of $\boldsymbol{\Omega}$ to make sure we do not end up with duplicate interactions. Following this reasoning, we might be tempted to group them into small spheres but it is impossible to partition a polygon with only spheres, not to mention the difficulties arising from the implementation point of view.

The MPI layer of Tinker-HP induces a first partition of $\boldsymbol{\Omega}$ in P subdomains $\boldsymbol{\psi}_p$, $p \in [0, ..., P]$, where P is the number of MPI processes. Tinker-HP uses the midpoint image convention[47] so that the interactions computed by the process assigned to $\boldsymbol{\psi}_p$ are the ones whose midpoint falls into $\boldsymbol{\psi}_p$. The approach used in Tinker-HP for the nonbonded neighbor list uses a cubic partition $\boldsymbol{\omega}_c$, $c \in [1, ..., N_c]$, of $\boldsymbol{\psi}_p$ and then collects the neighboring atoms among the neighboring cells of $\boldsymbol{\omega}_c$. Here, we proceed exactly in the same way with two additional conditions to the partitioning. First, the number of atoms inside each cell $\boldsymbol{\omega}_c$ must be less or equal than B_{size}. Second, we must preserve a common global numbering of the cells across all domains $\boldsymbol{\psi}_p$ to benefit from a unique partitioning of $\boldsymbol{\Omega}$.

Once the first partitioning in cells is done, an additional sorting operation is initiated to define groups so that each of them contains exactly B_{size} spatially aligned atoms following the cell numbering (note that because of the first constrain mention earlier, one cell can contain atoms belonging to a maximum of two groups). More precisely, the numbering of the cells follows a one-dimensional representation of the three dimension of the simulation box. Now, we want to find the best partitioning of $\boldsymbol{\psi}_p$ in groups that will ensure enough proximity between atoms inside a group, minimizing the

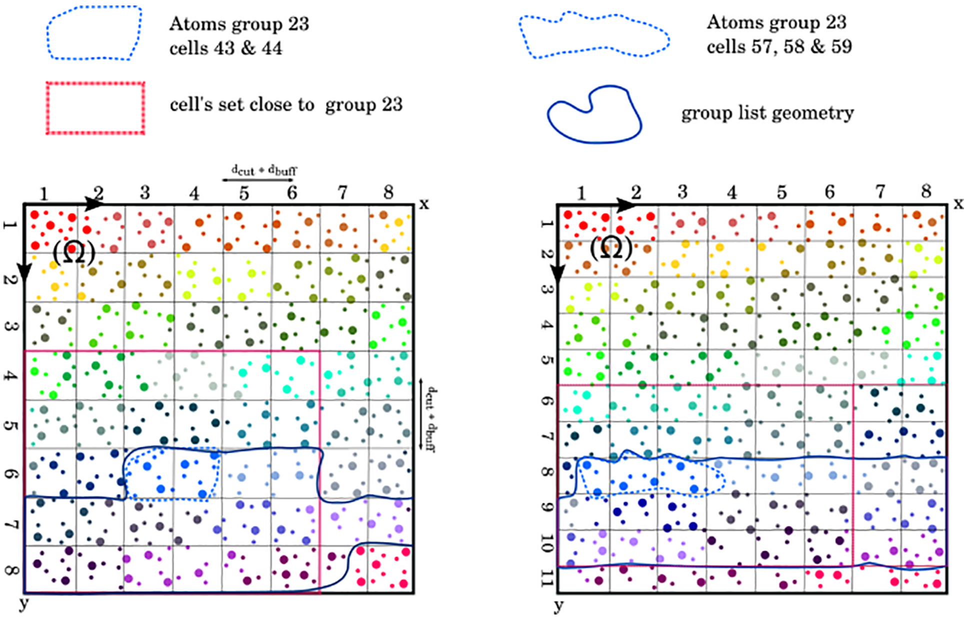


Figure 12. Illustration of a two-dimensional partition along with groups for a box of water. The left figure shows a 64-cell partition of **Ω** while the right one refines this partitioning into 88 cells. The groups are defined by reindexing the atoms following the cell numbering and their maximum size. Here $B_{\text{size}} = 16$. A unique color is associated with every atom belonging to the same group. No cell contains more than B_{size} atoms or 2 groups. Once a group is selected (23), searching for its neighboring groups is made through the set of cells (with respect to the periodic boundary conditions) near the cells it contains (43 and 44). Once this set is acquired, all the group indexes greater than or equal to the selected group constitute the actual list of neighbors to take the symmetry of the interactions into account. We see that the group's shape modulates the group neighborhood as illustrated with the right illustration and a spatially flat group 23.

number of neighboring groups and consequently maximizing B_{comp}.

When the partitioning generates too flat domains, each group might end up having too many neighboring groups. The optimal cell shape (close to a sphere) is the cube but we must not forget the first constraint and end up with a very thin partition either. However, atom groups are not affected by a partition along the innermost contiguous dimension in the cell numbering. We can exploit this to get better partitioning. Figure 12 illustrates and explains the scheme on a two-dimensional box. Partitioning is done in an iterative manner by cycling on every dimension. We progressively increase the number of cells along each dimension starting on the contiguous one until the first condition is fulfilled. During a parallel run, we keep track of the cell with the smallest number of atoms with a reduction operation. This allows to have a global partitioning of $\boldsymbol{\Omega}$ and not just $\boldsymbol{\psi}_p$.

Now that we do dispose of a spatial rearrangement of the atoms into groups, we need to construct pair-lists of all interacting groups according to the cutoff distance plus an additional buffer to avoid reconstructing it at each time step.

Groups are built in such a way that it is straightforward to jump from groups indexing to cells indexing. We chose to use an adjacency matrix which is GPU suitable and compatible with MPI parallelism.

Once it is built, the adjacency matrix directly gives the pair-list. Regarding the storage size involved with this approach, note that we only require single bit to tag pair–group interactions. This results in an $\left\lceil \frac{\mathbf{n}_l}{B_{\text{size}}} \right\rceil^2$ bits occupation that equals to $\left\lceil \frac{n}{B_{\text{size}}} \right\rceil^2 \frac{1}{8}$ bytes. $\boldsymbol{n}_l$ represents the number of atoms which participates to real space evaluation on a process domain ($\boldsymbol{\psi}_p$). Of course, in terms of memory we cannot afford a quadratic reservation. However, the scaling factor $\left\lceil \frac{1}{B_{\text{size}}} \right\rceil^2 \frac{1}{8}$ is small enough even for the smallest value of B_{size} set to 32 corresponding to device warp size. Not to mention that, in the context of multidevice simulation, the memory distribution is also quadratic. The pseudokernel is presented in Chart 3.

Once the adjacency matrix is built, a simple postprocessing gives us the adjacency list with optimal memory size and we can use the new list on real space computation kernels following the process described in the introduction of this subsection and illustrated in Figure 11. In addition, we benefit from a coalesced memory access pattern while loading blocks data and parameters when they are spatially reordered.

List Filtering. It is possible to improve the performance of the group–group pairing with a similar approach to the list reordering method mentioned in the OpenACC optimizations section above. By filtering every neighboring group, we can get a list of atoms that really belong to a group's neighborhood. The process is achieved by following the rule:

$$\alpha \in \mathcal{B}_I \quad \text{if } \exists \alpha_i \in \beta_I \quad \text{such that } \mathsf{dist}(\alpha_i, \alpha) \leq \mathrm{d}_{\text{cut}} + \mathrm{d}_{\text{buff}}.$$

α and α_i are atoms, β represents a group of B_{size} atoms, $\mathcal{B}$ is the neighborhood of a group and $\texttt{dist}$: $(\mathbb{R}^3 \times \mathbb{R}^3) \to \mathbb{R}$ is the euclidean distance.

An illustration of the results using the filtering process is depicted in Figure 13.

When the number of neighbor atoms is not a multiple of B_{size}, we create phantom atoms to complete the actual neighbor lists. A drawback of the filtering process is a loss of coalesced memory access pattern. As it has been entirely constructed in parallel, we do not have control of the output order. Nonetheless, this is compensated by an increase of B_{comp} for each interaction between groups, as represented by Figure 13.

Chart 3. Adjacency Matrix Construction Pseudo-kernel[a]

```
c$acc parallel loop default(present)
do i = 1, numCells
   celli = i
   !get blocks_i inside celli
   ...
c$acc loop vector
   do j = 1, numCellsNeigh
      ! Get cellj with number
      ...
      ! Get blocks_j inside cellk
      ...
      ! Apply symmetrical condition
      if ( cellj > celli ) cycle
c$acc loop seq
      do bi in blocks_i
         do bj in blocks_j
c$acc atomic
            set matrix(bj,bi) to 1
         end do
      end do
   end do
end do
```

[a]We browse through all the cells, and for each one we loop on their neighbors. It is easy to compute their ids since we know their length as well as their arrangement. Given the fact that all cells form a partition of the box, we can apply the symmetrical condition on pair-cells and retrieve the groups inside thanks to the partitioning condition, which ensures that each cell contains at most two groups.

In practice, we measure a 75% performance gain between the original list and the filtered one for the van der Waals interaction kernel. Moreover, Figure 14 (deep profile of the previous bottleneck kernel: matrix-vector product) shows a much better utilization of the device computational capability. We apply the same strategy for the other real space kernels (electrostatics and polarization).

PME Separation. As mentioned above, the Particle Mesh Ewald method separates electrostatics computation in two, real and reciprocal space. A new profiling of Tinker-HP in single-device mixed precision mode with the latest developments shows that the reciprocal part is the new bottleneck. More precisely, real space performs 20% faster than reciprocal space within a standard PME setup. Moreover, reciprocal space is even more a bottleneck in parallel because of the additional MPI communications induced by the cuFFt Transformations. This significantly narrows our chances of benefiting from the optimizations mentioned in the previous optimization subsection. However, as both parts are independent, we can distribute them on different MPI processes in order to reduce or even suppress communications inside FFts. During this operation, a subset of GPUs are assigned to reciprocal space computation only. Depending on the system size and the load balancing between real and reciprocal spaces, we can break through the scalability limit and gain additional performance on a multidevice configuration.

Mixed Precision Validation. To validate the precision study made above, we compare a 1 ns long simulation in DP on CPU (Tinker-HP 1.2) in a constant energy setup (*NVE*) with the exact same run using both GPU MP and FP implementations.

We used the solvated DHFR protein and the standard velocity verlet integrator with a 0.5 fs time step, 12 and 7 Å cutoff distances respectively for van der Waals and real space electrostatics and a convergence criteria of 1×10^{-6} for the polarization solver. A grid of 64 × 64 × 64 was used for reciprocal space with fifth-order splines. We also compare our results with a trajectory obtained with Tinker-OpenMM in MP in the exact same setup; see Figure 15.

The energy is remarkably conserved along the trajectories obtained with Tinker-HP in all cases: using DP, MP or FP with less oscillations than with Tinker-OpenMM with MP.

Available Features. The main features of Tinker-HP have been offloaded to GPU such as its various integrators like the multi-time-step integrators: RESPA1 and BAOAB-RESPA1,[48] which allow up to a 10 fs time step with PFF (this required to create new neighbor lists to perform short-range nonbonded interactions computations for both van der Waals and electrostatics). Aside from Langevin integrators, we ported

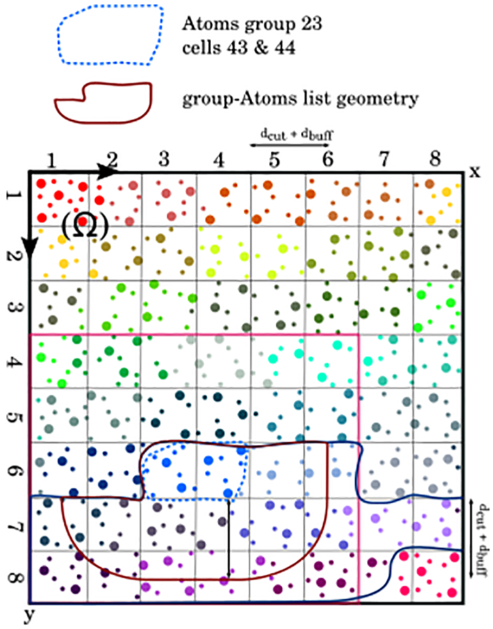


Figure 13. Starting from the situation illustrated by Figure 12, we represent the geometry resulting from the filtering process. We significantly reduce the group 23 neighborhood with the list filtering; it decreases from 144 atoms with the first list to 77 with the filtered one. B_{comp} increases, which corresponds to more interactions computed within each group pair.

the Bussi[42] (which is the default) and the Berendsen thermostats, as well as the Monte Carlo and the Berendsen barostats. We also ported free energy methods such as the Steered Molecular Dynamics[49] and van der Waals soft cores for alchemical transformations, as well as the enhanced sampling method Gaussian Accelerated Molecular Dynamics.[50] Even if it is not the main goal of our implementation as well optimized software suited to such simulations exist, we also ported the routines necessary to use standard nonpolarizable force fields such as CHARMM,[4] Amber,[5] or OPLS.[51] Still, we obtained already satisfactory performances with these models despite a simple portage, the associated numbers can be found in the Supporting Information and further optimization is ongoing. On top of all these features that concern a molecular dynamics simulation, we ported the "analyze" and "minimize" program of Tinker-HP, allowing to run single point calculations as well as geometry optimizations. All these capabilities are summed up in Table 4.

Performance and Scalability Results. We ran benchmarks with various systems on a set of different GPUs in addition to Tesla V100 nodes of the Jean-Zay supercomputer. We also ran the whole set of tests on the Irène Joliot Curie ATOS Sequana supercomputer V100 partition to ensure for the portability of the code. We used two different integrators: (2 fs RESPA along with 10 fs BAOAB-RESPA1 with heavy hydrogens). For each system, we performed 2.5 and 25 ps MD simulations with RESPA and BAOAB-RESPA1, respectively, and averaged the performance on the complete runs. van der Waals and real space electrostatics cutoffs were respectively set to 9 and 7 Å plus 0.7 Å neighbor list buffer for RESPA, 1 Å for BAOAB-RESPA1. We used the Bussi thermostat with the RESPA integrator. Induced dipoles were converged up to a 1×10^{-5} convergence threshold with the conjugate gradient solver and a diagonal preconditioner.[37] The test cases are water boxes within the range of 96000 atoms (i.e., `Puddle`) up to 2592000 atoms (i.e., `Bay`), the `DHFR`, `COX` and the Main Protease of Sars-Cov2 proteins ($\mathrm{M^{pro}}$)[52] as well as the `STMV` virus. Table 5 gathers all single devices performances, and Figure 16 illustrates the multidevice performance.

On a single GPU, the BAOAB-RESPA1 integrator performs almost twice as fast as RESPA in all cases: 22.53−42.83 ns/day on `DHFR`, 0.57−1.11 ns/day for the `STMV` virus. Regarding the RESPA integrator, results compared with those obtained in DP (Table 1) are now consistent with the Quadro V100 theoretical performance. Moreover, we observe a significant improvement on single V100 cards with DP in comparison to the OpenACC implementation, which shows that the algorithm is better suited to the architecture. However, this new algorithm considerably underperforms on Geforce architecture. For instance, for the `COX` system the speed goes from 0.65 ns/j with the OpenACC implementation to 0.19 ns/j with the adapted CUDA implementation on Geforce RTX-2080 Ti. This is obviously related to architecture constraints (lack of DP Compute units, sensitivity to SIMD divergence branch, instruction latency) and shows that there is still room for optimization. Tinker-HP is tuned to select the quickest algorithm depending on the target device. Concerning MP performance on Geforce Cards, we finally get the expected ratio compared with DP: increasing computation per access improves the use of the device (Table 3). Geforce RTX-2080 Ti and GV100 results are close until the `COX` test case, which

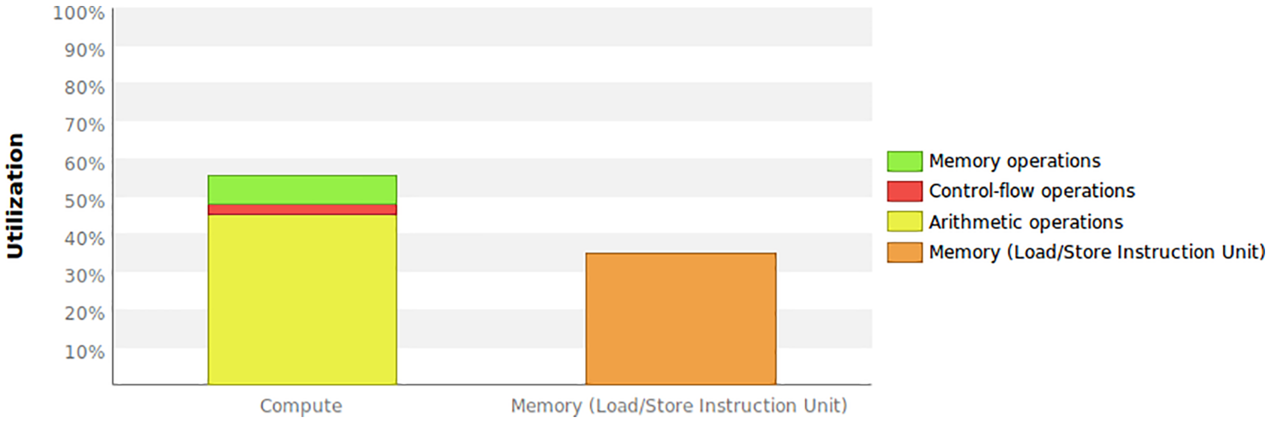


Figure 14. Real space kernel profiling results in mixed precision using our new group-Atoms list.

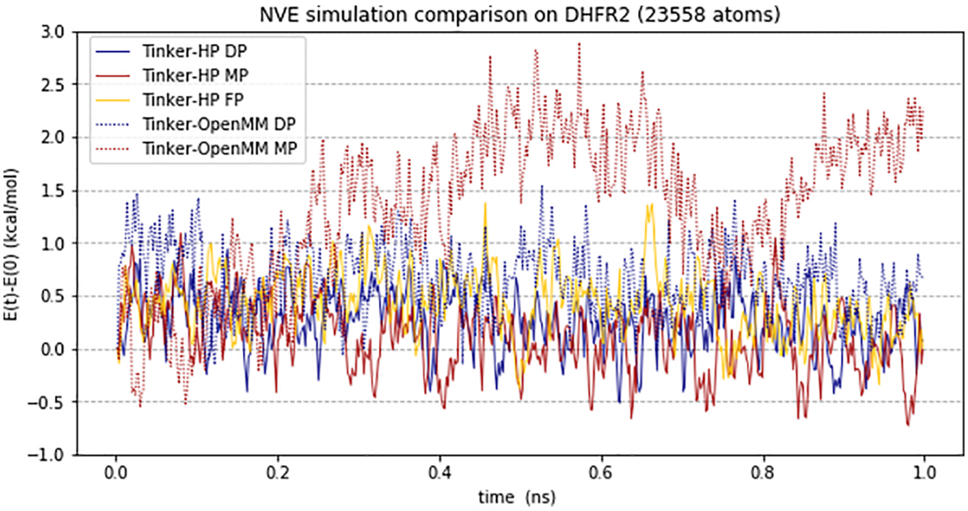


Figure 15. Variation of the total energy during a *NVE* molecular dynamics simulation of the DHFR protein in DP and MP and FP. Energy fluctuations are respectively within 1.45, 1.82, 1.75, 1.69, and 3.45 kcal/mol for Tinker-HP DP SP FP and Tinker-OpenMM DP MP.

Table 4. Available Features in the Initial Tinker-HP GPU Release

programs	dynamic; analyze; minimize; bar
integrator	VERLET (default); RESPA; RESPA1; BAOAB-RESPA; BAOAB-RESPA1
force fields	AMOEBA; CHARMM/AMBER/OPLS
miscellaneous	steered MD (SMD); Gaussian accelerated MD; restrained groups; soft cores; plumed
thermostat	Bussi (default); Berendsen
barostat	Berendsen (default); Monte Carlo

Table 5. Tinker-HP Performances in (ns/day) on Different Devices and Precision Modes

	systems						
	DHFR	M^{pro}	COX	Pond	Lake	STMV	Bay
DP Quadro GV100							
RESPA 2 fs	11.24	2.91	1.76	1.08	0.36	0.24	0.11
BAOAB-RESPA1 10 fs	22.03	6.09	3.61	2.25	0.76	0.53	0.24
MP							
RESPA 2 fs	21.75	5.98	3.69	2.20	0.70	0.44	0.20
BAOAB-RESPA1 10 fs	40.73	12.80	3.61	4.58	1.49	1.01	0.46
FP							
RESPA 2 fs	21.46	5.82	3.57	2.12	0.67	0.43	0.20
BAOAB-RESPA1 10 fs	40.65	12.65	7.77	4.52	1.47	1.00	0.45
MP Geforce RTX-2080 Ti							
RESPA 2 fs	22.52	5.35	3.21	1.82	0.54	0.33	0.15
BAOAB-RESPA1 10 fs	43.81	11.85	7.06	4.06	1.24	0.82	n/a
FP							
RESPA 2 fs	24.95	5.73	3.45	1.95	0.57	0.35	0.16
BAOAB-RESPA1 10 fs	47.31	12.78	7.63	4.35	1.32	0.87	n/a
MP Geforce RTX-3090							
RESPA 2 fs	29.14	7.79	4.76	2.81	0.91	0.60	0.28
BAOAB-RESPA1 10 fs	52.80	15.79	9.61	5.52	1.81	1.23	0.59
FP							
RESPA 2 fs	32.00	8.37	5.10	3.02	0.96	0.64	0.30
BAOAB-RESPA1 10 fs	57.67	17.20	10.46	5.96	1.90	1.32	0.63

is consistent with their computing power, but GV100 performs better for larger systems. It is certainly due to the difference in memory bandwidth which allows GV100 to perform better on memory bound kernels and to reach peak performance more easily. For example, most of PME reciprocal space kernels are memory bounded due to numerous accesses to the three-dimensional grid during the building and extracting process.

A further comparison between architectures is given in the Supporting Information.

For FP simulations, as expected, we do not see any performance difference with MP on V100 cards unlike Geforce ones, which exhibit an 8% acceleration in average as the DP accumulation is being replaced by an integer one (an instruction natively handled by compute cores). Table 6 shows the performance of Tinker-OpenMM: with the same RESPA framework, Tinker-HP performs 12−30% better on GV100 when the system size grows. With Geforce RTX-2080 Ti the difference is slightly more steady except for the Lake test case: around 18% and 25% better performance with Tinker-HP respectively with MP and FP compared to Tinker-OpenMM.

The parallel scalability starts to be effective above 100000 atoms. This is partly because of the mandatory host synchronizations needed by MPI and because of the difference in performance between synchronous and asynchronous computation under that scale (for example, DHFR production drops to 12 ns/day when running synchronously with the host). Kernel launching times are almost equivalent to their execution time and they do not overlap. Each GPU on the Jean-Zay Supercomputer comes with a 300 GB/s interconnection NVlink bandwidth. Four GPUs per node, all of them being interconnected, represents then a 100 Gb/s interconnection for each GPU pairs. The third generation PCI-Express bridge to the host memory only delivers 16 Gb/s. With the RESPA integrator operating on a full node made of 4 T V100, the speed ratio grows from 1.14 to 1.95, respectively, from Puddle to Bay test cases in comparison to a single device execution. The relatively balanced load between PME real and reciprocal space allows us to break through the scalability limit on almost every run with 2 GPUs with PME separation enabled. Performance is always worse on 4 GPUs with 1 GPU dedicated to the reciprocal space and the others to the direct space for the same reason mentioned earlier (direct/reciprocal

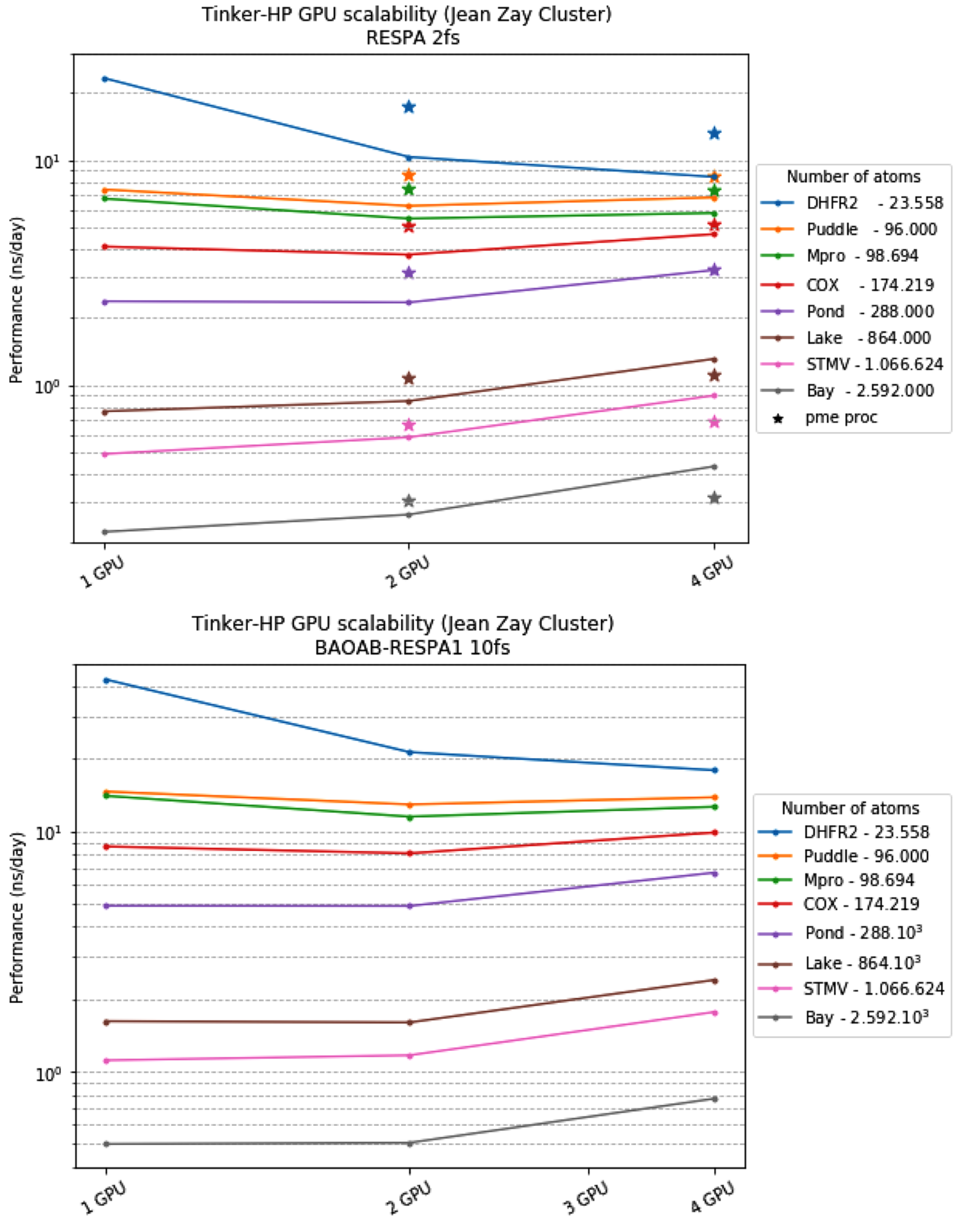


Figure 16. Single node mixed precision scalability on the Jean-Zay Cluster (V100) using the AMOEBA polarizable force field.

Table 6. Tinker-OpenMM Mixed Precision Performances Assessed with the RESPA Framework

	systems						
	DHFR	M^{pro}	COX	Pond	Lake	STMV	Bay
Quadro GV100	17.53	4.50	2.56	1.68	0.56	0.34	n/a
Geforce RTX-2080 Ti	18.97	4.37	2.63	1.66	0.55	0.28	n/a

space load balancing). We also diminished the communication overhead by overlapping communication and computation. Note that on a complete node of Jean-Zay with 4 GPUs, the bandwidth is statically shared between all of them, which means that the performance showed here on 2 GPUs is less that what can be expected on a node that would only consist in 2 GPUs interconnected through NVlink. With the BAOAB-RESPA1 integrator, ratios between a full node and a single device vary from 1.07 to a maximum of 1.58. Because of the additional short-range real space interactions, it is unsuited for PME separation, yet the reduced amount of FFt offers a potential for scalability higher than RESPA. Such a delay in the strong scalability is understandable given the device computational speed, the size of the messages size imposed by the parallel distribution, and the configuration run. The overhead of the MPI layer for STMV with BAOAB-RESPA1 and a 4 GPU bench is on average 41% of a time step. It consists mostly in FFt grid exchange in addition to the communication of dipoles in the polarization solver. This is an indication of the theoretical gain we can obtain with an improvement of the interconnect technology or the MPI layer. Ideally, we can expect to produce 2.63 ns/day on a single node instead of 1.55 ns/day. It is already satisfactory to be able to scale on such huge systems and further efforts will be made to improve multi-GPU results in the future.

■ TOWARD LARGER SYSTEMS

As one of the goals of the development of Tinker-HP is to be able to treat (very) large biological systems such as protein complexes or entire viruses encompassing up to several millions of atoms (as it is already the case with the CPU implementation[20,22] by using thousands of CPU cores), we review in the following section the scalability limit of the GPU implementation in terms of system size knowing that GPUs do not have the same memory capabilities: where classical CPU nodes routinely benefit from more than 128 GB of memory, the most advanced Ampere GPU architecture holds up to 40 GB of memory.

Tinker-HP Memory Management Model. MD with 3D spatial decomposition has its own pattern when it comes to memory distribution among MPI processes. We use the midpoint rule to compute real space interactions as it is done in the CPU implementation.

In practice, it means that each process holds information about its neighbors (to be able to compute the proper forces). More precisely, a domain $\boldsymbol{\psi}_q$ belongs to the neighborhood of $\boldsymbol{\psi}_p$ if the minimum distance between them is under some cutoff distance plus a buffer. To simplify data exchange between processes, we transfer all positions in a single message; the same thing is done with the forces.

An additional filtering is then performed to list the atoms actually involved in the interactions computed by a domain $\boldsymbol{\psi}_p$. An atom, $\alpha \in \boldsymbol{\Omega}$, belongs to domain $\boldsymbol{\psi}_p$'s interaction area $(\boldsymbol{\lambda}_p)$ if the distance between this atom and the domain is below $\frac{d_{\text{cut}} + d_{\text{buff}}}{2}$.

Let us call $\boldsymbol{n}_p$ the number of atoms belonging to $\boldsymbol{\psi}_p$, $\boldsymbol{n}_b$ the number of atoms belonging to a process domain and its neighbors, and $\boldsymbol{n}_l$ the number of atoms inside $\boldsymbol{\lambda}_p$. This is illustrated in Figure 17.

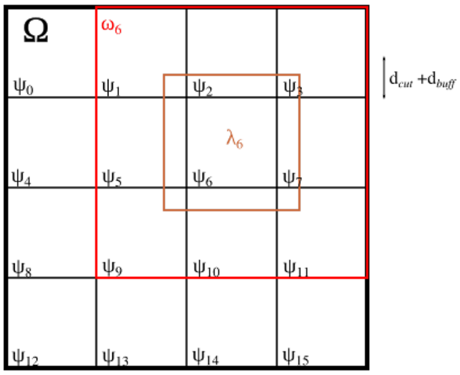


Figure 17. Two-dimensional spatial decomposition of a simulation box with MPI distribution across 16 processes. ω_6 collects all the neighboring domains of $\boldsymbol{\psi}_6$. Here $\boldsymbol{n}_b < \boldsymbol{n}$.

One can see that all data reserved with a size proportional to $\boldsymbol{n}_p$ are equally distributed among processes. Those with size proportional to $\boldsymbol{n}_b$ are only partially distributed. This means that these data structures are not distributed if all domains $\boldsymbol{\psi}_p$ are neighbors. This is why in practice the distribution only takes place at that level with a relatively high number of process, more than 26 at least on a large box with 3d domain decomposition. On the other hand, data allocated with a size proportional to $\boldsymbol{n}_l$ (like the neighbor list) are always more distributed when the number of processes increases.

On top of that, some data remain undistributed (proportional to $\boldsymbol{n}$) like the atomic parameters of each potential energy term. Splitting those among MPI processes would severely increase the communication cost, which we can not afford. As we cannot predict how one atom will interact and move inside $\boldsymbol{\Omega}$, the best strategy regarding such data is to make it available to each process. Reference Tinker-HP reduces the associated memory footprint by using MPI shared memory space: only one parameter data instance is shared among all processes within the same node.

No physical shared memory exists between GPUs of a node, and the only way to deal with undistributed data is by replicating them on each device, which is quickly impractical for large systems.

In the next section, we detail a strategy allowing to circumvent this limitation.

NVSHMEM Feature Implementation. As explained above, distribution of parameter data would necessarily results in additional communications. Regarding data exchange optimizations between GPU devices, NVIDIA develops a new library based on the OpenSHMEM[53] programming pattern, which is called NVSHMEM.[29] This library provides thread communication routines that operate on a symmetric memory on each device meaning that it is possible to initiate device communication inside kernels and not outside with an API like MPI. The immediate benefit of such approach resides in the fact that communications are automatically recovered by kernel instructions and can thereby participate to recover device internal latency. This library allows us to distribute $\boldsymbol{n}$ scale data over devices within one node.

Our implementation follows this scheme: divide a data structure (an array for instance) across devices belonging to the same node following the global numbering of the atoms and access this data inside a kernel with the help of NVSHMEM library. To do that, we rely on a NVHSMEM feature that consists of storing a symmetric memory allocation address in a pointer on which arithmetic operations can be done. Then, depending on the address returned by the pointer, either a global memory access (GBA) or a remote memory access (RMA) is instructed to fetch the data. The implementation requires a Fortran interface to be operational since NVSHMEM source code is written in the C language. Moreover, an additional operation is required for every allocation performed by the NVSHMEM specific allocator to make the data allocated accessible through OpenACC kernels. See Figure 18.

Such a singular approach affects performances since additional communications have to be made inside kernels. Furthermore, all communications do not follow a special pattern that would leave room for optimizations, meaning that each device accesses data randomly from the others depending on the atoms involved in the interactions it needs to compute. In order to limit performance loss, we can decide which data are going to be split across devices and which kernels are going to be involved with this approach. In practice, we use this scheme for the parameters of the bonded potentials.

Doing so, we distribute most of the parameter data (torsions, angles, bonds, ...) and therefore reduce the duplicated memory footprint.

Perspectives and Additional Results. During our NVSHMEM implementation, we were able to detect and optimize several memory wells. For instance, the adjacency matrix described in the Neighbor List subsection has a quadratic memory requirement following the groups of atoms. This means that this represents a potential risk of memory saturation on a single device. To prevent this, we implemented a buffer limit on this matrix to construct the pair-group list

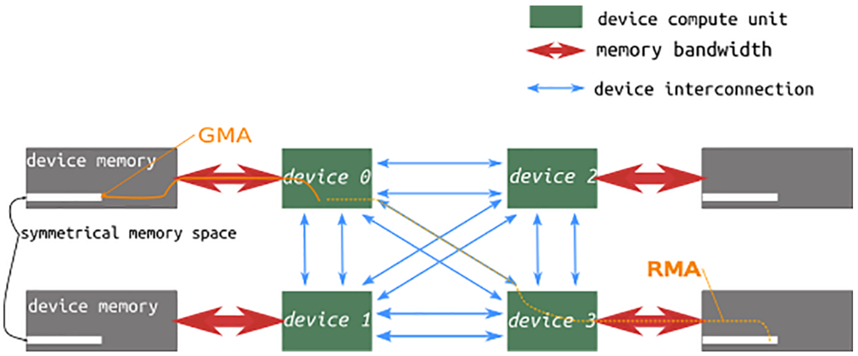


Figure 18. NVSHMEM memory distribution pattern across a four interconnected devices node. A symmetric reserved space is allocated by the NVSHMEM library at initialization. Thus, data are equally split across all devices in order. Every time a device needs to access data allocated with NVSHMEM, either a GMA or RMA is issued.

piece by piece. We also implemented algorithms that prioritize computing and searching over storing where ever needed, essentially scaling factor reconstruction. In the end, Tinker-HP is able to reach a performance of 0.15 ns/day for a 7776000 atoms water box with the AMOEBA force field and the BAOAB-RESPA1 integrator on a single V100 and scale-out to 0.25 ns/day on a complete node of Jean-Zay on the same system.

We also had the opportunity to test our implementation on the latest generation NVIDIA GPU Ampere architecture: the Selene supercomputer which is made of nodes consisting in DGX-A100 servers. A DGX-A100 server contains eight A100 graphic cards with 40 GB of memory each and with latest generation interconnection NVIDIA Switches. The results we obtained on such a node with the same systems as above in the same RESPA and BAOAB-RESPA1 framework are listed in Table 7 and Figure 19.

Table 7. Performance Synthesis and Scalability Results on the Jean-Zay (V100) and Selene (A100) Machines[a]

		Jean-Zay (V100)		Selene (A100)	
systems	size (no. of atoms)	perf (ns/day); 1GPU	best perf (ns/day); #GPU	perf (ns/day); 1GPU	best perf (ns/day); #GPU
DHFR	23558	43.83	43.83	44.96	44.96
Puddle	96000	14.63	15.76; 4	15.57	17.57
Mpro	98694	14.03	14.57; 4	16.36	17.47; 4
COX	174219	8.64	10.15; 4	10.47	11.75; 4
Pond	288000	4.90	6.72; 4	6.18	10.60; 8
Lake	864000	1.62	2.40; 4	2.11	5.50; 8
STMV	1066624	1.11	1.77; 4	1.50	4.51; 8
SARS-Cov2 Spike-ACE2	1509506	0.89	1.55; 4	1.32	4.16; 8
Bay	2592000	0.50	0.77; 4	0.59	2.38; 8
Sea	7776000	0.15	0.25; 4	0.22	0.78; 8

[a] MD production in ns/day with the AMOEBA polarizable force field.

We observe an average of 50% of performance gain for systems larger than 100000 atoms on a single A100 compared to a single V100 card. Also, the more efficient interconnection between cards (NV-switch compared to NV-link) allows us to scale better on several GPUs with the best performances ever obtained with our code on all the benchmark systems, the larger ones making use of all the 8 cards of the node. Although the code is designed to do so, the latency and the speed of the internode interconnection on the present Jean-Zay and Selene supercomputers did not allow us to scale efficiently across nodes, even on the largest systems. Jean-Zay provides 32 GB/s of network interconnection between nodes so that each GPU pair has access to a 16 GB/s bandwidth. Unlike the 100 GB/s shared between each GPU inside a node, we expect internode transit times to be 6.25−12.5 time slower without taking the latency into account. This is illustrated by the experiment summarized in Table 8 as we observe the sudden increase of the overhead of the MPI layer relative to the total duration of a time step when running on two nodes. In this case, changing the domain decomposition dimension to limit the number of neighboring process quadruples the production and exposes the latency issue (expressed here by the difference between the fastest and the slowest MPI process). In a multinode context the bottleneck clearly lies in the internode communications. The very fast evolution of the compilers, as well as the incoming availability of new classes of large pre-exascale supercomputers may improve this situation in the future. Presently, the use of multiple nodes for a single trajectory is the subject of active work within our group and results will be shared in due course. Still, one can already make use of several nodes with the present implementation by using methods such as unsupervised adaptive sampling as we recently proposed.[52] Such pleasingly parallel approach already offers the possibility to use hundreds (if not thousands!) of GPU cards simultaneously.

CONCLUSION

We presented the native Tinker-HP multi-GPU multiprecision acceleration platform. The new code is shown to be accurate and scalable across multiple GPU cards offering unprecedented performances and new capabilities to deal with long time scale simulations on large realistic systems using polarizable force fields such as AMOEBA. The approach strongly reduces the time to solution offering to achieve routine simulations that would have required thousands of CPUs on a single GPU card. Overall, the GPU-accelerated Tinker-HP reaches the best performances ever obtained for AMOEBA simulations and extends the applicability of polarizable force fields. The package is shown to be compatible with various computer GPU system architectures ranging from research laboratories to modern supercomputers.

Future work will focus on adding new features (sampling methods, integrators, ...) and on further optimizing the performance on multinodes/multi-GPUs to address the exascale challenge. We will improve the nonpolarizable force

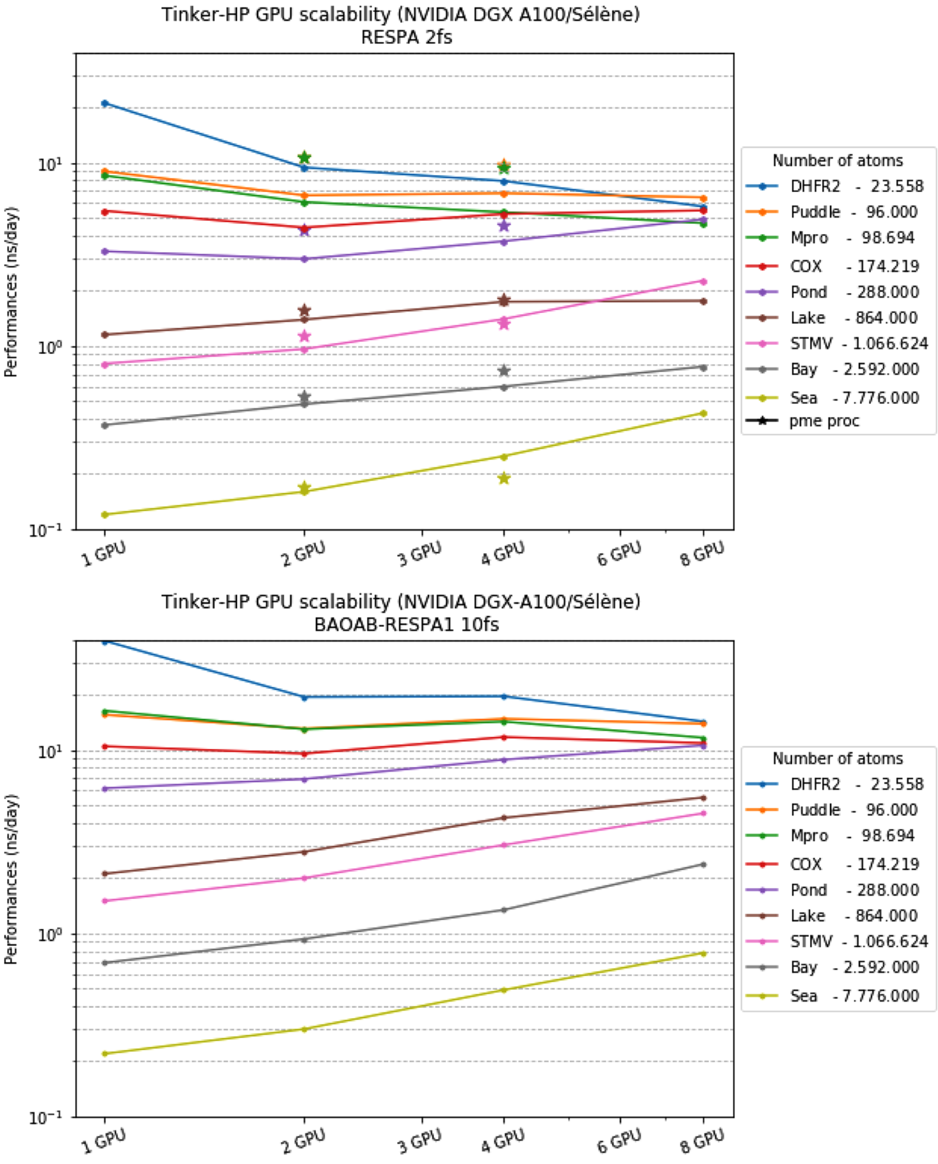


Figure 19. Performance and one node scalibility results with the AMOEBA force field.

Table 8. Multinode Performance on Jean-Zay with the Sea System and BAOAB-RESPA1[a]

no. of GPU:	4		8	
domain decomposition:	3d	1d	3d	1d
production speed (ns/day)	0.252	0.265	0.02	0.08
MPI layer (%)	24	22	97	91
MPI latency (%)	1	1	4	11

[a] Here the latency designates the time difference between the fastest and the slowest process.

field simulations capabilities as we will provide the high performance implementations of additional new generation polarizable many-body force fields such as AMOEBA+,[54,55] SIBFA,[56] and others. We will continue to develop the recently introduced adaptive sampling computing strategy enabling the simultaneous use of hundreds (thousands) of GPU cards to further reduce time to solution and deeper explore conformational spaces at high-resolution.[52] With such exascale-ready simulation setup, computations that would have taken years can now be achieved in days thanks to GPUs. Beyond this native Tinker-HP GPU platform and its various capabilities, an interface to the Plumed library[57] providing additional methodologies for enhanced-sampling, free-energy calculations, and the analysis of molecular dynamics simulations is also available. Finally, the present work, which extensively exploits low precision arithmetic, highlights the key fact that high-performance computing (HPC) grounded applications such as Tinker-HP can now efficiently use converged GPU-accelerated supercomputers, combining HPC and artificial intelligence (AI) such as the Jean-Zay machine to actually enhance their performances.

■ ASSOCIATED CONTENT

Supporting Information

The Supporting Information is available free of charge at https://pubs.acs.org/doi/10.1021/acs.jctc.0c01164.

Additional information regarding the performance using nonpolarizable force fields as well as a comparison between peak performance reachable by Tinker-HP in terms of FLOP/s (PDF)

AUTHOR INFORMATION

Corresponding Authors

Louis Lagardère − *Sorbonne Université, F-75005 Paris, France; Sorbonne Université, F-75005 Paris, France*; Email: louis.lagardere@sorbonne-universite.fr

Jean-Philip Piquemal − *Sorbonne Université, F-75005 Paris, France; Department of Biomedical Engineering, The University of Texas at Austin, Austin, Texas 78712, United States*; orcid.org/0000-0001-6615-9426; Email: jean-philip.piquemal@sorbonne-universite.fr

Authors

Olivier Adjoua − *Sorbonne Université, F-75005 Paris, France*

Luc-Henri Jolly − *Sorbonne Université, F-75005 Paris, France*

Arnaud Durocher − *Eolen, 75116 Paris, France*

Thibaut Very − *IDRIS, CNRS, 91403 Orsay, France*

Isabelle Dupays − *IDRIS, CNRS, 91403 Orsay, France*

Zhi Wang − *Department of Chemistry, Washington University in Saint Louis, Saint Louis, Missouri 63110, United States*

Théo Jaffrelot Inizan − *Sorbonne Université, F-75005 Paris, France*

Frédéric Célerse − *Sorbonne Université, F-75005 Paris, France; Sorbonne Université, F-75005 Paris, France*; orcid.org/0000-0001-8584-6547

Pengyu Ren − *Department of Biomedical Engineering, The University of Texas at Austin, Austin, Texas 78712, United States*; orcid.org/0000-0002-5613-1910

Jay W. Ponder − *Department of Chemistry, Washington University in Saint Louis, Saint Louis, Missouri 63110, United States*; orcid.org/0000-0001-5450-9230

Complete contact information is available at: https://pubs.acs.org/10.1021/acs.jctc.0c01164

Notes

The authors declare no competing financial interest.

Code Availability: The present code has been released in phase advance in link with the High Performance Computing community COVID-19 research efforts. The software is freely accessible to Academics via GitHub: https://github.com/TinkerTools/tinker-hp

ACKNOWLEDGMENTS

This work was made possible thanks to funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 810367), project EMC2. This project was initiated in 2019 with a "Contrat de Progrès" grant from GENCI (France) in collaboration with HPE and NVIDIA to port Tinker-HP on the Jean-Zay HPE SGI 8600 GPUs system (IDRIS supercomputer center, GENCI-CNRS, Orsay, France) using OpenACC. F.C. acknowledges funding from the French state funds managed by the CalSimLab LABEX and the ANR within the Investissements d'Avenir program (reference ANR11-IDEX-0004-02) and support from the Direction Génerale de l'Armement (DGA) Maîtrise NRBC of the French Ministry of Defense. Computations have been performed at GENCI on the Jean Zay machine (IDRIS) on grant no. A0070707671 and on the Irène Joliot Curie ATOS Sequana X1000 supercomputer (TGCC, Bruyères le Chatel, CEA, France) thanks to PRACE COVID-19 special allocation (projet COVID-HP). We thank NVIDIA (Romuald Josien and François Courteille, NVIDIA France) for offering us access to A100 supercomputer systems (DGX-A100 and Selene DGX-A100 SuperPod machines). P.R. and J.W.P. are grateful for support by National Institutes of Health (R01GM106137 and R01GM114237).

REFERENCES

(1) Hollingsworth, S. A.; Dror, R. O. Molecular Dynamics Simulation for All. *Neuron* **2018**, *99*, 1129−1143.

(2) Dror, R. O.; Dirks, R. M.; Grossman, J.; Xu, H.; Shaw, D. E. Biomolecular Simulation: A Computational Microscope for Molecular Biology. *Annu. Rev. Biophys.* **2012**, *41*, 429−452.

(3) Ponder, J. W.; Case, D. A. *Advances in protein chemistry*; Elsevier, 2003; Vol. *66*, pp 27−85.

(4) Huang, J.; Rauscher, S.; Nawrocki, G.; Ran, T.; Feig, M.; de Groot, B. L.; Grubmüller, H.; MacKerell, A. D. CHARMM36m: an improved force field for folded and intrinsically disordered proteins. *Nat. Methods* **2017**, *14*, 71−73.

(5) Maier, J. A.; Martinez, C.; Kasavajhala, K.; Wickstrom, L.; Hauser, K. E.; Simmerling, C. ff14SB: Improving the Accuracy of Protein Side Chain and Backbone Parameters from ff99SB. *J. Chem. Theory Comput.* **2015**, *11*, 3696−3713.

(6) Jorgensen, W. L.; Maxwell, D. S.; Tirado-Rives, J. Development and Testing of the OPLS All-Atom Force Field on Conformational Energetics and Properties of Organic Liquids. *J. Am. Chem. Soc.* **1996**, *118*, 11225−11236.

(7) Oostenbrink, C.; Villa, A.; Mark, A. E.; Van Gunsteren, W. F. A biomolecular force field based on the free enthalpy of hydration and solvation: The GROMOS force-field parameter sets 53A5 and 53A6. *J. Comput. Chem.* **2004**, *25*, 1656−1676.

(8) Shi, Y.; Ren, P.; Schnieders, M.; Piquemal, J.-P. Polarizable Force Fields for Biomolecular Modeling. In *Reviews in Computational Chemistry Vol. 28*; John Wiley and Sons, Ltd., 2015; Chapter 2, pp 51−86. DOI: 10.1002/9781118889886.ch2.

(9) Jing, Z.; Liu, C.; Cheng, S. Y.; Qi, R.; Walker, B. D.; Piquemal, J.-P.; Ren, P. Polarizable Force Fields for Biomolecular Simulations: Recent Advances and Applications. *Annu. Rev. Biophys.* **2019**, *48*, 371−394.

(10) Melcr, J.; Piquemal, J.-P. Accurate biomolecular simulations account for electronic polarization. *Front. Mol. Biosci.* **2019**, *6*, 143.

(11) Bedrov, D.; Piquemal, J.-P.; Borodin, O.; MacKerell, A. D.; Roux, B.; Schröder, C. Molecular Dynamics Simulations of Ionic Liquids and Electrolytes Using Polarizable Force Fields. *Chem. Rev.* **2019**, *119*, 7940−7995.

(12) Lopes, P. E. M.; Huang, J.; Shim, J.; Luo, Y.; Li, H.; Roux, B.; MacKerell, A. D. Polarizable Force Field for Peptides and Proteins Based on the Classical Drude Oscillator. *J. Chem. Theory Comput.* **2013**, *9*, 5430−5449.

(13) Lemkul, J. A.; Huang, J.; Roux, B.; MacKerell, A. D. An Empirical Polarizable Force Field Based on the Classical Drude Oscillator Model: Development History and Recent Applications. *Chem. Rev.* **2016**, *116*, 4983−5013.

(14) Lin, F.-Y.; Huang, J.; Pandey, P.; Rupakheti, C.; Li, J.; Roux, B.; MacKerell, A. D. Further Optimization and Validation of the Classical Drude Polarizable Protein Force Field. *J. Chem. Theory Comput.* **2020**, *16*, 3221−3239.

(15) Ren, P. Y.; Ponder, J. W. Polarizable Atomic Multipole Water Model for Molecular Mechanics Simulation. *J. Phys. Chem. B* **2003**, *107*, 5933−5947.

(16) Shi, Y.; Xia, Z.; Zhang, J.; Best, R.; Wu, C.; Ponder, J. W.; Ren, P. Polarizable Atomic Multipole-Based AMOEBA Force Field for Proteins. *J. Chem. Theory Comput.* **2013**, *9*, 4046−4063.

(17) Zhang, C.; Lu, C.; Jing, Z.; Wu, C.; Piquemal, J.-P.; Ponder, J. W.; Ren, P. AMOEBA Polarizable Atomic Multipole Force Field for Nucleic Acids. *J. Chem. Theory Comput.* **2018**, *14*, 2084−2108.

(18) Jiang, W.; Hardy, D. J.; Phillips, J. C.; MacKerell, A. D.; Schulten, K.; Roux, B. High-Performance Scalable Molecular Dynamics Simulations of a Polarizable Force Field Based on Classical Drude Oscillators in NAMD. *J. Phys. Chem. Lett.* **2011**, *2*, 87−92.

(19) Lemkul, J. A.; Roux, B.; van der Spoel, D.; MacKerell, A. D., Jr. Implementation of extended Lagrangian dynamics in GROMACS for polarizable simulations using the classical Drude oscillator model. *J. Comput. Chem.* **2015**, *36*, 1473−1479.

(20) Lagardère, L.; Jolly, L.-H.; Lipparini, F.; Aviat, F.; Stamm, B.; Jing, Z. F.; Harger, M.; Torabifard, H.; Cisneros, G. A.; Schnieders, M. J.; Gresh, N.; Maday, Y.; Ren, P. Y.; Ponder, J. W.; Piquemal, J.-P. Tinker-HP: a massively parallel molecular dynamics package for multiscale simulations of large complex systems with advanced point dipole polarizable force fields. *Chem. Sci.* **2018**, *9*, 956−972.

(21) Rackers, J. A.; Wang, Z.; Lu, C.; Laury, M. L.; Lagardère, L.; Schnieders, M. J.; Piquemal, J.-P.; Ren, P.; Ponder, J. W. Tinker 8: Software Tools for Molecular Design. *J. Chem. Theory Comput.* **2018**, *14*, 5273−5289.

(22) Jolly, L.-H.; Duran, A.; Lagardère, L.; Ponder, J. W.; Ren, P.; Piquemal, J.-P. Raising the Performance of the Tinker-HP Molecular Modeling Package [Article v1.0]. *LiveCoMS* **2019**, *1*, 10409.

(23) Stone, J. E.; Hardy, D. J.; Ufimtsev, I. S.; Schulten, K. GPU-accelerated molecular modeling coming of age. *J. Mol. Graphics Modell.* **2010**, *29*, 116−125.

(24) Götz, A. W.; Williamson, M. J.; Xu, D.; Poole, D.; Le Grand, S.; Walker, R. C. Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 1. Generalized Born. *J. Chem. Theory Comput.* **2012**, *8*, 1542−1555.

(25) Páll, S.; Zhmurov, A.; Bauer, P.; Abraham, M.; Lundborg, M.; Gray, A.; Hess, B.; Lindahl, E. Heterogeneous Parallelization and Acceleration of Molecular Dynamics Simulations in GROMACS. *J. Chem. Phys.* **2020**, *153*, 134110.

(26) Salomon-Ferrer, R.; Götz, A. W.; Poole, D.; Le Grand, S.; Walker, R. C. Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 2. Explicit Solvent Particle Mesh Ewald. *J. Chem. Theory Comput.* **2013**, *9*, 3878−3888.

(27) Eastman, P.; Swails, J.; Chodera, J. D.; McGibbon, R. T.; Zhao, Y.; Beauchamp, K. A.; Wang, L.-P.; Simmonett, A. C.; Harrigan, M. P.; Stern, C. D.; Wiewiora, R. P.; Brooks, B. R.; Pande, V. S. OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. *PLoS Comput. Biol.* **2017**, *13*, e1005659.

(28) Harger, M.; Li, D.; Wang, Z.; Dalby, K.; Lagardère, L.; Piquemal, J.-P.; Ponder, J.; Ren, P. Tinker-OpenMM: Absolute and relative alchemical free energies using AMOEBA on GPUs. *J. Comput. Chem.* **2017**, *38*, 2047−2055.

(29) Potluri, S.; Luehr, N.; Sakharnykh, N. Simplifying Multi-GPU Communication with NVSHMEM. *GPU Technology Conference*; NVIDIA, 2016.

(30) Frenkel, D.; Smit, B. *Understanding molecular simulation: from algorithms to applications*; Elsevier, 2001; Vol. *1*.

(31) Wienke, S.; Springer, P.; Terboven, C.; an Mey, D. OpenACC—First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*. Euro-Par 2012. Lecture Notes in Computer Science; Kaklamanis, C., Papatheodorou, T., Spirakis, P. G., Eds.; Springer: Berlin, Heidelberg, 2012; Vol. *7484*, pp 859−870 DOI: 10.1007/978-3-642-32820-6_85.

(32) Chandrasekaran, S.; Juckeland, G. *OpenACC for Programmers: Concepts and Strategies*, 1st ed.; Addison-Wesley Professional, 2017.

(33) Sanders, J.; Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming*; Addison-Wesley Professional, 2010.

(34) Volkov, V. Understanding Latency Hiding on GPUs. *Ph.D. thesis*, EECS Department, University of California, Berkeley, 2016.

(35) Kraus, J. An introduction to CUDA-aware MPI. *NVIDIA Developer Blog*; 2013; https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

(36) Essmann, U.; Perera, L.; Berkowitz, M. L.; Darden, T.; Lee, H.; Pedersen, L. G. A smooth particle mesh Ewald method. *J. Chem. Phys.* **1995**, *103*, 8577−8593.

(37) Lagardère, L.; Lipparini, F.; Polack, E.; Stamm, B.; Cancès, E.; Schnieders, M.; Ren, P.; Maday, Y.; Piquemal, J.-P. Scalable Evaluation of Polarization Energy and Associated Forces in Polarizable Molecular Dynamics: II. Toward Massively Parallel Computations Using Smooth Particle Mesh Ewald. *J. Chem. Theory Comput.* **2015**, *11*, 2589−2599.

(38) NVIDIA Corporation. *CUDA Toolkit 11.1 CUFFT Library Programming Guide 2020*; NVIDIA, 2020;http://developer.nvidia.com/nvidia-gpu-computing-documentation.

(39) Lipparini, F.; Lagardère, L.; Stamm, B.; Cancès, E.; Schnieders, M.; Ren, P.; Maday, Y.; Piquemal, J.-P. Scalable Evaluation of Polarization Energy and Associated Forces in Polarizable Molecular Dynamics: I. Toward Massively Parallel Direct Space Computations. *J. Chem. Theory Comput.* **2014**, *10*, 1638−1651.

(40) Phillips, J. C.; Zheng, Gengbin; Kumar, S.; Kale, L. V. NAMD: Biomolecular Simulation on Thousands of Processors. *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*; ACM, 2002; pp 36−36.

(41) Tuckerman, M.; Berne, B. J.; Martyna, G. J. Reversible multiple time scale molecular dynamics. *J. Chem. Phys.* **1992**, *97*, 1990−2001.

(42) Bussi, G.; Donadio, D.; Parrinello, M. Canonical sampling through velocity rescaling. *J. Chem. Phys.* **2007**, *126*, 014101.

(43) Zhou, J.; Ross, K. A. Implementing database operations using SIMD instructions. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*; ACM, 2002; pp 145−156.

(44) Nickolls, J.; Dally, W. J. The GPU computing era. *IEEE micro* **2010**, *30*, 56−69.

(45) Le Grand, S.; Götz, A. W.; Walker, R. C. SPFP: Speed without compromise—A mixed precision model for GPU accelerated molecular dynamics simulations. *Comput. Phys. Commun.* **2013**, *184*, 374−380.

(46) Yates, R. Fixed-point arithmetic: An introduction. *Digital Signal Labs* **2009**, *81*, 198.

(47) Bowers, K. J.; Dror, R. O.; Shaw, D. E. The midpoint method for parallelization of particle simulations. *J. Chem. Phys.* **2006**, *124*, 184109.

(48) Lagardère, L.; Aviat, F.; Piquemal, J.-P. Pushing the Limits of Multiple-Time-Step Strategies for Polarizable Point Dipole Molecular Dynamics. *J. Phys. Chem. Lett.* **2019**, *10*, 2593−2599.

(49) Célerse, F.; Lagardère, L.; Derat, E.; Piquemal, J.-P. Massively parallel implementation of Steered Molecular Dynamics in Tinker-HP:comparisons of polarizable and non-polarizable simulations of realistic systems. *J. Chem. Theory Comput.* **2019**, *15*, 3694−3709.

(50) Miao, Y.; Feher, V. A.; McCammon, J. A. Gaussian accelerated molecular dynamics: Unconstrained enhanced sampling and free energy calculation. *J. Chem. Theory Comput.* **2015**, *11*, 3584−3595.

(51) Jorgensen, W. L.; Maxwell, D. S.; Tirado-Rives, J. Development and Testing of the OPLS All-Atom Force Field on Conformational Energetics and Properties of Organic Liquids. *J. Am. Chem. Soc.* **1996**, *117*, 11225−11236.

(52) Jaffrelot-Inizan, T.; Célerse, F.; Adjoua, O.; El Ahdab, D.; Jolly, L.-H.; Liu, C.; Ren, P.; Montes, M.; Lagarde, N.; Lagardère, L. High-Resolution Mining of SARS-CoV-2 Main Protease Conformational Space: Supercomputer-Driven Unsupervised Adaptive Sampling. *Chem. Sci.* **2021**, DOI: 10.1039/D1SC00145K.

(53) Chapman, B.; Curtis, T.; Pophale, S.; Poole, S.; Kuehn, J.; Koelbel, C.; Smith, L. Introducing OpenSHMEM: SHMEM for the PGAS community. *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*; ACM, 2010; pp 1−3.

(54) Liu, C.; Piquemal, J.-P.; Ren, P. AMOEBA+ Classical Potential for Modeling Molecular Interactions. *J. Chem. Theory Comput.* **2019**, *15*, 4122−4139.

(55) Liu, C.; Piquemal, J.-P.; Ren, P. Implementation of Geometry-Dependent Charge Flux into the Polarizable AMOEBA+ Potential. *J. Phys. Chem. Lett.* **2020**, *11*, 419−426.

(56) Gresh, N.; Cisneros, G. A.; Darden, T. A.; Piquemal, J.-P. Anisotropic, polarizable molecular mechanics studies of inter-, intramolecular interactions, and ligand-macromolecule complexes. A bottom-up strategy. *J. Chem. Theory Comput.* **2007**, *3*, 1960−1986.

(57) Bonomi, M.; Bussi, G.; Camilloni, C.; Tribello, G. A.; Banáš, P.; Barducci, A.; Bernetti, M.; Bolhuis, P. G.; Bottaro, S.; Branduardi, D.; et al. Promoting transparency and reproducibility in enhanced molecular simulations. *Nat. Methods* **2019**, *16*, 670−673.