Computational Materials Science xxx (xxxx) xxxx

Contents lists available at ScienceDirect

Computational Materials Science

journal homepage: www.elsevier.com/locate/commatsci

Unified memory in HOOMD-blue improves node-level strong scaling

Jens Glaser[a], Peter S. Schwendeman[a], Joshua A. Anderson[a], Sharon C. Glotzer[a,b,c,*]

[a] Dept. of Chemical Engineering, University of Michigan, 2800 Plymouth Rd, Ann Arbor, MI 48109, United States
[b] Biointerfaces Institute, University of Michigan, Ann Arbor, MI 48109, United States
[c] Department of Materials Science and Engineering, University of Michigan, Ann Arbor, MI 48109, United States

ARTICLE INFO

Keywords:
Molecular dynamics
CUDA
GPUs
NVLINK
Unified memory
Rigid bodies

ABSTRACT

Current supercomputer designs rely on increasing the compute density inside a node to maximize the performance of applications that tightly integrate the processors within a shared memory space. HOOMD-blue 2.5 enables molecular dynamics simulations that take advantage of multiple GPUs inside the same node which are connected via NVLINK. We describe the native implementation of CUDA unified memory in HOOMD-blue for strong scaling on this hardware, and provide performance benchmarks.

1. Introduction

By optimizing the performance of molecular simulations on modern high performance computing (HPC) architectures, researchers are able to study more complex models on unprecedented spatial and temporal scales. For example, coarse-grained or atomistic models of biomolecular systems require high performance to sample a large number of configurations relevant to their higher order assembly. Key to overcoming these challenges is the strong scaling capability of the codes used, whereby the time to solution is reduced by taking advantage of parallelism. Current mainstream molecular simulation codes, such as AMBER, GROMACS, NAMD, and LAMMPS, exploit parallelism at various levels. AMBER, GROMACS and NAMD are biomolecular simulation codes. AMBER supports acceleration on GPUs and spatial decomposition using the message passing interface (MPI). GROMACS parallelizes using SIMD, multithreading, GPUs, and MPI domain decomposition [1]. NAMD is designed for large scale simulations, supports GPUs and a load balancing scheme based on "patches" with CHARM++ [2]. LAMMPS is a general-purpose simulation code, and supports GPUs, Intel Xeon Phi coprocessors, threads *via* OpenMP and spatial domain decomposition *via* MPI[3]. Extensions of LAMMPS, such as the USERMESO package have been benchmarked on the Summit supercomputer at Oak Ridge National Laboratory, measuring the impact of the NVLINK interconnect on MPI communication performance [4].

The Summit supercomputer, an IBM AC922 machine at Oak Ridge National Laboratory, features a 'dense node' design that includes six GPUs in a single compute node. These GPUs are connected by the NVLINK 2 interconnect and support addressing of a shared memory space via the unified memory feature of the CUDA programming model [5]. Here, we outline the implementation, challenges overcome and performance of enabling unified memory in HOOMD-blue. We released the new features in version 2.5 [6]. Previous versions of HOOMD-blue [7], beginning with release 1.0, only supported strong scaling on multiple GPUs based on MPI [8], a capability that was tailored to the previous generation of supercomputers such as the Titan supercomputer, a Cray XK7 machine at Oak Ridge National Laboratory, which only contains a single GPU per compute node.

Here, we describe an implementation of unified memory in HOOMD-blue. Unified memory was introduced in CUDA with release 6.0 in early 2014, and it was further refined for Pascal GPUs which have compute capability 6.0, to support concurrent access to memory from both the GPU and CPU. The core idea of unified memory is the ability to address memory that physically resides either in GPU or in CPU *via* a single memory space. CUDA implements a page-faulting capability, which ties in with the Linux kernel and allows it to automatically bring chunks of data into the device-local memory space for reading or writing *on demand*. The size of these chunks can vary, from the Linux kernel page size (4 kB on Intel CPUs, 64 kB on IBM Power CPUs) to several pages. Hardware of compute architecture 6.0 enables this capability through a virtual 49bit memory pointer. In this way, the total available memory space and even the size of a single allocation can exceed the size of local GPU memory, expanding the possibilities for memory-intensive applications (Fig. 1).

In the context of molecular simulations, which tend to be compute and/or memory bandwidth bound, it is important to enable fast access to the unified memory space between different GPUs. The NVLINK

* Corresponding author.
E-mail address: sglotzer@umich.edu (S.C. Glotzer).

https://doi.org/10.1016/j.commatsci.2019.109359
Received 29 May 2019; Received in revised form 16 October 2019; Accepted 17 October 2019

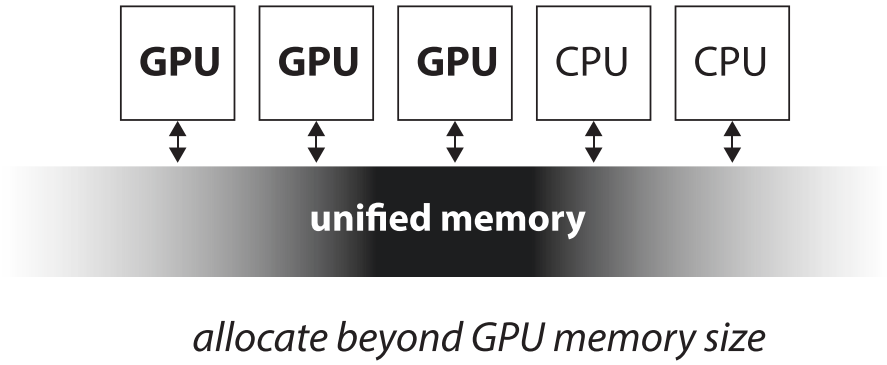

Fig. 1. Schematic visualization of the unified memory space. Adapted from http://devblogs.nvidia.com/unified-memory-cuda-beginners/.

interconnect is specialized for fast data exchange between GPUs at up to 50 GB/s bidirectional bandwidth per lane [9]. Additionally, on the IBM AC 922 architecture, it allows data transfer between the GPUs and the CPU at the same rates. While the initial version of NVLINK (NVLINK 1) supports Pascal GPUs, the current version (NVLINK 2) supports the current generation of Volta GPUs. Improvements of NVLINK 2 include concurrent access between the GPUs and the CPU, multi-GPU atomics, higher bandwidth and number of lanes, and CPU–GPU cache coherence.

The unified memory design of HOOMD-blue emerged out of considerations how to effectively implement a load-balancing communciation algorithm [10] for multi-GPU simulations. It became clear that due to HOOMD-blue's existing capability of performing a spatial sort of the particle data along a Hilbert curve [7], the sorting algorithm should already provide good load balancing compared to spatial domain decomposition, if it is possible to map the particle data onto multiple GPUs in a contiguous fashion. However, this is precisely what is enabled through unified memory and NVLINK technology. Therefore, the main challenge in achieving strong-scaling on multiple GPUs of the same compute node consists in enabling support for unified memory in HOOMD-blue.

The remainder of the paper is organized as follows. In Section 2.1 we describe how we ported HOOMD-blue's memory management routines to unified memory through CUDA's `cudaMallocManaged` interface. Section 2.2 discusses how particle data is distributed among GPUs. In Section 2.3 we address the challenges of improving data locality. Some GPU algorithms pose unique optimization challenges, which we will summarize in Section 2.4. Section 3 compares the performance of the new code path, between NVLINK 1 and 2 (Section 3.1), with and without spatial sort (Section 3.2), between MPI and unified memory code paths for a Lennard-Jones (LJ) liquid (Section 3.3), and finally more complex examples (Section 3.4).

2. Implementation

We refactored the core memory management routines in HOOMD-blue, with additional features (unified memory support), modernization (`C++11`), and backwards code compatibility in mind. Only a subset of HOOMD-blue's `C++` classes has been enabled to take advantage of unified memory currently (see Table 1). However, the framework put in place allows one to easily generalize additional classes as needed.

2.1. Unified memory

We implemented a new class, `GlobalArray`, to manage unified memory allocations. Up to version 2.4, memory was managed through a `C++` template class called `GPUArray` that keeps track of the current memory location (host or device). `GPUArray` provides a light-weight API, `ArrayHandle`, that takes a requested access location and mode (`read`, `readwrite`, and `overwrite`) as arguments. In this way, we avoid unnecessary copies of the pinned memory regions [7]. Since all of HOOMD-blue's algorithms rely on this interface, `GlobalArray` also supports it for backwards compatibility. It provides a wrapper to memory allocated with `cudaMallocManaged`. On GPUs with compute capability $\geqslant 6.0$, these memory regions are simultaneously accessible from all GPUs in the same execution context. To facilitate implementation of the new class, and the phasing-out of the old one, `GPUArray` and `GlobalArray` share the same interface through static polymorphism, as outlined in Fig. 2. An additional requirement is that `GlobalArray`s should be able to fall back onto pinned memory behavior when unified memory is needed, *i.e.* in single GPU simulations (see also Section 2.4.4). Internally, we manage the data pointers using `C++11` smart pointers (`std::unique_ptr< >`) with custom deleters, to support all standard `C++11` operators, including move operators.

Table 1
Features enabled for multi-GPU execution with unified memory in HOOMD-blue 2.5.2.

- Core data structures (`ParticleData`, `ForceCompute`, `ParticleGroup`)
- Cell neighbor lists (`md.nlist.cell()`)
- Velocity Verlet integrators (`md.integrate.nve()`, `md.integrate.nvt()`, `md.integrate.npt()`, `md.integrate.nph()`).
- Pair potentials (`md.pair.*`, with exception of multi-body potentials)
- Rigid bodies (`md.constrain.rigid()`)
- Particle–particle particle-mesh electrostatics (`md.charge.pppm()`).

2.2. Particle data partitions and kernel launches

We rely on and extend the existing *atom decomposition* approach [7], which assigns one particle to a fixed number of one or more threads of the GPU. In the multi-GPU code path, we split the atoms evenly between GPUs according to their index in the particle data. The spatial sort along a Hilbert curve guarantees data locality, which is important to minimize page faults between GPUs. The particle split is handled by the `GPUPartition` class, which stores the range of particle indices assigned to a GPU and also switches the CUDA context to that particular GPU (Fig. 3). Porting a kernel to multi-GPU execution is as simple as providing an additional argument for the starting particle index together with the size of the contiguous interval of particles indices that this kernel processes. Then, we invoke the kernels by iteration over the logical GPUs and issue kernel launches (⋘…⋙ syntax) from the same host process. In particular, we count down from the GPU with the highest index to GPU 0, which is the default GPU that we use for all kernels that do not (yet) support multi-GPU execution. `GPUPartition`s in HOOMD-blue are always associated with particle indices, but sometimes only for a subset of the particles, e.g. those which are members of a `ParticleGroup`.

To guarantee the validity of the simulation results in the presence of data dependencies, we need to insert explicit synchronization barriers. For

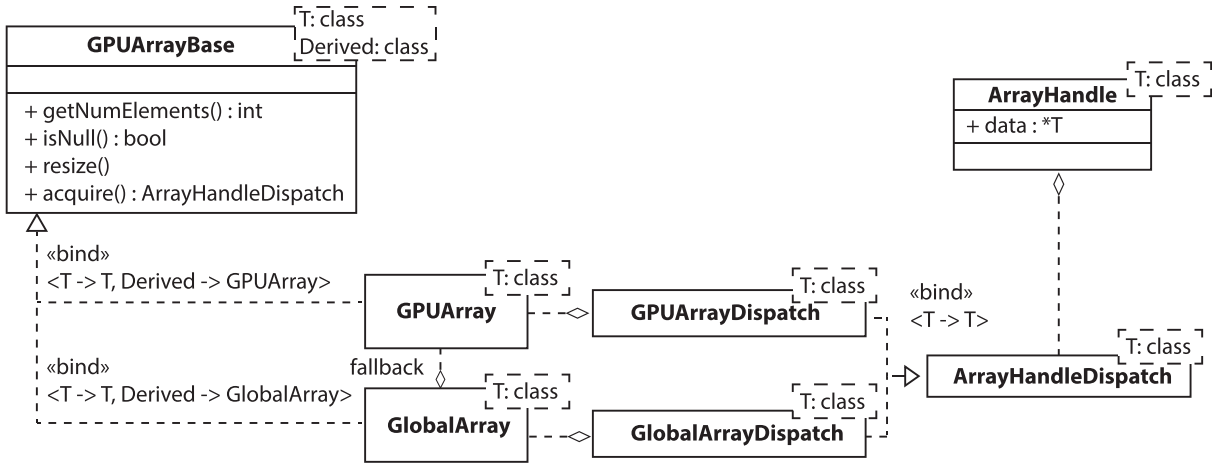

Fig. 2. Class hierarchy (UML diagram) of GPUArray/GlobalArray. The use of static polymorphism based on the curiously recurring template pattern (CRTP) avoids link time ambiguities associated with virtual template classes. To not have to explicitly instantiate templates of the `ArrayHandle` class that depend on the data storage mechanism of `GPUArray/GlobalArray` whenever the arrays are accessed, the `ArrayHandleDispatch` abstract class holds a pointer to the data and the owning `GPUArray/GlobalArray`, and therefore converts static into runtime polymorphism. `ArrayHandleDispatch` objects are passed from the data management classes to the `ArrayHandle` class. `GlobalArray` optionally falls back on `GPUArray` (see Section 2.4.4).

a single GPU and execution stream, synchronization is implicit. However, CUDA does not perform implicit synchronization between different GPUs, so that a kernel launched on one GPU may overlap with a kernel launched on a different GPU. We therefore explicitly place barriers before and after every multi-GPU kernel launch. In practice, we accomplish this through a sequence of `cudaEventRecord` and `cudaStreamWaitEvent` calls as outlined in Fig. 4, which we accomplish using the methods `ExecutionConfiguration::beginMultiGPU()` and `endMultiGPU()`.

2.3. Data locality and memory hints

Instructing the CUDA driver about the access pattern for the memory regions it manages is indispensable for good multi-GPU performance. Every memory region is physically stored in some memory space, but this location is determined by the CUDA driver based on performance heuristics. For instance, the driver is capable of avoiding page faults by setting up mappings between GPUs. It can also tie a specific memory range to a given GPU and therefore prevent it from migrating to a different one. The behavior is controlled through *hints* for memory locality as arguments to `cudaMemAdvise`. These hints are optional from a correctness viewpoint, but they allow the driver to make informed migration decisions that produce the fastest performance for a given data access pattern. As most kernels in HOOMD-blue operate on a fixed subset of particle indices, it is advantageous to associate subintervals of the particle data with specific GPUs (see Fig. 5). The `cudaMemAdviseSetPreferredLocation` hint assigns a memory range to a specific GPU or the CPU. To ensure that the data initially resides on that device, the `cudaMemAdvise` call needs to be accompanied by a call to `cudaMemPrefetchAsync`. The hint `cudaMemAdviseSetAccessedBy` indicates that a portion of memory will also be accessed by another device, and instructs the driver to set up a permanent memory mapping between the preferred or actual data location and the accessing device. Finally, some memory allocations change less frequently (such as pair potential parameters), but are read from in every time step. Those memory ranges are marked with the `cudaMemAdviseSetReadMostly` flag, which caches their contents in the memory of all devices that access them once. However, write accesses invalidate those memory pages and cause expensive re-duplication.

For debugging and performance optimizations, the CUDA visual profiler (`nvvp`) provides useful information on every page fault that occurs, and these are visible when the option “Use fixed width segments for unified memory timeline” is unchecked. As of the time of writing this article, technical limitations in `nvvp` do not allow tracing the location of the page fault back to the source line where it occurred. However, this correlation is important and allows one to identify the

TwoStepNVTMTKGPU.cu

```
// iterate over active GPUs in reverse, to finish on GPU 0
for (int idev = gpu_partition.getNumActiveGPUs() - 1; idev >= 0; --idev)
    {
    auto range = gpu_partition.getRangeAndSetGPU(idev);

    unsigned int nwork = range.second - range.first;
    unsigned int offset = range.first;

    // setup the grid to run the kernel
    dim3 grid( (nwork/run_block_size) + 1, 1, 1);
    dim3 threads(run_block_size, 1, 1);

    // run the kernel, starting with offset
    gpu_nvt_mtk_step_one_kernel<<< grid, threads >>>(d_pos, .., nwork, offset);
    }
```

Fig. 3. Example of a kernel launch with work distributed over multiple GPUs (`CUDA/C++ code`).

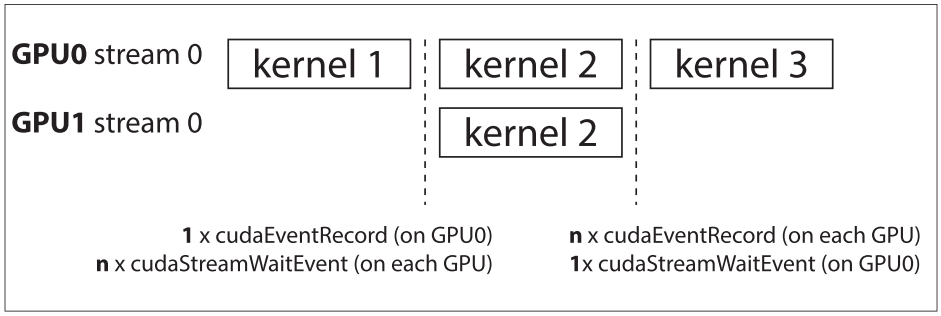

Fig. 4. Explicit synchronization is necessary before and after concurrent kernel launches on multiple GPUs.

ParticleGroup.cc

```
GlobalArray<unsigned int> member_idx(member_tags.size(),
                                     m_pdata->getExecConf());
m_member_idx.swap(member_idx);

#ifdef ENABLE_CUDA
if (m_pdata->getExecConf()->isCUDAEnabled())
    {
    cudaMemAdvise(m_member_idx.get(),
        m_member_idx.getNumElements()*sizeof(unsigned int),
        cudaMemAdviseSetReadMostly, 0);
    CHECK_CUDA_ERROR();
    }
#endif
```

ParticleData.cc

```
// split preferred location of particle data across GPUs
for (unsigned int idev = 0; idev < m_exec_conf->getNumActiveGPUs(); ++idev)
    {
    cudaDeviceProp dev_prop = m_exec_conf->getDeviceProperties(idev);

    if (!dev_prop.concurrentManagedAccess)
        continue;

    auto range = m_gpu_partition.getRange(idev);
    unsigned int nelem =  range.second - range.first;

    if (!nelem)
        continue;

    cudaMemAdvise(m_pos.get()+range.first, sizeof(Scalar4)*nelem,
        cudaMemAdviseSetPreferredLocation, gpu_map[idev]);
    ...

    // migrate data to preferred location
    cudaMemPrefetchAsync(m_pos.get()+range.first, sizeof(Scalar4)*nelem,
        gpu_map[idev]);
    ...
    }
```

Fig. 5. Examples for applying memory usage hints (`cudaMemAdvise`) and pre-fetching (`cudaMemPrefetchAsync`) to improve data locality and reduce the number of page faults.

place in the code that would benefit from e.g. a `cudaMemAdvise` instruction or which should be refactored for performance. Nevertheless, `nvvp` does provide the memory *address*. Therefore, we implemented a tracking feature (a `memory_traceback=True` option in `context.initialize()`) that outputs the names, data types and memory start addresses of all unified memory allocations used by the program, at the end of each run. For each memory region, it also provides a stack trace of the last three function calls at time of allocation. The latter information allows one to easily identify the provenance of that allocation. In the code, memory regions are optionally named through a `TAG_ALLOCATION()` macro when they are allocated. Fig. 6 shows example output from this feature.

2.4. Particular challenges

For the initial implementation of unified memory in HOOMD-blue 2.5.x, only the features and algorithms outline in Table 1 have been ported to multi-GPU execution. One may still combine them with single GPU features, however, this will violate strict data locality and therefore increase the frequency of page faults and data migrations. The resulting performance may be poor. Some of the core algorithms in HOOMD-blue pose particular optimization challenges, which we outline here.

```
>>> hoomd.context.initialize(memory_traceback=True)
...
>>> hoomd.run(100)
...

notice(2): Total amount of managed memory allocated through Global[Array,Vector]: 90.3MB
notice(2): Actual allocation sizes may be larger by up to the OS page size due to alignment.
notice(2): List of memory allocations and last 3 functions called at time of (re-)allocation
notice(2): ** Address 0x7f3826000000, 37.1MB, data type unsigned int [m_nlist]
notice(2): (1) ...
notice(2): (2) ...
notice(2): (3) NeighborListGPU::buildHeadList()
notice(2): ** Address 0x7f3828600000, 541kB, data type unsigned int [m_idx_scratch]
notice(2): (1) ...
notice(2): (2) ...
notice(2): (3) CellListGPU::initializeMemory()
notice(2): ** Address 0x7f3846000000, 1.95MB, data type double4 [m_angmom_alt]
notice(2): (1) ...
notice(2): (2) ...
notice(2): (3) ParticleData::allocateAlternateArrays(unsigned int)
...
```

Fig. 6. Example output of the memory traceback – source code correlation capability to identify memory regions involved in page faults.

2.4.1. Cell list

The cell list is used in the neighbor search for pairwise interactions and sorts particles in grid cells, which are search volumes for neighbors within the cut-off distance of a given particle. It has a different data layout than the particle data and maps a three-dimensional cartesian cell index onto a one-dimensional cell index using row-major layout. While particles that are close together typically also fall into the same cell, splitting the one-dimensional cell index across GPUs in analogy to the particle data partition does not guarantee that there will be a direct correspondence between multi-GPU partitions of the particle data and the cell index. To mitigate this performance issue, we decided to construct a separate cell list on each GPU, which we fill only with particles that are handled by that particular GPU. Only if necessary, these cell lists are combined into a single list for classes that do not (yet) support per-device cell lists. For the special case of cubic boxes, it is possible to align the sequence of the space-filling curve with the cell order to further improve data locality [11], but this has not been attempted here.

2.4.2. Neighbor list

Based on the cell list, the neighbor list `nlist.cell()` constructs the table of particle neighbors that lie exactly within a spherical cutoff r_{cut}. The changes to the neighbor list for multi GPU execution with unified memory turned out to be rather straightforward, because it maps GPU threads on particle indices. We split particle indices on GPUs using the `GPUPartition` as described above. In addition, the class `NeighborListGPUBinned` iterates over the per-device cell list when searching for neighbors of a particle. It is beneficial for performance to look up particles by index, rather than to store position information in the cell list. This strategy reduces the excess amount of data transferred when page faults occur.

Bounding volume hierarchies (BVHs) [12] and non-rectilinear cell lists [13] offer better performance for systems with very disparate pair cut-offs. However, the BVH-based neighbor list in HOOMD-blue [14,], `nlist.tree()`, is not yet optimized for unified memory.

2.4.3. Rigid bodies

Rigid bodies in HOOMD-blue are groups of particles held together by rigid constraints (`md.constrain.rigid()`). To optimize rigid bodies for multi-GPU execution, we introduce a separate `GPUPartition` of the central particles of rigid bodies. Every time step, the forces on the central particle are computed by a reduction over the constituent particle forces. During this phase, threads are split onto GPUs according to the association of the central particles with a given GPU, and multiple threads are assigned to each central particle. Additionally, we look up the properties of central particles *via* a table that is split across GPUs using `GPUPartition` for improved memory locality.

2.4.4. GPUArray fallback

During performance testing we found that some single-GPU simulations in HOOMD-blue 2.4 without unified memory ran faster than those which used unified memory. We determined the root cause to be an increased launch latency for kernels that launch after a synchronization point (`cudaDeviceSynchronize`). At the time of implementation, a fix for this CUDA issue was not yet available. We therefore decided to implement a capability in `GlobalArray` to fall back onto standard pinned memory, in order not to compromise single GPU performance (see Fig. 2). At compile time, the user can decide to always enable unified memory (`ALWAYS_USE_MANAGED_MEMORY = ON`).

2.4.5. PPPM electrostatics

Electrostatics in HOOMD-blue are computed using the particle-particle, particle-mesh algorithm (`md.charge.pppm()`) [15]. The short range part benefits from multi GPU acceleration as all other pair potentials do. To be able to run the long range calculations on multiple GPUs, we updated the charge assignment and force interpolation kernels. Therefore, the data access pattern is consistent with GPU locality. For the current implementation, we decided to leave the Fast Fourier Transform (FFT) and all reciprocal space calculations as they are. As a micro-optimization, we perform the same FFT on all GPUs in parallel, avoiding extra data transfer. In the future, the FFT may additionally be accelerated by exploiting the multi-GPU capabilities of CUFFT (`cufftXt`, [16]).

3. Performance results

We assess the performance benefit of the unified memory code path using examples on Pascal and Volta architectures, which support compute capability 6.0 and NVLINK 1, and compute capability 7.0 and NVLINK 2, respectively. We benchmark on one, two and three GPUs. A Summit node has six GPUs, however only three of them connect within the same NVLINK topology and to the same CPU. Moreover, we compare between unified memory and MPI code paths. To launch HOOMD 2.x ($x \geqslant 5$) on multiple GPUs, one passes a command line option with the requested GPU ids, *e.g.*, `-gpu = 0,1,2`. This explicit choice of GPUs

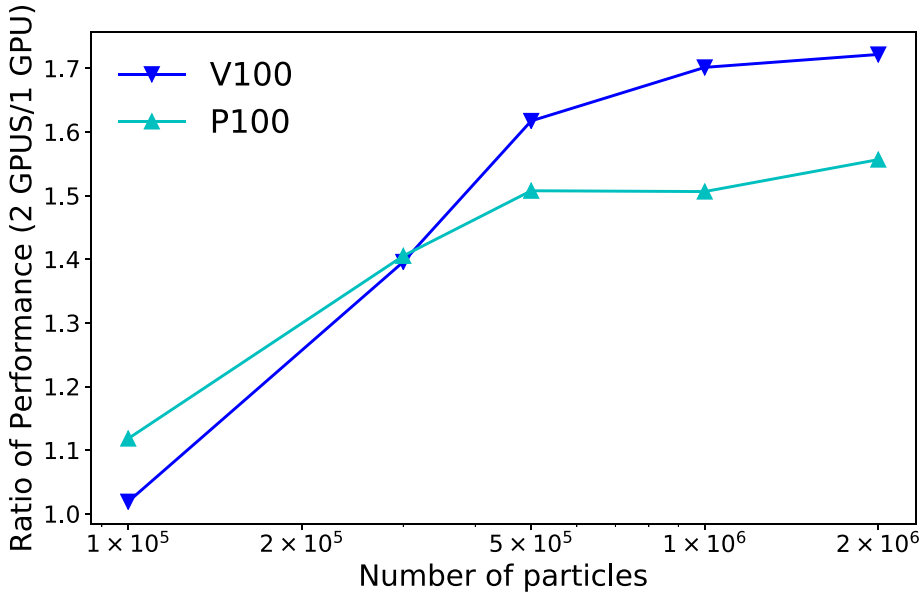

Fig. 7. Relative performance of the LJ liquid ($r_{cut} = 3\sigma$) with varying number of particles N on two GPUs compared to one GPU, for Pascal (NVLINK1, upward facing triangles) and Volta (NVLINK2, downward facing triangles).

allows the user to assign a variable number of devices to every MPI rank; by default, HOOMD-blue runs on the single fastest GPU. In the following, we only benchmark on NVLINK hardware; even though possible, execution on Pascal generation devices and later *without* NVLINK relies on peer copies over the PCIe bus and usually leads to performance degradation.

3.1. NVLINK 1 vs. NVLINK 2

We benchmark the two versions of NVLINK (1 & 2). Using a workstation (Intel XeonE5-1680 CPU) with two NVIDIA P100 GPUs with NVLINK 1, and another workstation with two V100 GPUs with NVLINK2 (Intel Xeon 4110 CPU), we establish a performance difference between the two interconnects of about 20%, when normalized by the performance of a single GPU, cf. Fig. 7. Not accounting for the hardware differences between the two GPU generations, this suggests that the type of interconnect itself is important in speeding up multi-GPU simulations.

3.2. Importance of spatial sort

To demonstrate the importance of memory locality, we compare the performance between a simulation with and without spatial sort. Due to the central role of data locality in reducing the number of page faults, we expect the spatial sorting to have a large impact on the performance of multi-GPU simulations. Fig. 8 demonstrates that disabling the spatial sort results in a tenfold decrease in performance, showing that sorting the particles is indeed vital. The comparison also shows that domain decomposition simulations on the same GPUs using MPI are less affected by the spatial sort and incur only a roughly 50% decrease in performance when the sorter is disabled. This benchmark does not take into account the improvement of CUDA-aware MPI, and without CUDA-aware MPI, the performance of the unified memory code path is superior to that of the MPI code path. We will, however, further analyze this difference below (Section 3.3).

3.3. Strong scaling of a LJ liquid compared to CUDA-aware MPI

In Fig. 9, we compare performance of a simple LJ liquid between the unified memory and MPI code paths, showing particle time steps per second ($N \times$ TPS) as a function of system size N. Strong scaling is indicated by the increase in absolute performance as a function of the number P of processors at $N = const.$, whereas ideal weak scaling would result in a horizontal line $N \times \text{TPS} = const.$. The performance-relevant code paths for this benchmark are the cell and neighbor lists, and the pair potential [8]. For fair comparison, we run on up to three V100's with NVLINK 2 interconnect between them and with the CPU, on an IBM AC 922 compute node of the Summit supercomputer at Oak Ridge National Laboratory. To accelerate the MPI code path using the cuda-aware IBM Spectrum MPI, which on Summit also takes advantage of NVLINK, we use the compile time option `-DENABLE_MPI_CUDA=ON` for HOOMD-blue, the runtime option `-smpiargs='-gpu'` to the `jsrun` job launcher, and the environment variable `PAMI_DISABLE_IPC=1`. We find that the performance of the unified memory code path with two GPUs is practically identical to that of the MPI code path, and slightly worse than cuda-aware MPI with NVLINK on three GPUs. We explain the performance gap by the fact that the MPI code path on Summit also takes advantage of NVLINK and that the ghost particle regions in this benchmark reduce the demand for data communication in this benchmark for a short-ranged pair potential [8].

3.4. Benchmarks of various soft matter systems

To assess the usefulness of our optimizations across a wider ranger of simulation models, we implemented benchmarks of various soft matter systems that represent different capabilities of the HOOMD-blue code. See Fig. 10 for the normalized performance of the code across all benchmarks, for different unified memory and MPI execution configurations. The benchmarks include a LJ liquid [7,8], a block copolymer micelle [18] in explicit solvent, a crystallizing fluid of rigid proteins in implicit solvent [19], a system of point particles forming a quasicrystal [20], and liquid SPC/E water [21,22]. We chose system sizes large enough to demonstrate the scaling capabilities of the two code paths. As a general observation, benchmarks which only involve pair forces (LJ liquid and quasicrystal), perform better with MPI than with NVLINK, owing to their low compute density. The microsphere benchmark also performs better with MPI, because it involves bonded interactions which the NVLINK code path does not optimize for. On balance, the patchy protein and the SPC/E water benchmarks involve rigid bodies which exhibit poor MPI scaling due to the communication of ghost particle forces, and they demonstrate good scaling with unified memory. In particular, configurations of the SPC/E water benchmark

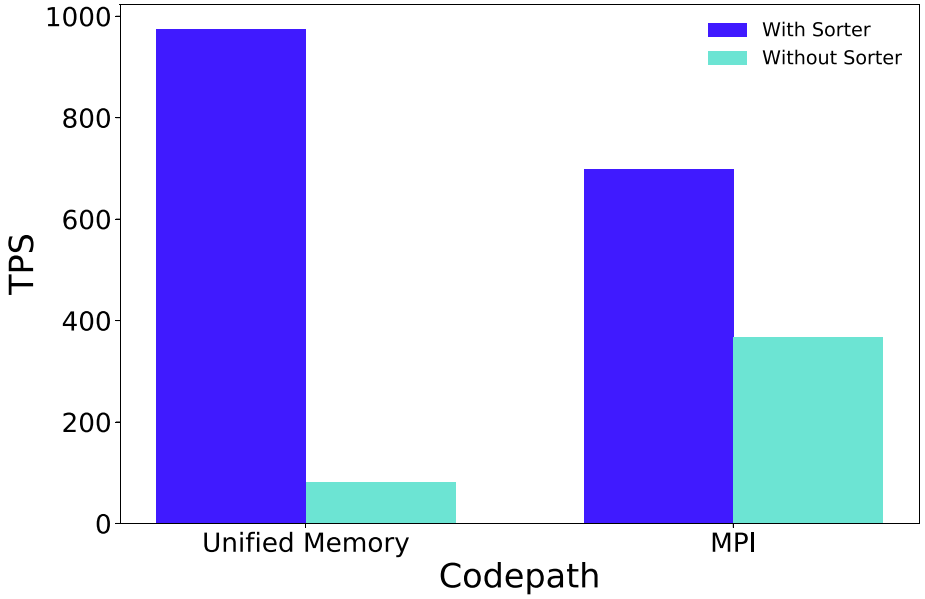

Fig. 8. Performance in number of time steps per second for simulations of the LJ Liquid with $N = 500,000$ particles ($r_{cut} = 3.0\sigma$), using unified memory and MPI on 2× V100s, with and without the sorter.

with non-power of two numbers of ranks do not execute at all with MPI because of the limitations of the distributed FFT algorithm [23] used for PPPM electrostatics, whereas three GPUs using NVLINK realize a 50% performance improvement over single GPU execution. However, absolute performance of this SPC/E water benchmark on one NVIDIA V100 GPU is on the order of 10 ns/day and *not* competitive with biophysical simulation codes such as GROMACS [1,24]. GROMACS uses optimized implementations of the Coulomb interactions, including particle-mesh (PME) only methods and PPPM with analytical differentiation.

4. Conclusions

To take advantage of the NVLINK architecture on the Summit supercomputer and on similar machines, we extend HOOMD-blue 2.5.x to support multi-GPU execution using unified memory. The unified memory code path enables strong scaling of systems with pair potentials, rigid body constraints and electrostatic calculations on multiple GPUs inside the same compute node. Native support for unified memory can be very useful to speed up algorithms that perform poorly in spatial domain decomposition, and in particular those which handle rigid bodies. It coexists with the MPI code path, which scales beyond a single compute node. One may also combine both code paths using multiple GPUs per MPI process, but we did not specifically optimize for this situation and leave this task for the future. Because particles are spatially sorted and evenly split across GPUs, the new code paths lends itself particularly well to the simulation of phenomena that are hard to load-balance using domain decomposition, such as crystal growth from a seed or coexistence of phases with different densities. Only considerably more complex force decomposition schemes, such as those implemented in NAMD [2], and which are not considered here, have

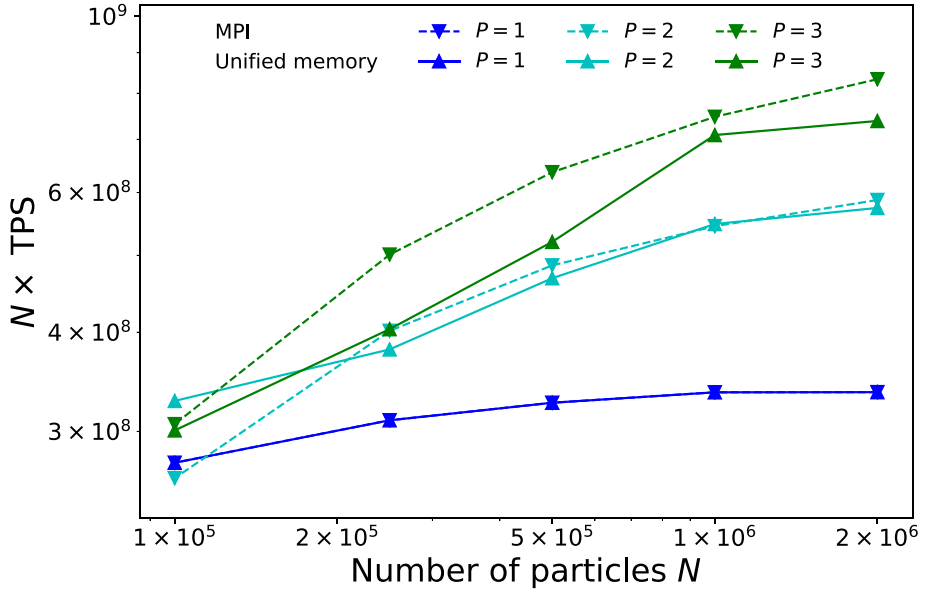

Fig. 9. Scaling of the unified memory (upward-facing triangles) and MPI code paths (downward facing triangles) on a Summit IBM AC922 compute node with number of particles N and number $P = 1, 2, 3$ of GPU, showing the performance in terms of particle time steps per second (TPS × N) for a LJ liquid with $r_{cut} = 3\sigma$ *vs.* number of particles N. Linear scaling of wall clock time with number of particles results in $N \times \text{TPS} = const.$.

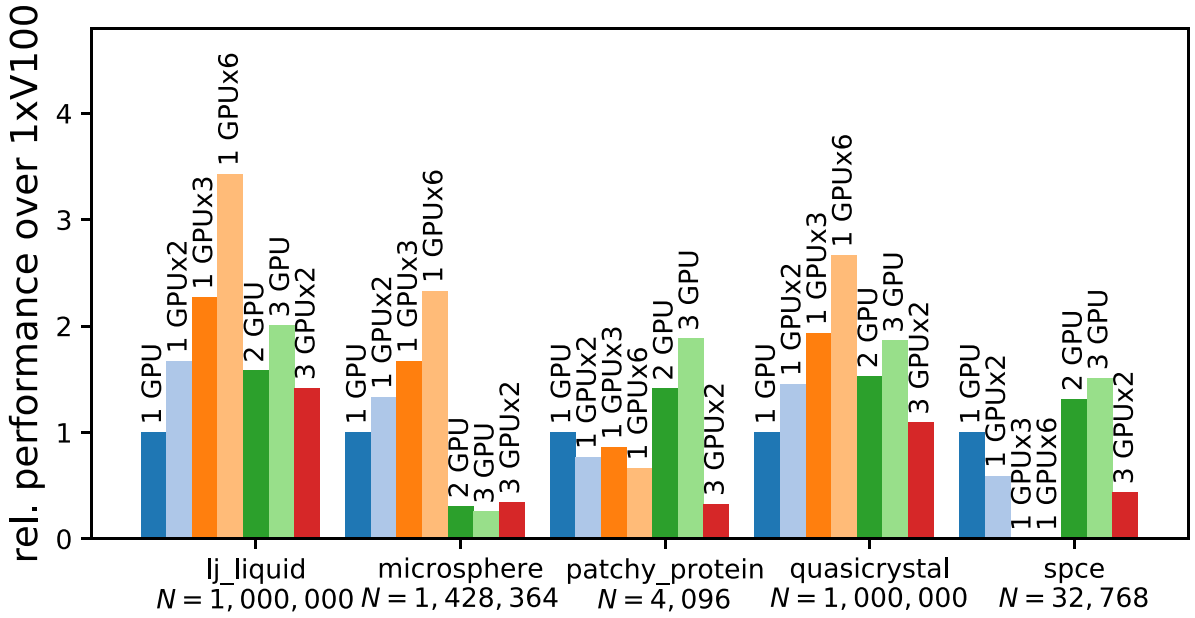

Fig. 10. Benchmarks of representative soft matter systems with N particles on one node of OLCF Summit, showing normalized performance in terms of particle time steps per second relative to one V100 GPU. For every model, the tested execution configurations are (from left to right): single GPU, two MPI ranks with one GPU each, three MPI ranks with one GPU each, two GPUs with unified memory, three GPUs with unified memory, two MPI ranks with three GPUs and unified memory each. Parameter choices, visualizations and job scripts utilizing `signac-flow` [17] are available under http://github.com/glotzerlab/hoomd-benchmarks.

the potential to improve upon our atom decomposition approach. Currently, only a subset of routines in HOOMD-blue supports multi-GPU execution with unified memory. In the future, we will enable more features such as bonded interactions, various updaters, and additional integration methods. HOOMD-blue takes advantage of unified memory capabilities at the lowest level, which sets it apart from other codes that e.g. support NVLINK only when the MPI library supports it [4]. Additionally, the two code paths in our software may provide a good baseline for benchmarks of novel MPI implementations that explicitly support unified memory [25]. Our optimizations should prove useful for accelerating large-scale self-assembly simulations in soft and biological matter and materials science.

5. Author contributions

Jens Glaser: Conceptualization, Investigation, Validation, Writing - original draft, Writing - review & editing. **Peter S. Schwendeman:** Conceptualization, Investigation, Validation, Writing - original draft, Writing - review & editing. **Joshua A. Anderson:** Conceptualization, Investigation, Validation, Writing - original draft, Writing - review & editing. **Sharon C. Glotzer:** Conceptualization, Writing - review & editing, Supervision.

Data availability

All computer code is available open-source from http://github.com/glotzerlab/hoomd-blue. Benchmarks are available under http://github.com/glotzerlab/hoomd-benchmarks.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This material is based upon work supported in part by the U.S. Army Research Laboratory and the U. S. Army Research Office under Grant No. W911NF-18-1-0167 (J.G.) and by the National Science Foundation, Division of Materials Research Award #DMR 1409620 (J.A. and S.C.G). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725: INCITE Project MAT110 “Nucleation and growth of colloidal crystals,” and also an Early Science Project on OLCF’s Summit supercomputer. Additional hardware support by NVIDIA Corp. to the Glotzer Group is gratefully acknowledged.

References

[1] M.J. Abraham, T. Murtola, R. Schulz, S. Pll, J.C. Smith, B. Hess, E. Lindahl, GROMACS: high performance molecular simulations through multi-level parallelism from laptops to supercomputers, SoftwareX 1–2 (2015) 19–25, https://doi.org/10.1016/j.softx.2015.06.001.

[2] B. Acun, D.J. Hardy, L.V. Kale, K. Li, J.C. Phillips, J.E. Stone, Scalable molecular dynamics with NAMD on the summit system, IBM J. Res. Dev. 62 (6) (2018) 4:1–4:9, https://doi.org/10.1147/JRD.2018.2888986.

[3] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys 117 (1) (1995) 1–19, https://doi.org/10.1006/jcph.1995.1039.

[4] Y. Xia, A. Blumers, Z. Li, L. Luo, Y.-H. Tang, J. Kane, J. Goral, H. Huang, M. Deo, M. Andrew, A GPU-accelerated package for simulation of flow in nanoporous source rocks with many-body dissipative particle dynamics, Comput. Phys. Commun. (2019) 106874, , https://doi.org/10.1016/j.cpc.2019.106874.

[5] https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[6] https://github.com/glotzerlab/hoomd-blue.

[7] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (10) (2008) 5342–5359, https://doi.org/10.1016/j.jcp.2008.01.047.

[8] J. Glaser, T.D. Nguyen, J.A. Anderson, P. Lui, F. Spiga, J.A. Millan, D.C. Morse, S.C. Glotzer, Strong scaling of general-purpose molecular dynamics simulations on GPUs, Comput. Phys. Commun. 192 (2015) 97–107, https://doi.org/10.1016/j.cpc.2015.02.028.

[9] IEEE HotChips 28 (2016) http://www.hotchips.org/archives/2010s/hc28/, and IEEE HotChips 29 (2017) https://www.hotchips.org/archives/2010s/hc29/, and https://en.wikichip.org/wiki/nvidia/nvlink.

[10] J.M.A. Grime, G.A. Voth, Highly scalable and memory efficient ultra-coarse-grained molecular dynamics simulations, J. Chem. Theory Comput. 10 (1) (2014) 423–431, https://doi.org/10.1021/ct400727q.

[11] Y.-H. Tang, G.E. Karniadakis, Accelerating dissipative particle dynamics simulations on GPUs: algorithms, numerics and applications, Comput. Phys. Commun. 185 (11) (2014) 2809–2822, https://doi.org/10.1016/j.cpc.2014.06.015.

[12] M.P. Howard, J.A. Anderson, A. Nikoubashman, S.C. Glotzer, A.Z. Panagiotopoulos, Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units, Comput. Phys. Commun. 203 (2016) 45–52,

https://doi.org/10.1016/j.cpc.2016.02.003.
[13] Y.-H. Tang, L. Lu, H. Li, C. Evangelinos, L. Grinberg, V. Sachdeva, G.E. Karniadakis, OpenRBC: a fast simulator of red blood cells at protein resolution, Biophys. J. 112 (10) (2017) 2030–2037, https://doi.org/10.1016/j.bpj.2017.04.020.
[14] M.P. Howard, A. Statt, F. Madutsa, T.M. Truskett, A.Z. Panagiotopoulos, Quantized bounding volume hierarchies for neighbor search in molecular simulations on graphics processing units, Comput. Mater. Sci. 164 (2019) 139–146, https://doi.org/10.1016/j.commatsci.2019.04.004.
[15] D.N. LeBard, B.G. Levine, P. Mertmann, S.A. Barr, A. Jusufi, S. Sanders, M.L. Klein, A.Z. Panagiotopoulos, Self-assembly of coarse-grained ionic surfactants accelerated by graphics processing units, Soft matter 8 (8) (2012) 2385–2397, https://doi.org/10.1039/C1SM06787G.
[16] https://docs.nvidia.com/cuda/cufft/index.html.
[17] C.S. Adorf, P.M. Dodd, V. Ramasubramani, S.C. Glotzer, Simple data and workflow management with the signac framework, Comput. Mater. Sci. 146 (2018) 220–229, https://doi.org/10.1016/j.commatsci.2018.01.035.
[18] Z. Zhang, R.L. Marson, Z. Ge, S.C. Glotzer, P.X. Ma, Simultaneous nano- and microscale control of nanofibrous microspheres self-assembled from star-shaped polymers, Adv. Mater. 27 (26) (2015) 3947–3952, https://doi.org/10.1002/adma.201501329.
[19] J. Glaser, S.C. Glotzer, Looped liquid-liquid coexistence in protein crystallization (2019), https://arxiv.org/abs/1910.06865.
[20] M. Engel, P.F. Damasceno, C.L. Phillips, S.C. Glotzer, Computational self-assembly of a one-component icosahedral quasicrystal, Nat. Mater. 14 (1) (2015) 109–116, https://doi.org/10.1038/nmat4152.
[21] J. Glaser, X. Zha, J.A. Anderson, S.C. Glotzer, A. Travesset, Pressure in rigid body molecular dynamics (2019).
[22] H.J.C. Berendsen, J.R. Grigera, T.P. Straatsma, The missing term in effective pair potentials, J. Phys. Chem. 91 (24) (1987) 6269–6271, https://doi.org/10.1021/j100308a038.
[23] http://github.com/jglaser/dfftlib.
[24] C. Kutzner, S. Pll, M. Fechner, A. Esztermann, B.L. de Groot, H. Grubmüller, More bang for your buck: improved use of GPU nodes for GROMACS 2018, J. Comput. Chem. 40 (27) (2019) 2418–2431, https://doi.org/10.1002/jcc.26011.
[25] K.V. Manian, A.A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, D.K. Panda, Characterizing CUDA unified memory (UM)-aware MPI designs on modern GPU architectures, in: A. Jog, O. Kayiran (Eds.), Proceedings of the 12th Workshop on General Purpose Processing Using GPUs - GPGPU '19, ACM Press, New York, New York, USA, 2019, pp. 43–52, , https://doi.org/10.1145/3300053.3319419.