The Vectorization of the Tersoff Multi-Body Potential: An Exercise in Performance Portability

Markus Höhnerbach
RWTH Aachen University

Ahmed E. Ismail
RWTH Aachen University,
West Virginia University

Paolo Bientinesi
RWTH Aachen University

***Abstract*—Molecular dynamics simulations, an indispensable research tool in computational chemistry and materials science, consume a significant portion of the supercomputing cycles around the world. We focus on multi-body potentials and aim at achieving performance portability. Compared with well-studied pair potentials, multibody potentials deliver increased simulation accuracy but are too complex for effective compiler optimization. Because of this, achieving cross-platform performance remains an open question. By abstracting from target architecture and computing precision, we develop a vectorization scheme applicable to both CPUs and accelerators. We present results for the Tersoff potential within the molecular dynamics code LAMMPS on several architectures, demonstrating efficiency gains not only for computational kernels, but also for large-scale simulations. On a cluster of Intel Xeon Phi's, our optimized solver is between 3 and 5 times faster than the pure MPI reference.**

I. Introduction

Molecular dynamics (MD) simulations trace the trajectory of thousands, millions, and even billions of particles over millions of timesteps, enabling materials science research at the atomistic level. Such simulations are commonly run on highly parallel architectures, and take up a sizeable portion of the computing cycles provided by today's supercomputers. In this paper, we extend the LAMMPS molecular dynamics simulator [1] with a new, optimized and portable implementation of the Tersoff multi-body potential [7].

In many simulations, interactions among particles are typically described by pairwise (particle-to-particle) functions, as dictated by potentials such as the Coulomb or Lennard-Jones potentials. However, several applications in materials science require multi-body potentials [13], in which the force between two particles does not depend solely on their distance, but also on the relative position of surrounding particles. This added degree of freedom in the potential enables more accurate modeling but comes at the cost of additional complexity in its evaluation. Because of this complexity, the optimization of multi-body potential is still largely unexplored.

The Tersoff multi-body potential is especially suited for inorganic carbon-based systems, such as carbon nanotubes (CNTs) and graphene; more complicated systems, such as silicon-carbon composites, can also be handled. Given the incredible popularity of these materials in the research community, the optimization of the Tersoff potential is highly relevant.

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

By design, LAMMPS' main parallelization scheme is an MPI-based domain decomposition; this enables LAMMPS to run on clusters and supercomputers alike, and to tackle large-scale systems. Optionally, LAMMPS can also take advantage of shared-memory parallelism via OpenMP. However, support for vectorization is limited to Intel architectures and to only a few kernels. Given the well-established and well-studied mechanisms for parallelism in MD codes [1], our efforts mostly focus on vectorization, as a further step to fully exploit the computational power of the available hardware. On current architectures, SIMD processing contributes greatly to the system's peak performance, as SIMD units offer hardware manufacturers a way to multiply performance for compute-intense applications. This principle is most pronounced in accelerators such as Intel's Xeon Phis and Nvidia's Teslas.[1]

Several successful open-source MD codes—e.g. Gromacs [9], NAMD, LAMMPS [6], and ls1 mardyn [8]—already exploit vectorization in their most important kernels. However, these vectorized kernels usually do not include multi-body potentials. Implementation methods for the kernels vary between hand-written assembly, intrinsics (compiler-provided functions that closely map to machine instructions), and annotations that guide the compiler's optimization [10]. Furthermore, the majority of these kernels are not readily portable to different architectures, but must instead be optimized for each target architecture. Our objective is an approach sufficiently general to attain high performance on a wide variety of architectures, and that requires changes localized in few, general building blocks. We identified such a solution for the Tersoff potential. When transitioning from one architecture to another, the numerical algorithm stays fixed, and only the interface to the surrounding software components (memory management, preprocessing) needs to be tailored.

We demonstrate our approach for performance portability on a range of architectures, including ARM, Westmere, Sandy Bridge, Ivy Bridge, Haswell and Broadwell, Nvidia's Kepler-generation Teslas, and two generations of Intel Xeon Phi (Knights Corner and Knights Landing). As already mentioned, the numerical algorithm—built on top of platform-specific

[1] "Vectorization" is not a term commonly used in the context of GPU computing; nonetheless we will use it to describe also the kind of data-parallelism that exists within GPU's, otherwise referred to as a "warp".

building blocks—stays the same across all architectures; it is the building blocks that are implemented (once and for all) for each of the instruction sets. Our evaluation ranges from single-threaded to a cluster of nodes containing two Xeon Phi's. Depending on architecture and benchmark, we report speedups ranging from 2x to 8x.

Related Work: LAMMPS [1] is a simulator written in C++ and designed to favor extensibility. It excels at scalability using MPI, and comes with a number of optional packages to run on various platforms and parallel paradigms. LAMMPS supports OpenMP shared-memory programming, GPUs [17], KOKKOS [16], and vector-enabled Intel hardware [18]. Our work is partly based on previous efforts to port MD simulations, and LAMMPS in particular, to the Xeon Phi [18]; specifically, we use the same data and computation management scheme.

The vectorization of pair potential calculations has been addressed in an MD code called miniMD [21] (proxy [19] for the short-ranged portion of LAMMPS). miniMD focuses on various x86 instruction set extensions, including the Xeon Phi's IMCI, and the optimization of cache access patterns.

In addition to LAMMPS, several other MD codes are available, including IMD [12], Gromacs [14], NAMD [26], DL_POLY2 [28], and ls1_mardyn [27]. In IMD, scalar optimizations similar to those we describe here are implemented [11]; however, no vectorization is included. Gromacs provides support for the Xeon Phi [9], and already contains a highly portable scheme for the vectorization of pair potentials [15]. The other codes also contain routines specific to certain platforms such as the Phi [8] or CUDA.

There have been efforts to speed up pair potentials on GPUs [20]. These techniques are similar to those used for vectorization, and the pattern of communication between the GPU and the host is similar to what is needed to achieve high performance on the first generation of the Xeon Phi. In general, GPUs have been used to great effect for speeding up MD simulations [22], [23], [24]. When it comes to multi-body potentials, GPU implementations exist for the EAM [18], Stillinger-Weber [4] and Tersoff [25] potentials. As opposed to our work, the Tersoff implementation for the GPU requires explicit neighbor assignments and thus is only suitable for rigid problems; by contrast, our approach is suitable for general scenarios.

Organization of the paper: Sec. II is a quick general introduction to MD simulations, while a discussion of the Tersoff potential and its computational challenges comes in Sec. III. We introduce our optimizations and vectorization schemes in Sec. IV, and then describe the techniques to achieve portability in Sec. V. In Sec. VI, we provide the results of our multi-platform evaluations, and in Sec. VII we offer conclusions.

Open Source: The associated code is available at [33].

II. Molecular Dynamics Background

A typical MD simulation consists of a set of N atoms (particles) and a sequence of timesteps. At each timestep, the forces for each atom are calculated, and velocity and position are updated accordingly. The forces are modeled by a potential $V(\mathbf{x})$ that depends solely on the positions $\mathbf{x}_i$ of each atom i. V represents the potential energy of the system; the force on an atom i is then the negative derivative of V with respect to the atom's position $\mathbf{x}_i$ [30]:

$$\mathbf{f}_i = -\nabla_{\mathbf{x}_i} V(\mathbf{x}). \tag{1}$$

Most non-bonded potentials, such as Lennard-Jones and Coulomb, are pair potentials: As such, they can be expressed as a double sum over the atoms, where the additive term depends only on the relative distance between the atoms:

$$V = \sum_i \sum_j \phi(r_{ij}). \tag{2}$$

In practice, Eq. 2 is computed by limiting the inner summation (j) to the set of atoms $\mathcal{N}_i$—known as the "neighbor list"—that are within a certain distance r_C from atom i:

$$V = \sum_i \sum_{j \in \mathcal{N}_i} \phi(r_{ij}), \tag{3}$$

$$\mathcal{N}_i = \{j : r_{ij} \leq r_C, j \neq i\}. \tag{4}$$

With this second formulation, the complexity for the computation of V decreases from $O(N^2)$ to $O(N)$, thus making large-scale simulations feasible. This simplification is based on the assumption that ϕ goes to zero as the distance r increases. The assumption is valid for short-ranged potentials of the form r^{-p}, $p \geq 3$; long-ranged potentials, with $p < 3$, must be augmented using long-ranged calculation schemes.

Algorithm 1 illustrates how, based on a potential V as given in Eq. 3, the potential energy V and the forces F_i on each atom i can be evaluated.

for i **do**
  for $j \in \mathcal{N}_i$ **do**
    $V \leftarrow V + \phi(r_{ij})$;
    $F_i \leftarrow F_i - \partial_{x_i} \phi(r_{ij})$;
    $F_j \leftarrow F_j - \partial_{x_j} \phi(r_{ij})$;

Algorithm 1: Calculation of potential and forces due to a pair potential.

III. The Tersoff potential

As opposed to pair potentials, multi-body potentials deviate from the form of Eq. 2. In particular, ϕ is replaced by a term that depends on more than the distance r_{ij}. Instead, it might depend also on the distance of other atoms close to atom i or j, and on the angle between i, j and the surrounding atoms.

Omitting trivial definitions, the Tersoff potential [7] is defined as follows:

$$V = \sum_i \sum_{j \in \mathcal{N}_i} \overbrace{f_C(r_{ij})\left[f_R(r_{ij}) + b_{ij} f_A(r_{ij})\right]}^{V(i,j,\zeta_{ij})}, \tag{5}$$

$$b_{ij} = (1 + \beta^\eta \zeta_{ij}^\eta)^{-\frac{1}{2\eta}}, \eta \in \mathbb{R}, \tag{6}$$

$$\zeta_{ij} = \sum_{k \in \mathcal{N}_i \setminus \{j\}} \underbrace{f_C(r_{ik}) g(\theta_{ijk}) \exp(\lambda_3 (r_{ij} - r_{ik}))}_{\zeta(i,j,k)}. \tag{7}$$

Eq. 5 indicates that two forces act between each pair of atoms (i, j): an attractive force modeled by f_A, and a repulsive force modeled by f_R. Both depend only on the distance r_{ij} between atom i and atom j. The bond-order factor b_{ij}, defined by Eq. 6, however, is a scalar that depends on all the other atoms k in the neighbor list of atom i, by means of their distance r_{ik}, and angle θ_{ijk} via ζ_{ij} (from Eq. 7). Since the contribution of the (i, j) pair depends on other atoms k, Tersoff is a multi-body potential. f_C is a cutoff function, smoothly transitioning from 1 to 0; g describes the influence of the angle on the potential; all other symbols in Eq. 5–7 are parameters that were empirically determined by fitting to known properties of the modelled material. Although these parameters mean that many lookups are necessary, the functions within the potential $(f_R, f_A, f_C, g, \exp)$ are expensive to compute, thus making the Tersoff potential a good target for vectorization.

Eq. 5–7 give rise to a triple summation; this is mirrored by the triple loop structure of Algorithm 2, which describes the implementation found in LAMMPS in terms of the functions $V(i, j, \zeta)$ and $\zeta(i, j, k)$, and calculates forces F and potential energy E: For all (i, j) pairs of atoms, first ζ_{ij} is accumulated, and then the forces are updated in two stages, first with the contribution of the $V(i, j, \zeta_{ij})$ term, and finally with the contributions of the $\zeta(i, j, k)$ terms.

for i **do**
  for $j \in \mathcal{N}_i$ **do**
    $\zeta_{ij} \leftarrow 0$;
    for $k \in \mathcal{N}_i \setminus \{j\}$ **do**
      $\zeta_{ij} \leftarrow \zeta_{ij} + \zeta(i, j, k)$;
    $E \leftarrow E + V(i, j, \zeta_{ij})$;
    $F_i \leftarrow F_i - \partial_{x_i} V(i, j, \zeta_{ij})$;
    $F_j \leftarrow F_j - \partial_{x_j} V(i, j, \zeta_{ij})$;
    $\delta\zeta \leftarrow \partial_\zeta V(i, j, \zeta_{ij})$;
    for $k \in \mathcal{N}_i \setminus \{j\}$ **do**
      $F_i \leftarrow F_i - \delta\zeta \cdot \partial_{x_i} \zeta(i, j, k)$;
      $F_j \leftarrow F_j - \delta\zeta \cdot \partial_{x_j} \zeta(i, j, k)$;
      $F_k \leftarrow F_k - \delta\zeta \cdot \partial_{x_k} \zeta(i, j, k)$;

Algorithm 2: Calculation of potential energy and forces due to the Tersoff potential.

For the following discussions, it is important to keep in mind the loop structure of Algorithm 2: It consists of an outer loop over all atoms (denoted by the capital letter I), an inner loop over a neighbor list (denoted by the capital letter J), and inside the latter, two more loops over the same neighbor list (denoted by the capital letter K).

As opposed to pair potentials, multi-body potentials are used with extremely short neighbor lists $\mathcal{N}_i$. In a representative simulation run, $\mathcal{N}_i$ rarely contains more than four atoms. Assuming that the size of $\mathcal{N}_i$ is n, the algorithm accesses the atoms in $\mathcal{N}_i$ a total of $2n^2$ times. In practice, constructing the neighbor list every timestep would be too expensive. Instead, the cutoff radius r_C is extended by a so-called "skin distance". Because atoms move a limited distance per timestep, one can guarantee that no atom enters or exits the cutoff region for some number of timesteps by tracking all atoms also within the skin distance. Consequently, the neighbor list also only needs to be rebuilt after this many steps. We denote the extended neighbor list by $\mathcal{S}_i$ instead of $\mathcal{N}_i$. Given that the Tersoff potential incorporates a cutoff function f_C, the mathematical formulation is equivalent no matter if iterating through $\mathcal{N}_i$ or $\mathcal{S}_i$. Nevertheless, as little computation as possible should be performed on skin atoms. Efficiently excluding skin atoms is one of the major challenges for vectorization.

IV. Optimizations

This section discusses the various optimizations that we applied to the algorithm described in the previous section. Some are inherited from the libraries that we integrate with (USER-INTEL and KOKKOS), such as optimized neighbor list build, time integration, and data access (e.g. alignment, packing, atomics). These optimizations are generic in that they apply to any potential that uses that particular package.

We devised several other optimizations which are instead specific to the Tersoff potential; they are detailed here.

1) Scalar optimizations. These improvements are useful whether one vectorizes or not. We improve parameter lookup by reducing indirection, and eliminate redundant calculation by inlining function calls. This group of optimizations also includes the method described in Sec. IV-A, which aims to remove redundant calculations of the ζ term.
2) Vectorization. We discuss details of our vectorization strategy in Sec. IV-B, where we present different schemes, and describe their effectiveness for various vector lengths.
3) Optimizations that aid vectorization. As described in Sec. IV-C and IV-D, we aim at reduce the waste of computing resources on skin atoms.

A. Pre-calculating Derivatives

The first optimization we discuss consists in restructuring the algorithm so that ζ and its derivatives are computed only once, in the first loop, and the product with $\delta\zeta$ is only performed in the second loop. Since ζ and its derivatives—naturally—share terms, this modification has a measurable impact on performance. Indeed, ζ can be calculated from intermediate results of the derivative evaluation at the cost of just one additional multiplication.

However, the computation of the derivatives in the first loop over k requires additional storage: While the derivatives with respect to the positions of atoms i and j can be accumulated, the derivatives for k have to be stored separately, as they belong to different k's. In our implementation, this list can contain up to a specified number k_{max} of elements. Should more than k_{max} elements be necessary, the algorithm falls back to the original scheme, thus maintaining complete generality. Algorithm 3 implements this idea.

```
...;
ζ_ij ← 0; ∂_iζ ← 0; ∂_jζ ← 0; ∀k.∂_kζ ← 0;
K ← {};
for k ∈ S_i \ {j} ∧ |K| < k_max do
    if r_ik > r_C then
        continue;
    ζ_ij ← ζ_ij + ζ(i, j, k);
    ∂_iζ ← ∂_iζ + ∂_{x_i}ζ(i, j, k);
    ∂_jζ ← ∂_jζ + ∂_{x_j}ζ(i, j, k);
    ∂_kζ ← ∂_kζ + ∂_{x_k}ζ(i, j, k);
    K ← K ∪ {k};
for k ∈ S_i \ {j} \ K do
    ...;
...;
F_i ← F_i − δζ · ∂_iζ;
F_j ← F_j − δζ · ∂_jζ;
for k ∈ K do
    F_k ← F_k − δζ · ∂_kζ;
for k ∈ S_i \ {j} \ K do
    ...;
...;
```

Algorithm 3: Calculation of Tersoff potential and forces, taking into account skin atoms, pre-calculation derivatives, reverting to original approach after k_{max} elements are stored.

B. Vectorization Choices

In Alg. 3 (and Alg. 2) the iteration space ultimately is three-dimensional, corresponding to the three nested loops I, J and K. This space needs to be mapped onto the available execution schemes, that is, data-parallelism, parallel execution, and sequential execution. We propose three different mappings that are useful in different scenarios. For all of them, it is convenient to map the K dimension onto sequential execution, because values calculated in the K loop, the ζ's, are used in the surrounding J loop, and data computed in the J loop, i.e. $\delta\zeta$, is then used within the second K loop.

Therefore, the problem boils down to mapping the I and J dimensions onto a combination of parallel execution, data-parallelism, and if necessary, sequential execution. We assume that the amount of available data-parallelism is unlimited. In practice, the program sequentially executes chunks, and each chunk takes advantage of data-parallelism.

As shown in Fig. 1, to perform the mapping sensibly we propose three schemes:

(1a) I is mapped to parallel execution, and J to data-parallelism.

(1b) I is mapped to parallel execution, and I and J to data-parallelism.

(1c) I is mapped to parallel execution and data-parallelism, and J to sequential execution.

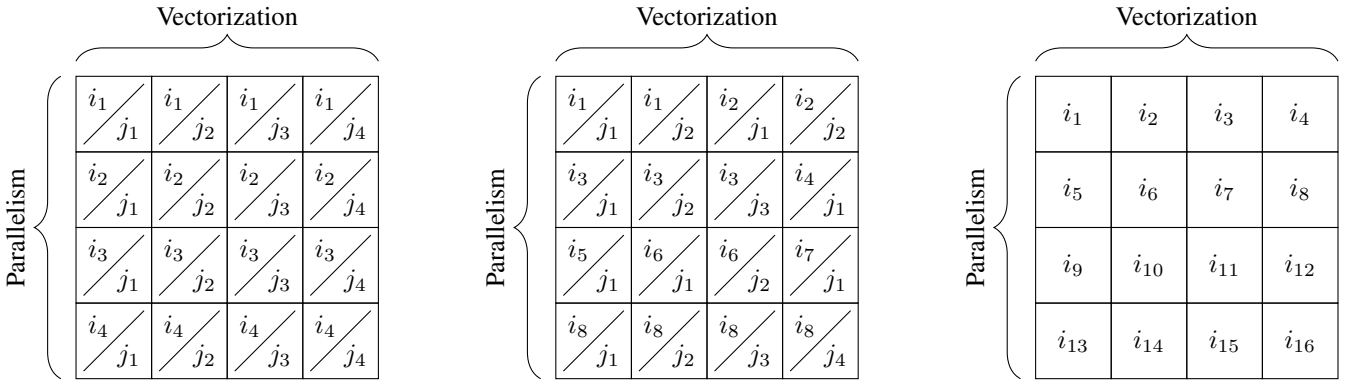
Scheme (1a) is natural for vector architectures with short vectors, such as single precision SSE and double precision AVX. In these, it makes sense to map the j iterations directly to vector lanes, as there is a good match among them: 3-4 iterations to 4 vector lanes. This scheme is most commonly used to vectorize pair potentials. The advantage of this approach is that the atom i is constant across all lanes. While performing iterations in k through the neighbor list of atom i, the same neighbor list is traversed across all lanes, leading to an efficient vectorization. However, with long vectors and short neighbor lists, this approach is destined to fail on accelerators and CPUs with long vectors.

Scheme (1b) is best suited for vector lenghts (8 or 16) that exceed the iteration count of j, as it handles the shortcomings of (1b). With this approach, iterations of i and j are fused, and the fused loop is used for data-parallelism. Given that i contains many iterations (as many as atoms in the system), this scheme achieves an unlimited potential for data-parallelism. However, in contrast to (1a), atom i is not constant across all lanes; consequently, the innermost loops iterates over the neighbor lists of different i, leading to a more involved iteration scheme. Even if this iteration is efficient, it can not attain the same performance of an iteration scheme where all vector lanes iterate over the same neighbor list. The vectorization of the i loop invalidated a number of assumptions of the algorithm: i and k are always identical across all lanes, while j's, coming from the same neighbor list, are always distinct. Without these assumptions, special care has to be taken when accumulating the forces to avoid conflicts. For the program to be correct under all circumstances, the updates have to be serialized. In the future, AVX-512 conflict detection support may change this.

Whether the disadvantages of Scheme (1b) outweigh its advantages or not is primarily a question of amortization. The answer depends on the used floating point data type, the vector length, and the features of the underlying instruction set.

Scheme (1c) is the natural model for the GPU, where data-parallelism and parallel execution are blurred together. An iteration i is assigned to each thread, and the thread sequentially works through the j iterations.

To implement these schemes, the algorithms are split into two components: a "computational" one, and a "filter". The computational component carries out the numerical calculations, including the innermost loop and the updates to force and energy; the input to this component are pairs of i and j for which the force and energy calculations are to be carried out. Given that the majority of the runtime is spent in computation, this is the part of the algorithm that has to be vectorized. The filter component is instead responsible to feed work to the computational one; its duty is to determine which pairs to



(a) Mapping I to parallelism, and J to vectorization.

(b) Mapping fused I and J to both parallelism and vectorization.

(c) Mapping I to both parallelism and vectorization.

Fig. 1: Mapping of atoms (I) and elements of neighbor lists (J) to vector units and parallelism.

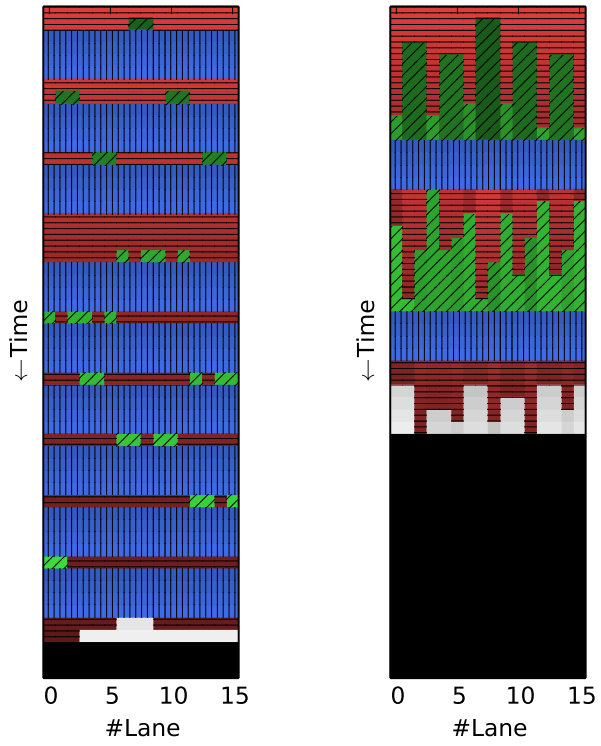


Fig. 2: Mask status during iteration of the K loop: Green (diagonally hatched) is ready-to-compute, red (horizontally hatched) is not ready-to-compute, and blue (vertically hatched) is actual calculation. The left is an unoptimized variant, where calculation takes place as soon as one lane is ready-to-compute, while on the right, the calculation is delayed until all lanes are ready-to-compute.

pass. To this end, the data is filtered to make sure that work is assigned to as many vector lanes as possible before entering the vectorized part. This means that interactions outside the cutoff region never even reach the computational component.

C. Avoiding Masking or Divergence

Section IV-B just described how we avoid calculation for skin atoms in the J loop. The remaining issue is how we skip them in the K loop. The same argument, that resources must not be wasted on calculations that do not contribute to the final result, applies here, too. As such, as many vector lanes as possible need to be active before entering numerical kernels such as those computing $\zeta(i,j)$ and $V(i,j,\zeta)$. These kernels are almost entirely straight-line floating-point intense code, with intermittent lookups for potential parameters.

This optimization is most important for Scheme (1b) and (1c), as they traverse multiple neighbor lists in parallel. As such, no guarantee can be made that interacting atoms have the same position in all neighbor lists. This leads to sparse masks for the compute kernel: For example, in a typical simulation that uses a vector length of sixteen, no more than four lanes will be active at a time. On GPUs, this effect is even worse, where 95% of the threads in a warp might be inactive.

Our optimization extends Scheme (1b) and (1c) to fast forward through the K loop, until as many lanes as possible can take part in calculating a numerical kernel. Instead of traversing the neighbor list at equal speed for all vector lanes, we manipulate the iteration index independently in the various lanes. Fig. 2 visualizes the way this modification affects the behaviour of the algorithm. In that figure, the shade of the particular color roughly corresponds to that lane's progress through the loop. Notice that on the left, calculation (blue) takes place as soon as at least one lane requires that calculation (green). Instead, on the right, a lane that is ready to compute (green), idles (does not change its shade), while the other lanes make further progress (going through shades of red) in search of the iteration where they become ready (green). In our implementation, the calculation (blue) only takes place if all lanes are ready (green). Effectively, we "fast-forward" in each lane, until all of them have are ready to compute.

D. Filtering the Neighbor List

To implement the idea from Sec. IV-C, a lot of masking is necessary, because the subset of lanes that have to progress when fast-forwarding changes every time. On platforms where masking has non-trivial overhead, performance can be further optimized.

Observe that in Fig. 2, lanes "spin" until computation is available. We can reduce the amount of spinning by filtering the neighbor list in the scalar segment of the program. To ensure correctness, the filtering is based on the maximum cutoff of all atom types in the system. This means that atoms that physically play a role can not be accidentally excluded from the calculation. Filtering with any other cutoff might lead to incorrect results in systems with multiple kinds of atoms, if the cutoff prescribed between any two atom kinds differs.

Filtering the neighbor list is especially effective with AVX, where the double precision implementation uses the mapping (1a), whereas the single precision variant uses (1b). Without this change, the overhead to spin is too big to lead to speedups with respect to the double precision version. With this change, most time is again spent in the numerical part of the algorithm.

V. Implementation

So far we have intentionally kept the description of the algorithms generic. We now cover the implementation of the schemes and optimizations from the previous section and their integration into LAMMPS. The main software engineering challenge was to make the implementation maintainable, while achieving portable performance. To this end, OpenMP 4.5's SIMD extensions would be the most appealing solution, but right now lack both compiler support and a number of critical features that are required for our implementation. We resorted to implementing these features ourself as modular and portable building blocks. In the following, we first introduce these building blocks, and then focus on a platform independent implementation. Finally, we characterize the different execution modes supported by our code.

A. Required Building Blocks

We identified four groups of building blocks necessary for a portable implementation.

(1) Vector-wide conditionals. These conditionals check if a condition is true across all vector lanes; since either all or no lanes enter these conditionals, excessive masking is prevented.

(2) Reduction operations. These are useful when all lanes accumulate to the same (uniform-across-lanes) memory location, so that reductions can be performed in-register, and only accumulated values are written to memory. This behaviour can not be achieved with OpenMP's reduction clause, since it only supports reductions in which the target memory location is known a priori, which is not the case for our code.

(3) Conflict write handling. This feature allows vector code to write to nondistinct memory locations (see the discussion in the previous bullet). In the vectorization of MD codes, all lanes are often guaranteed to write to distinct memory locations (since the atoms in a neighbor list are all distinct), which is the assumption that compilers typically make when performing user-specified vectorization. Unfortunately, this guarantee does not hold for Scheme (1b). By serializing the accesses, the `ordered simd` clause of the OpenMP 4.5 standard provides a solution to this issue; however, at time of writing, this directive is not yet supported by any major compiler. It is also questionable whether this approach will be "future proof" or not, as a conflict detection mechanism such as that in the AVX-512 extensions might make serialization unnecessary.

(4) Adjacent gather optimizations. These provide improved performance on systems that do not support a native gather instruction or where such an instruction has a high latency. An adjacent-gather operation is a sequence of gather operations that access adjacent memory locations. Instead of using gather instructions or gather emulations here, it is possible to load continuously from memory into registers, and then permute the data in-register. This operation can lead to significant performance improvements in our code, because adjacent-gather operations are necessary to load the parameters of our potential; it is also important for backwards-compatibility reasons, because old systems lack efficient native gather operations.

B. Vector Abstraction

Since our objective is to integrate with the LAMMPS MD simulator, support for different instruction sets and for different floating point precisions is necessary. It is crucial to support CPU instruction sets to balance the load between host and accelerator. Additionally, such an abstraction enables us to evaluate the influence of vector lengths and instruction set features on performance. Considering all combinations of instruction sets, data types and vectorization variants, it becomes clear that it is infeasible to implement everything with intrinsics.

We created a single algorithm, and paired it with a vectorization back-end. As a consequence, instead of coding the Tersoff potential's algorithm $n \cdot m$ times (n architectures and m precision modes), we only had to implement the building blocks for the vectorization back-end. Some of these building blocks provide the features described in Sec. V-A, while others provide one-to-one mappings to intrinsics, mostly unary and binary operators.

The vectorization back-end uses C++ templates specialized for each targeted architecture. We developed back-ends for single, double and mixed precision using a variety of instruction set extensions: Scalar, SSE4.2, AVX, AVX2, IMCI (the Xeon Phi Knights Corner instruction set), as well as experimental support for AVX-512, Cilk array notation and CUDA.

The library is designed to be easily extended to new architectures. Even though the tuning might take some time, it is simplified by the fact that a number of building blocks, such as wide adjacent-gather operations, can be optimized in one go. Contrary to most other vector libraries, which allow the programmer to pick a vector length that may be emulated internally, our library only allows for algorithms that are oblivious of the used vector length.

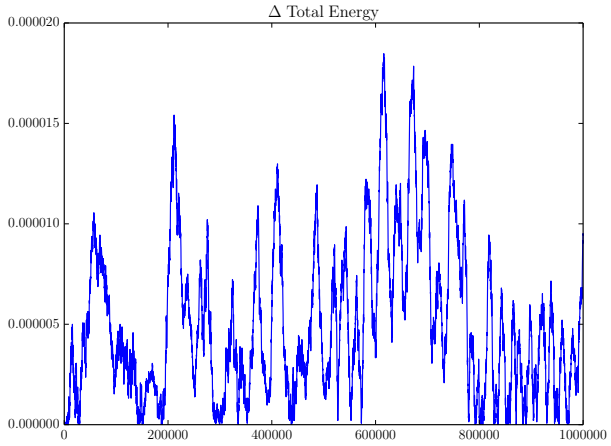


Fig. 3: Validation of the single precision solver: relative difference between the single and double precision solvers for a system of 32 000 atoms for 10^6 timesteps.

C. Integration Points

We use vanilla LAMMPS' MPI-based domain decomposition scheme and build upon optional packages that offer various optimizations and capabilities. Specifically, all our x86 and ARM implementations use the USER-INTEL [18] package, which collects optimizations for Intel hardware, to manage offloading to the Xeon Phi, data-packing, alignment and simulation orchestration. For the GPU implementation, the same role is fulfilled by the KOKKOS package [16]. Since KOKKOS abstracts the data layout of the variables used in a simulation (e.g. position, velocity, mass, force), the code needs to be changed wherever data is accessed from memory. We also need to change the routine that feeds our algorithm, to conform with the model of parallelism that is used by KOKKOS. As a consequence, comparisons between x86 and ARM, and between x86 and the GPU implementation cannot reasonably be drawn. Furthermore, the KOKKOS package is still under development, while the USER-INTEL package is more mature; as such, we believe that the GPU results are likely to have room for improvement.

D. Accuracy

In addition to a double precision code, which is the default in LAMMPS, we created versions that compute the Tersoff potential in single and mixed precision. To validate these reduced precision implementations, we measured the energy in a long-running simulation, a tactic commonly used in GPU-acceleration of MD simulations to justify reduced precision [31], [32], [22]. As Fig. 3 illustrates, for a system of 32 000 atoms, the deviation is within 0.002% of the reference.

In part, this effect can be explained by the short neighbor lists which are characteristic of the Tersoff potential: Since only few different atoms interact with any given atom, and only these accumulate their contributions to the force, there is little chance for round-off error to accumulate.

E. Execution Modes for Measurement

In the following section, we present performance results for several hardware platforms, and four different codes: Ref, Opt-D, Opt-S, Opt-M.

Ref: The reference for our optimization and testing is the implementation shipped with LAMMPS itself, which performs all the calculations in double precision.

Opt-D: The most accurate version of our code, performing calculations in double precision. It includes the optimizations due to both scalar improvements and vectorization.

Opt-S: The least accurate version of our code, implemented entirely in single precision. Like Opt-D, it includes both scalar improvements, and takes advantage of vectorization. The vector length typically is twice that of Opt-D. Referring to Sec. V-D, the accuracy of the single precision solver is perfectly in line with that offered by *Opt-D* or *Ref*.

Opt-M: We also provide a mixed precision version of our code that performs all calculations, except for accumulations, in single precision. It is the default mode for the USER-INTEL package code, as it offers a compromise between speed and accuracy. From an software engineering perspective, the mixed precision version costs very little, as its can leverage the existing single and double precision codes; indeed, our vector library performs this step (from single and double implementation to mixed implementation) automatically.

In addition to these four modes, the algorithms are run on a single thread (*1T*) or on an entire node (*1N*). The single-threaded run gives the most pure representation of the speedup obtained by our optimizations; the results for an entire node (*1N*) and a cluster instead give a realistic assessment of the expected speedup in real-world applications. Such parallel runs use MPI, as provided by LAMMPS itself; as a consequence, the parallelization scheme used in *Ref* and *Opt* is the same.

VI. Results

In this section, we validate the effectiveness of our optimizations for the Tersoff potential by presenting experimental results on a variety of hardware platforms, ranging from a low-power ARM to the second generation Xeon Phi. As a test case we use a standard benchmark, distributed directly with LAMMPS' source code, for the simulation of Silicon atoms. The wide availability of said benchmark, and its unpartial nature can hopefully add to the reproducibility of this study. Since the atoms in the benchmark are laid out in a regular lattice so that each of them has exactly four nearest neighbors, this test case captures well the scenario of small neighbor lists discussed in Sec. III and Sec. IV. Moreover, since the benchmark sets the skin distance to 1 Å, Optimizations C and D (Sec. IV) excluding skin atoms apply here.

We also experimented with other test-sets, derived from actual production runs, and based on CNT simulations (three neighbors per atom on average); in all cases, the results are aligned with those presented hereafter. While not showcased, our code also supports multi species systems. Since no optimization specifically designed for the single species case was

incorporated, we do not expect the performance for multi-species systems to be severely affected.

Instead of isolating the impact of each optimization on each architecture, our results focus on overall performance. In particular, this means that most of the measurements refer to fully parallel runs, using MPI+OpenMP. Such a regime is quite unfavourable to us, as the impact of our optimizations is reduced by the efficiency losses due to parallelism. This perspective should be valuable since it paints a realistic picture of the speedups for actual simulations, as opposed to just measuring single-threaded performance.

We start by presenting single-threaded and single-node results for the CPUs (and the respective instruction sets) listed in Table I; we continue with measurements for two GPUs (Table II), and conclude with results for the Xeon Phi (Table III), in number of configurations. We also present data for a cluster of nodes, to demonstrate the degree to which the scalar and vector improvements lead to performance at scale. Since we do not provide a breakdown of the improvements, we mention here that the speedups due to scalar optimizations are roughly between 1.2x and 1.5x, depending on the architecture.

TABLE I: Hardware used for CPU benchmarks.

Name	Processor	Cores	Vector ISA
ARM	ARM Cortex-A15	2×4[2]	NEON
WM	Intel Xeon X5675	2×6	SSE4.2
SB	Intel Xeon E5-2450	2×8	AVX
HW	Intel Xeon E5-2680v3	2×12	AVX2
HW2	Intel Xeon E5-2697v3	2×14	AVX2
BW	Intel Xeon E5-2697v4	2×18	AVX2

Timing Methodology: LAMMPS uses MPI timers to reports the time spent in various stages throughout a simulation. Our primary performance metric is "simulated time over run time", in "nanoseconds per second" (equivalent to "iterations per seconds"). All the timings exclude initialization and cleanup times, since in any real-world simulation these stages are entirely negligible. However, the timings do include all other stages, such as communication, data transfer, neighbor list construction, and time integration.

A. CPUs

Fig. 4 shows single-threaded performance for all the different execution modes described in Sec. V-E.

On ARM, the speedups for Opt-D[3] and Opt-S over Ref are 2.4 and 6.4, respectively. These improvements can be partly attributed to alignment, to the use of lower accuracy math functions, and to our scalar optimizations; still, these influences do not explain the 2.7x improvement from Opt-S to Opt-D, which is mostly due to vectorization.

[2]little.BIG: Two different CPUs on a single chip, one powerful Cortex-A15, and a less powerful Cortex-A7. We will measure performance using only the Cortex-A15.

[3]Since NEON does not support vectorized double precision, Opt-D corresponds to optimized, but not vectorized, scalar version of the code. Due to lack of double precision vector instructions, we did not implement a mixed precision mode either.

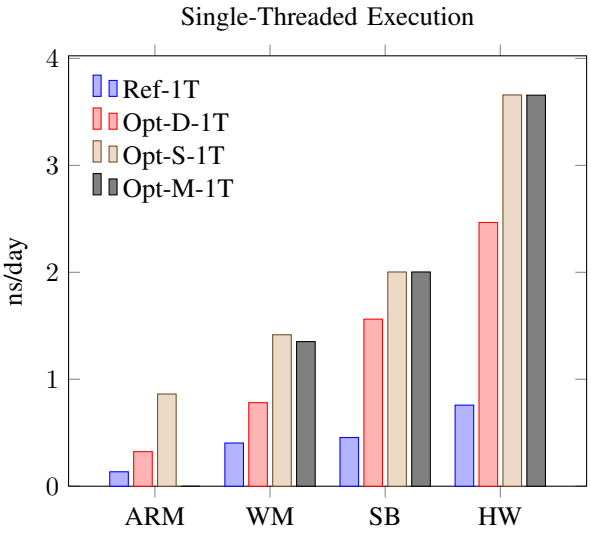


Fig. 4: Evaluation of performance portability across CPUs; single-threaded execution. 32 000 atoms.

On WM, we observe speedups of 1.9 between Ref and Opt-D,[4] and 3.5 between Ref and Opt-S. SSE4.2 supports vectorized integer instructions, so performance is limited by vector length and not features of the instruction set.

On SB (and HW), Opt-D is vectorized.[5] This is readily apparent by the more than threefold speedup going from Ref to Opt-D on SB. Again, a portion of that speedup is due to scalar improvements, but it is clear that vectorization plays a significant role. Since AVX lacks the integer instructions necessary to efficiently implement the (1b) scheme from Section IV-B, the Opt-S/M measurements perform below expectations.

In contrast to SB, the HW systems utilizes the AVX2 instruction set and attains a factor of 4.8 between Opt-S and Ref. AVX2 adds integer and gather instructions, which our code takes advantage of.

One might argue that the benchmark is especially favorable for certain architecures, since their vector length matches the number of nearest neighbors. In that respect, the benchmark can be considered as a best-case scenario for such architectures. However, the results also demonstrate high performance on architectures with much longer vectors. Such an example of performance portability is indeed a main point of this paper.

In Fig. 5, we give results for the execution on entire nodes. As these measurements use a bigger, more expensive simulation run, we only report on mixed precision measurements—mixed precision being the setting most likely to be used in practice. The observed relative speedup between Ref and Opt-M ranges between 2.7 and 5.0; these improvements are lower than what Fig. 4 would suggest, because the communication layer takes up between 5% and 30% of the execution time.

[4]The SSE4.2 double precision results use the scalar back-end (since with a vector length of two, vectorization does not yield speedups).

[5]For AVX/AVX2 in double precision (i.e. SB/HW/BW), and SSE4.2 in single precision, we use scheme (1a) from Section IV-B, whereas all longer vector lengths use the fused loop scheme (1b).

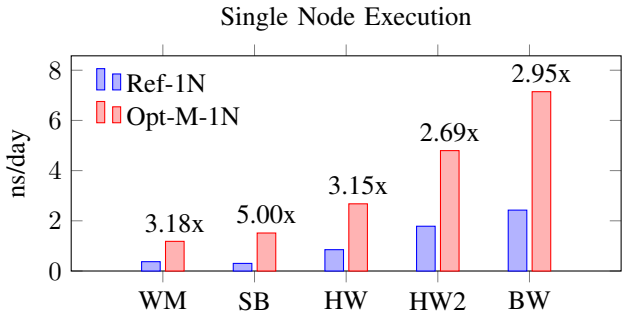


Fig. 5: Evaluation of performance portability across CPUs; single-node execution. 512 000 atoms.

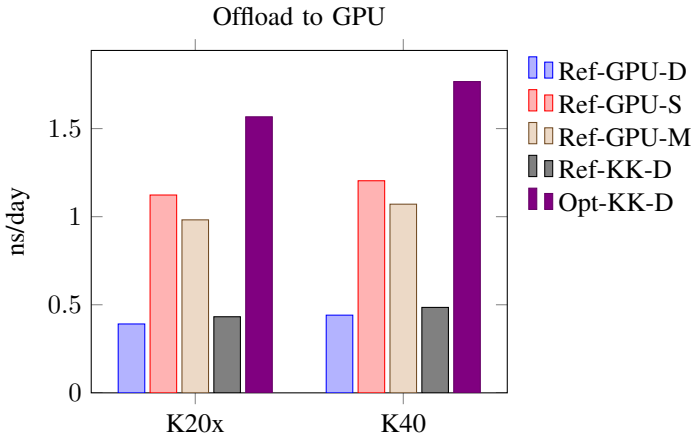


Fig. 6: Evaluation of performance portability for Nvidia GPUs. 256 000 atoms.

B. GPUs

TABLE II: Hardware used for GPU benchmarks.

Name	CPU	Cores	ISA	Accelerator
K20X	Intel Xeon E5-2650	2 × 8	AVX	Nvidia Tesla K20x
K40	Intel Xeon E5-2650	2 × 8	AVX	Nvidia Tesla K40

We now present results for two Kepler generation GPUs (listed in Table II). As reference measurements, we use the LAMMPS GPU package in double (Ref-GPU-D), single (Ref-GPU-S) and mixed (Ref-GPU-M) precision, and the KOKKOS package in double precision (Ref-KK-D). Our optimized implementation Opt-KK-D is based on KOKKOS, and runs in double precision.[6]

Fig. 6 reports performance measurements when utilizing one entire GPU. These measurements include very limited host involvement to orchestrate the offloading process: no force calculation, no communication, and no domain decomposition takes place on the host. This mode of running is probably most comparable to an "entire host" run from the previous section, as it uses the full GPU. Even though the GPU programming paradigm is substantially different than the CPU one, the speedup we achieve, roughly three, is similar. While a single precision version of our optimized GPU code (Opt-KK-S) does not exist yet, the reported measurements suggest that even higher performance (likely around 5ns/s) should be obtained.

In order to get a better idea of the benefits of the optimizations we performed, we look at Ref-KK-D and Opt-KK-D, and only compare the time for the routines affected by our optimizations (the others, such as communication and neighbor list builds, are identical for both versions). This isolated speedup is even higher, at approximately 5x.

C. Intel Xeon Phi

We conclude with a discussion of the portability of our optimizations on two generations of Intel Xeon Phi accelerators—"Knights Corner" (KNC), and "Knights Landing" (KNL)—scaling from a single accelerator to a cluster.

[6]Like CUDA, Opt-KK-D uses a scalar back-end, with an important distinction: the implementation of the vector-wide conditional operation uses a warp vote to determine if the condition is true for all threads in the warp. This is semantically equivalent to the implementation for the vector instruction sets.

Fig. 7 measures the impact of our optimizations while using all cores of a Xeon Phi accelerator. For a fair comparison, the benchmark is run on the device, without any involvement of the host in the calculation. On both platforms, the speedup of Opt-M with respect to Ref is roughly 5x. Single-threaded measurements (not reported) indicate that the "pure" speedup in the kernel is even higher, at approximately 9x.

With a relative performance improvement of about 3x, the difference between KNC and KNL is in line with our expectations; in fact, the theoretical peak performance also roughly tripled, along with the bandwidth, which roughly doubled. We point out that no optimization specific to KNL was incorporated in our code; the speedup was achieved by simply making sure that the vector abstraction complied with AVX-512. Additional optimizations for KNL might take advantage of the AVX-512CD conflict detection instructions and different code for gather operations.

To lead up to the scaling results across multiple Xeon Phi augmented nodes, Fig. 8 measures the performance of individual such nodes as listed in Table III. Like in a real simulation, the workload is shared among CPU and accelerator. Given that our KNL system is self-hosted, we include it in this list. The measurements for CPU+KNC, include both the overheads incurred in a single node execution (such as MPI and threading), and the overhead due to offloading. In view of the performance of the CPU-only systems relative to KNC, these performance numbers are then plausible. A single KNC delivers higher simulation speed than the CPU-only SB node; however, a CPU-only HW node is more powerful than the KNC, and thus also noticeably contributes to the combined performance. Adding a second accelerator also seems to improve performance, as seen in the IV+2KNC measurement. The KNL system delivers higher performance than the combination of two first-generation Xeon Phis and two Ivy Bridge CPUs.

The question with any kind of serial improvement is "will it translate to similar speedups in a (highly) parallel envi-

TABLE III: Hardware used in the evaluation of the Xeon Phi performance.

Name	CPU	Cores	ISA	Accelerator	Cores	ISA
SB+KNC	Intel Xeon E5-2450	2 × 8	AVX	Intel Xeon Phi 5110P	60	IMCI
IV+2KNC	Intel Xeon E5-2650v2	2 × 8	AVX	Intel Xeon Phi 5110P	2 × 60	IMCI
HW+KNC	Intel Xeon E5-2680v3	2 × 12	AVX2	Intel Xeon Phi 5110P	60	IMCI
KNL	–	–	–	Intel Xeon Phi 7250	68	AVX-512

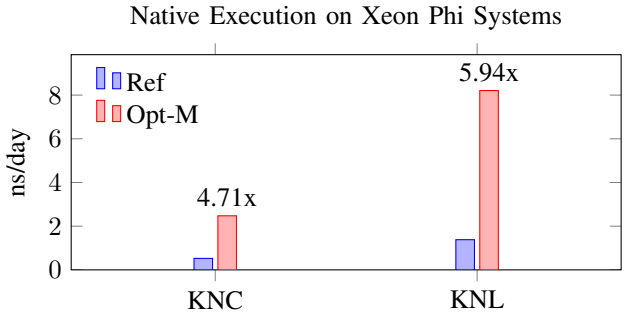


Fig. 7: Evaluation of performance portability for Xeon Phi's. 512 000 atoms.

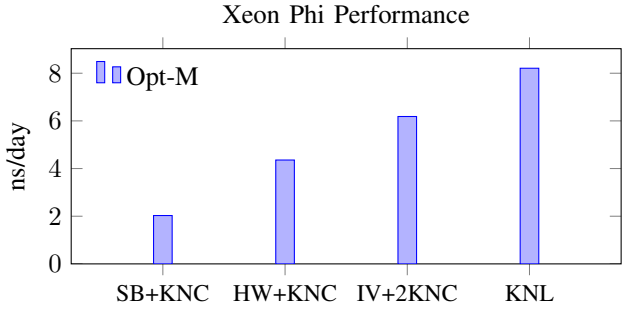


Fig. 8: Performance gains with Xeon Phis. 512 000 atoms.

ronment?". In theory, sequential improvements multiply with the performance achieved from parallelism; in practice, much of those improvements are lost to various overheads, and a realistic assessment can only be made from measurement. Figure 9 depicts results for up to eight nodes in a cluster of IV+2KNC nodes. Here, overheads are incurred both by parallelism within a node as well as communication among nodes. The vector optimizations port to large-scale computations seamlessly: without accelerator, the performance improvement for 196 MPI ranks is 2.5x; when two accelerators are added per node, the performance improvement becomes 6.5x.

VII. Conclusions

We discussed the problem of calculating the Tersoff potential efficiently and in a portable manner; we described a number of optimization schemes, and validated their effectiveness by means of realistic use cases. We showed that vectorization can achieve considerable speedups also in scenarios—such as a multi-body potential with a short neighbor list—which do not immediately lend themselves to the SIMD paradigm. To achieve portability, it proved useful to isolate target-specific code into a library of abstract operations; this separation of

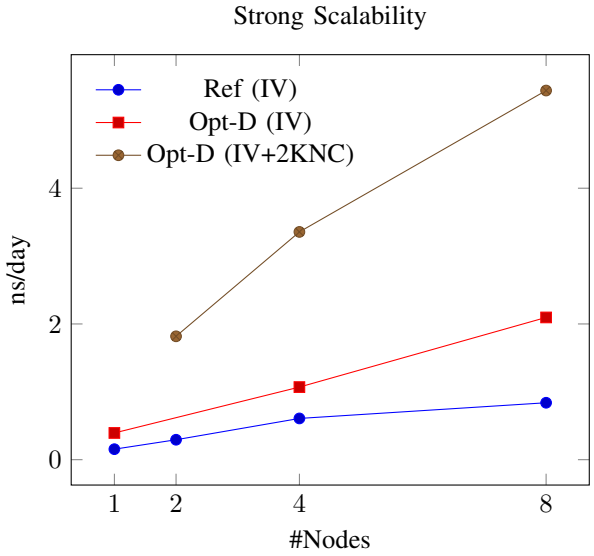


Fig. 9: Optimization results on SuperMIC, a Xeon Phi augmented cluster. 2 million atoms.

concerns leads to a clean division between the algorithm implementation and the hardware support for vectorization. It also makes it possible to map the vector paradigm to GPUs, while attaining considerable speedup. The ideas behind our optimizations were described, and their effectiveness was validated by means of realistic use cases. Indeed, we observe speedups between 2x and 3x on most CPUs, and between 3x and 5x on accelerators; performance scales also to clusters and clusters of accelerators. Finally, we believe that the main success of this work lies in the achieved degree of cross-platform code reuse, and in the portability of the proposed optimizations; combined, these two features lead to a success story with respect to performance portability.

VIII. Acknowledgments

The authors gratefully acknowledge financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111, and from Intel via the Intel Parallel Computing Center initiative. We thank the RWTH computing center and the Leibniz Rechenzentrum München for computing resources to conduct this research. We would like to thank Marcus Schmidt for providing one of the benchmarks used in this work, and M. W. Brown for conducting the benchmarks on the 2nd generation Xeon Phi hardware.

References

[1] S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, J Comp Phys, 1995.

[2] Wolf et al, Assessing the Performance of OpenMP Programs on the Intel Xeon Phi, Lecture Notes in Computer Science, Euro-Par 2013 Parallel Processing, 2013.
[3] Brown et al, An Evaluation of Molecular Dynamics Performance on the Hybrid Cray XK6 Supercomputer, Procedia Computer Science, 2012.
[4] Brown at al, Implementing molecular dynamics on hybrid high performance computers—Three-body potentials, Computer Physics Communications, 2013.
[5] Hou et al, Efficient GPU-accelerated molecular dynamics simulation of solid covalent crystals, Computer Physics Communications, 2013.
[6] Brown et al, Optimizing legacy molecular dynamics software with directive-based offload, Computer Physics Communications, 2015.
[7] Tersoff, New empirical approach for the structure and energy of covalent systems, Phys. Rev. B, 1988.
[8] Heinecke et al, Supercomputing for Molecular Dynamics Simulations, Springer International Publishing, 2-15.
[9] Páll et al, Tackling Exascale Software Challenges in Molecular Dynamics with GROMACS, Solving Software Challenges for Exascale, Lecture Notes in Computer Science, 2015.
[10] Tian et al, Compiling C/C++ SIMD Extensions for Function and Loop Vectorization on Multicore-SIMD Processors, IPDPSW, 2012.
[11] E. Bitzek, et al, Recent developments in IMD: Interactions for covalent and metallic systems, High Performance Computing in Science and Engineering '2000, 2001.
[12] J. Roth et al, IMD - A Massively Parallel Molecular Dynamics Package for Classical Simulations in Condensed Matter Physics, High Performance Computing in Science and Engineering '99, 2000.
[13] S. J. Plimpton and A. P. Thompson, Computational Aspects of Many-body Potentials, MRS Bulletin, 37, 2012.
[14] H.J.C. Berendsen, D. van der Spoel, R. van Drunen, GROMACS: A message-passing parallel molecular dynamics implementation, Computer Physics Communications, 1995.
[15] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilrd Pll, Jeremy C. Smith, Berk Hess, Erik Lindahl, GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers, SoftwareX, September 2015.
[16] H. Carter Edwards, Christian R. Trott, Daniel Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput., 2014.
[17] W. M. Brown, P. Wang, S. J. Plimpton, A. N. Tharrington, Implementing Molecular Dynamics on Hybrid High Performance Computers - Short Range Forces, Comp Phys Comm, 2011
[18] Brown, W.M., Carrillo, J.-M.Y., Gavhane, N., Thakkar, F.M., Plimpton, S.J., Optimizing Legacy Molecular Dynamics Software with Directive-Based Offload, Computer Physics Communications, To appear.
[19] O. E. B. Messer, E. DAzevedo, J. Hill, W. Joubert, S. Laosooksathit and A. Tharrington, Developing MiniApps on Modern Platforms Using Multiple Programming Models, Cluster Computing (CLUSTER), 2015 IEEE International Conference on, 2015.
[20] S. Pll, B. Hess, A flexible algorithm for calculating pair interactions on SIMD architectures, Computer Physics Communications, 2013.
[21] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, S.A. Jarvis, Exploring SIMD for Molecular Dynamics, Using Intel®Xeon®Processors and Intel®Xeon Phi Coprocessors, IPDPS '13, 2013.
[22] J. A. Anderson, C. D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics, 2008.
[23] D. C. Rapaport, Enhanced molecular dynamics performance with a programmable graphics processor, Computer Physics Communications, 2011.
[24] Z. Fan, T. Siro, A. Harju,Accelerated molecular dynamics force evaluation on graphics processing units for thermal conductivity calculations, Computer Physics Communications, 2013.
[25] C. Hou, J. Xu, P. Wang, W. Huang, X. Wang, Efficient GPU-accelerated molecular dynamics simulation of solid covalent crystals, Computer Physics Communications, 2013.
[26] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, K. Schulten, Scalable molecular dynamics with NAMD, Journal of Computational Chemistry, 2005.
[27] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. W. Glass, H. Hasse, J. Vrabec, M. Horsch, ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems, Journal of Chemical Theory and Computation, 2014.
[28] W. Smith, I. T. Todorov, A short description of DL_POLY, Molecular Simulation, 2006.
[29] F. W. J. Olver, D. W. Lozier, R. F. Boisvert, C. W. Clark, editors, NIST Handbook of Mathematical Functions, Cambridge University Press, 2010.
[30] M. P. Allen, D. J. Tildesley, Computer Simulation of Liquids, Oxford University Press, 1987.
[31] S. Hampton, P. K. Agarwal, S. R. Alam, P. S. Crozier, Towards microsecond biological molecular dynamics simulations on hybrid processors, Proceedings of the 2010 International Conference on High Performance Computing and Simulation, 2010.
[32] N. Schmid, M. Bötschi, W. F. van Gunsteren, A GPU solvent-solvent interaction calculation accelerator for biomolecular simulations using the GROMOS software, Journal of Computational Chemistry, 2010.
[33] http://github.com/HPAC/lammps-tersoff-vector.

APPENDIX

ARTIFACT DESCRIPTION: THE VECTORIZATION OF THE TERSOFF MULTI-BODY POTENTIAL: AN EXERCISE IN PERFORMANCE PORTABILITY

A. Abstract

We provide source code implementing all the optimizations discussed in the paper, as well as the LAMMPS source code in the version used for benchmarking. To create binaries from the source code, we supply build scripts for various systems used in the study, providing sufficient example for testers to create their own build procedures.We also provide input scripts used to conduct the benchmarks and the accuracy test presented in the paper. Finally, we include output files from many benchmarking runs, exemplifying the results one would expect to see.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Tersoff Potential.
- **Program:** LAMMPS.
- **Compilation:** CUDA Dev-Kit, ICC, GCC (only ARM).
- **Transformations:** See `build.sh` files.
- **Binary:** -
- **Data set:** LAMMPS-provided benchmark.
- **Run-time environment:** Intel MPI (except ARM).
- **Hardware:** Various x86 (including Xeon Phi 1st and 2nd generation), ARM, GPU from Kepler onwards.
- **Output:** `Performancec` as reported by LAMMPS.
- **Experiment workflow:** See below.
- **Publicly available?:** Yes.

2) How delivered: The entire source tree is delivered in both an archive file in the supplementary material, and can also be downloaded from GitHub at `github.com/HPAC/lammps-tersoff-vector`. The GitHub repository will be maintained (the version in the archive is tagged as "sc16"), and can be used to submit issues via the issue tracking system.

3) Hardware dependencies: Since the paper is meant to showcase performance portability, the software runs on a wide selection of hardware, including any x86 platform starting from Westmere, and platforms with ARM NEON support. It should also run on platforms without any vector instruction sets (running in scalar mode), and still attaub speedups due to scalar optimizations, alignment etc.The GPU code requires CUDA, and compute capability 3.0 up (i.e. Kepler and onwards). The code was tested for compute capability 3.5. Note that changing the compute capability requires edits to some Makefiles (consult LAMMPS documentation).

4) Software dependencies: For x86 platforms, one requires the Intel Compiler 14.0 or up. Additionally, GPU builds require the NVIDIA CUDA Development Kit. ARM builds use gcc.

5) Datasets: For testing and benchmarking, we extended the Tersoff benchmark provided by LAMMPS at `http://lammps.sandia.gov/bench/bench_tersoff.tar.gz`. The directory `benchmarks/lammps` contains all the benchmark's scripts, input files and parameter files; the experiments should be executed from that directory, since it not only contains the input files specifying simulation procedures, but also parameter files describing materials properties.

C. Installation

As examples, we provide build scripts that target various systems (`machines/*/build.sh`), following a specific naming scheme best explained by example: The folder `rwth-sb_tesla` contains a build script for a system owned by RWTH Aachen, with a Sandy Bridge Core, and a NVIDIA Tesla GPU.

To allow for multiple system's builds to live in the same source tree, we recommend to use a new subdirectory for every new system. Copy a script that matches your system (If you want to measure GPUs, take a GPU system's script, if ARM, the ARM script, and so on) to that directory, and customize it to fit your machine configuration: Load appropriate modules, make sure all software versions are consistent, etc. Then invoke the customized build script to create binaries.

Note that each build script will create a whole variety of binaries (some meant for experimentation) and that the build process is rather convoluted when building for GPU systems, as some patchwork is necessary toso that ICC and NVCC work together. See the example scripts for detail.

The produced binaries will be reated in the directory `machines/<your-machine>/lammps-10Mar16/src`. They are named using the following scheme:

- `lmp_intel_{cpu,mic,phi}_<suffix>` Binaries containing code from the USER-INTEL package (including our code), specifically optimized for the current system's architecture. The naming reflects the binaries target and compilation options. CPU: Only runs on the CPU; MIC: Only runs natively on a 1st gen Xeon Phi; PHI: runs on current arch, can offload to 1st gen. Xeon Phi. The `SUFFIX` part describes the configuration of our optimized code, and has the shape `{default,packi,nopacki}_{vector,scalar}`. DEFAULT: software chooses packing scheme; PACKI: use the scheme from Fig. 1b; NOPACKI: use the scheme from Fig. 1a; VECTOR: use vectorization; SCALAR: only use scalar improvements.
- `lmp_kokkos_{cuda,omp}_{novect,vect}` CUDA: Offload calculation exclusively to GPU; OMP: Run code using OpenMP. NOVECT: use the vanilla code for Tersoff that KOKKOS comes with; VECT: Use our code.
- `lmp_mpi_gpu_{single,double,mixed}` Reference versions for Tersoff on the GPU, no modifications from our side. Build in single, double, and mixed precision.
- `lmp_serial_<suffix>` On ARM, we lack MPI support, so we only build a serial version (i.e. no MPI, but only OpenMP support). The naming conventions from above apply.

Example: Let us run the code on an x86 system using the original benchmark of 32 000 atoms.

```
$ cd benchmarks/lammps
$ ../../machines/<your-system>/lammps-10Mar16/src/lmp_intel_cpu_default_vector \
> -in in.tersoff -v p vanilla -pk intel 0 mode mixed -sf intel
...
Performance: ... xxx ns/day ...
...
```

This will run our vectorized code in mixed precision, with a system specified in the `in.tersoff` file.

D. Experiment workflow

For the paper, two kinds of experiments were carried out: First, we investigated the impact of reduced accuracy on the simulation quality (Fig. 3). Second, we spent most of our effort to accurately estimate the performance gained through our optimizations.

Typically, these experiments are conducted using the shell scripts found in `benchmarks/lammps`, but to replicate individual data points it is useful to give an overview over the general command syntax.

One typically needs to run LAMMPS from the directory where the input file is situated. In our case, that means that any benchmarks should be run from `benchmarks/lammps`.

The most important command line parameters to pass to LAMMPS follow.

- `-in <input-file>`
  Specify the input file.
- `-pk intel 1 mode <mode> balance {-1/0/1} tpc 1,2,3,4 -sf intel`
  Runs the code from this paper, in the chosen precision `mode`, which can be `single`, `double` or `mixed`, with the offload setting given by `balance`: -1 is automatic, 1 is fully offloaded, and 0 is not offloaded at all; `tpc`: For offloading, how many threads should run on each Xeon Phi core.
- `-pk omp 0 -sf omp`
  Run reference code that has OpenMP support, always uses double precision.
- `-v p {kokkos/vanilla}`
  Choose KOKKOS if running a KOKKOS binary.
- `-sf kk -k on t 0 g 1`
  Enable KOKKOS run with a single GPU.

In addition, one might want to invoke LAMMPS using the MPI startup program of choice. It is recommended use on rank per physical core. OpenMP threads can be used to take advantage of hyper-threading. It is important to properly bind these threads.

Please note that many of the aforementioned scripts contain hard-coded paths, so please inspect them before execution.

1) Accuracy: An accuracy study just performs a high number of timesteps to check if the total energy deviates from the reference. To do so, the script `accuracy.sh` was used. Depending on available machine resources, it needs to be customized in the number of MPI ranks to use.

2) Performance: To measure performance, the paper presents a big number of data points. As such, their collection is automated using shell scripts `bench-*.sh` with rather self-explanatory names. The multi-node benchmarks, which require job scripts, use code under `machines/lrz-ivy_phi`. There is a python script to generate the job files, to be submitted using LoadLeveler (as is used on SuperMUC). This would most likely also be a good starting point to generate batch scripts for other cluster environments.

E. Evaluation and expected result

For an accuracy experiment, one would compare the output's total energy columns from a reference run (in double precision) with those from a run with reduced precision. In the paper, we plotted the relative deviation, i.e. $\frac{|\mathrm{ref}-\mathrm{trial}|}{|\mathrm{ref}|}$ over the entire simulation. We would expect this relative deviation to both stay small (less than 1 percent), and oscillate (i.e. have minima and maxima, return to zero). The script `benchmarks/lammps/accuracy.sh` will produce the data for the accuracy plot in the paper, the raw data being stored in `benchmarks/lammps/acc-*`.

For performance experiments, one would compare our code against a reference code. The speedup due to running with our code should be at least 2x on any architecture, and approach 5x on specific architectures such as KNC.

For reference, we provide the output from our runs of the benchmarking scripts on a variety of hosts. The raw results are stored in the directories `benchmarks/lammps/results*/*`, and filenames always contain the hostname of the benchmarked machine. The hosts `cluster-phi, linuxihdc*, linuxihbc*` are x86 machines, some of which augmented by Xeon Phis. The hosts `linuxnvc*` are x86 machines augmented with GPUs. The `machines/lrz-ib_phi/run*` directories contain results from SuperMIC and the ARM results are available in `benchmarks/lammps/harm-*`.