Redesigning LAMMPS for Peta-Scale and Hundred-Billion-Atom Simulation on Sunway TaihuLight

Xiaohui Duan[1,4], Ping Gao[1,4], Tingjian Zhang[1,4], Meng Zhang[1,4], Weiguo Liu[1,4], Wusheng Zhang[2,4], Wei Xue[2,4], Haohuan Fu[3,4], Lin Gan[2,4], Dexun Chen[2,4], Xiangxu Meng[1], Guangwen Yang[2,3,4]

[1]School of Software, Shandong University, Jinan, China

[2]Department of Computer Science and Technology, Tsinghua University, Beijing, China

[3]Ministry of Education Key Laboratory for Earth System Modeling, and Department of Earth System Science, Tsinghua University, Beijing, China

[4]National Supercomputing Center in Wuxi, Wuxi, China

sunrise.duan@mail.sdu.edu.cn, adch@263.net, missximon_7@163.com, {haohuan, xuewei, ygw}@tsinghua.edu.cn
{lingan, zws}@mail.tsinghua.edu.cn, {qdgaoping, sdubeyhhhh}@gmail.com, {weiguo.liu, mxx}@sdu.edu.cn

***Abstract*—Large-scale molecular dynamics (MD) simulations on supercomputers play an increasingly important role in many research areas. In this paper, we present our efforts on redesigning the widely used LAMMPS MD simulator for Sunway TaihuLight supercomputer and its ShenWei many-core architecture (SW26010). The memory constraints of SW26010 bring a number of new challenges for achieving efficient MD implementation on it. In order to overcome these constraints, we employ four levels of optimization: (1) a hybrid memory update strategy; (2) a software cache strategy; (3) customized transcendental math functions; and (4) a full pipeline acceleration. Furthermore, we redesign the code to enable all possible vectorization. Experiments show that our redesigned software on a single SW26010 processor can outperform over 100 E5-2650 v2 cores for running the latest stable release (11Aug17) of LAMMPS. We also achieve a performance of over 2.43 PFlops for a *Tersoff* simulation when using 16,384 nodes on Sunway TaihuLight.**

I. Introduction

Molecular dynamics (MD) simulation is one of the most popular supercomputing applications. It solves dynamics equations in particle level to achieve accurate simulations of the characteristic of materials or the structure of big moleculars. There are some well implemented MD tools like LAMMPS [18], GROMACS [2], Amber [3] and NAMD [16]. Among those tools, LAMMPS is a very popular MD simulator with widely supported potentials and easily extensible function modules. In practice, LAMMPS has been widely used to support simulations for real world applications.

LAMMPS uses neighbor lists to accelerate the N-body interactions computation and it employs the domain decomposition method to implement the MPI parallelization. While neighbor lists can eliminate unnecessary interactions between far atoms, it is still very compute intensive for some complex interactions with many transcendental mathematical functions. In practice, the simulation progress can be very time-consuming: a top-level desktop processor could simulate only 0.4 ns per day on a $512,000$ atoms system for the *Tersoff* potential, and for more complex potentials, e.g. the ReaxFF [12], it is several times slower.

In practice, there are high demands to accelerate large-scale MD simulations on supercomputers. Since newly manufactured supercomputers often come with many-core architectures, there have been some previous work on optimizing LAMMPS on heterogeneous many-core platforms such as Xeon Phis, GPUs [15], and SW26010 [4].

The Sunway TaihuLight supercomputer [8] provides a theoretical peak performance of 125 PFlop/s and an effective performance-to-power ratio of over 6 GFlop/s per watt. It consists of $40,960$ nodes with 1.4 PB attached memory. Each node on TaihuLight is equipped with a single SW26010 processor that is subdivided into four core groups (CGs). Each CG contains a *management processing element* (MPE) and 64 *computing processing elements* (CPEs), and 8 GB of attached DDR3 shared memory. The 8 GB attached shared memory can be accessed by both the MPE and the 64 CPEs via a memory controller with a shared bandwidth of approximately 136 GB/s. The MPE has 32 KB L1 instruction cache and 32 KB L1 data cache, with 256 KB L2 cache for both instructions and data. Each CPE has its own 16 KB L1 instruction cache and no data cache. And each CPE can access 64 KB of fast *local device memory* (LDM). Data residing in attached shared memory can be written to LDM by using DMA intrinsics and subsequently communicated via a broadcast to the LDM of other CPEs.

SW26010 has a peak performance up to 3.06 TFlops/s. However, its following memory constraints have posed great new challenges for redesigning efficient LAMMPS on it:

1) Low memory bandwidth: The 136 GB/s peak bandwidth is a big bottleneck for LAMMPS applications.

2) Lack of memory hierarchy: MPE has 256 KB L2 cache as the last level of cache, and CPEs do not have data cache, the last buffer from each CPE to shared memory is the 64 KB LDM. This forces us to access the shared memory frequently for fetching or storing data. This increases the demand of memory bandwidth. Also, software controlled LDM requires us predict the memory access or use inefficient discrete memory access. Besides, There is no shared caches, which makes us hard to store constants (e.g. math function lookup tables).

3) DMA may block LDM access: DMA instructions can be done asynchronously, but it may cause CPE's LDM access blocked. This will reduce the effect of prefetching or overlapping.

4) The weak MPE: there is only one MPE in each CG, and the MPE comes as a very low performance core, with low frequency, small cache, long latency on floating point operations. In [7], even data pack/unpack is done on CPEs for better performance.

5) Lack of SIMD instructions on CPEs: there is no *gather* or *scatter* instructions. MD requires many irregular memory accesses, and thus it is difficult to implement vectorized memory R/W without these instructions.

Based on the above characters of SW26010, we can see that applications with regular memory accessing patters can be usually handled efficiently on it. However, LAMMPS is a typical N-body application with irregular memory accessing patterns. Thus to achieve high efficiency, it will be not enough for us to only use traditional optimization approaches to port LAMMPS onto SW26010. In order to solve the new challenges, we have introduced the following major contributions in the paper:

1) We provide a ***hybrid memory update strategy*** to solve the write conflicts in three body interactions without involving redundant computations.
2) We design a ***software cache strategy*** to fully utilize the memory bandwidth when there are random read in pairwise interactions.
3) We implement customized transcendental math functions so that to eliminate searching lookup tables.
4) We do a full pipeline acceleration targeting the interactions, integration and neighbor list building.

The rest of this paper is organized as follows: In Sec.II, we introduce the basic MD simulation algorithm and previous work on accelerating MD on different computer architectures. Sec.III presents our optimization strategies and their efficient implementation. Performance is evaluated in Sec.IV. Sec.V concludes the paper.

II. Background

A. Molecular Dynamics Simulations

There are two categories of molecular simulation methods: MD and Monte Carlo (MC). Compared to MC, MD uses actual time evolution so that trajectory information can be acquired from MD results, enabling people to research a molecular procedure in a femtosecond scale. MD simulation usually contains two parts, the integration part and the interaction part.

$$\begin{cases} \vec{v}_{k+1} = \vec{v}_k + c_k \dfrac{\vec{F}(x_k)}{m} t \\ \vec{x}_{k+1} = \vec{x_k} + d_k \vec{v}_{k+1} t \end{cases} \tag{1}$$

The integration part do time domain integration following a set of simple motion equations with symplectic methods. For example, if the volume is constant and the energy is constrained, it follows Equation (1), where $\vec{x}$, $\vec{v}$ and m is the coordinate, velocity and mass of an atom separately. t denotes timestep and $\vec{F}$ is the force that atom affected by. c and d are parameters for the symplectic method. For instance, in the verlet method, $c_1 = 0, c_2 = 1, d_1 = d_2 = \frac{1}{2}$.

$$\begin{cases} V = \sum_{i,j}^{N} V_2(r_{ij}) + \sum_{i,j,k}^{N} V_3(r_{ij}, r_{ik}, \theta_{ijk}) + \cdots \\ \vec{F_i} = -\nabla_{\vec{ri}} V \end{cases} \tag{2}$$

Equation (2) shows the basic principle of interaction part. There are various approaches for analyzing the $V_2, V_3, \cdots$, which are also called *potentials*.

Also, since the atoms in a far distance contributes little effects to an atom, a neighbor list containing only short-range neighbors is often involved in MD for reducing the computation workload.

Our work targets both the *Lennard-Jones (L-J)* potential [13] and the *Tersoff* potential [20]. *L-J* potential is one of the most simple potentials in LAMMPS. It approximates the sum of *Pauli Repulsion* and *van der Walls Force* as follows:

$$V_{ij}(r) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{3}$$

In Equation (3), r_{ij} stands for the distance between atom i and atom j. As shown in Equation (1), the *L-J* potential is not difficult to compute. It just iterates over the neighbors of each atom and computes forces between atom pair (i, j) and then sums up the forces.

Tersoff potential is far more complex than *L-J* potential. Its V_{ij} is described as follows:

$$V_{ij} = f_C(r_{ij})[f_R(r_{ij}) + b_{ij} f_A(r_{ij})] \tag{4}$$

where $f_C(r)$ is a smooth cutoff function which contains trigonometric functions, $f_R(r) = Ae^{-\lambda_1 r}$ and $f_A(r) = -Be^{-\lambda_2 r}$ are repulsive and attractive terms.

What matters the most is the b_{ij} term, it is calculated by a few functions:

$$\begin{aligned} b_{ij} &= \frac{1}{(1 + \beta^n \zeta_{ij}^n)^{\frac{1}{2n}}} \\ \zeta_{ij} &= \sum_{k \neq i,j}^{N} f_C(r_{ij}) g(\theta_{ijk}) e^{\lambda_3^3 (r_{ij} - r_{ik})^3} \\ g(\theta) &= 1 + \frac{c^2}{d^2} - \frac{c^2}{d^2 + (h - \cos\theta)^2} \end{aligned} \tag{5}$$

.

As we can see from Equation (5), asymmetric three body interactions is needed to calculate *Tersoff* potential. Also, since the three body interactions is involved, it has better accuracy than *L-J* potential when simulating crystalline materials.

B. Previous Work on Accelerating MD

There have been a variety of techniques used to accelerate MD simulations. They range from typical high performance computing (HPC) strategies such as clustering to novel processing architectures. In this section, we discuss the various strategies used in accelerating MD. We broadly classify these strategies into two categories: coarse-grained and fine-grained.

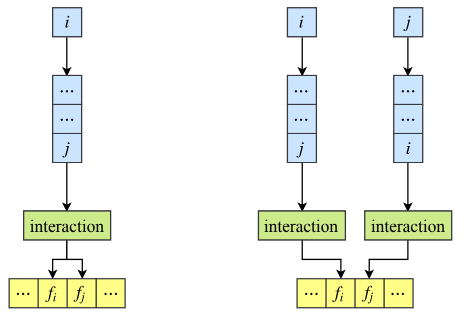
The architectures in the coarse-grained category include general-purpose supercomputers and PC clusters. In the year of 2002, NAMD [17] won the Gordon Bell prize by scaling MD to thousands of processor cores. Using supercomputers such as Blue Gene [6] for MD can also provide a tremendous performance. In addition, as many accurate simulation using a long-range solver like PPPM, W. Mcdoniel et al. [14] present an approach to optimizing the computational kernels with an implementation of PPPM particularly suitable by means of vectorization on the Knights Landing self-boot processors.

Computer architectures in the fine-grained category include special-purpose architectures, reconfigurable architectures and GPUs. D.E. Shaw et al. introduce their new machine Anton [19] which is a completed special-purpose supercomputer designed for MD simulations of biomolecular systems with customized ASICs to gain better performance.

Reconfigurable systems are based on programmable logic such as field-programmable gate arrays (FPGAs), expecially for short-ranged potentials, N. Varini et al. [21] present an implementation about Clathrate Hydrates on a specialized hardware platforms within a FPGA-based accelerator MD-GRAPE 3. Besides, S. Kasap et al. [11] give the first attempt to port LAMMPS to a FPGA-based parallel computers.

As for GPUs, the main advantage of them is that they are commodity components. Glaser. J et al. [9] offer an approach called *HOOMD-blue* [1], and they scale the *L-J* potential to $3,375$ GPUs simulating 108 million particles. Z. Fan et al. do a further research on GPUs as they propose a new force computation algorithm, GPUMD [5], which is free of write conflicts because the force, virial stress, and heat status of an atom can be accumulated in each thread.

The paper presented by W. Dong et al. [4] and M. Höhnerbach et al. [10] are close to the approach presented in this paper. The work by W. Dong also used Sunway TaihuiLight and its SW26010 to accelerate LAMMPS. However, it supports only the already well-studied *L-J* potential. And memory access optimizations are mainly done by software based prefetching, which can not make full use of the bandwidth of SW26010. Also, the vector read/write in it was done in the scalar form which is less efficient. Our approach utilizes vectorized shuffle instructions to reduce the number of load/store so that to achieve much higher memory access efficiency. The work by M. Höhnerbach implemented a very impressive and efficient vectorization strategy for the *Tersoff* potential and optimization techniques mainly from the computation side were carried out. This work does not have to do much memory access optimization since platforms involved in it have sufficient bandwidth. In our work ,however, we have to face more new challenges from the memory access side. This is mainly because many memory constraints of SW26010 (see Sec.I) have made the efficient redesigning for both the pair *L-J* and the complex multi-body *Tersoff* potentials in LAMMPS a very difficult task. The solution presented in this paper overcomes these memory constraints by using a combination of optimization techniques mainly from the memory access side. In particular, our solutions can handle the irregular memory accessing patterns in LAMMPS efficiently on SW26010. Experiments show that both the *L-J* and *Tersoff* potentials in LAMMPS can be efficiently parallelized on SW26010 using our approaches.



(a) Using half list for pairwise interactions
(b) Using full list for pairwise interactions.

Fig. 1. Two forms of neighbor list for pairwise interactions.

III. Optimization

In order to make full use of the computing power of SW26010 and overcome its memory constraints mentioned in Sec.I, we have applied various optimization techniques which are introduced in this section. Our optimization methods are generic and can be applied to potentials with similar algorithm characteristics.

A. Hybrid Memory Updating Strategy

Random memory updating in pairwise interactions is one of the largest challenges in parallelization of LAMMPS. Original algorithm in LAMMPS uses *Newton's 3rd Law* as shown in Fig. 1(a) to save computation costs. In Fig. 1(a), i-atom is accessed sequentially while j-atom is accessed randomly, and this brings high cost for maintaining data consistency in parallel implementation.

We use an existing redundant computation approach (RCA) on GPUs [15] which changes the list to a full neighbor list shown in Fig. 1(b). RCA is used to trade computation for memory accessing. In this case, we can write only i-side of the pair (i, j), and the j-side will be written when computing pair (j, i).

But for three body interactions, the above method will require almost triple computation workload, which is too expensive for us. Moreover, RCA will generate more data swapping between CPEs and shared memory. This is not feasible on SW26010 since we do not have enough memory bandwidth.

The workflow of RCA is shown in Algorithm 1. Loop at line 1 computes the short range repulsive term of *Tersoff* and

Algorithm 1 RCA for *Tersoff*

```
Require:
    FETCH: fetch data from shared memory.
    STORE: store data to shared memory.
    CLEAN: clean a list
    PUSH: add an element to a list.
    IDX($x$): $x$'s index in short neighbor list.
    FILTER: filter atoms not needed to sum up in short neighbor as much as possible.
    SYNC: Sync across the cores.
 1: for $i \in atoms$ do
 2:     $\textsc{fetch}(neighbor(i))$
 3:     for $j \in neighbor(i)$ do
 4:         if $\textsc{r}(i, j) < cutoff$ then
 5:             $F_i \leftarrow F_i + \textsc{repulsive}(i, j)$        ▷ Repulsive term
 6:         if $\textsc{r}(i, j) < cutoff\_short$ then
 7:             $\textsc{push}(short\_neighbor(i), j)$        ▷ Filtration
 8:     $\textsc{store}(F_i)$
 9: for $i \in atoms$ do
10:     $\textsc{fetch}(short\_neighbor(i), F_i)$
11:     for $j \in short\_neighbor(i)$ do
12:         $\zeta_{ij,fw} \leftarrow 0$
13:         $\zeta_{ij,rev} \leftarrow 0$
14:         for $k \in short\_neighbor(i)$ do
15:             $\zeta_{ij,fw} \leftarrow \zeta_{ij,fw} + \textsc{zeta}(i, j, k)$        ▷ Sum $\zeta_{ij}$
16:         $\textsc{fetch}(short\_neighbor(j))$
17:         for $k \in short\_neighbor(j)$ do
18:             $\zeta_{ij,rev} \leftarrow \zeta_{ij,rev} + \textsc{zeta}(j, i, k)$        ▷ Sum $\zeta_{ji}$
19:         $F_i \leftarrow F_i + \textsc{force-zeta}(i, j, \zeta_{ij,fw})$
20:         $F_i \leftarrow F_i + \textsc{force-zeta}(j, i, \zeta_{ij,rev})$
21:     $\textsc{store}(F_i, \zeta_{ij})$
22: for $i \in atoms$ do
23:     $\textsc{fetch}(short\_neighbor(i), F_i)$
24:     for $j \in short\_neighbor(i)$ do
25:         for $k \in short\_neighbor(i)$ do
26:             $(fi, fj, fk) \leftarrow \textsc{attractive}(i, j, k, \zeta_{ij,fw})$
27:             $F_i \leftarrow F_i + fi$        ▷ Accumulate attractive term
28:         $\textsc{fetch}(short\_neighbor(j))$
29:         for $k \in short\_neighbor(j)$ do
30:             $(fj, fi, fk) \leftarrow \textsc{attractive}(j, i, k, \zeta_{ij,rev})$
31:             $F_i \leftarrow F_i + fi$        ▷ Accumulate attractive term
32:             $(fj, fk, fi) \leftarrow \textsc{attractive}(j, k, i, \zeta_{jk,fw})$
33:             $F_i \leftarrow F_i + fi$        ▷ Accumulate attractive term
34:     $\textsc{store}(F_i)$
```

filters shorter ranged neighbors for three body interactions. Loop at line 9 and line 22 computes attractive term. Since the attractive term is not symmetric for atom i, j, k, RCA has to calculate this term for i as both center (line 26) and end atom (line 30 and line 32). As we can see, the computation of ZETA is doubled and ATTRACTIVE is tripled. This method also requires $short\_neigh$ and ζ to be swapped to shared memory, and the access of $short\_neighbor(j)$ is also repeated for each neighbor (line 16 and line 28). RCA may be useful for three body interactions on GPUs since they have so many threads and a good GDDR/HBM bandwidth.

In our work, we notice that $short\_neighbor$ represents atoms in a bonded range. Thus, the number of $short\_neighbor$ should be small enough to be put in the LDM. Based on this fact, we can compute three body interactions in parallel but summing up them serially. Fortunately,

Algorithm 2 Hybrid memory updating for *Tersoff*

```
Require:
    Refer to Algorithm 1 for predefined operations.
 1: for $i \in atoms$ do        ▷ Parallellized by CPEs
 2:     $\textsc{clean}(short\_neighbor(i))$
 3:     $\textsc{fetch}(neighbor(i))$
 4:     for $j \in neighbor(i)$ do
 5:         $\textsc{fetch}(atom_j)$
 6:         if $\textsc{r}(i, j) < cutoff$ then
 7:             $F_i \leftarrow F_i + \textsc{repulsive}(i, j)$
 8:         if $\textsc{r}(i, j) < cutoff\_short$ then
 9:             $\textsc{push}(short\_neighbor(i), j)$
10:     $F\_end_{i,*} \leftarrow 0$
11:     for $j \in short\_neighbor(i)$ do
12:         $\zeta_{ij} \leftarrow 0$
13:         for $k \in short\_neighbor(i)$ do
14:             $\zeta_{ij} \leftarrow \zeta_{ij} + \textsc{zeta}(i, j, k)$
15:         $fpair \leftarrow \textsc{force-zeta}(i, j, \zeta_{ij})$
16:         $F_i \leftarrow F_i + fpair$
17:         $F\_end_{i,\textsc{idx}(j)} \leftarrow F\_end_{i,\textsc{idx}(j)} - fpair$
18:     for $j \in short\_neighbor(i)$ do
19:         for $k \in short\_neighbor(i)$ do
20:             $(fi, fj, fk) \leftarrow \textsc{attractive}(i, j, k, \zeta_{ij})$
21:             $F_i \leftarrow F_i + fi$
22:             $F\_end_{i,\textsc{idx}(j)} \leftarrow F\_end_{i,\textsc{idx}(j)} + fj$
23:             $F\_end_{i,\textsc{idx}(k)} \leftarrow F\_end_{i,\textsc{idx}(k)} + fk$
24:     $\textsc{filter}(short\_neighbor(i), F\_end_{i,*})$
25:     $\textsc{store}(F_i, F\_end_{i,*}, short\_neighbor(i))$
26: for $i \in atoms$ do        ▷ Done by MPE
27:     $\textsc{spin-wait}(size(short\_neighbor(i)) \geq 0)$
28:     for $j \in short\_neighbor(i)$ do
29:         $F\_end\_sum_j \leftarrow F\_end\_sum_j + F\_end_{i,\textsc{idx}(j)}$
30: $\textsc{sync}(MPE, CPEs)$
31: for $i \in atoms$ do        ▷ Parallelized by CPEs
32:     $F_i \leftarrow F\_end\_sum_i + F_i$
```

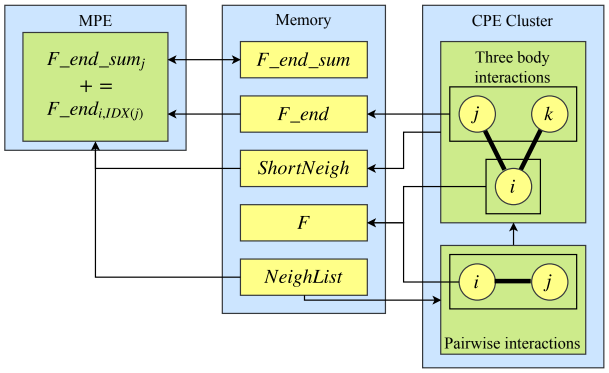


Fig. 2. MPE-CPE cooperation in three body interactions computation.

we have not assigned any computation task to MPE yet, and we can use MPE for summing up the forces. Then it comes to our hybrid memory updating strategy.

The workflow of our hybrid memory updating strategy is shown in Algorithm 2 and Fig. 2. In our method, we use pairwise interactions to filter the shorter range neighbors for three body interactions. The CPE cluster calculates the three body interactions, and we put the force status of the i-atom as usual (F). But for the j/k-atoms, we accumulate the force grouped by the short neighbor index (F_end), and then the

F_end is accumulated to F_end_sum on the MPE. Finally, we just need to sum up F and F_end_sum to acquire the total force that an atom affected by.

In this way, there is no dependency of ζ_{jk}, and we can fuse the loops to reduce data transportation. Besides, the cache of MPE is utilized for summing up the forces.

To guarantee the data consistency, we put the number of short range neighbors to shared memory after the three body interactions of a center atom is completely done, and the MPE spin waits until it comes to a non-negative value. Therefore we have a solution for the three-body interactions without redundant computation or random memory writing.

Furthermore, we add a filter before storing F_end and $short\_neighbor$, which will filter out ghost atoms, merge the short neighbor list across i-atom blocks, and pad the block to a multiplier of 8. Then we can eliminate the conditional statements of MPE's code and unroll the loop on MPE so that we can keep a balanced workload between MPE and CPEs.

B. Software Cache Strategy

Usually, the pairwise interactions is the first one to be done before other interactions, and it also filters neighbor lists for shorter-ranged interactions. Therefore the pairwise interactions faces a random memory reading for the original neighbor lists.

Some projects based on Sunway TaihuLight use pre-processing to reduce the cost of random memory access, for example, *swsptrsv* [22] accelerates sparse matrix solving by pre-processing. But for MD applications, the neighbor list is always changing, making it difficult to accelerate iterations by pre-processing.

We can not make the memory access predictable, but we can try to enlarge the memory accessing block to an acceptable size, and reuse data as much as possible to reduce the requirement of memory bandwidth. In our work, we have designed and implemented a software cache strategy to simulate cache behaviors by software logic and preserve some LDM space for caching data.

To gain a good cache performance, the key task is to reduce the *Average Memory Access Time (AMAT)*[23].

$$AMAT = HC + MR \cdot AMP \tag{6}$$

Equation (6) shows how $AMAT$ is determined, where HC is the number of cycles to load data when cache hits, MR is cache-miss ratio and AMP is average number of cycles to load data to cache when cache-miss. This shows that we must keep a balance of the three terms to acquire a good cache performance.

Usually, the cache design should concentrate on MR, but for simulating caches, HC is also critical because of software's overhead. To our best effort, we consider doing this by bit operations. By setting the cache line count and cache line size to power of 2, we can do the cache hit decision by using bit-wise operations, which can minimize the cost of cache hit decision.

For the MR term, we want to decrease it as much as possible. As it is known, the MR is usually determined

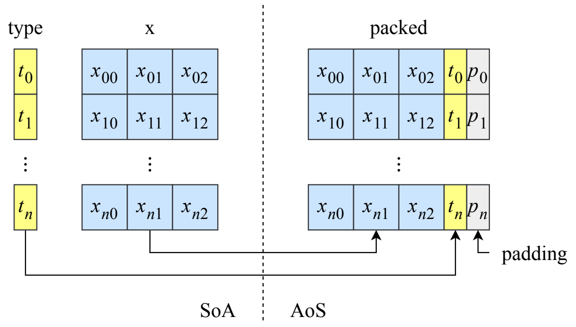

type
x
packed
t_0
t_1
t_n
x_{00} x_{01} x_{02}
x_{10} x_{11} x_{12}
x_{n0} x_{n1} x_{n2}
x_{00} x_{01} x_{02} t_0 p_0
x_{10} x_{11} x_{12} t_1 p_1
x_{n0} x_{n1} x_{n2} t_n p_n
padding
SoA
AoS

Fig. 3. Packing data for DMA access with an additional p to make the memory access aligned.

by cache size and cache strategy. We can not increase the cache size due to the limited size of LDM, so we choose a total cache size of 32 KB, which is the largest power of 2 less than the size of LDM. For the strategy, we try direct-mapped cache and group-associative cache methods. We evaluate the two strategies by applying them to CPE parallelized version of *L-J* potential, as a result, the MR of direct-mapped cache is $\sim 18\%$ while the MR of the group-associative cache is $\sim 15\%$. But for the group-associative cache, HC is much larger than direct-mapped cache, thus leading to lower performance.

At last, we consider reducing the AMP term by accelerating DMA operations to fetch data to LDM. Like many platforms, memory access performance is related to the alignment and size of memory access block. DMA instructions almost follow the same rules: **1) alignment:** DMA can achieve its peak bandwidth only if the accessed address is in an alignment of 128 bytes. **2) size:** DMA access is done by a 128-byte accessing block physically. But in our test, a 128-byte accessing block can achieve its peak bandwidth only if the accessed address is increased sequentially. Also, when an accessing block has a size of 256 bytes, it will reach the peak bandwidth even if the accessed address is distributed randomly.

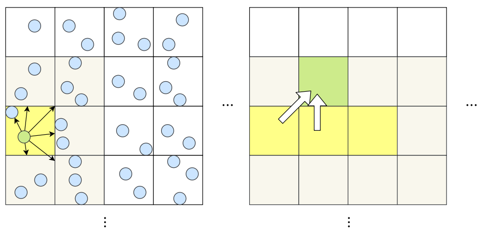
According to above preconditions, we carefully design our caching strategy in the following way: **1) Total size of the cache is 32 KB**, which is the maximum possible value to use bit-wise operations for cache hit decision. **2) Cache line size is 256 B**. This is the minimum size to achieve peak bandwidth for random memory access, and a larger size will increase MR and AMP. **3) The size of each cache line should be a multiplier of the atom data size**, or we must do a transform between atom index and memory address, which has a penalty on HC.

To achieve **3)** in the strategy above, we consider reorganizing the atom data from SoA (structure of arrays) form to AoS (array of structures) form as shown in Fig. 3. And an additional 4-byte padding is appended to the structure, thus we have an atom data structure of 32 bytes. By forming cache lines of 8 atoms (256 bytes), we align memory access blocks whichever cache line we fetch from memory.

After all preconditions are ready, we can implement a cache module and plug it into pairwise interactions as shown in Algorithm 3. The cache checking is done before the calcu-

Algorithm 3 The Caching Strategy

```
Require:
 1: $nlines \leftarrow 2^n$
 2: $linesize \leftarrow 2^m$
 3: $lmask \leftarrow nlines - 1$
 4: $emask \leftarrow linesize - 1$
 5: for $i\_atom \in all\_atoms$ do
 6:     for $j \in neighbors(i)$ do
 7:         $tag\_j \leftarrow$ SHIFT-RIGHT$(j, n+m)$
 8:         $line\_j \leftarrow$ SHIFT-RIGHT$(j, m) \wedge lmask$
 9:         $off\_j \leftarrow j \wedge emask$
10:         if $tag(line\_j) \neq tag\_j$ then
11:             $line\_off \leftarrow j \wedge \neg emask$
12:             DMA$(mem(line\_off), cache(line\_j))$
13:             $tag(line\_j) \leftarrow tag\_j$
14:         $j\_atom \leftarrow cache(line\_j, off\_j)$
15:         PAIRWISE-INTERACTION$(i\_atom, j\_atom)$
```



(a) Bining atoms and scan bins. (b) Bundling bins for scanning.

Fig. 4. Difference between original bin scanning method and bundled bin scanning method in a 2-D case.

lation of pairwise interactions of atoms (i, j), and we firstly determine whether the data of atom j is in cache. If it is missing, we use DMA to fetch the corresponding cache line from shared memory. Then we calculate the cache mapping address and obtain atom j's data from LDM.

C. Bundled Bin Scan for Neighbor List Build

The building of neighbor list is another hotspot in LAMMPS since the rebuilding may happen frequently during the iterations. The mostly used method to build a neighbor list is the *bin* method.

As shown in Fig. 4(a) and Algorithm 4, the traditional *bin* method firstly puts all particles into different bins (the grid in Fig. 4(a)) according to their coordinates, and then scans the bins around each atom's bin (STENCIL-BIN) to find its neighbors (the arrows in Fig. 4(a)).

Since the outer loop is used to handle the atom level, this will contribute a bad locality. In our method, we readjust the loop order and take bins as the outer loop as shown in Algorithm 5. In our case, a few atoms in a same *bin* will have the same stencil-bins, which can reduce the total amount of memory access and increase computation locality.

Algorithm 4 Original bin scanning algorithm for scanning

```
1: for $atom_i \in atoms$ do
2:     $bin_i \leftarrow$ ATOM2BIN$(atom_i)$
3:     for $bin_j \in$ STENCIL-BINS$(bin_i)$ do
4:         for $atom_j$ in $bin_j$ do
5:             if R$(atom_i, atom_j) < cut\_r$ then
6:                 PUSH$(neigh\_list(atom_i), atom_j)$
```

Algorithm 5 Use bins for outer loop

```
1: for $bin_i \in bins$ do
2:     for $bin_j \in$ STENCIL-BINS$(bin_i)$ do
3:         for $atom_i$ in $bin_i$ do
4:             for $atom_j$ in $bin_j$ do
5:                 if R$(atom_i, atom_j) < cut\_r$ then
6:                     PUSH$(neigh\_list(atom_i), atom_j)$
```

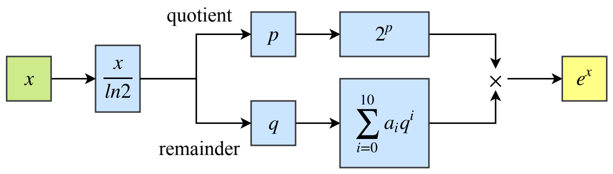


Fig. 5. Calculation of exponential functions, x is divided to a quotient part and remainder part and calculated by different methods.

In addition, we notice that adjacent bins share quite a few stencil-bins, so we bundle bins for CPE parallelized implementation, that means, one CPE will be distributed more bins (yellow cells in Fig. 4(b)), so that they can share some stencil-bins (e.g. the green bin in Fig. 4(b)) to reduce the pressure from memory bandwidth.

D. Eliminate Lookup Table in Transcendental Functions

Since the *GNU libm* implementations for transcendental functions on CPEs need to search lookup tables whose size is usually beyond the available LDM. This is because for transcendental functions used in *Tersoff* potential, *sin* and *cos* share a 4,096-byte lookup table while *pow* and *exp* occupy another 8,432 bytes, that is around 12 KB in size. Since 32 KB of LDM is reserved for software cache and another > 20 KB is used by various buffers, if 12 KB of LDM is occupied by lookup tables, LDM will overflow. But if we just leave the tables in the shared DDR3 memory, the performance will not be satisfying. Therefore in our work, we have designed a polynomial based approach to approximate these transcendental functions.

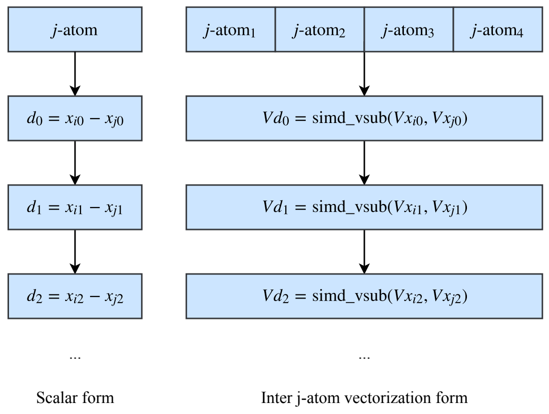
Our approach is illustrated in Fig. 5. We change the e^x to $e^{p \cdot ln2} \cdot e^q = 2^p \cdot e^q$ and $0 < q < ln2$ in Fig. 5. The 2^p can be calculated by setting the exponential part to a floating point number. For the e^q, we use a polynomial $P(x)$ to approximate it. When the order of $P(x)$ is 11, $|P(x) - e^x| < 10^{-14}$ for $0 < x < ln2$, which is accurate enough for the computation of *Tersoff* potential. Then we can calculate e^q by that polynomial. Finally, e^x is calculated by multiplying 2^p and e^q.

In the similar way, we implement $ln(x) = ln(2^p \cdot q)$ by calculating $ln(2^p) = p \cdot ln(2)$ and approximating $ln(q)$ by polynomials. Then *pow* is implemented by combining ln and exp. Also, sin and cos in $(-\pi, \pi)$ is approximated by polynomials while other inputs will be mapped to this range according to $sin(2k\pi + x) = sin(x)$.

In addition, we notice that for the calculation of polynomials, a naive implementation of *Horner's method* (Algorithm 6, line 1) can be accelerated easily by using FMA instructions, but the inter-instruction data dependency on p prevents us

Algorithm 6 Compare of naive implementation Horner's method and our adjusted method

```
1:  function NAIVE-HORNER(x, a, n)
2:      $p \leftarrow 0$
3:      for $i \in [n, n-1, ..., 2, 1, 0]$ do
4:          $p \leftarrow p \cdot x + a_i$
        return $p$
5:  function ADJUSTED-HORNER(x, a, n)
6:      $p_{even} \leftarrow 0$
7:      $p_{odd} \leftarrow 0$
8:      $xsq \leftarrow x^2$
9:      for $i \in [\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor - 1, ..., 2, 1, 0]$ do    ▷ Assume $n$ is odd
10:         $p_{odd} \leftarrow p_{odd} \cdot xsq + a_{2i+1}$
11:         $p_{even} \leftarrow p_{even} \cdot xsq + a_{2i}$
        return $p_{even} + p_{odd} \cdot x$
```



j-atom
$d_0 = x_{i0} - x_{j0}$
$d_1 = x_{i1} - x_{j1}$
$d_2 = x_{i2} - x_{j2}$
...
Scalar form
j-atom1
j-atom2
j-atom3
j-atom4
Vd_0 = simd_vsub(Vx_{i0}, Vx_{j0})
Vd_1 = simd_vsub(Vx_{i1}, Vx_{j1})
Vd_2 = simd_vsub(Vx_{i2}, Vx_{j2})
...
Inter j-atom vectorization form

Fig. 6. Inter j atom vectorization. We can have four operations at once using SIMD instructions.

from utilizing *Instruction Level Parallelism (ILP)*. We have adjusted it to a more pipeline friendly method (Algorithm 6, line 5) by accumulating even and odd terms of the polynomial separately to achieve a slightly better utilization of ILP. Since the pipeline is still far from full, our modification can provide about 80% speedup in calculating the 11-order polynomial in the calculation of e^x.

Our customized transcendental functions not only save a lot of space on LDM, but also make the vectorization more friendly. Performance of our implementation is a little slower than loading the table to LDM to compute for scalars. But vectorized version is slightly faster than manufacturer's version of vectorized math functions.

E. Vectorization

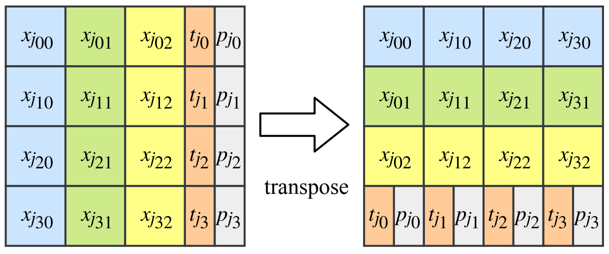
In our work, we have also implemented vectorization to exploit the available 256-bit SIMD vector registers.

Since the pairwise interactions computation need to compute a number of j-atoms for each i-atom, we prefer to use the *Sunway's SIMD extension* on CPEs to calculate the interactions of four j-atoms at once for the same i-atom in a low parallelization level. As there are a lot of scalar operations in the pairwise interactions, we decide to use an inter-j-atoms vectorization method shown in Fig. 6.

In practice, there are still some cases that prevent efficient vectorization. Three typical cases are shown in Algorithm 7,

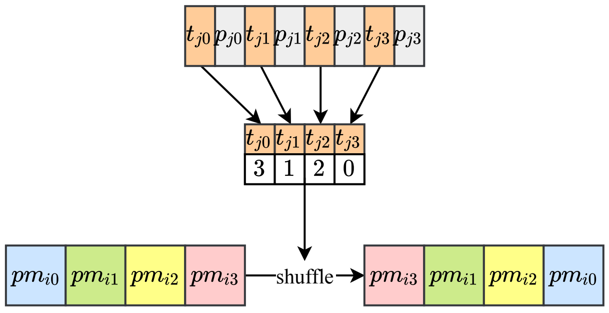
Algorithm 7 A simplified workflow for pairwise interaction for pair (i,j)

```
1: $itype \leftarrow type(i)$
2: $jtype \leftarrow type(j)$                      ▷ Load types from LDM
3: $d_{ij} \leftarrow x_j - x_i$
4: $r^2 \leftarrow |d_{ij}|^2$
5: if $r^2 < cut^2(itype, jtype)$ then
6:     $pm_{ij} \leftarrow pm(itype, jtype)$
7:     $fpair \leftarrow$ PAIR-WISE-INTERACTIONS$(i, j, d_{ij}, pm_{ij})$
8:     $f_i \leftarrow f_i + d_{ij} * fpair$
```



x_{j00} x_{j01} x_{j02} t_{j0} p_{j0}
x_{j10} x_{j11} x_{j12} t_{j1} p_{j1}
x_{j20} x_{j21} x_{j22} t_{j2} p_{j2}
x_{j30} x_{j31} x_{j32} t_{j3} p_{j3}
transpose
x_{j00} x_{j10} x_{j20} x_{j30}
x_{j01} x_{j11} x_{j21} x_{j31}
x_{j02} x_{j12} x_{j22} x_{j32}
t_{j0} p_{j0} t_{j1} p_{j1} t_{j2} p_{j2} t_{j3} p_{j3}

(a) Using transpose for loading j-atom data



t_{j0} p_{j0} t_{j1} p_{j1} t_{j2} p_{j2} t_{j3} p_{j3}
t_{j0} t_{j1} t_{j2} t_{j3}
3 1 2 0
pm_{i0} pm_{i1} pm_{i2} pm_{i3}
shuffle
pm_{i3} pm_{i1} pm_{i2} pm_{i0}

(b) Using shuffle for parameter lookup

Fig. 7. Two methods to organize the vectors needed to access LDM efficiently.

they include:

1) Irregular memory access for index j in line 2 and line 3.
2) Seeking $type(i), type(j)$ in a parameter table in line 6.
3) The if statement about the cutoff radius in line 5.

It is hard to solve case 1) because we cannot decide whether the four adjacent j are in LDM with no *gather* instruction supported. In our method, we access the data by j index and use vector loading for a cache entry of that atom. Then we make a transpose on the four entries to make an SIMD approach like Fig. 7(a). Meanwhile, the $type$ is also loaded and transposed during the process.

For case 2), we have two solutions: one is to reorganize all parameters to an AoS form, then solve it like case 1). The other solution can only be used when the number of atom types is less than 4. By forming the 4 $jtype$s as a mask, and loading pm_{i*} in a vector, we can use *shuffle* instruction to get the pm_{ij} vectors. The latter solution is shown in Fig. 7(b). The pm in both Algorithm 7 and Fig. 7(b) denotes parameters which depend on the types of atoms.

For case 3), there is no *masked instructions*, but we have *select instructions* to combine values from two vectors by a condition. Therefore, adding a *select* instructions to clear the

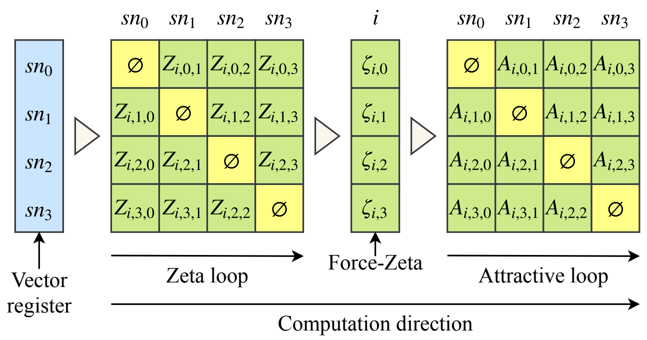


Fig. 8. Vectorization of *Tersoff* potential (based on our *hybrid memory updating* strategy in Algorithm 2). sn_* means the short neighbor-data. Z_{ijk} is the computation of $\text{ZETA}(i, j, k)$, ζ is the computation of $\text{FORCE-ZETA}(i, j)$, and A_{ijk} is the computation of $\text{ATTRACTIVE}(i, j, k)$, and $\varnothing$ denotes that cell does not need computation since $j = k$, and its result is cleared by *select* instructions.

TABLE I
CONFIGURATION OF OUR BENCHMARK

Potential	*L-J*	*Tersoff*
Units	lj	metal
Lattice	0.8442 FCC	5.431 Diamond
Cutoff	2.5	3.2
Skin	0.3	1.0
Timestep	0.005 tau	0.001 psec
Neigh_modify	20	5
Fix	NVE	NVE
Potential Specific Parameters	sigma: 1 epsilon: 1	Potential file: Si.tersoff

$fpair$ for j-atoms not in the cutoff can solve this problem.

For three-body interactions of short neighbors in the workflow of *Tersoff*, we notice that FORCE-ZETA in Algorithm 2 procedure is at the level of j-loop, so we choose to use this level for vectorization instead of the inner level k-loop as shown in Fig. 8, which can keep FORCE-ZETA vectorized. Parameter loading can be done which is almost the same as pairwise interactions.

F. Discussion of the Portability of Our Strategies

While our optimizations target TaihuLight, but some of our ideas are also portable to other platforms: 1) The implementation of software cache strategy can be abstracted as a general-purpose module. In order to use this module on a similar architecture, we only need to re-implement the memory accessing part by replacing the current DMA instructions with specific memory accessing instructions. 2) The hybrid memory updating strategy is also portable to shared memory architectures to avoid write conflicts. 3) Our customized transcendental functions can be abstracted to a frontend + backend framework. The frontend contains a series of math operations while the backend translates those operations to corresponding hardware instructions, so it can be ported to most architectures by implementing the backend.

IV. EVALUATION

We evaluate our optimized version of LAMMPS on TaihuLight. The single node performance, scalability and correctness are all evaluated, and performance result is acquired by calculating the average value of 5 runs. Also, as a comparison, we compare our optimization on a single node of TaihuLight to the reference implementation and Intel version of LAMMPS.

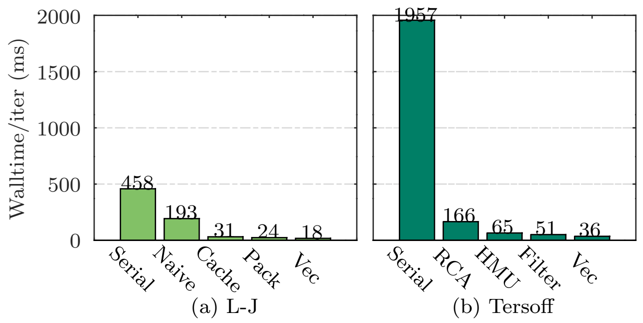


Fig. 9. The average iteration time of the two potentials among different optimization steps. RCA in Fig. 9(b) means *redundant computation approach* while HMU means *hybrid memory updating*.

TABLE II
THE TIME DECOMPOSITION OF OPTIMIZED *L-J* AND *Tersoff* POTENTIALS ON EACH SECTION IN PERCENTAGE.

Potential[1]	Section[2]				
	Force	Neigh	Comm	Modify	Other
Ref-MPE-*L-J*	83.34	11.68	1.29	3.33	0.35
Opt-CPE-*L-J*	52.13	20.39	16.72	4.92	5.84
Ref-MPE-*Tersoff*	96.96	1.12	0.65	1.15	0.12
Opt-CPE-*Tersoff*	64.08	14.95	13.81	3.29	3.88

[1] Ref version of code means pure MPE running reference code, including *L-J* and *Tersoff* potential. Opt version of code means CPEs running code for *L-J* and *Tersoff* potential.
[2] There are five sections in *L-J* and *Tersoff* potentials. Force is the computation of the potential. Neigh is the building the neighbour list. Comm is the exchanging of atoms migration and boundary atom information with neighbouring processors. The section Modify updates the integration steps for velocities and positions. Other is any remaining time.

Our optimization is based on the *11Aug17* stable version of LAMMPS. In order to compare results of TaihuLight with other platforms, the test cases are selected from a standard LAMMPS benchmark for the simulation. Detailed benchmark sets are shown in Table I. The potential file of *Tersoff* in Table I is Si.tersoff which is provided by the LAMMPS package.

A. Single Node Evaluation

As shown in Fig. 9(a), initially, the pairwise interactions kernel's performance is heavily dominated by memory access patterns, and we have an obvious speedup for this kernel after using the software cache method. The idea of packing some atomic data also provides a slight speedup since it can avoid a number of unaligned DMA access.

For the *Tersoff* potential, we can see that the HMU method is many times faster than RCA. And we also try to filter the F_end list on CPEs which will also gain a little speedup.

Vectorization also provides a slight speedup in *L-J* and *Tersoff* kernel. At last we gain a total memory access bandwidth of $\sim$ 112 GB/s for *L-J* and $\sim$ 98 GB/s for *Tersoff*, which almost reaches the peak memory bandwidth.

Table II shows the time decomposition of running the reference implementation and our optimized implementation. The parallelized neighbor list building and update procedure do not

TABLE III
THE CONFIGURATION OF EVALUATED VERSION OF LAMMPS.

Evaluation	Platform	Version[1]	Memroy	Scale
Ref-MPE	SW26010	Ref	4-channel DDR3	4 Core
Ref-Intel	E5-2650 v2	Ref	8-channel DDR3	4 Core
Opt-CPE	SW26010	CPE	4-channel DDR3	1 Node
Vec-KNL	Phi 7210	Vec	HBW (MCDRAM)	1 Node

[1] Which contains three types: Ref, CPE and Vec. Ref is the original code in the LAMMPS for MPE and Intel, Vec means the vectorized version on KNL by M.Höhnerbach et al.[10] and CPE is our optimized implementation.

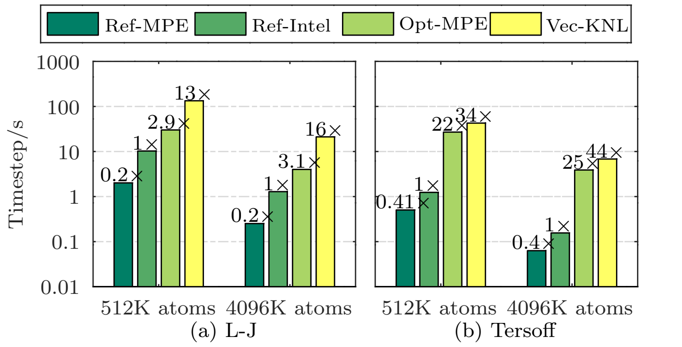


Fig. 10. Compare performance with other platforms, the configuration of each evaluation is shown in Table III. y axis is logarithmic scaled.

have a good speedup compared to the potential computation since they are more memory bounded.

B. Comparison to Other Implementations

We also compare the performance of our implementation to other popular LAMMPS implementations.

Table III shows the configuration of these evaluations, and test cases are described in the beginning of Sec.IV.

C. Single Node Evaluation

Fig. 10 gives a comparison of the evaluated versions, which shows an SW26010 processor can match $\sim$ 12 E5-2650 v2 cores for *L-J* potential and $\sim$ 100 for *Tersoff* potential.

As is shown, our implementation of *Tersoff* has a good performance, which is just a little slower than the KNL version. We take a look at our performance and KNL's. In the evaluation of *L-J* at 512 K atoms, our performance on an SW26010 processor is almost $1/4$ of KNL's. While the memory bandwidth is only $\sim$ 112 GB/s, which is close to the theoretical bandwidth of the SW26010 processor. For the KNL evaluations, they requires less than 8 GB of memory, so we can deduce that all data are binded in the MCDRAM with $\sim$ 400 GB/s bandwidth.

Even though we know that there is a number of memory access operations issued by software, we can not deduce the memory bandwidth requirement directly due to the cache. And it is easy to prove that if we know the cache missing rate, the memory access amount and the time to fulfill the memory access can be deduced using the following equations.

$$RAA = LAA \cdot MR \tag{7}$$

$$TTF = \frac{RAA}{BW} \tag{8}$$

TABLE IV
COMPARISON TO OUR KNOWN IMPLEMENTATIONS ON THE TAIHULIGHT

Source	Target		Speedup[1]	
	Software	Potential	Force	Overall
[24]	GROMACS	Non-bonded[2]	16	5
[4]	LAMMPS	*L-J*	24	8
Opt-CPE-*L-J*	LAMMPS	*L-J*	32	16
Opt-CPE-*Tersoff*	LAMMPS	*Tersoff*	68	62

[1] Which is the maximum reported CPE parallelized speedup over MPE in the paper, both the speedup of force kernel and overall iteration are collected.
[2] Which is the non-bonded potential of GROMACS, which may contain many components, e.g. combining *L-J* potential and *Coulomb Forces*.

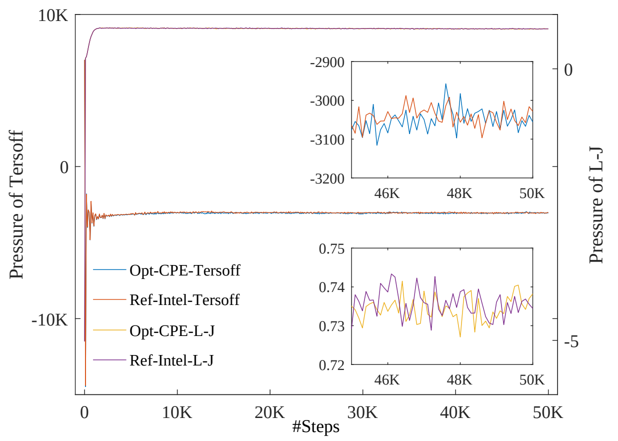


Fig. 11. Validation of pressure for 512 K atoms system with $50,000$-running steps for *L-J* and *Tersoff* potential. The results of Ref-Intel for *L-J* and *Tersoff* potential are taken as references with double precision. The Opt-CPE versions is our optimization with double precision. The subgraph in the northeast enlarges the final trend of pressure for *Tersoff* potential on CPE and Intel platforms. And the southeast subgraph indicates the final trend of *L-J* potential for CPE and Intel.

In Equation (7), RAA means real access amount, which is the real amount of data loaded/stored from/to memory. LAA means logical access amount, which is the amount of data access required by software. MR means cache missing ratio. In Equation (8), time to fulfill (TTF) the memory accesses is decided by RAA and bandwidth (BW).

By acquiring our MR with code injection and KNL's MR with VTune, we know that our MR is $\sim 18\%$ while KNL's MR is $\sim 10\%$. Now assume that we have the same LAA, then we have:

$$\frac{TTF_{SW}}{TTF_{KNL}} = \frac{\frac{LAA \cdot MR_{SW}}{BW_{SW}}}{\frac{LAA \cdot MR_{KNL}}{BW_{KNL}}} = \frac{MR_{SW} \cdot BW_{KNL}}{MR_{KNL} \cdot BW_{SW}} \approx 5.5 \tag{9}$$

From Equation (9) we can see that SW26010's TTF is about 5.5 times as many as KNL's. As we know that *L-J* is a kind of memory bounded kernel on our machine, thus we almost have done our best to accelerate memory accessing to gain the current performance.

Also, we have collected a set of previous work for porting MD on TaihuLight and compared our performance to them. We have compared the maximum speedup over an MPE, which are shown in Table IV.

To our best knowledge, we are the first to implement parallelized MD with effective speedup on the SW26010 processor to gain a more than 10 fold overall speedup.

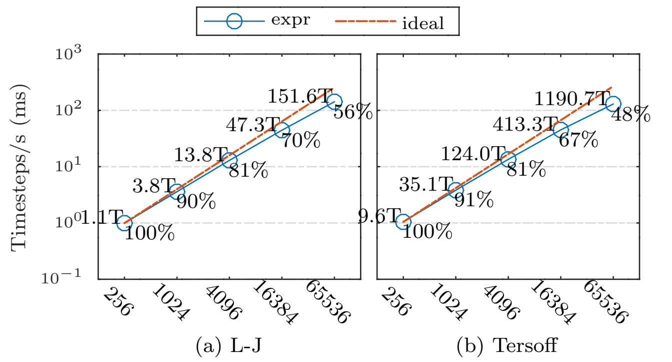


Fig. 12. Strong Scalability for 2^{30} atoms with x and y axes logarithmic scaled. The number below the line is the parallel efficiency and the number above is the Flops.

D. Accuracy

In order to validate the accuracy of our implementation, we test the pressure of the system in a 50, 000-running steps with 512 K atoms both for *Tersoff* and *L-J* potentials. The two subgraphs at the right of the Fig. 11 are the blown-up contrasts of the final pressures of last 5, 000 steps for *L-J* and *Tersoff* potentials on CPE and Intel platforms. Although there is a difference between the CPE and Intel results, they are stable enough to be similar in long time running steps.

E. Scalability

We also evaluate the scalability of our optimized version of LAMMPS. The parallel efficiency for the strong scalability and weak scalability are calculated using Equation (10) and Equation (11) separately. In these two equations, T_x is the execution time of x processes.

$$Eff_{strong}(N) = \frac{T_{256}}{\frac{N}{256} * T_N} \tag{10}$$

$$Eff_{weak}(N) = \frac{T_{16}}{T_N} \tag{11}$$

For the strong scalability, we use a case of $1,073,741,824$ (2^{30}) atoms, and take the performance of 256 processes as baseline. When the number of processes varies from 256 to $65,536$, the timesteps/s is shown in Fig. 12. When the number of atoms drops to $\frac{1}{256}$ of the number, the efficiency drops from 100% to $\sim$ 50%. It shows that *Tersoff* has worse efficiency than *L-J*, because *Tersoff* has more operations on ghost atoms, and the ratio of ghost atoms is increasing as the simulation box of one process is shrinking.

Fig. 13 shows the weak scaling performance. We initialize force fields with an average of $4,194,304$ atoms per process and take the performance of 16 processes as baseline. At last, it reaches $\sim$ 275 billions of atoms and achieves a peak performance of 2.43 PFlops (collected from sampling mode of hardware counters). It can be seen that as the number of atoms increases, as expected, it has almost linear speedup and finally reaches $65,536$ processes. This indicates that we have the ability to simulate large scale force field with large scale parallelism.

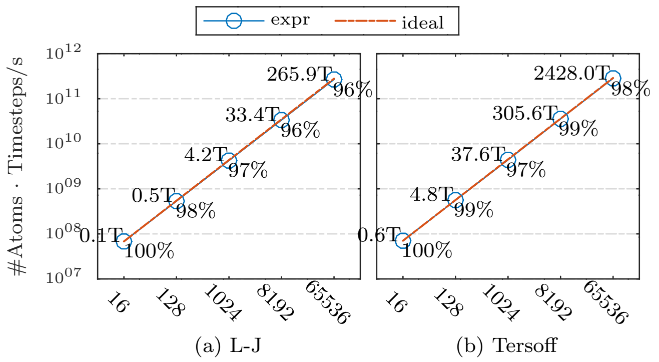


Fig. 13. Weak scalability for $4,194,304$ (2^{22}) atoms per process, with both x and y axes logarithmic scaled. The number below the line is the parallel efficiency and the number above is the Flops. In the computation of *Tersoff* potential on $65,536$ processes, we reach an performance of 2.43 PFlops.

V. Conclusion

In this paper, we present an efficient and scalable parallel implementation of LAMMPS for both *L-J* and *Tersoff* potentials on Sunway TaihuLight supercomputer and its fourth-generation SW26010 processor. We have employed a combination of optimization techniques to overcome the memory-bound and the compute-bound bottlenecks in LAMMPS for both pair and multi-body potentials. In particular, in order to break the memory constraints of SW26010, we have designed and implemented a number of memory access optimization strategies. Experiments show that our redesigned software on a single SW26010 processor can outperform over 100 E5-2650 v2 cores for running the latest stable release (11Aug17) of LAMMPS. We also achieve a performance of over 2.43 PFlops for a *Tersoff* simulation when using $16,384$ nodes on Sunway TaihuLight.

The optimization techniques presented in this paper are quite generic. And they can also be adapted to map similar applications such as Gromacs and NAMD onto the heterogeneous many-core cluster architecture of Sunway TaihuLight. Also, while this work is targeting TaihuLight and its SW26010 processor, we expect the presented methods to be portable to similar type of architectures, such as integrated heterogeneous many-core platforms like hUMA enabled APUs and the next generation ShenWei processor.

Acknowledgment

We would like to thank all the anonymous reviewers for their insightful comments and suggestions, also, Xun Zhang and Hongliang Sun from the Material Genome Institute of Shanghai University provided a number of suggestions for developing LAMMPS on the TaihuLight. This work is partially supported by the National Natural Science Foundation of China (Grant No. 61702297, 91530323, 41776010), National Key R&D Program of China (Grant No. 2016YFA0602100 and 2017YFA0604500), China Postdoctoral Science Foundation (Grant No. 2016M601031). The corresponding authors are Weiguo Liu (email: weiguo.liu@sdu.edu.cn), Wei Xue (email: xuewei@tsinghua.edu.cn), Haohuan Fu (email: haohuan@tsinghua.edu.cn), Dexun Chen (email: adch@263.net).

References

[1] J. A. Anderson, C. D. Lorenz, and A. Travesset. *General purpose molecular dynamics simulations fully implemented on graphics processing units*. Academic Press Professional, Inc., 2008.

[2] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. Gromacs: a message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1-3):43–56, 1995.

[3] D. A. Case, D. S. Cerutti, T. E. Cheatham, III, T. A. Darden, R. E. Duke, T. J. Giese, H. Gohlke, A. W. Goetz, and D. Greene. AMBER 10. AMBER 2017, University of California, San Francisco.

[4] W. Dong, L. Kang, Z. Quan, K. Li, K. Li, Z. Hao, and X. Xie. Implementing molecular dynamics simulation on sunway taihulight system. In *IEEE International Conference on High PERFORMANCE Computing and Communications; IEEE International Conference on Smart City; IEEE International Conference on Data Science and Systems*, pages 443–450, 2017.

[5] Z. Fan, W. Chen, V. Vierimaa, and A. Harju. Efficient molecular dynamics simulations with many-body potentials on graphics processing units. *Computer Physics Communications*, 218:10–16, 2017.

[6] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. Giampapa, Y. Zhestkov, M. C. Pitman, F. Suits, A. Grossfield, J. Pitera, et al. Blue matter: strong scaling of molecular dynamics on blue gene/l. *international conference on computational science*, pages 846–854, 2006.

[7] H. Fu, J. Liao, N. Ding, X. Duan, L. Gan, Y. Liang, X. Wang, J. Yang, Y. Zheng, W. Liu, et al. Redesigning cam-se for peta-scale climate modeling performance and ultra-high resolution on sunway taihulight. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1. ACM, 2017.

[8] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[9] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97–107, 2014.

[10] M. Höhnerbach, A. E. Ismail, and P. Bientinesi. The vectorization of the tersoff multi-body potential: an exercise in performance portability. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 69–81. IEEE, 2016.

[11] S. Kasap and K. Benkrid. Parallel processor design and implementation for molecular dynamics simulations on a fpga-based supercomputer. *Journal of Computers*, 7(6):1312–1328, 2012.

[12] Sudhir B. Kylasa, Hasan Metin Aktulga, and Ananth Y. Grama. Reactive molecular dynamics on massively parallel heterogeneous architectures. *IEEE Transactions on Parallel & Distributed Systems*, 28(1):202–214, 2017.

[13] J.E. Lennard-Jones and Cohesion. in: Proceedings of physical society,. pages 461–482, 1931.

[14] W. Mcdoniel, M. Hhnerbach, R. Canales, A. E. Ismail, and P. Bientinesi. Lammps pppm long-range solver for the second generation xeon phi. In *International Supercomputing Conference*, pages 61–78, 2017.

[15] T. D. Nguyen. Gpu-accelerated tersoff potentials for massively parallel molecular dynamics simulations. *Computer Physics Communications*, 212:113–122, 2017.

[16] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. V. Kalé, and K. Schulten. Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[17] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 36–36. IEEE, 2002.

[18] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[19] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C.Young, B. Batson, K. J. Bowers, J. C. Chao, et al. Anton, a special-purpose machine for molecular dynamics simulation. *international symposium on computer architecture*, 35(2):1–12, 2007.

[20] Jerry Tersoff. New empirical approach for the structure and energy of covalent systems. *Physical Review B*, 37(12):6991, 1988.

[21] N. Varini, N. J. English, and C. Trott. Molecular dynamics simulations of clathrate hydrates on specialised hardware platforms. *Energies*, 5(9):3526–3533, 2012.

[22] X. Wang, W. Xue, W. Liu, and L. Wu. swsptrsv: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 338–353. ACM, 2018.

[23] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[24] Y. Yu, H. An, J. Chen, W. Liang, Q. Xu, and Y. Chen. Pipelining computation and optimization strategies for scaling gromacs on the sunway many-core processor. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 18–32. Springer, 2017.

Appendix A
Artifact Description Appendix: [Redesigning LAMMPS for Peta-scale and Hundred-billion-atom Simulation on Sunway TaihuLight]

A. Abstract

Large-scale molecular dynamics (MD) simulations on supercomputers play an increasingly important role in many research areas. In this paper, we present our efforts on redesigning the widely used LAMMPS MD simulator for Sunway TaihuLight supercomputer and its ShenWei many-core architecture (SW26010). The memory constraints of SW26010 bring a number of new challenges for achieving efficient MD implementation on it. In order to overcome these constraints, we employ four levels of optimization: (1) a hybrid memory update strategy; (2) a software cache strategy; (3) customized transcendental math functions; and (4) a full pipeline acceleration. Furthermore, we redesign the code to enable all possible vectorization. Experiments show that our redesigned software on a single SW26010 processor can outperform over 100 E5-2650 v2 cores for running the latest stable release (11Aug17) of LAMMPS. We also achieve a performance of over 2.43 PFlops for a *Tersoff* simulation when using 16,384 nodes on Sunway TaihuLight.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** *Tersoff* potential and *L-J* potential.
- **Program:** LAMMPS.
- **Compilation:** Shenwei 5 Compiler Collection.
- **Transformations:** See MAKE/MACHINES/Makefile.sunway.
- **Data set:** LAMMPS provided benchmark or modified benchmark for TaihuLight.
- **Run-time environment:** Shenwei MPI
- **Hardware:** TaihuLight.
- **Execution:** By the job system.
- **Output:** Reported performance in LAMMPS.
- **Experiment workflow:** See below.
- **Publicly available?:** Yes.

2) How software can be obtained: The code of Sunway's version of LAMMPS is available at https://github.com/dxhisboy/lammps-sunway.

3) Hardware dependencies: Apply an account of NSCC Wuxi at http://www.nsccwx.cn/wxcyw/. Then the experimental queue can adopt most of small scale experiments.

4) Software dependencies: Sunway's version of GPTL and printing utils (available from the same repo).

5) Datasets: Original datasets for LAMMPS benchmark is in LAMMPS's benchmark directory.

Modified benchmark datasets for TaihuLight is in the directory of *SUNWAY-TESTS*.

C. Installation

Pull the source and replace it to the *11Aug17* version of LAMMPS. Use:

```
make sunway -j <njobs for compiling>
```

to compile the source code.

D. Experiment workflow

When the complation is done, there will be an binary file named *lmp_sunway*.

Submit it to the experimental job queue using:

```
bsub -I -b -cgsp 64 -share_size 6144 \
    -host_stack 16 -priv_size 4 \
    -p -m 1 -sw3runarg "-q" -n <nprocs> \
    ./lmp_sunway -sf sunway -var N off \
    -in <input script>
```

to start an experiment. **Caution:** each 4 processes share a node, so reproducing our result of 1 node requires submitting 4 processes to the job queue.

Some modified benchmark scripts is available in *SUNWAY-TESTS* directory for testing LAMMPS on TaihuLight work easier.

The *in.sunway.lj* and *in.sunway.tersoff* are the modified benchmark. The modification mainly contains variable controlled size of simulation box, output steps and running steps. Also, for the *Tersoff* potential, the neighbor list building is changed to constant frequency instead of checking for avoiding MPI_ALL_REDUCE in large scale simulations.

E. Evaluation and expected result

The expected thermo output should near the output of reference version of LAMMPS compiled on X86 processors.

The performance will be written by LAMMPS in console.

For the evaluation of 512K atoms, the first 10 steps of evaluations, *L-J* should have a Timestep/s of $\sim$ 49 while *Tersoff* should have a Timestep/s of $\sim$ 34.

For the long time evaluations, *L-J* should have a Timestep/s of $\sim$ 30 while *Tersoff* should have a Timestep/s of $\sim$ 26.

F. Experiment customization

Append *-var S 20* to the command line arguments for the simulation of 4,096K atoms.

G. Notes

Due to the limitation of memory size, It is not promised a simulation of over 4,096K atoms in one CG, and some memory allocation does not come with *fail-fast* feature, may be there will be unexpected result if a too large atoms/process defined for a simulation instead of an error message.

If there are conflicts between the github README file and this appendix, follow that README.