Exploring SIMD for Molecular Dynamics, Using Intel®Xeon®Processors and Intel®Xeon Phi™Coprocessors

S. J. Pennycook*, C. J. Hughes†, M. Smelyanskiy† and S. A. Jarvis*
*Department of Computer Science, University of Warwick, Coventry, UK
†Parallel Computing Lab, Intel Corporation, Santa Clara, CA
Email: sjp@dcs.warwick.ac.uk

***Abstract*—We analyse gather-scatter performance bottlenecks in molecular dynamics codes and the challenges that they pose for obtaining benefits from SIMD execution. This analysis informs a number of novel code-level and algorithmic improvements to Sandia's miniMD benchmark, which we demonstrate using three SIMD widths (128-, 256- and 512-bit). The applicability of these optimisations to wider SIMD is discussed, and we show that the conventional approach of exposing more parallelism through redundant computation is not necessarily best.**

In single precision, our optimised implementation is up to 5x faster than the original scalar code running on Intel®Xeon®processors with 256-bit SIMD, and adding a single Intel®Xeon Phi™coprocessor provides up to an additional 2x performance increase. These results demonstrate: (*i*) the importance of effective SIMD utilisation for molecular dynamics codes on current and future hardware; and (*ii*) the considerable performance increase afforded by the use of Intel®Xeon Phi™coprocessors for highly parallel workloads.

***Keywords*-scientific computing; accelerator architectures; parallel programming; performance analysis; high performance computing**

I. Introduction

Many modern processors are capable of exploiting data-level parallelism through the use of Single-Instruction-Multiple-Data (SIMD) execution. SIMD execution is a power-efficient way of boosting peak performance, and SIMD widths have been following an upward trend: the 128-bit Streaming SIMD Extensions (SSE) of x86 architectures have been augmented by 256-bit Advanced Vector Extensions (AVX); the new Intel®Many Integrated Core (MIC) architecture supports 512-bit SIMD; and GPUs typically execute "threads" in groups of 32 or 64 (1024- or 2048-bit SIMD, for single precision floating point). For the high-performance computing (HPC) industry, effective utilisation of SIMD on current hardware – and preparing for potentially wider SIMD in the future – is crucial.

However, mapping scientific codes to SIMD can be challenging. For example, molecular dynamics (MD) simulations are highly parallel but have key memory accesses to non-contiguous elements via gather and scatter operations, which can lead to poor performance. We use the SIMD instruction sets of Intel®Xeon®processors and Intel®Xeon Phi™coprocessors to explore the acceleration of MD codes through SIMD execution, with a focus on improving gather-scatter behaviour. We also provide full application runtimes for simulations executing on both architectures, to demonstrate the full impact of our optimisations.

The contributions of this paper are:

- We present an analysis of the gather-scatter bottlenecks in the short-range force calculation and neighbour list build functions of Sandia's miniMD benchmark [1], [2]. Together, these account for approximately 90% of its execution time.
- We describe, evaluate, and compare the performance of several SIMD implementations of these two functions using three SIMD widths (128-, 256- and 512-bit). We show that an Intel®Xeon®E5-2660 is 3.4x and 2.8x faster with 256-bit SIMD than with scalar execution, for force compute and neighbour list build, respectively. A Knights Corner Intel®Xeon Phi™coprocessor is 5.2x and 3.8x faster with 512-bit SIMD than with scalar execution.
- We discuss the applicability of our findings to wider SIMD, and explore the issue of achieving high utilisation of SIMD units. We show that the conventional approach of exposing more parallelism through redundant computation is not necessarily best.
- We outline design choices that enable the utilisation of servers comprising both Intel®Xeon®processors and Intel®Xeon Phi™coprocessors for MD simulations, namely: a spatial decomposition across CPU and MIC hardware; the use of Newton's third law (N3) between different threads, sockets and nodes in a system; and re-ordering atoms to facilitate the hiding of PCIe communication behind useful work.
- We demonstrate a methodology of sharing code between traditional CPU and MIC architectures, and hence a suitable development path for the acceleration of legacy HPC codes.
- We achieve up to a 5x speed-up over the original miniMD implementation running on the same Intel®Xeon®processor, highlighting the necessity

© 2013 IEEE

of performance tuning for traditional CPU architectures. We show that a single Intel® Xeon Phi™ coprocessor matches the performance of two oct-core Intel® Xeon® processors, and we achieve up to 93% efficiency when combining processor and coprocessor.
- Finally, we present an in-depth analysis of the performance of our optimised code running on Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors, including weak- and strong-scaling studies (within a node). These results provide insight into the speed-ups shown and give direction for future work.

The remainder of this paper is organised as follows: Section II provides a survey of related work; Section III offers a brief introduction to miniMD and the SIMD instruction sets of the hardware we use; Section IV explores the use of SIMD in MD simulations; Section V details a number of additional, non-SIMD improvements to miniMD, targeted at multi-threaded shared memory architectures; Section VI presents an extensive performance evaluation of the code; and finally Section VII concludes the work.

II. Related Work

The acceleration of MD simulations is the subject of much research, and many popular MD codes have already been rewritten to target SSE or the SIMD instruction sets of GPUs and other coprocessors [3]–[9]. Our work differs from these in that we draw a distinction between SIMD and threading – a line that is blurred by the CUDA and OpenCL programming models – and evaluate our proposed optimisations for multiple SIMD widths. We believe that these optimisations are applicable to other SIMD instruction sets, and should be of benefit to MD codes optimised for other hardware.

Other research has focused on improving the algorithmic complexity of MD. One common aim is to reduce the number of distance comparisons made during the computation of short-range forces; extensions to both Verlet's original method [10] of maintaining a list of interacting atom pairs [11]–[13] and the so-called "link-cell" method [14]–[16] have been proposed, along with new approaches [17]–[20] that make use of domain-specific knowledge to improve the search for near-neighbours. Improvements to communication complexity have also been investigated [21].

More hardware-focused optimisations have been considered, including: the potential of scheduling MD across heterogeneous systems featuring some mix of CPU and GPU cores [5], [22]; sorting atoms according to their position in space, to improve cache behaviour [4], [23]; and the use of novel hardware purpose-built for MD simulations [24], [25].

The use of single or "mixed precision" is common in GPU implementations [5], [6], as double precision is known to be at least 2x slower than single precision across GPU architectures. In [1], the authors allude to a performance study of miniMD in single precision on CPUs and note that there was "no appreciable performance enhancement." We verify that there is no significant performance benefit from using single precision in scalar code, since the application is not memory bound. However, as shown in this work, the use of single precision in SIMD can lead to significant performance improvements on CPUs.

III. Background

A. miniMD

miniMD is a simplified version of the popular LAMMPS package [26], [27], intended for use in optimisation studies such as this one; due to its simplicity, it is possible to explore potential optimisations much more rapidly. Despite supporting only the Lennard-Jones (LJ) inter-atomic potential, miniMD's performance and scaling behaviour is representative of much larger and more complex codes.

Simulation parameters such as the number of atoms (N), density (ρ), potential cut-off (R_c), skin distance (R_s), frequency of neighbour list rebuilds ($N_{rebuild}$) and initial temperature (T) are defined in a simple input file.

B. SIMD on Intel® Hardware

The instructions included in SSE (and its extensions, SSE2, SSE3 and SSE4) operate on 128 bits worth of integer or floating-point data (*e.g.* four 32-bit integer/single precision values, or two double precision values). In addition to mathematical operations, SSE provides instruction support for common "horizontal" operations such as permuting (or *shuffling*) the contents of a SIMD register. There is no instruction support for gathers or scatters, and loading scalar values into a SIMD register is expensive if they are not contiguous in memory.

The introduction of AVX increased the SIMD width from 128 to 256 bits (for floating-point), and the number of operands in each instruction from two (destructive source) to three (non-destructive source). 256-bit AVX instructions typically treat the lower and upper 128 bits of a SIMD register as independent, and this makes loading non-contiguous values into SIMD registers more expensive; two 128-bit values must be built and combined to form a 256-bit register. Intel® has announced AVX2, which includes support for gathers to help alleviate this cost and also adds 256-bit integer support. In this paper, we use AVX for both our 128- and 256-bit experiments, to better isolate the effects of SIMD width – any performance difference between 256-bit AVX and SSE will arise from a combination of increased SIMD width and reduced register pressure due to three-operand instructions.

The Knights Corner instructions (KCi) introduced in the first generation Intel® Xeon Phi™ coprocessor have a SIMD width of 512 bits, and include a number of instructions that do not have SSE or AVX equivalents. Gather and scatter

Table I
SYSTEM CONFIGURATION.

	Intel®Xeon®E5-2660	**KNC[a]**
Sockets×Cores×Threads	$2 \times 8 \times 2$	$1 \times 60 \times 4$
Clock (GHz)	2.2	1.3
Single Precision GFLOP/s	563	2496
L1 / L2 / L3 Cache (KB)	32 / 256 / 20,480	32 / 512 / -
DRAM	128 GB	4 GB GDDR
STREAM [28] Bandwidth	76 GB/s	150 GB/s
PCIe Bandwidth	6 GB/s	
Compiler Version	Intel®v13.0.030	
MPI Version	Intel®v4.0.3	

[a]Evaluation card only and not necessarily reflective of production card specifications.

instructions allow for the contents of SIMD registers to be loaded/stored from/to non-contiguous memory locations, and all SIMD instructions support conditional execution on individual elements based on the contents of a 16-bit mask. These additions greatly help with the vectorisation of scientific HPC codes – the memory-related instructions make it easier to populate SIMD registers, while masking makes it significantly cheaper to handle control divergence (*i.e.* different SIMD elements needing to take different directions at a branch).

C. Experimental Setup

The system configuration for the server used in our experiments is given in Table I. We use a Knights Corner (KNC) Intel®Xeon Phi™coprocessor, which has 60 x86 cores and hyper-threading support for four hardware threads per core. For area and power efficiency, KNC cores are less aggressive (*i.e.* have lower single-threaded instruction throughput) than Intel®Xeon®processor cores and run at a lower frequency. Hyper-threading helps to hide memory latency and multi-cycle instruction latency.

KNC is on a card that connects to the system via PCI-Express (PCIe) and which holds dedicated memory (GDDR). Communication between the CPU and KNC is therefore done explicitly through message passing, and communication between KNC and the node's network controller passes through the CPU. However, unlike many other coprocessors, KNC runs a full operating system. It also supports a shared memory model across all threads, and includes hardware cache coherence. Thus, in addition to common programming models for coprocessors (*e.g.* OpenCL), KNC supports more traditional multiprocessor programming models such as Pthreads and OpenMP.

The CPU and KNC binaries were compiled using the Intel®compiler[1] with the following flags: `-O3 -xHost -restrict -ipo -fno-alias`. In all experiments (except where noted), we use all of the available cores on the CPU and/or KNC, and each core runs the maximum number of hyper-threads supported (two per CPU core and four per KNC core). All experiments use a single node, but we aim to faithfully represent multi-node execution costs; messages that would be sent/received via MPI in a multi-node run are packed/unpacked appropriately, with the CPU effectively communicating with itself. The sending/receiving of messages for the KNC card is handled by the host CPU, with data forwarded on over PCIe.

Since we are interested primarily in the effects of SIMD, especially wide SIMD, all experiments use single precision. Although the use of "fast math" flags and instructions can boost performance, their cumulative effect on long-running simulations should ideally be examined by domain experts. Any performance numbers we report result from using exact floating-point math, and are on average 10–15% worse than when approximate reciprocals are employed.

To ensure that we start from a strong baseline implementation, we apply some previously proposed optimisations to miniMD before beginning our SIMD analysis. A number of these optimisations are already present in some form within LAMMPS: the aggregation of an atom's forces takes place in registers; atoms are sorted according to their position in space; and alignment is ensured via the insertion of padding into the array-of-structs (AoS) layout used to store atom data. We also adopt the neighbour list build algorithm proposed in [11], which is not currently present in LAMMPS.

IV. SIMD OPTIMISATIONS

Our SIMD analysis focuses on the two most expensive components of miniMD: the short-range force calculation and neighbour list build, which typically account for approximately 80% and 10% of execution time, respectively. We consider the SIMD acceleration of both components, since any performance improvement to either will make the other relatively more expensive. Further, the contribution of the neighbour list build to execution time is dependent upon the frequency of rebuilds, an input parameter that may be higher for other simulations.

For both components, we begin by examining whether the code is suitable for auto-vectorisation, and what speed-ups this delivers. We show that gathers and scatters, SIMD

[1]Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

```
1: for all atoms i do
2:     for all neighbours k do
3:         j = neighbour_list[k]
4:         delx = xi - pos[j+0]
5:         dely = yi - pos[j+1]
6:         delz = zi - pos[j+2]
7:         rsq = (delx × delx) + (dely × dely) + (delz × delz)
8:         if (rsq ≤ R_c) then
9:             sr2 = 1.0 / rsq
10:            sr6 = sr2 × sr2 × sr2
11:            f = sr6 × (sr6 - 0.5) × sr2
12:            fxi += f × delx
13:            fyi += f × dely
14:            fzi += f × delz
15:            force[j+0] -= f × delx
16:            force[j+1] -= f × dely
17:            force[j+2] -= f × delz
18:        end if
19:    end for
20: end for
```

Figure 1. Short-range force calculation.

memory accesses to non-contiguous elements, are key to achieving SIMD benefit for both components, and consider hand-vectorisation (with intrinsics) to improve gather-scatter behaviour.

A. Short-Range Force Calculation

The short-range force calculation loop (Figure 1) evaluates the force between all atoms separated by less than some "cut-off" distance R_c. To avoid evaluating the distance between all pairs of atoms at each time step, the force compute makes use of a pre-computed list of near neighbours for each atom.

1) Vectorisation: A common approach to the vectorisation of nested loops is to target the inner-most, which in this case is the loop over an atom's neighbours. Kim *et al.* demonstrate that the force compute loop from GROMACS can be auto-vectorised [29]; similarly, we find that the Intel®C++ compiler is able to auto-vectorise miniMD's loop with a little assistance. In particular, we must add a compiler directive (`#pragma ivdep`) to resolve the possible dependence in force array updates, since the compiler does not know that each of an atom's neighbours is unique.

The auto-vectorised code can be made more efficient by moving the force updates outside of the branch; otherwise, the compiler cannot know that the memory accesses involved are safe for iterations that fail the if-check. We also pad the number of neighbours to the nearest multiple of the SIMD width (W) using "dummy" neighbours – atoms placed at infinity that always fail the cut-off check – to handle situations where the number of neighbours is not divisible by W. Although the compiler still generates code to handle this case, it does not get executed at run-time.

After vectorisation across inner loop iterations, each of the arithmetic operations on Lines 4–17 operates at 100% SIMD efficiency (*e.g.* delx is computed for W neighbours in one instruction: $\{x_i, x_i, x_i, x_i\} - \{x_{j0}, x_{j1}, x_{j2}, x_{j3}\}$). However, the branch on Line 8 and the memory accesses on Lines 4–6 and 15–17 may introduce significant inefficiency.

The branch is handled via blending/masking; Lines 9–11 are executed for all neighbours, but the f value for neighbours that fail the cut-off is set to 0. The amount of inefficiency this causes depends upon the fraction of neighbours that fail the cut-off check, which is tied to neighbour list build frequency (a more frequent build gives a more accurate list). More fundamental to this loop, the memory accesses on Lines 4–6 (neighbour positions) and 15–17 (neighbour forces) are to potentially non-contiguous memory locations. Each of Lines 4–6 becomes a gather operation that reads W positions from the position array in AoS format, and packs them into a single SIMD register; this is known as AoS-to-SoA conversion, because we hold the position data as a struct-of-arrays (SoA) in registers. Each of Lines 15–17 becomes a gather operation, a subtract (the result of the multiplications can be re-used from Lines 12–14), and a scatter operation that writes the packed results back to memory (SoA-to-AoS conversion).

The auto-vectorised code is faster than the scalar code; on the CPU with AVX, it is 1.7x faster; on KNC with KCi, it is 1.3x faster. As discussed, we expect to see smaller speed-ups than W due to SIMD inefficiencies, but these results show that the inefficiency is quite high. One cause is the cut-off branch; in the default neighbour list, approximately 25% of neighbours do not fall within the cut-off, and padding the list to a multiple of W increases this inefficiency by 5–10%.

The primary culprit, however, is the gather and scatter operations, which implicitly transpose between AoS and SoA. If the gathers and scatters were as cheap as vector loads and stores, then we would see a significant speed-up – 7.04x (out of a maximum 8x) on the CPU using 256-bit SIMD. The remaining inefficiency is relatively small and, since it comes from the branch, is a trade-off with neighbour list build cost. The overhead of the gather and scatter operations lies not in the cost of memory accesses (as might be expected), but rather in instruction count; we next explore alternative SIMD approaches for handling these data transformations.

2) Gather-Scatter Overhead: Table II compares the number of clock cycles per neighbour (and speed-up over a scalar implementation on the same hardware) for a hand-vectorised loop with four different gather-scatter approaches.

Table II
CLOCK-CYCLES PER NEIGHBOUR AND SPEED-UP VERSUS SCALAR FOR FORCE COMPUTE GATHER-SCATTER APPROACHES.

Approach	CPU		KNC
	128-bit SIMD	256-bit SIMD	512-bit SIMD
Scalar L/S	12.97 (2.02x)	11.48 (2.28x)	23.75 (2.59x)
Scalar G/S	N/A	N/A	15.82 (3.89x)
Vector L/S + Shuffles	10.89 (2.40x)	8.02 (3.26x)	12.94 (4.75x)
Vector L/S + Dot Product	10.34 (2.53x)	7.64 (3.43x)	11.78 (5.22x)

Using scalar loads and stores (Scalar L/S) to popu-

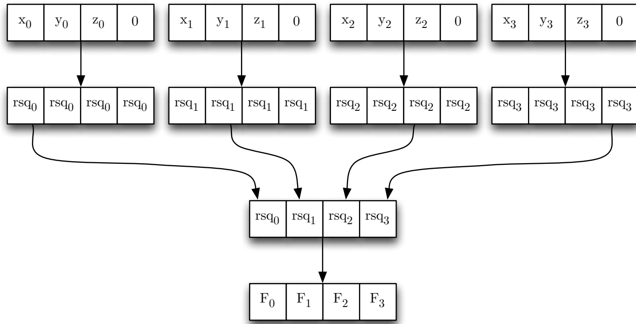


Figure 2. Combining a dot-product and transpose in 128-bit SIMD.

Table III
STATIC INSTRUCTIONS FOR FORCE COMPUTE.

	Scalar	128-bit	256-bit	512-bit
# Neighbours/Iteration	1	4	8	16
Load Neighbour IDs	1	4	8	16
Gather j Positions	0	4	8	16
Compute delx, dely, delz	3	4	4	4
Compute rsq	5	7	7	16
Compare rsq to R_c	2	2	2	1
Compute f $\times$ del*	9	14	14	7
Update i Forces	3	4	4	4
Gather/Scatter j Forces	6	8	16	32
Update j Forces	3	4	4	4
Other Instructions	6	3	3	52
Total Instructions/Neighbour	38.0	13.5	8.75	9.5

late SIMD registers (*i.e.* mimicking the compiler's auto-vectorisation) is at least 2x faster than scalar for all three SIMD widths. Using the dedicated gather and scatter instructions (Scalar G/S) on KNC is 1.5x faster still, demonstrating the utility of these new instructions. Our hand-vectorised implementations of this approach are significantly faster than the compiler's auto-vectorised code, since the compiler does not have domain knowledge and thus cannot make certain assumptions (*e.g.* all inputs being valid and not NaN).

Although considerably faster than scalar execution, using scalar gathers and scatters in this way is expensive: each scalar load or store requires an instruction, as does each insertion/extraction of a scalar to/from a SIMD register. Using the dedicated gather/scatter instructions on KNC reduces the required number of instructions, but not as much as we might hope. Each gather/scatter *instruction* accesses only a single cache line, reading/writing all required elements on that line. A single gather/scatter *operation* accesses W elements that may be spread across as many as W cache lines – to perform a full gather/scatter operation, we must place the instruction in a loop to ensure all elements are handled. Since we gather/scatter from/to AoS, the W elements accessed will definitely not be on the same cache line and each gather/scatter operation will therefore require several iterations of this loop.

We can decrease the instruction counts for these gather/scatter operations by taking advantage of the data being stored in AoS format: we can load/store an entire atom ($\{x, y, z, 0\}$) using one 128-bit instruction, and can replace the scalar insertion/extraction code with an in-register transpose. This approach (Vector L/S + Shuffles) improves performance by 1.2–1.4x.

Combining the calculation of rsq with the AoS-to-SoA transpose, as shown in Figure 2 (Vector L/S + Dot Product), leads to a small improvement in performance (5–10%). For 128-bit and 256-bit SIMD, this is achieved with a dot-product instruction; for 512-bit SIMD, it relies on KNC's ability to perform certain permutations (*swizzles*) of a SIMD operand for free during arithmetic operations. This approach may not be faster than a simple transpose on other architectures, which may lack support for dot-products or fast horizontal adds, but demonstrates that arithmetic should be combined with data movement wherever possible.

Table III presents a breakdown of the number of static instructions in the inner-most loop for our best approach. These instruction counts include arithmetic and unique data accesses, but ignore instructions introduced by the compiler due to hardware constraints (*e.g.* a finite number of registers) – these instructions are listed as *Other Instructions*. In general, arithmetic-dominated operations scale well, and the number of instructions is comparable across SIMD widths. Even following our optimisations, the number of instructions for gathers and scatters scales poorly with SIMD width; ignoring "Other Instructions", gathers and scatters account for 31%, 48% and 64% of the remaining instructions for 128-, 256- and 512-bit SIMD respectively. KNC has a high number of "Other Instructions" due primarily to register pressure on the general-purpose and mask registers; this may not manifest for another instruction set with the same or higher SIMD width.

To estimate the performance loss (in cycles) due to the high instruction overhead for gathers and scatters, we use 256-bit AVX to evaluate the performance of two "ideal" cases, where all of an atom's neighbours are contiguous in memory. With data still stored in AoS format, and thus still needing transposition, performance improves by 1.4x; with data stored in SoA, performance improves by 2x. This result demonstrates that there is still potential to improve gather-scatter behaviour further.

B. Neighbour List Build

miniMD uses a "link-cell" approach to reduce the size of the set of potential neighbours examined during the neighbour list build. Atoms are placed into subvolumes of space called "bins", using a spatial hash. For each atom, the set of potential neighbours can then be defined as those atoms that fall into a "stencil" of surrounding bins pre-computed at the start of the simulation.

The majority of the neighbour list build's execution time is spent in a loop (Figure 3) that runs after this binning process.

```
 1: for all atoms i do
 2:     numneigh[i] = 0
 3:     for all potential neighbours k do
 4:         j = potential_neighbour[k]
 5:         delx = xi - pos[j+0]
 6:         dely = yi - pos[j+1]
 7:         delz = zi - pos[j+2]
 8:         rsq = (delx × delx) + (dely × dely) + (delz × delz)
 9:         if (rsq ≤ Rc + Rs) then
10:             neighbour[i][numneigh[i]] = j
11:             numneigh[i]++
12:         end if
13:     end for
14: end for
```

Figure 3. Neighbour list build.

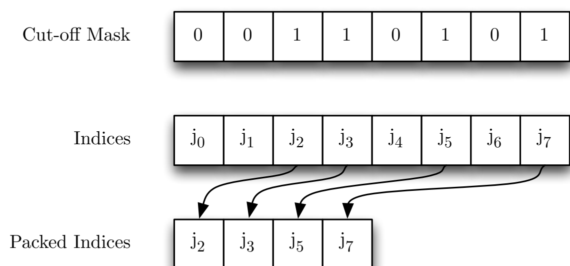


Figure 4. Using a packed store to append to the neighbour list.

For each atom, this loop iterates through the set of potential neighbours, storing in the neighbour list those which are closer than $R_c + R_s$. R_s is a "skin distance", chosen such that no atom can move by more than R_s between neighbour list builds.

1) Vectorisation: As before, our vectorisation targets the inner-most loop over neighbours. The core behaviour of this loop is very similar to that of the force compute – it computes the distance between two atoms and compares that distance to some cut-off. However, the loop does not auto-vectorise due to a loop dependence on Lines 10 and 11; the memory location to which each neighbour index should be written depends upon the number of previous neighbours that pass the cut-off check.

An efficient way to vectorise appending to a list in this manner is to use a *packed store*, the basic operation of which is demonstrated in Figure 4. For a SIMD register packed with rsq values, the result of a comparison with $R_c + R_s$ is a W-bit mask, and a packed store writes a subset of indices (from another SIMD register) to contiguous memory based upon this mask. KCi includes a packed store instruction, which we can emulate in other SIMD instruction sets; for both 128-bit and 256-bit AVX, we achieve it with a mask look-up, a single shuffle and a vector store. We determine the number of neighbours appended to the list by counting the number of bits set in the comparison mask.

The rest of the loop is vectorised in much the same way as the force compute loop: Lines 5–7 gather the positions of W neighbours, and the arithmetic operations on Lines 5–8 operate at 100% SIMD efficiency. The complete contents of the branch are handled by the packed store operation, and so no blending/masking is required. The loop does not update an atom's neighbours, and so there are no scatters – thus, besides the packed store, the major source of SIMD inefficiency is the gather of positions.

2) Gather-Scatter Overhead: As shown previously, this gather can be accelerated significantly by implementing an AoS-to-SoA transpose in intrinsics. For the force compute, each atom gathers a distinct set of neighbours, and therefore there is no opportunity to re-use any transposed data. This is not true of the neighbour list build, since all atoms within the same bin loop over the same stencil, and gather the same set of positions each time.

Surrounding the outer loop over atoms with a new loop over bins enables us to gather (and transpose) the set of potential neighbours once and then re-use it for several atoms. For the architectures considered here, this is beneficial for two reasons: first, the cost of the AoS-to-SoA transpose is amortised over several atoms; and second, the transposed set of neighbours exhibits better cache behaviour. We believe our approach to be applicable to GPU architectures also, since the transposed set could be stored in local memory.

The choice of bin size is an important trade-off: with large bins, the gathered SoA stencil receives more re-use but will contain more atoms; with small bins, the SoA stencil receives less re-use but also contains fewer atoms. The best choice depends on whether the cost of gathering atoms is more than that of extra distance calculations – the CPU favours smaller bins, whereas KNC favours larger. Besides this simple parameter change, the algorithm is the same across both architectures and consistently 2–3x faster than the conventional approach.

Table IV
STATIC INSTRUCTIONS FOR NEIGHBOUR LIST BUILD.

	Scalar	128-bit	256-bit	512-bit
# Neighbours/Iteration	1	4	8	16
Load Positions & Compute rsq	8	8	8	6
Compare rsq to $R_c + R_s$	2	1	1	1
Load Neighbour IDs	1	1	2	1
Append to Neighbour List	2	8	17	5
Other Instructions	3	5	5	15
Total Instructions/Neighbour	16.00	5.75	4.13	1.75

Table IV presents a breakdown of the number of static instructions in the inner-most loop for our optimised approach. As before, "Other Instructions" accounts for those introduced by the compiler due to hardware constraints. Since the data transpose happens outside of the key loop, the number of instructions to load positions and compute rsq remains constant across SIMD widths, except for 512-bit SIMD on KNC; KNC has fewer instructions here because it has fused multiply-add instructions, which eliminates two

arithmetic instructions. KNC has a higher number of "Other Instructions" per neighbour, but these are mostly software prefetches and mask manipulations (to handle iterations with fewer than 16 neighbours).

The number of instructions required to append to the neighbour list is the least consistent across architectures. Due to the lack of 256-bit integer support in AVX, our implementation uses 128-bit stores, and thus this operation does not scale with SIMD width. In contrast, KNC's cost for this operation is very low, due to its packed store instruction.

C. Effects of SIMD Width

The SIMD analysis thus far has assumed that the number of neighbours per atom is sufficiently large (compared to the SIMD width) that the amount of padding required to make the number of neighbours a multiple of W is small. This holds true for miniMD's default simulation (28 neighbours per atom) on the hardware we use, but architectures with wider SIMD, or simulations with small cut-off distances, require a different approach to achieve high efficiency.

There are typically thousands of atoms in an MD simulation, and thus moving to "cross-atom" SIMD (*i.e.* vectorising the loop over atoms) exposes significantly more parallelism than a "cross-neighbour" approach – and sufficient parallelism for many hardware generations to come. For the force compute loop, using SIMD in this fashion results in two changes compared to our previous code: (*i*) gathers and scatters are required in the outer loop, to accumulate forces into atoms, if we group non-contiguous atoms; and (*ii*) there is a potential for update conflicts, since several atoms may share a neighbour – if we attempt to update the same neighbour multiple times simultaneously, only one of the updates will take effect.

One way to address these update conflicts is to abandon the use of N3, removing the scatters of updated neighbour forces within the inner loop at the expense of evaluating each atom-pair twice. This scheme is common on GPUs [5], [6]. However, for all of the approaches explored in this paper, replacing the neighbour updates with redundant computation results in a slow-down. For the LJ benchmark using 256-bit SIMD on the CPU, this slow-down is only 24%, but the cost of redundant computation will be higher for more computationally expensive potentials.

In the absence of fast atomic gather-scatter operations [30], we propose that update conflicts be handled by arranging the neighbour lists such that there are no duplicate indices in any set of W neighbours. Detecting duplicates within a group of W neighbours requires $O(W^2)$ comparisons but, since comparison hardware typically scales with SIMD width, we expect the number of needed SIMD comparison instructions to be a more tractable $O(W)$.

Our algorithm for conflict resolution is as follows. For every set of W indices, check for a conflict (*i.e.* a duplicate index). If there are no conflicts, then this set of indices can be written to the neighbour list. If there are conflicts, we split the set into at most W subsets: one subset containing all of the indices that do not conflict, and up to $W - 1$ subsets containing one index each. In place of conflicting indices, we insert "dummy" neighbours (index 0) as padding, and store a bit-mask denoting their location in each subset. When all neighbour sets have been considered, subsets are matched based upon their masks, combined if possible, and then written to the neighbour list. Figure 5 demonstrates this process for 128-bit SIMD, for two sets of W indices with a single conflict each.

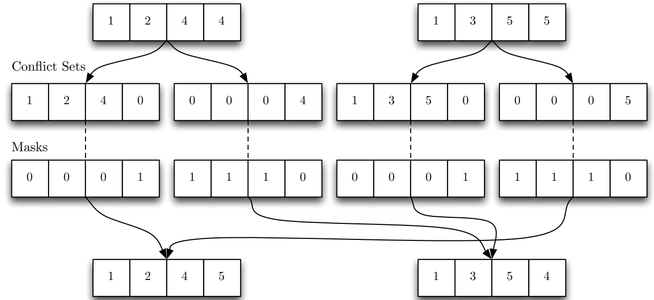


Figure 5. Resolving conflicts using 128-bit SIMD.

Table V
PERCENTAGE INCREASE IN NEIGHBOUR LIST SIZE (DUE TO PADDING).

	128-bit	**256-bit**	**512-bit**	**1024-bit**	**2048-bit**
Cross-Neighbour	3.19%	9.36%	18.21%	45.04%	71.60%
Cross-Atom (No CR)	1.44%	2.45%	4.51%	10.63%	26.28%
Cross-Atom (CR)	1.69%	3.50%	6.55%	13.74%	33.21%

Where a group of W atoms have a different number of neighbours, or it is impossible to combine two subsets, some padding will remain in the neighbour list. Table V compares the amount of padding introduced by: (*i*) cross-neighbour SIMD; (*ii*) cross-atom SIMD, before conflict resolution (No CR); and (*iii*) cross-atom SIMD, after conflict resolution (CR), for a simulation of 256k atoms using a cut-off of 2.5. These results show that the cross-atom approach is more efficient than the cross-neighbour approach for several SIMD widths, and that conflict resolution itself introduces little additional padding.

Since the operation of the proposed conflict resolution algorithm is orthogonal to that of the neighbour list algorithm proposed in Section IV-B, we implement it as a post-processing step that removes conflicts from an existing neighbour list. This step also: transposes the list (such that it stores the 0th neighbour of W atoms, followed by the 1st, and so on) to improve performance during force compute; and sorts "windows" of atoms according to their number of neighbours, to alleviate compute imbalance (a group of W atoms must process the maximum number of neighbours within the group).

The results in Table VI give the speed-up of cross-atom

Table VI
SPEED-UP OF CONFLICT RESOLUTION APPROACH.

	256k			2048k		
	1.5	**2.5**	**5.0**	**1.5**	**2.5**	**5.0**
Force Compute	1.37x	1.02x	0.95x	1.30x	1.02x	0.95x
Total Simulation	1.04x	0.94x	0.87x	1.04x	0.96x	0.87x

SIMD with conflict resolution (relative to our best cross-neighbour SIMD approach) on the CPU with 256-bit AVX. We include results for six different MD simulations: 256k and 2048k atoms, with R_c set to 1.5, 2.5 and 5.0 (giving an average of 6, 28 and 221 neighbours respectively). We see a slow-down for the larger two cut-off distances, since cut-off distances of 2.5 and 5.0 have sufficient cross-neighbour parallelism that we do not improve SIMD efficiency. For a cut-off of 1.5, we see a small speed-up (1.04x) for the complete application; the speed-up for force compute alone is more significant (1.37x), but its contribution to execution time is low for this particular simulation.

That a post-processing step such as this one can be performed efficiently on current hardware, and results in a speed-up where expected, demonstrates that it is a suitable method for handling SIMD update conflicts in MD simulations. However, MD implementations that use threading must also guarantee the independence of tasks allowed to execute simultaneously. Our conflict resolution approach could be used to guarantee independence beyond a single thread, but this is not practical for a many-core architecture; for KNC, we would need to resolve conflicts for 3840 parallel tasks (60 cores $\times$ 4 threads $\times$ 16 SIMD units). We address the issue of handling update conflicts between threads in the next section.

V. COPROCESSOR OPTIMISATIONS

Optimising an MD application for a hybrid system comprising a traditional CPU and coprocessor involves more challenges than making efficient use of SIMD. In this section, we detail a number of additional optimisations we have employed to maximise the performance of miniMD executing on a server containing both Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors.

A. Problem Decomposition

miniMD uses a spatial decomposition across MPI ranks (*i.e.* each rank is assigned a subvolume of space, and computes the forces for the atoms in that space). As atoms move during the simulation, they may move from one subvolume to another; furthermore, the forces acting on an atom may be dependent on atoms in a different subvolume. The main simulation loop thus contains two communication steps. First, before the short-range force calculation, all processors exchange position information for atoms near their subvolume boundaries with neighbouring processors. The receiving processor allocates space for these *ghost* atoms. Second, after the force calculation, all processors must send some force contributions back to the "owner" of each ghost.

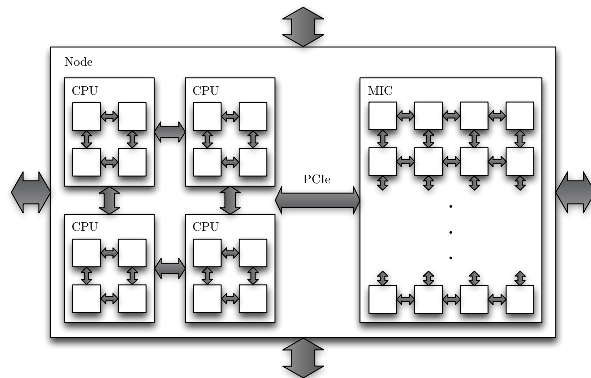


Figure 6. The hardware/class hierarchy.

We augment this MPI decomposition with a hierarchy of subdomains, as shown in Figure 6. At the first level, we divide the problem domain amongst *nodes*, and each node runs a single MPI task. We then further subdivide a node's subdomain amongst *sockets* (where a socket is either a CPU socket or a KNC socket/card), and finally we split each socket's subdomain amongst some number of *threads*. We specify the fraction of a node's subvolume assigned to the KNC hardware at run-time.

The use of such a hierarchy allows for communication between subdomains to be specialised: threads running on the same socket can communicate directly through shared memory; threads running on different sockets can communicate either through shared memory or over PCIe; and all threads can pack their off-node communication into a single MPI message rather than competing for the network interface. Using a spatial decomposition at each level allows us to use ghost atoms to handle the update conflicts between threads, and helps to minimise the size of messages sent between the CPU and KNC.

B. PCIe Bandwidth/Latency

A possible bottleneck for KNC performance is the latency and bandwidth of the PCIe bus. To minimise the amount of PCIe communication, we adopt the same communication mechanism as [5] and opt not to use N3 between different sockets. Although this results in a small amount of redundant computation (for those atom-pairs that cross socket boundaries), it reduces the amount of PCIe communication by a factor of two since we can skip sending force contributions back to the "owner" socket of each atom.

We further optimise PCIe communication by overlapping it with useful work. Our decision to not use N3 across sockets means that we need only hide a single exchange of messages between the CPU and KNC for each iteration of the simulation loop (sending position updates for atoms

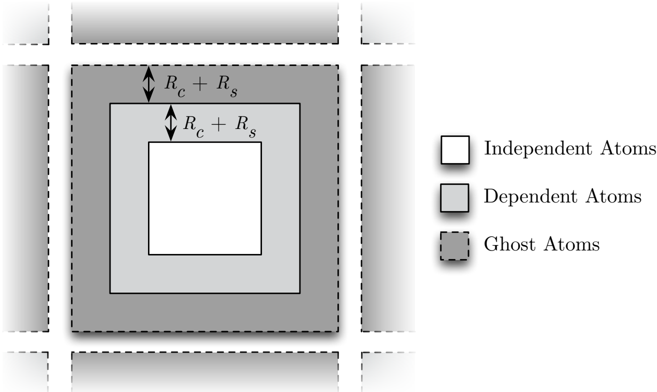


Figure 7. A subdomain split into dependent and independent volumes.

used by neighbour sockets) unless we need to rebuild the neighbour lists. To facilitate this, we divide atoms into the three types shown in Figure 7: those that interact only with other atoms in the same subdomain (*independent* atoms); those that potentially interact with atoms in another subdomain (*dependent* atoms); and those that are copies of atoms in another subdomain (*ghost* atoms). We can compute the forces for all atom-pairs not featuring ghost atoms without any cross-domain communication; therefore, we overlap PCIe communication with this computation, and hide it almost completely.

C. Code Re-use

The arrangement of subdomains described in Section V-A is represented in code as a hierarchy of abstract C++ classes: an MPI_Domain, a Socket_Domain and a Thread_Domain. These abstract classes contain all of the common functionality across CPU and KNC hardware, and make up the bulk of the code. Where different code is required (*e.g.* for performance reasons, or because of differing communication mechanisms), this is implemented in a subclass. This minimises the amount of code that must be re-written for optimisation on a particular architecture.

Specifically, around 60% of the code base for our optimised implementation of miniMD is shared between Intel®Xeon®processors and Intel®Xeon Phi™coprocessors. The evaluation hardware used in this study did not support the use of MPI between CPU and KNC, and the majority of the remaining code differs due to our use of a low-level interface for PCIe communication. Only 5–10% (the force compute and neighbour list kernels) is written in platform-specific intrinsics, and these sections of the code are in fact very similar (as shown in Figure 8); algorithmically, they are identical, but each instruction set uses a distinct set of intrinsics.

```
// Compute square distance (w/ dot product).
del1 = _mm_sub_ps(xi, xj);
rsq1 = _mm_dp_ps(del1, del1, 0x71);

// Calculate 1/rsq^3
sr2 = _mm_div_ps(_mm_set1_ps(1.0f), rsq);
sr6 = _mm_mul_ps(sr2, _mm_mul_ps(sr2, sr2));
```

(a) SSE

```
// Calculate square distance (w/ dot product).
del1 = _mm256_sub_ps(xi, xj);
rsq1 = _mm256_dp_ps(del1, del1, 0x71);

// Calculate 1/rsq^3
sr2 = _mm256_div_ps(_mm256_set1_ps(1.0f), rsq);
sr6 = _mm256_mul_ps(sr2, _mm256_mul_ps(sr2, sr2));
```

(b) AVX

```
// Calculate square distance (w/o dot product).
del1 = _mm512_sub_ps(xi, xj);
delsq = _mm512_mul_ps(del1, del1);
rsq1 = _mm512_add_ps(delsq,
        _mm512_swizzle_r32(delsq, _MM_SWIZ_REG_BADC));
rsq1 = _mm512_add_ps(rsq1,
        _mm512_swizzle_r32(rsq1, _MM_SWIZ_REG_CDAB));
rsq = _mm512_mask_add_ps(rsq, mask_AAAA, rsq, rsq1);

// Calculate 1/rsq^3
sr2 = _mm512_rcp23_ps(rsq);
sr6 = _mm512_mul_ps(sr2, _mm512_mul_ps(sr2, sr2));
```

(c) KCi

Figure 8. Comparison of platform-specific intrinsics for force compute.

VI. Results

To demonstrate the performance and scalability of our optimised code[2], we present results for simulations with multiple atom counts and two different cut-off distances (2.5 and 5.0). The first of these is the standard cut-off distance used in miniMD's LJ benchmark, whereas the second is used to investigate the effects of inter-atomic potentials with larger cut-off distances. This approach matches that of [5], and the number of neighbours per atom under these conditions is similar to that of the LAMMPS Rhodopsin protein benchmark. All experiments use cross-neighbour SIMD, since both cut-off distances provide sufficient parallelism.

We fix the other simulation parameters as follows: $\rho = 0.8442$, $T = 1.44$, $N_{rebuild} = 20$, $R_s = 0.3$, timesteps $= 100$. We report performance in atom-steps per second (*i.e.* (# atoms $\times$ timesteps)$/$execution time), and execution time in seconds. We repeated experiments 10 times, and report the average (mean). Although we report results here from single precision runs, early results from our double and mixed precision implementations suggest that the performance impact is what one would expect and is similar to that reported for GPU codes (*i.e.* that the double precision code is twice as slow, and the mixed precision code somewhere between).

[2]Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Configurations: Refer to Table I. For more information go to http://www.intel.com/performance.

Table VII
PERFORMANCE SUMMARY FOR 2.048M ATOMS.

	Orig.	**CPU (AVX)**	**KNC**	**CPU+KNC**
Cut-off of 2.5				
Atom-steps/s	20.2M	62.3M	85.6M	131.0M
Time (s)	10.14	3.29	2.39	1.56
Cut-off of 5.0				
Atom-steps/s	2.90M	13.5M	19.2M	30.5M
Time (s)	70.71	15.14	10.65	6.72

Table VIII
PERFORMANCE BREAKDOWN FOR 2.048M ATOMS.

Comp.	**Orig.**	**CPU (AVX)**	**KNC**	**CPU+KNC (CPU)**	**CPU+KNC (KNC)**
Cut-off of 2.5					
Force %	82.0	63.9	63.2	55.8	62.4
NL %	9.0	13.5	12.6	10.7	12.1
Comm %	4.9	9.5	15.4	26.5	17.7
Other %	4.0	13.1	8.7	7.1	7.8
Cut-off of 5.0					
Force %	90.3	86.0	85.4	78.9	80.5
NL %	6.7	6.9	6.4	6.2	6.7
Comm %	1.7	4.1	6.3	13.3	11.0
Other %	0.4	3.0	2.0	1.7	1.9

Throughout this section, we use the term "communication" to cover all costs incurred as a result of sending or receiving a message over a given channel: the packing and unpacking of message buffers, waiting for messages to arrive and the transfer of the message itself. Also throughout this section, all experiments involving the CPU use 256-bit AVX.

A. Performance Summary

Table VII reports the performance of a 2.048M atom simulation for: (*i*) the original miniMD; (*ii*) our implementation using AVX; (*iii*) KNC running alone; and (*iv*) using the CPU and KNC together (CPU+KNC), where the CPU uses AVX.

Running on the CPU with AVX, our version of miniMD achieves speed-ups of 3.1x and 4.7x over the original miniMD for the cut-offs of 2.5 and 5.0, respectively; KNC provides further speed-ups of 1.4x in both cases. CPU+KNC execution gives speed-ups over the original miniMD of 6.5x and 10.5x, and speed-ups over our version of miniMD on the CPU of 2.1x and 2.3x. The CPU+KNC execution efficiencies (*i.e.* CPU+KNC performance divided by the sum of CPU and KNC performance) are 89% and 93%.

KNC has a peak floating-point rate over four times that of the two CPU sockets used for these experiments (Table I), but achieves only 1.4x higher performance. For force compute, the CPU has a significant per-thread advantage over KNC, requiring fewer cycles per neighbour (Table II) and running at 1.7x the frequency; that KNC is faster overall is due to its much larger number of threads.

The per-thread performance of KNC is worse because many operations are either not implemented in SIMD, or do not achieve 100% SIMD efficiency due to being dominated by gathers/scatters. These problems also apply to CPUs, but their effects are more prominent on KNC due to its wider SIMD. Further, MD (particularly the LJ potential) is not dominated by fused multiply-adds (FMAs), which leads to reduced utilisation of KNC's SIMD arithmetic hardware – every regular addition, subtraction or multiplication wastes 50% of the compute capability. The CPUs do not have FMA hardware, but where the number of additions/subtractions and multiplications is not balanced, we also waste compute capability. KNC threads (like those of other coprocessors) are also more sensitive to exposed cache or memory latency due to the simplicity of KNC cores – cache misses that cannot be hidden by other threads on the same core are more expensive. This is particularly problematic during gather/scatter operations, since the access pattern is too unpredictable to be captured by a hardware prefetcher and the overhead of software prefetching is too high.

B. Performance Breakdown

Table VIII gives a breakdown of a 2.048M atom simulation into four components: short-range force calculation (*Force*); building the neighbour lists (*NL*); communication (*Comm*), including force/position communication and the exchanging of atoms across subvolume boundaries; and any remaining time (*Other*), comprising the integration steps for computing updated velocities and positions. The CPU+KNC breakdown is given from the perspective of the CPU and KNC in separate columns. For the original miniMD, *Force* consumes the vast majority of time, followed by *NL*. *Comm* is the next most expensive, and *Other* is the smallest component.

For our versions of miniMD, *Force* remains the largest component of execution time for both cut-offs. However, as the component that is most accelerated by our use of SIMD, it takes a smaller fraction of time. For the AVX implementation, we see speed-ups of 4.0x and 4.9x for the cut-offs of 2.5 and 5.0 respectively, and KNC provides an additional speed-up of 1.4x in both cases. We see a larger speed-up for the larger cut-off for two reasons: firstly, the time spent in force compute is dependent upon the number of inter-atomic distances that must be evaluated, which grows with the cube of the cut-off; and secondly, due to our use of SIMD across neighbours, SIMD efficiency is improved.

Our SIMD acceleration of the neighbour list construction improves its performance considerably; thus, the contribution of *NL* to execution time remains relatively constant. For the AVX implementation, we see speed-ups of 2.1x and 4.6x for the cut-offs of 2.5 and 5.0 respectively, and KNC provides an additional speed-up of 1.5x in both cases. One might expect this component of the simulation to become relatively more expensive for larger cut-offs (as with the force compute), since it also depends upon the number of atom-pairs. However, although the distance computation

costs scale similarly, a larger cut-off results in more atoms per bin and therefore significantly lowers looping overheads.

As we increase the number of threads for a fixed problem size, the fraction of time spent in *Comm* increases. This is due to a larger number of dependent atoms, which leads to higher inter-thread communication costs; we must exchange partial force and position information (through shared memory) for more atoms in each iteration. KNC uses significantly more threads than the CPU and thus spends a larger fraction of time in inter-thread communication. Further, it spends time in PCIe communication. Although this cost is mostly hidden for the exchanging of updated atom positions every iteration, it remains exposed when moving atoms between nodes and for the first exchange of position information after a neighbour list build. These factors lead to KNC spending 19% and 7% more time in communication than the AVX implementation on the CPU. For CPU+KNC, we see that the CPU spends a much larger fraction of its time in *Comm* than when running without KNC; this increase is due to handling PCIe communication not present in CPU-only runs.

The fraction of time spent in *Other* is larger in our versions of miniMD than in the original, since it benefits least from the use of SIMD and threading. On the CPU, the position/velocity updates scale very poorly due to limited memory bandwidth – these operations involve very little computation per atom and require streaming through multiple large arrays that do not fit in the on-die caches (*e.g.* for 2.048M atoms, the velocity update touches 48 bytes per atom – a total of $\approx$ 98MB). As noted in Table I, KNC's effective memory bandwidth is twice that of the CPU, and this is reflected in its performance for this operation – KNC is 2.1x and 2.2x faster than the CPU for the two cut-offs.

C. Thread Scaling

Figure 9 shows the execution times for the original miniMD and our implementation when weak-scaled, with a cut-off of 2.5. We made every effort to ensure that the number of atoms per core remained ≈32K (in line with the LAMMPS benchmark) and that the total problem volume remained a cube. For AVX, execution time grows by 24% from 1 to 16 cores; for KNC, it grows by 24% from 1 to 60 cores. Scaling is better for a cut-off of 5.0, due to the larger fraction of time spent in *Force* and *NL*: for AVX, the execution time grows by 6% from 1 to 16 cores; and for KNC, it grows by 12% from 1 to 60 cores. The original miniMD scales slightly better in both cases due to its much worse overall performance. Its execution time grows by 15% and 5% going from 1 to 16 cores, for the cut-offs of 2.5 and 5.0 respectively.

Figure 10 shows the execution times for the original miniMD and our implementation when strong-scaled on a 1.372M atom problem, with a cut-off of 2.5. The original code achieves a 14x speed-up on 16 cores, while our AVX implementation achieves only 12x. This is due to the significant speed-up we see for *Force* and *NL*; *Comm* and *Other* do not scale as well and are relatively more expensive. KNC achieves only a 40x speed-up on 60 cores for the same reason – *Force*, *NL*, *Comm* and *Other* see speed-ups of 52x, 41x, 7x and 14x respectively. A cut-off of 5.0 leads to much better parallel efficiency, and our implementation achieves a 14x speed-up on 16 CPU cores. We see a 50x speed-up on 60 KNC cores – *Force*, *NL*, *Comm* and *Other* see speed-ups of 55x, 45x, 6x and 14x respectively.

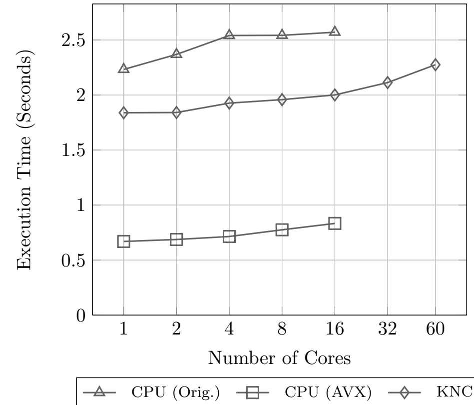


Figure 9. Weak-scaling results for a cut-off of 2.5.

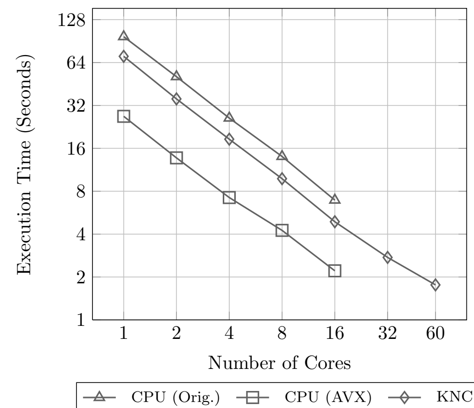
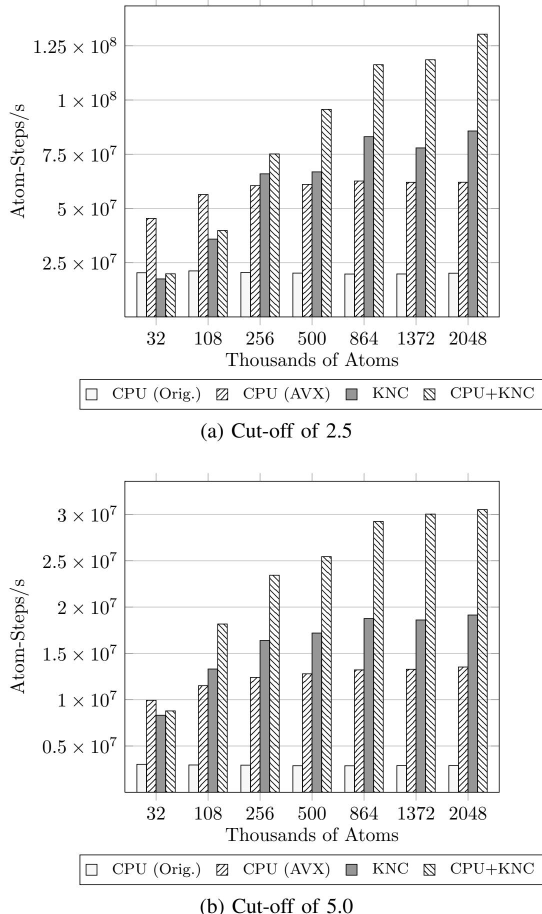


Figure 10. Strong-scaling results for a cut-off of 2.5. Note that the y-axis is scaled logarithmically.

D. Performance Comparison

Figure 11 compares the absolute performance (in atom-steps/s) of our implementation with that of the original miniMD, for different problem sizes. The performance gap between traditional architectures and coprocessors is often exaggerated [31]–[34], but we avoid this by: (*i*) comparing against an optimised CPU code, to avoid inflating claims about KNC's performance; (*ii*) drawing comparisons between codes of the same floating-point precision; and (*iii*) reporting full application times, rather than focusing on kernels that may be more amenable to acceleration.

For the problem sizes shown, the performance of miniMD for a given cut-off distance is almost constant – atom density



(a) Cut-off of 2.5

(b) Cut-off of 5.0

Figure 11. Absolute performance in atom-steps/s (higher is better).

is fixed, and thus the computational cost *per atom* remains the same across problem sizes. Our implementations, however, improve as problem size increases, because smaller problems are dominated by inter-thread communication costs. For very small atom counts (≈ 4K) miniMD exhibits the same behaviour; for some simulations, it is quicker to run on less than the maximum number of cores (all numbers in the graphs are for the maximum number of cores). CPU performance starts to level off at 256K atoms, while KNC and especially CPU+KNC see better performance from even larger simulations since KNC has more threads.

Our optimised AVX code is consistently faster than the original scalar implementation, although the gains grow with problem size, as already described. For a cut-off of 2.5, it is up to 4x faster, and for a cut-off of 5.0 up to 5x faster – a difference that can be attributed to the increased amount of parallelism in problems with higher cut-off distances. One takeaway from this result, besides the effectiveness of our particular optimisations, is the need to revisit and re-tune CPU code when investigating the utility of coprocessors.

The KNC and CPU+KNC implementations can both provide significantly higher performance than the CPU alone; KNC by itself is up to 1.42x faster than the CPU, and CPU+KNC is up to 2.26x faster. The coprocessor requires sufficient work to be helpful, improving performance starting at 256K atoms for a cut-off of 2.5, and at 108K atoms for a cut-off of 5.0. Real-world implementations utilising KNC should thus take problem size into account when choosing the number of threads and cores to use, to avoid degrading performance on small problems.

VII. Conclusions

We present an analysis of the vectorisation of MD codes, and demonstrate that gathers/scatters are one of the key bottlenecks. We detail efficient implementations of the neighbour list build and short-range force calculation loops that scale with both SIMD width and number of threads, and propose an alternative method of resolving N3 write-conflicts. On current Intel® hardware, this conflict-resolution approach boosts short-range force calculation performance by almost 40% for simulations with small cut-off distances; its applicability to future architectures (with wider SIMD) remains to be demonstrated in future work.

We compare the performance of our optimised implementation to that of the original miniMD benchmark, and show it to be consistently faster (by up to 5x on the same hardware and up to 10x with the addition of an Intel® Xeon Phi™ coprocessor) for a range of problem sizes and cut-off distances. This considerable performance increase highlights the need to tune codes and ensure that SIMD is being used effectively. For problems with a large amount of exploitable parallelism, we show that KNC is up to 1.4x faster than a dual-socket, oct-core Intel® Xeon® E5-2660 server.

Although specialised for MD, we believe that the techniques that we describe are applicable to other classes of applications. Other codes featuring gather-scatter memory access patterns (*e.g.* unstructured mesh) could benefit from similar SIMD optimisations, while our methodology of sharing code and computational work between the CPU and KNC could be utilised by codes solving other spatially decomposed problems (*e.g.* computational fluid dynamics).

The high single-node performance of our implementation risks exposing inter-node latency/bandwidth, and the inter-thread communication costs for some simulations are already significant. While these costs are arguably exaggerated by our use of a potential with low arithmetic intensity, minimising and hiding communication costs is clearly a critical direction for future research.

Acknowledgements

The authors would like to thank Steve Plimpton, Paul Crozier and Simon Hammond of Sandia National Laboratories for their help in using miniMD and LAMMPS; and Steven Wright for his assistance in preparing diagrams.

This research is supported in part by The Royal Society through their Industry Fellowship Scheme (IF090020/AM).

References

[1] M. A. Heroux *et al.*, "Improving Performance via Mini-applications," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2009-5574, 2009.

[2] "Mantevo," http://software.sandia.gov/mantevo/, May 2011.

[3] S. Olivier, J. Prins, J. Derby, and K. Vu, "Porting the GROMACS Molecular Dynamics Code to the Cell Processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2007.

[4] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units," *Journal of Computer Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[5] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing Molecular Dynamics on Hybrid High Performance Computers – Short Range Forces," *Computer Physics Communications*, vol. 182, pp. 898–911, 2011.

[6] C. R. Trott, "LAMMPScuda - A New GPU-Accelerated Molecular Dynamics Simulations Package and its Application to Ion-Conducting Glasses," Ph.D. dissertation, Ilmenau University of Technology, 2011.

[7] D. Case *et al.*, "AMBER," http://www.ambermd.org, 2011.

[8] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008, pp. 1–9.

[9] M. J. Harvey, G. Giupponi, and G. D. Fabritiis, "ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale," *Journal of Chemical Theory and Computation*, vol. 5, no. 6, pp. 1632–1639, 2009.

[10] L. Verlet, "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules," *Physical Review*, vol. 159, no. 1, pp. 98–103, 1967.

[11] T. N. Heinz and P. H. Hünenberger, "A Fast Pairlist-Construction Algorithm for Molecular Simulations Under Periodic Boundary Conditions," *Journal of Computational Chemistry*, vol. 25, no. 12, pp. 1474–1486, 2004.

[12] T. Maximova and C. Keasar, "A Novel Algorithm for Non-Bonded-List Updating in Molecular Simulations," *Journal of Computational Biology*, vol. 13, no. 5, pp. 1041–1048, 2006.

[13] P. Gonnet, "Pairwise Verlet Lists: Combining Cell Lists and Verlet Lists to Improve Memory Locality and Parallelism," *Journal of Computational Chemistry*, vol. 33, no. 1, pp. 76–81, 2012.

[14] W. Mattson and B. M. Rice, "Near-Neighbor Calculations Using a Modified Cell-Linked List Method," *Computer Physics Communications*, vol. 119, pp. 135–148, 1999.

[15] Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng, "Improved Neighbor List Algorithm in Molecular Simulations Using Cell Decomposition and Data Sorting Method," *Computer Physics Communications*, vol. 161, pp. 27–35, 2004.

[16] P. Gonnet, "A Simple Algorithm to Accelerate the Computation of Non-Bonded Interactions in Cell-Based Molecular Dynamics Simulations," *Journal of Computational Chemistry*, vol. 28, no. 2, pp. 570–573, 2007.

[17] D. R. Mason, "Faster Neighbor List Generation Using a Novel Lattice Vector Representation," *Computer Physics Communications*, vol. 170, pp. 31–41, 2005.

[18] Q. F. Fang, R. Wang, and C. S. Liu, "Movable Hash Algorithm for Search of the Neighbor Atoms in Molecular Dynamics Simulation," *Computational Materials Science*, vol. 24, pp. 453–456, 2002.

[19] R. J. Petrella, I. Andricioaei, B. R. Brooks, and M. Karplus, "An Improved Method for Nonbonded List Generation: Rapid Determination of Near-Neighbor Pairs," *Journal of Computational Chemistry*, vol. 24, no. 2, pp. 222–231, 2003.

[20] S. Artemova, S. Grudinin, and S. Redon, "A Comparison of Neighbor Search Algorithms for Large Rigid Molecules," *Journal of Computational Chemistry*, vol. 32, no. 13, pp. 2865–2877, 2011.

[21] D. E. Shaw, "A Fast, Scalable Method for the Parallel Evaluation of Distance-Limited Pairwise Particle Interactions," *Journal of Computational Chemistry*, vol. 26, no. 13, pp. 1318–1328, 2005.

[22] S. S. Hampton, S. R. Alam, P. S. Crozier, and P. K. Agarwal, "Optimal Utilization of Heterogeneous Resources for Biomolecular Simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[23] S. Meloni, M. Rosati, and L. Colombo, "Efficient Particle Labeling in Atomistic Simulations," *Journal of Chemical Physics*, vol. 126, no. 12, 2007.

[24] R. O. Dror *et al.*, "Exploiting 162-Nanosecond End-to-End Communication Latency on Anton," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–12.

[25] T. Narumi *et al.*, "A 55 TFLOPS Simulation of Amyloid-Forming Peptides from Yeast Prion Sup35 with the Special-Purpose Computer System MDGRAPE-3," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2006, pp. 1–13.

[26] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.

[27] S. Plimpton *et al.*, "LAMMPS Molecular Dynamics Simulator," http://lammps.sandia.gov/, May 2011.

[28] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[29] S. Kim and H. Han, "Efficient SIMD Code Generation for Irregular Kernels," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, 25-29 February 2012, pp. 55–64.

[30] S. Kumar *et al.*, "Atomic Vector Operations on Chip Multiprocessors," in *Proceedings of the International Symposium on Computer Architecture*, 2008, pp. 441–452.

[31] R. Bordawekar, U. Bondhugula, and R. Rao, "Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application!" IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC24982, 2010.

[32] ——, "Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU," IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC25033, 2010.

[33] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. E. Guney, and A. Shringarpure, "On the Limits of GPU Acceleration," in *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2010.

[34] V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, 2010, pp. 451–460.