Developing Performance-Portable Molecular Dynamics Kernels in OpenCL

S. J. Pennycook, S. A. Jarvis
Department of Computer Science,
University of Warwick,
Coventry, UK
e-mail: sjp@dcs.warwick.ac.uk

***Abstract*—This paper investigates the development of a molecular dynamics code that is highly portable between architectures. Using OpenCL, we develop an implementation of Sandia's miniMD benchmark that achieves good levels of performance across a wide range of hardware: CPUs, discrete GPUs and integrated GPUs.**

We demonstrate that the performance bottlenecks of miniMD's short-range force calculation kernel are the same across these architectures, and detail a number of platform-agnostic optimisations that improve its performance by at least 2x on all hardware considered. Our complete code is shown to be 1.7x faster than the original miniMD, and at most 2x slower than implementations individually hand-tuned for a specific architecture.

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

I. Introduction

In recent years, CPU alternatives have gained significant traction within the high-performance computing (HPC) industry, and many of the world's fastest supercomputers [1] currently employ hybrid system designs that pair traditional CPUs with discrete co-processors (*e.g.* Cell, FPGAs, GPUs, Intel Xeon Phi). Some vendors are already placing CPU and GPU cores on the same chip (AMD Fusion, Intel HD Graphics), while others are exploring this option (Nvidia Project Denver).

Optimising codes for each new hardware offering is clearly desirable, ensuring fair comparisons between them [2]–[5], and enabling HPC sites to switch between vendors. However, creating and maintaining highly optimised implementations for each platform is often not feasible – legacy production codes are too large to re-write using multiple programming languages/toolkits, and maximising performance on a new platform requires significant programmer effort and expertise.

The Open Computing Language (OpenCL [6]) is a cross-vendor standard which ensures the *functional* portability of codes across hardware, thereby eliminating the need for applications to be re-coded on a per-device basis. However, it makes no guarantees of *performance* portability – an OpenCL application may be highly tuned for a particular architecture, but exhibit very different performance on others.

This paper investigates the development of a "performance-portable" OpenCL implementation of Sandia's miniMD benchmark [7], [8], which seeks to exhibit good levels of performance across multiple architectures. miniMD is a simplified version of the popular LAMMPS package [9], [10], intended for use in optimisation studies such as this one; due to its simplicity, potential optimisations can be explored much more rapidly than in the context of a production application.

Although this is not the first paper to address the issue of performance portability for OpenCL codes [11]–[15], it is the first (to our knowledge) to do so for molecular dynamics (MD). The use of co-processors in this domain has been shown in previous research to deliver significant speed-ups [16]–[22]. However, it remains to be seen if a single set of optimisations allows for efficient execution across CPUs, discrete GPUs, and integrated GPUs.

The contributions of this paper are as follows:

- We detail the development of a performance-portable kernel for short-range force calculation, highlighting a number of important design decisions: the storage format of the neighbour list, the use of Newton's third law (N3), and using vector types to improve both gather/scatter performance and SIMD efficiency.
- We report the performance of our kernel executing on a wide range of hardware from multiple vendors, including: CPUs (Intel E3-1240; AMD A8-3850), discrete GPUs (Nvidia C1060, C2050, GTX 680; AMD FirePro V7800) and integrated GPUs (Intel HD4000; AMD HD6550D). Our optimisations improve performance by at least 2x across all architectures considered.
- We compare the performance of kernels auto-vectorised by the Intel OpenCL compiler to hand-vectorised kernels, and identify issues in the compilers from both AMD and Intel that result in the generation of inefficient assembly code for gather/scatter operations.
- Finally, we compare the performance of our complete application to the original miniMD, a hand-vectorised implementation using AVX instructions, and the CUDA kernel found within LAMMPS. Our OpenCL implementation is 1.7x faster than the original, and at most 2x slower than highly-optimised "native" implementations.

The remainder of this paper is structured as follows: Sec-

tion II presents a survey of related work; Section III introduces the OpenCL programming model; Section IV details the development and optimisation of our performance-portable MD code; Section V compares the performance of our code with implementations highly optimised for a specific platform; Section VI discusses the implications of our results; and Section VII concludes the work.

II. Related Work

A. Performance Portability

In previous work [15], we investigated the performance portability of OpenCL kernels in the context of wavefront applications, demonstrating that the performance of OpenCL on CPUs can match that of a scalar, MPI-based implementation. This work extends our approach to a new class of application (molecular dynamics), with a significantly different memory access pattern (gather-scatter), and focuses on the efficient utilisation of modern single-instruction-multiple-data (SIMD) hardware.

Others have examined the performance portability of OpenCL kernels from different domains [11]–[14]. In all cases, the authors conclude that "auto-tuning" (*i.e.* writing kernels in a parametrised fashion and determining optimal parameter values at runtime) is an appropriate method of achieving portability across hardware. This methodology is well-suited to OpenCL; library calls are provided for obtaining detailed hardware information, and kernels can be compiled with different parameter values at run time.

B. OpenCL and Molecular Dynamics

The most similar MD codes to the implementation presented here are the $\text{LAMMPS}_{\text{GPU}}$ package [18] and the MD benchmark from the Scalable HeterOgeneous Computing (SHOC) suite, both of which can be compiled to use OpenCL. They also both carry out a simple Lennard-Jones (LJ) simulation, although with different parameters and initial conditions.

SHOC is intended to be run across multiple architectures, but its input is sufficiently different from the default LAMMPS and miniMD simulations that its performance is not necessarily representative. Our OpenCL implementation does not build on either of these existing implementations – a serial, scalar baseline is more representative of a legacy application and will better enable us to identify the performance benefits of each individual optimisation.

III. OpenCL

The OpenCL standard [6] is maintained by the Khronos Group, but is contributed to by many vendors and institutions. At the time of writing, the language is supported on the majority of HPC hardware, although compilers are at different stages of maturity. Owing to its similarity to Nvidia's proprietary CUDA programming model, the mapping between OpenCL and GPU hardware is quite straightforward; the best way to map OpenCL to CPU hardware is less clear.

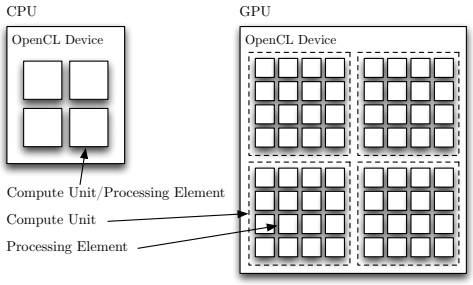

Figure 1. Mapping between CPU/GPU hardware and OpenCL.

A. Programming Model

In OpenCL, a *host* CPU running C/C++ code uses library calls to communicate with and control one or more *devices*. Each device is made up of a number of *compute units*, which can be further sub-divided into *processing elements*.

Device functions are known as *kernels* and can be compiled just-in-time (JIT) from source, or loaded from a cached binary if one exists for the current target platform. These kernels are executed in a single-program-multiple-data (SPMD) fashion by a one-, two- or three-dimensional set of *work-items*, which are grouped together into *work-groups*.

Work-groups can be scheduled for execution on any available compute unit and in any order, thus permitting OpenCL applications to scale to fit different hardware configurations. There is no method of global synchronisation during a kernel, but local synchronisation is possible between all work-items in a given work-group.

B. Hardware Mapping

How the concepts of *devices*, *compute units*, *processing elements*, *work-groups* and *work-items* map to different architectures is decided by the OpenCL runtime, a library that is often provided by the hardware vendor (but can also be supplied by others [23], [24]). A typical mapping between CPU/GPU hardware and OpenCL is shown in Fig. 1.

All of the CPU cores in a node are grouped into a single device. For CPUs with hyperthreading support, each hyperthread appears as a compute unit (*e.g.* the E3-1240 used in this work has 4 cores, but 2-way hyperthreading, and thus 8 compute units). The AMD and Intel runtimes expose the SIMD architectures of modern CPUs differently; the AMD SDK generates SSE/AVX code only if OpenCL's vector types (*e.g.* `float4`, `float8`) are used explicitly, whereas the Intel SDK attempts to auto-vectorise kernels by packing contiguous work-items into SIMD execution units.

GPUs from different vendors are divided into compute units and processing elements in different ways: on Intel GPUs, each execution unit is a compute unit of 8 processing elements; on Nvidia GPUs, each stream multiprocessor (compute unit) consists of 8, 32 or 192 "CUDA cores" (processing elements) on the Tesla, Fermi and Kepler architectures respectively; and on AMD GPUs, a compute unit

Table I
HARDWARE AND SOFTWARE CONFIGURATION.

	CPUs		Integrated GPUs		Discrete GPUs			
	E3-1240	**A8-3850**	**HD4000**	**HD6550D**	**C1060**	**C2050**	**GTX680**	**V7800**
Manufacturer	Intel	AMD	Intel	AMD	Nvidia	Nvidia	Nvidia	AMD
Compute Units	8	4	16	5	30	14	8	18
Proc. Elements	8	4	128	400	240	448	1536	1440
Peak[a] GFLOP/s	211	93	147	480	933	1288	3090	2016
Bandwidth (GB/s)	21	30	26	30	102	144	192	128
Power[b] (Watts)	80	100	77	100	189	238	195	150
Host Compiler	Intel 11.1.072							
Host Flags	`-O3 -xHost -restrict -fno-alias -ipo`							
MPI Library	OpenMPI 1.4.3							
Intel SDK	Intel OpenCL 2012							
AMD SDK	AMD APP 2.7							
NVIDIA SDK	CUDA Toolkit 5.0							
OpenCL Flags	`-cl-single-precision-constant -cl-mad-enable -cl-fast-relaxed-math`							

[a]Peak performance for single precision.
[b]TDP for "fused" architectures includes CPU and GPU power.

```
1: for all atoms i do
2:     for all neighbours k do
3:         j = neighbour_list_for_atom_i[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 2. Scalar short-range force calculation.

contains 16 stream cores of 5 simple processing elements each. Work-groups are scheduled to compute units, and processing elements execute work-items in 8-, 32- and 64-wide SIMD on Intel, Nvidia and AMD GPUs respectively.

Since GPUs rely on a CPU host, communication with the host can impact application performance. Integrated GPUs can communicate with the host through shared memory, but all data to be used by discrete GPUs must be transferred explicitly to the device via PCI-Express (PCIe).

C. Experimental Setup

A detailed description of the hardware and software setup used in this paper is given in Table I. The only exceptions to this setup are: the HD4000, which can only be used as an OpenCL device under Windows; and the A8-3850 CPU (plus its integrated HD6550D GPU), which is hosted elsewhere. On these two systems, we use the Visual Studio 2010 C++ compiler and GCC respectively.

Although the Intel i7-3770 (Ivy Bridge) CPU attached to the HD4000 can also be used as an OpenCL device, we do not report on its performance. We instead report performance on an Intel Xeon E3-1240, a server part built on a closely related microarchitecture (Sandy Bridge).

In all experiments, we round the number of work-items up to a multiple of the hardware's "preferred work-group size multiple", as reported by the device, and allow the OpenCL runtime to choose the number of work-groups.

All performance figures are given as execution times in seconds, and all implementations make use of single precision floating-point arithmetic. We configure miniMD to simulate 256,000 atoms with a uniform density of 0.8442, using the default cut-off distance of 2.5, for 100 timesteps. Except where noted, all experiments pair hardware with the OpenCL SDK provided by the vendor.

IV. DEVELOPING PERFORMANCE-PORTABLE KERNELS

The LJ simulation performed by miniMD can be broken down into four components. The calculation of short-range forces is responsible for more than 80% of its runtime, and 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 "neighbour list" [25] for each atom. Building this list is the second-most expensive step, accounting for approximately 10% of execution time. The remaining time is split between: inter-node communication, which sees nodes exchange position and force information for atoms near the boundary of their sub-volume; and the update of atom velocities and positions.

In this section, we discuss the challenges of achieving high performance portability for all four components of the simulation. However, our optimisation efforts focus exclusively on short-range force calculation, since it accounts for such a high fraction of execution time. As we demonstrate in Section V, the other components of the simulation may become a significant bottleneck on some hardware; there is a clear need to investigate these issues further in future work.

A. Short-Range Force Calculation

Fig. 2 shows pseudo-code for the scalar short-range force calculation loop in miniMD. Most of the work in this loop could potentially be executed in parallel; the positions of atoms are fixed during this operation, and thus the inter-atomic distance (and force) between all pairs of atoms in the system can be computed concurrently. A natural way to map this work to the OpenCL programming model is to assign one work-item to each atom.

This loop makes use of N3 – the force i exerts on j has the same magnitude as the force j exerts on i, but in the opposite direction. A given atom-pair (i, j) appears in the neighbour list for either i or j, and the force between them is calculated only once. This avoids redundant computation of forces, but introduces a potential for update conflicts (Lines 15–17), since several atoms may share a neighbour – if we attempt to update the same neighbour multiple times simultaneously, only one update will take effect.

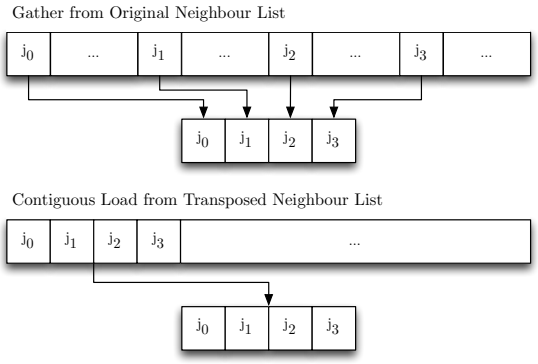

Figure 3. Effect of neighbour list transposition on memory access pattern. Each arrow represents a single load instruction.

Since OpenCL does not support global synchronisation between work-items, it is difficult to prevent such conflicts. Although many devices support atomic memory operations, on current-generation hardware they are too expensive to be used in an inner-loop such as this one [18]. We instead opt to not use N3, modifying the neighbour list build to store (i, j) in the lists of both i and j, and removing Lines 15–17 from the kernel. This removes the potential for conflicts, at the expense of twice as much computation.

1) Performance Bottlenecks: If contiguous work-items are mapped to SIMD execution units (as is the case on GPUs, and for auto-vectorised code on CPUs), then Lines 3–6 have the biggest performance impact. Each of the memory accesses on these lines requires a *gather* operation (*i.e.* they are "uncoalesced memory accesses", in CUDA parlance). If work-items instead/also expose parallelism through the explicit use of vector types (as is the case on AMD GPUs and for non-vectorised code on CPUs), then the use of only scalar types results in significant SIMD inefficiency.

Improving the performance of this kernel on all of the architectures considered in this study thus depends upon achieving better utilisation of SIMD hardware. As we demonstrate in the remainder of this section, all hardware therefore benefits from the same set of optimisations.

2) Neighbour List Gather: If the neighbour list of each atom is stored in contiguous memory, any access to this list from work-items operating in SIMD requires a gather; with n neighbours per atom, W contiguous work-items read from memory with a stride of n.

A much more efficient way to store the neighbour list is in transposed form (*i.e.* the 0th neighbour of W atoms, followed by the 1st neighbour, and so on). Where these W atoms have a different number of neighbours, we insert "dummy" neighbours, which are located at infinity and always fail the cut-off check. This is likely to give better performance than a completely transposed neighbour list (*i.e.* the 0th neighbour of all atoms, followed by the 1st neighbour, and so on) on architectures with caches.

Table II
PERFORMANCE IMPACT OF NEIGHBOUR LIST TRANSPOSITION.

Device	No Transpose (Seconds)	Transpose (Seconds)	Speed-up
Intel Xeon E3-1240	3.34	2.67	1.25x
AMD A8-3850	7.13	7.13	1.00x
Intel HD4000	5.58	2.76	2.02x
AMD HD6550D	9.33	1.89	4.95x
Nvidia Tesla C1060	3.68	2.91	1.26x
Nvidia Tesla C2050	2.52	0.86	2.92x
Nvidia GTX 680	2.49	1.04	2.40x
AMD FirePro V7800	1.95	0.35	5.63x

```
__kernel force_compute(float4* pos, float4* force...) {
    int i = get_global_id(0);
    float4 posi = pos[i];
    float4 fi = (float4) (0.0, 0.0, 0.0, 0.0);
    for (k = 0; k < number_of_neighbours[i]; k++) {
        int j = neighbour_list_for_atom_i[k];
        float delx = posi.x - pos[j].x;
        float dely = posi.y - pos[j].y;
        float delz = posi.z - pos[j].z;
        ...
    }
    force[i] = fi;
}
```

Figure 4. OpenCL code-snippet for a kernel storing atom data in `float4`s.

When W is set to the hardware's SIMD width, all neighbour list accesses become a single load from contiguous memory locations (Fig. 3). We therefore set W to 1 for the AMD CPU; 8 for the Intel CPU and GPU; 32 for Nvidia GPUs; and 64 for AMD GPUs.

This transposition also enables the Intel SDK to auto-vectorise our kernel, which is only attempted when the compiler expects it to improve performance (based on some heuristic). We believe that the presence of a large number of gather operations caused the original kernel to fail this check, thus preventing auto-vectorisation.

Table II gives the execution times for the kernel with and without a transposed neighbour list. The A8-3850 is the only architecture not to see a speed-up, and this is due to it executing work-items in scalar; when W is set to 1, the neighbour list layout is not changed. The cost of a gather operation increases with SIMD width, and we therefore see higher speed-ups on architectures with wider SIMD.

3) Position Gather: The loads on Lines 4–6 become three gather operations; one for each of x, y and z. For W contiguous work-items, this equates to $3W$ scalar loads, from up to W cache lines. Defining the position and force arrays using vector types (Fig. 4) serves as a hint to the compiler that x, y and z are stored contiguously; for all of the hardware we consider, the full position of an atom can be loaded in a single memory access, and this hint permits the compiler to employ a more efficient gather (Fig. 5).

On Nvidia GPUs, the accesses to x, y and z become coalesced and the Intel compiler is able to emit a more efficient sequence of instructions – an array-of-structs (AoS) to struct-of-arrays (SoA) transpose, in place of a series of scalar loads – on both CPU and integrated GPU hardware.

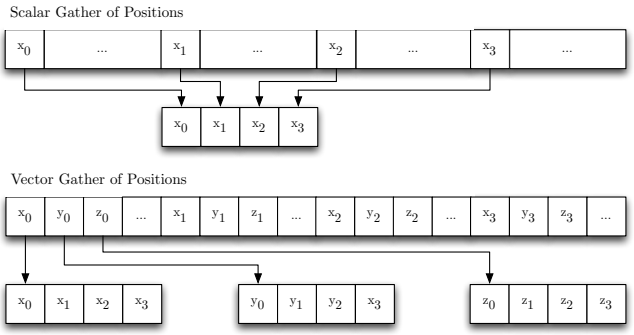

Figure 5. Effect of using vector types on memory access pattern.

Table III
PERFORMANCE IMPACT OF USING VECTOR TYPES.

Device	Scalar Gather (Seconds)	Vector Gather (Seconds)	Speed-up
Intel Xeon E3-1240	2.67	2.00	1.33x
AMD A8-3850	7.13	8.62	0.83x
Intel HD4000	2.76	1.59	1.74x
AMD HD6550D	1.89	1.88	1.00x
Nvidia Tesla C1060	2.91	1.54	1.89x
Nvidia Tesla C2050	0.86	0.67	1.29x
Nvidia GTX 680	1.04	0.45	2.29x
AMD FirePro V7800	0.35	0.34	1.02x

As shown in Table III, most architectures benefit from this change, with only AMD hardware seeing little (or no) improvement. Performance is not significantly impacted on the HD6550D or V7800, suggesting that they are already able to hide the latency of these memory accesses. The A8-3850 experiences a slow-down of 1.2x, again because it does not execute work-items in SIMD. Although the SDK uses a vector load to gather x, y and z as expected, the remainder of the kernel continues to use scalar arithmetic; therefore, each component of the loaded vector must be extracted into a scalar register, resulting in worse performance.

4) Explicit Vectorisation: After converting the position array to an array of `float4` vectors, our next step is to convert the arithmetic to vector form (Fig. 6). This exposes intra-loop parallelism to the compiler, and leads to a more succinct (and arguably more readable) version of the kernel.

Following this change, the Intel SDK no longer performs auto-vectorisation; rather than packing contiguous work-items into SIMD execution units, each of the explicit vector operations in the kernel is mapped directly to an SSE instruction. Since each work-item now runs independently, the neighbour list access is no longer a gather, and we disable the neighbour list transposition (by setting W to 1, as we did on the AMD CPU) to compensate.

The performance of this kernel on the E3-1240 is worse than those the compiler was able to auto-vectorise, due to SIMD inefficiencies: our explicit vectorisation only uses four of the eight SIMD execution units available in AVX (Fig. 7); and the majority of the arithmetic (the calculation

```
__kernel force_compute(float4* pos, float4* force...) {
    int i = get_global_id(0);
    float4 posi = pos[i];
    float4 fi = (float4) (0.0, 0.0, 0.0, 0.0);
    for (k = 0; k < number_of_neighbours[i]; k++) {
        int j = neighbour_list_for_atom_i[w*k];
        float4 del = posi - pos[j];
        float rsq = dot(del, del);
        float sr2 = 1.0 / rsq;
        float sr6 = sr2 * sr2 * sr2;
        float f = sr6 * (sr6 - 0.5) * sr2;
        f = (rsq < cutoff) ? f : 0.0;
        fi += del * f;
    }
    force[i] = fi;
}
```

Figure 6. OpenCL code-snippet for a kernel using vector arithmetic.

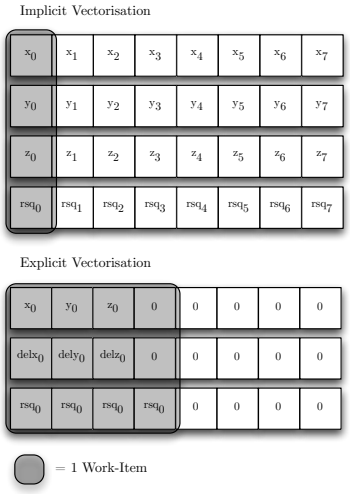

Figure 7. Comparison of implicit and explicit vectorisation, for 8-wide SIMD.

of the force, based on rsq) remains scalar. On the A8-3850, performance is improved by 1.3x, since this is the first kernel for which the AMD SDK emits any vector instructions.

On the GPUs, the mapping of work-items to SIMD units is unchanged, and we make no changes to the neighbour list transposition. We would thus expect to see no performance impact; the compilers convert each of the vector operations in the kernel into a set of four scalar operations, and then combine contiguous work-items as before. However, we see slightly worse performance across all GPU hardware, due to our use of the dot product function to calculate rsq. This introduces two additional instructions per loop iteration; the compiler does not know that the last element of the `float4` is 0, and so includes it in the dot-product.

5) Loop Unrolling: Unrolling the loop over neighbours in our explicit vectorisation kernel, such that it computes the force between i and several neighbours simultaneously, allows us to regain lost SIMD efficiency. Theoretically, the compilers could employ such an optimisation themselves, but we found that this is not yet the case.

Table IV lists the speed-ups for two alternative unrolling

Table IV
PERFORMANCE IMPACT OF UNROLLING.

Device	4-way	8-way
Intel Xeon E3-1240	1.42x	1.44x
AMD A8-3850	2.58x	2.36x
Intel HD4000	0.91x	0.87x
AMD HD6550D	1.32x	1.86x
Nvidia Tesla C1060	0.91x	0.92x
Nvidia Tesla C2050	1.17x	0.93x
Nvidia GTX 680	1.10x	0.89x
AMD FirePro V7800	1.78x	1.94x

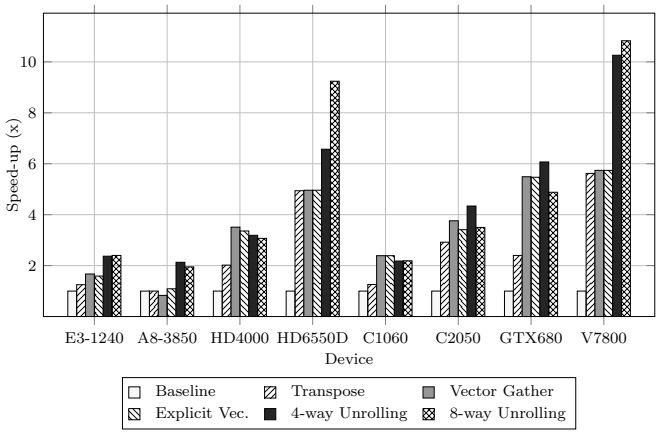

Figure 8. Comparison of speed-ups for each approach on all architectures.

factors (4 and 8), relative to the previous best approach (*i.e.* vector gather, with scalar arithmetic). For 8-way unrolling, we use `float8` types, to explore the potential performance benefits of AVX instructions on CPUs; for 4-way unrolling, we use `float4` types. To handle the case where the number of loop iterations does not divide exactly by 4 or 8, we pad an atom's neighbour list with dummy atoms.

Both CPUs benefit from 4-way unrolling, due to running all SIMD instructions at 100% efficiency. Unrolling also exposes more parallelism to the very-long-instruction-word (VLIW) architecture of AMD GPUs, which see a similar performance increase. The performance impact of unrolling on other architectures is inconsistent, and further investigation into this matter is necessary.

8-way unrolling is more efficient than 4-way unrolling on the AMD GPUs, again because more parallelism is exposed to the underlying microarchitecture. It does not, however, lead to efficient utilisation of AVX on the E3-1240. Examination of the assembly generated by the Intel SDK (Appendix A) reveals that the compiler emits an inefficient instruction sequence for the AoS-to-SoA transpose; instead of using vector *shuffle* instructions to permute the gathered AoS data, as it does when auto-vectorising the kernel, it uses scalar instructions to build the SoA data one element at a time. The A8-3850 also experiences a slow-down, but this is because it does not support AVX instructions.

6) Summary: The graph in Fig. 8 presents the total performance improvement of our optimisations, relative to the original scalar baseline code. This performance improvement is cumulative, since each optimisation is built upon the previous version; the only exception to this is 8-way unrolling, which replaces 4-way unrolling. The speed-up of our best kernel (*i.e.* explicit vectorisation with unrolling) is consistently greater than 2x across all architectures.

Generally speaking, the GPUs see more benefit from the memory optimisations while the CPUs see more benefit from explicit vectorisation. This reflects both the simpler cores of GPUs (which are more sensitive to memory latencies, and are thus more impacted by the original memory access pattern) and the immaturity of OpenCL compilers for CPUs. AMD GPUs are the only architecture to benefit significantly from both types of optimisation (with a speed-up of $\approx$ 11x).

However, where a code change does not improve performance, it does not significantly degrade it. These results thus show that it is possible to accelerate OpenCL code without focusing on any particular microarchitecture, and demonstrates the performance portability of our optimisations.

B. Neighbour List Build

The core behaviour of the neighbour list build is very similar to that of force compute; each work-item computes the distance between an atom and a set of potential neighbours, and compares that distance to the cut-off. We optimise this using `float4` vector types, as we did for force compute. However, instead of updating forces, each work-item appends the indices of those atoms that pass the cut-off check to a list – an operation that does not map well to OpenCL.

For serial code, where the lists for each atom are built one after the other, each index can be written to memory sequentially; further, if the amount of memory allocated for the neighbour list is too small, it can be reallocated. OpenCL does not support the allocation of device memory within kernels, and assigning one work-item per atom leads to all neighbour lists being built in parallel; the starting offset of each neighbour list must be known ahead of time. In our code, we allocate enough memory for each atom to store some maximum number of neighbours (derived from simulation parameters), and calculate the offsets accordingly.

C. Communication

The MPI communication found in miniMD can be split into four components. As atoms move during a simulation, they may move from one node to another – this is known as atom *exchange*. Some atom-pairings may cross node boundaries (*i.e.* the force acting on atom i may depend on atom j, which exists on another node). Following atom exchange, miniMD thus builds lists of two kinds of atoms near the *borders* of a node's subvolume; (1) *dependent* atoms, which require some force contribution from atoms on another node; and (2) *ghost* atoms, which exist on another node and may contribute to

the force acting on dependent atoms. Prior to force compute, each node sends the positions of its dependent atoms to neighbouring nodes (*position communication*); and, after the force compute, each node receives some force contribution from its neighbouring nodes (*force communication*).

Exchange and borders pose a similar challenge to OpenCL as the neighbour list build, since the number of atoms each work-item appends to the list (of atoms being exchanged or at the borders) is not known ahead of time. The packing and unpacking of messages for position and force communication is much simpler, and the transfer of the packed messages can be hidden behind useful work; however, the current communication scheme used by miniMD imposes an ordering on the sending and receiving of these messages, which complicates matters.

Our current implementation does not implement any of these components in OpenCL. Instead, we make use of the default implementation provided by the original miniMD, with device-to-host and host-to-device transfers either side. This is likely to be a performance bottleneck for discrete GPUs, due to the latency and bandwidth of the PCIe bus. The performance of the code executing on CPUs and integrated GPUs will also be decreased, although to a lesser extent; our code does not make any distinction between devices, and each transfer results in a `memcpy` on these platforms. There is a clear need to port these components to OpenCL in future work – in addition to decreasing the amount of data transferred between host and device, the operations themselves would benefit from executing in parallel.

D. Other Components

The remaining components of the simulation are uninteresting in a performance portability study. One kernel zeroes the forces of each atom (prior to the force compute); another updates atom positions, based on their current velocities; and the final kernel updates atom velocities based on the forces acting upon them. Representing these streaming operations in OpenCL such that their performance scales with memory bandwidth (as expected) is trivial.

V. Comparison with Native Implementations

A performance-portable code is clearly more desirable if its performance is competitive with that of native implementations. We therefore compare our OpenCL implementation to (a) the original miniMD, (b) a hand-vectorised implementation using AVX instructions on the Intel Xeon, and (c) $\text{LAMMPS}_{\text{CUDA}}$ on the C1060 and C2050.

A. CPU Comparison

Table V compares the performance of our OpenCL implementation of miniMD with the two native C++ implementations, executing on the Intel E3-1240. In keeping with the rest of the paper, we present OpenCL results from the Intel SDK; we also present results from pairing the AMD SDK with Intel hardware, as a means of comparing the two compilers. For both SDKs, we use the kernel with explicit vectorisation and 4-way unrolling.

Table V
Comparison of Intel CPU execution times (in seconds)

Implementation	Force	Neigh	Comm	Other	Total	Speed-up (Orig.)	Speed-up (AVX)
Original C++	2.77	0.26	0.19	0.21	3.42	1.00x	0.35x
C++ with AVX	0.76	0.11	0.12	0.20	1.19	2.87x	1.00x
OpenCL (Intel)	1.41	0.60	0.25	0.17	2.43	1.41x	0.49x
OpenCL (AMD)	1.33	0.58	0.16	0.17	2.25	1.52x	0.53x

The AMD SDK provides the fastest runtime for OpenCL, beating the Intel SDK by approximately 10%. This is also 1.5x faster than the original C++ version of miniMD, demonstrating that OpenCL is a suitable development tool for utilising the SIMD architectures of modern CPUs. However, our OpenCL implementation is 2x slower than the heavily-optimised version of miniMD which we developed in previous work [26]. This code employs a number of novel SIMD and threading optimisations, which give it a significant advantage over OpenCL; its force compute and neighbour list build are 2x and 6x faster respectively.

The biggest remaining causes of poor performance in our OpenCL implementation are: firstly, that the AVX code generated by current OpenCL compilers is inefficient; and secondly, that both the original miniMD and our optimised AVX code make use of N3, and thus do half as much work during the force compute and neighbour list build. As discussed in Section IV, making use of N3 in OpenCL is difficult, due to restrictions in the programming model regarding work-item synchronisation.

A relatively new addition to the OpenCL standard is the ability to *fission* (*i.e.* split) a device into multiple sub-devices, and issue work to them separately. We can leverage the existing MPI parallelism in miniMD, combined with fission, to permit the use of N3 on CPUs; specifically, we can start 8 MPI ranks and assign one OpenCL sub-device to each. We consider this optimisation only because it does not impact performance portability – although we re-introduce the neighbour force updates (Lines 15–17, from Fig. 2), we guard them with a pragma (*i.e.* `#ifdef N3`), and the code changes necessary to support fission are minimal, affecting only the initial OpenCL setup code. Fission is not (currently) supported by GPUs, but Kepler's ability to support multiple MPI tasks feeding work to the same GPU could be used to the same effect.

For the Intel SDK, using fission improves force compute performance by 1.1x, and neighbour list build performance by 2x, increasing the speed-up over the original miniMD to 1.7x. For the AMD SDK, using fission results in worse performance than the original miniMD; although the improvements to the force compute and neighbour list kernels are similar, communication costs become 9x more expen-

Table VI
COMPARISON OF NVIDIA GPU EXECUTION TIMES (IN SECONDS)

Device	Implementation	Force	Neigh	Comm	Other	Total	Speed-up
C1060	CUDA	0.67	0.35	0.06	0.09	1.18	1.00x
	OpenCL	1.69	0.31	0.34	0.07	2.40	0.49x
C2050	CUDA	0.36	0.22	0.05	0.06	0.70	1.00x
	OpenCL	0.58	0.12	0.28	0.05	1.03	0.68x

sive, possibly due to scheduling conflicts between the threads spawned by MPI and AMD's OpenCL runtime.

B. GPU Comparison

Table VI compares the performance of our OpenCL implementation of miniMD (using the kernel with explicit vectorisation and 4-way unrolling) and $\text{LAMMPS}_{\text{CUDA}}$.

For the complete application, our implementation is 2x slower than CUDA on the C1060, and 1.5x slower than CUDA on the C2050. A large portion of this difference can be attributed to the poor communication scheme used by our OpenCL code, which is almost 6x slower than that used by $\text{LAMMPS}_{\text{CUDA}}$. The CUDA code executes all communication routines on the device and, since we are using a single node, is able to avoid all PCIe communication; our OpenCL code, on the other hand, executes all communication routines on the host.

For the force compute kernel alone, our OpenCL implementation is 2.5x and 1.6x slower than CUDA on the C1060 and C2050 respectively. This is partly due to documented differences between Nvidia's CUDA and OpenCL compilers [11], [12]; despite the programming models and languages being very similar, the CUDA compiler emits more efficient code. $\text{LAMMPS}_{\text{CUDA}}$ also makes use of read-only texture memory, which is cached, to store atom positions. Support for texture memory could be added to our kernel and guarded with pragmas, enabled for GPUs in a similar way to how fission is enabled for CPUs – we intend to explore this option in future work.

VI. DISCUSSION

Many legacy scientific and engineering applications are not performance-optimal, achieving a very low percentage of peak FLOP/s even on a single node (*i.e.* before performance degradations due to less-than-perfect node scaling). Traditionally, application scientists have been reluctant to optimise codes for two reasons: (1) tuning for a specific platform reduces an HPC site's ability to move between vendors; and (2) the ability to maintain code is seen as more important than achieving the best possible performance.

However, studies performed by both academia and industry have shown that sections of these codes are amenable to GPU acceleration – and that programmers are willing to spend time porting kernels to CUDA (or selecting appropriate OpenACC [27] pragmas) in exchange for significant speed-ups. The results in this paper (specifically, that our OpenCL implementation is 1.7x faster than the original miniMD) suggest that, for HPC sites starting from a sub-optimal scalar baseline code, OpenCL offers an opportunity to explore the potential of new architectures while also improving performance on their existing hardware.

```
float8 fxjtmp = (f[j1].x, f[j2].x, f[j3].x, f[j4].x,
                 f[j5].x, f[j6].x, f[j7].x, f[j8].x);

float8 fyjtmp = (f[j1].y, f[j2].y, f[j3].y, f[j4].y,
                 f[j5].y, f[j6].y, f[j7].y, f[j8].y);

float8 fzjtmp = (f[j1].z, f[j2].z, f[j3].z, f[j4].z,
                 f[j5].z, f[j6].z, f[j7].z, f[j8].z);
```

Figure 9. OpenCL code-snippet for gathering x, y and z positions in the kernel with 8-way unrolling.

We feel it is unlikely that OpenCL codes will exceed the performance of heavily optimised native implementations, but there is hope that future compiler releases will decrease the performance gap between them. The performance of scalar kernels should improve as the auto-vectorisation capabilities of Intel's compiler mature; OpenCL is well-suited to expressing parallelism and vectorisation opportunities, so these kernels may even out-perform auto-vectorised C and C++ codes. For kernels employing explicit vector types, the generation of inefficient instruction sequences for gather/scatter operations currently leads to poor performance, but this too could be improved. OpenCL lacks gather/scatter and transpose constructs for its vector types, and so we implement our gather of positions as shown in Figure 9; neither the AMD or Intel compiler recognises that this can be performed as an in-register AoS-to-SoA transpose, and we are currently investigating alternative ways of expressing this operation in OpenCL.

VII. CONCLUSIONS

In this paper, we report on the development of a performance-portable OpenCL implementation of Sandia's miniMD benchmark. We show that the performance bottlenecks of the force compute kernel are the same across several architectures, and that the optimisations that we apply to the original scalar code improve performance by more than 2x across a wide range of hardware types from different vendors: CPUs and integrated GPUs from AMD and Intel; and discrete GPUs from AMD and Nvidia.

Our complete OpenCL implementation is 1.7x faster than the original miniMD code running on the same hardware, and at most 2x slower than "native" implementations highly optimised for particular platforms. Whether this performance compromise is an acceptable penalty for increased code portability is open to debate, but the results in this paper suggest that OpenCL is a suitable development tool to bridge the gap between architectures – enabling HPC sites to develop codes that exhibit good levels of performance, across a wide variety of hardware, without sacrificing maintainability.

ACKNOWLEDGEMENTS

The authors thank Chris Hughes and Mikhail Smelyanskiy of Intel Corporation for their valuable comments and suggestions. Thanks also to Sandia National Laboratories for access to the Teller Testbed system; Teller is provided under the United States Department of Energy ASCR and NNSA Testbed Computing Program.

This research is supported in part by The Royal Society through their Industry Fellowship Scheme (IF090020/AM). Access to hardware is made possible through collaboration with the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Performance Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme).

CODE ACCESS

Our port of miniMD is available for download from:

go.warwick.ac.uk/pcav/areas/hpc/download/minimd_opencl

REFERENCES

[1] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top 500 Supercomputer Sites," http://top500.org/, November 2011.

[2] 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.

[3] ——, "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.

[4] 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*, Berkeley, CA, 14-15 June 2010.

[5] 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*, Saint-Malo, France, 21-23 June 2010, pp. 451–460.

[6] Khronos Group, "OpenCL 1.2 Specification," http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, November 2011.

[7] M. Heroux and R. Barrett, "Mantevo Project," http://software.sandia.gov/mantevo/, May 2011.

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

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

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

[11] P. Du *et al.*, "From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-platform GPU Programming," Knoxville, TN, Tech. Rep. UT-CS-10-656, 2010.

[12] K. Komatsu *et al.*, "Evaluating Performance and Portability of OpenCL Programs," in *Proceedings of the Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[13] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 58–68, January 2011.

[14] S. Seo, G. Jo, and J. Lee, "Performance Characterization of the NAS Parallel Benchmarks in OpenCL," in *Proceedings of the International Symposium on Workload Characterization*, Austin, TX, 6–8 November 2011, pp. 137–148.

[15] S. J. Pennycook *et al.*, "An Investigation of the Performance Portability of OpenCL," *Journal of Parallel and Distributed Computing*, in press.

[16] 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*, Long Beach, CA, 26-30 March 2007.

[17] 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.

[18] 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.

[19] 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.

[20] D. Case *et al.*, "AMBER 11," http://www.ambermd.org, May 2011.

[21] 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*, Austin, TX, 15-21 November 2008, pp. 1–9.

[22] 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.

[23] J. Kim *et al.*, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," in *Proceedings of the International Conference on Supercomputing*, Venice, Italy, 25–29 June 2012, pp. 341–352.

[24] R. Karrenberg and S. Hack, "Improving Performance of OpenCL on CPUs," in *Proceedings of the International Conference on Compiler Construction*, Tallinn, Estonia, March 2012, pp. 1–20.

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

[26] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring SIMD for Molecular Dynamics, Using Intel Xeon and Xeon Phi Processors," unpublished.

[27] "OpenACC 1.0 Specification," http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2011.

APPENDIX
ASSEMBLY GENERATED FOR GATHER OPERATIONS

Figures 10 and 11 list the assembly generated by the Intel SDK for two of our kernels: the auto-vectorised vector gather kernel, and the kernel with explicit vectorisation and 8-way unrolling, respectively. The instructions have been re-ordered and grouped to improve readability, and any instructions not directly related to the gather of positions have been removed.

Both listings begin in the same way, loading the positions of eight neighbours into eight 128-bit (XMM) registers. These positions are stored in AoS format ($\{x, y, z, 0\}$) and both pieces of assembly transpose the gathered positions into SoA format ($\{x, x, ...\}$, $\{y, y, ...\}$ and $\{z, z, ...\}$), storing the result in three 256-bit (YMM) registers. During auto-vectorisation, the compiler recognises that this operation can be performed by an in-register AoS-to-SoA transpose, emitting a sequence of 14 shuffle and permute instructions. The sequence generated for our explicit vector kernel is significantly less efficient, containing 37 instructions.

Each of the `vinsertps` instructions in Fig. 11 extracts the x, y or z component from one XMM register (in AoS), and inserts it into another (in SoA). The result is six XMM registers, two for each of x, y and z, which are then combined into three YMM registers using three `vinsertf128`s. These inserts together account for 21 of the 37 instructions.

The `vinsertps` instruction has the capability to extract *any* 32-bit element from an XMM register, and we were therefore surprised to see that each of its uses here extracts the lowest 32 bits. This requires the generation of the 16 remaining instructions, to rearrange the XMM registers – each `vpshufd` instruction moves an atom's y component to the lowest 32-bits of the register, and each `vmovhlps` instruction does the same for z.

We believe that these behaviours are caused by a combination of compiler immaturity and the way in which we have expressed the gather operation in OpenCL (Fig.9). Future compiler releases, or an alternative representation of the gather in code, may address these issues, leading to considerably better performance for this kernel on hardware supporting AVX instructions.

```
// Load {x, y, z, 0} for eight atoms.
vmovaps     XMM7, XMMWORD PTR [R11 + R15]
vmovaps     XMM9, XMMWORD PTR [R10]
vmovaps     XMM10, XMMWORD PTR [R9]
vmovaps     XMM11, XMMWORD PTR [R8]
vmovaps     XMM12, XMMWORD PTR [RDI]
vmovaps     XMM13, XMMWORD PTR [RSI]
vmovaps     XMM14, XMMWORD PTR [RDX]
vmovaps     XMM15, XMMWORD PTR [R12]

// Build SoA registers for x, y and z.
vperm2f128  YMM11, YMM11, YMM15, 32
vperm2f128  YMM9, YMM9, YMM13, 32
vshufps     YMM13, YMM9, YMM11, 68
vperm2f128  YMM10, YMM10, YMM14, 32
vperm2f128  YMM7, YMM7, YMM12, 32
vshufps     YMM12, YMM7, YMM10, 68
vshufps     YMM14, YMM12, YMM13, -35
vpermilps   YMM14, YMM14, -40
vshufps     YMM12, YMM12, YMM13, -120
vpermilps   YMM12, YMM12, -40
vshufps     YMM9, YMM9, YMM11, -18
vshufps     YMM7, YMM7, YMM10, -18
vshufps     YMM7, YMM7, YMM9, -120
vpermilps   YMM7, YMM7, -40
```

Figure 10. Assembly from our vector gather kernel.

```
// Load {x, y, z, 0} for eight atoms.
vmovdqa     XMM3, XMMWORD PTR [R9 + R13]
vmovdqa     XMM4, XMMWORD PTR [R9 + R13]
vmovdqa     XMM6, XMMWORD PTR [R9 + R13]
vmovdqa     XMM7, XMMWORD PTR [R9 + R13]
vmovdqa     XMM8, XMMWORD PTR [R9 + R13]
vmovdqa     XMM9, XMMWORD PTR [R9 + R13]
vmovdqa     XMM11, XMMWORD PTR [R9 + R13]
vmovdqa     XMM12, XMMWORD PTR [R9 + R13]

// Build SoA register for x component.
vinsertps   XMM5, XMM4, XMM3, 16
vinsertps   XMM5, XMM5, XMM6, 32
vinsertps   XMM5, XMM5, XMM7, 48
vinsertps   XMM10, XMM9, XMM8, 16
vinsertps   XMM10, XMM10, XMM11, 32
vinsertps   XMM10, XMM10, XMM12, 48
vinsertf128 YMM5, YMM10, XMM5, 1

// Build SoA register for y component.
vpshufd     XMM13, XMM4, 1
vpshufd     XMM14, XMM3, 1
vinsertps   XMM13, XMM13, XMM14, 16
vpshufd     XMM14, XMM6, 1
vinsertps   XMM13, XMM13, XMM14, 32
vpshufd     XMM14, XMM7, 1
vinsertps   XMM13, XMM13, XMM14, 48
vpshufd     XMM14, XMM9, 1
vpshufd     XMM15, XMM8, 1
vinsertps   XMM14, XMM14, XMM15, 16
vpshufd     XMM15, XMM11, 1
vinsertps   XMM14, XMM14, XMM15, 32
vpshufd     XMM15, XMM12, 1
vinsertps   XMM14, XMM14, XMM15, 48
vinsertf128 YMM13, YMM14, XMM13, 1

// Build SoA register for z component.
vmovhlps    XMM4, XMM4, XMM4
vmovhlps    XMM3, XMM3, XMM3
vinsertps   XMM3, XMM4, XMM3, 16
vmovhlps    XMM4, XMM6, XMM6
vinsertps   XMM3, XMM3, XMM4, 32
vmovhlps    XMM4, XMM7, XMM7
vinsertps   XMM3, XMM3, XMM4, 48
vmovhlps    XMM4, XMM9, XMM9
vmovhlps    XMM6, XMM8, XMM8
vinsertps   XMM4, XMM4, XMM6, 16
vmovhlps    XMM6, XMM11, XMM11
vinsertps   XMM4, XMM4, XMM6, 32
vmovhlps    XMM6, XMM12, XMM12
vinsertps   XMM4, XMM4, XMM6, 48
vinsertf128 YMM3, YMM4, XMM3, 1
```

Figure 11. Assembly from our kernel with 8-way unrolling.