Benchmarking Molecular Dynamics with OpenCL on Many-Core Architectures

Rene Halver[1], Wilhelm Homberg[1], and Godehard Sutmann[1,2,*]

[1] Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation (IAS), Forschungszentrum Jülich (JSC), D-52425 Jülich, Germany,
g.sutmann@fz-juelich.de
[2] ICAMS, Ruhr-University Bochum, D-44801 Bochum, Germany

Abstract. Molecular Dynamics (MD) is a widely used tool for simulations of particle systems with pair-wise interactions. Since large scale MD simulations are very demanding in computation time, parallelisation is an important factor. As in the current HPC environment different heterogeneous computing architectures are emerging, a benchmark tool for a representative number of these architectures is desirable. OpenCL as a platform-overarching standard provides the capabilities for such a benchmark. This paper describes the implementation of an OpenCL MD benchmark code and discusses the results achieved on different types of computing hardware.

Keywords: Molecular Dynamics, OpenCL, shared memory parallelisation, many-core architectures

1 Introduction

Molecular Dynamics (MD) is widely used in various scientific domains, e.g. materials science or biophysics, where the evolution of specific systems can be described by point-like or extended particles, obeying the classical equations of motion [5,8]. Parametrised potentials describe pair-wise interactions between particles that may have either short-ranged (e.g. Lennard Jones interactions [5]) or long-ranged (e.g. electrostatics [7]) influences. The essential difference between long- and short-range interactions is the number of interaction partners, which strongly determines the performance of the method and also determines different types of, e.g., parallelisation schemes. While long range interactions require essentially all atoms in the system as interaction partners, short range interactions can be restricted to a narrow range, defined by a spherical region of radius R_c (the cutoff radius), in which interactions decrease to a sufficiently small value which can be tolerated as error.

In the present paper we will focus on short-range interactions, for which also neighbour list techniques will be considered which allow for a linear computational complexity with increasing number of particles in the system (cmp. Fig. 1).

For a shared-memory parallelisation the construction of these neighbour lists must avoid race conditions which would occur, when multiple threads try to update the list of a single cell at the same time, which can be expected for the case of a thread-parallel implementation where particles are divided between threads and are sorted into the cell structure simultaneously. To avoid possible race-conditions there exist basically three options: (i) explicit synchronisation; (ii) list copies; or (iii) atomic memory access implemented via compare-and-swap (CAS) operations.

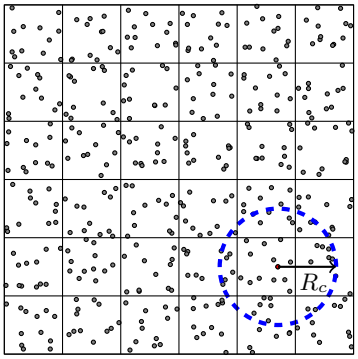


Fig. 1: Schematic of a particle system in 2D with overlayed cell structure for sorting. The circle with radius R_c illustrates the interaction range of a tagged particle.

In order to implement a function portable benchmark that can be run on a variety of different architectures, it is required to use a programming language that supports a large variety of architectures. OpenCL is one possible choice as it supports code execution on CPUs, GPUs, FPGAs as well as Intels Xeon Phi architecture.[1] Although other languages or language extensions, such as Intel TBB or OpenACC exist, which provide the possibility to run code on a set of different architectures, for this work OpenCL was chosen, because it supports all of the platforms available at the Jülich Supercomputing Centre (JSC). The set of features of OpenCL that can be used for a platform-overarching benchmark is limited by the lowest commonly supported OpenCL standard as well as by the set of extensions commonly available on these platforms. While the standard level defines the syntax and general features that can be used, some functionality is kept in extensions, e.g. the use of double-precision calculations or CAS functionality. To execute the same program version on all considered architectures we had to comply with OpenCL 1.2, which was found as common standard level on all machines.

2 Benchmark

In the present article we focus on short range interacting particle systems, where the range of influence is defined by the cutoff radius R_C. When introducing the concept of linked-cell lists, the computational complexity is $\mathcal{O}(NM)$, where N is the number of particles in the system and M the maximum number of particles in a cell [8,10]. As benchmark system we consider particles in a cuboid 3-dimensional box of lengths L_α ($\alpha = x, y, z$) with periodic boundary conditions, interacting via the repelling part of the Lennard-Jones potential [5], which is a typical representative for a short range potential

$$U(r \leq r^*) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} + \frac{1}{4} \right] \tag{1}$$

with cut-off radius $r^* = 2^{1/6}\sigma$ and $U(r > r^*) = 0$ from where forces $\mathbf{F} = -\nabla U(r)$ onto particles are computed; r is the distance between two particles, ε the depth of the potential well and σ the characteristic size of a particle. To propagate particles continuously in space the classical equations of motion are integrated via the standard Verlet algorithm [5].

In the present article we consider a constant average number of particles per cell $\langle M \rangle$, which leads to a total number of particles in the system $N = n_x n_y n_z \langle M \rangle$, with n_x, n_y, n_z being the number of cells in each cartesian direction. The relation of cell size l_c to the cutoff radius R_c is simply given by $R_c = l_c = 2^{1/6}\sigma$ or $\sigma = l_c/2^{1/6}$ and therefore the boxsize is $L_\alpha = n_\alpha R_c$.

In the next section different techniques are described, which improve the performance of the simulation before presenting the results of the benchmarks in the final section.

3 Implementation of the Algorithm and Data Structures

3.1 Algorithmic Implementation Details

While a multi-node parallelization of MD simulations, e.g. based on a domain decomposition [9], is standard, the shared-memory parallelization of neighbor cell construction within each of the decomposed domains is not trivial. Since the most efficient way to sort particles into spatial cells is to distribute particles onto different threads. This might lead to possible concurrent memory accesses, if more than one thread attempts to sort particles into the same cell. As the memory access is not synchronized by default, race conditions can occur due to simultaneous write operations into single memory locations, leading to erroneous accounting of particles and list constructions.
Three different approaches will be described to avoid such race conditions: (i) list copying, (ii) explicit synchronization and (iii) atomic memory access. Each of these approaches has its specific advantages and disadvantages.

Of the three different approaches the synchronization-based approach is the one that requires the fewest changes to the sequential implementation (ref. Alg 1). The array containing the cell entries needs to be initialized with a terminating value indicating the end of the list, the array containing the particle lists needs to be initialized similarily.

Algorithm 1: Sequential implementation of neighbor cell sorting

```
entry ← EOA; list ← EOA // initialize to end-of-array (EOA)
for all particles pidx ∈ {i}_N do
    cidx ← f(x[pidx], y[pidx], z[pidx]) // calculate cell index from particle coordinates
    list[pidx] ← entry[cidx] // update list element to current cell entry
    entry[cidx] ← pidx // update cell entry ;
end
```

As is seen in Alg. 1 a problem might occur if two threads try to simultaneously update the cell information. If these threads execute the first step, i.e.

updating `list[pidx]`, before any of them completes the second step, i.e. updating `entry[cidx]`, it follows that their particles are pointing in `list` to the same former entry particle, `entry`[`cidx`], ignorant of each other. Then, after updating `entry`[`cidx`] in the second step, `entry`[`cidx`] will contain only one of those particle indices, the linked list is broken and excludes the other particle from being accessible through the list.

In order to combine the two statements into one 'atomic' statement, one can either use critical sections or locks. Both of these techniques introduce overhead cost due to the implicit synchronization. Of the two the latter shows a much better scaling behavior than the first, since the creation of a critical section will serialize that section [6]. For this benchmark all of these synchronization-based approaches are not feasible, since the asynchronous execution of work groups excludes global synchronization. Therefore race-conditions happening between two different work groups cannot be avoided by these techniques.
In contrast to the synchronization-based approaches the copy-based approach utilizes thread-local partial copies of the final result in order to avoid race-conditions. Each thread independently works on its local copy and sorts all of its particles into this copy. After each thread has finished the local copies are merged. While this variant can be very efficient for a small number of threads, it becomes less effective once the number of threads reaches a threshold value, which depends on memory size and bandwidth [6]. This is due to the increased number of copies, which have to be filled simultaneously, leading to random access to (main) memory. For massively parallel systems, this approach is not feasible since the memory size requirements grow linearly with the number of threads, particularly on GPUs or Intel MICs, where hundreds and thousands of threads work concurrently. As a consequence, this approach has been discarded for the present benchmark.

Listing 1: Creation of a neighbor list using CAS operations in OpenCL

```
1: int old, cmp;
2: // arrays of particle positions (x,y,z), NC = no. of cells per dim
3: int cidx = (int)(x[pidx]/l_c) +
4:            (int)(y[pidx]/l_c) * NC +
5:            (int)(z[pidx]/l_c) * NC * NC;
6: //application of CAS operation to update list/entry
7: do
8: {
9:   // store old entry particle index
10:  old = entry[cidx];
11:  // update next particle in list for particle pidx
12:  list[pidx] = old;
13:  // try to update entry particle of target cell cidx
14:  cmp = atomic_cmpxchg(entry+idx,old,gid);
15: }
16: // if update failed, repeat the process
17: while(old != cmp);
```

Therefore, an approach is required which ensures the correctness of a parallel list construction without exacerbating the memory demand. Atomic memory access is a possible solution utilizing the compare and swap (CAS) operation. This hardware operation compares the value stored at a memory address to a test value before updating the memory address. Listing 1 shows as an example for

the parallel list construction implemented with a CAS operation in OpenCL: First, the cell index `cidx` is calculated for the local particle `pidx`. Within a loop the first entry of the particle list of this cell `entry[cidx]` is stored in a temporary test value `old`. Then, the particle list at index `pidx` is updated to the value of `old`. This operation can be performed safely, since no other work item processes particle `pidx`. Next, the CAS operation is used to attempt an update of `entry[cidx]`. Now two different outcomes might occur: (i) `entry[cidx]` was changed in the mean time by another work item. In this case the value of `old` is not equal to the current value of `entry[cidx]`, the update is omitted and the loop is repeated. (ii) the value of `entry[cidx]` was not changed and its value is identical to that of `old`. In this case `entry[cidx]` is updated. The CAS operation returns the current value of `entry[cidx]`, for (i) a value different from `old`, for (ii) the same value as `old`. In order to check the success of the update, the return value of the CAS operation is stored to `cmp` and compared with `old` at the end of the loop.
Compared to an update of a memory location by an assignment, hardware supported CAS operations slightly increase the runtime due to the performed compare operation. The essential advantage of the CAS operation is that it can be applied to work items of different work groups in an OpenCL implementation. Therefore the implementation using the CAS operation was the method of choice for this benchmark.

3.2 Organization and Distribution of Data Structures

An important aspect aspect concerns data locality, i.e. both the important difference between sorted and unsorted particle data and their access as well as the layout of the data structures containing this data. To this end, two different memory layouts were implemented for the arrays used to store the particle data. The first is an *array of structures* (AoS), where each structure contains the data of a single particle, the second is a *structure of arrays* (SoA), where a single structure contains a collection of arrays, each representing a parameter of a particle. In a SoA the data of a single particle has the same index in each of the arrays. For both implementations (AoS and SoA) a sorting algorithm was implemented, that sorts the particle data array with regard to the neighbor list, i.e. particles in the same neighbor cell are grouped together in the particle data array to have higher data locality. In the result section differences between these four possible implementations will be shown.

kernel function	distrib.
creation of particles	p
initial. of cells	c
initial. of particle lists	p
creation of neighbor lists	p
counting of cell contents	c
prefix sums calc. (sorting)	c
resort of particle array	c
calculation of interactions	c
integration of particles	p

Table 1: List of implemented OpenCL kernels: distribution of (p) particles or (c) cells onto work items.

In order to compare a representative set of multi- and many-core architectures, while avoiding a rewrite of the code for each individual type, OpenCL was selected as a portable programming language. Basic parts of the MD algorithm were implemented into different OpenCL kernels (see Table 1), in order to benchmark the functional units of the program independently. The table shows the different ways of distributing data to the work items and related kernel functions which either act on a single particle or on a complete cell. Some of the kernel functions require data of all particles in a cell to compute properties resulting from the whole environment of a particle, e.g. the calculation of total forces on a particle. Others can update the individual particle state information independently from other particles, e.g. the propagation of position or velocities.
If not mentioned otherwise, the number of work items (wi) within a work group (wg) is kept constant, so that the number of required work groups for a given system is calculated by $n_{wg} = n/n_{wi}$, where $n = N$ (number of particles) or $n = N_c$ (number of cells), depending on the kernel (Table 1).

4 Architectures

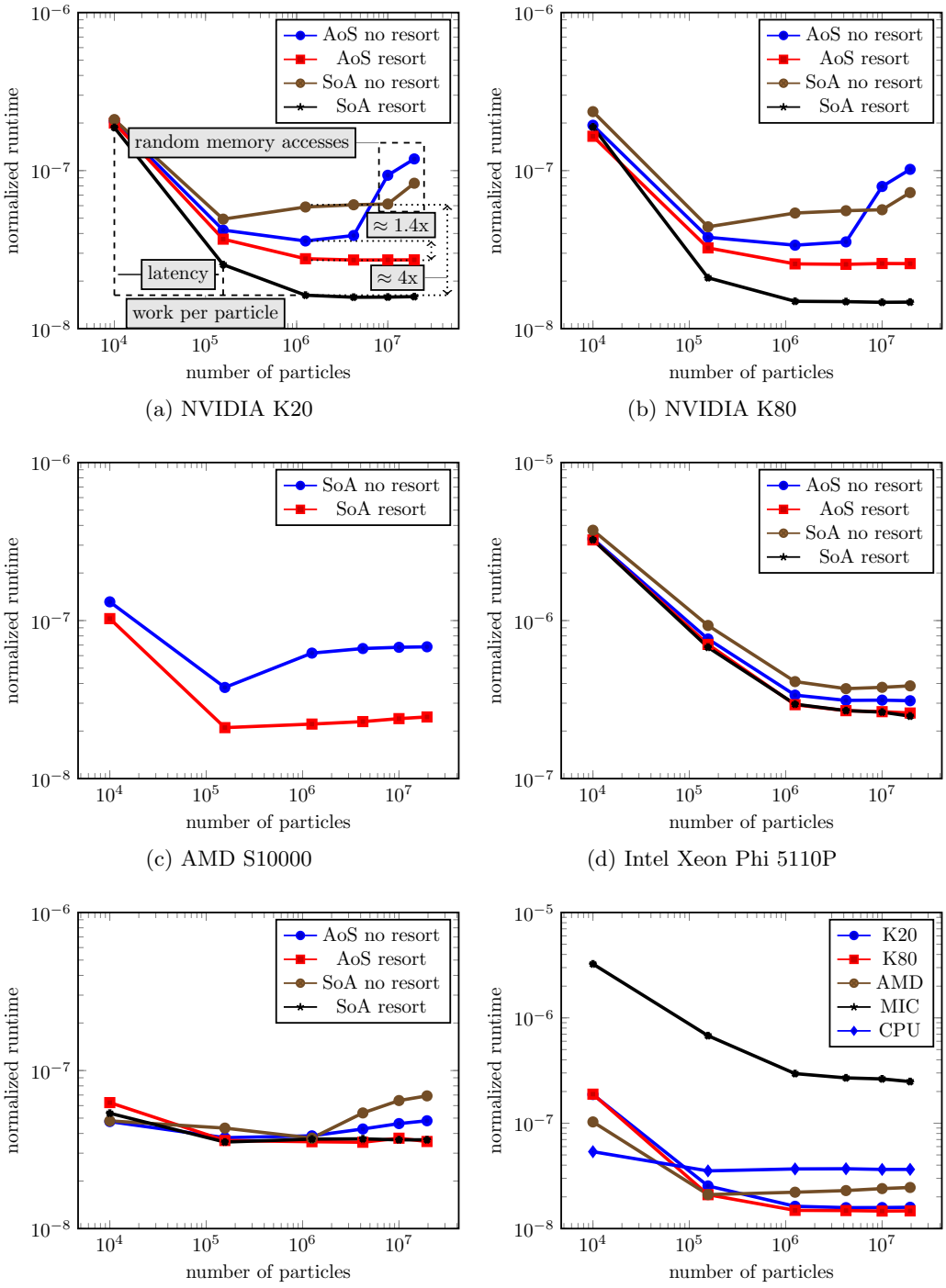
The benchmarks were conducted on three different machines: JURECA [2], JUROPA3 [3] (two of the supercomputer systems at JSC) as well as on a testing system for GPUs. Table 2 shows the specifications for the different architectures used for benchmarking. JUROPA3 has different partitions, that employ the same CPUs (Intel E5 2650), but contain different accelerators. For the scope of this paper the benchmarks were run on the GPU and the Intel Xeon Phi partitions. Due to the availability of the Intel OpenCL driver on the Intel Xeon Phi partition, the CPU comparison was conducted on JUROPA3 instead of running them on the faster E5-2680 Haswell processors on JURECA, where no OpenCL support is available for the CPU. Since the AMD GPU has less memory available than its NVIDIA counterparts, larger benchmarks could not be performed on the card. For all the tests the code was compiled with the GNU compiler version 4.9.3., with operating system CentOS 7.

Architecture	Relevant Components	Peak Performance
GPU (K20)	NVIDIA K20X	3.95 TF (sp), 1.31 TF (dp)
GPU (K80)	NVIDIA K80	5.6 TF (sp), 1.87 TF (dp)
GPU (S10000)	AMD S10000	5.91 TF (sp), 1.48 TF (dp)
Xeon Phi (5110P)	Intel Xeon Phi 5110P	1.011 TF (sp)
CPU (SandyBridge)	Intel Xeon E5 2650	128 GF

Table 2: Specifications of the systems used for the different architectures: (sp) single precision (dp) double precision

5 Results

To compare the performance on the different architectures, several benchmark runs were conducted. The most basic one was the comparison of the normalised runtime of the interaction kernel on each architecture (see Figs. 2a - 2f). Here,



(a) NVIDIA K20

(b) NVIDIA K80

(c) AMD S10000

(d) Intel Xeon Phi 5110P

(e) Intel Xeon E5-2650

(f) SoA single precision, all architectures

Fig. 2: Runtime comparison on all architectures, using single-precision calculations. Note the shift in scale in (d) and (f) in order to show the full range.

normalisation is defined as the measured runtime divided by the number of particles and the number of timesteps. No data transfer times were included, since the data is kept resident in the device memory for the complete benchmark and no additional data transfer is required. For the case of resorting the time required to resort the data is included into the presented times, i.e. the runtime is the sum of the time spent in the interaction kernel and the time spent to resort the data. All results were obtained with single precision calculations. For the AMD GPU (Fig. 2c) the AoS variant of the benchmark failed to execute for unknown reasons and therefore only the SoA results are presented. When comparing the benchmark results, it can be seen that the different architectures show a specific behavior; since the K20X (Fig. 2a) and the K80 GPUs (Fig. 2b) are different versions of the same production line, their results look fairly similar. With the exception of the Xeon E5-2650 CPU (Fig. 2e) all benchmarks suggest an architecture-dependent minimum problem size that must be reached before a stable performance is achieved. I.e. for small system sizes the specific runtime is exceedingly large compared with large system sizes. Exemplary this is shown in Fig. 2a and is due to the latency of high frequency accesses to small chunks of memory. Since transfer times are excluded in the results, an additional contribution to this behaviour is expected to result from a device-dependent overhead induced by the scheduling of the work-groups on the device. On the CPU this behaviour can only be observed to a smaller extent than on the other devices. For nearly all architectures, except the Xeon Phi (Fig. 2d) it can be observed that in the case of non-sorted data the runtime for larger problem sizes increases again. This can be understood by the fact that non-sorted data are scattered over memory and will have an unfavourable access pattern compared to the case of sorted data, where all particles belonging to a single cell are stored consecutively in memory. A possible explanation, why this behaviour cannot be observed on the Xeon Phi is the much lower overall performance of the Xeon Phi which hides the data access time behind a large computation time. For all other architectures the compute time is already so small that data access time is a crucial measure for the overall performance.

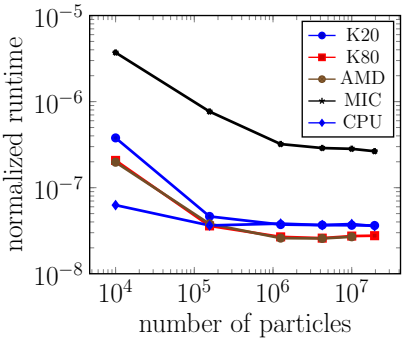


Fig. 3: all architecture comparison (double precision)

The choice of the data structure has a strong impact on the GPU performance and to a smaller extent also on the Xeon and Xeon Phi architectures. Since for the calculation of the forces within the interaction kernel only parts of the particle data is required (position and forces), it is very inefficient to store the whole set of data (velocities, masses, indices) within a single structure. In this case all the particle data within the complete structure would be loaded into the cache, filling it with nonessential data, i.e. leading to unnecessary data transfer

and inefficient cache usage. However, if data is stored in individual arrays, data of consecutive particles is loaded into cache, and can be reused more efficiently. The difference between GPU and CPU performance comes into play when considering the size of data loaded into the registers. CPUs have a smaller capacity of data size loaded into registers, i.e. the load-operations need to be performed with a higher frequency than on a GPU. Therefore, the ratio between performance and load-operations is less favourable on a CPU and data layout patterns have less impact on the overall performance. On the other hand a GPU can operate most efficiently on large streams of data which can be consecutively processed. If the overall number of data loading operations is increased due to nonessential items in the data structures, performance degradation becomes more severe. This might explain the differences in the performance between GPU architectures (NVIDIA GPUs in Figs. 2a, 2b) and CPUs (Fig. 2e).
One peculiar detail that was observed on the AMD S10000 is the low single-precision performance compared to the NVIDIA cards. From the specification of the peak performance characteristics (cmp. Table 2), the AMD card should perform on the same level as the K80. However, the memory bandwidth of the S10000 is lower when compared to the NVIDIA GPUs leading to a reduced overall performance of the single-precision benchmark. For the case of double precision calculations, the frequency of load operations gets lower and therefore the effect of memory bandwidth limitations gets less pronounced (cmp. Fig. 3). Therefore, measuring double-precision performance on the AMD GPU shows comparable results to the NVIDIA K80, as could be expected.
Overall the benchmark shows a better performance for GPUs in comparison to the Xeon E5-2650 when test systems have a sufficient size, i.e. beyond showing memory latencies (Fig. 2f). Only for double-precision calculations it is observed that the K20X is only as fast as the CPU (cmp. Fig. 3). We only note here that the performance on the Xeon Phi is lacking behind all other architectures, since it is roughly a magnitude slower than other machines (Fig. 2f). A main reason for this behaviour is the missing vectorisation optimisation of the code, which was out of focus for this paper. Furthermore, the OpenCL drivers might lack best optimisation for the Intel Xeon Phi KNC, since the OpenCL support for for KNC is deprecated [1].

6 Conclusion

One of the main goals of the present investigation was to investigate the performance characteristics of a function portable cell based MD program on various architectures. OpenCL has been selected as a programming language which allows for interoperability on different types of architectures, e.g. CPU and GPU based systems. Without any changes of the code it was possible to execute the code on several multi- and many-core systems available at JSC. As a downside of that approach the benchmark was not optimised for either of the architectures and therefore did not present optimal performance achievable. This especially accounts for the Xeon Phi system, where vectorisation is an essential issue to

outperform a simple porting of a given program in comparison to running (even not optimised) on other architectures. On the other side this approach offers the possibility to have a comparison between basic features on the different architectures, e.g. memory bandwidth and core speed. To account for different data access patterns, we included the comparison of AoS and SoA, which is an important issue for the GPU architectures. It could be shown, that the SoA layout even improves the performance on the CPU for the non-sorted case and does not degrade performance in the sorted case. In this respect it can be concluded that further optimisation for GPU architectures will lead most probable to a significant performance gain compared to CPUs. Since the OpenCL support for Xeon Phi is already deprecated, it remains to be seen, if OpenCL will remain to be a valid option supporting newer Intel Xeon Phi architectures, e.g., KNL. Nevertheless, the choice of OpenCL provides the option to design MD simulation packages that can run on a variety of different architectures, without the need to provide specialised kernels or programs for each individual machine. In the present article we focused the work on function portability. At least as important as this is the request for performance portability, which we excluded from the present investigation, but which is of high importance in view of the developments towards more complex and heterogeneous architectures. International consortia are considering this aspect and it has to be awaited whether this leads to simplified porting and improved accessibility of future architectures (for a collection of contributions, see e.g. [4]).

References

1. Intel OpenCL SDK, https://software.intel.com/en-us/articles/opencl-drivers
2. JURECA, http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html
3. JUROPA3, http://www.fz-juelich.de/ias/jsc/EN/Research/HPCTechnology/ClusterComputing/JUROPA-3/JUROPA-3_node.html
4. Performance Portability WS DOE, https://asc.llnl.gov/DOE-COE-Mtg-2016/
5. Frenkel, D., Smit, B.: Understanding molecular simulation. From algorithms to applications. Academic Press, San Diego (2002)
6. Halver, R., Sutmann, G.: Multi-Threaded Construction of Neighbour Lists for Particle Systems in OpenMP. 11th International Conference on Parallel Processing and Applied Mathematics, Krakow (Poland), 6 Sep 2015 - 9 Sep 2015 (Sep 2015), http://juser.fz-juelich.de/record/279249
7. Hockney, R.W., Eastwood, J.W.: Computer simulation using particles. McGraw-Hill, New York (1981)
8. Rapaport, D.: The Art of Molecular Dynamics Simulation. Cambridge University Press, Cambridge (2001)
9. Sutmann, G.: Classical molecular dynamics. In: Grotendorst, J., Marx, D., Muramatsu, A. (eds.) Quantum simulations of many-body systems: from theory to algorithms. vol. 10, pp. 211–254. John von Neumann Institute for Computing, Jülich (2002)
10. Sutmann, G., Stegailov, V.: Optimization of neighbor list techniques in liquid matter simulations. J. Mol. Liq. 125, 197–203 (2006)