Noname manuscript No.
(will be inserted by the editor)

Function Portability of Molecular Dynamics on Heterogeneous Parallel Architectures with OpenCL

Rene Halver, Wilhelm Homberg and Godehard Sutmann*

the date of receipt and acceptance should be inserted later

Abstract Classical molecular dynamics simulation for atomistic systems is implemented in OpenCL and benchmarked on a variety of different hardware platforms. Modifying the number of particles and system size in the study provides insight into characteristics of parallel compute platforms, where latency, data transfer, memory access characteristics and compute intense work can be identified as fingerprints in benchmark runs. Data layouts are compared, for which the access of structure-of-arrays shows best performance in most cases. It is demonstrated that function portability can be achieved straightforwardly with OpenCL, while performance portability lacks behind as various architectures strongly depend on specific vectorisation optimisation.

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, soft matter or biophysics, where the evolution of specific systems can be described by point-like or extended particles, obeying the

Rene Halver
Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation (IAS), Forschungszentrum Jülich (JSC), D-52425 Jülich, Germany, E-mail: r.halver@fz-juelich.de

Wilhelm Homberg
Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation (IAS), Forschungszentrum Jülich (JSC), D-52425 Jülich, Germany, E-mail: w.homberg@fz-juelich.de

Godehard Sutmann
Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation (IAS), Forschungszentrum Jülich (JSC), D-52425 Jülich, Germany, E-mail: g.sutmann@fz-juelich.de
ICAMS, Ruhr-University Bochum, D-44801 Bochum, Germany

classical equations of motion [2]. Parametrised potentials describe pair-wise interactions between particles that may have either short-ranged (e.g. excluded volume interactions [2]) or long-ranged (e.g. electrostatics [4]) influences. Short range interactions, in combination with pair lists are considered in the present article (Fig. 1) and can be reduced to a linear complexity, $\mathcal{O}(N)$, as interactions decay sufficiently fast to be omitted beyond the cutoff radius R_c. A shared-memory parallelisation of neighbour lists is prone to race conditions if no special care is taken. Collisions between reading and writing of data in a thread-parallel implementation, where particles are divided between threads, is very likely to occur [3]. 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.

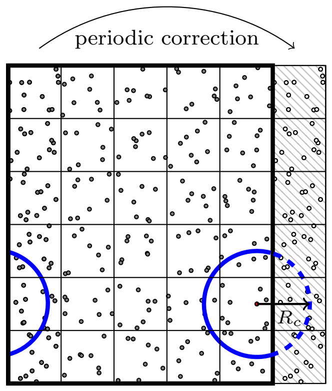


Fig. 1: Schematic of a particle system in 2D with periodic boundary conditions. The circle indicates the interaction range of a tagged particle, restricted to region with cutoff-radius R_c (with cell length $l_c = R_c$).

For a function portable benchmark OpenCL is one possible choice among others (e.g. Intel TBB or OpenACC) as it supports code execution on CPUs, GPUs, FPGAs as well as Intels Xeon Phi architecture [5]. For the present work, OpenCL was chosen, as it is supported on all shared memory architectures, available at 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 compare-and-swap (CAS) functionality. To execute the same program version on all considered architectures, OpenCL 1.2 had to be chosen, 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 $\mathcal{O}(NM)$, with N being the number of particles in the system and M the maximum number of particles in a cell [8, 10]. The benchmark system consists of 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 a Lennard-Jones potential [2], 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 [2].

The average number of particles per cell $\langle M \rangle$, is kept constant, giving a total number of particles $N = n_x n_y n_z \langle M \rangle$, 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 parallelisation of MD simulations, e.g. based on a domain decomposition [9], is standard, a shared-memory parallelisation of neighbor cell construction with each domain is not trivial. If particles are distributed onto different threads on domains, concurrent memory access is likely, 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 copies, (ii) explicit synchronisation and (iii) atomic memory access. Each of these approaches has its specific advantages and disadvantages.

Of the three different approaches the synchronisation-based approach is the one that requires the fewest changes to the sequential implementation. 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`]. 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 principle, the two statements need to be combined into one 'atomic' statement for which either critical sections or locks could be used. However, as OpenCL is intrinsically based on asynchronous execution, all solutions for race-condition

Algorithm 1: Sequential implementation of neighbor cell sorting

```
for all particles $P$ do
    calculate cell index $idx$;
    update list element $L[P]$ to current cell entry $E[idx]$;
    update cell entry $E[idx]$ to $P$;
end
```

prevention, relying on global synchronisation of work groups are excluded by definition.
In contrast to the synchronisation-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 into which all of its particles are sorted. After completion, local copies are merged. While this variant can be very efficient for a small number of threads, it becomes inefficient once the number of threads reaches a threshold value, which depends on memory size and bandwidth [3]. 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. 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 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

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

```
int old, cmp;
// arrays of particle positions (x,y,z), NC = no. of cells per dim
int cidx = (int)(x[pidx]/l_c) +
           (int)(y[pidx]/l_c) * NC +
           (int)(z[pidx]/l_c) * NC * NC;
//application of CAS operation to update list/entry
do
{
  // store old entry particle index
  old = entry[cidx];
  // update next particle in list for particle pidx
  list[pidx] = old;
  // try to update entry particle of target cell cidx
  cmp = atomic_cmpxchg(entry+idx,old,gid);
}
// if update failed, repeat the process
while(old != cmp);
```

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

Listing 2: Calculation of the prefix sum of contained particles in OpenCL

```
// check if global index of work item is smaller than number of cells
int gid = get_global_id(0);
if (gid >= n) return;
int lid = get_local_id(0);      // get index of work item in work group
int grid = get_group_id(0);     // get index of work group
int gsize = get_local_size(0); // get size of work group

// if not the first pass of the kernel, i.e prefix sum calculation on
// cell with cells that are not neighboring
if (pass > 0)
{
  // calculate the shift for the partial prefix sum calculation;
  // the shift equals 2**(pass-1)
  int shift = 1<<(pass-1);
  // check if the local cell index is larger than the shift to avoid
  // accessing out of bound memory
  if (gid > shift)
  {
    // calculate the partial prefix sum of the local cell together
    // with the shifted cell
    out[gid] = in[gid] + in[gid - shift];
  }
  // if the shift is larger than the local cell index, the partial
  // prefix sum is unchanged
  else
    out[gid] = in[gid];
}
else                            // the first call of the kernel
{
  // count the number of particles in the local cell
  int loc_count = 0; int gidx = heads[gid];
  while(gidx != -1)
  {
    loc_count++;
    gidx = list[gidx];
  }
  // if the cell is the last cell, set the partial prefix sum for
  // the first cell to zero
  if (gid == n-1)
    out[0] = 0;
  // in all other cases set the prefix sum for the cell with the next
  // higher index to the number of particles in the local cell
  else
    out[gid+1] = loc_count;
}
```

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.
Another feature that was implemented in the benchmark was a resorting algorithm restructuring the particle array in such a way, that particles within a single cell are located in a continous chunk of memory. Particles contained in cells with neighboring indices are stored analogously in the particle array. This resorting algorithm consists of several steps.
In a first step, a prefix sum counting the number of particles in the cells is calculated. Due to the restrictions of OpenCL concerning global operation, the prefix sum has to be calculated in multiple steps. These steps calculate the prefix sum in a tree-based calculation. In the first step the number of particles in each cell is is counted and the partial prefix sum for each cell is set to the number of particles in the previous cell. For the first cell the prefix sum is set to zero. In the subsequent steps a shift $s = 2^{n_s-1}$, where n_s is the step number, is computed. Each partial result is now added to the partial result of the cell that has a distance of s within the cell array. This procedure is repeated for $\max\left(\lceil\log_2\left(n_{cells}\right)\rceil, 1\right)$ times, due to the tree-based computation. For the implementation of the prefix sum calculation in OpenCL refer to listing 2.
The result of the prefix sum calculation is then used to resort the particles within the particle array. Using the prefix sum the required space for each cell within the particle array is determined. In order to simplify the resorting procudure two arrays are used. The first array contains the particles in their unsorted state. With the use of the prefix sum the particles are transfered to the new array, while sorting them so that they are located according to the

Listing 3: Correction of the particles list in cell during resort on the device

```
// heads:  array of ints, containing the entry particles
//         for each cell
// list:   linked list of the particle lists for the cells
// offset: array of integers, containing the prefix sum
// get global index of work item
int gid = get_global_id(0);
// check if work item index is larger then number of cells
if (gid >= nc_total) return;
// check if the cell is not empty
if (heads[gid] == -1) return;
// initialise the start index of the local cell
int i = offset[gid];
// the last index in a cell is either the particle index
// preceeding the start of the next cell or for the last
// cell the highest particle index in the system.
int endpoint = (gid != nc_total-1)?(offset[gid+1]-1):(np-1);
// fill the particle list with intermediate indices
while(i < endpoint)
{
  list[i] = i+1;
  i++;
}
// terminate the particle list of the cell with an invalid index (-1)
list[i] = -1;
```

cell they are sorted into. Figure 2 illustrates this resorting procedure. After the particles are resorted, the particle list for each cell needs to be updated. To find the correct start and end indices of the particles within each cell, the prefix sum is required again. Since the cell contents are stored in a consequence order within the new particle array, each start particle for the list is given by the prefix sum and each end particle is the particle before the prefix sum of the next cell (cmp. Listing 3).

3.2 Organisation and Distribution of Data Structures

An important 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.

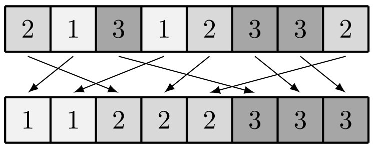


Fig. 2: Resorting of particles from unsorted input particle array to sorted output particle array.

In order to compare a reprensentative set of multi- and many-core architectures, the benchmark was implemented in OpenCL. This allowed to run the benchmark on all test platforms without the need to change the benchmark source code. The OpenCL kernels were implemented in such a way, that the data is either distributed to each cell either on particle or cell basis. This depends on the specific task of each kernel, as e.g. the integration of particles requires only data from a single particle, while e.g. the calculation of interactions requires all particles from a local cell, as well as surrounding cells.

The number of work items per work group are kept constant for all kernels. To calculate the number of work groups n_{wg} required to run a given system size, the number of required work items n_{wi} is divided by the number of work items per work group n_{wig}: $n_{wg} = \frac{n_{wi}}{n_{wig}}$. Depending on the type of kernel, the number of required work items depends on the number of particles or the number of cells in the system. The number of work items per work group is dependent on the given architecture and has an upper limit, which is the reason why for larger system sizes it is not possible to synchronize on a work group level alone, since more than one work group is required to execute the code.

4 Architectures

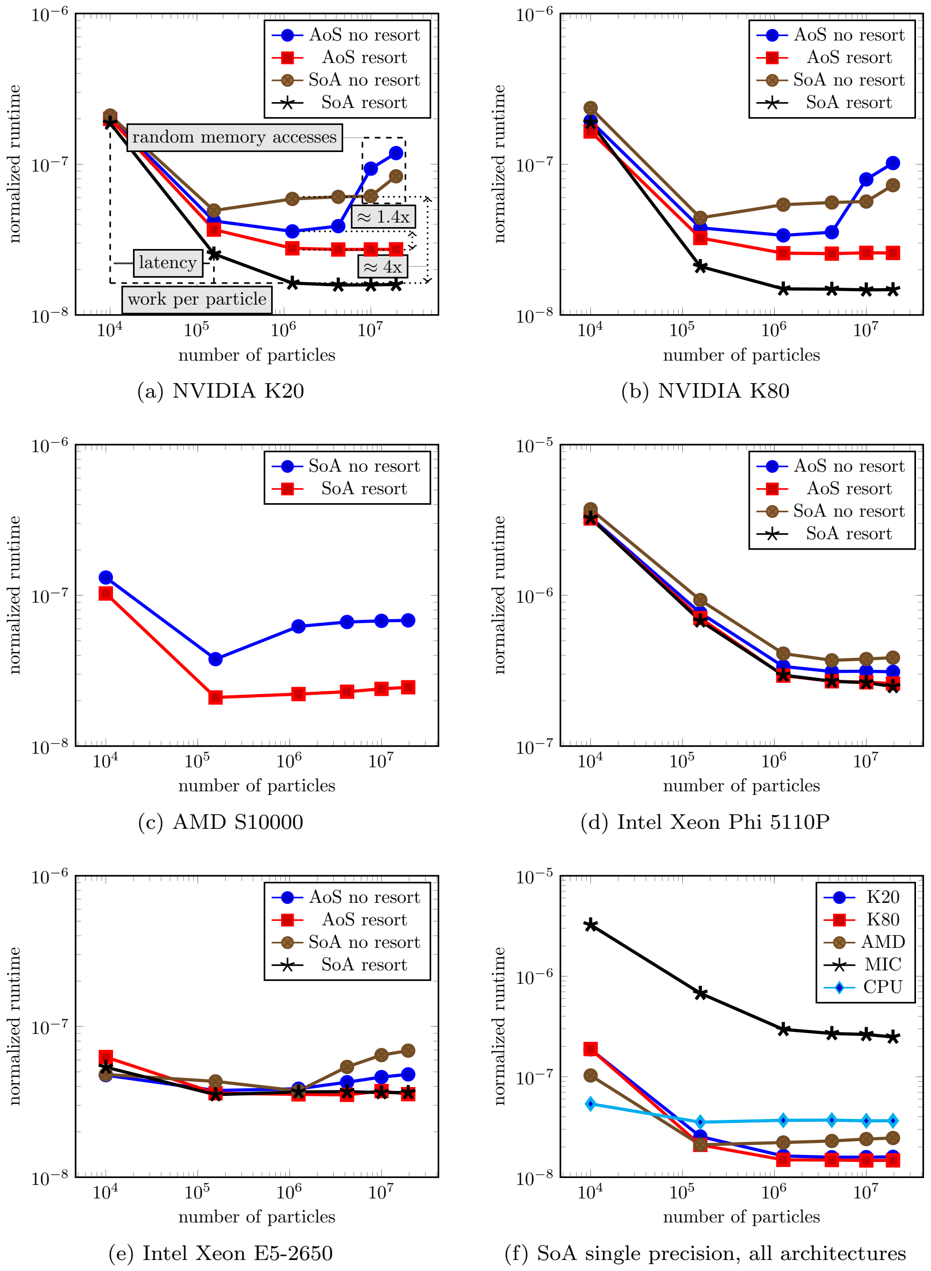
The benchmarks were conducted on three different machines: JURECA [6], JUROPA3 [7] (two of the supercomputer systems at JSC) as well as on a testing system for GPUs. Table 1 shows the specifications for the different architectures used for benchmarking. JUROPA3 has different partitions, that employ the 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 1: 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. 3a - 3f). Here, 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. 3c) 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. 3a) and the K80 GPUs (Fig. 3b) are different versions of the same production line, their results look fairly similar. With the exception of the Xeon E5-2650 CPU



(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. 3: 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.

(Fig. 3e) 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. 3a 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. 3d) 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.

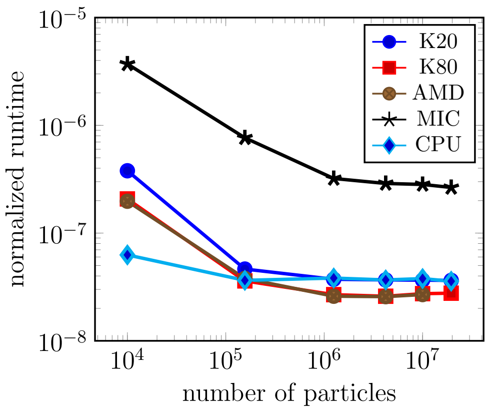


Fig. 4: Performance comparison between all architectures with sorted SoA data structure in double precision arithmetics.

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. 3a, 3b) and CPUs (Fig. 3e).
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 1), 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. 4). 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. 3f). Only for double-precision calculations it is observed that the K20X is only as fast as the CPU (cmp. Fig. 4). 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. 3f). 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 [5].

6 Conclusion

One of the main goals of the present work was to investigate the performance characteristics of a function portable cell based MD program on a set of different 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 a benchmark on several multi- and many-core systems available at Jülich Supercomputing Centre. As a downside of that approach the code was not optimised specifically for any of the architectures and therefore did not present an optimal performance. This especially accounts for the Xeon Phi system, where vectorisation is an essential issue to outperform an implementation without specific optimisation. On the other side this approach offers the possibility to have a comparison between basic features of the different architectures, e.g. memory bandwidth, core speed and effects of data structure layout. To account for different data access patterns, we included the comparison of arrays-of-structures (AoS) and structures-of-arrays (SoA), which is an important issue for, e.g., GPU architectures. It was shown that the SoA layout improves the performance on both the GPU and CPU for the case of non-sorted data and does not degrade performance in the case of sorted data. In this respect it can be concluded that further optimisation targeted to GPU architectures will lead most probable also to a significant performance gain on CPUs. Unfortunately, the future of OpenCL cannot be foreseen at present. E.g., the support of OpenCL for Intel Xeon Phi is already deprecated and so it remains to be seen, if OpenCL will remain to be a valid option supporting future architectures. Nevertheless, at present 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. A most desirable feature would be 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. [1]).

References

1. DOE (2017) Performance Portability WS DOE. URL `https://asc.llnl.gov/DOE-COE-Mtg-2016/`
2. Frenkel D, Smit B (2002) Understanding molecular simulation. From algorithms to applications. Academic Press, San Diego
3. Halver R, Sutmann G (2015) Multi-Threaded Construction of Neighbour Lists for Particle Systems in OpenMP. In: Parallel Processing and Applied Mathematics / Wyrzykowski, Roman (Editor), 11th International Conference on Parallel Processing and Applied Mathematics, Krakow (Poland), 6 Sep 2015 - 9 Sep 2015, DOI 10.1007/978-3-319-32152-3_15
4. Hockney RW, Eastwood JW (1981) Computer simulation using particles. McGraw-Hill, New York
5. Intel (2017) Intel OpenCL SDK. URL `https://software.intel.com/en-us/articles/opencl-drivers`
6. JSC (2017) JURECA. URL `http://www.fz-juelich.de/ias/jsc/jureca`
7. JSC (2017) JUROPA3. URL `http://www.fz-juelich.de/ias/jsc/EN/Research/HPCTechnology/ClusterComputing/JUROPA-3/JUROPA-3_node.html`
8. Rapaport D (2001) The Art of Molecular Dynamics Simulation. Cambridge University Press, Cambridge
9. Sutmann G (2002) Classical molecular dynamics. In: Grotendorst J, Marx D, Muramatsu A (eds) Quantum simulations of many-body systems: from theory to algorithms, NIC, Jülich, vol 10, pp 211–254
10. Sutmann G, Stegailov V (2006) Optimization of neighbor list techniques in liquid matter simulations. J Mol Liq 125:197–203