Kokkos Implementation of an Ewald Coulomb Solver and Analysis of Performance Portability

Rene Halver[a], Jan H. Meinke[a], Godehard Sutmann[a,b]

[a] *Jülich Supercomputing Centre, Institute for Advanced Simulation, Forschungszentrum Jülich, 52425 Jülich, Germany*
[b] *ICAMS, Ruhr-Universität Bochum, 44801 Bochum, Germany*

Abstract

We have implemented the computation of Coulomb interactions in particle systems using the performance portable C++ framework Kokkos. For the computation of the electrostatic interactions in particle systems we used an Ewald summation. This implementation we consider as a basis for a performance portability study. As target architectures we used Intel CPUs, including Intel Xeon Phi, as well as Nvidia GPUs. To provide a measure for performance portability we compute the number of needed operations and required cycles, i.e. runtime, and compare these with the measured runtime. Results indicate a similar quality of performance portability on all investigated architectures.

1. Introduction

The development of modern computer architectures shows a clear trend towards increased complexity and heterogeneity. This increases the complexity of efficient code development for multiple architectures that takes advantage of all available components. GPUs, for example, are powerful processors available in cell phones as well as supercomputers usually requiring their own programming model. As a matter of fact GPUs have become more and more important as a source of computing power in supercomputers, as can be seen in the increase of systems using GPUs in the Top500 list [1] over the past ten years. While in November 2008 there was no system that included GPUs, in November 2013 there were 39 systems, and in the current list from November 2018 126 systems contained GPUs. There are many ways of programming GPUs but unfortunately few are even function portable without large changes to the source code [2]. This matter becomes even worse if we want to write code for different kind of accelerators, e.g., Intel's Xeon Phi series.

In the domain of particle simulation methods of complex systems, electrostatic interactions represent a class of algorithms of high computational com-

Email addresses: r.halver@fz-juelich.de (Rene Halver), j.meinke@fz-juelich.de (Jan H. Meinke), g.sutmann@fz-juelich.de (Godehard Sutmann)

plexity. This arises as a result of pair-wise interactions between all particles in a system, which basically scale as $\mathcal{O}(N^2)$. More efficient methods can be reduced to $\mathcal{O}(N\ \log(N))$ or even $\mathcal{O}(N)$ but come with a large implementation effort [3, 4, 5, 6, 7]. In the present paper we consider the Ewald summation method, which is suitable for particle systems in three dimensions under periodic boundary conditions and which can be optimised by proper choice of paramters to $\mathcal{O}(N^{3/2})$ (essentially there are also formulations for one- or two-dimensional systems, which we do not consider here). The basic structure of the Ewald summation is sufficiently transparent and not too complex, allowing an analysis of the operational count and providing insight into the procedure to measure performance portability.

Performance portable approaches have recently attracted attention [8], where Raja [9, 10] and Kokkos [11, 12] are frameworks, offering a set of C++ software abstractions for code execution and memory management.

In this paper, we compare the performance of an Ewald sum implemented in Kokkos on a variety of architectures including Intel Xeon Phi Knights Landing and Nvidia GPUs. We first introduce the problem of a system of electric charges with periodic boundary conditions and show how the Ewald sum can be efficiently used for its calculation (Sec. 2). Then we provide a quick overview of Kokkos and its main features (Sec. 3). In Sec. 4 we establish a base line for the achievable performance. Afterwards we present our implementation and show our performance benchmarks (Sec. 5).

2. Calculating Long-range Interactions with Periodic Boundaries

When computing energies and forces in particle systems composed of N particles, which are dominated by long range interactions, each particle i gets partial contributions of each other particle $j \in [1, N]$. Long range interactions arise when the potential energy function $\phi(r)$ decays slower than $1/r^d$, where d is the dimension of the system and r is the distance from a point in space to a particle. Here, we consider electrostatic potentials created by point charges, for which the potential energy at a point r in free space is given by $\phi(r) = q_j/|r - r_j|$ which leads to a total electrostatic energy $U = 1/2 \sum_{i,j} q_i \phi(r_{ij})$. When simulating bulk systems, the number of particles in a simulation is always small, compared with laboratory samples and therefore, in order to avoid surface effects, periodic boundary conditions are often applied [13] and the electrostatic potential energy at particle position $\mathbf{r}_i$ can formally be written as

$$\phi(\mathbf{r}_i) = \sum_{\mathbf{n}}{}^{\dagger} \sum_{j=1}^{N} \frac{q_j}{\|\mathbf{r}_{ij} + \mathbf{n}L\|_2}$$

where $\mathbf{n} \in \mathbb{Z}^3$ is a so called lattice vector, L the length of the (cubic) system and "†" indicates that $j \neq i$ for $\|\mathbf{n}\|_2 = 0$. This sum cannot be evaluated by a straightforward summation rule, since (i) the first sum is formally over an infinite number of lattice vectors; and (ii) the lattice sum is conditionally convergent,

i.e., the result depends on the order of summation. Ewald proposed [14] a way to overcome the conditional convergence by subdividing the expression into a short range and a long range part, which is introduced via a splitting function, $f(r)$, which decays to zero within a finite range. A function $u(r) = 1/r$ can then be rewritten as $u(r) = f(r)/r + (1 - f(r))/r$. The first term is short range (since it decays to zero), while the second one is long range (since asymptotically it decays as $1/r$). This reformulation has the advantage that it can be transformed into an unconditionally convergent sum for a proper choice of f. Originally, $f(r) = \mathrm{erfc}(\alpha r)$ was chosen, where α is a splitting parameter, controlling the width of the short range part. The long range part can be elegantly computed in Fourier space, which leads to [13]

$$\phi(\mathbf{r}_i) = \sum_{j=1}^{N} {\sum_{\mathbf{n}}}^{\dagger} q_j \frac{\mathrm{erfc}(\alpha \|\mathbf{r}_{ij} + \mathbf{n}L\|)}{\|\mathbf{r}_{ij} + \mathbf{n}L\|_2} + \frac{4\pi}{L} \sum_{|\mathbf{k}|\neq 0} \sum_{j=1}^{N} \frac{q_j}{|\mathbf{k}|^2} e^{-\frac{|\mathbf{k}|^2}{4\alpha^2}} e^{i\mathbf{k}\mathbf{r}_{ij}} - q_i \frac{2\alpha}{\sqrt{\pi}} \quad (1)$$

The last term corresponds to a correction for particle i, which also appears in the k-space summation (second term). For practical computations the infinite sums (over $\mathbf{n}$ and $\mathbf{k}$) have to be approximated. For large arguments $\mathrm{erfc}(x)$ decays as a Gaussian, as it does the k-space summation. Therefore, both sums can be limited to a finite range of values, which still allows for control of approximation error. In most cases, due to the spherical symmetry of $\mathrm{erfc}(x)$ and a fast decay, the first sum can be restricted to contributions within a spherical region of radius R_c. Furthermore, it can be shown that via a proper set of parameters [15, 16], the computational complexity is reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N^{3/2})$.

3. Kokkos at a Glance

Kokkos uses C++ to provide an abstraction of parallel algorithms, their execution and memory spaces. The basic algorithms include `parallel_for`, `parallel_reduce`, and `parallel_scan`. Each of these algorithms can be executed in different execution spaces, for example, using an OpenMP execution space on the CPU or a CUDA execution space on an Nvidia GPU.

CPUs and GPUs use different approaches to vectorization. CPUs use a single instruction multiple data (SIMD) paradigm. GPUs use a single instruction multiple threads paradigm (SIMT). These two approaches lead to different preferred memory layouts.

On GPUs neighboring threads should access consecutive memory. Thread `i` should access `a[i]` and thread `i+1` `a[i+1]`, but on a CPU this pattern prevents vectorization and can introduce an unnecessary dependency. If `a[i]` and `a[i+1]` belong to the same cache line and thread `i` writes to `a[i]` it invalidates the entire cache line. If thread `i+1` wants to access `a[i+1]` it first needs to read the entire cache line again. This effect is called false sharing [17]. On CPUs a single thread should deal with a chunk of data. The effects due to

different memory layout requirements become even more pronounced for multi-dimensional data. Note that in Kokkos, the left-most index is assumed to be the one over which parallelization is performed. This needs to be taken into account for the implementation of the algorithms and the memory layout.

To accommodate different memory layouts and memory locations Kokkos introduces `Views`. A `View` is a thin wrapper around the data. It knows its dimensionality, its sizes, its layout, and its memory space.

```
Kokkos::View<double*> v(n);
```

for example, initializes a one dimensional array of doubles of size n in the default execution space, which can be set at compile time. A `View` can be mirrored on the host side. In GPU computing it is not uncommon to initialize data on the host, transfer them to the GPU, perform computations on the GPU, and transfer the results back. A mirrored `View` can do just that. Any transfer between a `View` and its mirror needs to be done explicitly using `Kokkos::deep_copy`.

`Views` are not limited to a single dimension. Multidimensional `Views` can be declared with a combination fixed (compile time) and variable (run time) dimensions. An array of 3d coordinates could, e.g., be specified as

```
Kokkos::View<double*[3]> r(n);
```

The code shown in Listing 1 creates a 1d `View` and a host mirror of it, fills the mirrored `View` with random numbers, copies the numbers to the `View`, sums them up in parallel, and gets the result.

If the program is compiled for OpenMP, the mirror `View` becomes an alias and the deep copy does not have to do anything, but if the program is compiled for CUDA, the original `View` lives on the GPU and the deep copy transfers the data from the CPU to the GPU.

The developer is not limited to the default execution space but execution and memory spaces have to been known at compile time. `Views` can take a memory space and algorithms an execution space as argument. If the `parallel_reduce` in Listing 1 should always execute on an Nvidia GPU, we would change the definition of `v` to include the CUDA memory space and the call to `parallel_reduce` would include the CUDA execution space as can been seen in Listing 2

4. Achievable Performance

To determine how well our implementation takes advantage of the available hardware, we need to know what the hardware is capable of. Theoretical floating point peak performance is *not* a good measure of the performance that is achievable for a particular algorithm. The performance of many applications is limited by the available memory bandwidth. For these the floating point peak performance is obviously irrelevant, but even if the application is compute bound the mix of operations is crucial. If the calculation is dominated by square roots

Listing 1: Reduction using Kokkos. This program can be executed using OpenMP on a CPU or on a GPU. Second level curly brackets are needed to ensure deallocation of views before calling `Kokkos::finalize`.

```
#include <random>
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]){
  Kokkos::initialize(argc, argv);
  {
    std::default_random_engine generator;
    std::uniform_real_distribution<double> uniform_dist
        (0, 1);
    auto uniform = [&]{return uniform_dist(generator);};
    int n = 1024;
    double sum = 0;
    // Create a view in the default memory space
    Kokkos::View<double*> v("v", n);
    // Create a mirror of v in host memory
    auto h_v = Kokkos::create_mirror_view(v);
    for(int i = 0; i < n; ++i) h_v(i) = uniform();
    // Copy data from host to device if necessary
    Kokkos::deep_copy(v, h_v);
    // Parallel reduction in default execution space
    Kokkos::parallel_reduce(n, KOKKOS_LAMBDA(int i,
        double& localSum){
      localSum += v(i);
    }, sum);
    std::cout << "The average value of the elements of v
        is " << (sum / n) << ".\n";
  }
  Kokkos::finalize();
}
```

Listing 2: Explicit memory and execution spaces. Changes compared to Listing 1 to always use CUDA for the reduction. `Kokkos::CudaSpace` has to be specefied in the definition of the `View` and `Kokkos::Cuda` has to be passed to the `Kokkos::RangePolicy`, which also needs to be given explicitely.

```
...
// Create a view in the CUDA memory space
Kokkos::View<double*, Kokkos::CudaSpace> v("v", n);
...
// Parallel reduction in CUDA execution space
Kokkos::parallel_reduce(Kokkos::RangePolicy<Kokkos::
    CUDA>(n), KOKKOS_LAMBDA(int i, double& localSum){
  localSum += v(i);
}, sum);
...
```

or exponentials, for example, it does not matter how quickly a compute device can calculate multiplications and additions. To estimate the minimum number of cycles needed for the Ewald summation (Eq. 1), we use vendor information and mini benchmarks for the set of operations given below.

In these benchmarks we first initialize an array of elements to some range of values and then loop over this array applying the operation in question one to a few times. The idea is to access data from cache or registers to minimize the effect of memory bandwidth and latency. We check that vectorized versions of the functions are used where available. The important operations are multiplication, division, square roots, exponentials (exp), sine (sin) and cosine (cos), and the error function (erfc). Table 1 lists the duration of an operation in cycles for each device. For operations for which we found information from the vendors, the values are listed in parenthesis as well. In the following sections we look at the number of instructions performed by the Ewald solver.

4.1. Ewald Solver

The Ewald solver consists of a k-space (Fourier space) and a real-space part (cmp. Eq. 1). Let N be the number of particles in the central cell and N_k be the number of wave vectors.

4.1.1. Real-space Contributions

To calculate a single particle-particle interaction energy, we first need to calculate the distance between the particles (c.f. first term of Eq. 1). In our implementation the central cell is large enough that we do not need to add additional image cells and thus do not have contributions of the type $\boldsymbol{n}L$. A distance calculation consists of 3 subtraction, 2 multiply-adds, 1 multiplication and a square root (sqrt). For the particles within a cutoff radius defined by a tunable parameter α, we then calculate the error function (erfc) of the distance, divide by it, and multiply the result by the charge of particle j. All these partial

Table 1: Average duration in cycles per core or streaming multiprocessor (SMs) for additions, multiplication, (fused) multiply-add, division, square roots, exponentials, sin and cos, and the error function. All values are approximate. The number of cycles is calculated as the number of cycles per vector instruction divided by the width of the vector. A division using AVX512 instruction on Skylake-X (SKX), for example, takes 16 cycles and does 8 division in parallel during those 16 cycles. We therefore have 2 cycles per division. The numbers in parenthesis are from [18, 19]. The line for SKXZ shows results when the compiler was asked for a high usage of zmm registers. On the CPU architectures, we used a single core for the measurements. On the GPUs, we used all SMs and divided the throughput by the number of SMs.

	add	mul	(f)ma	div	sqrt	exp	sin	cos	erfc
SKX	$(\frac{1}{16})$	$(\frac{1}{16})$	$(\frac{1}{16})$	2.0 (2)	3.06 (3)	12.5	11.58	12.63	16.77
SKXZ	$(\frac{1}{16})$	$(\frac{1}{16})$	$(\frac{1}{16})$	0.96	1.20	6.32	6.51	6.76	9.25
Haswell	$(\frac{1}{8})$	$(\frac{1}{8})$	$(\frac{1}{8})$	3.97	3.97	2.66	2.84	3.35	5.85
KNL	$(\frac{1}{16})$	$(\frac{1}{16})$	$(\frac{1}{16})$	1.12	2.04	3.82	6.94	7.09	9.20
Kepler	$(\frac{1}{64})$	$(\frac{1}{64})$	$(\frac{1}{64})$	0.15	0.21	0.37	0.55	0.55	1.25
Volta	$(\frac{1}{32})$	$(\frac{1}{32})$	$(\frac{1}{32})$	0.30	0.31	0.57	0.84	0.84	2.04

Table 2: Minimum number of cycles needed. This table lists the minimum number of cycles needed to perform the real- and k-space calculations of the Ewald sum for each used architecture based on a set of mini benchmarks. N is the number of charges. V_f is the fraction of the volume of the base cell that is within the cut off radius of the real space calculation. N_k is the number of wave vectors and V_{fk} is volume fraction of the k-space used for the inner sum.

Arch.	Real space calculation	K-Space Calculation
Haswell	$(4.72 + 10.07V_f)N^2 + N/8$	$(0.38 + (15.70 + 6.57N)V_{fk})N_k$
Skylake-X	$(1.58 + 10.34V_f)N^2 + N/16$	$(0.19 + (9.76 + 13.46N)V_{fk})N_k$
Kepler	$(0.30 + 1.43V_f)N^2 + N/64$	$(0.047 + (0.30 + 1.43N)V_{fk})N_k$
Volta	$(0.50 + 2.40V_f)N^2 + N/32$	$(0.50 + 2.40V_{fk})N^2 + N/32$
KNL	$(2.415 + 9.39V_f)N^2 + N/16$	$(2.415 + 9.39V_{fk})N^2 + N/16$

results need to be added up for each particle i and the result is multiplied by the charge of particle i and a constant. Finally, the potential energy of all particles needs to be summed up to get the total energy. The total amount of operations $\#_r$ is then found as

$$\begin{aligned}\#_r &= N^2 \times (3 \times (\text{sub}) + 2 \times (\text{multiply-add}) + 1 \times (\text{mul}) + 1 \times (\text{sqrt}) \\ &\quad + V_f \times (2 \times \text{mul} + 1 \times (\text{div}) + 1 \times (\text{erfc}))) + N \times (\text{multiply-add})\end{aligned}$$

where V_f is the fraction of the total volume within the cutoff radius. The resulting number of cycles can be found in the central column of Table 2.

4.1.2. K-space Contributions

The second term of Eq. 1 contains 2 nested sums. The outer sum is over $N_k = (2k_{\text{int}}+1)^3$, where k_{int} is the integer ceiling of $k_{\max}$ and $k_{\max}$ is determined by the required precision and the factor α mentioned in the Sec. 2. It requires

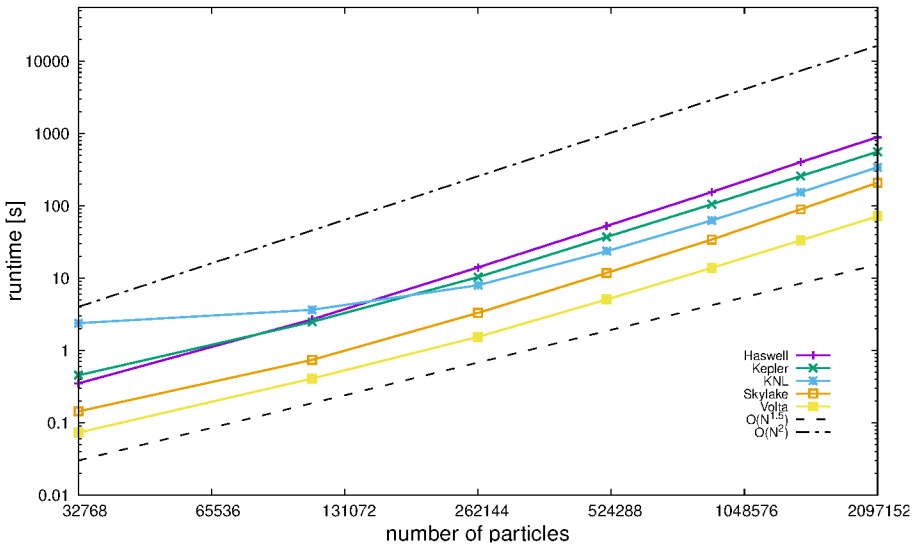

Figure 1: Comparison of the runtime for different systems sizes on all tested architectures. For reference two guide lines showing a complexity of $\mathcal{O}(N^{3/2})$ and $\mathcal{O}(N^2)$ are shown.

the calculation of the square of the length of the k-vector (2 multiply-add, 1 multiplication), which is used twice. Only wave vectors with a length less than $k_{\max}$ are included for the remaining calculations. There are 3 divisions, 7 multiplication, 2 multiply-adds, and 1 exponential. The argument of the inner sum includes the dot product between $\boldsymbol{k}$ and $\boldsymbol{r}_i$(2 multiply-add, 1 multiplication). The exponential of the complex argument is calculated using 1 sin and 1 cos. This is then multiplied by q_i and summed up (1 multiply-add). The argument of the inner sum is executed NN_kV_{fk} times, where $V_{fk} = \frac{4\pi\, k_{\max}^3}{3N_k}$. In total the number of operations $\#_k$ becomes

$$\begin{aligned}\#_k \;=\; & N_k \times (\,3\,(\text{sub,multiply-add,mul}) + V_{fk} \times (\,9\; \times (\text{sub,multiply-add,mul}) \\ & +3 \times (\text{div}) + 1 \times (\text{exp}) + N \times (1 \times (\text{sin}) + 1 \times (\text{cos}) \\ & +3 \times (\text{sub,multiply-add,mul}))\,)\,)\end{aligned}$$

The formulas to determine the amount of cycles required to compute the k-space contribution on each architecture are presented in the right column of Table 2.

5. Results

The program was benchmarked on five different architectures: three different Intel CPUs and two different Nvidia GPUs. The benchmarks were performed on the JURECA and JUWELS clusters at the Jülich Supercomputing Centre [20]. On JURECA the tests were run on i) a CPU compute node, equipped with two Intel Xeon E5-2680 v3 Haswell CPUs, ii) a GPU node equipped with

Table 3: Runtime in seconds and performance portability for calculating the electrostatic potential using the Ewald sum for Intel Haswell (HSW), Skylake-X (SKX), Xeon Phi (KNL), Nvidia Kepler (K80), and Volta (V100). The performance portability is calculated according to Eq. 3 for each problem size based on the reached percentage of the achievable performance as determined in Sec. 4. The last column provides the performance portability factor according to Pennycook[21] for the given problem size.

N	HSW	SKX	KNL	K80	V100	$P(a,p)$
32	0.349	0.143	2.378	0.453	0.073	0.060
48	2.682	0.737	3.634	2.482	0.409	0.190
64	14.06	3.300	7.966	10.32	1.536	0.286
80	52.63	11.79	23.57	36.94	5.053	0.309
96	154.55	34.16	62.70	105.05	13.84	0.322
112	400.97	89.40	153.95	257.54	33.24	0.320
128	885.55	208.5	339.28	559.47	71.62	0.320
160	3371.65	774.5	1301	2129.78	287.19	0.314
192	9987.50	2270	3910	6265.08	837.75	0.316

two NVIDIA K80 (Kepler) cards, of which only a single one was used, and iii) a booster node consisting of a single Xeon-Phi 7250-F Knights Landing (KNL) processor. The nodes used on JUWELS are i) a CPU node containing two Intel Xeon Platinum 8168 Skylake-X (SKX) processors and ii) a GPU node with four NVIDIA V100 (Volta) cards, of which again only one is used for the benchmarks.

For each benchmark the same source code was used, containing only minor adjustments concerning the used *ExecutionSpace* and *MemorySpace*, depending on the use of i) a GPU architecture and ii) the use of the host_mirror mechanic of Kokkos. The possibility to change the memory layout is also included. For the CPU benchmark runs a complete node was used, i.e., two processers of Haswell and Skylake and one KNL processor, while for the GPU benchmarks only a single GPU was used, i.e., 'half' a K80 and a single Volta V100 card. Therefore the presented runtimes are per-node runtimes, not per processor runtimes.

For the benchmarks a cubic NaCl crystal was simulated, for which the exact solution to the Coulomb potential is known, so that the accuracy of the computed solution could be compared to the exact solution. During the benchmark the size of the crystal was increased by increasing the edge length L of the crystal, thereby increasing the number of particles as L^3. Due to the nature of the system, the contribution of the Fourier-space is much smaller than the real-space contribution, due to screening effects. This does not decrease the computational demand of the algorithm if a given accuracy has to be achieved.

In the optimal case the Ewald solver shows an complexity of $\mathcal{O}(N^{3/2})$, which depends on an optimal choice of the splitting parameter α, the real-space cut-off radius r_c and the k-space cut-off $\mathrm{k}_{\max}$. Due to the implementation of the real-space computation, which is basically a direct solver of complexity $\mathcal{O}(N^2)$, in our results it can be seen that for larger systems sizes the resulting runtimes

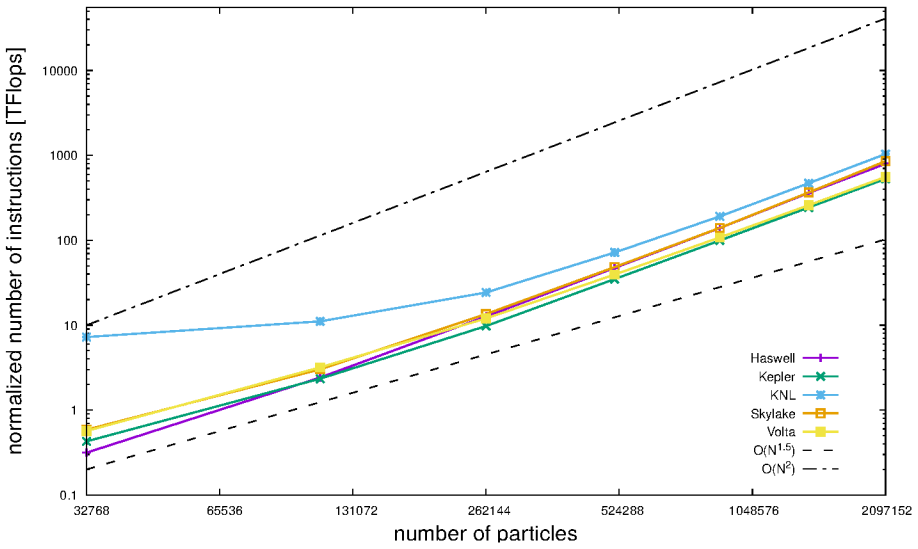

Figure 2: Runtime on each architecture normalized by the nominal peak performance of the architecture. If the achieved performance on two architectures relative to the nominal peak performance is equal, the lines should be overlapping.

behave more like $\mathcal{O}(N^2)$ than $\mathcal{O}(N^{3/2})$ (see Fig. 1). The figure also shows the expected relations of runtime to architecture, as the more powerful architectures shows faster runtimes than the less powerful ones. Another detail that can be seen is that the GPUs show the same scaling behavior as the CPUs, with the Volta card resulting in the shortest runtimes of all architectures.

In order to achieve some more insight into the performance portability between the same types of architectures, i.e., CPU-CPU, GPU-GPU, and across types of architectures, i.e., CPU-GPU, we first consider the peak-performance normalized runtime on the architectures since the nominal peak performance is often used for such comparisons. The runtime can be expressed by the number of operations divided by a fraction γ, where γ is a measure for the proximity to maximum performance, $t_{\text{run}} = \frac{N_{\text{instruc}}}{\gamma P_{\text{peak}}}; \gamma \in [0, 1]$. This can be rewritten as

$$t_{\text{run}} \cdot P_{\text{peak}} = N_{\text{instruct}}/\gamma. \tag{2}$$

To compare the performance portability of the implementation the runtimes need to be compared between the different architectures. Assuming that on each architecture a comparable number of instructions are executed for a given simulation, one can assume that the product of runtime t_{run} and the nominal peak performance P_{peak} will be equal across all platforms, if the reached relative performance γ is equal (equation 2).

For a qualitative comparison based on this thought, the runtimes are multiplied with the nominal peak performance for each of the architectures given in table 4. All peak performance data is with regard to double precision computa-

Table 4: Nominal peak performance data for each of the architectures used in the benchmarks

Architecture	Note	Nominal peak performance [TFlops/sec]
Haswell	complete node (two processors)	0.9
Kepler	single GPU (half a K80 card)	0.945
KNL	one processor	3.05
Skylake	complete node (two processors)	4.1
Volta	single GPU	7.8

tions, which are used in the code and are necessary to get the precision we want. The resulting plot (Fig. 2) shows that the normalized number of instructions computed for each of the different architectures is similar. It can also be seen that the lines for the same type of architecture (CPU, GPU) are nearly identical to each other, indicating on a qualitative level, that the achieved relative performance is similar within a given type of architecture (with the exception of KNL).

No significant difference could be measured between the variants of the code using the Kokkos host_mirror functionality and using the option with unified memory. While this was expected on CPU architectures, it was interesting to see that required hard copies of data between devices were handled in a very efficient way on GPU architectures independent of the GPU model.

We can can now compare the reached performance to the achievable performance for the different architectures. If the code is performance portable, the reached percentage of the achievable performance should be similar across the different architectures. Fig. 3 shows that the percentages of the achievabel performance are close together, indicating that the code indeed is performance portable.

Our estimate of the achievable performance assumes full vectorization, pipelining, and the usage of all compute resources. For small systems this will not be true, which explains the much smaller percentage achieved.

The reason the percentage of achieved performance decreases for large L on Skylake-X needs to be investigated. It could be related to changing frequencies due to thermal reasons or cache effects, which are not covered by our performance model. On average the reached fraction across all platforms is ≈ 0.26, including the smaller systems that show a lesser fraction of achieved performance than the larger ones. Pennycook et al. [21] suggest the harmonic mean over the performance values for a given set of platforms H as measure for performance portability:

$$P(a, p, H) = \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}}, \tag{3}$$

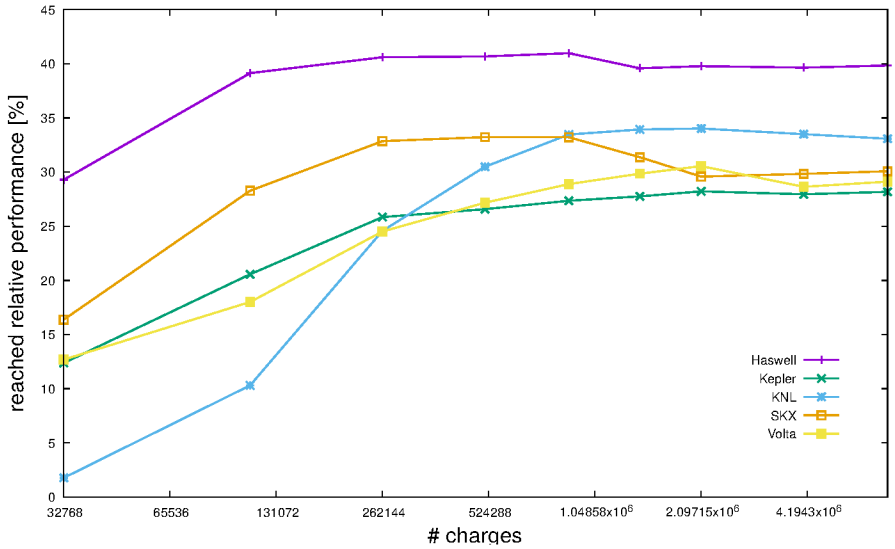

Figure 3: Reached fraction of the achievable performance for Intel Skylake-X (SKX), Intel Xeon Phi (KNL), and Nvidia Volta.

where a is an application that solves problem p. In our case $e_i(a, p)$ is the reached percentage of the achievable performance for a given L.

It is interesting that while the results in Fig. 2 suggest that the fraction of the peak performance achieved on Volta is better than on CPUs, the findings in Fig. 3 indicate that the code utilizes the CPU architecture just as well as the GPU one. In fact, the program takes better advantage of Haswell and KNL than of Volta and SKX. Nevertheless the total runtime is smaller on Volta then on Skylake as can be seen in Fig. 1.

6. Discussion and Conclusions

The performed benchmarks indicate that it is possible to write a performance portable Ewald solver code with Kokkos that can utilize different architectures without the requirement of intensive code adaptations for each of the architectures. Of course it might be possible to write more efficient code specialized for certain architectures, but this kind of code would lose the advantage of versatility concerning architectures it could usefully run on. The quantitative analysis shows that the expected runtime for Skylake is three times longer than for Volta, which our measurement confirm. On the other hand, the nominal peak performance predicts only a factor of two leading to the discrepancy with Fig. 2.

It is noticeable that the performance of the KNL is worse than the performance of the other architectures when using smaller number of particles. This could be related to a massive overhead in the administration of threads, as each

thread might not be fully utilized due to the smaller amount of work for each thread. For larger system sizes, it can be seen that the KNL behaves comparable to the other architectures, with regard to the scaling behavior.

Implementing the Ewald solver with Kokkos was slightly more difficult than implementing the code with OpenMP, as the correct usage of the corresponding *parallel_for* and *parallel_reduce* constructs is a bit more intricate than the usage of OpenMP pragmas. The advantage is that they can be used on GPUs as well if certain restrictions regarding memory access are obeyed. As can be seen from our benchmarks this can be done with nearly no loss of relative performance on the different architectures.

Our first results indicate that implementations of algorithms based on Kokkos on a given architecture allows a simplified way of porting to other architectures without a redesign of code (e.g., porting an efficient code for GPUs from standard C++ to CUDA). This allows for an easier transition to other (future) architectures and to investigate and utilise this hardware in an earlier stage of their availability.

For the future it needs to be examined, if the $\mathcal{O}(N^{3/2})$ complexity can be achieved for the Kokkos implementation, e.g., by implementing nearest-neighbor lists for the real-space contribution computation. Also, it would be beneficial to implement more advanced Coulomb solvers, like PME, P3M or the fast multipole method, with Kokkos to see if the solvers can also be used performance portable. With regard to Kokkos features, it will also be investigated how large the impact is of choosing an unsuitable memory layout for a given architecture.

Another thing that needs to be investigated, is the question, if the performance portability of Kokkos can be maintained, while providing an inter-node parallelization with MPI. This will be a major issue in connection with modular HPC systems, consisting of CPU nodes and GPU nodes, in order to utilize them to the best possible degree.

[1] Top500, TOP500 Supercomputer Sites, https://www.top500.org/.

[2] R. Halver, W. Homberg, G. Sutmann, Function portability of molecular dynamics on heterogeneous parallel architectures with OpenCL, J Supercomput 74 (4) (2018) 1522–1533. doi:10.1007/s11227-017-2232-2.

[3] M. Deserno, C. Holm, How to mesh up Ewald sums. I. A theoretical and numerical comparison of various particle mesh routines, J. Chem. Phys. 109 (1998) 7678.

[4] B. Luty, M. Davis, I. Tironi, W. van Gunsteren, A comparison of particle-particle, particle-mesh and Ewald methods for calculating electrostatic interactions in periodic molecular systems, Mol. Sim. 14 (1994) 11–20.

[5] E. L. Pollock, J. Glosli, Comments on P^3M, FMM, and the Ewald method for large periodic Coulombic systems, Comp. Phys. Comm. 95 (1996) 93–110.

[6] A. Y. Toukmaji, J. A. B. Jr., Ewald summation techniques in perspective: a survey, Comp. Phys. Comm. 95 (1996) 73–92.

[7] A. Arnold, F. Fahrenberger, C. Holm, O. Lenz, M. Bolten, H. Dachsel, R. Halver, I. Kabadshow, F. Gähler, F. Heber, J. Iseringhausen, M. Hofmann, M. Pippig, D. Potts, G. Sutmann, Comparison of scalable fast methods for long-range interactions, Phys. Rev. E 88 (2013) 063308.

[8] `https://performanceportability.org/perfport/frameworks`.

[9] D. Beckingsale, R. Hornung, T. Scogland, A. Vargas, Performance portable c++ programming with raja, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP '19, ACM, New York, NY, USA, 2019, pp. 455–456. doi:10.1145/3293883.3302577.
URL `http://doi.acm.org/10.1145/3293883.3302577`

[10] `https://github.com/LLNL/RAJAPerf`.

[11] H. Carter Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202–3216. doi:10.1016/j.jpdc.2014.07.003.

[12] `https://github.com/kokkos/kokkos`.

[13] D. Frenkel, B. Smit, Understanding Molecular Simulation: From Algorithms to Applications, 2nd Edition, Academic Press, San Diego, 2001.

[14] P. P. Ewald, Die Berechnung optischer und elektrostatischer Gitterpotentiale, Annalen der Physik 369 (3) (1921) 253–287. doi:10.1002/andp.19213690304.

[15] D. Fincham, Optimisation of the Ewald Sum for Large Systems, Molecular Simulation 13 (1) (1994) 1–9. doi:10.1080/08927029408022180.

[16] G. Sutmann, Molecular Dynamics - Vision and Reality, in: J. Grotendorst, S. Blügel, John von Neumann-Institut für Computing (Eds.), Computational Nanoscience: Do It Yourself! Winter School, 14 - 22 February 2006, Forschungszentrum Jülich, Germany ; Lecture Notes, no. 31 in NIC Series, NIC-Secretariat, Research Centre Jülich, Jülich, 2006, oCLC: 181556319.

[17] J. Torrellas, H. S. Lam, J. L. Hennessy, False sharing and spatial locality in multiprocessor caches, IEEE Transactions on Computers 43 (6) (1994) 651–663. doi:10.1109/12.286299.

[18] Intel, Intel® 64 and IA-32 Architectures Optimization Reference Manual.

[19] Nvidia, CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (Mar. 2019).

[20] JSC, Forschungszentrum Jülich - Jülich Supercomputing Centre (JSC), https://www.fz-juelich.de/ias/jsc/ (2019).

[21] S. J. Pennycook, J. D. Sewall, V. W. Lee, A Metric for Performance Portability, arXiv:1611.07409 [cs]arXiv:1611.07409.