# Unified Code Generation for the Parallel Computation of Pairwise Interactions using Partial Evaluation

Jonas Schmitt*, Harald Köstler*, Jan Eitzinger† and Richard Membarth‡
*Chair for System Simulation, University of Erlangen-Nürnberg
†Regional Computing Center Erlangen (RRZE), University of Erlangen-Nürnberg
‡German Research Center for Artificial Intelligence (DFKI), Saarland University

***Abstract*—The evaluation of pairwise interactions is fundamental for the simulation of most molecular processes. Their efficient computation is therefore crucial for the overall performance of these simulations on modern computer architectures. We show how code for the computation of pairwise interactions on parallel and heterogeneous platforms can be generated from a unified base through partial evaluation of higher-order functions. For this purpose we introduce a complete implementation of the neighbor list algorithm based on the AnyDSL framework, from which we are able to generate executables for both CPU and GPU through compile-time specialization. Furthermore, we discuss the advantages and disadvantages of our approach and compare it with the miniMD simulation package from the Mantevo project, which is implemented in the C++ programming language and uses a similar computational core as the widely used molecular dynamics package LAMMPS. Finally, we assess the performance of our implementation in a number of test cases on modern CPU and GPU hardware.**

## I. Introduction

In 2004, Colella identified seven numerical methods of major importance in science and engineering which became famous as the seven dwarfs of HPC [1]. N-body methods which depend on computing the interactions between a large number of discrete points are one of these dwarfs. Many interesting phenomena can not be described in the continuous domain and therefore need to be modeled as a system of interacting bodies (particles), from galaxies and stars to molecules and atoms. Here the number of particles can readily reach the order of magnitude of multiple millions or more. For example, every cubic meter of gas contains approximately $2.69 \cdot 10^{25}$ molecules (Loschmidt constant) and our own galaxy, the milky way, alone consists of 200 billion stars. The fundamental mathematical model for the simulation of these systems is Newton's second law (Equation 1), which describes the dependence of a particle's acceleration from the force acting on it.

$$\boldsymbol{F} = m \cdot \frac{d\boldsymbol{v}}{dt} = m \cdot \boldsymbol{a} \tag{1}$$

To simulate its development over time, this system of ordinary differential equations needs to be approximated numerically. After discretizing the time derivatives in a suitable way, this is achieved by calculating the force acting on every single particle as a result of their pairwise interactions. Naively this requires the computation of $\mathcal{O}(N^2)$ interactions during each time step, where $N$ is the number of particles. However, the potential fields responsible for these interactions can be of different range and thus, dependent on their characteristics, only a limited number of particles influence each other. To permit the simulation of large systems, the use of data structures which enable a fast and memory efficient computation of particle-particle interactions with different range is indispensable. The implementation of these methods imposes significant challenges on modern architectures. Each molecular dynamics implementation needs to be optimized with respect to the present force fields and the compute environment the implementation is supposed to be executed in. Both C/C++ and Fortran, the predominant programming languages in high performance computing, are based on an inherently static and sequential model of computation. To deal with the increasing architectural complexity of modern architectures, these languages have been augmented with various extensions, most notably in form of parallel programming libraries like OpenMP, MPI or CUDA. Although they allow targeting new architectures, their power and expressiveness is limited by these libraries and they are by no means extensible. In order to overcome these limitations, new programming models and tools which make software both reusable and extensible, while also meeting the performance requirements of large scale HPC systems, are necessary.

In molecular dynamics typically the majority of the compute time is spent on the evaluation of pairwise non-bonded interactions[1] associated with intermolecular forces and electrostatic charge [2]. Intermolecular forces, including Van der Waals forces and dipole-dipole interactions, are even present in case of the absence of charged particles and thus need to be considered in most molecular dynamics simulations. The underlying potentials are often modeled as pairwise interactions of limited range that can be efficiently computed using neighbor lists [3]. Porting the computation of pair-interactions to modern multi-core CPU and GPU architectures has been subject to a number of publications [4]–[6]. Widely used frameworks such as GROMACS [7], LAMMPS [8], NAMD [9], AMBER [10], ESPResSo [11], CHARMM [12] and DL_POLY [13] are implemented in C/C++ or Fortran and employ distinct hand-

[1]Non-bonded interactions are present between atoms that are not linked by covalent bounds.

optimized pair interaction kernels for the CPU or GPU to achieve optimal performance on different architectures. In contrast, the goal of this work is to develop a unified scheme for the computation of pairwise interactions that can be specialized at compile time to generate optimized kernels for modern multi-core CPUs but also accelerators like GPUs. To perform this specialization, we make use of partial evaluation, which means that the original program is evaluated with respect to a static fraction of its input, to generate a new residual program whose input does no longer depend on this static fraction. In our case this static fraction corresponds to an execution model for the given platform. We express this model in form of a higher-order function[2], to which the kernel that performs the actual computation can be passed in form of another function. Partial evaluation of this higher-order function with respect to a certain kernel generates a residual program that can be run on each platform that supports the corresponding execution model. For instance, to execute a pair interaction kernel on the CPU a certain higher-order function can be employed, while a different one is used to generate code for the GPU. Essentially, the implementation of a platform-specific execution model constitutes writing an interpreter for an embedded domain-specific language (DSL) [14], where the DSL is defined as the set of valid computation kernels that can be executed on the corresponding platform.

*AnyDSL* is a framework that aims to enable the fast and easy development of embedded domain specific libraries. It grants the ability to create functional abstractions over multiple levels of hierarchies while the overhead is completely removed by utilizing partial evaluation. As frontend the programming language Impala together with an integrated partial evaluator [15] is offered. *Impala* is an imperative and functional programming language whose syntax is inspired by Rust [16]. AnyDSL has its own intermediate representation *Thorin*, that makes use of continuation-passing style (CPS) with first-class support for higher-order functions [17]. In general Impala code is first translated to Thorin, where a number of transformations that remove the overhead typically associated with the use of higher-order function take place. By exploiting the correspondence between CPS and the static single assignment (SSA) form [18], Thorin can be converted to SSA. SSA finds widespread use as a low-level intermediate representation, which facilitates the integration of Thorin into many existing compilers. By default, Thorin uses LLVM as a backend. So far AnyDSL has been used to implement a number of applications, where it has been shown that the generated code is able to achieve similar performance to hand-optimized code written in low-level programming languages or assembly. In [19] a platform-specific optimization of stencil codes is represented. [20] extends this approach to the generation of multigrid solvers while [21] targets the domain of ray tracing. AnyDSL and its frontend-language Impala are therefore suitable for the implementation of our approach for the generation of platform-specific pair interaction kernels from a unified base.

[2]A higher-order function is a function that takes one or more functions as arguments

## II. Background

The goal of this section is to provide the prerequisites that are required to understand the rest of the paper. First of all, we provide a summary of the code generation capabilities supported in AnyDSL's runtime. A more detailed description can be found on the AnyDSL website[3]. Note that the AnyDSL framework is ongoing research and Impala's syntax and the presented interfaces will probably be subject to change in the future. The section is concluded with a brief description of the neighbor list algorithm upon which our implementation is based.

### A. Device Code Generation

AnyDSL's runtime provides the required functionality to execute code on different compute devices:

- allocation, release and copying;
- code generation and execution;
- intrinsics.

AnyDSL currently supports the following platforms, which are detected when building the runtime:

- Host CPU (default, always present)
- CUDA
- OpenCL
- HSA

To generate code for a platform, the user can choose between one or multiple backends. For instance, the CUDA platform can be targeted with either CUDA or NVVM code. Each platform that is provided will have devices associated at runtime, which are enumerated starting with zero. For example, a properly configured system with a NVIDIA GPU will result in the following configuration:

- Platform 0: Host CPU
- Platform 1: CUDA
  Device 0: GPU
- Platform 2: OpenCL
  Device 0: GPU
- Platform 3: HSA
  dummy platform, no device

Consequently, this configuration enables the generation of code for the CPU (platform 0) and GPU with either CUDA (platform 1) or OpenCL (platform 2).

In Impala the allocation of memory on different platforms and devices is managed with the Buffer structure (see Listing 1), which tracks the platform and device on which the memory is allocated in device. To allocate memory on different

```
struct Buffer {
  data : &[i8],
  size : i64,
  device : i32
}
```

Listing 1: Buffer structure

[3]https://anydsl.github.io

devices, a number of convenience functions are available that automatically insert the correct platform into the returned Buffer (Listing 2). After the required memory has been

```
fn alloc_cpu(size: i32) -> Buffer;
fn alloc_cuda(dev: i32, size: i32) -> Buffer;
fn alloc_opencl(dev: i32, size: i32) -> Buffer;
fn alloc_hsa(dev: i32, size: i32) -> Buffer;

fn release(buf: Buffer) -> ();

fn copy(src: Buffer, dst: Buffer) -> ();
fn copy_offset(src: Buffer, off_src: i32,
               dst: Buffer, off_dst: i32,
               size: i32) -> ();
```

Listing 2: Memory management functions

allocated and the relevant data has been transfered to the respective devices, code generation for a specific platform can be performed by calling the respective function for each backend with the following syntax: backend(device, grid, block, fun)

- device: The device of the corresponding platform
- grid and block: Partitioning of the problem into subdomains
- fun: Function (or kernel) for which code will be generated

A typical example is shown in Listing 3. To abstract over

```
let grid   = (1024, 1024, 1);
let block  = (32, 1, 1);
let device = 0;
cuda(device, grid, block, || {
  ... out(idx) = in(idx);
});
synchronize_cuda(device);
```

Listing 3: CUDA code generation example

different platforms and devices, the accelerator structure can be employed, which is illustrated in Listing 4.

```
let device = 0;
let acc    = cuda_accelerator(device);
let grid   = (1024, 1, 1);
let block  = (32, 1, 1);
for tid, ..., gid in acc.exec(grid, block) {
  let (gidx, _, _) = gid;
  out(gidx()) = in(gidx());
}
acc.sync();
```

Listing 4: Example usage of the accelerator struct

### B. The Neighbor List Algorithm

In principle there exist $N^2$ mutual interactions in a system of $N$ particles and doubling their number results in a fourfold increase of the number of operations. To reduce the overall complexity, the range of pairwise interactions is usually restricted to a certain radius, which can then be exploited through the use of custom data structures that facilitate the detection of all relevant interactions. For this purpose the neighbor (or verlet) list [3] and the linked cell algorithm [22] are commonly employed. The neighbor list algorithm keeps track of the interactions of every single particle through repeated construction of a list of all particles within range. By including a certain buffer to this range, an update of this list can be postponed for a certain number of time steps, but still requires the evaluation of all pairwise interactions once. The linked cell method subdivides the simulated domain into cells with a side length greater than or equal to the interaction range. Therewith, only particles in the same and neighboring cells need to be considered. Typically both approaches are combined and the neighbor list is constructed from a cell-based decomposition to reduce its overall cost [23]. In [24] an adaption of this algorithm to the requirements of modern SIMD and SIMT architectures is described. This approach is implemented in the GROMACS framework since version 4.6 [7] and it forms the foundation for the implementation presented in this work. In contrast to the classical neighbor list scheme, this algorithm uses clusters of particles instead of individual ones as its building block. Hence interactions are not computed between individual particles but between clusters of nearby particles. By choosing the size of a cluster according to the SIMD width of a CPU or the number of threads per block on a GPU, an implementation can be optimized to the performance characteristics of these architectures. To efficiently organize all particles into clusters a bucket sort algorithm is employed. First of all, the domain is partitioned into a two dimensional grid of cells. Particles are inserted into the cells according to their location within the two-dimensional grid. The particles in each cell are then sorted with respect to the remaining dimension using a conventional sorting algorithm like insertion sort. Assume the size of cluster is $n$, then the first $n$ particles form the first cluster, the next $n$ particles the second cluster and so on. Because the number of particles in each cell is not necessarily a multiple of the cluster size, this implies the introduction of additional dummy particles, which must later be handled specially in the interaction computation. Finally, each cluster constructs a list of all clusters that are potentially in range for an interaction, again with an additional threshold to avoid the necessity for repeated neighbor list updates.

## III. Implementation

In the following we describe our implementation in AnyDSL's frontend language Impala and demonstrate how we are able to generate code for three different backends.

### A. Data Structures

The most fundamental decision that remains is the choice of data structures. Ensuring a high degree of data locality is key to optimize the performance on modern architectures. Additionally, the data must be organized in a way that allows dependency-free parallel computation on consecutive data elements to exploit data level parallelism on modern SIMD and SIMT architectures. Listing 6 shows a generalized pair interaction kernel in Impala which computes the force a particle with position pos2 causes to a particle with position pos1. First of all, the squared

```
struct Vec3D {
  x: f64,
  y: f64,
  z: f64
}
```

Listing 5: Basic data structure

```
fn compute_pairwise_force(pos1: Vec3D, pos2: Vec3D,
  squared_cutoff_radius: f64) -> Vec3D {
  let dx = pos2.x - pos1.x;
  let dy = pos2.y - pos1.y;
  let dz = pos2.z - pos1.z;
  let squared_distance = dx*dx + dy*dy + dz*dz;
  if squared_distance < squared_cutoff_radius {
    let f = potential(squared_distance);
    Vec3D{x: f*dx, y: f*dy, z: f*dz}
  }
  else {
    Vec3D{x: 0.0, y: 0.0, z: 0.0}
  }
}
```

Listing 6: General pair interaction kernel

distance between both particles is computed and compared with the squared cutoff radius, i.e. the range of the potential. If the particles are in range, the force is computed according the present potential, which in turn is used to update the total force of one or both particles. In case Newton's third law applies, in theory, only half of the interactions need to be computed and both particles can be updated. On the downside, in a parallel setting updating the same particle on different threads induces race conditions that require the use of atomic data updates and significantly reduce the overall performance. Apart from that, the computation of the interaction itself does not induce any data dependencies and can be computed for all particle pairs in parallel. In the best case, the data of all interacting particles is accessed in a consecutive way, which allows the exploitation of data parallelism on multiple particles and ensures spatial data locality. To enable consecutive accesses on both the positions and forces of a particle, the data of all particles can be stored in a single global structure of arrays (SoA) containing individual arrays for the masses, positions and velocities of all particles, as shown in Listing 7. Therewith, there is a high probability that subtracting the positions of two interacting particles corresponds to accessing consecutive elements within the same array. The advantage of organizing particles into clusters is that this can ensured by construction. When the interaction between two clusters is computed, the data elements of the individual particles are guaranteed to be located at consecutive memory addresses. The remaining challenge is to organize the data in a way that allows us to perform all computation on clusters of consecutive particles. As it has been mentioned in the last section, this can be achieved through a bucket sort algorithm. In x and y direction the simulation domain is partitioned into equidistant grid cells. Each cell contains all particles located within the respective subdomain. To efficiently transfer data between the cells and the Particles structure, we again store all relevant data in a structure of arrays. Because particles must be redistributed prior to the sorting, it is necessary to manage dynamically growing arrays within each cell. Though in our experience, in most cases it is sufficient to initially allocate enough memory, such that reallocations are only rarely necessary. After the clusters have been identified, the bounding box is computed for each cluster, which in turn is used within the distance computation during the neighbor list creation. To assemble the neighbor list of a cluster, only clusters in the same and neighboring cells must be considered by computing the distance between the respective bounding boxes. The number of unnecessary computations within the interaction kernel can be reduced by computing the pairwise distance between particles of clusters whose bounding box distance is close to the cutoff radius. Finally, after all particles are organized in clusters and the neighbor list has been assembled for each cluster, the global arrays can be constructed. In order to maintain the obtained knowledge about each individual cluster and its neighbors, three additional integer arrays are required (see Listing 7):

- the global indices of all neighbors (neighborlists),
- the number of neighbors per cluster (neighbors_per_cluster),
- the offset of the first neighbor within the global arrays containing the actual particle data (neighborlist_offsets).

Additionally, we want to omit all dummy particles within the computation, which can be achieved by storing a bit value for each individual particle that can later be used for masking out individual particles. The resulting global data structure is shown in Listing 7.

```
struct Particles {
  number_of_particles: i32,
  masses: &[f64],
  positions: &[f64],
  velocities: &[Vec3D],
  forces: &[Vec3D],
  number_of_clusters: i32,
  neighborlists: &[i32],
  neighborlist_offsets: &[i32],
  neighbors_per_cluster: &[i32],
  mask: &[bool]
}
```

Listing 7: Global particle arrays

## B. Code Generation and Execution

The next step is to define a unified base in Impala from which code for different platforms can be generated and executed. Here we first describe our generalized code for the computation of all pairwise interactions between a system of particles. Assume there is a higher-order function execute that allows us to generate and execute arbitrary code on different platforms and to perform an update on all particles in parallel. All details about the execution model of each platform are hidden within the implementation of this function. As we perform all computations on clusters of particles, the following information is required to perform an update on a certain particle with respect to the present interactions: The particle's global index pi within the arrays containing all particle data, the index of its cluster ci and the size of a cluster, i.e. the number of particles per cluster. Listing 8 shows the resulting interface for the execute function. With the availability of a general

```
fn execute(particles: Particles,
  fun: fn(i32, i32, i32) -> ()) -> ();

// example application
for pi, ci, cluster_size in execute(particles) {
  // update particle pi in cluster ci
  ...
}
```

Listing 8: Interface for executing arbitrary particle updates

code generation and execution function, the computation of all pairwise interactions can be implemented. This implementation is shown in Listing 9. Impala provides the for construct as syntactic sugar for calling a higher-order function with an anonymous function as its last argument. Here we call the execute function with an anonymous function that possesses the three arguments pi, ci, cluster_size and executes the body of the construct dependent on those arguments. We assume that functions for accessing the individual data elements within the global arrays described in Listing 7 are available. On which platform the data is located can be abstracted using AnyDSL's Buffer structure as described in the last section. If the data is allocated on a different device than the host CPU, it is necessary to transfer it before the interaction computation. To avoid unnecessary computations from the beginning, all dummy particles are masked out. Then the interactions of the particle pi with all particles in the same cluster are evaluated and its force is updated accordingly. Both the computation of a single interaction and the force update happen in the function update_force. Here we leave open if the force of only one particle is updated or of both. As it has been discussed in the last section, the latter requires the use of atomic operations. If the size of a cluster is statically known, the computation can be unrolled using the unroll function, which is provided by the AnyDSL runtime. Interactions with dummy particles are again masked out. Next the interactions with all clusters in the neighbor list are computed in a similar way. This can be achieved through an iteration over the respective part of the global array of all neighbors (neighborlists). For a simply range-based iteration similar to a for-loop in C, AnyDSL provides the range function.

The integration of the positions and velocities of all particles can be implemented in a similar way, but because the overall runtime of a molecular dynamics simulation is typically dominated by the interaction computation, we omit their description here. What now remains is the implementation of the execute function for different platforms. As it has been described in Section II, AnyDSL provides the Accelerator structure to generate and execute code on platforms other than the host CPU. Consequently, only two different implementations, one for the CPU and one for different accelerator hardware, either represented by a CUDA, OpenCL or HSA platform needs

```
fn compute_forces(particles: Particles,
    r_cut_sqr: real_t,
    potential: fn(real_t) -> real_t) -> () {
  for pi, ci, cluster_size in execute(particles) {
    if get_mask_value(pi, particles) {
      let begin = ci * cluster_size;
      for j in unroll(0, cluster_size) {
        let pj = begin + j;
        // Calculate interactions within cluster
        if pi != pj && get_mask_value(pj, particles) {
          update_force(particles, pi, pj,
          r_cut_sqr, potential);
        }
      }
      let number_of_neighbors =
      get_number_of_neighbors(ci, particles);
      let offset = get_offset(ci, particles);
      let nls = get_neighborlists(particles);
      // calculate interactions with all neighbors
      for cj in range(0, number_of_neighbors) {
        let begin_neighbor = nls(offset + cj);
        for j in unroll(0, cluster_size) {
          let pj = begin_neighbor + j;
          if get_mask_value(pj, particles) {
            update_force(particles, pi, pj,
            r_cut_sqr, potential);
          }
        }
      }
    }
  }
}
```

Listing 9: Interaction computation

to be provided. Listing 10 shows the implementation for the CPU. To make use of the multithreading capabilities of modern CPU hardware, AnyDSL provides the parallel function, that automatically parallelizes the execution of functions on a certain range, similar to the functionality provided by OpenMP. Each call to parallel automatically divides the provided range by the number of threads while each thread performs the execution of the function on the respective fraction[4]. Within each thread, the function body is then executed on all particles of the respective clusters. The size of a cluster is globally fixed and can be obtained with the function get_cluster_size. If the cluster size is statically known, we make use of the unroll function, to unroll the computation on the particles of a certain cluster. We need to mention here that AnyDSL provides the possibility to automatically vectorize code regions using RV, a unified region vectorizer [25], which could be used as an alternative to unrolling. But since so far we have not managed to generate an efficient vectorization with this approach, we only enforce unrolling within Impala, while relying on LLVM for the vectorization. As a final step, we describe the implementation of execute for the code generation and execution on accelerator hardware, which is shown in Listing 11. To abstract over different platforms that all target the same hardware, we

[4]Based on OpenMP, the same could be achieved in C or Fortran via annotation with the compiler directive #pragma omp parallel for schedule(static) while OMP_NUM_THREADS is set accordingly.

```
fn execute(particles: Particles,
           body: fn(i32, i32, i32) -> ()) -> () {
  for ci in parallel(get_number_of_threads(), 0,
                     particles.number_of_clusters) {
    let cluster_size = get_cluster_size();
    let begin = ci * cluster_size;
    for i in unroll(0, cluster_size) {
      let pi = begin + i;
      body(pi, ci, cluster_size);
    }
  }
}
```

Listing 10: Execution function on the CPU

make use of AnyDSL's Accelerator structure. Depending on which platform one wants to target, we provide a different implementation of the function get_accelerator. For example, to generate code for a CUDA platform with device device_id cuda_accelerator(device_id) is returned. Next a partitioning of the particle data into subproblems must be defined. Because our algorithm already provides a partitioning of particles into clusters, we simply create a one dimensional grid with a size equal to the number of particles, and then choose the block size to be equal to the size of one cluster. Consequently, the x component of the global grid index gidx corresponds to the particle index pi, and the x component of the block index bidx corresponds to the cluster index ci. With the implementation

```
fn execute(particles: Particles,
           body: fn(i32, i32, i32) -> ()) -> () {
  let acc = get_accelerator(device_id);
  let size = particles.number_of_clusters;
  let grid = (size * get_cluster_size(), 1, 1);
  let block = (get_cluster_size(), 1, 1);
  for bid, bdim, gid in acc.exec(grid, block) {
    let (gidx, _, _) = gid;
    let (bidx, _, _) = bid;
    let (bdimx, _, _) = bdim;
    body(gidx(), bidx(), bdimx());
  }
  acc.sync();
}
```

Listing 11: Execution function on accelerators

of an execute function that represents the execution model of the given hardware, we are able to generate and execute code on arbitrary platforms without adapting our kernel for the computation of pairwise interactions. The only adaption required is the choice of an appropriate cluster size. On the CPU it makes sense to choose a cluster size equal to the width of the SIMD units for vectorization. On the GPU the cluster size directly maps to the size of a block, and therefore should be chosen as a multiple of the warp size. In the next section we demonstrate how we are able to generate and execute code on different CPU and GPU hardware. As baseline for an evaluation of the performance of our implementation we compare it with miniMD, a molecular dynamics miniapplication from the Mantevo project [26], that uses the same computational core as the popular molecular dynamics package LAMMPS [8].

## IV. Experiments

The miniMD simulation package is based on an implementation of the classical verlet list scheme that works with individual particles instead of clusters. Therefore, to allow a fair comparison we set the size of a cluster within our implementation to one. Because updating both particles for every computed interaction requires the use of atomics, which impedes an efficient thread-based parallelization, we use a full neighbor computation scheme in both implementations, where only one particle is updated per interaction and all interactions are computed twice. The following settings are used within all test runs:

- Particles are placed on an equidistant grid and initialized with a small random velocity.
- Number of simulated time steps: 100
- Time step size: $dt = 0.001$
- Lennard-Jones Potential with $\varepsilon = 1$, $\sigma = 1$, $r_{cut} = 2.5$.
- Verlet buffer: 0.3
- Neighbor list update after 20 time steps.
- Particle sorting in each cell after 20 time steps.
- Full neighbor interaction scheme.
- All floating point operations are performed in double precision.
- AnyDSL: LLVM version 5.0.1
- 5 runs per benchmark. The average runtime is measured.
- Compiler flags: -O3, -march=native

### A. *Single-Core Performance*

To evaluate the single-core performance of our implementation compared to miniMD, we measure the performance on three recent Intel CPU generation in a simulation of 50 000 particles with LIKWID [27]. To ensure reliable measurements the frequency of each CPU is fixed. The number of floating point operations per cycle serves as the performance metric, which means that all floating point operations are counted, including those required for non-interacting particles within the neighbor lists. MiniMD is compiled using the Intel C compiler version 17.0.5, whereas the Impala code is first translated to LLVM IR using the recent version of Impala and Thorin and then compiled using Clang 5.0.1. Since our implementation so far does not produce efficient AVX code, we use two different vectorized versions of miniMD for the comparisons, once without and once with AVX instructions. The results are summarized in Table I. On all three architectures LLVM was only able to partially vectorize the generated IR code with SSE instructions. According to LIKWID, 25–27 % of the generated instructions are SSE instructions and no AVX instructions are generated at all. Still, on the recent Intel CPU generations Skylake and Broadwell we are able to achieve similar performance to miniMD without AVX. Although as the miniMD AVX version shows, the lack of generated AVX instructions limits the performance of our implementation on the CPU.

Table I: Single-Core Performance in FLOPS/cycle

| Processor | AnyDSL (SSE) | miniMD (SSE) | miniMD (AVX) |
|---|---|---|---|
| Skylake ESP | 0.73 | 0.76 | 0.94 |
| Broadwell EP | 0.79 | 0.81 | 1.00 |
| Ivy Bridge EP | 0.52 | 0.81 | 1.02 |

## B. Parallel Efficiency

Next we evaluate the parallel efficiency of our implementation in a strong and weak scaling setup on a Socket with four physical cores (Intel Xeon CPU E3-1275 v5 with 3.60 GHz). For strong scaling the particle number is fixed to 100 000 while the number of threads is increased. For weak scaling the number of particles per thread is set to 50 000. The resulting parallel efficiency is shown in Figure 1a and Figure 1b, respectively. In both settings our implementation achieves a parallel efficiency of 96 % on two cores, which is similar to miniMD, though its efficiency drops to about 80 % when four cores are used, which is slightly worse than that of miniMD.

## C. GPU Acceleration

Finally, we evaluate how much acceleration our implementation achieves on recent GPU hardware. As test platform an NVIDIA GTX 1080 is used, whereas the code is generated with AnyDSL's CUDA backend. For comparison we again employ an Intel Xeon Skylake with four cores and measure the runtime of both implementations (AnyDSL and miniMD with AVX) with different particle numbers using four threads. The generated CUDA code is then run with the same settings on the NVIDIA GTX 1080. Within our GPU implementation we set the size of a cluster to 32, which has consistently shown to lead to the best performance. The results of this comparison are shown in Table II. Even for relatively small particle numbers we are able to achieve a speedup of 3.2 or more on the GPU and are at least 2.7 times faster than miniMD with AVX, whereas the overhead for moving the particle data to the GPU is only about 1 % of the total runtime.

Table II: Average runtime on the CPU and GPU

| Particles | miniMD (AVX) | AnyDSL (SSE) | AnyDSL (GPU) |
|---|---|---|---|
| 100 000 | 1.464 s | 1.791 s | 0.546 s |
| 500 000 | 7.641 s | 9.347 s | 2.838 s |
| 1 000 000 | 14.81 s | 19.39 s | 4.923 s |
| 2 000 000 | 32.77 s | 39.06 s | 11.60 s |

## V. Conclusion and Future Work

Within this paper we have presented a general approach to generate code for parallel pair interaction computations on different platforms based on a unified base. This means we do not have to use different languages like a combination of C++ and CUDA in order to make use of the GPU. In contrast to code generation based on external DSLs, such as [28], that require the implementation of a dedicated code generator, our approach can in principle be implemented as embedded DSL in any programming language that enables the partial evaluation of higher-order functions, which we have demonstrated with the presented implementation in AnyDSL. However, currently AnyDSL is the only framework that supports the partial evaluation of higher-order functions.

From a performance perspective, although we already achieve similar performance to the miniMD simulation package without AVX, we perform no architecture-specific optimizations so far and we were not able to generate fully vectorized code based on our implementation. Thus we will next strive to achieve a complete vectorization based on the use of AnyDSL's integrated vectorization capabilities. Moreover, this work only presents a node-level parallelization and we plan to extend our approach to a cluster-level parallelization based on MPI in order to target multi-node systems.

## Acknowledgement

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the Metacca and ProThOS projects. Furthermore, the authors would like to thank Roland Leißa, Arsène Pérard-Gayot, and Klaas Boesche for helping to resolve problems with the AnyDSL framework.

## References

[1] P. Colella, "Defining software requirements for scientific computing", 2004.

[2] E. Lindahl, B. Hess, and D. Van Der Spoel, "GROMACS 3.0: A package for molecular simulation and trajectory analysis", *Journal of Molecular Modeling*, vol. 7, no. 8, pp. 306–317, 2001.

[3] L. Verlet, "Computer experiments on classical fluids. i. Thermodynamical properties of Lennard-Jones molecules", *Physical Review*, vol. 159, no. 1, p. 98, 1967.

[4] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units", *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[5] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers – short range forces", *Computer Physics Communications*, vol. 182, no. 4, pp. 898–911, 2011.

[6] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer, "Strong scaling of general-purpose molecular dynamics simulations on GPUs", *Computer Physics Communications*, vol. 192, pp. 97–107, 2015.

[7] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers", *SoftwareX*, vol. 1, pp. 19–25, 2015.

[8] S. Plimpton, P. Crozier, and A. Thompson, "LAMMPS - large-scale atomic/molecular massively parallel simulator", *Sandia National Laboratories*, vol. 18, 2007.

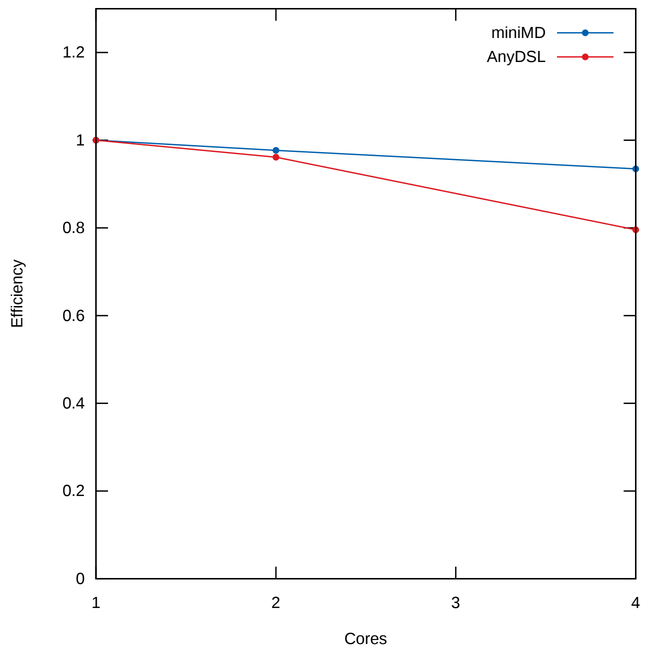
[9] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD", *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[10] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The amber biomolecular simulation programs", *Journal of computational chemistry*, vol. 26, no. 16, pp. 1668–1688, 2005.

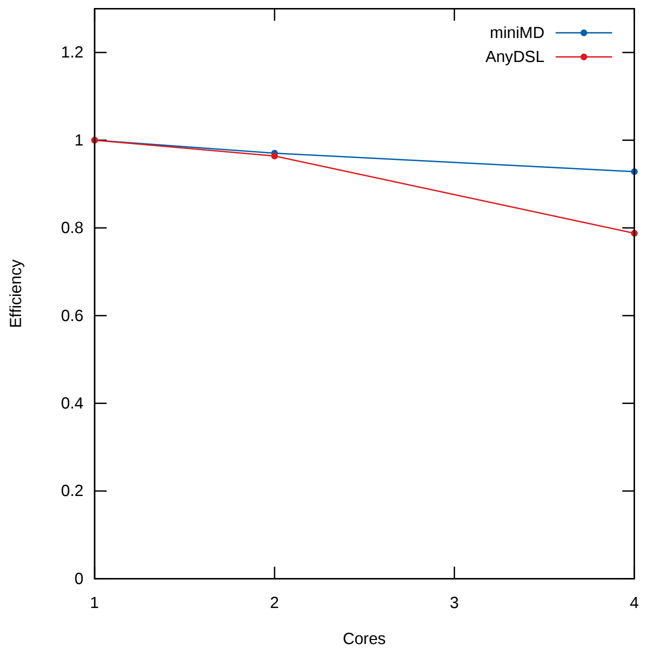
[11] H.-J. Limbach, A. Arnold, B. A. Mann, and C. Holm, "ESPResSo — an extensible simulation package for research on soft matter systems", *Computer Physics Communications*, vol. 174, no. 9, pp. 704–727, 2006.

[12] A.-P. Hynninen and M. F. Crowley, "New faster CHARMM molecular dynamics engine", *Journal of computational chemistry*, vol. 35, no. 5, pp. 406–413, 2014.

[13] I. T. Todorov, W. Smith, K. Trachenko, and M. T. Dove, "Dl_poly_3: New dimensions in molecular dynamics simulations via massive parallelism", *Journal of Materials Chemistry*, vol. 16, no. 20, pp. 1911–1918, 2006.



(a) Strong Scaling



(b) Weak Scaling

Figure 1: Parallel efficiency on an Intel Xeon Skylake with four cores

[14] J. Carette, O. Kiselyov, and C.-C. Shan, "Finally tagless, partially evaluated.", in *APLAS*, Springer, vol. 4807, 2007, pp. 222–238.

[15] R. Leißa, K. Boesche, S. Hack, R. Membarth, and P. Slusallek, "Shallow embedding of DSLs via online partial evaluation", in *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences*, ser. GPCE, ACM, 2015, pp. 11–20. DOI: 10.1145/2814204.2814208.

[16] N. D. Matsakis and F. S. Klock II, "The Rust language", in *ACM SIGAda Ada Letters*, ACM, vol. 34, 2014, pp. 103–104.

[17] R. Leißa, M. Köster, and S. Hack, "A graph-based higher-order intermediate representation", in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, IEEE, 2015, pp. 202–212.

[18] R. A. Kelsey, "A correspondence between continuation passing style and static single assignment form", in *ACM SIGPLAN Notices*, ACM, vol. 30, 1995, pp. 13–22.

[19] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek, "Platform-specific optimization and mapping of stencil codes through refinement", in *In Proceedings of the First International Workshop on High-Performance Stencil Computations (HiStencils)*, 2014, pp. 1–6.

[20] R. Membarth, P. Slusallek, M. Köster, R. Leißa, and S. Hack, "Target-specific refinement of multigrid codes", in *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, IEEE, 2014, pp. 52–57.

[21] A. Pérard-Gayot, M. Weier, R. Membarth, P. Slusallek, R. Leißa, and S. Hack, "RaTrace: Simple and efficient abstractions for BVH ray traversal algorithms", in *ACM SIGPLAN Notices*, ACM, vol. 52, 2017, pp. 157–168.

[22] B. Quentrec and C. Brot, "New method for searching for neighbors in molecular dynamics computations", *Journal of Computational Physics*, vol. 13, no. 3, pp. 430–432, 1973.

[23] P. Gonnet, "Pairwise Verlet lists: Combining cell lists and Verlet lists to improve memory locality and parallelism", *Journal of Computational Chemistry*, vol. 33, no. 1, pp. 76–81, 2012.

[24] S. Páll and B. Hess, "A flexible algorithm for calculating pair interactions on SIMD architectures", *Computer Physics Communications*, vol. 184, no. 12, pp. 2641–2650, 2013.

[25] S. Moll and S. Hack, "Partial control-flow linearization", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, to appear, ACM, 2018.

[26] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications", Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[27] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments", in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, IEEE, 2010, pp. 207–216.

[28] W. R. Saunders, J. Grant, and E. H. Müller, "A domain specific language for performance portable molecular dynamics algorithms", *Computer Physics Communications*, 2017.