TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR FOR COMPILER CONSTRUCTION

Faculty of Computer Science

Center for Advancing Electronics Dresden
Chair for Compiler Construction

Development of a Custom Compilation Workflow With MLIR Leveraging OpenFPM to Accelerate Particle Simulations

Matthias Cornel

Born on: 24th August 1992 in Würzburg
Matriculation number: 5032916

Master Thesis

to achieve the academic degree

Master of Science (M.Sc.)

First referee
Prof. Dr.-Ing. Jeronimo Castrillon

Second referee
Prof. Dr. sc. techn. Ivo F. Sbalzarini

Supervisor
Nesrine Khouzami

Submitted on: 18th August 2025

Statement of authorship

I hereby certify that I have authored this document entitled *Development of a Custom Compilation Workflow With MLIR Leveraging OpenFPM to Accelerate Particle Simulations* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Aschaffenburg, 18th August 2025

Matthias Cornel

Abstract

Particle simulations are a cornerstone of modern scientific computing. Due to the high computational demands of large-scale models, they are often run on heterogenous, distributed systems. This work presents the `particles` dialect, a domain-specific MLIR dialect for particles-only simulations, along with the conversion pipeline that translates it to standard MLIR dialects. The pipeline targets both CPUs and CUDA GPUs and integrates OpenFPM, a state-of-the-art C++ parallel computing library, using an automatically generated runtime to enable Distributed-Memory Parallelism (DMP). Update and communication operations required for DMP are inserted automatically using a placement strategy built on specialized Data-Flow Analysis (DFA) variants. To evaluate the quality of the generated code, an implementation of Molecular Dynamics (MD) based on the `particles` dialect was benchmarked against equivalent C++ implementations using OpenFPM. On CPUs, performance results are very close, suggesting we achieved high code quality for this target. On CUDA GPUs, the performance is significantly lower than that of the reference, which is expected given the early development stage of this backend. In addition to the `particles` dialect, the work introduces several multi-level IR design concepts and patterns, which can be adapted for use in other projects. A new general-purpose `box` dialect is also proposed, which is designed to enable 1-to-N conversion.

Acknowledgements

I would like to begin by thanking my supervisor, Nesrine Khouzami, for her support throughout the often challenging process of creating this thesis. Without the many insightful discussions and constructive criticism, this work would not have reached its current form.

I am grateful to Pietro Incardona for his help in answering my questions regarding OpenFPM.

I also wish to thank all members of the Chair for Compiler Construction at the TU Dresden for inspiring my interest in MLIR and compiler engineering, and for giving me the opportunity to contribute to this research field.

I sincerely thank my parents and my mother-in-law for their constant support. Without them, pursuing a Master's degree would not have been possible.

Finally, my deepest gratitude goes to my wife, Christin Cornel, who supported me throughout this thesis with great patience, especially during stressful times.

Contents

Acronyms

CFD Computational Fluid Dynamics
CFG Control-Flow Graph
CIM Compute In-Memory
CNM Compute Near-Memory
CPU Central Processing Unit
CSE Common Subexpression Elimination

DAG Directed Acyclic Graph
DCE Dead-Code Elimination
DFA Data-Flow Analysis
DMP Distributed-Memory Parallelism
DSL Domain-Specific Language

FFT Fast Fourier Transform
FPGA Field-Programmable Gate Array

GCC GNU Compiler Collection
GPU Graphics Processing Unit

HLS High-Level Synthesis
HPC High-Performance Computing

IDE Integrated Development Environment
IR Intermediate Representation

LTO Link-Time Optimization

MD Molecular Dynamics
MLIR Multi-Level Intermediate Representation
MPI Message Passing Interface

PPM Parallel Particle Mesh
PSE Problem Solving Environment

SIMT Single Instruction, Multiple Threads
SMT Simultaneous Multithreading

SPMD Single Program, Multiple Data
SSA Static Single-Assignment

TMP Template Meta-Programming

1 Introduction

Computer simulations have become an essential tool in scientific computing, enabling the study of complex models beyond what is possible when relying only on analytical and experimental methods. For this reason, numerical simulations are commonly regarded as the "third pillar of science" [1]. A prominent class of simulation algorithms is particle methods, with applications such as plasma physics [2], fluid simulations [3], and traffic simulations [4].

Particle methods are very versatile as they are capable of simulating discrete, continuous, deterministic, as well as stochastic models. They work by simulating the behavior of particles, which represent either discrete objects or discretization points of continuous fields. At each time step, particles interact with each other within a neighborhood and evolve independently [5]. To efficiently compute long-range interactions, particles may be combined with a uniform Cartesian background mesh [6].

The computational demands of large-scale models often exceed the capabilities of single machines. As a result, simulations are frequently executed on massively parallel or heterogeneous hardware, such as High-Performance Computing (HPC) clusters. However, developing software for HPC environments remains highly complex and time-consuming. Due to the complexity and a high entry barrier, the ability to implement programs for modern supercomputer platforms is limited to a small group of people. The reduction of this "knowledge gap" has become a key research focus in HPC [7]. General-purpose solutions include programming language extensions such as OpenMP [8] and CUDA [9], and software libraries like OpenMPI [10] and MPICH [11]. However, general-purpose solutions cannot be universal and concise at the same time. Solutions with fine abstractions remain hard to use, while those using coarse abstractions limit flexibility. By tailoring abstractions to specific domains, solutions can strike a reasonable balance between conciseness and flexibility. OpenFPM [12, 13] and OpenPME [14] are two such domain-specific solutions designed for parallel particles-only and hybrid particle-mesh simulations.

OpenFPM is an open-source C++ framework, developed as the successor of the discontinued Parallel Particle Mesh (PPM) library [15, 16]. It offers high-level abstractions for particle sets and meshes and provides support for domain decomposition, dynamic load balancing, ghost layer management, as well as cell list and Verlet list [17] implementations. In terms of execution hardware, it supports both multi-core CPUs and CUDA GPUs. By using Template Meta-Programming (TMP), OpenFPM manages to remain flexible while at the same time delivering high performance.

OpenPME is a Problem Solving Environment (PSE) [18] that builds on top of OpenFPM, offering both a Domain-Specific Language (DSL) and an Integrated Development Environment (IDE). The DSL is lowered via a series of model-to-model transformations to C++ code that uses OpenFPM. By hiding the distributed memory constructs and automatically placing update and communication operations, OpenPME further narrows the knowledge gap. Additionally, it detects common programming errors and provides easier-to-understand error messages than those typically produced by the template-engines of C++ compilers. As OpenFPM is the successor of the PPM library, OpenPME can be regarded as the successor of the PPM Environment (PPME) [19], which is likewise PSE for particle simulations but generates Fortran code targeting the PPM library instead.

In several cases, OpenPME can match or even exceed the performance of handwritten OpenFPM code. However, in certain scenarios, its performance falls significantly behind. This has been attributed to the generation of inefficient code and missing optimizations such as loop fusion. OpenPME is implemented in JetBrains MPS [20], which primarily aims at bridging the semantic gap between the domain expressed by the DSL and the underlying implementation. Fine-grained optimizations are performed by the target language compiler, which in this case is the C++ compiler. However, these compilers often fail to fully optimize the occasionally suboptimal code generated by OpenPME. While JetBrains MPS could support the implementation of certain optimizations, this work explores an alternative approach by leveraging MLIR to replace a portion of OpenPME's lowering infrastructure.

MLIR [21] is a multi-level IR compiler framework that offers a modular approach to building reusable and extensible compilers and compiler components. It aims to reduce software fragmentation, connect existing compilers, and improve compilation flows for heterogeneous systems. The framework provides an extensive infrastructure with common compiler features, such as parsing and printing, location tracking, symbol tables, and pass management. MLIR enables the construction of code generators, translators, and optimizers across various abstraction levels, application domains, hardware targets, and execution environments. Notably, in the context of computer simulations and HPC programming, it has been used to develop a dialect for stencil computations [22], which was later used to unify the code generation pipelines of two stencil-focused DSLs, Devito [23] and Psyclone [24] [25].

Motivated by the success of MLIR, in particular the `stencil` dialect, this work investigates the use of MLIR for particles-only simulations, aiming to replace parts of the OpenPME compilation pipeline with MLIR to improve the quality of the generated code, enable and facilitate optimizations, and target heterogeneous systems. To this end, a new dialect named `particles` was developed, along with a sophisticated lowering pipeline that generates code for both CPUs and CUDA GPUs. To optimize performance, update and communication operations are placed using a specifically designed Data-Flow Analysis (DFA) variant called "staleness analysis", and loops are fused aggressively. The solution is tightly integrated with OpenFPM to leverage its domain decomposition and dynamic load balancing features, as well as its ghost layer management and neighbor list implementations.

To evaluate the quality of the code generation, Molecular Dynamics (MD) was implemented using the `particles` dialect and compared to equivalent C++ implementations leveraging OpenFPM. Comparisons were performed for both CPUs and CUDA GPUs. On the CPU, the generated code did not reach the reference performance, but it came remarkably close, with performance gaps as narrow as 0.5%–2.5%. On CUDA GPUs, the performance was significantly lower than that of the reference and showed signs of rapid degradation throughout a simulation. Considering the early development stage of this backend, however,

the results are nevertheless promising.

The remainder of this work is structured as follows: Chapter 2 discusses the relevant background and related work. Chapter 3 introduces the motivation behind the project and provides an overview of the compilation pipeline, the developed dialects, and the transformation steps. Chapters 4 through 8 examine the MLIR transformation pipeline, divided into four phases. Chapter 9 evaluates the quality of the code generation by comparing the performance of the generated code against MD implementations done with OpenFPM. Finally, Chapter 10 concludes the work with a summary, discussion of the limitations, and suggestions for future research.

2 Background and Related Work

2.1 Particle Methods

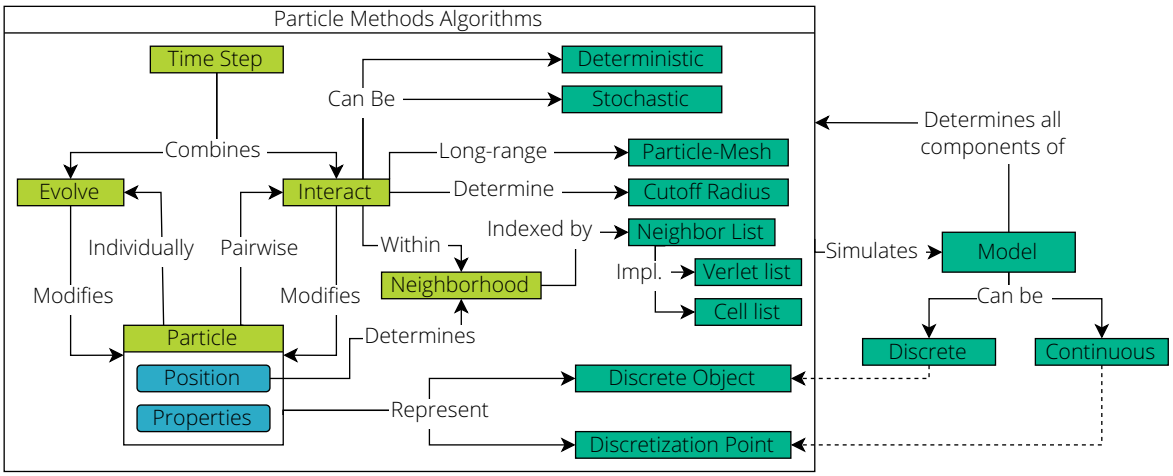

Figure 2.1: Particle methods concept map

Particle methods form a well-established class of algorithms for computer simulations [5]. Its basic concepts are visualized in Figure 2.1. All members of this class share five structural features. Each algorithm operates on entities called "particles", each of which has a position and a set of algorithm-specific properties. Computations are driven by two key functions: "interact" and "evolve". The interact function models pairwise interactions between particles within a "neighborhood". The evolve function updates each particle independently, operating only on its position and properties. Together, the evolve and interact functions are part of a "time step", which updates the state of the system as a whole.

The model simulated by a particle method algorithm determines all five components. Models may represent discrete or continuous phenomena, with interactions that are either stochastic or deterministic. In discrete models, particles correspond to the individual entities of the model, such as atoms in a molecular dynamics simulation [26], or cars in traffic simulations [4]. In continuous models, particles represent Lagrangian tracer or collocation points [27, 28].

In many simulation scenarios, the particle interactions are limited to a finite range or become negligible beyond a certain cutoff radius. Under these conditions, only the particles

separated by distances equal to or less than the cutoff radius are considered neighbors. For constant cutoff radii, two data structures are commonly used to identify the neighbors of a particle: cell lists [6] and Verlet lists [17]. Cell lists partition the simulation domain into (hyper)cubic cells with side lengths matching the cutoff radius, where each cell stores the indices of the particles located within its bounds. Each particle only interacts with the particles within the same or directly adjacent cells, trading the additional construction cost for lower interaction costs. Although more efficient than computing the interactions between all particles, this method includes significantly more neighbors than are actually within the cutoff sphere. Verlet lists address this by storing for each particle a list containing the indices of its neighbors. They are constructed using intermediate cell lists and commonly use a buffer region or "skin" to avoid reconstruction every time the particles move. Although Verlet lists incur less overhead during interactions, they come at increased memory and construction costs. Due to the importance of neighborhood data structures, they remain an active field of research, including work on adaptive-resolution neighbor lists [29].

In scenarios where the particle interactions are not limited to a finite range and do not become negligible beyond a cutoff radius, a mesh may be used to improve the efficiency of computing long-range interactions. For this purpose, the inter-particle forces are split into two components: Long-range forces and short-range forces. For each particle, the short-range component is calculated using particle-particle interactions within the cutoff radius specified by the short-range forces, while the long-range component is calculated using the mesh as an interpolation medium. This method is also called the Particle-Particle–Particle-Mesh Method (P^3M), as it combines the Particle-Particle (PP) and Particle-Mesh (PM) methods. [6]

2.2 OpenFPM

The Open Framework for Particles and Meshes (OpenFPM) [12, 13] is an open source C++ library designed for both particles-only and hybrid particle-mesh simulations on shared-memory and distributed-memory parallel architectures. It uses C++ TMP to define flexible, template data structures that allow simulations in arbitrary dimensions with particles carrying customizable properties. Through the use of TMP it also enables the configuration of the memory layouts for many of these data structures, allowing them to be fine-tuned for different requirements. To support DMP, OpenFPM provides custom memory allocators and a dynamic load balancer that evenly distributes the particles across processes.

OpenFPM defines two data abstractions for n-dimensional computational domains: particle sets and meshes. A computational domain is the n-dimensional hyperbox that defines the simulation space. To enable parallelism, the computational domain is partitioned into smaller subdomains, each of which is assigned to a process. Each process performs computations only on the particles and mesh nodes within its own subdomain. The procedure of partitioning the computational domain is called "domain decomposition". To ensure an even parallelization of data and work, the particle positions must be taken into account during this domain decomposition.

As particles move across inter-process boundaries, an initially optimized decomposition gradually becomes less optimal. This results in an unbalanced distribution and increased communication overhead. To address this, the domain decomposition must be updated regularly. When particles move between subdomains, they must be transferred between

the corresponding processes. The procedure that performs this transfer also transparently updates the domain decomposition as needed.

At the edges of a subdomain, the particles and mesh nodes must interact with those of its neighboring subdomains. To keep computations local, each subdomain is extended by a ghost layer. A ghost layer is a thin region around a subdomain, with a width determined by the interaction cutoff radius. These ghost layers are populated before performing any interactions, every time the domain decomposition has changed.

In terms of neighbor lists, OpenFPM provides implementations for both cell lists and Verlet lists. For cell lists, it supports several memory layouts that come with different memory cost and access time trade-offs.

OpenFPM supports both multi-core CPUs and CUDA GPUs. Through the many abstractions it provides, the differences between these hardware targets are largely hidden from the programmer. A key component of these abstractions are its iterators. Iterators are small data structures used to iterate over the particles or mesh nodes within a subdomain or neighborhood. They hide the non-trivial memory layouts while also enabling cache friendly visiting patterns.

The following OpenFPM classes are referenced in this work:

- `vector_dist` A distributed vector used to store the particles. Stores both the positions and the properties of all particles as a struct of arrays (SoA).
- `CellList_gen` A cell list implementation for the CPU. Its memory layout can be configured using a template parameter. In this work, only the `Mem_fast` configuration is used, which utilizes a two-dimensional array to store the particle indices for each cell.
- `CellList_gpu` A host-side cell list implementation for CUDA GPUs.
- `CellList_gpu_ker` The device-side counterpart for the `CellList_gpu` class.
- `NN_gpu_it_box` A device-side iterator for iterating over the indices of a particle's neighbors stored by a `CellList_gpu_ker`.

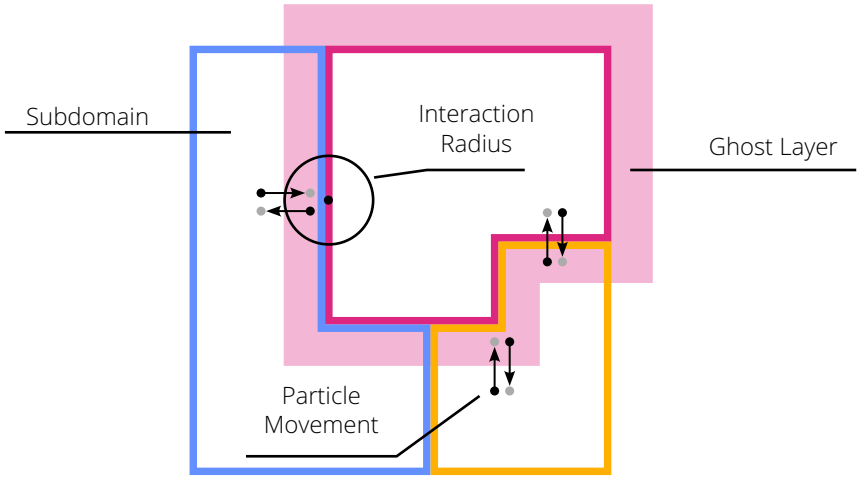

Figure 2.2: The effects of particle movements on the freshness of the domain decomposition, ghost layers, and neighbor lists

Figure 2.2 illustrates how – by moving in and out of ghost layers and interaction radii, and moving between subdomains – particle movements cause the domain decomposition, ghost layers, and neighbor lists to become stale. Before these data structures can be used again, they must be updated by the following functions:

- `vector_dist.map` Transfers particles that moved across process boundaries and updates the domain decomposition.
- `vector_dist.ghost_get` Populates each process's ghost layer. The fields (position and properties) to retrieve are specified via a combination of template parameters and function arguments.
- `vector_dist.update_cell_list` Updates the cell list instance passed as function argument.

These functions must be placed in the program at the correct position and because the respective data structures depend on each other, in the correct order. Furthermore, to minimize communication and computation overhead, they should only be placed where necessary.

2.3 OpenPME

The Open Particle-Mesh Environment (OpenPME) [14] builds on top of OpenFPM and offers both a DSL and an IDE. The DSL reduces implementation efforts and lowers the entry barrier, making the framework accessible to more potential users than OpenFPM. It hides many complexities of the OpenFPM library and automatically injects the calls to update and communication functions, freeing the programmer of the burden to do so correctly. Additionally, it improves the development experience by providing easier to understand error messages than those produced by the template-engines of C++ compilers. The OpenPME DSL supports both, imperative and declarative programming. In declarative programming, control flow structures, such as loops and conditional statements, are largely hidden from the programmer. The DSL also enables equations to be expressed using mathematical notations.

OpenPME uses JetBrains MPS [20] to define two metamodels. The first model captures particle-mesh simulations and supports particles-only, mesh-only, and hybrid particle-mesh simulations of both continuous and discrete models. The second metamodel represents the C++ code that is compiled against OpenFPM. It is generated from the particle-mesh model through a series of model-to-model transformations. These transformations enable expression rewriting, the insertion of update and communication operations, and the detection of programming mistakes. The refined and optimized C++ metamodel is used to generate C++ code.

2.4 MLIR and xDSL

MLIR [21] is a multi-level IR compiler framework that offers a modular approach to building reusable and extensible compilers and compiler components. Its goals include mitigating software fragmentation, connecting existing compilers, and improving compilation flows for heterogeneous systems. It provides an extensive infrastructure with common compiler features, such as parsing and printing, location tracking, symbol tables, and pass management. This section provides an overview of the MLIR IR and pass infrastructure. Refer to the MLIR language reference [30] for further details.

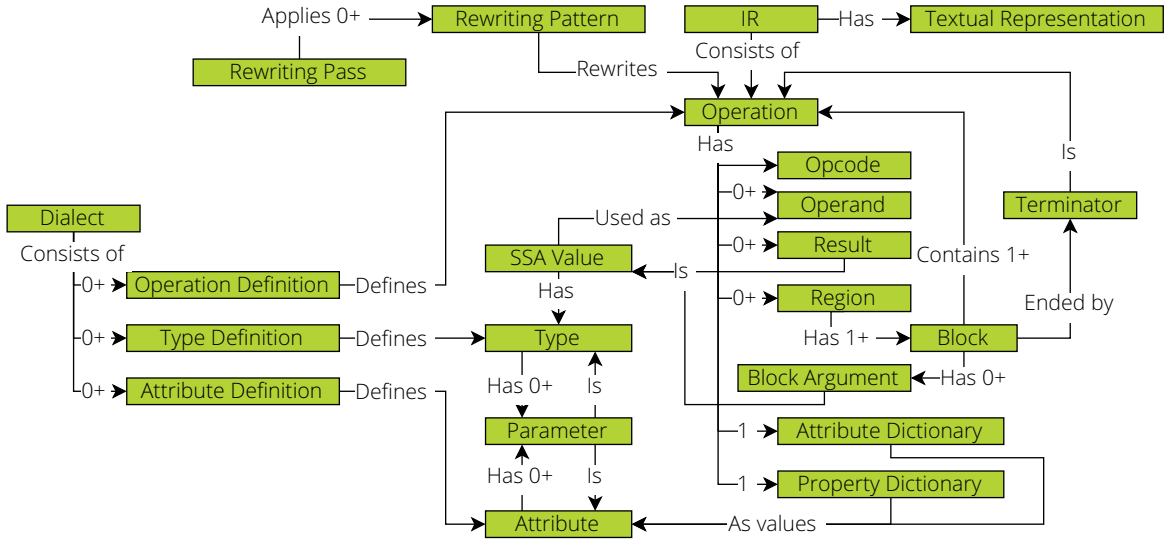

Figure 2.3: SSA compiler concepts shared between MLIR and xDSL

```
%res1, %res2 = "op.code" (%op1, %op2)                  // Results, opcode, operands
  <{"prop1" = 1 : index, "prop2" = 2 : index}>         // Property dictionary
(                                                      // Start regions
{                                                      // Start of region 1
^bb0(%arg1 : i64, %arg2: f32):                         // Block ID, block args with types
  // In this work, omitted code segments are denoted by >>
  >> Compute %ret1 and %ret2 from %arg1 and %arg2      // Block contents
  "op.yield"(%ret1, %ret2) : (i64, f32) -> ()          // Terminator operation
}                                                      // End of region 1
)                                                      // End of regions
  {"attr1" = true, "attr2" = false}                    // Attribute dictionary
  : (i64, f32) -> (i64, f32)                           // Operand types, result types
```

Listing 2.1: MLIR's generic operation format

Operations Figure 2.3 illustrates the main concepts of MLIR. At its core lies the IR, which is composed of one or more, potentially nested, operations. Operations are the semantic unit in MLIR, everything from "instruction" to "function" to "module" is modeled as an operation. Listing 2.1 exemplifies the generic operation format. Every operation can be expressed using this generic format, even those that define a custom format (e.g., Listing 2.2 and Listing 2.3). Each operation has an opcode, takes zero or more values as operands and produces zero or more values as results. Values follow the Static Single-Assignment (SSA) form [31], represent data at runtime, and have static types.

The meaning of an operation's inputs and outputs is completely specific to the operation itself. For instance, the `arith.addi` operation takes two integer values as operands and returns the sum of the two. Despite using the same syntax, the `arith.subi` operation performs subtraction, giving its operands and results entirely different meanings.

Every operation has a property dictionary and an attribute dictionary, both of which consist of key-value pairs, with strings as names and attributes as values. They may be empty, in which case they are omitted from the generic operation format. Properties and attributes convey compile-time information about operations. Properties are always intrinsic to an operation, whereas attributes may be used for storing extrinsic metadata. Typical use cases for extrinsic information include marking operations that have been rewritten by a pass or storing information for later passes.

```
%one = arith.constant 1 : index
%two = arith.constant 2 : index
%cond = arith.constant true
// Custom format:
%res = scf.if %cond -> (index) {
  // True branch
  scf.yield %one : index
} else {
  // False branch
  scf.yield %two : index
}
// %res has value of 1

// Generic format:
%res = "scf.if" (%cond)({
  // True branch
  "scf.yield" %one : (index) -> ()
},{
  // False branch
  "scf.yield" %two : (index) -> ()
}) : (i1) -> index
```

Listing 2.2: `scf.if` example

```
// Custom format:
%res = scf.for (%step) = (%zero) to (%hundred) step (%one)
               iter_args(%arg0 = %zero) -> (index) {
  %ret0 = arith.addi %arg0, %one : index
  // Transfers control and data to next iteration if number of steps not reached
  // Otherwise transfers control back to scf.for:
  scf.yield %ret0 : index
}
// %res has value of %hundred

// Generic format:
//               from , to      , step, inititalize iter_args
%res = "scf.for"(%zero, %hundred, %one, %zero) ({
//   induction var, iter_args
^bb0(%step : index, %arg0 : index):
  %ret0 = "arith.addi"(%arg0, %one) : (index, index)  ->index
 "scf.yield"(%ret0) : (index) -> ()
}) : (index, index, index, index) -> index
```

Listing 2.3: `scf.for` example

Regions and Blocks Operations may have zero or more regions, which provide the nesting mechanism in MLIR. The semantics of a region are determined by the operation it is attached to. For example, Listing 2.2 shows the `scf.if` operation, which has two regions: one for the true branch and one for the false branch. The `scf.for` operation, as seen in Listing 2.3, has one region that represents the loop body. A region comprises one or more blocks, where each block contains a list of operations. The operations, in turn, may have regions themselves, thereby enabling arbitrary nesting. The first block of a region is called entry block and is always executed first.

Every block ends with a special terminator operation, which transfers control to another block within the same region, called successor, or returns it to the operation enclosing the region. While doing so, it can also transfer SSA values, thereby linking data flow with control flow. Each `scf.yield` operation in Listing 2.2 transfers control back to its parent `scf.if` operation, passing along the results of its branch, which are subsequently returned

by the `scf.if` operation. Listing 2.3 shows an example of the `scf.for` operations, where the `scf.yield` operation transfers data and control back to its `scf.for` parent operation if the end condition is reached. Otherwise, it transfers them to the beginning of the same block. All blocks of a region, linked by the successor relationship, form a Control-Flow Graph (CFG).

Each block may have zero or more block arguments. Block arguments are regular SSA values available within the block. The semantics of the block arguments are, again, defined by the operation enclosing the block. The loop region of the `scf.for` operation shown in Listing 2.3 has a single block. The first argument of this block is always of type `index` and holds the loop induction variable. If a `scf.for` operation has "iter_args", which are arbitrary values passed alongside the control flow, the block has one additional argument for each iter_arg. In this work, all regions have a single block, the mention of which is often skipped. Furthermore, the block arguments of a block within a single-block region are referred to as region arguments.

Types and Attributes Each SSA value is associated with a type. Listing 2.4 shows a selection of `builtin` types. If a value is the result of an operation, its type is defined by that operation. If it is a block argument, the type is specified by the block. Types encode compile-time information about values and are used for strict type checking. They can be parametrized, with parameters being either attributes or other types. For example, the `vector` and `memref` types have parameters for specifying the shape and element type. Types can also have custom formats.

Attributes are very similar to types. They also have parameters that are either attributes or types, and can have custom formats. The key difference is that they encode compile-time information about operations instead of values.

```
// Integer types
i64, i32, i1, index
// Floating-point types
f64, f32
// Multi-dimensional SIMD vector type
vector<3xf64>
// Shaped reference to region of Memory
memref<3x?xf64>
```

Listing 2.4: Selection of builtin MLIR types

Dialects In MLIR, operations, types, and attributes are instances of definitions that provide the syntactical and structural rules to which each instance must adhere. An operation definition, for instance, defines the opcode, the number and types of operands and results, the number of regions, the blocks within each region, the block arguments of each block, the terminator operation of each block, the properties and intrinsic attributes, and potentially a custom format. Additionally, it defines the overarching rules that tie all components together, which are enforced via verification.

MLIR provides a set of built-in definitions that are always active, but does not constrain the set of definitions. Instead, it manages them via dialects. A dialect is a logical grouping of operation, type, and attribute definitions under a unique namespace (e.g., `builtin`, `scf`, `func`). Each dialect encapsulates a concise concept into a small, manageable package. This

approach of separating definitions is akin to separating program code into packages or libraries. While every IR construct (operation, type, or attribute) belongs to exactly one dialect, mixing dialects in the IR is explicitly supported. For example, in Listing 2.3, the `scf` and `arith` dialects, as well as the `index` type from the `builtin` dialect coexist. This composability increases reuse, extensibility and provides a high degree of flexibility. MLIR provides several standard dialects, such as the `scf` dialect which encapsulates structured control flow, and the `func` dialect, which encapsulates operations for declaring, defining, and calling functions. The `builtin` dialect is always active and contains many commonly used constructs, such as numeric types and attributes (e.g., `i64`, `f64`).

Pattern-Based Rewriting MLIR offers an extensible infrastructure for pattern-based rewriting. Rewrite patterns match and modify operations in the IR by replacing, removing, or transforming them as a whole or in parts. They can be applied at any level of nesting. Patterns targeting the `builtin.module` operation, which is always the top-level operation, can perform global transformations that affect more than one operation, such as loop fusion.

Patterns are grouped into transformation passes. Passes serve various purposes, ranging from canonicalization and optimization to conversion. MLIR includes numerous passes. Some implement general compiler transformations, such as Common Subexpression Elimination (CSE) and Dead-Code Elimination (DCE), while others are dialect-specific. Conversion passes, in this work also referred to as lowering passes, convert the operations, types, and attribute of one dialect into those of others. These passes are essential for lowering the IR from high-level dialects to low-level dialects.

The `llvm` Dialect A very important low-level dialect is the `llvm` dialect, which encapsulates the LLVM IR [32]. An IR that only contains `llvm` constructs, `builtin` types, and the `builtin.module` operation, can be translated to LLVM IR. As such, the `llvm` dialect often serves as the final target for lowering pipelines. MLIR provides a wide variety of conversion patterns for gradually lowering the integrated standard dialects to `llvm`. When developing a new dialect, it is therefore not necessary to lower it to `llvm` directly. Instead, it can be lowered to higher-level dialects that are easier to target, followed by the application of the required integrated lowering passes to finish the conversion.

xDSL MLIR is designed for the development of production-grade compilers and is implemented in C++. While this ensures high performance, it comes at the expense of software simplicity, making it less suitable for prototyping. xDSL [33] is a sidekick framework written in Python 3 that is specifically designed for prototyping and education. It interoperates with MLIR by building on the same compiler concepts, such as SSA-based IR, dialects, pattern-based rewriting, etc., and by using the same IR format. Broadly speaking, it can be viewed as a Python-native reimplementation of MLIR. Its interoperability allows xDSL to be interleaved with MLIR at any point in the compilation pipeline. All dialects and rewriting passes presented in this work have been implemented using xDSL.

2.5 Related Work: MLIR in Scientific Computing

To the best of our knowledge, there exist no other efforts to develop an MLIR dialect tailored to particles-only or hybrid particle-mesh simulations. Therefore, this section

broadens the scope to the application of MLIR in scientific computing in general, while excluding research that primarily focuses on machine learning, High-Level Synthesis (HLS), or heterogeneous computing. With these criteria, three principal areas of research emerge: stencil computations, tensor computations, and Fast Fourier Transforms (FFTs).

The most influential and most closely related work is the Open Earth Compiler, a DSL-compiler for weather and climate simulations [22]. At the front and center of this compiler lies the `stencil` dialect, which was designed to serve as a lowering target for various user-facing stencil-oriented DSLs. To support both AMD and Nvidia GPUs, the researchers also developed the `gpu` dialect to vendor-independently abstract the GPU execution model. Features of this `gpu` dialect, which meanwhile has been integrated into MLIR and xDSL, are used in this project.

Bisbas et al. ported the `stencil` dialect to xDSL, where they used it to unify the backend compiler stacks of two stencil-focused DSLs, Devito [23] and Psyclone [24] [25]. They chose the `stencil` dialect introduced by Gysi et al. [22], instead of an alternative dialect developed by Essadki et al. [34], due to its domain-, problem-, and hardware-agnostic design. To support DMP, the researchers also created the `dmp` dialect, which enables the exchange of rectangular subsections of data between processes. Furthermore, for lowering this dialect, the `mpi` dialect was implemented, integrating MPI's point-to-point and collective communication mechanisms into MLIR. To bring stencil-computation to Field-Programmable Gate Arrays (FPGAs), Rodriguez-Canal et al. developed the `hls` dialect, which mirrors the HLS features provided by the AMD Xilinx Vitis tooling in a vendor-neutral manner [35].

Rink et al. introduced CFDLang, a target-agnostic DSL tailored for tensor operations commonly found in Computational Fluid Dynamics (CFD) programs [36]. To complement this DSL, the TeIL imperative intermediate tensor language, designed for expressing typical tensor kernels, was introduced [37]. It serves as an intermediate representation within the CFDLang compiler stack. Later, the CFDLang compiler was reengineered to leverage MLIR as backend [38]. As part of this redesign, both CFDLang and TeIL were reimplemented as MLIR dialects.

Discrete Fourier transforms are a crucial component in scientific computing. He et al. introduced the `fft` dialect as part of their development of FFTc, a domain-specific language for FFTs [39]. With FFTc 2.0, the `fft` dialect was dropped. Instead, the standard `linalg` dialect was extended with FFT-specific operations to simplify the formulation of FFT algorithms [40].

3 Motivation and Overview

This chapter presents the underlying motivation for the presented work and provides an overview of the compilation pipeline, with a focus on the newly developed MLIR dialects and transformation steps. The first section identifies key issues and limitations of OpenPME's current lowering infrastructure, from which it derives a set of long-term objectives. Section 3.2 defines the set of short-term objectives that guided the development of the presented work. Next, Section 3.3 introduces the full compilation stack, the most important part of which is the MLIR lowering pipeline, which is outlined by Section 3.4.

3.1 Issues and Limitations of OpenPME and Long-Term Objectives

OpenPME [14] aims to provide a DSL that enables users to create high-performance particle simulations without requiring the expertise to write optimized C++ code that leverages OpenFPM. To achieve this objective, it is essential that the lowering infrastructure, which transforms the OpenPME DSL to C++, produces optimized code. Benchmarks have shown, however, that, in some cases, the performance of the generated code significantly lags the performance of handwritten OpenFPM code. In the mesh-only simulation Gray-Scott, for example, the bulk access of mesh properties resulted in independent mesh loops, which were not automatically merged. Furthermore, the logic for placing the communication and update operations is too simplistic, resulting in additional communication and computation. The resulting OpenPME implementation was on average 3.25 times slower than the handwritten OpenFPM implementation, demonstrating the importance of high-quality code generation and the role optimizations like loop fusion play in this process.

The generation of inefficient code and missed optimization opportunities are the first two issues future versions of the lowering infrastructure must address. The third issue is the limited number of hardware platforms that OpenPME currently supports. OpenFPM supports both CPUs and CUDA GPUs, while OpenPME currently only supports CPUs. Furthermore, in the future, new processing platforms such as FPGA, Compute Near-Memory (CNM) and Compute In-Memory (CIM) may also need support. The forth issue concerns the limitations imposed by JetBrains MPS, which is not integrated with other compilers and is not part of an advancing compiler infrastructure such as MLIR. As a consequence, OpenPME in its current form cannot benefit from the advancements modern compiler infrastructures have to offer.

To summarize, future developments on the OpenPME lowering infrastructure must improve on four key aspects:

1. Resolve the issue of generating inefficient code, which includes the too simplistic strategy employed for placing communication and update operations.
2. Enable and implement crucial optimizations such as loop fusion.
3. Add support for CUDA GPUs and pave the way for future processing platforms, such as FPGA, CIM, and CNM.
4. Integrate the infrastructure with a modern and evolving compiler infrastructure to benefit from its advancements.

To address all four aspects, we decided to replace parts of the lowering infrastructure with MLIR. Figure 3.1 depicts a conceptual overview of the re-engineered lowering infrastructure. At present, the goal is to develop a new dialect, named `openpme`, to integrate the OpenPME DSL into the MLIR ecosystem. The aspired `openpme` dialect should closely resemble the OpenPME DSL to facilitate seamless integration.

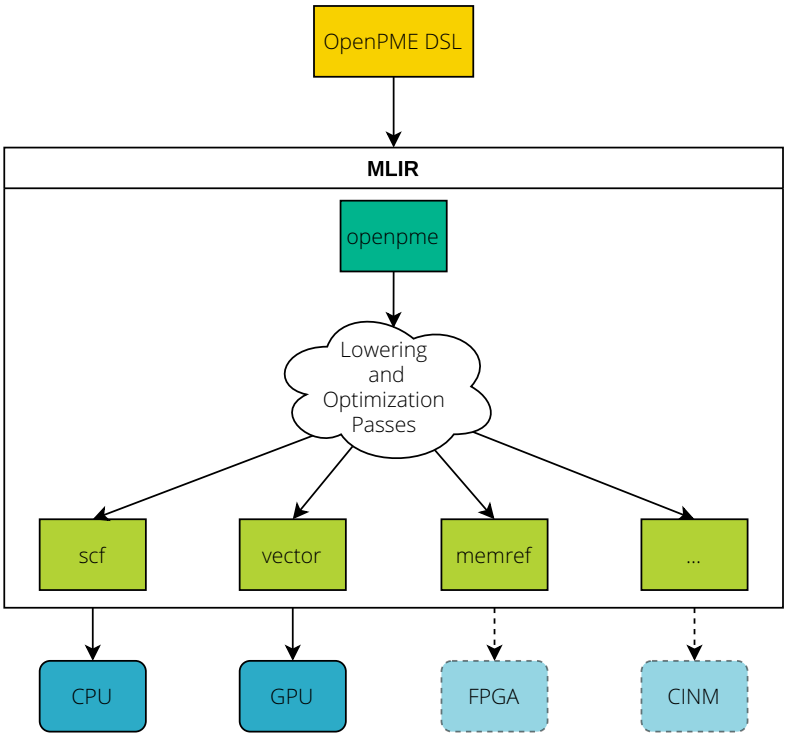

Figure 3.1: Updated OpenPME lowering infrastructure leveraging MLIR

3.2 Goals and Objectives of This Work

Although an `openpme` dialect is the long-term goal, it became evident early during development that the foundation for it is lacking. While implementing the dialect itself was feasible, lowering it to existing dialects posed too big of a task. This is due to the large conceptual distance between the aspired `openpme` dialect and the existing dialects it has to be lowered to. Consequently, the decision was made to develop the foundation from the bottom up by incrementally building on top of existing dialects toward a possible future `openpme` dialect. This build-up can happen in multiple ways:

1. **Extension**: Develop a cut-down version of the `openpme` dialect and incrementally change and extend it until it is transformed into the full `openpme` dialect.
2. **Lowering**: Develop a lower-level dialect that will later serve as part of the lowering pipeline for the `openpme` dialect.
3. **Supplementation**: Develop a dialect that captures a part of the capabilities of OpenPME and later supplement these capabilities using other dialects.

This list is not exhaustive, and at this stage, it is difficult to predict which development strategy will ultimately prove effective. The immediate priority is to simply begin constructing, while deferring the choice of strategy to the future.

The primary objective of this work is the development of `particles`, a specialized dialect for particles-only simulation, along with the necessary transformation passes to lower it to existing dialects. Early in development, it became apparent that addressing both particles and meshes, as OpenPME does, would significantly exceed the scope of this work. Given that the `stencil` dialect already encapsulates meshes, the decision was made to focus on particles.

Despite the already exclusive focus on particles, the initial scope was further restrained to cover only the fundamentals of particle simulations. Advanced capabilities, such as dynamically removing particles during or after interactions, were excluded to maintain feasibility. Thus, the initial iteration of the `particles` dialect will only cover the foundations of particle simulations, with advanced features reserved for future developments.

The remainder of this section examines the requirements for the `particles` dialect and its associated lowering pipeline, for which Table 3.1 provides an overview. While doing so, it is important to distinguish between dialect and pipeline requirements. Even though both are developed together and heavily influence each other during development, the lowering pipeline is interchangeable and should therefore be considered an independent component. All requirements are classified into three categories:

- **Functional** requirements define specific capabilities that the dialect and the lowering pipeline must support.
- **Non-functional** requirements do not introduce any new functionality; instead, they address critical aspects such as reusability and extensibility.
- **Semi-functional** requirements do not directly impact functionality, but are essential to the project's overall success. Their fulfillment is necessary for the project to be considered functional.

3.2.1 Dialect Requirements

(f1) Implement Base Capabilities for Particle Simulations While developing a fully featured dialect for particle simulations is not the goal, the `particles` dialect must provide a set of essential capabilities to qualify as a minimum viable product. The first iteration should support the following core functionalities:

- **Evolve**: Iterate over all particles in a particle set while reading and updating their positions and properties.
- **Interact**: Iterate over all neighboring pairs in a set, computing pairwise interactions, reducing the results of these interactions to the combined interaction results for each particle and its neighbors, and storing the reduced results as position or properties.

	`particles` Dialect	Lowering Pipeline
Functional	(f1) Implement base capabilities for particle simulations (f2) Support parametrization of particle & particle set types	(f3) Support DMP on CPUs & CUDA GPUs (f4) Automatically place update & communication operations (f5) Implement loop fusion
Non-Functional	(n1) Be general-purpose & target-agnostic	(n2) Facilitate extensibility (n3) Leverage OpenFPM
Semi-Functional	(s1) Preserve all parallelism (s2) Enable analysis & transformation	(s2) Enable integration with C++ (s3) Deliver adequate performance

Table 3.1: Requirements for the `particles` and the associated lowering pipeline

Both features imply several secondary requirements, including, but not limited to, the ability to load and store particle data and the implementation of neighbor lists, such as cell lists. These capabilities suffice to implement simple simulations such as MD.

(f2) Support Parametrization of Particle and Particle Set Types Particle simulations require varying particle properties and operate in different dimensions. As such, the `particles` dialect must enable parametrizing both the set of particle properties and the dimensionality. Initially, basic scalar and vector types should be supported as property types. Furthermore, the numeric type used for the position (e.g., `float32`, `float64`) should also be configurable, as is the case in OpenPME and OpenFPM.

(n1) Be General-Purpose and Target-Agnostic Two of MLIR's primary goals are to reduce software fragmentation and connect existing compilers [21]. The `particles` dialect should align with these objectives and, in the future, should allow compiler engineers to integrate it into their own designs. To achieve this, the dialect must be general-purpose and target-agnostic, remaining independent of OpenPME, OpenFPM, the processing hardware, as well as the execution model (e.g., DMP, Simultaneous Multithreading (SMT)). Therefore, the `particles` dialect should capture only the key concepts of the domain, shifting all target-specific aspects to the lowering pipeline, which can be interchanged. This approach allows developers to work with the dialect without being restricted to a specific framework or compilation target and even gives them the freedom to develop their own lowering pipeline if the existing one does not fit their requirements.

(s1) Preserve All Parallelism The `particles` dialect must preserve all possible parallelism. This requirement is closely related to (n1), as both aim to enhance the `particles` dialect's adaptability to different hardware and execution models. As a general-purpose and target-agnostic dialect, it must avoid making any assumptions about how and where parallelism is exploited. To achieve this, it is crucial to prevent the introduction of sequential constraints to the IR structure, such as data dependencies that limit parallel computation. As a result, decisions on parallelism exploitation are fully delegated to the lowering pipeline.

(s2) Enable Analysis and Transformation As outlined in Section 2.2, distributed particle simulations using OpenFPM depend on distributed data structures and communication, which necessitate the correct placement of update and communication operations. To ensure that these operations are placed only where necessary, the `particles` dialect must enable the required analysis for informed placement decisions. Additionally, its design should facilitate optimizations such as loop fusion.

3.2.2 Lowering Pipeline Requirements

(f3) Target DMP on CPUs and CUDA GPUs As outlined in dialect requirement (n1), the dialect should remain entirely target-agnostic, delegating all decisions regarding the processing hardware and execution model to the lowering pipeline. Consequently, the lowering pipeline must be capable of generating efficient code for multiple different targets. The initial implementation should prioritize the support for CPUs and CUDA GPUs, integrated with DMP to ensure full scalability in HPC cluster environments.

(f4) Automatically Place Update and Communication Operations Update and communication operations should be automatically inserted during lowering, relieving the programmer of the responsibility to place them correctly. In fact, because the `particles` dialect is target-agnostic, it does not even define any such operations. The placement should rely on careful analysis, as established by dialect requirement (s2), instead of simple rules to avoid unnecessary communicating and computation.

(f5) Implement Loop Fusion Section 3.1 highlights the importance of optimizations such as loop fusion. To avoid encountering the same issues as OpenPME, the `particles` dialect must support the implementation of key optimization passes. Evaluating whether the dialect really enables these optimizations, however, is challenging. Therefore, the lowering pipeline should implement loop fusion to empirically validate that the dialect facilitates essential optimizations.

(n2) Facilitate Extensibility While the initial implementation will only support CPUs and CUDA GPUs, it should establish a foundation for integrating additional hardware targets in the future. Similarly, while only cell lists will be supported in the first version, additional neighbor lists, such as Verlet lists, should be incorporable later. As a non-functional requirement, extensibility cannot be directly measured, but remains an important design consideration during development. To achieve extensibility, the focus should be on developing reusable dialects that streamline the lowering process while isolating target-specific components into separate dialects and passes. Furthermore, transformation passes targeting multiple hardware targets should be designed for extensibility.

(n3) Leverage OpenFPM OpenFPM provides optimized implementations of shared data structures and efficient communication mechanisms while also offering a high degree of flexibility. Instead of developing these complex features from scratch, the lowering pipeline should aim at incorporating OpenFPM to take full advantage of its existing capabilities. This approach is essential, as implementing the required functionality in MLIR would severely

exceed the scope of this work. In future versions, OpenFPM may be phased out if required or if a more suitable alternative emerges.

(s2) Enable Integration With C++ In line with requirement (n3), the lowering pipeline must support the integration of MLIR with C++ to utilize OpenFPM's capabilities. This presents a significant challenge, as MLIR currently does not natively support the use of C++ objects or classes, especially templated ones, which are extensively used by OpenFPM. Additionally, modern C++ standards, like C++17, enforce specific optimizations, such as copy elision [41], which prohibit passing instances of aggregate types by value between functions. This complicates the exchange of complex data between MLIR and C++ because in MLIR aggregate data is typically passed by value and the standard lowering pipeline does not replicate the C++ optimizations. To address this issue, an effective mechanism for communicating aggregate data structures between MLIR and C++ must be developed. The solution should be general-purpose and reusable to ensure broad applicability. Beyond leveraging OpenFPM, this also allows falling back to C++ where necessary. This is important since MLIR provides significantly less functionality than C++, as it lacks access to its extensive ecosystem of libraries. Limiting the programmers to using only MLIR would therefore severely limit the usability of the `particles` dialect.

(s3) Deliver Adequate Performance Achieving all other objectives becomes irrelevant if the fully compiled simulations fail to deliver adequate performance. The resulting particle simulations should achieve similar performance to equivalent C++ implementations leveraging OpenFPM.

3.3 Full Compilation Stack

Figure 3.2 illustrates the full compilation stack. At least two files must be implemented: a C++ file and an MLIR file. The C++ file is responsible for initializing all OpenFPM data structures, but may also contain additional functionality such as loading and storing checkpoints and providing terminal output. Meanwhile, the MLIR file encapsulates the simulation logic, including particle interactions, evolution steps, and time stepping procedures. While both files can have arbitrary names, they are referred to as `startup.cpp/.cu` and `simulation.mlir` for simplicity. When targeting CUDA GPUs, the C++ file uses the `.cu` extension, whereas the `.cpp` extension is used for targeting CPUs. Control is transferred between both files via function calls and simple storage structs that are passed by reference. While the compilation pipeline supports multiple C++ files, this section assumes a single-file approach for clarity.

Step (1): Lowering MLIR to LLVM IR The first step in the compilation process is lowering the MLIR code to LLVM IR. This involves progressively lowering the `particles` dialect to standard dialects (e.g., `scf`, `func`), followed by further lowering these dialects using integrated conversion passes. During this process, two runtime files are created (Step (1a)). For CUDA GPUs, the runtime must be compiled to llvm bitcode and reintegrated into the MLIR lowering pipeline (Step (1b)). The final output, `simulation.ll`, contains the host-side LLVM IR code and, when targeting CUDA, the compiled device code as a binary blob.

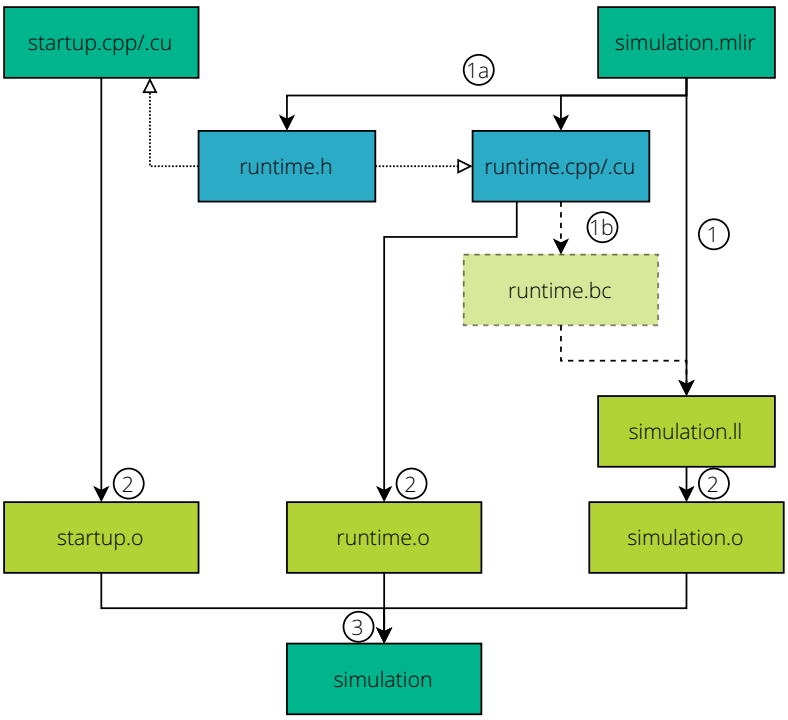

Figure 3.2: Full compilation stack. Dotted lines lined with hollow arrow tips represent inclusion. Dashed lines with solid arrow tips represent optional steps that are only executed when targeting CUDA GPUs. Solid lines with arrow tips represent transformation steps.

Step ①a: Generating Runtime Files As part of lowering the `particles` dialect to standard dialects, a runtime is generated. The primary purpose of this runtime is to act as a bridge between MLIR and C++/OpenFPM, enabling the use of OpenFPM's data and procedures within MLIR. Two runtime files are created: a header file and a source file. While their names and paths are configurable, they are referred to as `runtime.h` and `runtime.cpp/.cu` for simplicity. The header file declares all runtime functions and defines all required data types, including fully parametrized OpenFPM template classes and storage structs. The source file (`runtime.cpp/.cu`) defines all runtime functions declared by the header. Since it relies on the data types defined in the header file, it includes `runtime.h`. The header file is also typically included in `startup.cpp/.cu`.

Step ①b: Compiling and Reintegrating CUDA Device Code Into the MLIR Lowering Pipeline When targeting CUDA GPUs, the device code in `runtime.cu` must be compiled and reintegrated into the lowering pipeline. This is because all CUDA device code must be compiled and linked before `simulation.mlir` can be fully lowered to LLVM IR. For CPUs, all object files can simply be linked during the final compilation stage.

Step ②: Compiling Everything Into Object Files With the runtime files generated and `simulation.mlir` lowered to LLVM IR, the next step is to compile all components into object files:

- `.cpp` files are compiled using `mpic++` with either `clang++` or `g++` as the base compiler.
- `.cu` files are compiled with `nvcc`.
- `.ll` files are either directly compiled into an object file using `clang++`, or first compiled into assembly using `llc` and then compiled into an object file with `g++`.

Step ③: Linking In the final step, all object files are linked together to produce the executable. Device and host code are linked separately beforehand to facilitate link-time optimization.

3.4 MLIR Lowering Pipeline

This section provides an overview of the MLIR lowering pipeline, focusing on the dialects and transformation passes developed to lower the `particles` dialect to standard dialects while enabling DMP and integrating OpenFPM. Whether CPUs or CUDA GPUs are targeted does not significantly affect the overall lowering pipeline, with each hardware target requiring only slight adaptions. Figure 3.3 illustrates the involved key dialects and transformation steps. Each number denotes a step, with each step representing one more combined passes. The pipeline begins with the `particles` dialect and concludes with standard dialects, meaning after all steps have been performed, only standard dialects remain in the IR. Lowering these dialects is not covered in this work, as passes integrated into MLIR are used to complete their conversion. Appendix A.2 provides a complete overview of the passes comprising the lowering pipeline.

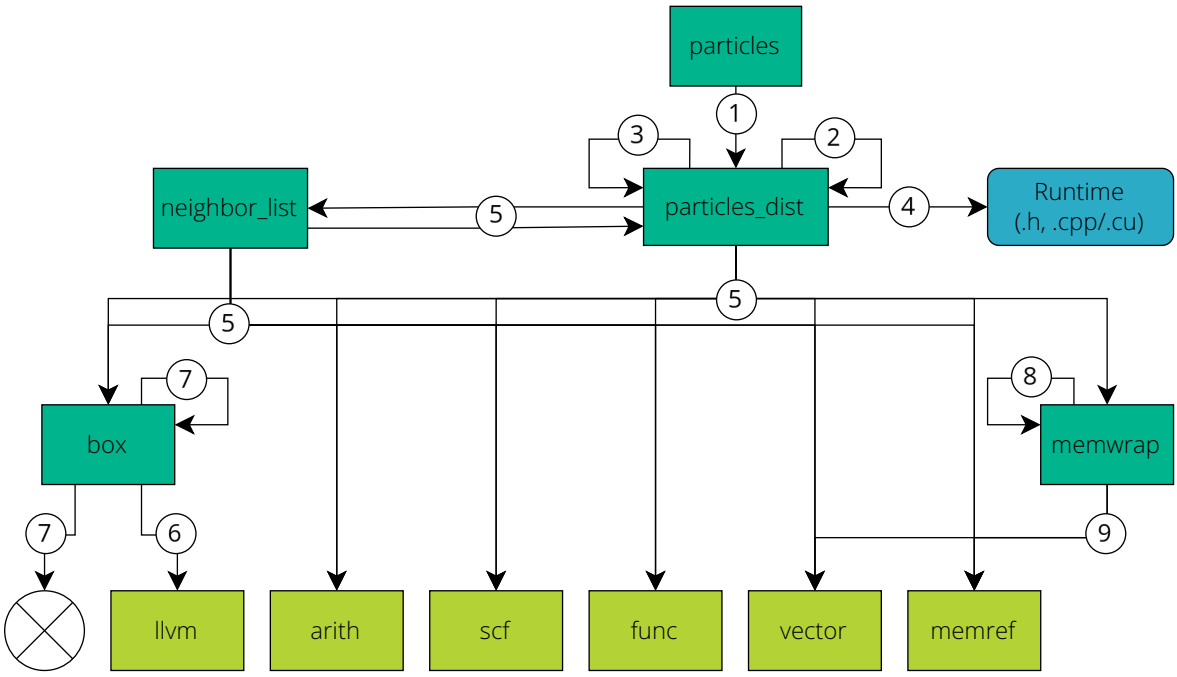

Figure 3.3: MLIR lowering pipeline

The `particles` Dialect The `particles` dialect encapsulates particle simulations. Its IR constructs are centered around iterating over particle sets and neighboring particle pairs, as well as modifying the particles within these sets. As outlined in Section 3.2, its objective is to be general-purpose and target-agnostic. It therefore does not contain any DMP or OpenFPM-specific IR constructs.

The `particles_dist` Dialect The `particles_dist` dialect is a specialization of `particles`. A specialization dialect adapts a "root" dialect to a specific target, e.g. hardware or execution

model, by adopting and extending its IR constructs while also introducing new ones. This enables the introduction of target-specific operations, types, and attributes and the adaption of existing ones while ensuring that the base dialect remains general-purpose and target-agnostic. The `particles_dist` dialect adapts `particles` to DMP while also enabling the integration of OpenFPM. Furthermore, it introduces the concept of neighbor lists.

Step ①: Specialization of `particles` to `particles_dist` The first step in lowering the `particles` dialect is to replace each `particles` operation, type, or attribute in the IR with its `particles_dist` counterpart. This process eliminates the `particles` dialect from the IR.

Step ②: Optimizations Since many optimizations require target-specific information, they can only be performed after specialization. The current set of optimizations focuses on aggressive loop fusion for different loop-like operations.

Step ③: Placement of Update and Communication Operations Update and communication operations are inserted where necessary, leveraging several specifically developed DFA variants to guide the placement.

Step ④: Runtime Generation The fourth step involves creating the runtime. Runtime generation must occur after Step ③, as the presence of specific update and communication operations in the IR dictates which runtime functions must be created.

Neighbor List Dialects The `particles_dist` dialect introduces the concept of neighbor lists. Rather than incorporating all supported neighbor lists directly into `particles_dist`, however, each kind is encapsulated within its own separate companion dialect of `particles_dist`. Two neighbor list dialects have been developed. The `cell_list` dialect is, as the name suggests, a dialect encapsulating cell lists. The `local_domain` dialect, on the other hand, does not represent a conventional neighbor list. Instead, it treats all particles within the same local subdomain as neighbors. Both dialects inherit from a common `neighbor_list` dialect.

Step ⑤: Lowering `particles_dist` and Its `neighbor_list` Companion Dialects Once all additive, generative, and optimizing passes have been completed, the process of lowering `particles_dist` and its `neighbor_list` companion dialects is performed. Because the `particles_dist` and its neighbor list dialects are closely intertwined, they must be lowered in a combined process.

The `box` Dialect The `particles_dist` dialect and its companion dialects define complex types that must be converted into aggregate types such as `!llvm.struct`. The `!llvm.struct` type, however, imposes several restrictions that make it cumbersome to use. To address this, the `box` dialect introduces a more flexible aggregate type: `!box.box`. It also introduces a storage type, `!box.storage`, to support loading and storing `!box.box` values. Most significantly, however, the `box` dialect comes with a set of expansion and elimination passes that enable 1-to-N conversion. 1-to-N conversion is the process of replacing an

aggregate type or value with its constituent elements, which is crucial for enabling subsequent optimizations.

Step ⑥: Lowering the `!box.storage` Type and the `box.load` and `box.store` Operations To store a `!box.box` value, it must be converted into an `!llvm.struct` during storage and transfer. This conversion is automatically performed while lowering the load and store operations of the box dialect. During this process, `!box.storage` types are also converted. All other box operations and types are not lowered and remain in the IR.

Step ⑦: Expanding and Eliminating the `!box.box` Type and Associated Operations All `!box.box` types and values are replaced by their constituent elements via a series of expansion and elimination passes. This eliminates all remaining box operations.

The `memwrap` Dialect The memwrap dialect provides convenient wrappers around load and store operations from the `memref` and `vector` dialects. Its main purpose is to enable the elimination of redundant store operations that store unchanged particle values.

Step ⑧: Eliminating Redundant `memwrap` Store Operations Redundant memwrap store operations are eliminated.

Step ⑨: Lowering `memwrap` Operations The memwrap operations that were not eliminated are lowered. After this step, only types and operations from the standard dialects remain in the IR.

Phases The steps outlined in this section are consolidated into four phases, as shown in Table 3.2. The following chapters individually discuss these phases.

Phase	Steps	Description	Chapter
Phase 1	①	Input and Specialization	4
Phase 2	② ③	Target-Specific Transformations	5
Phase 3	④ ⑤	Lowering `particles_dist`	7
Phase 4	⑥ ⑦ ⑧ ⑨	Lowering box and memwrap	8

Table 3.2: Phases of the MLIR lowering pipeline

4 Phase 1: Input and Specialization

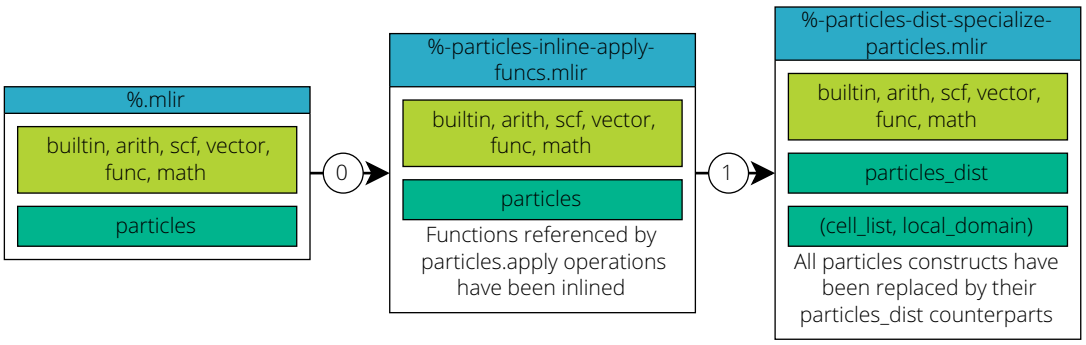

Figure 4.1: Phase 1 takes the unmodified IR as input and has two steps. The first step (overall step ⓪) inlines functions referenced by `particles.apply` operations. The second step (overall step ①) specializes `particles` to `particles_dist`.

Phase 1 focuses on three dialects, as well as the relations and transformations between them: The `particles` dialect, which captures the domain of particle simulations; the `particles_dist` dialect, which is a specialization of the former to DMP and OpenFPM; and the `particles_base` dialect, which forms the basis for the other two. The first section of this chapter introduces the concepts of abstract dialects, dialect inheritance, and specialization dialects, which form the foundation of the three implemented dialects. Next, Section 4.2 focuses on the approaches used to enable robust and efficient code analysis, necessary for optimizations like loop fusion and the informed placement of update and communication operations. Section 4.3 introduces the abstract base dialect `particles_base`. Afterwards, its inheriting dialects, `particles` and `particles_dist`, are described in Section 4.4. Finally, Section 4.5 discusses the transformation passes depicted in Figure 4.1.

4.1 Dialect Extension Methods

4.1.1 The Concepts of Abstract Dialects and Dialect Inheritance

The realization of the concepts presented in this section are very specific to the multi-level IR framework used to implement the dialects and their definitions. As xDSL was almost exclusively used in this work, this section focuses only on this framework. However, the basic concept of mapping dialect inheritance to class inheritance is transferable to MLIR.

As outlined in Section 2.4, dialects are collections of operation, type, and attribute definitions. An abstract dialect consists exclusively of abstract definitions, and a dialect is said to inherit from an abstract dialect if it inherits all of its definitions. In xDSL, operation, type, and attribute definitions are implemented as Python classes that use specific decorators, such as `@irdl_op_definition` and `@irdl_attr_definition`. An abstract definition is a definition class that is missing its decorator. In addition to being unfinished, these definition classes may be abstract, generic, or even parametrized metaclasses. Inheriting from an abstract definition directly corresponds to extending the unfinished definition class and finishing its implementation, which at the very least involves specifying the name of the construct and adding the corresponding decorator. An abstract definition can specify anything a normal definition can. For example, abstract operation definitions can specify operands, results, regions, properties, traits, and verification requirements. The inheriting definition retains all elements of the base definition while also being able to introduce new components such as additional operands and results.

Abstract IR construct definitions are not a new concept and are already utilized in both xDSL and MLIR. Abstract operation definitions are commonly used when multiple operations share the same or a highly similar syntax. For example, binary floating-point operations such as `arith.addf`, `arith.mulf`, `arith.subf`, and `arith.divf` all share the same syntax. Instead of redundantly implementing each operation definition separately, it is more efficient to define one abstract base definition that these operations inherit from. Abstract dialects simply take this approach and apply it to all operation, type, and attribute definitions of a dialect.

Abstract dialects are useful when two or more dialects represent similar or hierarchical concepts and therefore share a significant portion of their IR structure. For example, the dialects `cell_list` and `local_domain`, introduced later in this work, both represent neighbor lists and are used to iterate over the neighbor indices of a particle. Although internally these neighbor lists work differently, the dialects share a highly similar syntax, differing only in the parameters of a single type. This makes them ideal candidates for dialect inheritance. Introducing the abstract base dialect `neighbor_list`, which contains all common operations and types, reduces code duplication and improves maintainability. Furthermore, it also facilitates adding new neighbor list dialects in the future, such as a dialect for Verlet lists.

Dialects can have internal dependencies between their operations, types and attributes. When a dialect inherits from an abstract dialect that has internal dependencies, these dependencies must be internalized by the inheriting dialect. For example, as visualized in Figure 4.2, the abstract `neighbor_list.for_all_neighbor_indices` operation takes an operand of type `!neighbor_list.list`, which itself is an abstract type definition of the same dialect. When the `cell_list` dialect inherits all construct definitions from `neighbor_list`, it also inherits these operation and type definitions. However, in its case, the operand must be of the subtype `!cell_list.list`. This adaption requires class parametrization. In the case of xDSL, this is achieved by using generic metaclasses.

4.1.2 The Concept of Specialization Dialects

Simply speaking, a specialization dialect is a copy of another dialect, referred to as "root", tailored to one or more specific targets. A target can be, for example, a processing platform, such as FPGA or CIM, or an execution model, such as DMP. The specialization dialect clones all

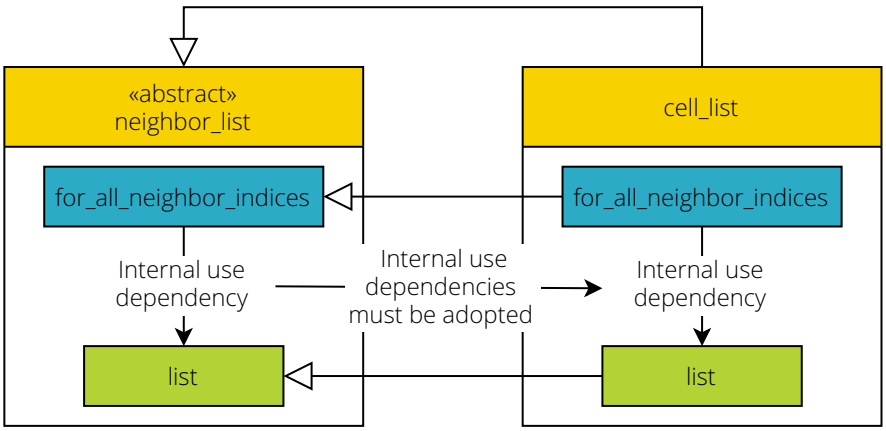

Figure 4.2: Adoption of internal dependencies during dialect inheritance. `neighbor_list` has internal dependencies. Inheriting dialects, such as `cell_list`, must adopt these internal dependencies.

operations, types, and attribute definitions of the root dialect, potentially extends them, and optionally introduces new ones. Operation definitions can be extended by adding operands, results, properties, regions, traits and verification steps. Similarly, types and attributes are extending by integrating additional parameters and verification requirements.

Dialect specialization and dialect inheritance are closely related, as both serve as forms of dialect extension. However, they differ in one crucial aspect: a specialization dialect does not inherit from its root dialect and the root dialect itself is not abstract but rather a fully realized dialect. This also means that the definitions of the specialization dialect do not inherit the IR construct definitions of the root dialect, since those are also fully realized. Instead, the specialization dialect copies the construct definitions of the root dialect and extends the copies.

The distinction between dialect specialization and dialect inheritance arises primarily from limitations of the underlying xDSL implementation. As discussed in Section 4.1.1, IR construct definition classes use specific decorators, and it is not possible to extend classes that have such a decorator. Furthermore, internal use dependencies within an abstract base dialect are handled via parametrized metaclasses, which also makes it impossible for that dialect to be fully realized. Consequently, there is no "is a" relationship between a root dialect and its specialization dialect, whereas such a relationship does exist between an abstract base dialect and its inheriting dialect.

In contrast to its root dialect, a specialization dialect is not intended to be targeted directly foreign transformation passes or lowering tools. Instead, it serves as part of the lowering infrastructure that lowers the target-agnostic root dialect to the targets of the specialization dialect. A key step in this process is the specialization pass, which completely replaces all constructs of the root dialect with their specialized counterparts. During or after this replacement step, target-specific information can be derived, operations placed, and transformations performed. The combination of root and specialization dialects together with a specialization pass allows the root dialect to remain target-agnostic and re-usable, while also enabling the integration of essential targeting-dependent elements.

As discussed in Section 3.2, the `particles` dialect should remain general-purpose and target-agnostic. The lowering pipeline, however, targets DMP while also integrating OpenFPM. This requires extending the constructs of the `particles` dialect with target-specific elements, such as OpenFPM- and DMP-specific operation-properties, or type-parameters. It also necessitates the introduction of entirely new operations, such as communication and

update operations. Including these target-specific elements in the `particles` dialect would violate its objective to remain target-agnostic. As a solution, the `particles_dist` dialect was introduced as a specialization of `particles`. It contains all operations and types of the `particles` dialect, while also integrating DMP and OpenFPM specific elements.

Dialect inheritance can be utilized to implement dialect specialization. By extracting all IR construct definitions of the root dialect into an abstract base dialect, both, the root dialect and the specialization dialect, can inherit these definitions. For example, as illustrated in Figure 4.3, both `particles` and `particles_dist` inherit from the abstract dialect, `particles_base`, which contains all shared definitions. This approach minimizes code duplication, thus improving maintainability. It also increases extensibility, as it facilitates the introduction of future specialization dialects. For example, it might be necessary to specialize `particles` to a specific hardware platform in the future, such as FPGA. Instead of re-implementing everything from the `particles` dialect, these new dialects can simply extend `particles_base`.

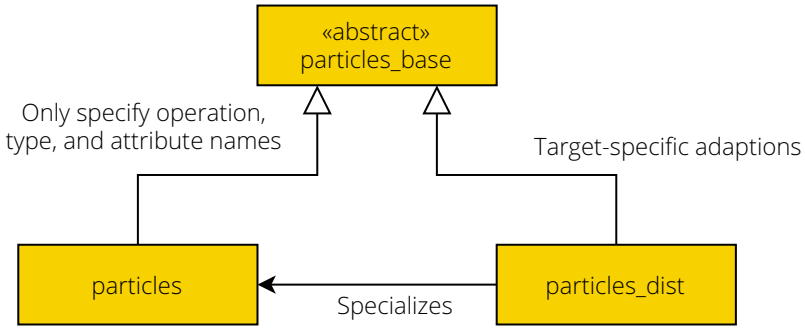

Figure 4.3: Relationships between `particles_base`, `particles`, and `particles_dist`

4.2 Enabling Robust Code Analysis

Section 3.2 lists multiple short-term objectives necessitating that the IR structure of the `particles` dialect, and consequently also its root dialect `particles_base`, enables robust analysis. This section introduces two concepts that are applied to the type representing particle sets, `!particles_base.set`, and the operations acting on it to facilitate the placement of update and communication operations and optimizations such as loop fusion.

4.2.1 Fake Value Semantics and Modification Graphs

Memory Semantics In this work, the term "memory semantics" refers to the combination of types that represent memory locations and operations that act on such types by loading data from or storing data in these locations, or by modifying the memory in-place. For example, the `memref` type is a reference to a region of memory. Using operations like `memref.load` and `memref.store`, data can be read from or written to the referenced region. Copying a `memref` value does not duplicate the memory contents; instead, it creates another reference to the same region. Furthermore, acting on a `memref` value, through operations such as `memref.store`, does not preserve the previous memory contents, which are overwritten and cannot be recovered.

Value Semantics The term "value semantics", in this work, refers to the combination of types that represent copyable data and operations that act on such types by producing modified copies while fully preserving the original values. For example, the `!llvm.struct` type can be modified using `llvm.insertvalue` operations. Applying such an operation to an `!llvm.struct` value produces a new `!llvm.struct` value, while leaving the original struct unchanged and available for further use.

Def-Use Relation In MLIR, value semantics offer a significant advantage over memory semantics: they enable simpler and more reliable analysis by naturally forming def-use relationships. Under this relationship, operation B relates to operation A if B uses one of the results produced by A as input. Similarly, operation B relates to block argument A if B uses A as input.

Construction of Modification Graphs for Types With Value Semantics Def-use relationships can be used to construct modification graphs. Modification graphs trace the modifications of a value from its origin to its final uses. Figure 4.4 shows an example of a modification graph for an `!llvm.struct` value. The nodes within a modification graph represent either operations (e.g., ⓪) or block arguments (e.g., ①). Two nodes, A and B, are connected via a directed edge if:

- Neither A nor B are control-flow parent operations (i.e., not `scf.if` or `scf.for`) and B uses the output of A as input, with the output of a block argument being the block argument itself (e.g., ① → ②, ② → ③).
- B is an `iter_arg` of an `scf.for` operation and uses the output of A as an initial value (e.g., ⓪ → ①).
- A is an `scf.yield` operation, and B uses the corresponding output of A's parent operation as input (e.g., ⑦ → ⑪, ⑪ → ⑫).
- A is an `scf.yield` operation associated with an `scf.for` parent, and B is an `iter_arg` block argument (e.g., ⑪ → ①) of that parent.

Neither `scf.if` nor `scf.for` operations appear in modification graphs, however, their associated `scf.yield` operations and block arguments do.

Difference to Control-Flow Graphs Even though modification graphs look similar to CFGs, there are some distinct differences. A branch in a CFG indicates diverging control flow, whereas a branch in a modification graph indicates that two operations use the same value as input. This may result from diverging control flow, as is the case with ⑥ and ⑧, but may also occur within the same basic block, as illustrated by ② and ④. Furthermore, CFGs have a single sink that represents the end of the program flow, whereas modification graphs can have multiple sinks, where each sink represents the use of a value that does not result in a modified output (e.g., ⑩). However, these uses can coincide with the end of the program flow (e.g, ⑫). Similarly, CFGs have a single source that represents the start of the program flow. While modification graphs also have a single source, this source represents the program point where the modified value originates. The origin of a value can coincide with the start of program flow, for example if it is a block argument of a `func.func` operation, but does not have to (e.g., ⓪).

```
!struct =  !llvm.struct<(index, index)>
func.func @foo() {
  %0 = llvm.mlir.undef : !struct                    // 0

  %9 = scf.for %step = %zero to %stop step %step
             iter_args(%1 = %0) -> (!struct) {      // 1

    %2 = llvm.insertvalue %one, %1[0] : !struct     // 2
    %3 = llvm.insertvalue %two, %2[1] : !struct     // 3

    %4 = llvm.insertvalue %two, %1[0] : !struct     // 4
    %5 = scf.if %cond -> (!struct) {                // 5
      %6 = llvm.insertvalue %one, %4[1] : !struct   // 6
      scf.yield %6 : !struct                        // 7
    } else {
      %7 = llvm.insertvalue %two, %4[1] : !struct   // 8
      scf.yield %7 : !struct                        // 9
    }

    %8 = llvm.extractvalue %3[0] : !struct          // 10

    scf.yield %5                                    // 11
  }
  return %9 : !struct                               // 12
}
```

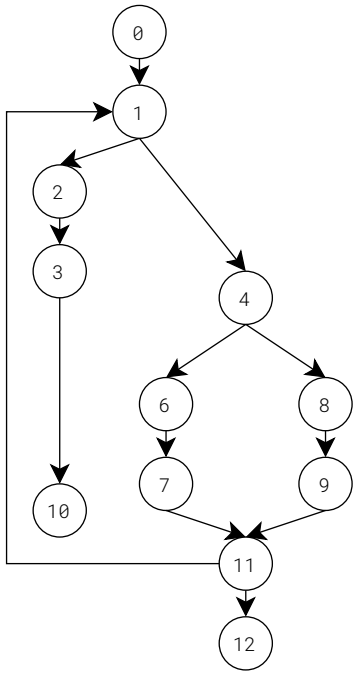

Figure 4.4: Modifying structs in MLIR and corresponding modification graph

Construction of Modification Graphs for Types With Memory Semantics Constructing modification graphs for types with memory semantics is significantly more difficult, as it cannot be directly derived from def-use relationships and must instead be inferred through more sophisticated methods. Furthermore, a single memory region may have multiple references, and at compile time it is often unclear whether two references point to the same or different regions. This uncertainty further increases the complexity of constructing modification graphs for types with memory semantics.

Particle Sets Have True Memory Semantics Particle sets are far too large to be treated as values. Modifying the particles of a set, therefore, does not produce a new set with modified copies of the original particles. Instead, all particles are all stored in memory and modified in-place. This implies that, in practice, particle sets exhibit memory semantics. However, as discussed in the previous paragraph, memory semantics significantly complicate the construction of modification graphs. Analysis of particle set modification graphs is essential for subsequent transformations, such as loop fusion and the placement of update and communication operations.

Fake Value Semantics To enable the construction of modification graphs for particle sets, fake value semantics were introduced for the `!particles_base.set` type. Like true value semantics, operations acting on `!particles_base.set` types take a value as input and produce a new value of the same type as output. However, unlike real value semantics, the previous `!particles_base.set` value is not preserved and cannot be reused. The original set is still available as SSA value, but it must not be used again as it has been modified. Essentially, operations acting on `!particles_base.set` values consume a reference to a particle set, modify the set in-place, and return a new reference to the same set, which

must be used by the next operation. This allows the `!particles_base.set` type to exploit the advantages of value semantics, while using memory semantics under the hood. The approach of wrapping memory references with value-semantics was heavily inspired by the `!stencil.temp` type of the `stencil` dialect [22].

Further Advantages of Fake Value Semantics In addition to leveraging the advantages of true value semantics, fake value semantics offer further benefits. They enable a hybrid approach where some values are buffered and directly available as SSA values, while the majority of the data remains in memory and is accessed via references. For example, the size of a particle set is buffered this way and thus can be used at any time without first needing to retrieve it from memory. Additionally, fake value semantics allow transient state to be associated with the memory reference(s) they wrap. Section 5.2.3 discusses staleness analysis and staleness tracking. Staleness tracking uses boolean flags to track the staleness state of certain data structure associated with a particle set. These staleness flags have no persistence and are lost when the corresponding `!particles_base.set` is no longer in use, and are therefore considered transient state.

4.2.2 Single-Source Single-Sink Modification Graphs

Single-source single-sink modification graphs are modification graphs with additional restrictions that make them more suitable for analysis. In this work, these restrictions are partly enforced through syntactic constraints, and partly through the `particles_base.loop` operation, which, besides modeling a time stepping loop, serves as a container for all computations involving particle sets. This enables the `particles_base.loop` operation to enforce global constraints that are difficult to enforce using only the operations acting on `!particles_base.set` values, such as verifying that a `!particles_base.set` value is never used twice.

The following restrictions are imposed:

1. The only origin of `!particles_base.set` values is the first argument of `particles_base.loop` regions.
2. All non-terminator operations that act on `!particles_base.set` values must consume exactly one set value and return exactly one new set, even if they do not modify any data. This differs from, for instance, the `!llvm.struct` type, where operations such as `llvm.extractvalue` do not produce a new `!llvm.struct` value.
3. Each `scf.if` operation can return at most one `!particles_base.set` value, and both regions of such an `scf.if` operation must use the same `!particles_base.set` value as input for their computations.
4. For each `particles_base.loop` region, the modification graph of `!particles_base.set` values inside must form a Directed Acyclic Graph (DAG) with a single source (the `!particles_base.set` region argument) and a single sink (the `particles_base.next` terminator operation). Loops acting on particle sets, such as those modeled by `scf.for`, are currently not supported.

Figure 4.5 shows an example of a modification graph where all restrictions are upheld. Modification graphs that adhere to all restrictions guarantee that no outdated `!particles_base.set` values are used. Under these constraints, reusing a `!particles_base.set` value would require that another particle set value remains unused, which in turn creates

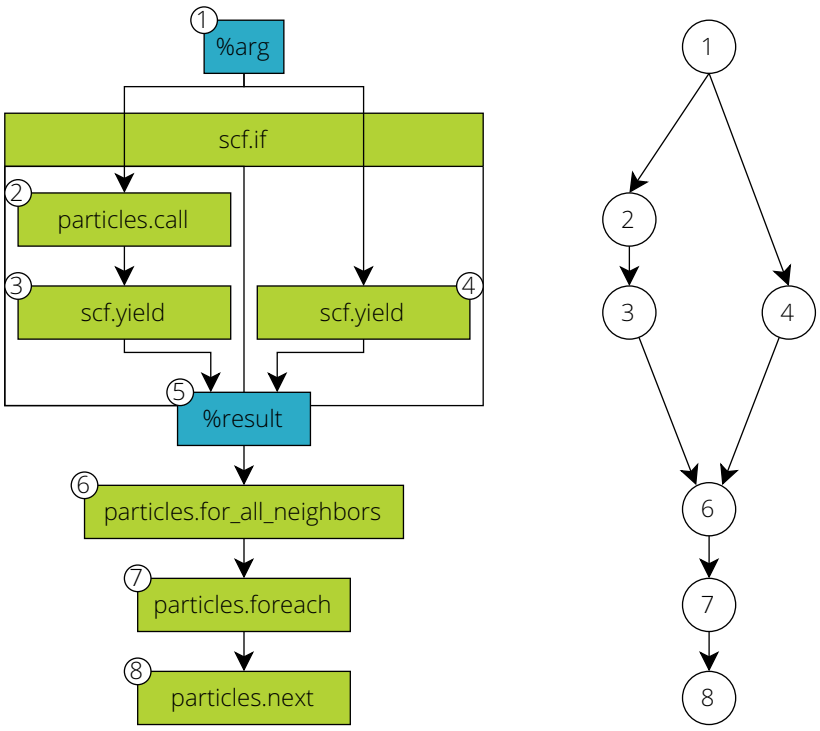

Figure 4.5: Left: region of a `particles.loop` operation. Right: corresponding modification graph.

a sink other than the `particles_base.next` operation. This does not imply that every `!particles_base.set` value must have exactly one use, as multiple uses are still possible in the presence of `scf.if` operations (e.g., node ①). However, due to restriction 3, branches in the modification graph fully coincide with control flow branches, and multiple uses arising from such branches do not constitute reuse.

In addition to facilitating verification, single-source single-sink modification graphs have another advantage: They are essentially CFGs that capture only the operations modifying the `!particles_base.set` value provided as region argument. This enables them to be used for DFA.

To facilitate the construction and analysis of the modification graph, all definitions of operations that act on `!particles_base.set` values must implement the `Particle` `SetConsumerProducer` interface. This interface requires each operation to provide two information: the `!particles_base.set` value it consumes, and the particle set result it produces.

4.3 The `particles_base` Dialect

As outlined in Section 4.1.2, the `particles_base` dialect forms the abstract basis for both the `particles` and `particles_dist` dialects. Because the `particles_base` dialect is abstract, its operations, types, and attributes cannot appear in the IR and therefore do not require names. Nonetheless, for clarity, its constructs are assigned names using the prefix `particles_base`.

4.3.1 Types

The !particles_base.set Type

The !particles_base.set type represents a loaded particle set. As explained in Section 4.2.1, this type and its associated operations employ fake value semantics to facilitate the construction of modification graphs, as well as to buffer values and maintain transient state.

Example:

```
!baseset = !particles_base.set<
  -1 : index,                    // size
  f64,                           // position_type
  3 : index,                     // dimensionality
  {                              // properties
    "velocity" = vector<3xf64>,
    "force" = vector<3xf64>
  }
>
```

Listing 4.1: !particles_base.set example (!baseset)

Parameters The !particles_base.set type has four parameters:

- **size**: An index attribute that specifies the size of the particle set. It must either be a positive integer or -1. A value of -1 indicates that the size of the particle set is dynamic and may change during execution.
- **position_type**: The numeric type used for positions. Accepted types are currently limited to f64 or f32.
- **dimensionality**: An index attribute that specifies the spatial dimensionality of the computational domain.
- **properties**: A dictionary attribute mapping particle property names to their types. Accepted property types are currently limited to scalars and single-rank vector types.

The !particles_base.set type only contains parameters required for validation and code generation. Parameters currently not used for these purposes, such as boundary conditions, which define how the particles should behave at the edges of the computational domain or even the size of the computational domain itself, were intentionally excluded.

The !particles_base.storage Type

The !particles_base.storage type represents an unloaded particle set that, together with its operations, employs memory semantics. As a result, it is not associated with any transient state or buffered values, but simply holds the necessary references required to work with a particle set and its associated data structures, such as neighbor lists. It expects a single parameter of type !particles_base.set, indicating the particle set type it stores.

Example:

```
!basestorage = !particles_base.storage<!baseset>
```

Listing 4.2: !particles_base.storage example (!basestorage)

Integration with C++ Section 3.2 lists integration with C++ as a semi-function requirement Ⓢ2. This necessitates the ability to pass particle sets between C++ and MLIR. Passing a `!particles_base.set` with all its buffers and transient state would make the corresponding C++ data structure very large and unwieldy, while also exposing the internal state of the particle set to C++, which may not always be desired. Furthermore, as outlined in the same section, passing or returning structs between MLIR and C++ must be done by reference. The `!particles_base.storage` type addresses this by lowering to a simple pointer to a struct containing the references to all integrated OpenFPM instances.

The `!particles_base.particle` Type

The `!particles_base.particle` type represents a particle. Together with its associated operations, it employs value semantics. Whether these value semantics are real or fake is determined by the implementation and chosen during lowering. For example, a `!particles_base.particle` value may represent a set of buffered values that have real value semantics or may wrap a reference to a particle using fake value semantics. In the context of this work, the value semantics are real. Note that this does not imply that particles are copied each time they are modified, but rather that, in their buffered state, they exhibit value semantics. The `!particles_base.particle` type has a subset of the parameters of the `!particles_base.set` type: **`position_type`**, **`dimensionality`**, and **`properties`**.

Example:

```
!baseparticle = !particles_base.particle<
  f64,                          // position_type
  3 : index,                    // dimensionality
  {                             // properties
    "velocity" = vector<3xf64>,
    "force" = vector<3xf64>
  }
>
```

Listing 4.3: `!particles_base.particle` example (`!baseparticle`)

4.3.2 Operations

The `particles_base.loop` Operation

The `particles_base.loop` operation represents a time-stepping loop with an initialization phase. It has two regions: one for initialization and one for the main loop. It takes a `!particles_base.storage` value as input, transparently loads it into a `!particles_base.set` and passes it to the initialization region. From the initialization region, the initialized particle set is then passed to the time-stepping region, where it is propagated from iteration to iteration until the loop terminates.

Example:

```
%iter_arg_result = "particles_base.loop" (%storage,
                                          %init_steps,
                                          %upper_bound_steps,
                                          %iter_arg)
({^0(%set : !baseset, %step : index, %iter_arg0 : f64):   // initialization region
  >> Initialize %set -> %initialized_set
  >> Update %iter_arg0 -> %initialized_iter_arg
  "particles_base.next" (%initialized_set,
                         %initialized_iter_arg): (!baseset, f64) -> ()

},
{^0(%set : !baseset, %step : index, %iter_arg0 : f64):    // time-stepping region
  >> Update %set -> %updated_set
  >> Update %iter_arg0 -> %updated_iter_arg
  "particles_base.next"(%updated_set,
                        %updated_iter_arg): (!baseset, f64) -> ()
}) : (!basestorage, index, index, f64) -> (f64)       // Returns final values of iter_args
```

Listing 4.4: `particles_base.loop` example

Operands The `particles_dist.loop` operation has four operands. The last operand, `iter_args`, is variadic and also determines the results of the operations. The operands are expected in the following order:

- **`storage`**: The `!particles_base.storage` value to be loaded and passed onto the initialization region.
- **`init_steps`**: An `index` value specifying the initial time step count.
- **`upper_bound_steps`**: An `index` value specifying the non-inclusive upper limit for the number of time steps. The actual number of time steps executed will be `init_steps - upper_bound_steps`.
- **`iter_args`**: A variadic set of values that are passed alongside the particle set, similar to `iter_args` of the `scf.for` operation. The results of the `particles_base.loop` operation hold the final values of the `iter_args`.

Additional Purposes Other than modeling a time-stepping loop, the `particles_base.loop` operation has two additional purposes. The first purpose, as outlined in Section 4.2.2, is to enforce the correct usage of `!particles_base.set` values by ensuring its regions form single-source single-sink modification graphs for their `!particles_base.set` region arguments. The second purpose is to transparently switch from the memory semantics employed by the `!particles_base.storage` type on the outside to the fake value semantics employed by the `!particles_base.set` type on the inside. Since this is the only way to load a `!particles_base.set`, the `particles_base.loop` operation naturally acts as a container for particle computations.

The `particles_base.next` Operation

The `particles_base.next` operation terminates all `particles_base.loop` regions and has two operands. The `set` operand takes the `!particles_base.set` value to be passed from the initialization region to the first iteration, or from one iteration to the next. The variadic `iter_args` operand takes a set of values to propagate alongside the particle set.

Example:

```
"particles_base.next"(%set, %iter_arg): (!baseset, f64) -> ()
```

Listing 4.5: `particles_base.next` example

The `particles_base.foreach` Operation

The `particles_base.foreach` operation has a single region, which is executed exactly once for each particle in the given particle set. Whether these executions happen in parallel or sequentially is up to the implementation. Each particle is transparently loaded at the start and stored at the end of the region. In between, each particle can be modified.

Example:

```
%evolved = "particles_base.foreach"(%set) ({
^0(%particle : !baseparticle):
  >> Modify %particle -> %updated_particle
  "particles_base.yield" (%updated_particle) : (!baseparticle) -> ()
}) : (!baseset) -> !baseset
```

Listing 4.6: `particles_base.foreach` example

Within a `particles_base.foreach` region, operations with side effects are forbidden. This ensures that there are no unforeseen data dependencies and all executions are fully isolated from each other.

The `particles_base.for_all_neighbors` Operation

The `particles_base.for_all_neighbors` operation has a single region that is executed once for every pair of neighboring particles. Within this region, pairwise interaction results between a particle and its neighbor can be calculated. Each pairwise interaction can have multiple results, which are yielded using the `particles_base.yield` operation. For each particle, the results from interacting with all its neighbors will be transparently reduced and then stored on the particle. The definition of what constitutes a neighborhood is intentionally left vague. Inheriting dialects must add this information themselves by, for example, adding properties.

Example:

```
%interacted = "particles_base.for_all_neighbors" (%set)
<{
  "with_self" = false,
  "reduction_kinds" = [#particles_base.reduction_kind<add>],
  "write_targets" = ["force"]
}>
({^0(%particle : !baseparticle, %neighbor : !baseparticle):
  >> Calculate the %force that %neighbor exhibits on %particle
  "particles_base.yield" (%force) : (vector<3xf64>) -> ()
}) : (!baseset) -> (!baseset)
```

Listing 4.7: `particles_base.for_all_neighbors` example

Properties The `particles_base.for_all_neighbors` operation requires the following properties:

- **with_self**: A boolean attribute specifying whether a particle should be considered its own neighbor and interact with itself.
- **reduction_kinds**: An array attribute specifying how each pairwise interaction result should be reduced. Currently, the only supported reduction kind is #particles_base.reduction_kind<add>, which indicates summation. Reductions are performed for each result index independently. Therefore, if each pairwise interaction has two results, reduction_kinds must also contain two corresponding entries.
- **write_targets**: An array attribute, with each entry specifying where the reduced interaction result with the same index should be stored on the particle. Each entry must be either "pos" or the name of a particle property.

Limitations of Data Dependencies As with the particles_base.foreach operation, operations with side effects are forbidden within a particles_base.for_all_neighbors region. Additionally, this region must not contain any data dependencies between particles, as these would constitute data races. For example, a particles_base.get_property operation cannot read the particle property X from a neighbor while the same property is a write_target. This constitutes a data race unless all pairwise interactions are performed before the reduced interaction results are stored on the particles. However, this would require implicitly buffering the reduced values, which is currently not supported. The particles_base.for_all_neighbors operation ensures that no such data dependencies exist during verification.

The particles_base.yield Operation

The particles_base.yield operation terminates the regions of both the particles_base.foreach and particles_base.for_all_neighbors operations. When used in a particles_base.foreach region, it yields the modified particle, whereas when used to terminate a particles_base.for_all_neighbors region, it yields the results of the pairwise interaction.
Example:

```
// Inside a particles_base.foreach it yields the modified particle
"particles_base.yield" (%udpated_particle) : (!baseparticle) -> ()

// Inside a particles_base.for_all_neighbors operation
// it yields the results of a pairwise interaction
"particles_base.yield" (%force) : (vector<3xf64>) -> ()
```

Listing 4.8: particles_base.yield example

The particles_base.get_position and particles_base.set_position Operations

The particles_base.set_position operation retrieves the position of the given !particles_base.particle value and returns it as vector. The particles_base.set_position operation overwrites the position of the given particle with the provided vector and returns the modified !particles_base.particle.

Example:

```
// Get position:
%pos = "particles_base.get_position"(%particle) : (!baseparticle) -> vector<3xf64>
// Modify position:
%updated_particle = "particles_base.set_position"(%particle, %new_pos)
                                        : (!baseparticle, vector<3xf64>) -> !baseparticle
```

Listing 4.9: `particles_base.get_position` and `particles_base.set_position` example

The `particles_base.get_property` and `particles_base.set_property` Operations

The `particles_base.get_property` operation retrieves the specified particle property of the given `!particles_base.particle` value and returns it. The `particles_base.set_property` operation overwrites the specified particle property of the given particle with the provided value and returns the modified `!particles_base.particle`. Which particle property to retrieve or modify is specified via the `property` operation property, which takes the form of a string attribute.

Example:

```
// Retrieve force value:
%force = "particles_base.get_property"(%particle) <{"property" = "force"}>
                                       : (!baseparticle) -> vector<3xf64>
// Modify force value:
%updated_particle = "particles_base.set_property"(%particle, %new_force)
                                         <{"property" = "force"}>
                                         : (!baseparticle, vector<3xf64>) -> !baseparticle
```

Listing 4.10: `particles_base.get_property` and `particles_base.set_property` example

The `particles_base.apply` Operation

The `particles_base.apply` operation enables outlining `particles_base.foreach` and `particles_base.for_all_neighbors` operations from `particles_base.loop` regions into standalone `func.func` functions. Simply speaking, it is a special `func.call` operation that is always inlined. To adhere to the restrictions imposed on operations acting on `!particles_base.set` values, detailed in Section 4.2.2, it always takes one `!particles_base.set` value as input and returns one `!particles_base.set` value as output.

Example:

```
// Function definition in builtin.module:
func.func @evolve(%set : !baseset, %deltat : f64) -> !baseset {
  // Initialize constants and such:
  %zero = arith.constant 0 : index
  // Single use of %set argument:
  %evolved_set = "particles_base.foreach"(%set) ({
    // Evolve particle set
  }) : (!baseset) -> !baseset
  // Immediately return result:
  func.return %evolved_set : !baseset
}

// Inside of particles_base.loop operation:
%evolved_set = "particles_base.apply" (%set, %deltat) <{"func" = @evolve}>
                                      : (!baseset, f64) -> !baseset
```

Listing 4.11: `particles_base.apply` example

Restrictions on the Referenced Function To satisfy the constraints imposed on the `particles_base.apply` operation, the referenced `func.func` operation must have a body and must accept a `!particles_base.set` value as the first argument, with all other arguments being of other types. The function must also return a single `!particles_base.set` type. The region argument representing the `!particles_base.set` function argument must have a single use, by either a `particles_base.foreach` or `particles_base.for_all_neighbors` operation, the result of which must be immediately returned. Other operations are permitted before the use of the `!particles_base.set` argument. For example, to initialize constants.

The `particles_base.call` Operation

The `particles_base.call` operation enables calling `func.func` functions that accept a `!particles_base.storage` value as the first argument from within `particles_base.loop` regions. As previously explained, one of the purposes of the `particles_base.loop` operation is to switch from the memory semantics employed by the `!particles_base.storage` type on the outside to the fake value semantics employed by the `!particles_base.set` type on the inside. To call a function that accepts a `!particles_base.storage` as the first argument, the `particles_base.call` operation reverses this switch by storing its `!particles_base.set` operand back into the `!particles_base.storage` it was originally loaded from and passing the storage (which is essentially a pointer) to the referenced function. After the function returns, the `!particles_base.set` is reloaded from the same storage and returned. During this process, the buffered values and transient state associated with the particle set are preserved.

Example:

```
// External function declaration in builtin.module:
func.func private @batch_done(%storage : !basestorage,
                              %context : !llvm.ptr,
                              %step : index) -> ()

// Inside of particles_base.loop region:
%result = "particles_base.call" (%set, %context, %step) <{"func" = @batch_done}>
                                : (!baseset, !llvm.ptr, index) -> !baseset
```

Listing 4.12: `particles_base.call` example

Integration with C++ Section 3.2 identifies integration with C++ as a key semi-functional requirement. The `!particles_base.storage` type already facilitates this by excluding all buffers and transient state and lowering to a simple pointer that can be passed to and from C++ functions. The `particles_base.call` operation further assists this integration by enabling calls to C++ functions from within `particles_base.loop` regions. This is important for enabling features like checkpointing and terminal output during time stepping, which currently must be implemented in C++.

4.4 Subdialects of `particles_base`

4.4.1 The `particles` Dialect

The `particles` dialect is a subdialect of `particles_base`, inheriting all its attribute, type, and operation definitions. It does not extend any of the IR constructs, but simply adopts them as they are, effectively only replacing the `particles_base` prefix with `particles`. Unlike the `particles_base` dialect, however, it is not abstract and does appear in the IR.

4.4.2 The `particles_dist` Dialect

The `particles_dist` dialect specializes the `particles` dialect to DMP and OpenFPM. Like `particles`, it is also a subdialect of `particles_base`. However, unlike `particles`, it extends many of the inherited operation and type definitions while also introducing new operations and an attribute. In this section, only the extensions to the inherited types and operations and the newly introduced attribute are discussed. The new operations are separated into two parts and introduced during later phases. DMP-specific operations are presented in Phase 2 (Chapter 5), whereas internal operations, which include those designed to integrate OpenFPM, are discussed in Phase 3 (Chapter 7). The `particles_dist` dialect also introduces the concept of neighbor lists. However, instead of integrating all neighbor lists directly, they are encapsulated within their own companion dialects, which are also introduced in Chapter 6. Whereas the `particles` dialect did not make any assumptions about the execution model, `particles_dist` assumes a Single Program, Multiple Data (SPMD) execution model.

The !particles_dist.set Type

The !particles_dist.set type represents a loaded distributed particle set. Similarly to a vector_dist instance from OpenFPM, which it essentially wraps, each !particles_dist.set value maintains only the particles within a subdomain, including ghost particles. In addition to wrapping a vector_dist instance, each !particles_dist.set value may also wrap a set of neighbor lists. To this end, the !particles_dist.set type extends !particles_base.set by introducing a fourth parameter, neighbor_lists, which specifies the list of neighbor list types associated with the particle set. This enables !particles_dist.set types to be linked to multiple neighbor lists, such as two cell lists with different cutoff radii. Because the subdomain sizes in a distributed particle set always fluctuate, the size parameter must always be -1.

Example:

```
!distset = !particles_dist.set<
  -1 : index,                  // size (must always be -1)
  f64,                         // position_type
  3 : index,                   // dimensionality
  {                            // properties
    "velocity" = vector<3xf64>,
    "force" = vector<3xf64>
  },
  [                            // neighbor_lists
    !cell_list.list<f64, 3 : index, 0.3 : f64>
  ]
>
```

Listing 4.13: !particles_dist.set example (!distset)

The particles_dist.for_all_neighbors Operation

The particles_dist.for_all_neighbors operation extends particles_base.for_all_neighbors by adding DMP-specific properties, as well as two additional optional regions.

Example:

```
%interacted = "particles_dist.for_all_neighbors" (%set)
<{
  "with_self" = false,
  "reduction_kinds" = [#particles_dist.reduction_kind<add>],
  "write_targets" = ["force"],
  "neighbor_list_kind" = #particles_dist.neighbor_list_kind<cell_list>, // NEW
  "max_distance" =  0.3 : f64                                           // NEW
}>
({^1(%particle : !distparticle):                          // NEW: pre-interaction region
  >> Modify particle before interactions: %particle -> %updated_particle
  "particles_dist.yield" (%updated_particle) : (!distparticle) -> ()
},
{^2(%particle : !distparticle):                           // NEW: post-interaction region
  >> Modify particle after interactions: %particle -> %updated_particle
  "particles_dist.yield" (%updated_particle) : (!distparticle) -> ()
},
{^3(%particle : !distparticle, %neighbor : !distparticle): // interaction region
  >> Calculate the %force that %neighbor exhibits on %particle
  "particles_dist.yield" (%force) : (vector<3xf64>) -> ()
}) : (!distset) -> (!distset)
```

Listing 4.14: particles_dist.for_all_neighbors example

Additional Properties The properties introduced by the `particles_dist.for_all_neighbors` operation specify which neighbor list to use to iterate over the indices of a particle's neighbors, and must therefore match one of the neighbor list types associated with the `!particles_dist.set` type it operates on:

- **`neighbor_list_kind`**: A `#particles_dist.neighbor_list_kind` attribute that specifies the kind of neighbor list to use. The `#particles_dist.neighbor_list_kind` enum attribute currently supports two values, `cell_list` and `local_domain`, which correspond to the neighbor list dialects of the same name.
- **`max_distance`**: A float attribute that specifies the maximum distance between two neighbors. This property is only required if `neighbor_list_kind` is set to `cell_list` and specifies the cutoff radius/cell size used for the cell list.

Additional Regions The `particles_dist.for_all_neighbors` operation introduces two additional regions. The pre-interaction region is executed once for each particle before it interacts with its neighbors. The post-interaction region is executed once for each particle after it has interacted with its neighbors. Both regions are optional and have been introduced to enable fusing `particles_dist.for_all_neighbors` and `particles_dist.foreach` operations, as detailed later in Section 5.1.2. As is the case with its `particles.for_all_neighbors` counterpart, data dependencies between particles are illegal, as these would lead to data races. To avoid such situations, the `particles_dist.for_all_neighbors` operation also verifies that the pre- and post-interaction regions do not introduce any inter-particle data dependencies.

The `particles_dist.call` Operation

The `particles_dist.call` operation extends `particles_base.call` by introducing additional properties to enable staleness analysis and staleness tracking. The details regarding these mechanisms and the associated data structures are explained later in Section 5.2.3

Example:

```
%result = "particles_dist.call" (%set, %context, %step)
<{
  "func" = @batch_done,
  "requires_up_to_date" = ["map"],  // NEW
  "updates" = ["ghost(pos)"],       // NEW
  "makes_stale" = []                // NEW
}>
: (!distset, !llvm.ptr, index) -> !distset
```

Listing 4.15: `particles_dist.call` example

Additional Properties The `particles_dist.call` operation introduces four additional properties:

- **`requires_up_to_date`**: An optional array attribute specifying the data structures that must be up-to-date before the function is called.
- **`updates`**: An optional array attribute specifying the data structures that are updated by the called function.

- **makes_stale**: An optional array attribute specifying the data structures that are made stale during the execution of the function.
- **uses_staleness_flags**: An optional unit attribute indicating that the called function dynamically tracks the staleness of the data structures.

A particles_dist.call operation must either provide the first three properties to enable static staleness analysis or must have the fourth, uses_staleness_flags, to indicate that it makes use of dynamic staleness tracking.

4.5 Specialization of particles to particles_dist

The first step in the MLIR lowering pipeline is specializing the particles dialect to particles_dist. Two passes make up this step: a preparatory pass, particles-inline-apply-funcs, and the actual specialization pass particles-dist-specialize-particles.

4.5.1 The particles-inline-apply-funcs Pass

The particles-inline-apply-funcs pass inlines the functions referenced by particles.apply operations. Because of the constraints each particles.apply operation imposes on its function, inlining the function's body preserves the single-source single-sink modification graph of the particles.loop region it is inserted in.

4.5.2 The particles-dist-specialize-particles Pass

The particles-dist-specialize-particles pass fully replaces all particles attributes, types, and operations in the IR with their particles_dist counterparts. This specialization step is mostly straightforward, since most particles_dist IR constructs do not extend their particles counterparts in any way. However, the operations particles_dist.call and particles_dist.for_all_neighbors require additional properties and the !particles_dist.set type requires an additional parameter.

This presents a Catch-22. On one hand, the particles dialect deliberately avoids incorporating any target-specific components. On the other hand, the particles_dist dialect requires target-specific information, and because it is not targeted directly, but only emitted through specialization, this information must already be present in the IR. To solve this Catch-22, the attribute dictionaries of the affected operations are used. As described in Section 2.4, the attribute dictionary of an operation may contain extrinsic data, which is not verified by the operation. By storing target-specific properties required by particles_dist operations in the attribute dictionaries of their particles counterparts, these entries can simply be promoted to properties during specialization. For instance, for the specialization to succeed, each particles.for_all_neighbors operation must include the neighbor_list_kind and, if required, the max_distance attribute within their attribute dictionaries.

Example:

```
%interacted = "particles.for_all_neighbors" (%set)
<{ // Property dictionary
  "with_self" = false,
  "reduction_kinds" = [#particles.reduction_kind<add>],
  "write_targets" = ["force"]
}>
({^0(%particle : !particle, %neighbor : !particle):
  >> Calculate the %force that %neighbor exhibits on %particle
  "particles.yield" (%force) : (vector<3xf64>) -> ()
})
{ // Attribute dictionary holding target-specific entries:
  "neighbor_list_kind" = #particles_dist.neighbor_list_kind<cell_list>,
  "max_distance" = 0.3 : f64
} : (!set) -> (!set)
```

is transformed into:

```
%interacted = "particles_dist.for_all_neighbors" (%set)
<{ // Property dictionary (attributes have become properties):
  "with_self" = false,
  "reduction_kinds" = [#particles_dist.reduction_kind<add>],
  "write_targets" = ["force"],
  "neighbor_list_kind" = #particles_dist.neighbor_list_kind<cell_list>,
  "max_distance" = 0.3 : f64
}>
({^0(%particle : !distparticle, %neighbor : !distparticle):
  >> Calculate the %force that %neighbor exhibits on %particle
  "particles_dist.yield" (%force) : (vector<3xf64>) -> ()
}) : (!distset) -> (!distset)
```

Listing 4.16: Specialization of `particles.for_all_neighbors` via `particles-dist-specialize-particles`

As outlined in Section 4.4.2, the `!particles_dist.set` type extends its `particles` counterpart by adding a `neighbor_lists` parameter. Unlike operation properties, this parameter cannot be provided via attribute dictionaries, as types do not have these. However, in this case it also is not necessary. Instead of requiring that this parameter is somehow provided, it is derived from the `neighbor_list_kind` and `max_distance` attributes found on all `particles.for_all_neighbors` operations in the IR.

Example:

```
!particles_base.set<
  -1 : index,
  f64,
  3 : index,
  {
    "velocity" = vector<3xf64>,
    "force" = vector<3xf64>
  }
>
```

is transformed into:

```
!particles_dist.set<
  -1 : index,
  f64,
  3 : index,
  {
    "velocity" = vector<3xf64>,
    "force" = vector<3xf64>
  },
  [
    // All particles.for_all_neighbors operations used
    // "neighbor_list_kind" = #particles_dist.neighbor_list_kind<cell_list>
    // and "max_distance" = "max_distance" = 0.3 : f64
    !cell_list.list<f64, 3 : index, 0.3 : f64>
  ]
>
```

Listing 4.17: Specialization of `!particles.set` via `particles-dist-specialize-particles`

5 Phase 2: Target-Specific Transformations

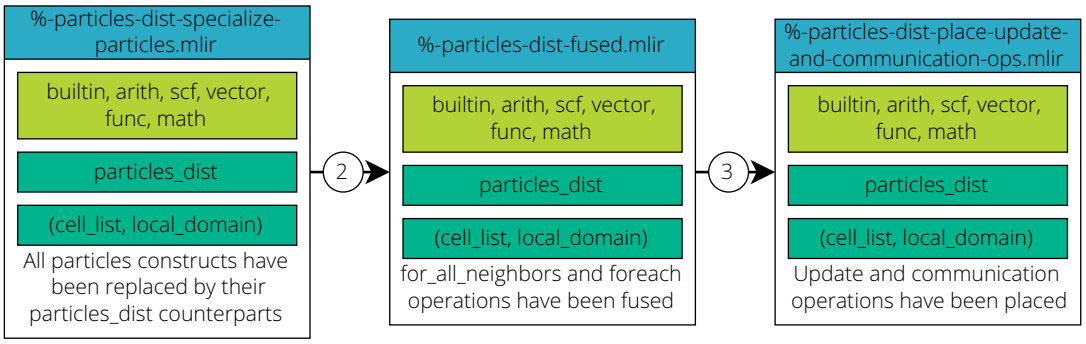

Figure 5.1: Phase 2 uses the output of phase 1 as input and has two steps. The first step fuses mergeable `particles_dist.foreach` and `particles_dist.for_all_neighbors` operations. The second step places update and communication operations.

Phase 2 is dedicated to target-specific transformations applied to the operations defined by the `particles_dist` dialect. As depicted in Figure 5.1, it is split into two steps. Each section of this chapter discusses one of these steps. The first section describes the fusion of `particles_dist.for_all_neighbors` and `particles_dist.foreach` operations. Section 5.2 introduces the DMP-specific operations defined by the `particles_dist` dialect, which includes update and communication operations, and explains the analysis techniques and placement strategies used to insert these operations into the IR.

5.1 Fusion of `particles_dist.for_all_neighbors` and `particles_dist.foreach` Operations

Among the operations defined by the `particles_dist` dialect, the `particles_dist.foreach` and `particles_dist.for_all_neighbors` operations are of particular interest for optimization, as most computation are performed within these two. Two optimization passes targeting these operations have been developed: The `particles-dist-fuse-foreach-ops`

pass, which fuses adjacent `particles_dist.foreach` operations, and the `particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops` pass, which fuses `particles_dist.foreach` and `particles_dist.for_all_neighbors` operations.

Optimizations are applied after specialization for two reasons. First, the choice of optimizations and their aggressiveness heavily depends on the lowering target. Second, the fusion of `particles_dist.foreach` and `particles_dist.for_all_neighbors` operations is made possible by the pre- and post-interaction regions introduced by the `particles_dist.for_all_neighbors` operation. The `particles.for_all_neighbors` operation does not have these regions, as it avoids making assumption about the strategy employed to visit neighboring pairs. For example, all neighboring pairs could be visited in parallel, in which case pre-/post-interaction regions would not make any sense.

In the case of the `particles_dist.for_all_neighbors` operation, it is known how it will be lowered and, consequently, also how neighboring pairs will be visited. When targeting CUDA GPUs, all individual particles will be visited in parallel, while for each particle its neighbors will be visited in sequence. In the case of CPUs, all individual particles will be visited sequentially, as will the neighbors for each particle. The visiting strategies for the individual particles match those of the `particles_dist.foreach` operation, which also visits all particles in parallel for CUDA GPUs and sequentially for CPUs. This is even made explicit by lowering each `particles_dist.for_all_neighbors` operation to a `particles_dist.foreach` operation with a region that iterates over all neighbor indices and computes the interaction results. Within this region, the pre-interaction region is simply inlined before iterating over all neighbor indices, while the post-interaction region is inlined afterward.

5.1.1 The `particles-dist-fuse-foreach-ops` Pass

The `particles-dist-fuse-foreach-ops` pass aggressively fuses `particles_dist.foreach` operations. Since each operation avoids data dependencies between executions of its region, two consecutive `particles_dist.foreach` operations can always be fused without creating any new data dependencies. The pass has three levels of aggression, with higher levels allowing for more code duplication:

- `aggression>=1`: Directly consecutive `particles_dist.foreach` operations are fused, which does not result in any code duplication.
- `aggression>=2`: `particles_dist.foreach` operations are copied/moved into the branches of directly adjacent `scf.if` operations if this enables further fusion. As visualized by Figure 5.2, this introduces code duplication, since each `foreach` operation that is moved must be copied/moved into both branches. Both upwards and downwards directions are supported:
  - Upwards: `foreach` operations directly after `scf.if` operations are copied/moved to the end of the `scf.if` regions.
  - Downwards: `foreach` operations directly before `scf.if` operations are copied/moved to the beginning of the `scf.if` regions.
- `aggression>=3`: `particles_dist.foreach` operations are moved from the end of `particles_dist.loop` time-stepping regions to the beginning if this enables further fusion. As illustrated by Figure 5.3, this requires substantial rewriting and introduces a significant amount of code duplication. Moving an operation from the end of the time-stepping region to the start does not alter the behavior of the intermediate iterations, but

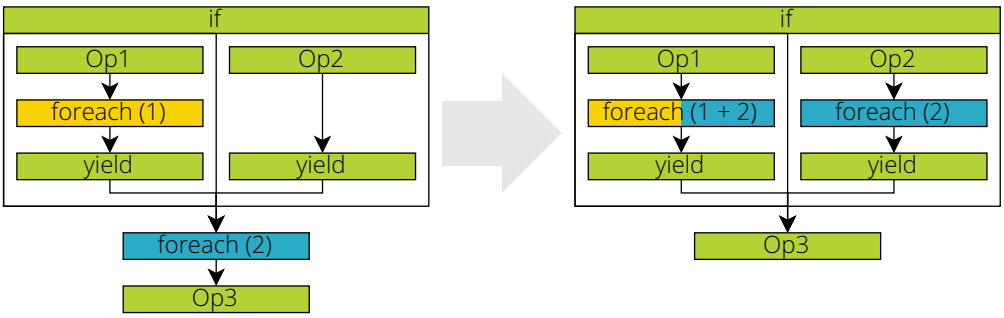

Figure 5.2: particles-dist-fuse-foreach-ops with aggression level 2

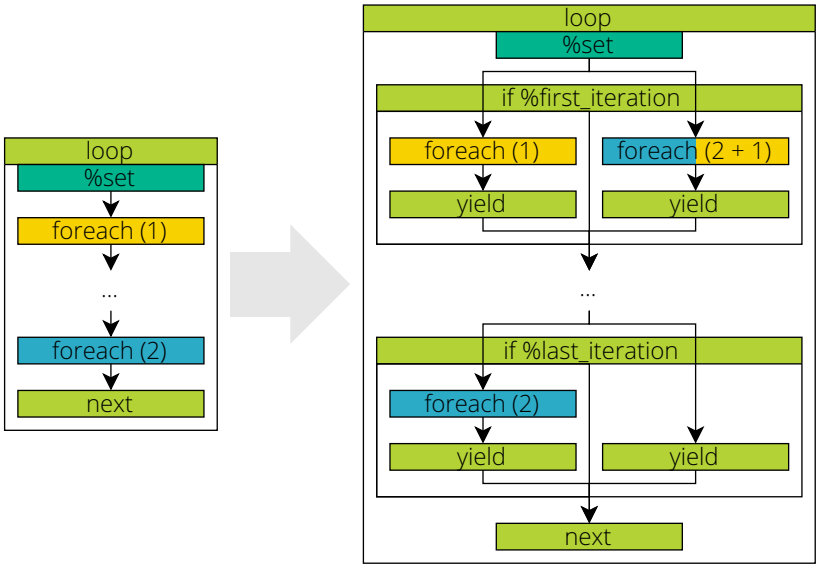

Figure 5.3: particles-dist-fuse-foreach-ops with aggression level 3

it does affect the first and last iterations, for which the original behavior must be preserved. Achieving this requires the introduction of an scf.if operation at the beginning to ensure the first iteration behaves as if the operation had not been moved. Likewise, during the final iteration, the moved particles_dist.foreach operation must be executed as if it has not been moved to the front, requiring the introduction of an scf.if operation at the end of the region.

5.1.2 The particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops Pass

The particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops pass aggressively fuses particles_dist.foreach operations with the pre- and post-interaction regions of adjacent particles_dist.for_all_neighbors. Unlike two consecutive foreach operations, consecutive for_all_neighbors and foreach operations cannot always be fused. As previously explained, the particles_dist.for_all_neighbors operation visits all particles in parallel on CUDA GPUs and sequentially on CPUs, while always iterating over each particle's neighbors sequentially. As usual, data dependencies between particles must be avoided, as these would lead to data races. This means, that the pre- and post-interaction regions cannot modify any field that is accessed from the neighbor in the interaction region. For

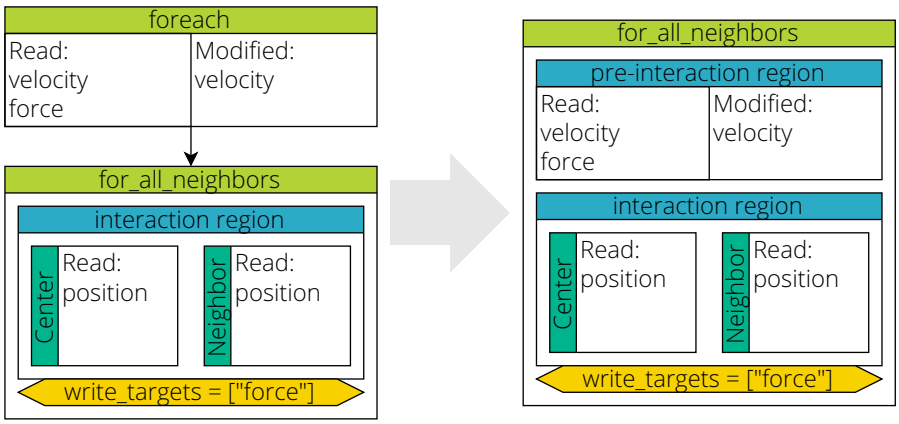

Figure 5.4: Fusing `particles_dist.for_all_neighbors` with `particles_dist.foreach`

example, if the position of a particle is modified in the pre-interaction region and read from the neighbors in the interaction region, the position of each neighbor depends on whether it has already been visited (and had its pre-interaction region executed) or not. Data dependencies within a particle are allowed, meaning that the pre- and post-interaction regions can modify fields that are read from the center particle in the interaction region and vice versa. Figure 5.4 illustrates an example where fusion does not lead to any illegal data dependencies, whereas Figure 5.5 shows a scenario where fusion would lead to illegal data dependencies.

Like the `particles-dist-fuse-foreach-ops` pass, the `particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops` pass also has three levels of aggression, controlling the amount of code duplication allowed during fusion:

- `aggression>=1`: Directly consecutive operations are fused, introducing no code duplication.
- `aggression>=2`: `particles_dist.for_all_neighbors` operations are moved into the branches of `scf.if` operations if this enables further fusion. Again, both upwards and downwards directions are supported.
- `aggression>=3`: `particles_dist.for_all_neighbors` operations are moved from the end of the time-stepping region of `particles_dist.loop` operations to the front if this enables further fusion.

Currently, only `for_all_neighbors` operations are moved. Future versions of this pass may also move `foreach` operations.

5.2 Enabling Distributed Memory Parallelism

5.2.1 `particles_dist` Update and Communication Operations

To enable DMP, the `particles_dist` dialect introduces dedicated operations for inter-process communication and for updating process-local data structures. Every update and communication operation has two variants: a normal variant and a "maybe" variant. Figure 5.6 illustrates the reason for this: control flow branches can lead to situations where it is not possible to make optimal update or communication decisions based purely on compile-time information. To address this, dynamic staleness tracking was introduced, which uses runtime

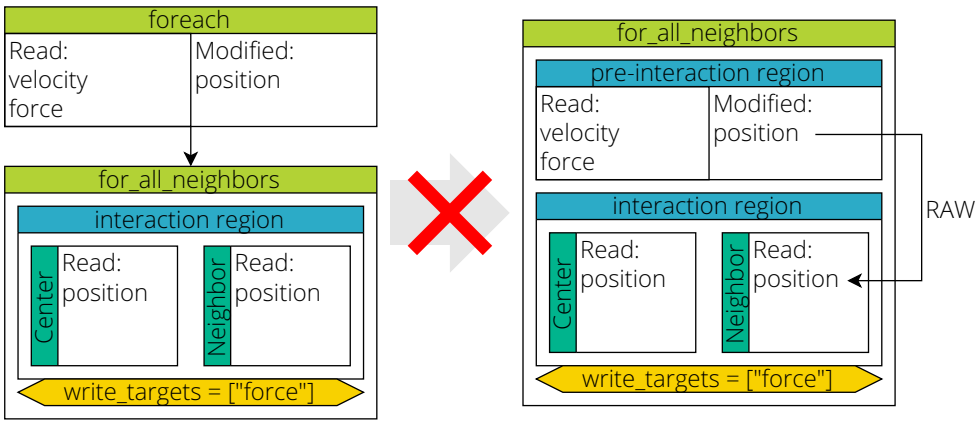

Figure 5.5: Illegal data dependency between pre-interaction region and interaction region in `particles_dist.for_all_neighbors`.

flags to track the staleness of the domain decomposition (abbreviated as "map"), each field in the ghosts layers, and each neighbor list. The mentioned "maybe" wrappers integrate these staleness flags. Each of these wrappers executes the wrapped operation only if the corresponding staleness flags are true.

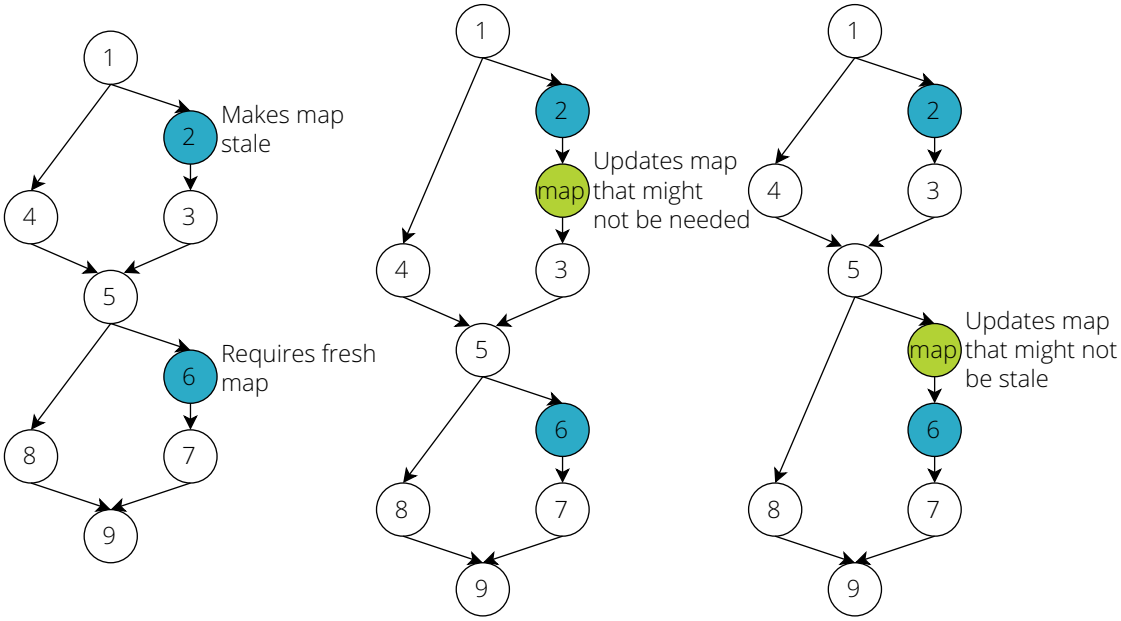

Figure 5.6: Need for staleness tracking

The `particles_dist.map` and `particles_dist.maybe_map` Operations

The `particles_dist.map` operation always updates the domain decomposition and corresponds to the `map` function of the `vector_dist` class from OpenFPM. The `particles_dist.maybe_map` operation only does so if the runtime flag tracking the staleness of the domain decomposition is set to true. Regardless of whether the domain decomposition is updated, it is guaranteed to be fresh afterward.

Example:

```
// Always recomputes the domain decomposition:
%updated_set = "particles_dist.map"(%set) : (!distset) -> !distset

// Recomputes the domain decomposition if the corresponding staleness flag is set to true:
%updated_set = "particles_dist.maybe_map"(%set) : (!distset) -> !distset
```

Listing 5.1: `particles_dist.map` and `particles_dist.maybe_map` example

The `particles_dist.ghost_get` and `particles_dist.maybe_ghost_get` Operations

The `particles_dist.ghost_get` operation populates the ghost layers of the given particle set for the specified fields. It corresponds to the `vector_dist.ghost_get` function. The fields to retrieve are listed in the `fields_to_get` operation property. Each entry of this array attribute must be either `"pos"` or the name of a particle property. The buffers holding the ghost values are deleted whenever new values are retrieved, so fetching a new set of fields makes all other fields unavailable. The `particles_dist.maybe_ghost_get` operation adds the operation property `fields_to_check` to specify the fields to check for staleness. It populates the ghost layers for all fields in `fields_to_get` if at least one staleness flag for a field in `fields_to_check` is set to true. Section 5.2.6 explains the rationale for this distinction.
Example:

```
// Retrieves the positions and the velocities of the ghost particles:
%updated_set = "particles_dist.ghost_get"(%set)
                        <{"fields_to_get" = ["pos", "velocity"]}> : (!distset) -> !distset

// Retrieves the positions and the velocities of the ghost particles
// if the positions are not up-to-date:
%updated_set = "particles_dist.maybe_ghost_get"(%set)
                       <{"fields_to_check" = ["pos"],
                         "fields_to_get" = ["pos", "velocity"] }> : (!distset) -> !distset
```

Listing 5.2: `particles_dist.ghost_get` and `particles_dist.maybe_ghost_get` example

The `particles_dist.update_neighbor_list` and `particles_dist.maybe_update_neighbor_list` Operations

The `particles_dist.update_neighbor_list` operation updates the specified neighbor list associated with the given particle set. In the case of cell lists, it corresponds to the `vector_dist.update_cell_list` function. Which neighbor list to update is specified via the `index` property. The index refers to the position of the neighbor list in the `neighbor_lists` parameter of the `!particles_dist.set` type. The `particles_dist.maybe_update_neighbor_list` operation performs the update only if the staleness flag associated with the neighbor list is set to true. In either case, the specified neighbor list is guaranteed to be up-to-date afterwards.

Example:

```
// Updates the neighbor list at the first position
// of the neighbor_list parameter of !distset:
%updated_set = "particles_dist.update_neighbor_list"(%set) <{"index" = 0 : index}>
                                                    : (!distset) -> !distset

// Updates the neighbor list if its associated staleness flag is set to :
%updated_set = "particles_dist.maybe_update_neighbor_list"(%set)
                                          <{"index" = 0 : index}> : (!distset) -> !distset
```

Listing 5.3: `particles_dist.update_neighbor_list` and `particles_dist.maybe_update_neighbor_list` example

5.2.2 Placement Strategy

Between the data structures, managed by the operations introduced in the previous section, intricate freshness requirements and staleness implications exist that dictate the order in which they must appear in the IR and, consequently, the order in which their placement passes must be executed.

Freshness Requirements

- The `particles_dist.ghost_get` operation requires an up-to-date domain decomposition to retrieve the fields for the correct ghost particles, as the domain decomposition affects which particles are in the ghost layers.
- The `particles_dist.update_neighbor_list` operation requires that the position values of the ghost particles are present and up-to-date, as these particles are included when updating a neighbor list. It therefore also indirectly requires a fresh domain decomposition.
- The `particles_dist.map` operation requires no other data structures to be up-to-date, although this may change in the future when symmetric interactions and ghost-put functionality are introduced.
- Any operation accessing the fields of ghost particles requires that the ghost layers are fresh for these fields. Transitively, it also requires a fresh domain decomposition.
- Any operation requiring an up-to-date neighbor list also requires that the domain decomposition is fresh.

Staleness Implications

- Any operation that modifies particle properties makes the ghost layers stale for these properties.
- Any operation that modifies particle positions makes the domain decomposition, and therefore also all other data structures, stale.
- Regarding ghost layers, OpenFPM introduces additional staleness implication by deleting all buffered ghost values every time new fields are retrieved. As a consequence, updating the ghost layers for a set of fields makes it stale for all other fields.

Placement Strategy From the freshness requirements and staleness implications, the following three-step placement strategy was derived:

1. Place `particles_dist.update_neighbor_list` operations in front of operations that require a fresh neighbor list, but not in front of those where it is already fresh or expected to be fresh.
2. Insert `particles_dist.ghost_get` operations in front of operations that depend on fresh values in the ghost layers, which includes `particles_dist.update_neighbor_list` operations. Again, do not place them in front of operations where the ghost layers are already fresh or expected to be fresh. Furthermore, to reduce the communication overhead, the number of `particles_dist.ghost_get` executions should be minimized by bundling the required ghost fields of multiple operations together.
3. Place `particles_dist.map` operations in front of operations that require an up-to-date domain decomposition – which includes `particles_dist.ghost_get` and `particles_dist.update_neighbor_list` operations – but not in front of those where it is already fresh or expected to be fresh.

A new variant of DFA, referred to as staleness analysis, was developed, which takes all these intricacies into account and supports the correct and optimized placement of all three operation types.

5.2.3 Staleness Analysis

Staleness analysis is a Data-Flow Analysis (DFA) variant that was specifically developed to determine for each node in a `!particles_dist.set` single-source single-sink modification graph which data structures associated with the particle set are stale. As detailed in Section 4.2.2, each single-source single-sink modification graph is also a CFG that focuses only on operations that act on particle sets. DFA has been used in the other works for identifying and eliminating stale data references and to generate SPMD code for distributed memory systems [42, 43].

Two variants of staleness analysis have been developed: potential and full. Potential staleness analysis determines for each node which data structures might be stale and works similar to the reaching definitions analysis. Rather than identifying which definitions may reach a node, however, it tracks which data structures might reach a node in a stale state. Full staleness analysis determines for each node which data structures are definitely stale and is comparable to the available expressions analysis.

Staleness States Both variants of staleness analysis utilize individual staleness states for each relevant data structure. Operations in the modification graph may kill or generates these states. An operation kills a state if it updates the data structure associated with the staleness state, and generates a state if it makes the data structure stale. For instance, a `particles_dist.foreach` operation that modifies particle positions generates the `map` state, which corresponds to the domain decomposition. Because all other data structures depend on a fresh domain decomposition, it also generates all other staleness states. Conversely, the `particles_dist.map` and `particles_dist.maybe_map` operations kill the `map` state, as the domain decomposition is guaranteed to be up-to-date afterward. The following staleness states have been defined:

- **`map`**: Refers to the domain decomposition.
- **`ghost(field)`**: Refers to the ghost layers for the given `field`, which must either be `pos` or the name of a particle property.

- `neighbor_list(index)`: Refers to the neighbor list with the specified `index`.

`ParticlesDistSetConsumerProducer` Interface For all variants of staleness analysis, it is essential to know for each operation that acts on `!particles_dist.set` values which staleness states it generates and which it kills. Furthermore, during placement, it is necessary to know for each operation the data structures it requires to be fresh. This can be generalized by asking for every operation and every data structure `X`, where `X` may refer to `map`, any `ghost(field)`, or any `neighbor_list(index)`:

- Does the operation definitely make `X` stale?
- Does the operation potentially make `X` stale?
- Does the operation definitely update `X`?
- Does the operation potentially update `X`?
- Does the operation require `X` to be up-to-date?

All of these questions are encapsulated in the Python interface `ParticlesDistSetConsumerProducer`, which extends the interface `ParticleSetConsumerProducer` introduced in Section 4.2.2. This interface must be implemented by all operations that act on `!particles_dist.set` values. From its functions, the GEN and KILL sets of each operation can be derived. For example, if an operation potentially makes the domain decomposition stale, it generates the `map` state during potential staleness analysis. However, for it to also generate the `map` state during full staleness analysis, it must definitely make the domain decomposition stale. Potential cases always include definitive ones: if an operation definitely makes `X` stale, it also qualifies as potentially making `X` stale.

Unique Challenges of the `particles_dist.call` Operation The `particles_dist.call` operation poses a unique challenge. For many operations, implementing the `ParticlesDistSetConsumerProducer` interface is straightforward and requires no analysis. The operations `particles_dist.foreach` and `particles_dist.for_all_neighbors` must first analyze their regions to determine which fields are read or modified, but this information is readily available in the IR. However, the `particles_dist.call` operation cannot analyze the functions it invokes, since these functions don't have to follow an analyzable structure or may be external and therefore inaccessible. As a result, it cannot derive any staleness information. Without staleness information, the worst case must be assumed to ensure the correctness of the program: everything is required fresh before the call and everything is stale after the function returns. While this approach may be acceptable for infrequently called functions, for those invoked every time step the overhead would be excessive, especially since the function may not even access or modify anything. To address this, the `particles_dist.call` operation extends its `particles.call` counterpart with four optional new operation properties: `requires_up_to_date`, `updates`, `makes_stale`, and `uses_staleness_flags` (See Listing 4.15). The properties `requires_up_to_date`, `updates`, and `makes_stale` must all be array attributes with string entries. The entries must follow the same naming scheme as the staleness states (e.g., `map`, `ghost(pos)`). The `uses_staleness_flags` property is a unit attribute, meaning it is either present or not. There are three valid combinations of these properties:

1. None are present: The worst-case scenario is assumed. All requires-up-to-date and makes-stale functions of the `ParticlesDistSetConsumerProducer` interface return true.

2. The three array properties `requires_up_to_date`, `updates`, and `makes_stale` are present: The values are interpreted as definite answers. For example, if `map` is listed under `makes_stale`, the function definitely makes the domain decomposition stale.
3. Only the `uses_staleness_flags` property is present: A reference is passed to the called function as the last argument. The reference points to a struct containing all boolean flags used for staleness tracking. This allows the function to read and update the staleness flags as needed. Consequently, the `particles_dist.call` operation returns true for all potential staleness and update queries, while all definite ones and all requires-up-to-date queries return false.

Differences between Initialization and Time-Stepping Regions The initialization and time-stepping regions of the `particles_dist.loop` operation must be treated differently. Determining which data structures are stale at the beginning of the initialization region is not possible. Consequently, all data structures are assumed to be stale. In DFA terminology, this means that the `!particles_dist.set` region argument generates all staleness states. By contrast, the `!particles_dist.set` argument of the interaction region generates no staleness states. Instead, the time-stepping region is analyzed under the assumption that no data structures are passed from the initialization region to the first iteration that are stale but are required to be fresh. To ensure this, liveness analysis is applied to the time-stepping region. Liveness analysis identifies, for each node in the modification graph, the data structures that are required to be fresh after the node, before they are either updated or made stale. Any data structure that is live at the entry node of the time-stepping region is required fresh by the `particles_dist.next` operation of the initialization region. As a final difference between both regions, to account for the loop structure of the time-stepping region, a back edge is added from the node representing the `particles_dist.next` operation (the sink) to the node for the `!particles_dist.set` region argument (the source).

Potential Staleness Analysis In potential staleness analysis, the IN set of a node contains the staleness states for all data structures that are potentially stale before the corresponding operation, while the OUT set contains the states for all data structures that are potentially stale afterward. To join the OUT sets of all predecessors, the union operator is used. This reflects that a data structure is potentially stale after two or more merging control flow branches if it is potentially stale at the end of at least one of them.

$$\text{IN}[n] = \bigcup_{p \in \text{pred}(n)} \text{OUT}[p], \quad \text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

with:

- KILL[**n**]: All staleness states whose associated data structures are definitely updated by node n. Additionally, all states whose data structures are required up-to-date by node n. Including these states reduces the importance of the visiting order during placement. For example, it prevents inserting a `particles_dist.map` operation between two operations that both require an up-to-date domain decomposition in the case where the second one is visited before the first. Unless, of course, the first operation makes **`map`** stale.

- GEN[**n**]: All staleness states associated with data structures that are potentially made stale by node n. If the node is the `!particles_dist.set` argument of the initialization region, all staleness states.

Full Staleness Analysis In full staleness analysis, the IN set of a node contains the staleness states for all data structures that are definitely stale before the corresponding operation, while the OUT set contains the states for all data structures that are definitely stale afterward. To join the OUT sets of all predecessors, the intersection operator is used. This reflects that a data structure is definitely stale after two or more merging control flow branches if it is definitely stale at the end of all of them.

$$\text{IN}[n] = \bigcap_{p \in \text{pred}(n)} \text{OUT}[p], \quad \text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

with:

- KILL[**n**]: All staleness states whose associated data structures are definitely updated or required to be up-to-date by node n. Additionally, all states whose data structures are potentially updated by the operation, since these are no longer definitely stale afterwards.
- GEN[**n**]: All staleness states associated with data structures that are definitely made stale by the operation. If the node is the `!particles_dist.set` argument of the initialization region, all staleness states.

Example Figure 5.7 illustrates both potential staleness analysis and full staleness analysis for the domain decomposition applied to the same graph. Together, both analysis results enable determining where the corresponding update operations must be placed. In the shown example, a `particles_dist.maybe_map` operation must be inserted in front of the `for_all_neighbors` operations, as the domain decomposition is required fresh at this node, is potentially stale, but not definitely so.

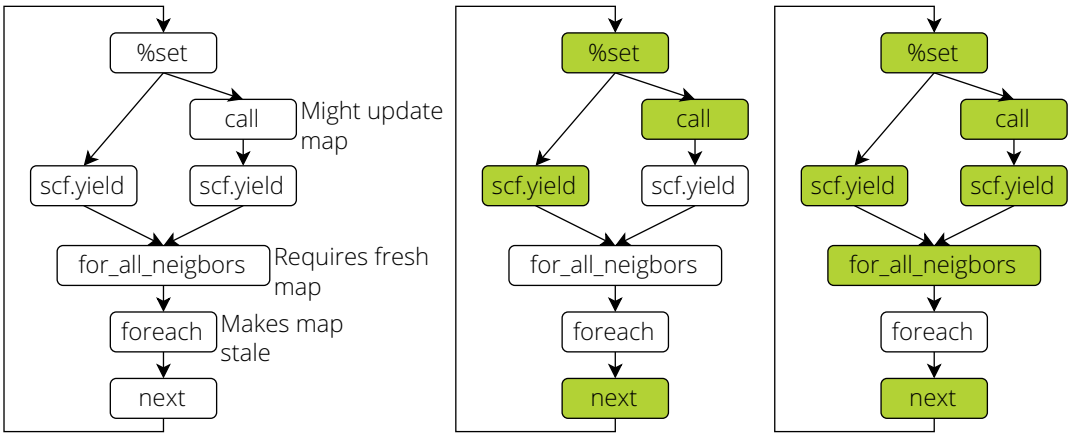

Figure 5.7: Left: Modification graph with staleness triggers and freshness requirements. Middle: Full staleness analysis for domain decomposition. Right: Potential staleness analysis for domain decomposition. Colored nodes have `map` in their IN set.

5.2.4 The `particles-dist-place-update-neighbor-list-ops` Pass

The `particles-dist-place-update-neighbor-list-ops` pass places `particles_dist.update_neighbor_list` and `particles_dist.maybe_update_neighbor_list` operations inside each `particles_dist.loop` region. To determine where to insert these operations, full and potential staleness analyses are utilized, exclusively tracking the staleness of neighbor lists. Algorithm 1 is used to place the operations in both the initialization and time-stepping regions. After the algorithm was executed on both regions, liveness analysis is applied to the time-stepping region to determine which neighbor lists need to be updated before being passed from the initialization region to the time-stepping region. As with the staleness analysis, this liveness analysis is limited to neighbor lists.

Algorithm 1 Placing `particles_dist.update_neighbor_list` and `particles_dist.maybe_update_neighbor_list` operations

Input: Region R

```
while first iteration or changes done in last iteration do
    perform potential staleness analysis on R
    perform full staleness analysis on R
    for all operations op ∈ R do
        if not op requires up-to-date neighbor lists then
            continue
        RF ← neighbor lists required fresh by op
        PS ← neighbor lists that are potentially stale at op
        DS ← neighbor lists that are definitely stale at op
        NU ← DS ∩ RF                          ▷ neighbor lists that need to be updated
        MU ← (PS ∩ RF) \ NU                   ▷ neighbor lists that may need to be updated
        if NU ≠ ∅ or MU ≠ ∅ then
            for all neighbor lists ∈ NU do
                place particles_dist.update_neighbor_list operation before op
            for all neighbor lists ∈ MU do
                place particles_dist.maybe_update_neighbor_list operation before
                op
            break
```

5.2.5 The `particles-dist-convert-maybe-ops-to-if-stale` Pass

The `particles-dist-convert-maybe-ops-to-if-stale` pass replaces all `particles_dist.maybe_X` operations with explicit if-stale-then-X constructs. To support this transformation, the `particles_dist` dialect defines three helper operations:

- `particles_dist.get_map_stale` Returns true if the domain decomposition is stale.
- `particles_dist.get_ghosts_stale` Returns true if the ghost layers are stale for at least one of the fields listed in its `fields` property.
- `particles_dist.get_neighbor_list_stale` Returns true if the neighbor list with the specified index is stale.

This pass must be executed after each placement pass so that the next placement pass can correctly insert its operations inside the generated `scf.if` regions. It must also be run after the final placement pass to lower the remaining `particles_dist.maybe_X` operations.

Example:

```
%updated_set = "particles_dist.maybe_ghost_get"(%set)
                                                <{"fields_to_check" = ["pos"],
                                                  "fields_to_get" = ["pos", "velocity"]}>
                                                : (!distset) -> !distset
```

is transformed into:

```
%set0, %pos_stale = "particles_dist.get_ghosts_stale"(%set)
                                                      <{"fields" = ["pos"]}>
                                                      : (!distset) -> (!distset, i1)

%updated_set = scf.if %pos_stale -> (!distset) {
  // Ghost layers stale for positions => retrieve ghosts:
  %updated_set0 = "particles_dist.ghost_get"(%set0)
                                            <{"fields_to_get" = ["pos", "velocity"]}>
                                            : (!distset) -> !distset
  scf.yield %updated_set0 : !distset
} else {
  scf.yield %set0 : !distset
}
```

Listing 5.4: Conversion of `particles_dist.maybe_ghost_get` via `particles-dist-convert-maybe-ops-to-if-stale`

5.2.6 The `particles-dist-place-ghost-get-ops` Pass

The `particles-dist-place-ghost-get-ops` pass places `particles_dist.ghost_get` and `particles_dist.maybe_ghost_get` operations inside each `particles_dist.loop` region. Like `particles-dist-place-update-neighbor-list-ops`, this pass also utilized staleness and liveness analysis to locate insertion points. The placement of `ghost_get` operations must follow the placement of `particles_dist.update_neighbor_list` operations, as those also require up-to-date ghost layers.

The Issue of Visiting Order Compared to placing `update_neighbor_list` operations, placing `ghost_get` operations involves additional challenges. `particles_dist.ghost_get` operations make the ghost layers stale for all fields not fetched, as all buffered ghost values are deleted when new ones are retrieved. Furthermore, to avoid redundant communication, the fields retrieved should be bundled together as much as possible. As a result, the visiting order of the operations in the time-stepping region during placement is critical. For example, Figure 5.8 illustrates a scenario where an operation requiring `ghost(force)` to be up-to-date is indirectly followed by another operation that requires a fresh `ghost(pos)`. If the operation at the end of the loop is visited first, both fields can be bundled into one `particles_dist.ghost_get` placed before it. If the operation at the beginning of the loop is visited first, a `particles_dist.ghost_get` is inserted before it, fetching `ghost(pos)`. When the other operation is visited later, a combined retrieval of `force` and `pos` is no longer possible.

Bundling Fields to Retrieve Using Liveness Analysis To determine which fields can be bundled together when populating the ghost layers, liveness analysis over sets of fields is employed. A set is live if every field it contains is live, and a field is live at a given node if its ghost values are used before they are refetched or made stale. Individual field liveness is not relevant, as operations accessing fields on ghost particles need all accessed fields

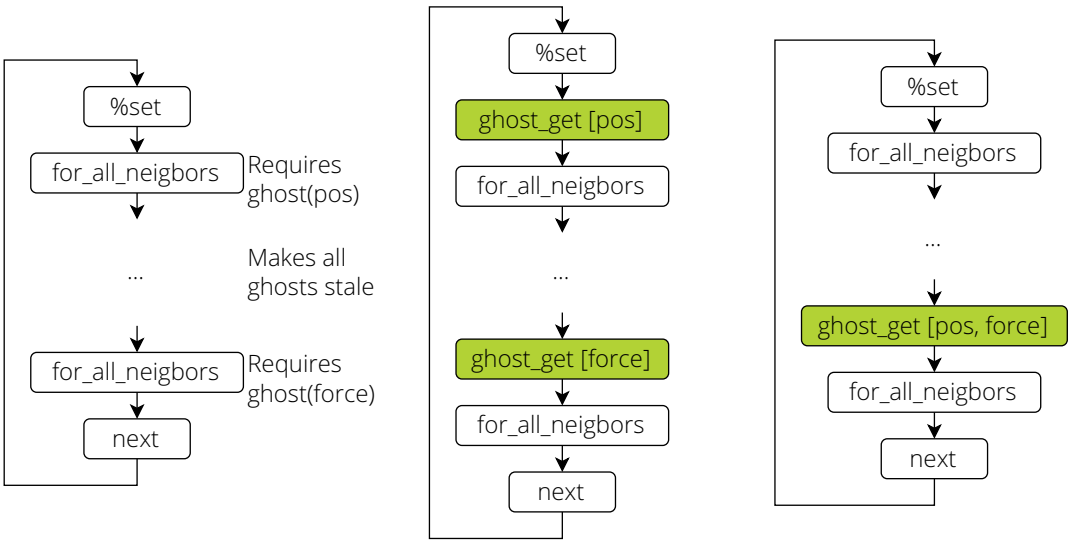

Figure 5.8: Issue of visiting order when placing `particles_dist.ghost_get` operations. Left: Modification graph before placement of `particles_dist.ghost_get` operations. Middle: Inserted `ghost_get` if the `particles_dist.for_all_neighbors` operation at the beginning of the loop was visited first. Right: Inserted `ghost_get` if the `for_all_neighbors` operation at the end of the loop was visited first.

up-to-date. As such, if any accessed field is missing or stale, all fields must be retrieved. To capture both definite and potential usage, two variants of liveness analysis – full and potential – were implemented, with the only difference between these two variants being their join function. Potential liveness analysis employs the union operation to reflect how a set of fields is potentially live before a control flow branch if it is potentially live at the beginning of at least one outgoing branch. Conversely, full liveness analysis uses the intersection operation to reflect how a set of fields is definitely live before a control flow branch if it is definitely live at the beginning of all outgoing branches.

Bundling Strategies The `particles-dist-place-ghost-get-ops` pass implements two bundling strategies, which are specified via the `strategy` pass option:

- **`optimistic`**: When a `particles_dist.ghost_get` or `particles_dist.maybe_ghost_get` operation is inserted, all fields in potentially live sets are also retrieved.
- **`balanced`**: All fields in definitely live sets are retrieved.

Mitigating the Issue of Visiting Order To mitigate the visiting-order issue, two adjustments were made to the placement algorithm, as seen in Algorithm 2. First, the traversal of the time-stepping region starts at a strategically chosen point and not at the region argument. An optimal starting point is after an operation that makes all fields in the ghost layers stale. If no such an operation exists, traversal starts after the operation that makes the most fields stale. When the end of the time-stepping region is reached during traversal, it continues at the beginning, until the starting point is encountered again. Because the initialization region is not executed in a loop and everything is assumed to be stale at the start, the region argument is always chosen as the starting point.

Second, only potential staleness analysis is utilized, and in cases where it is not certain whether any ghost values must be fetched, only `particles_dist.maybe_ghost_get` operations are placed. This approach prevents scenarios where a suboptimal `particles_dist.ghost_get` operation is inserted that would have been made redundant if its

predecessor had been visited first, as illustrated in Figure 5.8. By using only `maybe_ghost_get`, suboptimally placed operations are simply skipped during runtime. To facilitate this, the `particles_dist.maybe_ghost_get` operation has two properties: `fields_to_check`, which lists the ghost fields to check for staleness, and `fields_to_get`, which specifies the fields to retrieve if any checked field is stale. `fields_to_check` is used to specify the ghost fields that must be fresh after the operation, while `fields_to_get` also contains the fields that should be fetched alongside in anticipation of subsequent operations. After Algorithm 2 was executed on both regions, liveness analysis is applied to the time-stepping region to determine which ghost fields need to be retrieved at the end of the initialization region.

Algorithm 2 Placing `particles_dist.ghost_get` and `particles_dist.maybe_ghost_get` operations

Input: Region R, Bundling Strategy BS

```
while first iteration or changes done in last iteration do
    perform potential staleness analysis on R
    find a good starting point sp in R
    for all operations op ∈ R starting at sp do
        if not op requires fields in ghost layers then
            continue
        RF ← fields in ghost layers required fresh by op
        PS ← potentially stale fields in ghost layers at op
        MU ← PS ∩ RF                          ▷ fields that may need to be updated
        if MU ≠ ∅ then
            if BS = BALANCED then
                perform full liveness analysis on R
                LS ← definitely live sets before op
            else if BS = OPTIMISTIC then
                perform potential liveness analysis on R
                LS ← potentially live sets before op
            LF ← ⋃ LS                                          ▷ live fields
            if pred(op) makes any field in RF definitely stale or R = initialization region
            then
                place particles_dist.ghost_get updating LF before op
            else
                place particles_dist.maybe_ghost_get checking RF and updating LF
                before op
            break
```

5.2.7 The `particles-dist-place-map-ops` Pass

The `particles-dist-place-map-ops` pass places `particles_dist.map` and `particles_dist.maybe_map` operations inside each `particles_dist.loop` region. This pass is the simplest of all placement passes, as it only tracks the staleness of the domain decomposition. It must be executed after all `particles_dist.update_neighbor_list` and `particles_dist.ghost_get` operations have been placed, since these also require an up-to-date domain decomposition. Operations are inserted into both regions of the `particles_dist.loop` operation using Algorithm 3. As usual, after executing this algorithm on both regions, liveness analysis is used on the time-stepping region to place the necessary `particles_dist.map` operations at the end of the initialization region.

Algorithm 3 Placing `particles_dist.map` and `particles_dist.maybe_map` operations

Input: Region R

```
while first iteration or changes done in last iteration do
    perform potential staleness analysis on R
    perform full staleness analysis on R
    for all operations op ∈ R do
        if not op requires up-to-date domain decomposition then
            continue
        fs ← domain decomposition is fully stale at op
        ps ← domain decomposition is potentially stale at op
        if fs then
            place particles_dist.map operation before op
            break
        else if ps then
            place particles_dist.maybe_map operation before op
            break
```

5.2.8 The `particles-dist-place-set-stale-ops` Pass

The `particles-dist-place-set-stale-ops` pass inserts, if needed, `particles_dist.set_X_stale` operations inside `particles_dist.loop` regions after operations that alter the staleness state of a data structure. The `particles_dist` dialect defines three such operations:

- `particles_dist.set_map_stale` Sets the staleness flag for the domain decomposition to the provided value.
- `particles_dist.set_ghosts_stale` Sets the staleness flags for a specified list of ghost fields to the provided values.
- `particles_dist.set_neighbor_list_stale` Sets the staleness flag for the neighbor list with the specified index to the provided value.

Operations are only placed if necessary to avoid cluttering the IR. For example, placing `particles_dist.set_map_stale` operations inside the regions of a `particles_dist.loop` is only necessary if a `particles_dist.get_map_stale` operation is present in one of both regions, or if one of them contains a `particles_dist.call` operation with the `uses_staleness_flags` property.

Example:

```
%updated_set = "particles_dist.ghost_get"(%set)
                                          <{"fields_to_get" = ["pos", "velocity"]}>
                                          : (!distset) -> !distset
```

is transformed into:

```
updated_set0 = "particles_dist.ghost_get"(%set)
                                          <{"fields_to_get" = ["pos", "velocity"]}>
                                          : (!distset) -> !distset
updated_set = "particles_dist.set_ghosts_stale"(%set)
                                                <{"fields" = ["pos", "velocity", "force"],
                                                  "values" = [false, false, true]}>
                                                : (!distset) -> !distset
```

Listing 5.5: Placement of `particles_dist.set_ghosts_stale` operations via `particles-dist-place-set-stale-ops`

6 Interlude: Support Dialects and Integrating MLIR With OpenFPM

This interlude chapter introduces the main support dialects developed to facilitate the lowering of the `particles_dist` dialect. It also explains the conceptual framework for integrating MLIR with OpenFPM via a runtime.

The first section describes the challenges one has to face when lowering complex types, the concept of 1-to-N conversion, and the `box` dialect that was developed to tackle the former while facilitating the latter. Next, Section 6.2 explains how MLIR is integrated with OpenFPM and C++ via storage structs and runtime functions. Section 6.3 introduces the concept of companion dialects together with the implemented neighbor list dialects. Finally, Section 6.4 discusses how loads and stores of particle values are optimized using the `memwrap` dialect.

6.1 Enabling the Lowering of Complex Types and 1-to-N Conversion

Of the types introduced so far, the `!particles_dist.set` and `!particles_dist.particle` types each combine multiple elements of different types and must therefore be lowered to an aggregate type. The `!llvm.struct` type offered by the `llvm` dialect is the only suitable integrated candidate. However, this type is highly restrictive, as it only supports a limited selection of `builtin` and `llvm` types as elements. As a result, it cannot be directly applied in cases where element types are not natively supported, which includes `index` and `memref`, two very frequently used types.

This restriction can be circumvented by pre-lowering the unsupported types to supported ones, and using the lowered types instead of the original ones as elements. For example, as illustrated by Listing 6.1, the `index` type must be lowered to its corresponding signless integer type (e.g., `i32` or `i64`, depending on the platform), whereas the `memref` type must be lowered to its corresponding `!llvm.struct` type. When using this approach, extracting values of types that needed to be converted or inserting them into an `!llvm.struct` requires the use of `builtin.unrealized_conversion_cast` operations to convert between pre-lowered and original types, as exemplified by Listing 6.1. This approach is very tedious but works reasonably well for standard types where the conversion patterns are readily available. However, it falls short for more complex and non-standard types.

```
// Illegal struct type:
!llvm.struct<(
  i64,                                                  // Allowed
  index,                                                // Not allowed
  memref<3x?xf64>                                       // Not allowed
)>
// Corresponding legal struct type:
!mystruct = !llvm.struct<(
  i64,
  i64,                                                  // Pre-lowered index
  struct<(ptr,ptr,i64,array<2xi64>,array<2xi64>)>       // Pre-lowered memref
)>

// Extracting a value of a type that needed to be converted:
// Extract !llvm.struct<(ptr,ptr,i64,array<2xi64>,array<2xi64>)> entry:
%memref_struct = llvm.extractvalue %struct[2] : !mystruct
// Convert back to original type:
%memref = builtin.unrealized_conversion_cast %memref_struct
                : !llvm.struct<(ptr,ptr,i64,array<2xi64>,array<2xi64>)> to memref<3x?xf64>
```

Listing 6.1: Pre-lowering unsupported element types of an `!llvm.struct` type and extracting values of types that needed to be converted

To avoid pre-lowering unsupported element types and constantly converting between the pre-lowered and original types, the box dialect was developed. At the core of the box dialect lies the `!box.box` type, which functions similarly to a literal `!llvm.struct`. It also supports nesting and the extraction and insertion of elements. Unlike the `!llvm.struct` type, however, the `!box.box` type does not restrict the types of its elements and natively supports both `index` and `memref`. To store and load boxes, the box dialect also defines the `!box.storage` type, which is essentially a typed pointer to a region of memory for storing a `!box.box` value as `!llvm.struct`. When a `!box.box` value is stored as `!llvm.struct`, the `index` and `memref` types are automatically converted into their `!llvm.struct`-compatible types.

Beyond providing a generic struct-like type without the cumbersome restrictions of `!llvm.struct`, the box dialect introduces several rewriting passes that enable 1-to-N conversion by replacing all `!box.box` values and types with their constituent elements. The expansion and shortcutting strategy, and the passes implementing its steps, are discussed later in Section 8.2. 1-to-N conversion offers two advantages. First, it eliminates the need to explicitly lower every `!box.box` type, which effectively makes it compatible with all element types, even those that cannot be converted into `llvm`-compatible types. As explained, this applies only when a `!box.box` type is never loaded or stored, as in these cases, the box must be converted into an `!llvm.struct`. Second, operating on individual values is often preferable to operating on aggregate values, as many transformations, including many optimizations, are easier to implement for individual values. For example, constant folding becomes significantly more challenging when the constants are inserted into structs between their definition and subsequent use. This affects many of the passes integrated into MLIR and xDSL.

6.1.1 The box Dialect

The !box.box Type

The !box.box type is a struct-like aggregate type. It has a single dense array attribute parameter that holds the element types. Like !llvm.struct, it supports nesting.

Example:

```
!mybox0 = !box.box<[
  index,                        // Allowed, unlike in !llvm.struct
  !box.box<[                    // Nested box:
    memref<3x?xf64>             // Allowed, unlike in !llvm.struct
  ]>
]>
```

Listing 6.2: !box.box example (!mybox0)

The !box.storage Type

The !box.storage type, along with its associated allocation, load, and store operations, enables loading and storing of !box.box values. To store or load a !box.box value, it must be converted into an !llvm.struct. The ability to store boxes as structs is essential for enabling the exchange of complex data between MLIR and C++. The !box.storage type has a single parameter indicating the !box.box type it stores. When the !box.box type is lowered to an !llvm.struct during storage and transfer, index elements are automatically converted into unsigned integers, and memrefs into !llvm.structs.

Example:

```
// Stores !mybox0 value:
!box.storage<[!mybox0]>
// .. and corresponds to a !llvm.ptr to:
!llvm.struct<(
  i64,                                                  // Automatically converted index
  struct<(                                              // Nested box:
    struct<(ptr,ptr,i64,array<2xi64>,array<2xi64>)>     // Automatically converted memref
  )>
)>
```

Listing 6.3: !box.storage example

The box.insert and box.extract Operations

The box.insert operation inserts the provided value into the given !box.box argument at the position specified by the indices property and returns the updated box. The box.extract operation extracts the value at the position specified by the indices property from the given !box.box operand and returns it. For both operations, the indices property must be an integer array, with left-to-right values mapping to outer-to-inner indices. Inserting and extracting !box.box values is supported.

Example:

```
!mybox1 = !box.box<[              // indices:
           index,                 // 0
           !box.box<[             // 1
             f64,                 // 1,0
             f32                  // 1,1
         ]>]>
// Insert %value at 1,0 into given %box:
%updated_box0 = "box.insert"(%box, %value) <{"indices" = [1 : i64, 0 : i64]}>
                            : (!mybox1, f64) -> (!mybox1)
>> Modify %updated_box0 -> %updated_box1
// Extract updated value at 1,0:
%updated_value = "box.extract"(%updated_box1) <{"indices" = [1 : i64, 0 : i64]}>
                              : (!mybox1) -> (f64)
```

Listing 6.4: `box.insert` and `box.extract` example (`!mybox1`)

The `box.undef` Operation

The `box.undef` operation creates an undefined `!box.box` value of the specified return type. Before an entry can be extracted from the created box, it must be inserted. Furthermore, before a `!box.box` can be stored, all entries must be defined. This prevents the extraction and subsequent use of uninitialized entries, while also eliminating the need to implicitly initialize them.
Example:

```
%box = "box.undef"() : () -> !mybox1
```

Listing 6.5: `box.undef` example

The `box.load` and `box.store` Operations

The `box.load` operation loads the `!box.box` value stored at the location specified by the provided `!box.storage` operand and returns it. Internally, this operation first loads the `!llvm.struct` that corresponds to the `!box.box` type using `llvm.load` and then converts it into a `!box.box` value. The `box.store` operation stores the given `!box.box` operand at the location specified by the provided `!box.storage` argument. Under the hood, this operation first converts the `!box.box` value into an `!llvm.struct` and then stores the struct using `llvm.store`. The `!box.box` type must be convertible to `!llvm.struct` at the time of lowering any of these operations. During lowering, any `index` or `memref` element types are transparently converted into their `!llvm.struct`-compatible types.
Example:

```
!mystorage1 = !box.storage<!mybox>
// Load box from given %storage:
%box = "box.load"(%storage) : (!mystorage1) -> (!mybox1)
>> Modify %box -> %updated_box
// Store updated box:
"box.store"(%updated_box, %storage) : (!mybox, !mystorage1) -> ()
```

Listing 6.6: `box.load` and `box.store` example (`!mystorage1`)

The `box.alloca` Operation

The `box.alloca` operation dynamically allocates memory on the stack for the `!box.box` stored by the specified `!box.storage` return type. Internally, the `box.alloca` operation uses `llvm.alloca` to allocate memory for the corresponding `!llvm.struct`.

Example:

```
%storage = "box.alloca"() : () -> (!mystorage1)
```

Listing 6.7: `box.alloca` example

The `box.pack` and `box.unpack` Operations

The `box.pack` operation inserts all provided values into a new `!box.box` value of the specified return type and returns it. The types of the inserted values must match the element types of the returned `!box.box` type. Nested `!box.box` return types are supported, but their constituent values must be provided individually. The `box.unpack` operation recursively extracts all entries from the provided `!box.box` value and returns them.

Example:

```
// Extract all individual entries in the order in which they appear in !mybox1:
%0, %1_0, %1_1 = "box.unpack"(%box) : (!mybox1) -> (index, f64, f32)
// Provide values for indices (1), (1,0), and (1,1):
%new_box = "box.pack"(%0, %1_0, %1_1) : (index, f64, f32) -> !mybox1
```

Listing 6.8: `box.pack` and `box.unpack` example

6.2 Integrating MLIR With OpenFPM

The integration of MLIR with OpenFPM, and more generally C++, is essential to enable the use of OpenFPM data structures and procedures, which form the backbone of the `particles_dist` dialect, handling complex tasks such as domain decomposition, ghost layer synchronization, and cell list updates.

Exchanging Data and Control via Non-member Functions Integrating C++ with MLIR hinges on the transfer of data and control, which is most directly achieved using non-member functions. To avoid name mangling during the compilation of the C++ code, these functions are always declared `extern "C"` and in the global namespace, as shown by Listing 6.9. This ensures that C++ functions can be invoked from MLIR using their names, and vice versa.

Exchanging Data via Shared Types To use a C++ data structure directly within MLIR, it must have a corresponding MLIR type. At the time of writing, we are not aware of any native MLIR support for C++ classes or object instances, making their integration non-trivial and effort-intensive. Consequently, data exchanges between MLIR and C++ are largely restricted to numeric data types (`i32`, `i64`, `f32`, `f64`, etc.), structs (`!llvm.struct`), arrays (`!llvm.array`), and pointers (`!llvm.ptr`).

```
// C++
// Declare MLIR function:
extern "C" void md(ParticleSet *, size_t, size_t, double);
// Define C++ function:
extern "C" void batch_done(ParticleSet *set, size_t step) {
      // ...
}
```

```
// MLIR
// Declare C++ function:
func.func private @batch_done(%storage : !storage, %step : index) -> ()
// Define MLIR function:
func.func @md(%storage : !storage, %num_steps: index, %batch_size : index, %deltat: f64)
{
  // Perform calculations using particles.loop, call @batch_done after every batch
}
```

Listing 6.9: Matching function declaration in MLIR and C++

Passing Aggregate Types By Reference As outlined in Section 3.2, specific C++ optimizations prohibit passing aggregate types such as structs, arrays, or vectors by value. To circumvent this issue, such types are never passed by value between MLIR and C++, but are always passed by reference. While obtaining a reference to a struct or array in C++ is straightforward, this is not the case in MLIR. In MLIR structs, arrays, and vectors are SSA values, which are not directly associated with a memory region. Passing such values by reference therefore requires either dynamically allocating buffers for each function call or maintaining pre-allocated communication buffers for reuse, to store the value before passing a pointer to the buffer to the invoked function. This work uses the latter approach.

Integrating OpenFPM Data and Procedures via Runtime Functions As already discussed, OpenFPM classes and their instances cannot be used directly within MLIR because they lack the corresponding MLIR types. Since direct integration is not possible, indirect methods are employed to invoke member-functions and access object data. To invoke a member-function of an object from within MLIR, a reference (`!llvm.ptr`) to the object is passed to a non-member wrapper function implemented in C++, as illustrated in Listings 6.10. This function then internally invokes the desired member-function. Similarly, access to object data is also handled through non-member C++ functions that extract the data from the object reference, convert it into an MLIR compatible format, and return it. This approach reduces integration efforts to only those components of OpenFPM types that are needed directly within MLIR and allows choosing a format that compatible with both C++ and MLIR.

```
// C++ runtime
extern "C" void map(my_vector_dist *vd) {
  vd->map();
}
```

```
// MLIR
// Reference to vector_dist object is key component of a !particles_dist.set
%vector_dist_ptr = "particles_dist.get_vector_dist"(%set) : (!distset) -> !llvm.ptr
func.call @map(%vector_dist_ptr) : (!llvm.ptr) -> ()
```

Listing 6.10: Invoking `vector_dist.map` member-function from MLIR

Internals Structs and Extraction Functions Often, several individual data components from an OpenFPM object are needed for direct use within MLIR. To avoid defining numerous separate extraction functions, these components are grouped together into structs, which are then used to retrieve all components at once. For instance, the components required from a `vector_dist` object are:

- The local size, excluding ghost particles
- The local size, including ghost particles
- The `memref`s for the particle positions and particle properties

The first two values are needed to iterate over all particles, with or without ghosts, while the `memref` values are required to directly access their positions and properties within MLIR. Listing 6.11 shows the corresponding `ParticlesSetInternals` struct. To populate this struct, the C++ extraction function named `update_internals` is called from MLIR. This function takes a reference to the `vector_dist` object and a pointer to a `ParticlesSetInternals` struct as arguments, extracts all required data from the `vector_dist` object, converts it to MLIR compatible types, and writes it into the struct. In MLIR, once the extraction functions returns, the struct is loaded from the communication buffer and its values are extracted. As detailed in Section 4.2.1, the `!particles_dist.set` type employs fake value semantics, which enables buffering data as SSA values for direct use. The values extracted from the `ParticlesSetInternals` struct are buffered in this manner and are referred to as "internals", hence the name of the struct and the extraction function.

```
struct ParticleSetInternals {
  size_t size_local;
  size_t size_local_with_ghost;
  Memref<double, 2> pos;
  struct {
    MemRef<double, 2> velocity;
    MemRef<double, 2> force;
  } properties;
};
```

Listing 6.11: `ParticleSetInternals` struct example

with:

```
template <typename ElementType, size_t Rank> struct MemRef {
  ElementType *allocatedPtr;
  ElementType *alignedPtr;
  size_t offset;
  size_t sizes[Rank];
  size_t strides[Rank];
};
```

Listing 6.12: `MemRef` struct

The `particles_dist.update_internals` Operation Invoking the extraction function, loading the struct from the communication buffer, extracting the internals, and storing them in the `!particles_dist.set` value are all handled by the `particles_dist.update_internals` operation. It also updates the internals of all data structures associated with the particle set. For instance, if the particle set is associated with a cell list, its internals are updated using the same strategy.

Example:

```
// Before: Operations such as particles_dist.get_local_size may return stale values
// Updates internals of particle set and associated neighbor lists:
%set0 = "particles_dist.update_internals"(%set) : (!distset) -> !distset
// Afterward: Operations such as particles_dist.get_local_size return fresh values
```

Listing 6.13: `particles_dist.update_internals` example

Integrating Particle Data From `vector_dist` As discussed, `memref`s are used for directly accessing particle positions and particle properties within MLIR. To transfer a `memref` between MLIR and C++, it is converted into a struct containing the memory location and the layout parameters of the `memref`, as shown in Listing 6.12. The `memref` type is MLIR-native and not used by the `vector_dist` class, and thus cannot be directly extracted. However, both the `memref` type and the `vector_dist` template class are sufficiently flexible that their memory layouts can be matched. The memory layout of `memref`s is configurable using offsets, sizes, and strides, while the layout employed by the `vector_dist` class is controllable using the `memory_traits_inte` and `memory_traits_lin` memory configurations, together with the `Point` and `aggregate` types. `memory_traits_inte` specifies an interleaved memory layout, whereas `memory_traits_lin` leads to a linear memory layout.

Shared Memory Layouts Figure 6.1 depicts the shared memory layouts and the corresponding `vector_dist` and `memref` types. The positions are stored interleaved, with all x, y, and z coordinates being stored in consecutive arrays, resulting in a column-major memory layout. For each property, the x, y, and z entries of each particle are stored consecutively, resulting in two-dimensional arrays with row-major memory layouts. This layout was chosen because it was the easiest to integrate into MLIR while also providing decent spatial locality. A memory layout where the positions are also stored row-major is, because of limitations imposed by the `vector_dist` class, not possible without significant re-implementation efforts.

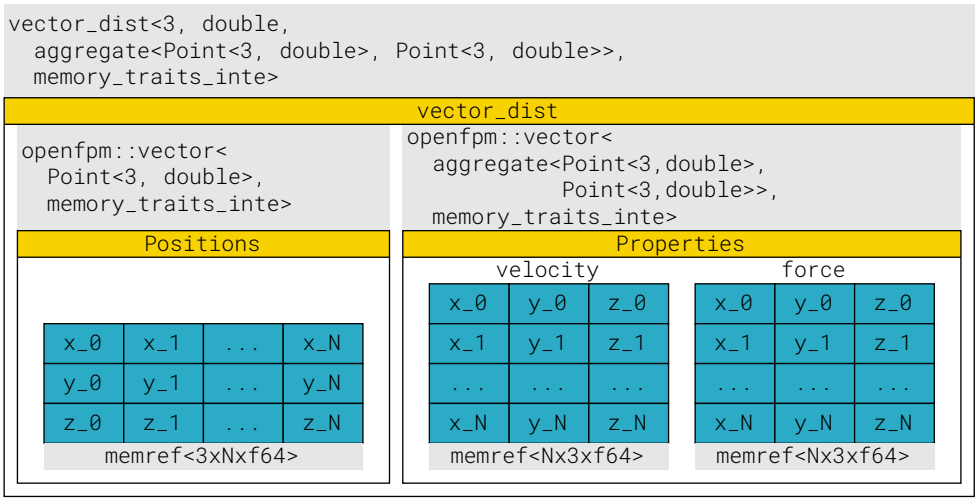

Figure 6.1: Shared memory layout between MLIR and OpenFPM

6.3 Neighbor List Dialects

Integrating the details of each neighbor list into the `particles_dist` dialect and its lowering passes may have been feasible. However, each neighbor list is sufficiently complex, while also being nearly self-contained, to warrant its own dialect. Because these dialects must be tightly integrated with `particles_dist`, they are not fully standalone. Instead, they constitute as special type of dialect, referred to as "companion dialect".

Two neighbor list dialects have been developed: `cell_list` and `local_domain`. The `cell_list` dialect is, as the name suggests, a dialect for cell lists. The `local_domain` dialect, on the other hand, does not represent a conventional neighbor list. Instead, it considers all particles within a subdomain, including ghost particles, as neighbors.

Both dialects share a highly similar syntax. In fact, their syntax is so similar that their commonalities were extracted into the abstract base dialect `neighbor_list`, from which both dialects inherit.

6.3.1 The Concept of Companion Dialects

A companion dialect complements another dialect, referred to as "leader", and exists only in conjunction with it. Using companion dialects enhances modularity and clarity by encapsulating distinct sub-concepts into separate dialects that would otherwise have to be integrated into the leader dialect. The alternative approach of integrating the functionality of all companion dialects directly into their leader dialect would risk making it overly complex, less modular, and harder to maintain, while diminishing its conceptual distinctness. Ultimately, using a companion dialect is primarily a stylistic choice rather than a technical necessity.

Dependencies between leader and companion dialects The benefits gained by employing a companion dialect come at the cost of increased lowering complexity. Just as a sub-concept cannot be fully separated from its super-concept, a companion dialect and its leader dialect are inherently linked due to dependencies between their operations, types and attributes. The left side of Figure 6.2 illustrates these inter-dialect dependencies, which can be categorized as follows:

"Use" dependencies:

- Operations of one dialect use types of the other as operand or result types.
- Operations of one dialect use attributes of the other as intrinsic attributes or properties.
- Types and attributes of one dialect use types or attributes of the other as parameters.

"Lowering" dependencies:

- Lowering an operation of one dialect introduces operations, types, or attributes of the other.
- Lowering a type or attribute of one dialect introduces types or attributes of the other.

Example The `particles_dist.for_all_neighbors` operation relies on the `cell_list.for_all_neighbor_indices` operation to iterate over the indices of a particle's neighbors. To this end, the `for_all_neighbor_indices` operation is introduced when lowering the `for_all_neighbors` operation. As illustrated by the right side of Figure 6.2, this constitutes

a lowering dependency. The `for_all_neighbor_indices` operation, in turn, requires an operand of type `!particles_dist.particle` to specify the particle over whose neighbors' indices to iterate. This constitutes a use dependency. Lowering the `for_all_neighbor_indices` operation introduces a `particles_dist.get_position` operation, as the position of a particle is used to compute the ID of the cell it resides in, constituting another lowering dependency.

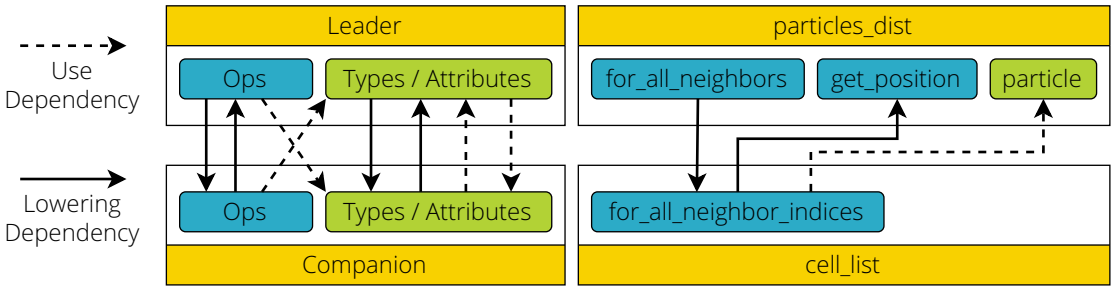

Figure 6.2: Left: Dependencies between a companion dialect and its leader. Right: Example of dependencies between `particles_dist` and `cell_list`.

Two-Step Interleaved Tick-Tock Strategy The left side of Figure 6.2 may suggest circular lowering dependencies between the leader dialect and its companions, which would make lowering impossible. However, the right side clarifies that there are no actual circular dependencies, but rather dependency trees that lead from one dialect to another. Nevertheless, a leader dialect must always be lowered alongside its companion dialects, either by lowering everything in a single pass, or by lowering all dialects using a "two-step tick-tock" strategy. In this work, the latter approach was selected as it provides higher modularity and better maintainability. The two-step tick-tock method consists of two steps, where each step is further divided into two alternating phases: a "tick" phase and a "tock" phase.

Step 1: Lower Operations The first step involves converting the operations. During a "tick" phase, the operations of the leader dialect are lowered, while during a "tock" phase, the operations of the companion dialects are lowered. The total number of individual tick and tock steps required corresponds to the length of the longest lowering dependency chain plus the final lowering step. For example, in the case of the `cell_list` and `particles_dist` dialects, the longest lowering dependency chain is two, meaning that after tick-tock-tick, all operations are lowered.

Step 2: Lower Types The second step involves converting attributes and types, which may also be performed in several alternating tick and tock phases if necessary. This must be done after lowering the operations because the operations and their conversion patterns depend on the presence of the non-lowered attributes and types.

Companion Dialects vs. Specialization Dialects Section 4.1.2 introduces the concept of specialization dialects. Companion dialects can sometimes serve as a viable alternative to specialization dialects. However, while companion dialects enable the introduction of additional IR constructs, they do not permit the extension of existing ones. The `particles_dist` dialect could therefore not have been a companion dialect of the `particles` dialect.

6.3.2 The `neighbor_list` Dialect

The `neighbor_list` dialect serves as an abstract base dialect for both `cell_list` and `local_domain` and all future neighbor list dialects. At the center of this dialect and its subdialects is the `neighbor_list.for_all_neighbor_indices` operation, which iterates over the indices of a particle's neighbors. Enabling this operation is the primary purpose of all other operations and types defined by this dialect.

The `!neighbor_list.list` Type

The `!neighbor_list.list` type represents a loaded neighbor list. Like `!particles_base.set` and its subtypes, it also employs fake value semantics. It has no parameters, as its subtypes do not share a common set of parameters.

The `!neighbor_list.storage` Type

The `!neighbor_list.storage` type represents an unloaded neighbor list. Like `!particles_base.storage` and its subtypes, it employs memory semantics and is thus not associated with any transient state or buffered values. Instead, it represents a pointer to a region of memory that holds the necessary reference(s) to load a `!neighbor_list.list`. It expects a single `!neighbor_list.list` type parameter, indicating the neighbor list it stores.

The `neighbor_list.for_all_neighbor_indices` Operation

The `neighbor_list.for_all_neighbor_indices` operation iterates over the neighboring indices of the given `!particles_dist.particle` value using the provided `!neighbor_list.list`. Its primary role is to support the lowering of `particles_dist.for_all_neighbors` operations, with which it shares a similar syntax. It has one region that defines the computations to be performed. This region is executed exactly once for each neighbor index and is terminated using a `neighbor_list.yield` operation that yields the results of the computation. Whether this region is executed sequentially or in parallel for all neighbor indices is up to the implementation. Like the `particles_dist.for_all_neighbors` operation, the `neighbor_list.for_all_neighbor_indices` operation also has `reduction_kinds` and `with_self` properties, controlling the same aspects. After all neighbor indices have been visited, it returns the reduced interaction results.
Example:

```
%combined_force = "neighbor_list.for_all_neighbor_indices"(%neighbor_list,
                                                           %particle)
                            <{"with_self" = false,
                              "reduction_kinds" = [#particles_dist.reduction_kind<add>]}>
({^1(%neighbor_index : index):
  // The first operation typically loads %neighbor using its %neighbor_index:
  %neighbor = "particles_dist.load_particle"(%set, %neighbor_index)
                                           : (!distset, index) -> !distparticle
  >> Calculate the %force that %neighbor exhibits on %particle
  "neighbor_list.yield"(%force) : (vector<3xf64>) -> ()
}) : (!neighbor_list.list, !distparticle) -> (vector<3xf64>)
// %combined_force holds reduced interaction results
```

Listing 6.14: `neighbor_list.for_all_neighbor_indices` example

The `neighbor_list.yield` Operation

The `neighbor_list.yield` operation is used to terminate the region of `neighbor_list.for_all_neighbor_indices` operations. It takes a variadic number of operands, representing the computational results of the region it terminates.

Example:

```
"neighbor_list.yield"(%force) : (vector<3xf64>) -> ()
```

Listing 6.15: `neighbor_list.yield` example

The `neighbor_list.update` Operation

The `neighbor_list.update` operation updates the given `!neighbor_list.list` value using the provided `!particles_dist.set` operand and returns the updated neighbor list. It is introduced when lowering the `particles_dist.update_neighbor_list` operation.

Example:

```
%updated_list = "neighbor_list.update"(%neighbor_list, %set)
                                : (!neighbor_list.list, !distset) -> !neighbor_list.list
```

Listing 6.16: `neighbor_list.update` example

The `neighbor_list.update_internals` Operation

The `neighbor_list.update_internals` operation updates the internals of the given `!neighbor_list.list` argument and returns the updated value. It is introduced when lowering the `particles_dist.update_internals` operation, which is also responsible for updating the internals of all associated data structures, like neighbor lists.

Example:

```
%updated_list = "neighbor_list.update_internals"(%neighbor_list)
                                        : (!neighbor_list.list) -> !neighbor_list.list
```

Listing 6.17: `neighbor_list.update` example

The `neighbor_list.undef` Operation

The `neighbor_list.undef` operation creates an undefined `!neighbor_list.list` value of the specified return type. The returned value must be fully initialized before being used in a `neighbor_list.for_all_neighbor_indices` operation.

Example:

```
%uninitialized_list = "neighbor_list.undef"() : () -> !neighbor_list.list
```

Listing 6.18: `neighbor_list.undef` example

The `neighbor_list.load` and `neighbor_list.store` Operations

The `neighbor_list.load` operation loads the reference(s) to the OpenFPM data structure(s) stored by the given `!neighbor_list.storage` (if there are any) and incorporates them

into the provided `!neighbor_list.list` value. Conversely, the `neighbor_list.store` operation stores the references integrated by the given `!neighbor_list.list` value at the provided `!neighbor_list.storage` location.

Example:

```
!list_storage = !neighbor_list.storage<neighbor_list.list>
// Load reference(s) to OpenFPM objects stored by %storage:
%loaded_list = "neighbor_list.load"(%storage, %uninitialized_list)
                            : (!list_storage, !neighbor_list.list) -> !neighbor_list.list
>> Modify %loaded_list -> %modified_list
// Store references of the %modified_list back in %storage:
"neighbor_list.store"(%modified_list, %storage):(!neighbor_list.list, !list_storage) -> ()
```

Listing 6.19: `neighbor_list.load` and `neighbor_list.store` example

The `neighbor_list.alloca_buffers` Operation

As explained in Section 6.2, passing structs by reference between MLIR and C++ requires pre-allocated communication buffers. The `neighbor_list.alloca_buffers` operation allocates the buffers necessary for carrying out runtime calls on the stack and inserts the references to the allocated memory regions into the given `!neighbor_list.list` value.

Example:

```
%list_with_allocated_buffers = "neighbor_list.alloca_buffers"(%list)
                                        : (!neighbor_list.list) -> !neighbor_list.list
```

Listing 6.20: `neighbor_list.load` example

6.3.3 The `local_domain` Dialect

The `local_domain` dialect is a subdialect of `neighbor_list`, inheriting all its type and operation definitions. It does not extend any of these definitions, but simply adopts them as they are, essentially replacing the `neighbor_list` prefix with `local_domain`. The `local_domain` neighbor list treats all particles within the same local subdomain, including ghost particles, as neighbors. Although this definition of a neighborhood is a simplistic, it is both practical and easy to implement.

6.3.4 The `cell_list` Dialect

Like `local_domain`, the `cell_list` dialect is also a subdialect of `neighbor_list`, inheriting all its type and operation definitions. As the name implies, it encapsulates cell lists. Depending on the hardware target, it either integrates the `CellList_gen` class from OpenFPM, specifically the one with the `Mem_fast` memory type, or the `CellList_gpu` class. It adopts most types and operations of the `neighbor_list` dialect without any adaption. Only the `!cell_list.list` type extends its base type.

The `!cell_list.list` Type The `!cell_list.list` type extends the `!neighbor_list.list` type by defining three additional parameters. The first parameter, `position_type`, specifies the numeric type used for the positions, while the second parameter, `dimensionality`, defines the dimensionality of the computational domain. Both must match

the corresponding parameters of the `!particles_dist.set` type they interact with. The third parameter, `cutoff_radius`, specifies the cutoff radius of the interaction, from which the size of the cells is derived.
Example:

```
!celllist = !cell_list.list<
  f64,                          // position_type
  3 : index,                    // dimensionality
  0.3 : f64                     // cutoff_radius
>
```

Listing 6.21: `!cell_list.list` example

6.4 Eliminating Redundant Stores of Particle Values

The `particles_dist.foreach` operation transparently loads the `!particles_dist.particle` region argument before each execution of its region and transparently stores the yielded value afterward. Internally, before entering the region, it loads all the fields that make up a particle from their respective `memref`s and inserts them into the `!particles_dist.particle` region argument. Conversely, after exiting the region, when storing a yielded particle, it extracts all its fields and stores them back into their respective `memref`s. This introduces potentially redundant store operations. A store operation is redundant if it stores a value back at the location it was loaded from. In the absence of data dependencies, such store operations can be removed.

6.4.1 The `memwrap` Dialect

The `memwrap` dialect together with the `memwrap-eliminate-redundant-store-ops` pass were developed to eliminate redundant stores in the guaranteed absence of data dependencies. The `memwrap` dialect wraps load and store operations from both the `memref` and `vector` dialects, while the `memwrap-eliminate-redundant-store-ops` pass identifies redundant `memwrap` store operations and removes them. Additionally, the `memwrap` dialect streamlines loading and storing of single-ranked vectors from two-ranked `memref`s with either row-major or column-major memory layouts.

The `memwrap.load` and `memwrap.store` Operations

The `memwrap.load` operation wraps the `memref.load` operation, while `memwrap.store` wraps the `memref.store` operation. Both operations have identical syntaxes to their `memref` counterparts, essentially replacing the `memref` prefix with `memwrap`.
Example:

```
%value = "memwrap.load"(%memref, %index) : (memref<?xf64>, index) -> f64
"memwrap.store"(%value, %memref, %index) : (f64, memref<?xf64>, index) -> ()
```

Listing 6.22: `memwrap.load` and `memwrap.store` example

The `memwrap.load_vector` and `memwrap.store_vector` Operations

The `memwrap.load_vector` operation loads a single-ranked `vector` from the provided two-ranked `memref`. An `index` operand specifies the position of the vector to load. Additionally, the operation accepts an optional `memref_transposed` unit attribute property, which acts as a flag to indicate whether the `memref` is transposed, i.e., uses a column-major memory layout. When the `memref_transposed` flag is absent, the index operand specifies the row to load. If present, it specifies the column. This property was introduced to streamline the loading and storing of particle positions, the `memref` of which uses a column-major memory layout (see Figure 6.1).

Example:

```
// Properties use ROW-MAJOR memory layout:
// %force_memref = | (x0 y0 z0) |
//                 | (x1 y1 z1) |
//                 |     ...    |
//                 | (xN yN zN) |
// Force of particle with index 1 is stored as second ROW.
// Returns (x1 y1 z1):
%force = "memwrap.load_vector"(%force_memref, %one)
                              : (memref<3x?xf64>, index) -> vector<3xf64>

// Position memref uses COLUMN-MAJOR memory layout:
// %pos_memref = | (x0 x1 ... xN) |
//               | (y0 y1 ... yN) |
//               | (z0 z1 ... zN) |
// Position of particle with index 1 is stored as second COLUMN.
// Returns (x1 y1 z1):
%pos = "memwrap.load_vector"(%pos_memref, %one) <{"memref_transposed"}>
                            : (memref<?x3xf64>, index) -> vector<3xf64>
```

Listing 6.23: `memwrap.load_vector` example

The `memwrap.store_vector` operation stores a single-ranked `vector` operand at the provided position in the given two-ranked `memref`. Like the `memwrap.load_vector` operation, the position is specified via an `index` operand, and the memory layout is specified via a `memref_transposed` property.

Example:

```
// %force_memref = | (x0 y0 z0) |
//                 | (x1 y1 z1) |
//                 |     ...    |
//                 | (xN yN zN) |
// Replaces x1 y1 z1:
"memwrap.store_vector"(%force, %force_memref, %one)
                      : (vector<3xf64>, memref<3x?xf64>, index) -> ()

// %pos_memref = | (x0 x1 ... xN) |
//               | (y0 y1 ... yN) |
//               | (z0 z1 ... zN) |
// Replaces x1 y1 z1:
"memwrap.store_vector"(%position, %pos_memref, %one) <{"memref_transposed"}>
                      : (memref<?x3xf64>, index) -> vector<3xf64>
```

Listing 6.24: `memwrap.load_vector` example

7 Phase 3: Generating the Runtime and Lowering the `particles_dist` Dialect

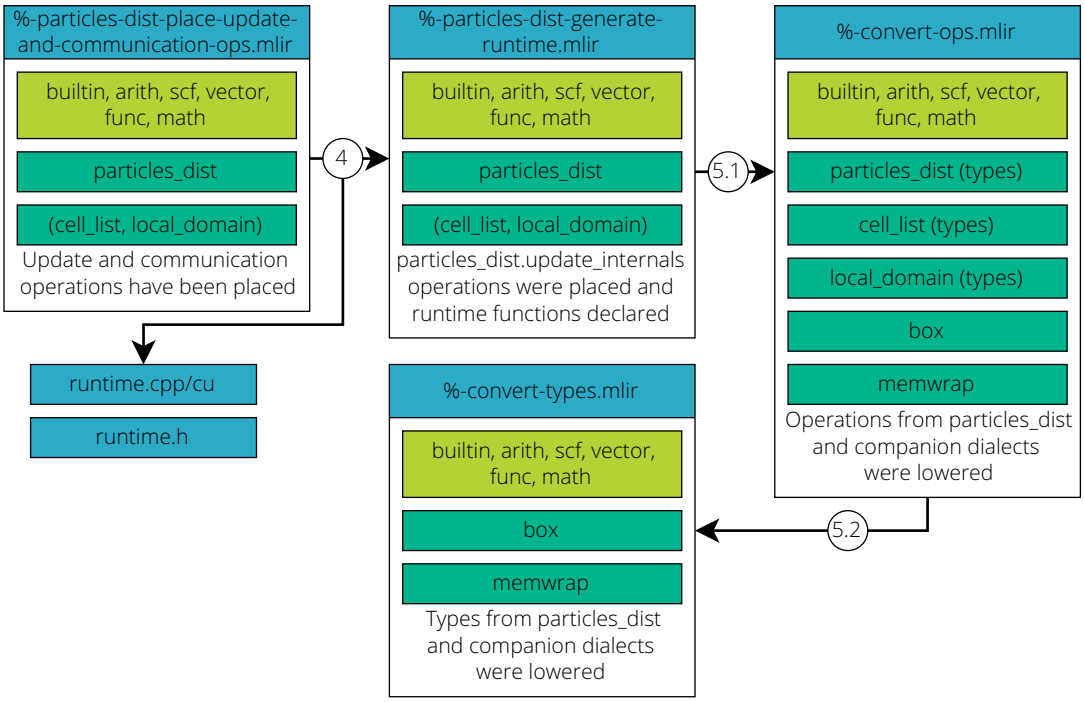

Figure 7.1: Phase 3 uses the output of phase 2 as input and has two steps. This first step (overall step ④) generates the runtime and places `particles_dist.update_internals` operations. The second step (overall step ⑤) converts the `particles_dist` dialect and its `cell_list` and `local_domain` companion dialects. It is split into two sub-steps; the first one converts the operations, while the second one converts the types.

Phase 3 focuses on the passes that generate the runtime and lower the `particles_dist` dialect and its `neighbor_list` companion dialects. The first section of this chapter covers the insertion of `particles_dist.update_internals` operations and the generation of the runtime. Next, Section 7.1 introduces the internal operations defined by the `particles_dist` dialect to support the lowering process. Finally, Section 7.3 describes the conversion passes that lower the `particles_dist` and its neighbor list companion dialects.

7.1 Pre-Lowering Steps

7.1.1 The particles-dist-place-update-internals-ops Pass

The particles-dist-place-update-internals-ops pass inserts particles_dist.update_internals operations before any operation that requires fresh internals. Operations such as particles_dist.foreach and particles_dist.for_all_neighbors require them up-to-date, whereas operations such as particles_dist.map make them stale. Because updating these internals is relatively inexpensive, no staleness analysis is performed. Instead, the pass simply places a particles_dist.update_internals operation before any operation that needs them fresh. Consequently, this pass must be executed only once to avoid redundant insertions.
Example:

```
%evolved_set = "particles.foreach_dist"(%set) ({
  // Evolve particle set
}) : (!distset) -> !distset
```

is transformed into:

```
%set0 = "particles_dist.update_internals"(%set) : (!distset) -> !distset
%evolved_set = "particles.foreach"(%set0) ({
  // Evolve particle set
}) : (!distset) -> !distset
```

Listing 7.1: Insertion of particles_dist.update_internals operations via particles-dist-place-update-internals-ops

7.1.2 The particles-dist-generate-runtime Pass

The particles-dist-generate-runtime pass generates two runtime files: a header file and a source file. The header defines all structs exchanged between MLIR and C++. This includes the MemRef struct shown in Listing 6.12 as well as the structs used for extracting internals, such as ParticleSetInternals from Listing 6.11. It also defines the struct used to pass the staleness flags to C++ functions invoked by particles_dist.call operations with the uses_staleness_flags property. The !particles_dist.storage type lowers to a pointer to a struct containing all necessary references to integrate a vector_dist instance along with all its associated neighbor lists. This struct, an example of which is shown in Listing 7.2, is also defined in the header. In addition to defining structs for exchanging data, the header includes fully specified definitions of all OpenFPM template classes in use. For instance, the my_vector_dist and my_cell_list types in Listing 7.2 are fully parametrized vector_dist and CellList_gen/CellList_gpu template classes, respectively.

```
struct ParticleSet {
  my_vector_dist *vd;
  struct {
    my_cell_list *cell_list_0;
  } neighbor_lists;
};
```

Listing 7.2: ParticleSet struct example

Other than defining types, the header also declares all necessary functions for extracting internals, as well as wrapper functions for invoking member functions of OpenFPM objects

(see Section 6.2). The definition of these functions are located in the source file. Both files are generated using a mixture of template files and code generation.

The functions and types to define or declare are derived from the particles_dist types and operations present in the IR. Therefore, the particles-dist-generate-runtime pass must run after all update and communication operation have been placed.

In addition to generating the header and source files, the particles-dist-generate-runtime pass also declares all C++ runtime function as func.func operations within the MLIR file. These declarations are needed later for converting operations that lower to runtime calls.

The particles-dist-generate-runtime pass is configured using four options:

- **header-file**: Output path for the generated header file.
- **source-file**: Output path for the generated source file.
- **header-file-include-path**: Include path for including the header within the source file.
- **target**: The hardware target, which must either be cpu or cuda.

As indicated by the target option, this pass marks the point where the lowering pipeline begins to differentiate between hardware targets. Based on the specified target, different runtime functions and type definitions are generated. For example, when targeting the cpu the CellList_gen class with the Mem_fast memory type is used as cell list implementation, whereas for cuda the CellList_gpu class is used. Similarly, the vector_dist class is instantiated with different Memory parameters, with HeapMemory being used for cpu, and CudaMemory being used for cuda. For the cuda target, device runtime functions are also emitted.

7.2 particles_dist Internal Operations

The particles_dist dialect defines several internal operations that are only introduced during lowering. They are typically converted immediately after being introduced and therefore usually never appear in the IR. Defining these operations instead of directly emitting the lowered code has two advantages. First, it modularizes the conversion process and facilitates reuse. It also helps when troubleshooting individual conversion patterns by keeping the changes done by each conversion pattern manageable. Second, it enables integration testing, which relies on the access to internal operations and data structures. Internal operations can be broadly grouped into two categories: data operations and memory operations.

Data Operations

Data operations primarily handle the insertion and extraction of elements from both !particles_dist.set and !particles_dist.particle values. All of these operations lower to box operations.

- particles_dist.undef: Creates an undefined !particles_dist.set value.
- particles_dist.get_index: Extracts and returns the index of the provided particle.
- particles_dist.get_local_size: Extracts and returns the number of particles in the subdomain represented by the given !particles_dist.set value. The with_ghost property specifies whether to include the number of ghost particles.

- particles_dist.get_neighbor_list: Extracts and returns the neighbor list of the specified return type from the given !particles_dist.set value.
- particles_dist.get_storage: Each !particles_dist.set value holds a !particles_dist.storage, typically the one it was loaded from. This operation extracts and returns this value from the given !particles_dist.set.
- particles_dist.set_storage: Inserts the given !particles_dist.storage value into the provided !particles_dist.set argument.
- particles_dist.get_memref: Extracts and returns the memref for the specified field from the provided !particles_dist.set operand. The field is specified by the field string property, which must be either "pos" or the name of a particle property.
- particles_dist.get_vector_dist: Extracts and returns the reference to the vector_dist instance integrated by the provided !particles_dist.set value.

Memory Operations

Memory operations handle allocating buffers and loading and storing of !particles_dist.set and !particles_dist.particle values.

- particles_dist.alloca_buffers: Allocates the communication buffers for the given !particles_dist.set value on the stack and inserts the references to these buffers into the !particles_dist.set operand.
- particles_dist.load: Loads the references to the OpenFPM instances to be integrated by the given !particles_dist.set value from the provided !particles_dist.storage argument and inserts them into the !particles_dist.set value.
- particles_dist.store: Stores the references to the OpenFPM instances integrated by the given !particles_dist.set value at the provided !particles_dist.storage.
- particles_dist.load_particle: Loads the !particles_dist.particle with the provided index from the given !particles_dist.set argument and returns it.
- particles_dist.store_particle: Stores the given !particles_dist.particle value in the provided !particles_dist.set operand at the given index.

7.3 Lowering particles_dist and neighbor_list Dialects

After generating the runtime, the lowering process can begin. The particles_dist dialect and its cell_list and local_domain companion dialects must be lowered using the two-step tick-tock process introduced in Section 6.3.1. Each phase of this process consists of the following passes:

- Step 1: Lower Operations
  - tick: particles-dist-convert-ops
  - tock: local-domain-convert-ops with either cell-list-convert-ops-to-cpu or cell-list-convert-ops-to-cuda
- Step 2: Lower Types
  - tick: particles-dist-convert-types
  - tock: local-domain-convert-types with either cell-list-convert-types-to-cpu or cell-list-convert-types-to-cuda

The type conversion passes, while being executed after the operation conversion passes, are presented first. Knowing how each type is lowered is crucial for understanding how the operations are converted, as the lowered operations must act on the lowered types.

7.3.1 The particles-dist-convert-types Pass

The particles-dist-convert-types pass converts all types of the particles_dist dialect to !box.box or !box.storage types.

Conversion of the !particles_dist.particle Type

The !particles_dist.particle type is lowered to a nested !box.box type with three entries: The index of the particle, the position as vector, and another !box.box that holds all particle properties.

Example:

```
!distparticle = !particles_dist.particle<
  f64,                              // position_type
  3 : index,                        // dimensionality
  {                                 // properties
    "velocity" = vector<3xf64>,     // velocity
    "force" = vector<3xf64>         // force
  }
>
```

is transformed into:

```
!particlebox = !box.box<[
  index,              // index
  vector<3xf64>,      // position
  !box.box<[          // properties
    vector<3xf64>,    // velocity
    vector<3xf64>     // force
  ]>
]>
```

Listing 7.3: Conversion of !particles_dist.particle via particles-dist-convert-types (!particlebox)

Conversion of the !particles_dist.set Type

Like the !particles_dist.particle type, the !particles_dist.set type is also lowered to a nested !box.box type, as shown in Listing 7.4. The resulting !box.box contains five nested boxes, each with a distinct purpose.

Example:

```
!distset = !particles_dist.set<
  -1 : index,                                       // size
  f64,                                              // position_type
  3 : index,                                        // dimensionality
  {                                                 // properties
    "velocity" = vector<3xf64>,
    "force" = vector<3xf64>
  },
  [!cell_list.list<f64, 3 : index, 0.3 : f64>]     // neighbor_lists
>
```

is transformed into:

```
!setbox = !box.box<[
  !storage_box,
  !neighbor_lists_box,
  !staleness_box,
  !internals_box,
  !buffers_box
]>
```

Listing 7.4: Conversion of `!particles_dist.set` via `particles-dist-convert-types` (`!setbox`)

Storage Box The first box is referred to as "storage box". It contains the references to the OpenFPM instances integrated by the `!particles_dist.set` value, and corresponds to the box stored by the matching `!particles_dist.storage` type. When stored as `!llvm.struct`, it matches the `ParticleSet` struct from Listing 7.2.

Example:

```
// Contains all references to OpenFPM data structures integrated by the particle set:
!storage_box = !box.box<[
  !llvm.ptr,                    // Pointer to vector_dist
  !box.box<[                    // Neighbor list storages:
    !cell_list.storage<
      !cell_list.list<f64, 3 : index, 0.3 : f64>>
  ]>
]>
```

Listing 7.5: Particle set storage box (`!storage_box`)

Neighbor Lists Box The second box is called "neighbor lists box" and holds all neighbor lists. First, in their non-lowered `!cell_list.list` or `!local_domain.list` form, and later as their lowered types.

Example:

```
// Contains all neighbor lists:
!neighbor_lists_box = !box.box<[
  !cell_list.list<f64, 3 : index, 0.3 : f64>
]>
```

Listing 7.6: Particle set neighbor lists box

Staleness Box The third box is named "staleness box". It holds the flags used for dynamic staleness tracking. As explained in Section 5.2.2, it contains one boolean entry for the domain decomposition, one for each field in the ghost layers, and one for each neighbor list.

Example:

```
// Contains staleness flags for all data structures associated with the particle set:
!staleness_box = !box.box<[
  i1,                           // Domain decomposition
  !box.box<[i1, i1, i1]>,       // Ghosts (position, velocity, force)
  !box.box<[i1]>                // Neighbor lists (cell list)
]>
```

Listing 7.7: Particle set staleness box (`!staleness_box`)

Internals Box The fourth box is called "internals box" and contains all internals of the `!particles_dist.set` value. Its entries correspond to the contents of the `ParticleSet_Internals` struct shown in Listing 6.11.

Example:

```
// Contains all internals (Values needed to integrate a vector_dist instance into MLIR):
!internals_box = !box.box<[
  index,                        // Size without ghost
  index,                        // Size with ghost
  memref<3x?xf64>,              // position memref
  !box.box<[                    // Property memrefs:
    memref<?x3xf64>,            // velocity memref
    memref<?x3xf64>             // force memref
  ]>
]>
```

Listing 7.8: Particle set internals box

Buffers Box The last box is referred to as "buffers box". As discussed in Section 6.2, to pass structs between MLIR and C++ by reference, communication buffers are required. To avoid allocating new buffers for every function call, buffers are pre-allocated once on the stack, and their references are stored in this box for reuse. No communication buffer is allocated for the storage box. Instead, the reference from which the particle set was loaded is re-used as communication buffer.

Example:

```
// Contains pointers to buffers for storing boxes as structs
!buffers_box = !box.box<[
  !box.storage<!storage_box>,   // Reference from which the storage box was loaded from
  !box.storage<!internals_box>, // Communication buffer for the internals box
  !box.storage<!staleness_box>  // Communication buffer for the staleness box

]>
```

Listing 7.9: Particle set buffers box

Conversion of the `!particles_dist.storage` Type

The `!particles_dist.storage` type is lowered to a `!box.storage` type, which stores the storage box detailed in the previous section.

Example:

```
!diststorage = !particles_dist.storage<!distset>
```

is transformed into:

```
!box.storage<!storage_box>
```

Listing 7.10: Conversion of `!particles_dist.storage` via `particles-dist-convert-types`

7.3.2 The `particles-dist-convert-ops` Pass

The `particles-dist-convert-ops` pass converts all `particles_dist` operations. It has two pass options: `target` and `is-final-conversion`. The `target` option specifies the hardware target, accepting either `cpu` or `cuda`. The hardware target affects only the conversion of

particles_dist.foreach operations. All other operation conversions are target-agnostic. The is-final-conversion option indicates whether the pass is used for the final lowering phase of the two-step interleaved tick-tock process introduced in Section 6.3.1. When set to true, conversions that usually introduce operations from the local_domain or cell_list dialect raise errors instead.

Conversion of particles_dist.for_all_neighbors Operations

Each particles_dist.for_all_neighbors operation is converted into a particles_dist.foreach containing a for_all_neighbor_indices operation from the appropriate neighbor list dialect. At the start of the for_all_neighbor_indices region, the neighbor particle is loaded from its index using a particles_dist.load_particle operation. The particle argument of the foreach region, together with the loaded neighbor, replace the center particle and neighbor particle arguments of the original interaction region, which is inlined into the for_all_neighbor_indices region. Following the for_all_neighbor_indices operation, the reduced results are inserted into the center particle via particles_dist.set_property and particles_dist.set_position operations. If the for_all_neighbors operation has a pre-interaction region, it is inlined before the for_all_neighbor_indices operation, while the post-interaction region is inlined after the insertion of the reduced results.
Example:

```
%interacted = "particles_dist.for_all_neighbors" (%set)
<{
  "with_self" = false,
  "reduction_kinds" = [#particles_dist.reduction_kind<add>],
  "write_targets" = ["force"],
  "neighbor_list_kind" = #particles_dist.neighbor_list_kind<cell_list>,
  "max_distance" =  0.3 : f64
}>
({^1(%particle : !distparticle):                                  // pre-interaction region
  >> Modify particle before interactions: %particle -> %updated_particle
  "particles_dist.yield" (%updated_particle) : (!distparticle) -> ()
},
{^2(%particle : !distparticle):                                   // post-interaction region
  >> Modify particle after interactions: %particle -> %updated_particle
  "particles_dist.yield" (%updated_particle) : (!distparticle) -> ()
},
{^3(%particle : !distparticle, %neighbor : !distparticle):        // interaction region
  >> Calculate the %force that %neighbor exhibits on %particle
  "particles_dist.yield" (%force) : (vector<3xf64>) -> ()
}) : (!distset) -> !distset
```

is transformed into:

```
%cell_list = "particles_dist.get_neighbor_list"(%set) : (!distset) -> !celllist
%interacted = "particles_dist.foreach"(%set)
({^1(%particle : !distparticle):
  >> Inlined pre-interaction region modifying %particle -> %updated_particle0
  // Interaction:
  %combined_force = "cell_list.for_all_neighbor_indices"(%cell_list, %updated_particle0)
                          <{"with_self" = false,
                            "reduction_kinds" = [#particles_dist.reduction_kind<add>]}>
  ({^2(%neighbor_index: index):
    // Load neighbor:
    %neighbor = "particles_dist.load_particle"(%set, %neighbor_index)
                                              : (!distset, index) -> !distparticle
    >> Calculate %force that %neighbor exibits on %updated_particle0
    "cell_list.yield"(%force) : (vector<3xf64>) -> ()
  }) : (!celllist) -> vector<3xf64>
  // Insert reduced interaction results into center particle
  %updated_particle1 = "particles_dist.set_property" (%updated_particle0, %combined_force)
                                       {"property" = "force"}
                                       : (!distparticle, vector<3xf64>) -> !distparticle
  >> Inlined post-interaction region modifying %updated_particle1 -> %updated_particle2
  "particles_dist.yield"(%updated_particle2) : (!distparticle) -> ()
}) : (!distset) -> !distset
```

Listing 7.11: Conversion of particles_dist.for_all_neighbors via particles-dist-convert-ops

Conversion of particles_dist.foreach Operations

Each particles_dist.foreach operation is converted into an scf.parallel operation that iterates over all particle indices within a subdomain, excluding ghost particles. At the start of the scf.parallel region, each particle is loaded from its index using a particles_dist.load_particle operation, while the updated particle is stored at the end with a particles_dist.store_particle operation. In between, the region of the particles_dist.foreach operation is inlined.

Example:

```
%evolved = "particles_dist.foreach"(%set)
({^1(%particle : !distparticle):
  >> Update %particle -> %updated_particle
  "particles_dist.yield"(%updated_particle) : (!distparticle) -> ()
}) : (!distset) -> !distset
```

is transformed into:

```
%zero = arith.constant 0 : index
%size = "particles_dist.get_local_size"(%set) <{with_ghost = false}> : (!distset) -> index
%one = arith.constant 1 : index
scf.parallel (%index) = (%zero) to (%size) step (%one) {
  // Load %particle:
  %particle = "particles_dist.load_particle"(%set, %index)
                                           : (!distset, index) -> !distparticle
  >> Update %particle -> %updated_particle
  // Store %updated_particle:
  "particles_dist.store_particle"(%set, %updated_particle, %index)
                                 : (!distset, !distparticle, index) -> ()
  scf.reduce
}
```

Listing 7.12: Conversion of particles_dist.foreach via particles-dist-convert-ops

If the target pass option of the particles-dist-convert-ops pass is cpu, no further rewriting is performed. If the target is cuda, the resulting one-dimensional scf.parallel

operation (Listing 7.12) is further transformed into a two-dimensional `scf.parallel` (Listing 7.13), with the first dimension being mapped to the block ID and the second dimension to the thread ID. Within the region of the new `scf.parallel` operation, the particle index is calculated from both IDs, replacing the index argument of the original `scf.parallel` region. If the particle index is within bounds, the original `scf.parallel` region is executed; otherwise, no calculations are performed. Furthermore, a `"mapping"` attribute is added to the new `scf.parallel` operation. This attribute is later used by the built-in `convert-parallel-loops-to-gpu` pass to convert the `scf.parallel` operation to a `gpu.launch` operation. For the block size, a constant value of 512 is used.

Example:

```
%block_size = arith.constant 512 : index
%num_blocks = arith.ceildivui %to, %block_size : index
scf.parallel
  (%block_idx , %thread_idx) = (%zero, %zero)
  to (%num_blocks, %block_size)
  step (%one, %one)
{
  // Calculate particle index from block ID and thread ID:
  %block_start = arith.muli %block_idx, %block_size : index
  %index = arith.addi %block_start, %thread_idx : index
  // Execute original region if particle %index is within bounds
  %within_bounds = arith.cmpi ult %index, %size : index
  scf.if %within_bounds {
    // Original scf.parallel body
  } else {}
  scf.reduce
} {"mapping" = [
    #gpu.loop_dim_map<
        processor = block_x,
        map = (d0) -> (d0),
        bound = (d0) -> (d0)>,
    #gpu.loop_dim_map<
        processor = thread_x,
        map = (d0) -> (d0),
        bound = (d0) -> (d0)>]}
```

Listing 7.13: Final result of converting `particles_dist.foreach` via `particles-dist-convert-ops` targeting cuda

Conversion of `particles_dist.load_particle` and `particles_dist.store_particle` Operations

As outlined in Section 6.2, particle positions and properties are stored in separate `memref`s. Loading a `!particles_dist.particle` involves loading its position and property values from those `memref`s using its index. Conversely, after modifying a `!particles_dist.particle`, its updated position and property values must be written back to the appropriate memory locations.

Each `particles_dist.load_particle` operation is lowered to a series of `memwrap.load` and `memwrap.load_vector` operations, which retrieve the position and each property value associated with the particle. The loaded values are then packed into a `!box.box` of the corresponding type (Listing 7.3). Conversely, each `particles_dist.store_particle` operation is converted into a `box.unpack` operation, followed by a series of `memwrap.store` and `memwrap.store_vector` operations, which store the particle's individual values. The

position memref has a column-major layout, meaning it is effectively transposed. Because of this, the memwrap.load_vector and memwrap.store_vector operations for the positions have the memref_transposed property present.

Example:

```
%particle = "particles_dist.load_particle" (%set, %index)
                                          : (!distset, index) -> !distparticle
```

is transformed into:

```
%setbox = builtin.unrealized_conversion_cast %set : (!distset) to !setbox
>> Extract all memrefs from %setbox -> %pos_memref, %vel_memref, %force_memref
// Extract position and property values:
%pos = "memwrap.load_vector"(%pos_memref, %index) <{"memref_transposed"}>
         : (memref<3x?xf64>, index, index) -> vector<3xf64>
%vel = "memwrap.load_vector"(%vel_memref, %index)
         : (memref<?x3xf64>, index, index) -> vector<3xf64>
%force = "memwrap.load_vector"(%force_memref, %index)
         : (memref<?x3xf64>, index, index) -> vector<3xf64>
// Pack values into a box and convert to !particles_dist.set type:
%particlebox = "box.pack"(%index, %pos, %vel, %force)
                 : (index, vector<3xf64>, vector<3xf64>, vector<3xf64>) -> !particlebox
%particle = "builtin.unrealized_conversion_cast"(%particlebox)
                                            : (!particlebox) -> !distparticle
```

Listing 7.14: Conversion of particles_dist.load_particle via particles-dist-convert-ops

Conversion of Update and Communication Operations

As outlined in Section 6.2, the particles_dist.map and particles_dist.ghost_get operations are converted into runtime calls. The required function declarations were inserted during the particles-dist-generate-runtime pass (Section 7.1.2). Each particles_dist.update_neighbor_list operation is lowered to either a cell_list.update or a local_domain.update operation, based on the type of the updated neighbor list.

Example:

```
%updated_set0 = "particles_dist.map"(%set) : (!distset) -> !distset
%updated_set1 = "particles_dist.ghost_get"(%updated_set0) <{"fields_to_get" = ["pos"]}>
                                          : (!distset) -> !distset
%updated_set2 = "particles_dist.update_neighbor_list"(%updated_set1)
                  <{"index" = 0 : index}> : (!distset) -> !distset
```

is transformed into:

```
// Converted particles_dist.map:
%vector_dist0 = "particles_dist.get_vector_dist"(%set) : (!distset) -> !llvm.ptr
func.call @map(%vector_dist0) : (!llvm.ptr) -> ()
// (Replace %updated_set0 with %set)
// Converted particles_dist.ghost_get:
%vector_dist1 = "particles_dist.get_vector_dist"(%set) : (!distset) -> !llvm.ptr
func.call @ghost_get_pos(%vector_dist1) : (!llvm.ptr) -> ()
// (Replace %updated_set1 with %set)
// Converted particles_dist.update_neighbor_list:
%cell_list = "particles_dist.get_neighbor_list"(%set) : (!distset) -> !celllist
%updated_cell_list = "cell_list.update"(%cell_list, %set)
                                       : (!celllist, !distset) -> !celllist
%updated_set2= "particles_dist.set_neighbor_list"(%set, %updated_cell_list)
                                            : (!distset, !celllist) -> !distset
```

Listing 7.15: Conversion of particles_dist update and communication operations via particles-dist-convert-ops

Conversion of particles_dist.update_internals Operations

Section 6.2 explains the concept of internals together with the strategy for updating them using runtime calls. It also introduces the ParticleSetInternals struct (Listing 6.11) used for transferring the internals of !particles_dist.set values. As described in Section 7.3.1, the internals box (Listing 7.8) of the lowered !particles_dist.set type corresponds to the ParticleSetInternals struct. To update the internals, the pointer to the communication buffer for this box, together with a reference to the integrated vector_dist instance, is passed to the update_internals runtime function. After the function returns, the internals box is loaded from the buffer and is inserted back into the lowered !particles_dist.set box.

When updating the internals of a !particles_dist.set value, the internals of all associated neighbor lists must also be updated. This is achieved by extracting each neighbor list, updating its internals using the corresponding neighbor_list.update_internals operations, and reinserting it.

Example:

```
%updated_set = "particles_dist.update_internals"(%set) : (!distset) -> !distset
```

is transformed into:

```
// Need !particles_dist.set as !box.box:
%setbox = builtin.unrealized_conversion_cast %set : (!distset) to !setbox
// Prepare and execute "update_internals" function call:
%vector_dist = "particles_dist.get_vector_dist"(%set) : !distset -> !llvm.ptr
%internals_box_buffer = "box.extract"(%setbox) <{"indices" = [4 : i64, 1 : i64]}>
                          : (!setbox) -> !box.storage<!internals_box>
func.call @update_internals(%vector_dist, %internals_box_buffer)
                           : (!llvm.ptr, !box.storage<!internals_box>) -> ()
// Load updated internals and insert them back into the box:
%updated_internals = "box.load"(%internals_box_buffer)
                                : (!box.storage<!internals_box>) -> !internals_box
%setbox0 = "box.insert"(%setbox, %updated_internals) <{"indices" = [3 : i64]}>
                       : (!setbox, !internals_box) -> !setbox
// Also update all internals of all neighbor lists:
%cell_list = "particles_dist.get_neighbor_list"(%set) : (!distset) -> !celllist
%updated_cell_list = "cell_list.update_internals"(%cell_list) : (!celllist) -> !celllist
%setbox1 = "box.insert"(%setbox0, %updated_cell_list) <{"indices" = [1 : i64, 0 : i64]}>
                       : (!setbox, !celllist) -> !setbox
// Convert !box.box back to original !particles_dist.set type:
%updated_set = builtin.unrealized_conversion_cast %setbox1 : !setbox to !distset
```

Listing 7.16: Conversion of particles_dist.update_internals via particles-dist-convert-ops

Conversion of particles_dist.loop Operations

Each particles_dist.loop operation is converted into an scf.for operation containing the time-stepping region, with the initialization region inlined before the loop. Prior to initialization, the !particles_dist.set value is created via a particles_dist.undef operation, and its buffers are allocated using particles_dist.alloca_buffers. Afterward, it is loaded with particles_dist.load, and its storage location is recorded using particles_dist.set_storage for later use. Finally, its internals are updated with a particles_dist.update_internals operation. After the scf.for operation, the particle set is stored using a particles_dist.store.

Example:

```
"particles_dist.loop" (%storage,
                       %zero,
                       %num_steps)
({^0(%uninitialized_set : !distset, %step : index):
  >> Initialize %uninitialized_set -> %initialized_set
  "particles.next" (%initialized_set) : (!distset) -> ()
},
{^1(%set : !distset, %step : index):
  >> Update %set -> %updated_set
  "particles.next"(%updated_set): (!distset) -> ()
}) : (!diststorage, index, index) -> ()
```

is transformed into:

```
%set0 = "particles_dist.undef"() : () -> !distset
%set1 = "particles_dist.alloca_buffers"(%set0) : (!distset) -> !distset
%set2 = "particles_dist.set_storage"(%set1, %storage)
                                    : (!distset, !diststorage) -> !distset
%set3 = "particles_dist.load"(%storage, %set2) : (!diststorage, !distset) -> !distset
%uninitialized_set = "particles_dist.update_internals"(%set3) : (!distset) -> !distset
>> Initialize %uninitialized_set -> %initialized_set
%finished_set = scf.for (%step) = (%zero) to (%num_steps) step (%one)
                        iter_args(%set = %initialized_set) -> !distset(
  >> Update %set -> %updated_set
  "scf.yield"(%updated_set) : (!distset) -> ()
)
"particles_dist.store"(%finished_set, %storage) : (!distset, !diststorage) -> ()
```

Listing 7.17: Conversion of `particles_dist.loop` via `particles-dist-convert-ops`

Conversion of Internal Operations

Getter and setter operations are lowered to `box.extract` and `box.insert` operations, while each `particles_dist.undef` operation is converted into a `box.undef`.

Example:

```
%uninitialized_set = "particles_dist.undef"() -> !distset

%pos = "particles_dist.get_position"(%set): (!particle) -> vector<3xf64>

%updated_set = "particles_dist.set_map_stale"(%set) <{"staleness_value" = true}>
                                            : (!distset) -> !distset
```

is transformed into:

```
// Converted particles_dist.undef:
%uninitialized_set_box = "box.undef"() -> !setbox
%uninitialized_set = builtin.unrealized_conversion_cast %setbox
                                                        : (!setbox) to !distset

// Converted particles_dist.get_position:
%particle_box = "builtin.unrealized_conversion_cast"(%box)
                                                    : (!distparticle) -> !particlebox
%pos = "box.extract"(%particle_box) <{"indices" = [1 : i64]}>
                    : (!particle_box) -> vector<3xf64>

// Converted particles_dist.set_map_stale:
%setbox = builtin.unrealized_conversion_cast %set : (!distset) to !setbox
%staleness_value = arith.constant true
%setbox = "box.insert"(%set, %staleness_value) <{"indices" = [2 : i64, 0 : i64]}>
                      : (!distset_box, i1) -> !distset_box
%updated_set = builtin.unrealized_conversion_cast %setbox : (!setbox) to !distset
```

Listing 7.18: Conversion of particles_dist getter and setter operations via particles-dist-convert-ops

Each particles_dist.alloca_buffers operation is lowered to a box.alloca operation, allocating memory for the internals box and the staleness box. A buffer for the storage box is not allocated, since the buffer from which it was originally loaded is reused to store the box. When allocating the buffers for a !particles_dist.set value, the buffers for the associated neighbor lists are also allocated using the appropriate neighbor_list.alloca_buffers operations.

Example:

```
%updated_set = "particles_dist.alloca_buffers"(%set) : (!distset) -> !distset
```

is transformed into:

```
%setbox = builtin.unrealized_conversion_cast %set : !distset to !setbox
// Allocate buffers for all boxes/structs communicated between MLIR and C++:
%storage_box_buffer = "box.alloca"() : () -> !box.storage<!storage_box>
%setbox0 = "box.insert"(%setbox, %storage_box_buffer) <{"indices" = [4 : i64, 1 : i64]}>
                       : (!setbox, !box.storage<!storage_box>) -> !setbox
%staleness_box_buffer = "box.alloca"() : () -> !box.storage<!staleness_box>
%setbox1 = "box.insert"(%setbox0, %staleness_box_buffer) <{"indices" = [4 : i64, 2 : i64]}>
                       : (!setbox, !box.storage<!staleness_box>) -> !setbox
// Allocate buffers for all neighbor lists:
%celllist = "box.extract"(%setbox1) <{"indices" = [1 : i64, 0 : i64]}>
                         : (!setbox) -> !celllist
%celllist0 = "cell_list.alloca_buffers"(%celllist) : (!celllist) -> !celllist
%setbox2 = "box.insert"(%setbox1, %celllist0) <{"indices" = [1 : i64, 0 : i64]}>
                       : (!setbox, !celllist) -> !setbox
%updated_set = builtin.unrealized_conversion_cast %setbox2 : !setbox to !distset
```

Listing 7.19: Conversion of particles_dist.alloca_buffers via particles-dist-convert-ops

Each particles_dist.load operation is converted into a box.load operation that loads the storage box, followed by a box.insert operation that inserts it into the !box.box representing the lowered !particles_dist.set. Furthermore, all associated neighbor lists are also loaded using the appropriate neighbor_list.load operations. Conversely, the particles_dist.store operation is lowered to a box.store operation that stores the storage box, followed by a series of neighbor_list.store operations storing all associated neighbor lists.

Example:

```
"particles_dist.store"(%set, %storage) : (!distset, !diststorage) -> ()
```

is transformed into:

```
%setbox = builtin.unrealized_conversion_cast %set : !distset to !setbox
%storageboxstorage = builtin.unrealized_conversion_cast %storage
                                                : !diststorage to !box.storage<!storage_box>
// Store the storage box:
%storagebox = "box.extract"(%setbox) <{"indices" = [0 : i64]}> : (!setbox) -> !storage_box
"box.store"(%storagebox, %storageboxstorage)
           : (!storage_box, !box.storage<!storage_box>) -> ()
// Store all neighbor lists:
%celllliststorage = "box.extract"(%setbox) <{"indices" = [0 : i64, 1 : i64, 0 : i64]}>
                                : (!setbox) -> !cell_list.storage<!celllist>
%celllist = "box.extract"(%setbox) <{"indices" = [1 : i64, 0 : i64]}>
                          : (!setbox) -> !celllist
"cell_list.store"(%celllist, %celllliststorage)
                 : (!celllist, !cell_list.storage<!celllist>) -> ()
```

Listing 7.20: Conversion of particles_dist.store via particles-dist-convert-ops

7.3.3 The local-domain-convert-types Pass

The local-domain-convert-types pass converts the !local_domain.list and !local_domain.storage types. Because the local_domain dialect defines a neighborhood as all particles within the same subdomain, it only needs the size of a subdomain to iterate over a particle's neighboring indices. The !local_domain.list type is therefore lowered to an index type that holds this value.

The !local_domain.list type does not integrate an OpenFPM object and must neither be stored nor loaded. As a result, the !local_domain.storage type serves no practical purpose. However, it serves a structural purpose. Because it is an element of the storage box introduced in Section 7.3.1, it must be converted into a placeholder type. The straightforward approach is to lower it also to index.

Example:

```
!local_domain.list

!local_domain.storage<!local_domain.list>
```

is transformed into:

```
// Converted !local_domain.list:
index           // Number of particles in the local subdomain, including ghosts

// Converted !local_domain.storage<!local_domain.list>:
index           // Placeholder value, has no practical use
```

Listing 7.21: Conversion of local_domain types via local-domain-convert-types

7.3.4 The local-domain-convert-ops Pass

The local-domain-convert-ops pass lowers all local_domain operations. Because the local_domain dialect uses a very simple notion of a neighborhood, converting most of its operations is straightforward. Several operations exist only because they were inherited from the neighbor_list dialect and fulfill no practical purpose. These operations, which

include local_domain.load, local_domain.store, local_domain.alloca_buffers, and local_domain.update_internals, are simply eliminated during conversion.

Conversion of local_domain.update Operations

The local_domain.update operation is converted into a runtime call that retrieves the size of the local subdomain, including the ghost particles. The returned index value then replaces the !local_domain.list operand.

Example:

```
%updated_list = "local_domain.update"(%list, %set) : (!local_domain.list) -> !local_domain.list
```

is transformed into:

```
%vector_dist_ptr = "particles_dist.get_vector_dist"(%set) : (!distset) -> !llvm.ptr
%num_particles = func.call @get_size_local_with_ghost(%vector_dist_ptr)
                                                    : (!llvm.ptr) -> index
%updated_list = unrealized_conversion_cast %num_particles : index to !local_domain.list
```

Listing 7.22: Conversion of local_domain.update via local-domain-convert-ops

Conversion of local_domain.for_all_neighbor_indices Operations

Each local_domain.for_all_neighbor_indices operation is lowered to an scf.parallel operation that iterates over all particle indices in a subdomain. To terminate the scf.parallel region, each reduction_kinds entry of the for_all_neighbor_indices operation is transformed into an scf.reduce operation that performs the specified reduction. If the with_self property of the for_all_neighbor_indices operation is set to true, the original for_all_neighbor_indices region is directly inlined at the beginning of the scf.parallel region. If the with_self property is set to false, the for_all_neighbor_indices region becomes the true region of an scf.if operation that takes its place instead. The true region is executed if the particle index and the neighbor index do not match. The false region is executed in case they match and yields a neutral value for each entry in reduction_kinds.

Example:

```
%combined_force = "local_domain.for_all_neighbor_indices"(%list, %particle)
                            <{"with_self" = false,
                              "reduction_kinds" = [#particles_dist.reduction_kind<add>]}>
({^1(%neighbor_index : index):
  >> Use %particle and %neighbor_index to calculate %force
  "local_domain.yield"(%force) : (vector<3xf64>) -> ()
}) : (!local_domain.list, !distparticle) -> vector<3xf64>
```

is transformed into:

```
// Neutral force for initialization and skipping self:
%zero_force = arith.constant dense<0.0> : vector<3xf64>
// Need !local_domain.list converted into index:
%local_size = unrealized_conversion_cast %list : !local_domain.list to index
%combined_force = scf.parallel (%neighbor_index) = (%zero) to (%local_size)
                              step (%one) init (%zero_force) -> vector<3xf64>
{
  %particle_index = "particles_dist.get_index"(%particle) : (!distparticle) -> index
  %not_self = arith.cmpi ne, %particle_index, %neighbor_index : index
  %force_or_neutral = scf.if %not_self -> (vector<3xf64>) {
    >> Use %particle and %neighbor_index to calculate %force
    scf.yield %force : vector<3xf64>
  } else {
    // Return neutral value for each reduction kind:
    scf.yield %zero_force : vector<3xf64>
  }
  // Reduce all pairwise interaction results:
  scf.reduce(%force_or_neutral : vector<3xf64>) {
  ^0(%force0: vector<3xf64>, %force1: vector<3xf64>):
    %result = arith.addf %force0, %force1 : vector<3xf64>
    scf.reduce.return %result : vector<3xf64>
  }
}
```

Listing 7.23: Conversion of `local_domain.for_all_neighbor_indices` via `local-domain-convert-ops`

7.3.5 The `cell-list-convert-types-to-cpu` Pass

The `cell-list-convert-types-to-cpu` pass converts the `!cell_list.list` and `!cell_list.storage` types to box types targeting CPUs. The `!cell_list.storage` type is lowered to an opaque `!llvm.ptr` that points to the integrated `CellList_gen` instance. The `!cell_list.list` type is converted into a `!box.box` with three entries. The first entry is an opaque `!llvm.ptr` that holds a reference to the integrated `CellList_gen` object. The second entry is a `!box.box`, referred to as "cell list CPU internals box". Analogous to the internals box of a converted `!particles_dist.set`, it contains the values required to access the data stored by the wrapped `CellList_gen` instance directly within MLIR. The third entry of the converted `!cell_list.list` type is a `!box.storage` that holds a reference to the communication buffer for the cell list CPU internals box.
Example:

```
!celllist = !cell_list.list<
  f64,                          // position_type
  3 : index,                    // dimensionality
  0.3 : f64>                    // cutoff_radius

!celllistst orage = !cell_list.storage<!cell_list>
```

is transformed into:

```
// Converted !celllist
!box.box<[
  !llvm.ptr,                                // Points to integrated CellList_gen instance
  !cl_cpu_internals_box,                    // Cell list CPU internals box
  !box.storage<!cl_cpu_internals_box>       // Com. buffer for cell list CPU internals box
]>

// Converted !celllststorage:
!llvm.ptr
```

Listing 7.24: Conversion of `!cell_list.list` and `!cell_list.storage` via `cell-list-convert-types-to-cpu`

As shown in Listing 7.25, the cell list CPU internals box has nine entries:

- The first entry (`cell_list`) is a `memref` of rank two, holding `index` values. The first dimension of this `memref` corresponds to the ID of a cell, and the second to the particle index within a cell. Given a cell ID and the number of particles in the cell, this `memref` is used to iterate over the indices of all particles located inside that cell.
- The second entry (`cl_n`) is a single-ranked `memref` that stores the number of particles within each cell as `index`.
- The third entry (`NNc_Full`) is also a single-ranked `memref` holding `index` values. It contains offset values that are used to calculate the IDs of all directly adjacent cells, given the ID of the cell in the center.
- The remaining six entries define the geometry of the cell list. Given a particle, these values are used to calculate the ID of the cell containing that particle from its position.

```
!cl_cpu_internals_box = !box.box<[
  memref<?x?xindex>,                // cell_list
  memref<?xindex>,                  // cl_n
  memref<?xindex>,                  // NNc_Full
  !box.box<[f64, f64, f64]>,        // .
  !box.box<[f64, f64, f64]>,        // ..
  !box.box<[index, index, index]>,  // ...
  !box.box<[index, index, index]>,  // cell list geometry
  !box.box<[index, index, index]>,  // ...
  !box.box<[index, index, index]>   // ..
]>
```

Listing 7.25: Cell list CPU internals box

7.3.6 The `cell-list-convert-ops-to-cpu` Pass

The `cell-list-convert-ops-to-cpu` pass converts all `cell_list` operations, targeting CPUs as processing hardware. The `cell_list.load`, `cell_list.store`, `cell_list.alloca_buffers`, `cell_list.undef`, and `cell_list.update_internals` operations are converted following the same approaches used for converting their `particles_dist` counterparts. Each `cell_list.update` operation is converted into a runtime call, similar to the `particles_dist.map` and `particles_dist.ghost_get` operations.

Conversion of `cell_list.for_all_neighbor_indices` Operations

Each `cell_list.for_all_neighbor_indices` operation is converted into two nested `scf.parallel` operations, with one operation inside the other. The outer `scf.parallel`

operation iterates over the IDs of all cells surrounding the given particle. This includes the cell containing the particle itself and all cells directly adjacent to it. Before entering the outer loop, the ID of the cell containing the given particle is calculated using the particle's position along with the last six geometry-defining entries of the cell list CPU internals box (see Listing 7.25). Within the outer `scf.parallel` region, each cell's ID is calculated using the ID of the cell in the center and its offset stored in the `NNc_Full` internals box entry. Before entering the inner loop, the number of particles in the cell is loaded from the `cl_n` internals entry.

The inner `scf.parallel` operation iterates over the indices of all particles within the cell whose ID was calculated. It uses the cell ID, together with a particle-in-cell induction variable, which goes from zero to the number of particles in that cell, to retrieve the index of a neighbor from the `cell_list` internals box entry. The remainder of the inner `scf.parallel` region looks identical to the `scf.parallel` region of a converted `local_domain.for_all_neighbor_indices` operation (Listing 7.23)

Example:

```
%combined_force = "cell_list.for_all_neighbor_indices"(%cell_list, %particle)
                            <{"with_self" = false,
                              "reduction_kinds" = [#particles_dist.reduction_kind<add>]}>
({^1(%neighbor_index : index):
  >> Use %particle and %neighbor_index to calculate %force
  "cell_list.yield"(%force) : (vector<3xf64>) -> ()
}) : (!celllist, !distparticle) -> vector<3xf64>
```

is transformed into:

```
>> Calculate ID of the cell containing %particle using its position and the last six
>> entries of the cell list CPU internals box -> %cell_id
// Neutral force for initialization and skipping self:
%zero_force = arith.constant dense<0.0> : vector<3xf64>
%num_surrounding_cells = arith.constant 27 : index // 3^D, with D=3
// Iterate over surrounding cells:
%combined_force = scf.parallel (%cell) in (%zero) to (%num_surrounding_cells)
                               step  (%one) init (%zero_force) {
  >> Retrieve %cell_id_offset from NNc_Full memref (internals) using %cell as index
  %cell_id = arith.addi %cell, %cell_id_offset : index
  >> Retrieve %num_particles_in_cell from cl_n memref (internals) using %cell_id as index
  // Iterate over particle indices in cell:
  %cmbnd_force_for_cell = scf.parallel (%particle_in_cell) in
                                       (%zero) to (%num_particles_in_cell)
                                       step (%one) init (%zero_force) {
    >> Retrieve %neighbor_index from cell_list memref (internals)
    >> using %cell_id and %particle_in_cell as indices

    // Remainder of body mirrors scf.parallel region
    // of lowered local_domain.for_all_neighbor_indices
  }
  scf.reduce(%cmbnd_force_for_cell : index) {
  ^1(%force0: vector<3xf64>, %force1: vector<3xf64>):
    %result = arith.addf %force0, %force1 : vector<3xf64>
    scf.reduce.return %result : vector<3xf64>
  }
}
```

Listing 7.26: Conversion of `cell_list.for_all_neighbor_indices` via `cell-list-convert-ops-to-cpu`

7.3.7 The cell-list-convert-types-to-cuda Pass

The cell-list-convert-types-to-cuda pass converts all cell_list types targeting CUDA GPUs. As for CPUs, the !cell_list.storage type is lowered to an opaque !llvm.ptr that references the integrated CellList_gpu instance. The conversion of the !cell_list.list type also follows the same approach used for CPUs, resulting in a !box.box with three entries. The main difference lies in the contents of the second entry, the "cell list GPU internals box".

The CellList_gpu class is not as tightly integrated into MLIR as the CellList_gen class. Rather than accessing its data directly within MLIR, CellList_gpu_ker and NN_gpu_it_box instances are used to iterate over the indices of a particle's neighbors. Their usage within MLIR mirrors their usage in C++.

The first entry of the internals box is an opaque !llvm.ptr pointing to a CellList_gpu_ker instance in device memory that corresponds to the integrated CellList_gpu object in host memory. The second entry is of type index and holds the size of a NN_gpu_it_box object in bytes. This value is required to allocate stack memory for NN_gpu_it_box instances.

Example:

```
!cl_gpu_internals_box = !box.box<[
  !llvm.ptr,                    // Pointer to CellList_gpu_ker
  index                         // sizeof(NN_gpu_it_box)
]>
```

Listing 7.27: Cell list GPU internals box

7.3.8 The cell-list-convert-ops-to-cuda Pass

The cell-list-convert-ops-to-cuda pass lowers all cell_list operations, targeting CUDA GPUs as processing hardware. As outlined in Section 7.3.7, the CellList_gpu class wrapped by the !cell_list.list type is not yet fully integrated into MLIR. Instead, an NN_gpu_it_box iterator instance, combined with device runtime calls, is used to iterate over the indices of a particle's neighbors. Integrating complex data structures like cell lists into MLIR requires considerable time and effort, which was simply not available. Although the partially integrated solution cannot fully exploit the advantages of MLIR, and is expected to deliver worse performance, it nevertheless provides full functionality. Furthermore, it offers a practical foundation for a fully integrated solution, where runtime function calls can be progressively replaced with direct data accesses and computations performed within MLIR.

Except for the cell_list.for_all_neighbor_indices operation, the conversation strategies of all operations follow the same approaches used by the cell-list-convert-ops-to-cpu pass.

Conversion of cell_list.for_all_neighbor_indices Operations

Each cell_list.for_all_neighbor_indices operation is lowered to an scf.while operation that utilizes a NN_gpu_it_box iterator instance to iterate over the indices of the given particle's neighbors. The NN_gpu_it_box instance is allocated on the stack and then initialized in-place via a device runtime call using the reference to the CellList_gpu_ker instance from the internals box. Since the scf.while operation is not terminated by an

scf.reduce operation, the pairwise interaction results are continuously reduced, with each iteration passing the reduced results to the next.

The NN_gpu_it_box iterator is used via the device runtime function get_next. This function takes a pointer to the NN_gpu_it_box instance to access its isNext and get member functions and to increment its counter. The function returns either the index of the next neighbor or -1 if no more neighbor indices are left. If the with_self property of the for_all_neighbor_indices operation is set to false, the get_next function is re-invoked when the neighbor index matches the center particle's index.

Example:

```
%combined_force = "cell_list.for_all_neighbor_indices"(%cell_list, %particle)
                            <{"with_self" = false,
                              "reduction_kinds" = [#particles_dist.reduction_kind<add>]}>
({^1(%neighbor_index : index):
  >> Use %particle and %neighbor_index to calculate %force
  "cell_list.yield"(%force) : (vector<3xf64>) -> ()
}) : (!celllist, !distparticle) -> vector<3xf64>
```

is transformed into:

```
>> Extract sizeof(NN_gpu_it_box) from internals box -> %size_of_nn_gpu_it_box
// Allocate stack memory for NN_gpu_it_box instance:
%nn_gpu_it_box_ptr = "llvm.alloca"(%size_of_nn_gpu_it_box)
                                  <{"elem_type" = i8}> : (i64) -> !llvm.ptr
>> Use runtime call to construct NN_gpu_it_box in-place for %particle at %nn_gpu_it_box_ptr
%particle_index = "particles_dist.get_index"(%particle) : (!distparticle) -> index
// Iterate over all neighbors indices of %particle using %nn_gpu_it_box_ptr:
%total_combined_force, %last_index = scf.while (%combined_force = %zero_force)
                                              : (vector<3xf64>) -> (vector<3xf64>, index)
{
  // Get next neighbor index by using device runtime call:
  %next = func.call @get_next(%nn_gpu_it_box_ptr) : (!llvm.ptr) -> index
  // Skip self:
  %is_self = arith.cmpi eq, %next, %particle_index : index
  %neighbor_index = scf.if %is_self -> (index) {
    %nextnext = func.call @get_next(%nn_gpu_it_box_ptr) : (!llvm.ptr) -> index
    scf.yield %nextnext : index
  } else {
    scf.yield %next : index
  }
  // @get_next returns -1 when NN_gpu_it_box.hasNext() returns false
  %not_done = arith.cmpi ne, %neighbor_index, %minus_one : index
  scf.condition(%not_done) %combined_force, %neighbor_index : vector<3xf64>, index
} do {
^0(%combined_force: vector<3xf64>, %neighbor_index: index):
  >> Use %particle and %neighbor_index to calculate %force
  // Continuously reduce interaction results:
  %new_combined_force = arith.addf %combined_force, %force : vector<3xf64>
  scf.yield %new_combined_force : vector<3xf64>
}
```

Listing 7.28: cell_list.for_all_neighbor_indices after applying cell-list-convert-ops-to-cuda

8 Phase 4: Lowering box and memwrap

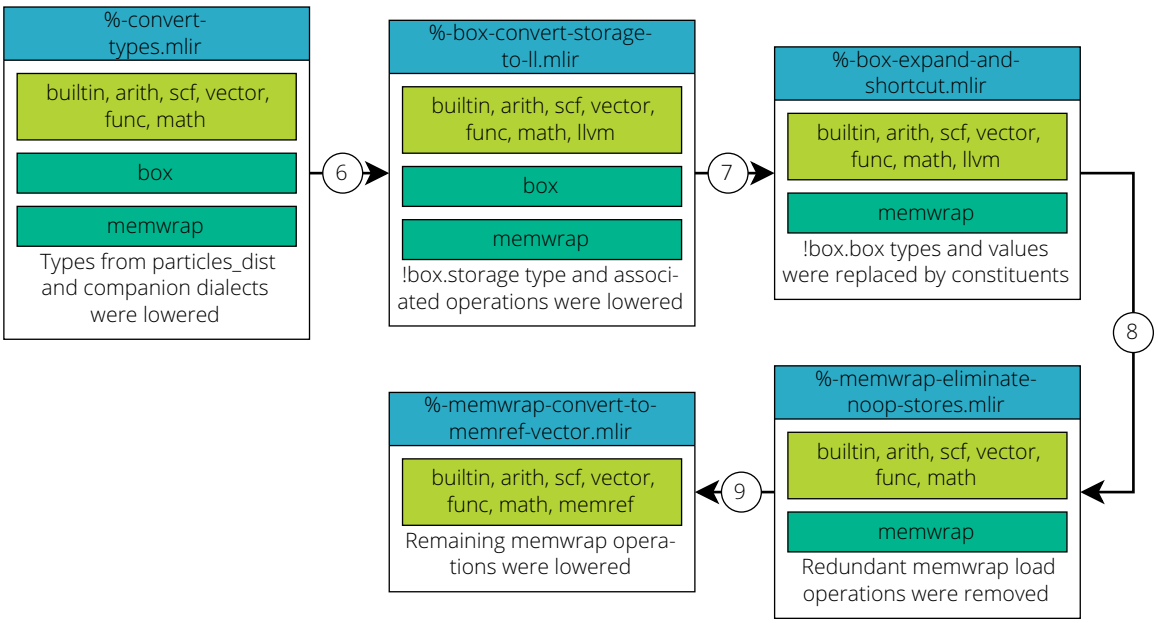

Figure 8.1: Phase 4 uses the output of phase 3 as input and consists of four steps. This first step (overall step ⑥) converts the `!box.storage` type and its associated operations to the `llvm` dialect. During the second step (overall step ⑦), all `!box.box` types and values are eliminated, as well as all remaining `box` operations, via 1-to-N conversion. In the third step (overall step ⑧), redundant `memwrap` load operations are removed. Finally, step four (overall step ⑨), converts any remaining `memwrap` operations to `memref` and `vector` operations.

Phase 4 focuses on transformation passes applied to the `box` and `memwrap` dialects, which are the only non-standard dialects remaining in the IR at the beginning of this phase. The first section of this chapter covers the pass that lowers the `!box.storage` type and its associated operations to the `llvm` dialect. Next, Section 8.2 describes the passes that eliminate all `!box.box` types and values, as well as all related operations, via expansion and shortcutting to achieve 1-to-N conversion. Section 8.3 discusses the pass that eliminates redundant `memwrap` load operations. Finally, Section 8.4 explains the pass that converts the remaining `memwrap` operations to `memref` and `vector`.

8.1 The box-convert-storage-to-ll Pass

The box-convert-storage-to-ll pass converts the !box.storage type, together with the box.load, box.store, and box.alloca operations, to llvm. Each !box.storage type is converted into an opaque !llvm.ptr. The !box.box types in the IR are not converted. However, as outlined in Section 6.1.1, during storage and transfer they are converted into !llvm.structs.

Conversion of the !box.box Type During Storage and Transfer

To be able to convert a !box.box type to an !llvm.struct, all its element types must either be natively supported as !llvm.struct element types or must be convertible. Convertible types include !box.box, !box.storage, index, and memref. All !box.box and !box.storage types are recursively converted into their corresponding llvm types. This means that nested !box.box types result in nested !llvm.structs. Each index entry is converted into the type specified by the index_type pass option. This option accepts only signless integer types (e.g., i32, i64). All memrefs element types are converted into !llvm.structs. Listing 8.2 shows the !llvm.struct a memref type is converted into. It matches the MemRef struct shown in Listing 6.12.

Example:

```
!boxtype = !box.box<[
  index,
  !box.box<[
    memref<?x3xf64>
  ]>
]>,
```

is transformed into:

```
!structtype = !llvm.struct<(
  i64,
  !llvm.struct<(
    !memref2dstruct
  )>
)>
```

Listing 8.1: Conversion of !box.box during storage and transfer via box-convert-storage-to-ll

with:

```
// Struct for a two-ranked memref:
!memref2dstruct = !llvm.struct<(
  !llvm.ptr,                    // allocated_ptr
  !llvm.ptr,                    // aligned_ptr
  i64,                          // offset
  !llvm.array<2xi64>,           // sizes
  !llvm.array<2xi64>            // strides
)>
```

Listing 8.2: Two-ranked memref converted into !llvm.struct. Corresponds to MemRef C++ struct shown in Listing 6.12.

Conversion of box.store and box.load Operations

A !box.box value that is stored via a box.store operation is unpacked using box.unpack, and its individual entries are inserted into a new !llvm.struct, the type of which

corresponds to the lowered !box.box type. During this process, !box.storage, index, and memref values are cast to their converted types using builtin.unrealized_conversion_cast operations. Once all entries have been inserted into the newly created struct, the resulting value is stored using llvm.store. The box.load operation simply reverses this process.

Example:

```
"box.store"(%box, %storage) : (!boxtype, !box.storage<!boxtype>) -> ()
```

is transformed into:

```
// Recursively unpack %box:
%index, %memref = "box.unpack"(%box) : (!boxtype) -> (index, memref<?x3xf64>)
%struct0 = llvm.mlir.undef : !structtype

// Convert index to i64 and insert:
%index_converted = builtin.unrealized_conversion_cast %size : index to i64
%struct1 = llvm.insertvalue %index_converted, %struct0[0] : !structtype

// Convert memref to !llvm.struct and insert:
%memref_converted = builtin.unrealized_conversion_cast %memref
                                                    : memref<3x?xf64> to !memref2dstruct
%struct2 = llvm.insertvalue %memref_converted, %struct1[1, 0] : !structtype

// Store struct:
%ptr = builtin.unrealized_conversion_cast %storage : !box.storage<!boxtype> to !llvm.ptr
llvm.store %struct2, %ptr : !structtype, !llvm.ptr
```

Listing 8.3: Conversion of box.store via box-convert-storage-to-ll

8.2 1-to-N Conversion of !box.box Values and Type via Expansion and Shortcutting

Rather than being converted, the !box.box type and its associated operations are eliminated through a series of expansion passes followed by a final shortcutting pass. Collectively, these passes replace each !box.box value or type with its constituent elements, while also eliminating all remaining box operations. Although the box-shortcut-extract-ops pass is executed last, it is presented first because its mechanism clarifies the objectives of the expansion passes.

8.2.1 The box-shortcut-extract-ops Pass

The box-shortcut-extract-ops pass finds for each box.extract operation the corresponding box.insert operation, removes the extract, and replaces its result with the inserted value. It expects the following preconditions to be satisfied:

1. Of the box dialect, only the box.undef, box.insert, and box.extract operations remain in the IR.
2. All box.insert and box.extract operations extract or insert individual entries, never !box.box values.
3. !box.box values are not used as operands of operations other than box.insert or box.extract.

4. Except for the box.undef and box.insert operations, no other operation returns a !box.box value.
5. No block arguments of type !box.box exist.

Due to the value semantics of the !box.box type and the listed conditions, each box.extract operation forms the tail of a def-use chain that originates at a box.undef, where each intermediate chain link is a box.insert operation. During the box-shortcut-extract-ops pass, the def-use chain of every box.extract operation is traversed backwards, while its indices are compared with the indices of each visited box.insert operation. When encountering a box.insert operation with matching indices, the box.extract operation is eliminated and its result is replaced with the inserted value. After all box.extract operations are eliminated, the remaining box.insert and box.undef operations are automatically removed during the next DCE pass.

Example:

```
!mybox = !box.box<[index, !box.box<[f64, f32]>]>
%ins0 = arith.constant 1 : index
%ins1_0 = arith.constant 1 : f64
%box = "box.undef" : () -> !mybox
// Insert values into box:
%box0 = "box.insert"(%box, %ins0) <{"indices" = [0 : i64]}> : (!mybox, index) -> !mybox
%box1 = "box.insert"(%box0, %ins1_0) <{"indices" = [1 : i64, 0 : i64]}>
                    : (!mybox, f64) -> !mybox
// Extract values from box:
%extr0 = "box.extract"(%box1) <{"indices" = [0 : i64]}> : (!mybox) -> index
%extr1_0 = "box.extract"(%box1) <{"indices" = [1 : i64, 0 : i64]}> : (!mybox) -> f64
// Use extract values:
"dummy.op"(%extr0, %extr1_0) : (index, i64) -> ()
```

gets shortcut to:

```
%ins0 = arith.constant 1 : index
%ins1_0 = arith.constant 1 : f64
%box = "box.undef" : () -> !mybox
// Insert values into box:
%box0 = "box.insert"(%box, %ins0) <{"indices" = [0 : i64]}> : (!mybox, index) -> !mybox
%box1 = "box.insert"(%box0, %ins1_0) <{"indices" = [1 : i64, 0 : i64]}>
                    : (!mybox, f64) -> !mybox
// Use inserted values instead of extracted ones:
"dummy.op"(%ins0, %ins1_0) : (index, i64) -> ()
```

Listing 8.4: Shortcutting of box.extract operations via box-shortcut-extract-ops (!mybox)

and once all box.extract operations are eliminated, the box.insert operations are removed during the next DCE pass:

```
%ins0 = arith.constant 1 : index
%ins1_0 = arith.constant 1 : f64
"dummy.op"(%ins0, %ins1_0) : (index, i64) -> ()
```

Listing 8.5: Elimination of unused box.extract and box.undef operations after applying box-shortcut-extract-ops

Reasons for Pass Failures

When traversing the def-use chain of a box.extract operation, reaching the box.undef operation at the origin causes the box-shortcut-extract-ops pass to fail. As exemplified in Listing 8.6, provided all preconditions are met, encountering a box.undef operation

indicates that the value extracted by the box.extract operation was never inserted. This, in turn, implies the use of an undefined value, making failure the correct outcome.

```
%ins0 = arith.constant 1 : index
%box = "box.undef" : () -> !mybox
%box0 = "box.insert"(%box, %ins0) <{"indices" = [0 : i64]}> : (!mybox, index) -> !mybox
%extr0 = "box.extract"(%box0) <{"indices" = [0 : i64]}> : (!mybox) -> index
// Fails because no entry was ever inserted at 1,0:
%extr1_0 = "box.extract"(%box0) <{"indices" = [1 : i64, 0 : i64]}> : (!mybox) -> f64
```

Listing 8.6: Failure of `box-shortcut-extract-ops` because of a missing inserted value

The `box-shortcut-extract-ops` pass also fails if any precondition is violated. For example, as shown in Listing 8.7, if an operation other than a box.undef or box.insert returns a !box.box value, the pass fails when encountering this operation during the traversal of the def-use chain. The same happens when the def-use chain originates at a block argument. If a !box.box value is used as an operand by an operation other than box.insert or box.extract, the pass does not directly fail during traversal, but it will fail to eliminate the box dialect from the IR. As illustrated by Listing 8.8, in cases where a box.insert operation inserts a !box.box value rather than an individual entry, matching the indices fails if a box.extract operation extracts an element of that box.

To prevent these situations and to guarantee that all preconditions are satisfied, the IR is prepared through several expansion passes, as discussed in the following sections.

```
%box = "dummy.op"() : () -> !mybox
// Extraction fails because !box.box def-use chain originates at illegal operation
%extr0 = "box.extract"(%box) <{"indices" = [0 : i64]}> : (!mybox) -> index
%extr1_0 = "box.extract"(%box) <{"indices" = [1 : i64, 0 : i64]}> : (!mybox) -> f64
```

Listing 8.7: Failure of `box-shortcut-extract-ops` caused by an operation other than box.insert or box.extract returning a !box.box

```
%ins0 = arith.constant 1 : index
%ins1_0 = arith.constant 1 : f64
%box = "box.undef" : () -> !mybox
%inner_box = "box.undef" : () -> !box.box<[f64, f32]>
>> Insert entries into %inner_box -> %inner_box2
// Insert inner_box2 into box:
%box0 = "box.insert"(%box, %inner_box) <{"indices" = [1 : i64]}>
                    : (!mybox, !box.box<[f64, f32]>) -> !mybox
// Fails because no direct indices match is found:
%extr1_0 = "box.extract"(%box1) <{"indices" = [1 : i64, 0 : i64]}> : (!mybox) -> f64
```

Listing 8.8: Failure of `box-shortcut-extract-ops` caused by the insertion of a !box.box value

8.2.2 General Strategy for Expanding !box.box Values and Types

Expansion refers to the process of rewriting operations by replacing all !box.box operands, results, and block arguments with their individual elements. While each expansion pass targets different operations, they all employ the same rewriting strategy explained in this section.

Expansion of `!box.box` Operands

If an operation uses a `!box.box` value as an operand, it is rewritten to use its constituent values instead. The individual values are obtained by disassembling the `!box.box` value using a `box.unpack` operation.

Example:

```
"dummy.op"(%box) : !mybox -> ()
```

is transformed into:

```
%val0, %val1_0, %val1_1 = "box.unpack" (%box) : (!mybox) -> (index, f64, f32)
"dummy.op"(%val0, %val1_0, %val1_1) -> ()
```

Listing 8.9: Expansion of `!box.box` operands

Expansion of `!box.box` Region Arguments

If a region has a `!box.box` region argument, its owning operation is rewritten to replace the argument with multiple arguments, each corresponding to one of the box's individual elements. At the beginning of the region, the individual arguments are reassembled into a `!box.box` of the original argument type using a `box.pack` operation. The resulting value then replaces the original region argument.

Example:

```
"dummy.op"(){
^1{%arg : !mybox}: // !box.box region argument
  "dummy.yield"(%arg) -> ()
} : () -> ()
```

is transformed into:

```
"dummy.op"(){
^1{%arg0: index, %arg1_0 : f64, %arg1_1 : f32}: // Arg types match element types of !mybox
  // Reassemble individual arguments into value of original !mybox type:
  %arg = "box.pack"(%arg0, %arg1_0, %arg1_1) : (index, f64, f32) -> !mybox
  "dummy.yield"(%arg) -> ()
} : () -> ()
```

Listing 8.10: Expansion of `!box.box` region arguments

Expansion of `!box.box` Results

If an operation has a `!box.box` result, it is rewritten to replace the result with multiple results, each corresponding to one of the box's individual elements. Immediately after the rewritten operation, the individual results are packed into a `!box.box` of the original result type.

Example:

```
// !box.box result:
%res = "dummy.op"() -> (!mybox)
```

is transformed into:

```
// !box.box result was replaced by individual entries:
%res0, %res1_0, %res1_1 = "dummy.op"() : () -> (index, f64, f32)
// Reassemble entries back into original !box.box value:
%res = "box.pack"(%res0, %res1_0, %res1_1) : (index, f64, f32) -> !mybox
```

Listing 8.11: Expansion of !box.box results

Combining Expansion

It is very common for the operand types, region argument types, and return types of an operation to be linked. Moreover, the operand types of a region's terminator operation are frequently connected to the result types of their parent operation or to the region argument types of other regions. In the presence of such dependencies, all linked !box.box values and types must be expanded together in a single rewriting step.

Example:

```
// Operand types, region argument types, return types, and yielded types are all linked:
%res = "dummy.op"(%box){
  ^1{%arg : !mybox}:
  "dummy.yield"(%arg) : (!mybox) -> ()
} : (!mybox) -> (!mybox)
```

is transformed into:

```
// Perform expansion of all linked elements together:
// Expand operand of "dummy.op":
%val0, %val1_0, %val1_1 = "box.unpack" (%box) : (!mybox) -> (index, f64, f32)
%res0, %res1_0, %res1_1 = "dummy.op"(%val0, %val1_0, %val1_1){
  ^1{%arg0 : index, %arg1_0 : f64, %arg1_1 : f32}: // Expanded region argument
  // Pack expanded block argument:
  %arg = "box.pack"(%arg0, %arg1_0, %arg1_1) : (index, f64, f32) -> !mybox
  // Expand operand of "dummy.yield":
  %yield0, %yield1_0, %yield1_1 = "box.unpack" (%arg) : (!mybox) -> (index, f64, f32)
  "dummy.yield"(%yield0, %yield1_0, %yield1_1) : (index, f64, f32) -> ()
} : (index, f64, f32) -> (index, f64, f32) // Unpacked results
// Pack expanded result:
%res = "box.pack"(%res0, %res1_0, %res1_1) : (index, f64, f32) -> !mybox
```

Listing 8.12: Combined expansion of operands, block arguments, and results

8.2.3 The box-expand-func Pass

The box-expand-func pass expands all !box.box values and types in func.func and func.call operations. Both operation-types must be rewritten in a single pass to maintain the validity of the IR. Each func.call operation is rewritten by expanding all !box.box operands

and results. Meanwhile, each `func.func` operation is rewritten by expanding its `!box.」box` region arguments and the `!box.box` operands of its `func.return` operation. Because a `func.func` operation's function-type is linked to its region argument and `func.return` operand types, the function-type is also updated.

Example:

```
func.func @foo(%arg: !mybox) -> !mybox {
  >> Modify %arg -> %modified_box
  return %modified_box : !mybox
}

%returned_box = func.call @foo(%box) : (!mybox) -> !mybox
```

is transformed into:

```
// Expanded func.func:
func.func @foo(%arg0 : index, %arg1_0 : f64, %arg1_1 : f32) ->  (index, f64, f32)
{
  %arg = "box.pack"(%arg0, %arg1_0, %arg1_1) : (index, f64, f32) -> !mybox
  >> Modify %arg -> %modified_box
  %ret0, %ret1_0, %ret1_1 = "box.unpack" (%modified_box) : (!mybox) -> (index, f64, f32)
  return %ret0, %ret1_0, %ret1_1
}

// Expanded func.call:
%val0, %val1_0, %val1_1 = "box.unpack" (%box) : (!mybox) -> (index, f64, f32)
%res0, %res1_0, %res1_1 = func.call @foo(%val0, %val1_0, %val1_1)
                                      : (index, f64, f32) -> (index, f64, f32)
%returned_box = "box.pack"(%res0, %res1_0, %res1_1) : (index, f64, f32) -> !mybox
```

Listing 8.13: Expansion of `func.func` and `func.call` via `box-expand-func`

8.2.4 The `box-expand-scf-if` Pass

The `box-expand-scf-if` pass expands all `!box.box` types and values in `scf.if` operations. Each operation is rewritten by simultaneously expanding its `!box.box` results and the box operands of its `scf.yield` operations.

Example:

```
%res = scf.if %condition -> (!mybox) {
  >> Compute %true_box
  scf.yield %true_box : !mybox
} else {
  >> Compute %false_box
  scf.yield %false_box : !mybox
}
```

is transformed into:

```
%res0, %res1_0, %res1_1 = scf.if %condition -> (index, f64, f32) {
  >> Compute %true_box
  // Expand yield operand:
  %yield0, %yield1_0, %yield1_1 = "box.unpack" (%true_box) : (!mybox) -> (index, f64, f32)
  scf.yield %yield0, %yield1_0, %yield1_1 : (index, f64, f32)
} else {
  >> Compute %false_box
  // Expand yield operand:
  %yield0, %yield1_0, %yield1_1 = "box.unpack" (%false_box) : (!mybox) -> (index, f64, f32)
  scf.yield %yield0, %yield1_0, %yield1_1 : (index, f64, f32)
}
// Pack expanded result:
%res = "box.pack"(%res0, %res1_0, %res1_1) : (index, f64, f32) -> !mybox
```

Listing 8.14: Expansion of `scf.if` via `box-expand-scf-if`

8.2.5 The `box-expand-scf-for` Pass

The `box-expand-scf-for` pass expands all `!box.box` types and values in `scf.for` operations. Each operation is rewritten by expanding its `!box.box` `iter_args` operands, while simultaneously expanding all linked region arguments and results.

Example:

```
%res = scf.for (%iteration) = (%zero) to (%stop) step (%one)
               iter_args(%arg = %box) -> (!mybox) {
  >> Modify %arg -> %modified_box
  scf.yield %modified_box : !mybox
}
```

is transformed into:

```
// Expand iter_args argument:
%val0, %val1_0, %val1_1 = "box.unpack" (%box) : (!mybox) -> (index, f64, f32)
%res0, %res1_0, %res1_1 = scf.for (%iteration) = (%zero) to %(stop) step (%one)
                          iter_args(%arg0 = %val0, %arg1_0 = %val1_0, %arg1_1 = %val1_1)
                          -> (index, f64, f32) {
  // Pack expanded block argument:
  %arg = "box.pack"(%arg0, %arg1_0, %arg1_1) : (index, f64, f32) -> !mybox
  >> Modify %arg -> %modified_box
  // Expand yield operand:
  %yld0, %yld1_0, %yld1_1 = "box.unpack" (%modified_box) : (!mybox) -> (index, f64, f32)
  scf.yield %yld0, %yld1_0, %yld1_1 : (index, f64, f32)
}
// Pack expanded result:
%res = "box.pack"(%res0, %res1_0, %res1_1) : (index, f64, f32) -> !mybox
```

Listing 8.15: Expansion of `scf.for` via `box-expand-scf-for`

8.2.6 The `box-expand-box-ops` Pass

The `box-expand-box-ops` pass rewrites all `box.insert`, `box.pack`, `box.extract`, and `box.unpack` operations to `box.insert` and `box.extract` operations only handling individual elements. It must be executed last, after all other expansion passes, as those typically introduce `box.unpack` and `box.pack` operations.

Each `box.insert` operation that inserts a `!box.box` value is replaced by a `box.unpack` operation that disassembles the inserted `!box.box`, followed by a series of `box.insert` operations that insert each entry individually. Conversely, each `box.extract` operation that extracts a `!box.box` entry is replaced by multiple `box.extract` operations that extract the

individual values of the extracted box, followed by a box.pack operation that reassembles them into a box of the original return type.

Example:

```
%res = "box.insert"(%box, %inner_box) <{"indices" = [1 : i64]}>
                    : (!mybox, !box.box<[f64, f32]>) -> !mybox
```

is transformed into:

```
%val1_0, %val1_1 = "box.unpack" (%innerbox) : (!box.box<[f64, f32]>) -> (f64, f32)
%res0 = "box.insert"(%box, %val1_0) <{"indices" = [1 : i64, 0 : i64]}>
                     : (!mybox, f64) -> !mybox
%res = "box.insert"(%box, %val1_1) <{"indices" = [1 : i64, 1 : i64]}>
                    : (!mybox, f32) -> !mybox
```

Listing 8.16: Expansion box.insert via box-expand-box-ops

Each box.pack operation is replaced by a box.undef operation, followed by a series of box.insert operations that insert the individual entries into the newly created !box.box. Conversely, each box.unpack operation is replaced by multiple box.extract operations that extract all entries individually.

Example:

```
%val0, %val1_0, %val1_1 = "box.unpack" (%box) : (!mybox) -> (index, f64, f32)
```

is transformed into:

```
%val0 = "box.extract"(%box) <{"indices" = [ 0 : i64]}> : !mybox -> index
%val1_0 = "box.extract"(%box) <{"indices" = [1 : i64, 0 : i64]}>  : !mybox -> f64
%val1_1 = "box.extract"(%box) <{"indices" = [1 : i64, 1 : i64]}>  : !mybox -> f32
```

Listing 8.17: Expansion box.unpack via box-expand-box-ops

This pass is the final expansion pass. After its execution, only box.undef, as well as box.insert and box.extract operations handling individual values, remain in the IR.

8.3 The memwrap-eliminate-redundant-store-ops Pass

The memwrap-eliminate-redundant-store-ops pass removes memwrap.store and memwrap.store_vector operations that store values at the same memory locations they were previously loaded from. Operations from the memwrap dialect are only introduced when lowering particles_dist.load_particle and particles_dist.store_particle operations. These, in turn, are only emitted during the conversion of particles_dist.foreach and particles_dist.for_all_neighbors operations (See Section 7.3.2). Great care was taken to avoid all data dependencies between particles within the foreach and for_all_neighbors operations. In the absence of data dependencies, if a value is stored back at the same memory location it was loaded from, the store operation has no effect and can be removed. The memwrap-eliminate-redundant-store-ops pass exploits this fact and removes all such memwrap load operations. Any resulting memwrap.load or memwrap.load_vector operations with unused results are eliminated during a subsequent DCE pass.

Example:

```
// (Types have been removed to improve readability)
// Converted particles_dist.load_particle:
%pos = "memwrap.load_vector"(%pos_memref, %particle_index) <{"memref_transposed"}>
%velocity = "memwrap.load_vector"(%velocity_memref, %particle_index)
%force = "memwrap.load_vector"(%force_memref, %particle_index)

// Modify velocity and position:
%new_velocity = math.fma %half_delta_t, %force, %velocity
%new_pos = math.fma %delta_t, %new_velocity, %pos

// Converted particles_dist.store_particle:
"memwrap.store_vector"(%new_pos, %pos_memref, %particle_index) <{"memref_transposed"}>
"memwrap.store_vector"(%new_velocity, %velocity_memref, %particle_index)
// Stores unmodified force back where it was loaded from:
"memwrap.store_vector"(%force, %force_memref, %particle_index)
```

is transformed into:

```
%pos = "memwrap.load_vector"(%pos_memref, %particle_index) <{"memref_transposed"}>
%velocity = "memwrap.load_vector"(%velocity_memref, %particle_index)
%force = "memwrap.load_vector"(%force_memref, %particle_index)

// Modify velocity and position:
%new_velocity = math.fma %half_delta_t, %force, %velocity
%new_pos = math.fma %delta_t, %new_velocity, %pos

"memwrap.store_vector"(%new_pos, %pos_memref, %particle_index) <{"memref_transposed"}>
"memwrap.store_vector"(%new_velocity, %velocity_memref, %particle_index)
// Store operation for force was eliminated
```

Listing 8.18: Elimination of a redundant `memwrap.store_vector` operation during the `memwrap-eliminate-redundant-store-ops` pass

This simple yet effective pass demonstrates the power that results from the use of the box dialect. If structs had been used for lowering the `!particles_dist.particle` type, this pass would need to be significantly more complex, as it would require analyzing the modification graph of the particle struct to determine which entries are never modified and therefore must not be stored.

8.4 The `convert-memwrap-to-memref-vector` Pass

The `convert-memwrap-to-memref-vector` pass converts all `memwrap` operations to operations from the `memref` and `vector` dialects. Each `memwrap.load` operation is converted one-to-one to a `memref.load` operation. Similarly, each `memwrap.store` operation is lowered to a `memref.store`. All `memwrap.load_vector` operations without the `memref_transposed` property are converted into `vector.load` operations, while all `memwrap.store_vector` without `memref_transposed` are lowered to `vector.store` operations.

Initially, `memwrap.load_vector` and `memwrap.store_vector` operations with `memref_transposed` were converted into `vector.transfer_read` and `vector.transfer_write` operations, respectively. However, the integrated `convert-vector-to-scf` pass, which lowers the `vector.transfer_read` and `vector.transfer_write` operations, introduced `memref.alloca` operations, which frequently caused out-of-memory errors. To prevent this issue, each `memwrap.load_vector` operation with the `memref_transposed` property is instead converted into a series of `memref.load` operations, each loading a single element

of the vector. Afterward, the loaded values are assembled into the resulting `vector` value. The `memwrap.store_vector` operation reverses this process, disassembling a vector into its elements and then storing each scalar individually.

Example:

```
// %pos_memref = | (x0 [x1] x2 x3  ...) |
//               | (y0 [y1] y2 y3  ...) |
//               | (z0 [z1] z2 z3  ...) |

// Extract (x1, y1, z1):
%pos = "memwrap.load_vector"(%pos_memref, %one) <{"memref_transposed"}>
                            : (memref<3x?xf64>, index) -> vector<3xf64>
```

is transformed into:

```
// %pos_memref = | (x0 [x1] x2 x3  ...) |
//               | (y0 [y1] y2 y3  ...) |
//               | (z0 [z1] z2 z3  ...) |

// Extract x1, y1, and z1 individually
%x = memref.load %pos_memref[%zero, %one] : memref<3x?xf64>
%y = memref.load %pos_memref[%one, %one] : memref<3x?xf64>
%z = memref.load %pos_memref[%two, %one] : memref<3x?xf64>

// Insert all elements into one vector
%zero_vector = arith.constant dense<0.0> : vector<3xf64>
%pos0 = vector.insertelement %x, %zero_vector[%zero : index] : vector<3xf64>
%pos1 = vector.insertelement %y, %pos0[%one : index] : vector<3xf64>
%pos = vector.insertelement %z, %pos1[%two : index] : vector<3xf64>
```

Listing 8.19: Conversion of `memwrap.load_vector` with `memref_transposed` via `convert-memwrap-to-memref-vector`

This pass marks the end of the `particles` lowering pipeline. Afterward, only standard dialects remain in the IR.

9 Evaluation

9.1 Benchmarking Setup

To evaluate the performance of the code generated by the compilation pipeline presented in this work, Molecular Dynamics [26] was implemented in MLIR using the `particles` dialect and compared against C++ implementations leveraging OpenFPM. For this purpose, two MD examples, one for CPUs and one for CUDA GPUs, provided by OpenFPM were adapted to closely mirror the MLIR code, which itself was based on these examples. Appendix A.1 shows the MLIR code used for both hardware targets. Mirroring involved using the same `vector_dist` memory layout, identical cell list implementations, and placing the communication and update operations at the same locations in the C++ code as they were placed automatically in the MLIR code. Since the automatic insertion of communication and update operations resulted in an optimal placement, this did not decrease the performance of the C++ implementations. Additionally, all loop fusion optimizations were disabled while lowering the MLIR code.

To verify that the C++ and MLIR implementations match, particle positions and properties were compared after each batch of 100 time steps. To this end, for both CPUs and CUDA GPUs, the MLIR and C++ implementations were combined into a single executable that executed batches using both implementations in an alternating fashion. Across all variants, the third batch was the earliest when the maximum difference of a field value exceeded 1e-3. Separate tests indicate that these deviations may stem from differences in floating-point behavior between the MLIR-generated code and the code generated by the C++ compilers. Exactly matching the results of the computations proved challenging and was ultimately abandoned. In addition to directly matching the fields, the energy of both simulations was calculated and compared after each batch.

The original plan was to perform the benchmarks in an HPC environment. However, even after several days of troubleshooting and experimenting with numerous compiler combinations, it was not possible to get the project fully up and running. This highlights one of the greatest limitations of this work: its large number of dependencies. In particular, OpenFPM has many dependencies of its own and appears to require very specific compiler versions.

Due to these issues, the benchmarks were conducted within the development environment, using a Docker container on a single machine. The system was equipped with an AMD

Ryzen Threadripper 3960X (24 cores/48 threads), 64GB of RAM, and an NVIDIA GeForce RTX 3090 with 24GB of VRAM. Since the focus of the evaluation is not the absolute performance of the MLIR implementation but the relative differences between implementations, all of which were executed within the same environment, the comparisons still provide valuable insights into possible future improvements by highlighting specific limitations in our current implementation. Identifying these limitations is crucial for improving the performance of future versions through better code generation and specific optimization strategies.

9.2 Benchmarking Molecular Dynamics on a Multi-Core CPU

The CPU executable, combining both the MLIR and C++ implementation, was built twice: once using the GNU Compiler Collection (GCC) (version 9.4.0) and once using Clang (version 18.1.8). For the GCC variant, the LLVM IR output from lowering pipeline was first translated to assembly using `llc`, then compiled with `g++` to an object file, and finally linked with the rest of the program also compiled with `g++`. In the Clang variant, the LLVM IR code was directly compiled to an object file using `clang++`, which was then linked with the rest of the code compiled with the same compiler. This approach resulted in two executables and four simulation variants: a binary combining both the MLIR and C++ reference implementation, compiled and linked with either the GCC or Clang toolkit.

Each simulation consisted of 1000 time steps, divided into ten batches with 100 time steps each. For every parameter (number of particles or number of processes) three simulation runs were conducted. The first two batches of each run were excluded from the results, as they consistently exhibited significantly lower runtimes across all simulation versions and parameter combinations. Including them would have rendered the error bars meaningless. Particles were initially arranged on a grid. We suspect that this initial grid layout caused a warm-up phase, and once the particles had moved sufficiently, the benefits of the grid layout were lost and the performance stabilized. To test this hypothesis, three simulations were run with particle positions initialized at random. No warm-up phase was observed in this case, supporting the hypothesis.

Figure 9.1 shows the average time required to complete a batch for different particle counts, while Figure 9.2 shows the average batch time across different process counts. Given three runs and eight included batches per run, each bar represents 24 measurements. The plots on the left display absolute measurements, while the plots on the right show the average batch runtimes normalized against the C++ reference implementation compiled with GCC. All implementations scaled well with both the number of particles and the number of processes.

Across all benchmark cases, the fastest variant was the C++ reference implementation compiled with GCC, with all other variants taking 0.5%–5.5% longer. The second-fastest implementation was the reference compiled using Clang, which was on average 0.5%–2% slower than the one compiled with GCC. The MLIR implementation compiled with Clang came in third, being 0.5%–2.5% slower than the GCC reference. The slowest implementation was always the MLIR version compiled with GCC, which was on average between 4.5% and 5.5% slower than the reference.

Except for 1000 particles using a single process, the normalized performance gap between the GCC reference implementation and both MLIR variants widened with decreased per-process particle counts. We hypothesized that there may be a constant overhead involved

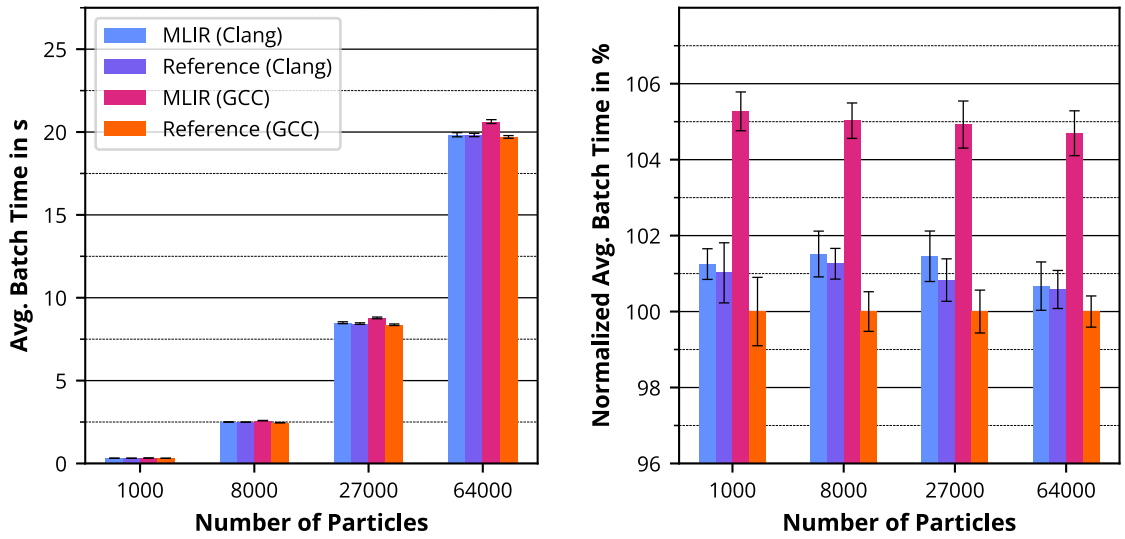

Figure 9.1: Performance comparison on CPU between C++ reference and MLIR implementations for different particle counts. Left: Average runtime required to complete a batch consisting of 100 time steps, with 10 batches per run across three simulation runs. Right: Average batch runtime normalized against the GCC reference implementation. Error bars correspond to the standard deviation. The first two batches of each run were excluded to ensure meaningful error bars.

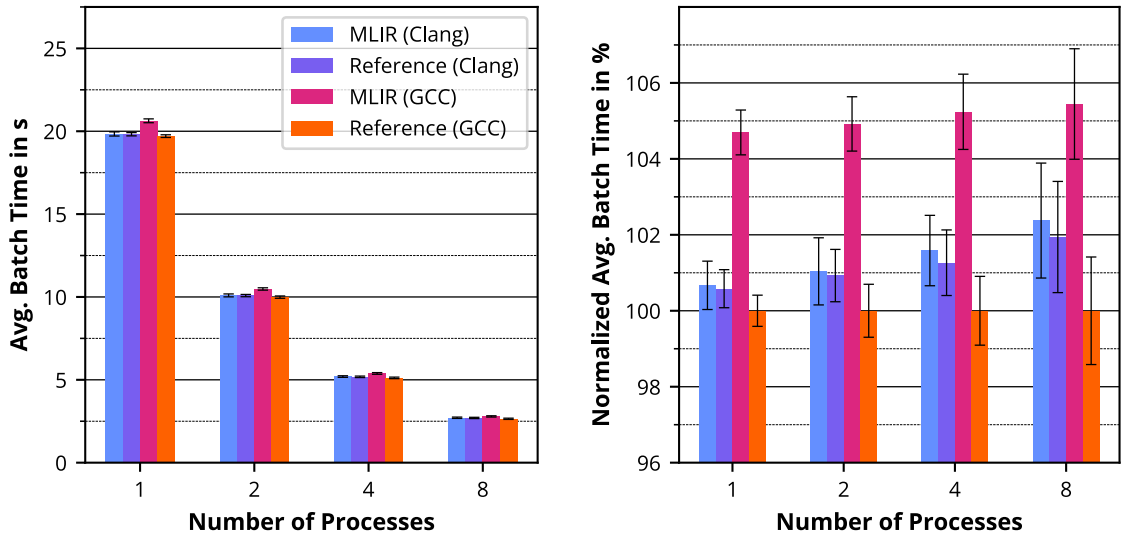

Figure 9.2: Performance comparison on CPU between C++ reference and MLIR implementations for different process counts, executed with 64,000 particles. Left: Average runtime required to complete a batch consisting of 100 time steps, with 10 batches per run across three simulation runs. Right: Average batch runtime normalized against the GCC reference implementation. Error bars correspond to the standard deviation. The first two batches of each run were excluded to ensure meaningful error bars.

in the MLIR implementations, which becomes more impactful when fewer particles are assigned to each process. However, running simulations with only a single particle resulted in differences of less than 1e-3 seconds between all four implementation variants. The observed performance gap therefore is unlikely to stem from the integration of update and communication operations, or from the additionally required `particles_dist.update_internals` operations. Notably, the same widening of the performance gap with decreased per-process particle counts was also observed for the Clang reference implementation.

The causes for the performance differences and the observed trends have not yet been

identified. However, numerous experiments conducted during development and result verification showed that minuscule changes in the `particles_dist.for_all_neighbors` region, and the C++ reference code it mirrors, can have significant performance impacts. We suspect that this is because the interaction code, which dominates the runtime, is very short but executed countless times per batch. Therefore, even small differences, such as a single additional jump in the compiled assembly code, can significantly impact simulation performance. We suspect that this effect, mixed with other factors such as cache misses and branch prediction, can result in performance variability that is hard to attribute to specific causes.

To illustrate this, the check that skips a neighbor if it is not within the cutoff radius during interaction, was temporarily disabled for all implementations. This has the following effects: Everything surrounding the interaction logic remains the same, including the code iterating over all particles and neighbor indices. Because the positions of the particle and its neighbor are required during interaction, this also does not eliminate any load operations. However, the interaction logic now contains at least one less branch, and on average performs 6.5 times the number of calculations (ratio between the volumes of a cube with side length 3 and a sphere with radius 1).

Figure 9.3 shows the results for the adapted simulations, using a single process and varying particle counts. In this scenario, the Clang reference implementation had by far the worst performance, followed by the MLIR implementation compiled with GCC. The MLIR implementation compiled with Clang came in second. Compared to the GCC reference implementation, the performance gaps of all other variants increased from 0.5%–5.5% to 6.5%–14.5%. Furthermore, new trends emerged. For instance, the performance gap between the MLIR implementation compiled with Clang and the one compiled with GCC seemed to decrease for higher particle counts. Notably in this experiment is the large performance gap between both reference implementations. Constant factors, such as Link-Time Optimization (LTO), were ruled out by the experiment using a single particle. The performance gap may be due to differences in the optimization or code generation strategies between the `g++` and `clang++` compilers.

As a conclusion, performance varies widely across different simulation codes and compiler combinations. We suspect that this stems from small differences in the compiled interaction code. Because it is generally very short but executed over and over, small differences can cause significant performance variations. Overall, the generated code performs closely to the reference, suggesting that its quality is already high. However, more examples need to be implemented and benchmarked, and the differences must be investigated further, before drawing any definitive conclusions.

9.3 Benchmarking Molecular Dynamics on a CUDA GPU

To build the CUDA executable, a toolchain combining `clang++`, `llc`, `g++`, and `nvcc` (version V11.3.109.) was employed. Section 3.3 outlines that lowering the MLIR code for CUDA GPUs requires compiling the device code of the runtime source file to LLVM bitcode using `clang++` and then reintroducing the result into the lowering pipeline, where it is then linked with the device code of the MLIR file. The final output of the pipeline was translated to assembly with `llc` and the remainder of the built steps were performed using `g++` and `nvcc`. To enable LTO, device and host code were compiled and linked separately.

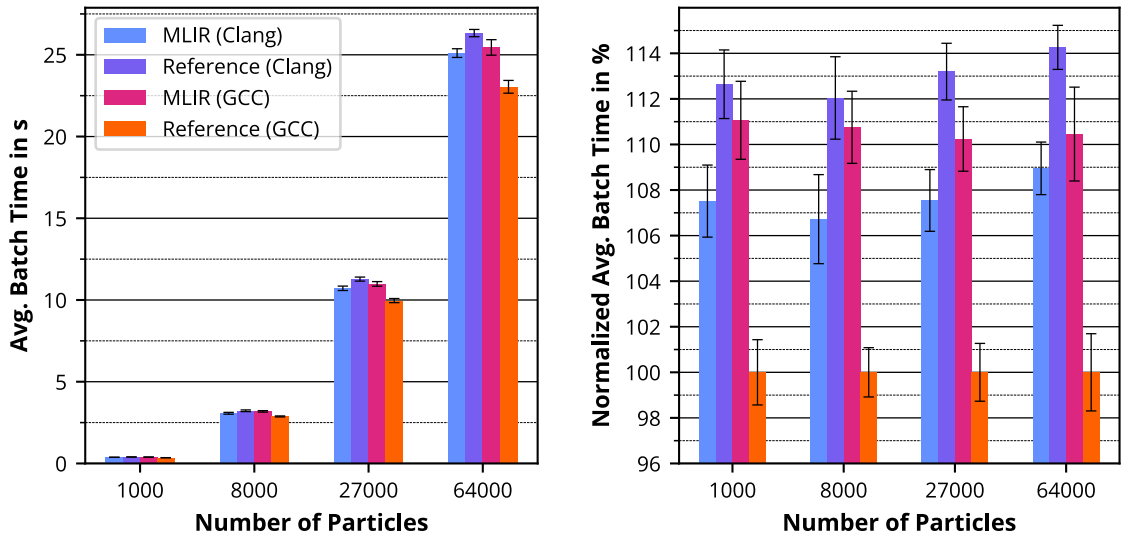

Figure 9.3: Performance comparison on CPU between C++ reference and MLIR implementations for different particle counts, executed with one process. The check that skips a neighbor if it is not within the cutoff radius was disabled. Left: Average runtime required to complete a batch consisting of 100 time steps, with 10 batches per run across three simulation runs. Right: Average batch runtime normalized against the GCC reference implementation. Error bars correspond to the standard deviation.

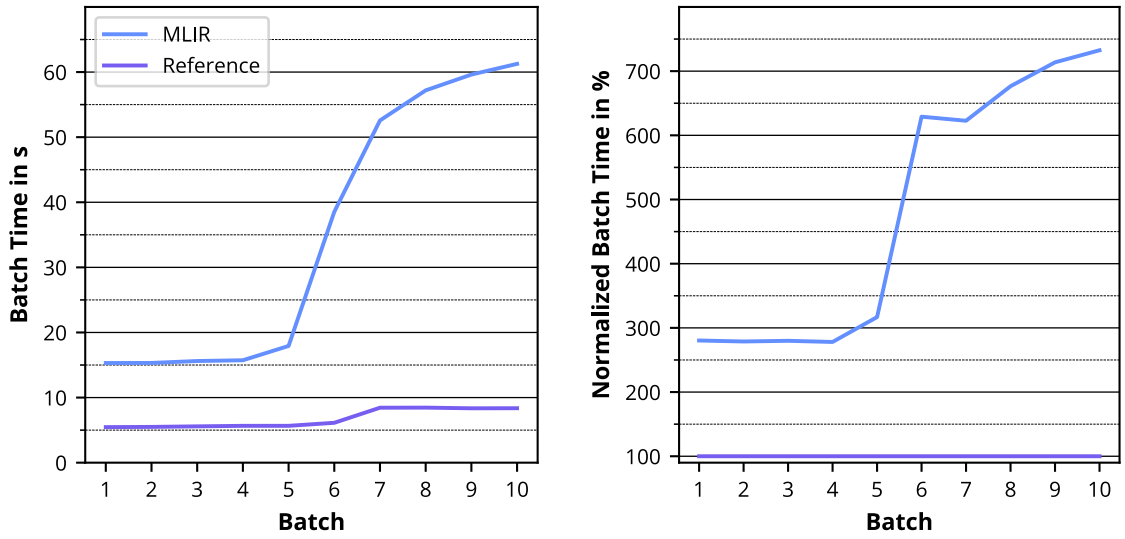

Figure 9.4: CUDA: Deterioration of the MLIR performance throughout a simulation with one million particles compared to the C++ reference implementation. Left: Execution time in seconds per batch of 100 time steps. Right: Batch times normalized against the C++ reference implementation. The MLIR performance rapidly deteriorated throughout the simulation.

Figure 9.4 presents the batch runtimes observed during a single simulation run with one million particles, executing using a single process. Each batch consisted of 100 time steps. The relative runtime of the MLIR implementation increased throughout the simulation. It started at around 300% and rapidly grew to over 700%. While the C++ reference implementation also showed signs of performance deterioration, the impact was not as pronounced.

Numerous checks were performed to investigate the observed performance deterioration. The code was checked multiple times to ensure that the calculations matched. Potential memory leaks were also examined, but no issues were found. While the energy fluctuations

differed slightly between implementations, the values generally stayed within a one percent of each other.

The current hypothesis is that the observed performance degradation stems from the lowering pipeline generating code that is poorly aligned with the Single Instruction, Multiple Threads (SIMT) execution model of CUDA GPUs. In particular, we suspect that lowering the `cell_list.for_all_neighbor_indices` operation results in an excessive number of control flow branches. This is, at least in part, caused by the integration of the `CellList_gpu_ker` and `NN_gpu_it_box` classes through runtime calls. The `get_next` runtime function, for instance, which retrieves the index of the next neighbor, contains an if-else block which could be eliminated through direct integration. The MLIR code iterating over all neighbor indices is also suspected to result in more control flow branches than necessary. Due to the lock-step execution of threads – where all threads must execute the same instruction path – a higher number of control flow branches can significantly degrade performance.

We suspect that at the beginning of a simulation, when particles are still arranged in a grid, threads generally follow the same control flow paths and the performance impact is less pronounced. However, as particles move, the benefits of the initial grid layout diminish, and threads become increasingly distributed across different basic blocks. The more blocks a program uses, the more lock-steps each thread spends being masked. Therefore, programs with more blocks are suspected to deteriorate faster as particles move.

To investigate this hypothesis, three experiments were conducted. First, particles were initialized at random positions. This resulted in an initial runtime of around 108 seconds per batch for the MLIR implementation, and 31 seconds for the C++ implementation. Both values are significantly higher than the initial times observed with a grid layout. Unfortunately, the performance degradation could not be measured, as the random particle layout proved unstable and the parameter controlling the force needed to be tuned down.

In the second experiment, a regular grid layout was used again, but with an additional force multiplier of 1e-5 to slow down particle movements. This resulted in no observable performance deterioration for either implementation. Combined with the results of the first experiment, this suggests that the observed performance degradation is likely caused by particle movements.

In the third experiment, the check that skips a neighbor if it is not within the cutoff radius during the interaction was again disabled. This increases the total interactions that need to be computed by a factor of 6.5, but at the same time reduces the number of control flow branches in the interaction code. Figure 9.5 shows the results of a simulation with one million particles. While a performance deterioration of the MLIR implementation still occurred, it was much less pronounced and more consistent with the C++ baseline. Notably, the performance of the MLIR implementation at the first batch was also over 25% faster without the check than with it. In contrast, the C++ version suffered from a performance decrease of over 40%. These observations reinforce the hypothesis that the observed performance degradation for the MLIR implementation is due to too many control flow branches.

Figure 9.6 shows a performance comparison between the C++ reference and MLIR implementation with varying particle counts using a single process. Due to the observed performance deterioration throughout a simulation, only the mean execution time of the first batch across five runs was measured. Notably, the performance gap increased with higher particle counts, indicating further issues to investigate. Because the benchmarks were performed on a single machine with a single GPU, benchmarks with more than one process were omitted.

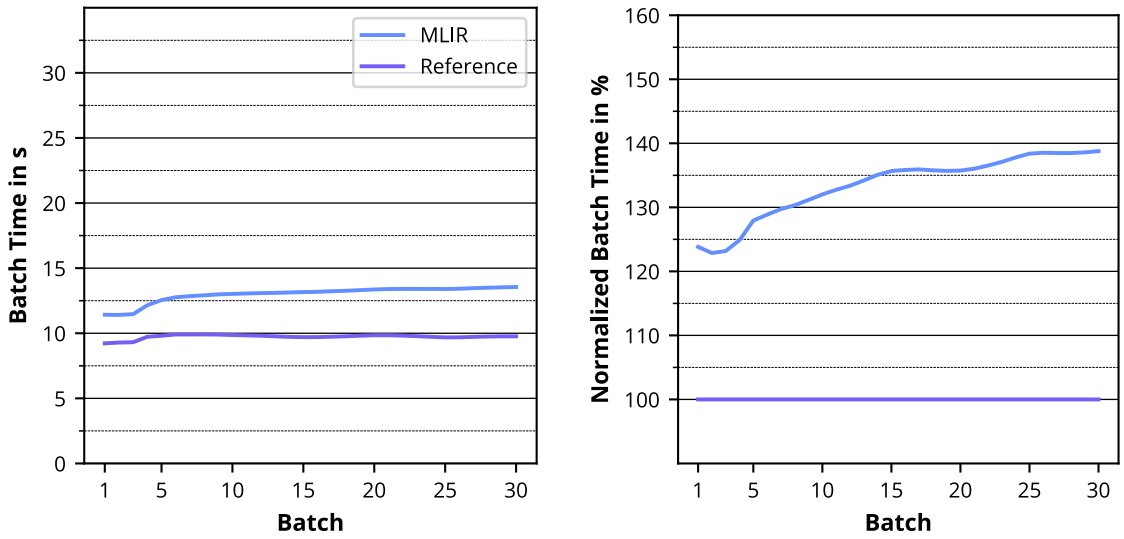

Figure 9.5: CUDA: Deterioration of the MLIR performance throughout a simulation with one million particles compared to the C++ reference implementation. The code that checks whether a neighbor is within the cutoff radius was disabled. Left: Execution time in seconds per batch of 100 time steps. Right: Batch times normalized against the C++ reference implementation. As shown in the plot on the right, the MLIR performance deteriorated throughout the simulation, but much slower than with the vicinity check enabled.

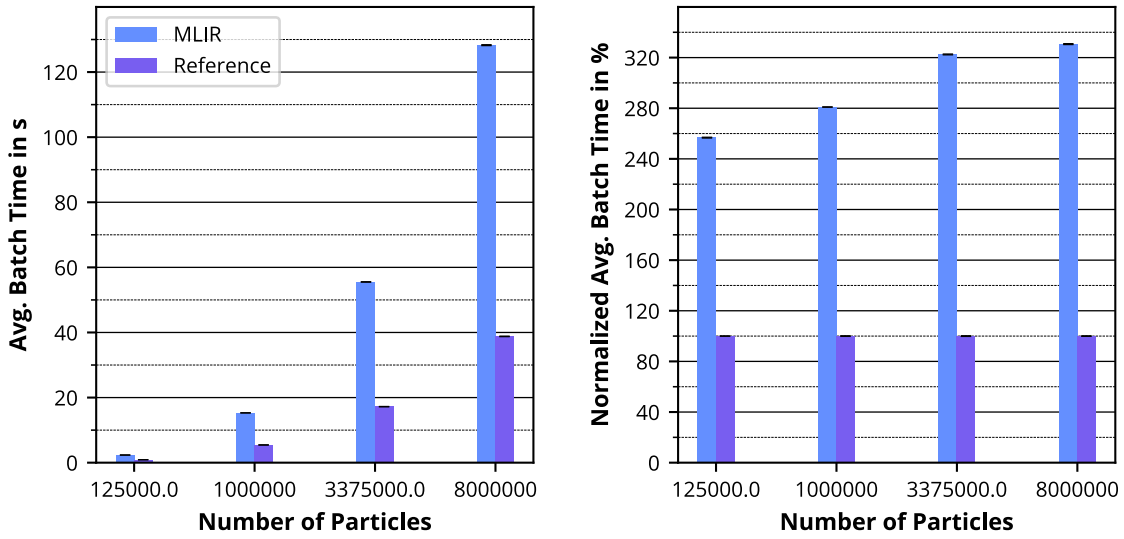

Figure 9.6: CUDA: Performance comparison between the C++ reference and MLIR implementation for different particle counts, executed using one process. Left: Mean execution times of the first batch across five runs. Right: Mean execution times normalized against the C++ reference implementation. Error bars correspond to the standard deviation.

Benchmark results on the CUDA GPU suggest that the generated code is currently of low quality, with multiple issues requiring further investigation and future fixes. Experiments indicate that the number of control flow branches may play a significant role in the observed performance gap and rapid deterioration. However, the current cell lists integration using device runtime calls is only a preliminary solution, with a full integration still on the horizon. Furthermore, the CUDA hardware target is the most recent addition to the lowering pipeline and was implemented in a matter of weeks, including example implementation, integration testing, debugging, and troubleshooting. Towards the end, no time was left to optimize the

code generation. We therefore do not see these results as discouraging, but rather as an insightful snapshot in an ongoing effort.

9.4 Benchmarking the Impact of Loop Fusion

To evaluate the impact of the `particles-dist-fuse-foreach-ops` and `particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops` optimization passes, the MLIR implementations of MD were benchmarked twice, once with loop fusion enabled and once without. Enabling these passes resulted in the fusion of a `particles_dist.for_all_neighbors` and a `particles_dist.foreach` operation. The `for_all_neighbors` operation computed the force exerted on a particle by its neighbor based on their relative positions. The immediately following `foreach` operation updated the velocity of each particle using the force exerted on it. Fusing both operations into one `particles_dist.for_all_neighbors` operation eliminated one load operation per particle: the load of the force within the `foreach` operation that was fused into the `for_all_neighbors`.

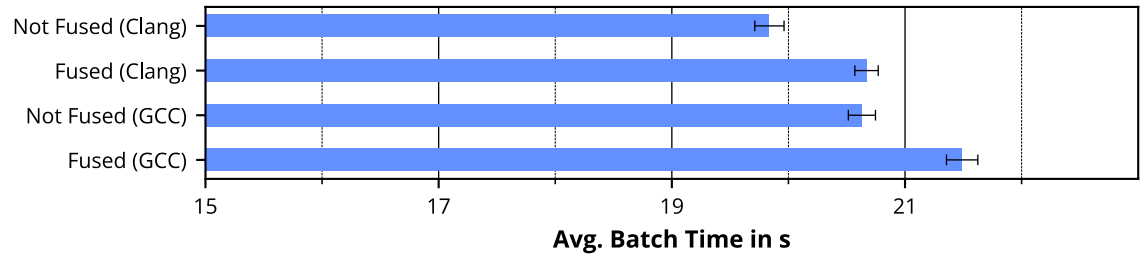

Figure 9.7: CPU: Performance comparison between MLIR builds with and without loop fusion. Each bar shows the average execution time for a batch of 100 time steps, averaged over three simulation runs with ten batches each. Simulations were conducted using one process and 64,000 particles. The first two batches of each simulation were excluded to keep the error bars meaningful (see Section 9.2).

Figure 9.7 presents the results of enabling loop fusion for the CPU. Simulations were conducted with 64,000 particles using one process. Contrary to expectations, enabling loop fusion led to performance degradation rather than improvement. We suspect that, even though the fusion of the `foreach` operation into the `for_all_neighbors` operation eliminated a loop and one load operation per particle, the original `foreach` operation was very well suited for optimizations such as unrolling, pipelining, and speculative execution. Furthermore, in its original form, the `foreach` operation may have benefitted more from the memory locality between the fields of consecutively visited particles. The inlined `foreach` code may no longer benefit from all these factors. Also, because the velocity is loaded early in the fused loop but only used after neighbor interactions, it possibly increased the chance of cache misses.

Figure 9.8 shows the results for the CUDA GPU. The benchmarks were conducted with one million particles using one process. Because of the performance deterioration observed in Section 9.3, the average runtime of the first batch across five runs is displayed. Loop fusion resulted in a performance increase of over 45%. It may be tempting to attribute this improvement to the elimination of the `foreach` operation, or to the removal of the load operation. However, separate measurements show that the contribution of the removed `foreach` operation to the total batch time is less than 0.02 seconds and therefore significantly smaller than the observed performance gain. We see no clear explanation as to why this

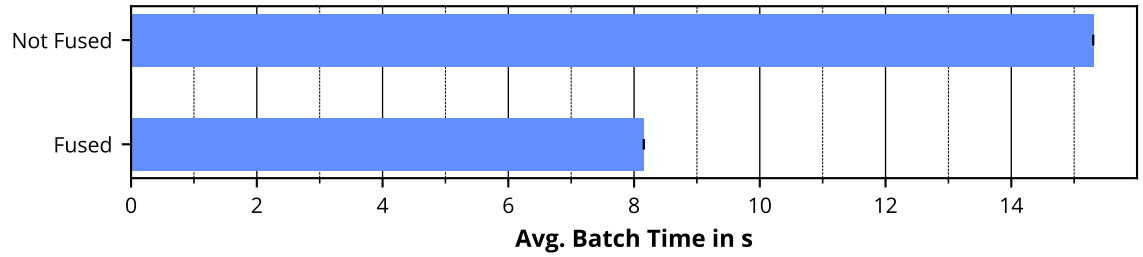

Figure 9.8: CUDA: Performance comparison between MLIR builds with and without loop fusion. Each bar shows the average execution time for the first batch of 100 time steps, averaged over five simulation runs. Simulations were conducted using one process and one million particles.

performance increase was observed. However, we suspect that it plays into the issues observed in Section 9.3.

9.5 Evaluation of Benchmark Results

The benchmark results presented in this chapter must be interpreted within the proper context. For one, the comparisons were made against OpenFPM, a highly optimized, state-of-the-art framework for particle simulations. As such, the performance baseline was exceptionally high from the outset. We consider achieving results close to the reference implementations as a significant accomplishment, and as an indication that the generated code is optimized. Thus, we view the CPU results, while not definitive, as very promising. More examples must be implemented and evaluated to gain a better understanding.

For CUDA GPUs, the benchmarks indicate that the generated code currently is of low quality. We suggest improving the code generation and cell list integration before any further in-depth benchmarks.

While the benchmark results on the impact of loop fusion look promising for CUDA GPUs, it is still too early to draw any meaningful conclusion, since the causes of the performance gains remain unclear and may be entangled with the issues identified for the CUDA target.

In summary, the CPU results are encouraging, whereas the CUDA GPU results indicate that there is substantial work to do.

In the future, while investigating the performance differences and the observed degradation on CUDA GPUs, additional profiling tools such perf or NVIDIA Nsight should be used to gain deeper insights. Other metrics, such as the number of basic blocks, cache-misses, and memory accesses should also be included in the comparisons.

10 Discussion

10.1 Summary

This work presents `particles`, an MLIR dialect designed for particles-only simulations. It is paired with a custom lowering infrastructure targeting both multi-core CPUs and CUDA GPUs, and supports DMP by integrating OpenFPM.

The `particles` dialect is designed to be target-agnostic and general-purpose, and thus does not include any DMP or OpenFPM specific elements. A notable feature of the `particles` dialect is the use of fake values semantics for its `!particles.set` type. These fake value semantics behave like real value semantics but use memory semantics under the hood. Their use facilitates the construction of modification graphs via def-use relationships. Furthermore, they enable maintaining transient state and the buffering of internal data.

The `particles_dist` dialect adapts `particles` to DMP while also integrating OpenFPM. As a specialization dialect, it adapts all IR construct definitions from the `particles` dialect, extends those where necessary, and introduces additional constructs of its own. Because both dialects share a significant portion of their IR structure, both extend the same abstract base dialect `particles_base`. A key early step in the lowering pipeline is the `particles_dist` specialization pass, which involves replacing all `particles` constructs with their `particles_dist` counterparts.

Following specialization, target-specific transformations are applied. These include the fusion of `particles_dist.foreach` and `particles_dist.for_all_neighbors` operations, as well as the automatic insertion of update and communication operations required for DMP. To guide the placement of these operations, a specialized DFA variant, called staleness analysis, was developed. This analysis determines where in the program each data structure is definitely or potentially stale. Combined with liveness analysis, it enables well-informed placement decisions. Where compile-time analysis is insufficient, staleness tracking is used to defer update and communication decisions to the execution time. To enable DFA, the `particles.loop` operation and its `particles_dist` counterparts leverage the fake value semantics employed by the `!particles.set` type. By ensuring that all modification graphs over `!particles.set` values form a DAG with a single-source and a single single-sink, their edges align with the control flow and the graphs can thus be used for DFA.

To facilitate optimized lowering of the `particles_dist` dialect, several support dialects were developed. Each type of neighbor list is encapsulated in a companion dialect

of particles_dist. Two such dialects currently exist: the local_domain dialect, which considers all particles in a subdomain as neighbors, and the cell_list dialect, which integrates cell lists. Because both dialects share the same syntax, both extend the same abstract base dialect neighbor_list.

To support the lowering of complex types and to enable 1-to-N conversion, the box dialect was created. At the core of this dialect lies the !box.box type, which closely resembles the !llvm.struct type but without the strict element type constraints.

To eliminate redundant stores of particle values, the memwrap dialect was developed. It facilitates the removal of store operations that store values at the same location they were loaded from. In the guaranteed absence of data dependencies, such stores can safely be removed.

Before lowering the particles_dist dialect, the runtime is generated, forming a bridge between MLIR and C++. The runtime defines simple MLIR-compatible structs that enable exchange of data between MLIR and C++. Additionally, it provides non-member functions that enable the invocation of member functions and the extraction of data from C++ objects. In particular, these runtime functions enable the integration of update and communication operations implemented by OpenFPM.

Because the particles_dist and its neighbor list companion dialects are so closely intertwined, they are lowered in a two-step tick-tock process. In this process, all operations are lowered first, followed by the types, alternating between lowering the constructs of the leader dialects and its companions. The hardware target, either CPUs or CUDA GPUs, has little impact on the conversion patterns for particles_dist operations. Only the scf.parallel operations, generated from lowering particles_dist.foreach operations, are further modified when targeting CUDA GPUs by mapping their iterations to threads and blocks.

On CPUs, the cell_list dialect fully integrates the CellList_gen class from OpenFPM, directly accessing its internal data. For CUDA GPUs, a full integration was not possible within the time constraints. Instead, the CellList_gpu class is integrated using device runtime calls that rely on the NN_gpu_it_box iterator class.

During the lowering of the particles_dist dialect and its companion neighbor list dialects, constructs from the box and the memwrap dialects are introduced. Each !box.box value that is stored in or loaded from a !box.storage, is converted into an !llvm.struct during storage and transfer. Afterward, all !box.box types and values are eliminated via a series of expansion passes and a final shortcutting step, replacing all !box.box types and values with their constituent elements. Finally, redundant memwrap load operations are eliminated, and the remaining memwrap operations are lowered.

Deploying the project on an HPC system for benchmarking proved to be challenging because of build issues. Due to time restrains, the benchmarks were performed on a single machine within a Docker container. The quality of the generated code for both hardware targets was evaluated by comparing their performances to equivalent C++ reference implementations of MD.

On the CPU, relative performances varied widely between different interaction codes and compiler combinations. Overall, benchmarks suggest that the MLIR implementation comes close to the consistently fasted reference implementation compiled with GCC, suggesting high code quality.

On CUDA GPUs, the MLIR implementation exhibited rapid performance deterioration over the course of an ongoing simulation run relative to the reference implementation compiled

with nvcc. The current hypothesis is that this is due to the lack of a full integration of the cell list, and the presence of too many control flow branches in the interaction code. Simulations with higher particles counts exhibited higher performance gaps. The performance of the MLIR implementation did not come close to the reference. Overall, the results indicate that the generated code has room for improvement and that further work is required to enhance its performance and robustness.

To assess the impact of the implemented loop fusion transformations, the MLIR code was compiled twice: once with loop fusion disabled and once with it enabled. On the CPU, enabling loop fusion significantly decreased performance. In contrast, on CUDA GPUs, loop fusion resulted in a performance increase over 45%. However, the causes of the performance gain could not be identified and are suspected to be entangled with existing issues with the CUDA target.

10.2 Conclusion

In conclusion, the introduced `particles` dialect, together with its sophisticated lowering pipeline, represents a substantial step forward in accelerating particle simulations through MLIR. The generated code achieves performance close to the reference implementation on CPUs, indicating a high code quality. For CUDA GPUs, benchmarks suggest substantial room for improvement. The range of introduced capabilities presents a solid foundation for future work. The ability to target both, CPUs and CUDA GPUs, combined with DMP, from a nearly target-agnostic IR is a major accomplishment. Furthermore, the automatic placement of update and communication operations based on careful code analysis represents a significant improvement over the simple placement rules used by OpenPME.

In addition to widely opening the door for future research on particle simulation using MLIR, this work also advances the field of multi-level IR engineering by introducing a range of concepts and design patterns that can guide future MLIR engineers. The concepts of dialect inheritance, dialect specialization, companion dialects, fake value semantics, and single-source single-sink modification graphs are not specific to particle simulations and can be adopted by any compiler engineer facing similar challenges. In particular, the `box` dialect is highly versatile and applicable to almost any situation where complex types must be lowered to aggregate types. Finally, this work also serves as an instructive case study for integrating MLIR with existing C++ frameworks.

10.3 Limitations and Future Work

Placement of Update and Communication Operations The placement algorithms described in Section 5.2 do not always yield optimal placements. For example, if an operation within a branch makes a data structure stale, and that data structure is required fresh right after the `scf.if` operation, the ideal placement of the update operation would be directly after the stale-inducing operation within the branch. However, the current implementation inserts a `particles_dist.maybe_X` operation after the branch, in front of the operation that requires the data structure to be fresh. Future versions of the placement algorithms could benefit from greater foresight by making more extensive use of liveness analysis to determine better placements.

Furthermore, the placement algorithm inserting ghost get operations relies heavily on `particles_dist.maybe_ghost_get` operations to work around the visiting order issue. In the future, this algorithm could, through the use of more sophisticated analysis techniques, eliminate the issue of visiting order without deferring the majority of the decision to the execution time.

To assess the quality of the automatic placements, a qualitative analysis should be conducted. Test cases can be constructed using the `particles_dist.call` operation and its associated `updates`, `requires_up_to_date`, `makes_stale`, and `uses_staleness_flags` properties. The placement passes should then be applied to these test cases, and the results analyzed. Possible evaluation methods could be comparing the automatic placements to optimal placements or those performed manually by a human.

Storing and Loading Particle Values Eliminating redundant stores of particle values is limited to compile-time decision-making. The `memwrap` dialect and its elimination pass fail in scenarios where runtime-information is necessary to make optimal store decisions. For example, even if a field is only modified under specific conditions (e.g., for certain particle types) the value for the field is always stored, regardless of whether it was modified. An implementation utilizing runtime flags, similar to the staleness flags used for the `!particles_dist.set` type, could address this limitation by tracking whether a field value has been modified. Using these flags, the decision to store a field can be deferred to the execution time when an optimal decision is not possible during compile-time.

A runtime flag approach could benefit the loads as well. Currently, any field potentially used within a `particles_dist.foreach` or `particles_dist.for_all_neighbors` region is loaded at the start of the region. Consequently, if a field is only required in specific cases (e.g., for particles within the interaction radius), it is still always loaded. Runtime flags tracking whether a field has already loaded would enable on-demand loading.

How this change would impact performance must be analyzed before proceeding with the implementation. This is especially important for the CUDA GPU target, where the excessive number of control flow branches is already suspected to be the main cause of current performance issues.

Dependency Issues A major limitation of this work is its extensive set of dependencies, the negative impact of which was observed during benchmarking (Chapter 9). The project depends on OpenFPM version 5.0.0, which at the time of writing is the newest available version on GitHub. This version is only compatible with CUDA up to version 11.4, which in turn restricts the range of usable Clang and GCC versions. However, even when using supported compiler versions, builds often failed unpredictably. Throughout development, build failures were frequent and many days of development were lost to resolving these issues. Sometimes a dependency of OpenFPM did not build successfully, sometimes OpenFPM failed to build, and at other times, compiling or linking the simulations itself failed where it previously succeeded. Ultimately, only a narrow set of compiler version combinations was found to be able to compile and link everything. Although it is nearly impossible to determine what problems were the result of inexperience and were caused by user error, this fragility still represents a major potential hurdle for all users with a similar level of expertise as the author of this work.

The restriction to a small set of compiler version combinations is particularly problematic on HPC systems, where the user is not in full control over the environment. Combined with the requirements of other dependencies, the intersection of available compilers and supported compilers can quickly become empty. The long-term goal of making particle simulations more accessible via a DSL or a problem-solving environment cannot succeed if building the toolchain requires several days of troubleshooting and the help of an expert. Reducing the number of build issues, for example by providing dependable build scripts, reducing the number of dependencies, or via extensive documentation, is critical for the long-term success of this project. To assist new users, a Dockerfile is already included in the project.

Further Benchmarks Benchmarking was constraint by the limited available time and had to be cut short. Additionally, a considerable portion of the available time was spent on investigating and verifying the various issues and unexpected result found during benchmarking. For this reason, only one example, MD, was benchmarked, using only cell list as neighbor list implementation. To further evaluate the performance of the generated code, more benchmarks must be performed, and more examples must be implemented. Currently, the project implements two additional artificial benchmarks.

A "dry run" benchmark was implemented to isolate the performance with which the `particles.foreach` operation iterates over the particles in a subdomain and the `particles.for_all_neighbors` operation iterates over all neighboring pairs. It performs no computations within each region and instead invokes a function containing only a `NOP` assembly instruction to avoid the elimination of the loops. By excluding computations, this benchmark focuses purely on the performance of the iteration mechanics without mixing in arithmetic performance. Although the total execution time is important from a practical perspective, optimizing the performance of the computations within the regions of both operations is arguably the responsibility of built-in MLIR passes and the LLVM compiler.

Another artificial benchmark, "load store", also performs no meaningful computation. Its purpose is to isolate the performance of loading and storing particle values and the reduction of pairwise interaction results. In this benchmark, a `foreach` operation loads the force of each particle, adds 1 to every component (to prevent optimization into a `memcpy`), and stores it as velocity. Similarly, a `for_all_neighbors` operation loads the position of a neighbor and directly yields it as the pair-wise interaction result, storing the reduced result as force.

Although these artificial benchmarks were prepared for the CPU, time constraints prevented the full execution, verification, and evaluation of the results. In the future, these benchmarks should be revisited and also ported to CUDA GPUs. Furthermore, to gain a more accurate view of the practical performance, more real-world examples, such as Particle Strength Exchange or Smoothed-Particle Hydrodynamics should be implemented and benchmarked. Lastly, benchmarks should be performed outside a container, on an HPC system with multiple nodes, using larger particle sets and over longer time periods.

Customizable Memory Layout To reduce development efforts, a fixed memory layout was chosen for the `!particles_dist.set` type and its associated `vector_dist` class. In the future, the memory layout should be made customizable, as it is in OpenFPM. This customizability could be achieved through various mechanisms. One option is to add a

dictionary attribute to the `!particles.set` type to supply the memory layout information for specialization. Alternatively, the memory layout could be attached to the `builtin.module` operation or supplied via a pass option to the specialization pass.

Compatibility with OpenMP The `convert-scf-to-openmp` pass converts `scf.parallel` loops to OpenMP parallel constructs. In the context of this work, it offers an effective method for integrating DMP with SMT. Using it would allow reducing the number of subdomains the computational domain is divided into, while still utilizing multiple CPU cores per node. Unfortunately, the current lowering pipeline seems to be incompatible with this pass, failing with an error.

Optional Results of Iteration Operations The `particles.foreach` operation currently returns only one result: the `!particles.set` value representing the updated particle set. By adding an optional variadic result along with an associated reduction kinds property, as seen in the `neighbor_list.for_all_neighbor_indices` operation, the `foreach` operation could perform additional calculations. For example, it could be used to compute the total kinetic energy or the total impulse of all particles. Similarly, an optional variadic result along with another reduction kinds property could be introduced for the `for_all_neighbors` operation. This could be utilized, for instance, to calculate the total potential energy of the system.

Optimize Generated CUDA Code The code generation by for CUDA GPUs needs to be optimized. A key priority should be the full integration of the `CellList_gpu_ker` class, which would reduce the number of control flow branches, eliminate the required `llvm.alloca` operation, and remove all device runtime calls. Additionally, rather than relying on the `scf` dialect when lowering the `cell_list.for_all_neighbor_indices` operation, it may be beneficial to directly emit code using operations from the `cf` (low-level control flow) dialect. This would allow precise control over the basic blocks and control flow branches, which is crucial for reducing their number. Finally, the check that determines whether a neighbor is within the cutoff sphere could be integrated directly into the `particles.for_all_neighbors` or `particles_dist.for_all_neighbors` operations. Doing so would allow performing this check in the most optimal way, potentially using branchless programming.

Bibliography

[1] Tobias Weinzierl. “The pillars of science”. In: *Principles of parallel scientific computing: A first guide to numerical concepts and programming methods*. Springer, 2022, pp. 3–9.

[2] RW Hockney. “Computer experiment of anomalous diffusion”. In: *The Physics of Fluids* 9.9 (1966), pp. 1826–1835.

[3] Geroges-Henri Cottet and Sylvie Mas-Gallic. “A particle method to solve the Navier-Stokes system”. In: *Numerische Mathematik* 57 (1990), pp. 805–827.

[4] Pablo Alvarez Lopez et al. “Microscopic traffic simulation using sumo”. In: *2018 21st international conference on intelligent transportation systems (ITSC)*. Ieee. 2018, pp. 2575–2582.

[5] Johannes Pahlke and Ivo F Sbalzarini. “A unifying mathematical definition of particle methods”. In: *IEEE Open Journal of the Computer Society* 4 (2023), pp. 97–108.

[6] Roger W Hockney and James W Eastwood. *Computer simulation using particles*. crc Press, 2021.

[7] Ivo F Sbalzarini. “Abstractions and middleware for petascale computing and beyond”. In: *Technology Integration Advancements in Distributed Systems and Computing* (2012), pp. 161–178.

[8] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[9] John Nickolls et al. “Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?” In: *Queue* 6.2 (2008), pp. 40–53.

[10] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings, 11th European PVM/MPI Users’ Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[11] *MPICH | High-Performance Performance Portable MPI*. https://www.mpich.org/. Accessed: 2025-05-20.

[12] Pietro Incardona et al. “OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers”. In: *Computer Physics Communications* 241 (2019), pp. 155–177.

[13] Pietro Incardona. *OpenFPM: A scalable environment for particle and particle-mesh codes on parallel computers*. Ph.D thesis. Max Planck Institute of Molecular Cell Biology and Genetics, Center for Systems Biology Dresden. Dresden, Germany, 2021.

[14] Nesrine Khouzami et al. "The OpenPME Problem Solving Environment for Numerical Simulations". In: June 2021, pp. 614–627. ISBN: 978-3-030-77960-3. DOI: 10.1007/978-3-030-77961-0_49.

[15] I.F. Sbalzarini et al. "PPM – A highly efficient parallel particle–mesh library for the simulation of continuum systems". In: *Journal of Computational Physics* 215.2 (2006), pp. 566–588. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2005.11.017. URL: https://www.sciencedirect.com/science/article/pii/S002199910500505X.

[16] Omar Awile, Ömer Demirel, and Ivo F Sbalzarini. "Toward an Object-Oriented Core of the PPM Library". In: *AIP Conference Proceedings*. Vol. 1281. 1. American Institute of Physics. 2010, pp. 1313–1316.

[17] Loup Verlet. "Computer" experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules". In: *Physical review* 159.1 (1967), p. 98.

[18] Efstratios Gallopoulos, Elias Houstis, and John R Rice. "Computer as thinker/doer: Problem-solving environments for computational science". In: *IEEE Computational Science and Engineering* 1.2 (2002), pp. 11–23.

[19] Sven Karol et al. "A Domain-Specific Language and Editor for Parallel Particle Methods". In: *ACM Trans. Math. Softw.* 44.3 (Mar. 2018). ISSN: 0098-3500. DOI: 10.1145/3175659. URL: https://doi.org/10.1145/3175659.

[20] *MPS: The Domain-Specific Language Creator by JetBrains*. https://www.jetbrains.com/mps/. Accessed: 2025-05-20.

[21] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[22] Tobias Gysi et al. "Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 18.4 (2021), pp. 1–23.

[23] Fabio Luporini et al. "Architecture and performance of Devito, a system for automated stencil computation". In: *ACM Transactions on Mathematical Software (TOMS)* 46.1 (2020), pp. 1–28.

[24] Samantha V Adams et al. "LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models". In: *Journal of Parallel and Distributed Computing* 132 (2019), pp. 383–396.

[25] George Bisbas et al. "A shared compilation stack for distributed-memory parallelism in stencil DSLs". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024, pp. 38–56.

[26] BJ Alder and TE Wainwright. "Molecular dynamics simulation of hard sphere system". In: *J. Chem. Phys* 27.1 (1957), pp. 208–1.

[27] G-H Cottet et al. "High order semi-Lagrangian particle methods for transport equations: numerical analysis and implementation issues". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 48.4 (2014), pp. 1029–1060.

[28] Sylvain Reboux, Birte Schrader, and Ivo F Sbalzarini. “A self-organizing Lagrangian particle method for adaptive-resolution advection–diffusion simulations”. In: *Journal of Computational Physics* 231.9 (2012), pp. 3623–3646.

[29] Omar Awile et al. “Fast neighbor lists for adaptive-resolution particle simulations”. In: *Computer Physics Communications* 183.5 (2012), pp. 1073–1081. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2012.01.003. URL: https://www.sciencedirect.com/science/article/pii/S0010465512000057.

[30] *MLIR Language Reference*. https://web.archive.org/web/20250418135528/https://mlir.llvm.org/docs/LangRef/. Archived: 2025-04-17.

[31] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[32] C. Lattner and V. Adve. “LLVM: a compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[33] Mathieu Fehr et al. “xDSL: Sidekick Compilation for SSA-Based Compilers”. In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 2025, pp. 179–192.

[34] Mohamed Essadki et al. “Code generation for in-place stencils”. In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 2023, pp. 2–13.

[35] Gabriel Rodriguez-Canal et al. “Stencil-HMLS: A multi-layered approach to the automatic optimisation of stencil codes on FPGA”. In: *Proceedings of the SC’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 556–565.

[36] Norman A Rink et al. “CFDlang: High-level code generation for high-order methods in fluid dynamics”. In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. 2018, pp. 1–10.

[37] Norman A Rink and Jeronimo Castrillon. “TeIL: a type-safe imperative Tensor Intermediate Language”. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2019, pp. 57–68.

[38] Stephanie Soldavini et al. “Automatic creation of high-bandwidth memory architectures from domain-specific languages: The case of computational fluid dynamics”. In: *ACM Transactions on Reconfigurable Technology and Systems* 16.2 (2023), pp. 1–34.

[39] Yifei He et al. “FFTc: An mlir dialect for developing hpc fast fourier transform libraries”. In: *European Conference on Parallel Processing*. Springer. 2022, pp. 80–92.

[40] Yifei He and Stefano Markidis. “High-Performance FFT Code Generation via MLIR Linalg Dialect and SIMD Micro-Kernels”. In: *2024 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2024, pp. 155–165.

[41] *Copy elision - cppreference.com*. https://web.archive.org/web/20250526143136/https://en.cppreference.com/w/cpp/language/copy_elision.html. Archived: 2025-05-20.

[42] Saman P Amarasinghe and Monica S Lam. "Communication optimization and code generation for distributed memory machines". In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 1993, pp. 126–138.

[43] Lynn Choi and Pen-Chung Yew. "Eliminating stale data references through array data-flow analysis". In: *Proceedings of International Conference on Parallel Processing*. IEEE. 1996, pp. 4–13.

List of Figures

List of Tables

List of Listings

List of Algorithms

A Appendix

A.1 Molecular Dynamics Code

```
// Floating point precision
!f = f64
// Dimensionality of the space the simulation is happening in
#dims = 3 : index
#neighbor_list = #particles_dist.neighbor_list_kind<cell_list>
// Type of vector used for position, velocity and force
!vector = vector<3x!f>
// property names
#force = "force"
#velocity = "velocity"
// Mapping property names to types
#properties = { "velocity" = !vector, "force" = !vector}
#max_distance = 0.3 : !f
// Combining everything into !particles.set and !particles.particle types
!set = !particles.set<?, !f, #dims, #properties>
!particle = !particles.particle<!f, #dims, #properties>
#add = #particles.reduction_kind<add>
!storage = !particles.storage<!set>

module {

  func.func private @batch_done(%storage : !storage,
                                %context : !llvm.ptr,
                                %step : index) -> ()

  func.func @calc_forces(
    %set : !set,
    %sigma12 : !f,
    %sigma6 : !f,
    %r_cut2 : !f,
    %ignore_r_cut : i1
  ) -> !set {

    %zero = arith.constant 0.0 : !f
    %zero_vector = "vector.broadcast"(%zero) : (!f) -> !vector
    %minus_one = arith.constant -1 : !f
    %minus_sigma6 = arith.mulf %minus_one, %sigma6 : !f
    %two = arith.constant 2.0 : !f
    %twentyfour = arith.constant 24.0 : !f

    %interacted = "particles.for_all_neighbors" (%set)
    <{"with_self" = false,
```

```
    "write_targets" = ["force"],
    "reduction_kinds" = [#add]}>
  ({^0(%particle : !particle, %neighbor : !particle):

      %pos_particle = "particles.get_position" (%particle) : (!particle) -> !vector
      %pos_neighbor = "particles.get_position" (%neighbor) : (!particle) -> !vector

      %delta_pos = arith.subf %pos_particle, %pos_neighbor : !vector

      // double rn = norm2(r);
      %delta_pos_squared = arith.mulf %delta_pos, %delta_pos : !vector

      %rn = vector.reduction <add>, %delta_pos_squared : !vector into !f

      %within_range = arith.cmpf ole, %rn, %r_cut2 : !f

      // Use this with care, it seems to affect the performance quite a lot,
      // better to comment out the code:
      // %perform_interaction = arith.ori %within_range, %ignore_r_cut : i1

      %result = scf.if %within_range -> (!vector) {

        %rn_pow2 = arith.mulf %rn, %rn : !f
        %rn_pow4 = arith.mulf %rn_pow2, %rn_pow2 : !f
        %rn_pow6 = arith.mulf %rn_pow4, %rn_pow2 : !f
        %rn_pow7 = arith.mulf %rn_pow6, %rn : !f

        // Point<3,double> f = 24.0*(2.0 *sigma12 / (rn*rn*rn*rn*rn*rn*rn)
        //                     - sigma6 / (rn*rn*rn*rn)) * r;

        // sigma12 / (rn*rn*rn*rn*rn*rn*rn)
        %force_0 = arith.divf %sigma12, %rn_pow7 : !f
        // - sigma6 / (rn*rn*rn*rn)
        %force_1 = arith.divf %minus_sigma6, %rn_pow4 : !f
        // 2.0 *sigma12 / (rn*rn*rn*rn*rn*rn*rn) -  sigma6 / (rn*rn*rn*rn)
        %force_2 = "math.fma"(%two, %force_0, %force_1) : (!f, !f, !f) -> !f
        // 24.0*(2.0 *sigma12 / (rn*rn*rn*rn*rn*rn*rn) -  sigma6 / (rn*rn*rn*rn))
        %force_3 = arith.mulf %twentyfour, %force_2 : !f

        %force_3_broadcast = "vector.broadcast"(%force_3) : (!f) -> !vector

        %final_force = "arith.mulf" (%force_3_broadcast, %delta_pos)
                                    : (!vector, !vector) -> !vector

        scf.yield %final_force : !vector
      } else {
        scf.yield %zero_vector : !vector
      }

      "particles.yield" (%result) : (!vector) -> ()

  }) {"neighbor_list_kind" = #neighbor_list,
      "max_distance" = #max_distance,
      "inside-unroll-factor" = 4 : i32,
      "outside-unroll-factor" = 4 : i32}  : (!set) -> (!set)

  func.return %interacted : !set
} // @calc_forces

// Makesw @calc_forces accessible from C++:
func.func @calc_forces_mlir(%storage : !storage,
                            %sigma12 : !f,
                            %sigma6 : !f,
```

```
                            %r_cut2 : !f,
                            %ignore_r_cut : i1) -> () {

  %zero_index = arith.constant 0 : index

  "particles.loop" (%storage,
                    %zero_index,
                    %zero_index)
  ({^0(%set_0 : !set,
       %step : index):

       %set_1 = "particles.apply" (%set_0,
                                   %sigma12,
                                   %sigma6,
                                   %r_cut2,
                                   %ignore_r_cut)
                                   {"func" = @calc_forces}
                                   : (!set, !f, !f, !f, i1) -> !set

       "particles.next" (%set_1) : (!set) -> ()

  },
  {^0(%set : !set, %step : index): "particles.next"(%set): (!set) -> ()}
  ) : (!storage, index, index) -> ()

  func.return
} // @calc_forces_mlir

// Verlet time-stepping: first evolve step
//
//     while (it3.isNext())
// {
//     auto p = it3.get();
//
//     // here we calculate v(tn + 0.5)
//     vd.template getProp<velocity>(p)[0] += 0.5*dt*vd.template getProp<force>(p)[0];
//     vd.template getProp<velocity>(p)[1] += 0.5*dt*vd.template getProp<force>(p)[1];
//     vd.template getProp<velocity>(p)[2] += 0.5*dt*vd.template getProp<force>(p)[2];
//
//     // here we calculate x(tn + 1)
//     vd.getPos(p)[0] += vd.template getProp<velocity>(p)[0]*dt;
//     vd.getPos(p)[1] += vd.template getProp<velocity>(p)[1]*dt;
//     vd.getPos(p)[2] += vd.template getProp<velocity>(p)[2]*dt;
//
//     ++it3;
// }
func.func @evolve_0(
  %set : !set,
  %deltat : !f
) -> !set {

  %half = arith.constant 0.5 : !f
  %half_deltat = arith.mulf %deltat, %half : !f

  %deltat_vector = "vector.broadcast"(%deltat) : (!f) -> !vector
  %half_deltat_vector = "vector.broadcast"(%half_deltat) : (!f) -> !vector

  %evolved = "particles.foreach"( %set) ({
    ^0( %particle : !particle):

      %force = "particles.get_property"(%particle) {"property" = #force}
                                        : (!particle) -> !vector
```

```
      %old_vel = "particles.get_property"(%particle) {"property" = #velocity}
                                        : (!particle) -> !vector

      %new_vel = "math.fma"(%half_deltat_vector, %force, %old_vel)
                           : ( !vector, !vector, !vector) -> !vector

      %updated_particle_0 = "particles.set_property"(%particle, %new_vel)
                                                    {"property" = #velocity}
                                                    : (!particle, !vector) -> !particle

      %pos = "particles.get_position"(%particle)
                                     : (!particle) -> !vector

      %new_pos = "math.fma"(%deltat_vector, %new_vel, %pos)
                           : ( !vector, !vector, !vector) -> !vector

      %updated_particle_1 = "particles.set_position"(%updated_particle_0, %new_pos)
                                                    : (!particle, !vector) -> !particle

      "particles.yield" (%updated_particle_1) : (!particle) -> ()

  }) {"unroll-factor" = 4 : i32} : (!set) -> !set

  func.return %evolved : !set
} // @evolve_0

// Makes @evovle_0 accessible from C++:
func.func @evolve_0_mlir(%storage : !storage, %deltat : !f) -> () {

  %zero_index = arith.constant 0 : index

  "particles.loop" (%storage,
                    %zero_index,
                    %zero_index)
  ({^0(%set_0 : !set,
      %step : index):

      %set_1 = "particles.apply" (%set_0, %deltat) <{"func" = @evolve_0}>
                                 : (!set, !f) -> !set

      "particles.next" (%set_1) : (!set) -> ()

  },
  {^0(%set : !set, %step : index): "particles.next"(%set): (!set) -> ()}
  ) : (!storage, index, index) -> ()

  func.return
}

// Verlet time-stepping: second evolve step
//
// while (it4.isNext())
// {
//     auto p = it4.get();
//
//     // here we calculate v(tn + 1)
//     vd.template getProp<velocity>(p)[0] += 0.5*dt*vd.template getProp<force>(p)[0];
//     vd.template getProp<velocity>(p)[1] += 0.5*dt*vd.template getProp<force>(p)[1];
//     vd.template getProp<velocity>(p)[2] += 0.5*dt*vd.template getProp<force>(p)[2];
//
//     ++it4;
// }
func.func @evolve_1(%set : !set, %deltat : !f) -> !set {
```

```
  %half = arith.constant 0.5 : !f
  %half_deltat = arith.mulf %deltat, %half : !f
  %half_deltat_vector = "vector.broadcast"(%half_deltat) : (!f) -> !vector

  %evolved = "particles.foreach"( %set) ({
    ^0( %particle : !particle):

      %force = "particles.get_property"(%particle) {"property" = #force}
                                        : (!particle) -> !vector

      %old_vel = "particles.get_property"(%particle) {"property" = #velocity}
                                          : (!particle) -> !vector

      %new_vel = "math.fma"(%half_deltat_vector, %force, %old_vel)
                           : ( !vector, !vector, !vector) -> !vector

      %updated_particle_0 = "particles.set_property"(%particle, %new_vel)
                                                     {"property" = #velocity}
                                                     : (!particle, !vector) -> !particle

      "particles.yield" (%updated_particle_0) : (!particle) -> ()

  }) {"unroll-factor" = 4 : i32} : (!set) -> !set

  func.return %evolved : !set
} // @evolve_1

// Makes @evovle_1 accessible from C++:
func.func @evolve_1_mlir(%storage : !storage, %deltat : !f) -> () {

  %zero_index = arith.constant 0 : index

  "particles.loop" (%storage,
                    %zero_index,
                    %zero_index)
  ({^0(%set_0 : !set,
      %step : index):

      %set_1 = "particles.apply" (%set_0, %deltat) {"func" = @evolve_1}
                                 : (!set, !f) -> !set

      "particles.next" (%set_1) : (!set) -> ()

  },
  {^0(%set : !set, %step : index): "particles.next"(%set): (!set) -> ()}
  ) : (!storage, index, index) -> ()

  func.return
} // @evolve_1_mlir

func.func @md(%context : !llvm.ptr,
              %storage : !storage,
              %sigma12 : !f,
              %sigma6 : !f,
              %r_cut2 : !f,
              %ignore_r_cut: i1,
              %num_steps: index,
              %batch_size : index,
              %deltat: !f) {

  %zero = arith.constant 0.0 : !f
  %minus_one = arith.constant -1 : !f
  %zero_index = arith.constant 0 : index
```

```
  "particles.loop" (%storage,
                    %zero_index,
                    %num_steps)
  (
  // initialization region:
  {^0(%set_0 : !set, %step : index):

      %interacted_set = "particles.apply" (%set_0,
                                           %sigma12,
                                           %sigma6,
                                           %r_cut2,
                                           %ignore_r_cut)
                                           {"func" = @calc_forces}
                                           : (!set, !f, !f, !f, i1) -> !set

      "particles.next" (%interacted_set)
        : (!set) -> ()
  },
  // time-stepping region:
  {^0(%set_0 : !set, %step : index):

      // Print after every batch
      %step_mod = arith.remsi %step, %batch_size: index

      %should_print = arith.cmpi eq, %step_mod, %zero_index : index

      %set_after_print= "scf.if"(%should_print)
      ({
        %result = "particles.call" (%set_0, %context, %step)
                                   <{"func" = @batch_done}>
                                   {"requires_up_to_date" = [],
                                    "updates" = [],
                                    "makes_stale" = []}
                                    : (!set, !llvm.ptr, index) -> !set
        scf.yield %result : !set

      },{
        scf.yield %set_0 : !set
      }) : (i1) -> !set

      // Verlet time stepping
      %evolved_0_set = "particles.apply" (%set_after_print, %deltat)
                                         {"func" = @evolve_0}
                                         : (!set, !f) -> !set

      %interacted_set = "particles.apply" (%evolved_0_set,
                                           %sigma12,
                                           %sigma6,
                                           %r_cut2,
                                           %ignore_r_cut)
                                           {"func" = @calc_forces}
                                           : (!set, !f, !f, !f, i1) -> !set

      %evolved_1_set = "particles.apply" (%interacted_set, %deltat)
                                         {"func" = @evolve_1}
                                         : (!set, !f) -> !set

      "particles.next" (%evolved_1_set)
        : (!set) -> ()
  }) : (!storage, index, index) -> () // particles.loop

  func.return
```

```
  } // @md

} // builtin.module
```

A.2 Lowering Pipeline

```
# Expected defined:
# VERBOSE_LOWERING : either YES or NO
# LOOP_FUSION : either YES or NO
# BASE : file base. E.g., md for md.mlir
# MLIR_OPT : mlir-opt program
# OPENPME_MLIR_OPT : openpme-mlir-opt program
# HW_TARGET : cpu or cuda
# LOG_RUNTIME_CALLS: either true or false
# GPU_MODULE_TO_BINARY_LINK: Whitespace-separated list of llvm bitcode files that should
# be linked with MLIR device code during the --gpu-module-to-binary pass


MLIR_OPT ?= mlir-opt
OPENPME_XDSL_OPT ?= openpme-xdsl-opt
OPENPME_MLIR_OPT ?= $(abspath $(WORKSPACE_DIR)/openpme_mlir/build/bin/openpme-mlir-opt)
LOOP_FUSION ?= NO

PRECIOUS_MLIR = \
   01_%-particles-inline-apply-funcs.mlir \
   02_%-particles-dist-specialize-particles.mlir \
   03_%-particles-dist-fused.mlir \
   04_%-particles-dist-place-update-and-communication-ops.mlir \
   06_%-particles-dist-generate-runtime.mlir \
   05-1_%-convert-operations.mlir \
   05-2_%-convert-types.mlir \
   05-3_%-cse.mlir \
   06_%-box-convert-storage-to-ll.mlir \
   07-1_%-box-expand.mlir \
   07-2_%-box-shortcut-extract-ops.mlir \
   08_%-memwrap-eliminate-redundant-load-ops.mlir \
   09_%-convert-memwrap-to-memref-vector.mlir \
   21_%-target-hardware.mlir \
   22_%-optimized.mlir \
   99_%.mlir \
   99_%.ll \
   99_%.asm \
   $(GPU_MODULE_TO_BINARY_LINK)

ifeq ($(VERBOSE_LOWERING),YES)
CAT_RESULT = @[ -s $@ ] && cat $@
PRINT_IR_BEFORE_ALL = --mlir-print-ir-before-all
else
CAT_RESULT =
PRINT_IR_BEFORE_ALL =
endif

# xdsl-opt outputs empty files, even when the pass fails.
# It's better when there are no empty files as this allows restarting the build pipeline,
# without having to manually delete the empty file.
REMOVE_IF_FAILED = (rm -f $@ && false)

ifeq ($(HW_TARGET),cpu)
RUNTIME = runtime.cpp
else ifeq ($(HW_TARGET),cuda)
RUNTIME = runtime.cu
else
```

```
$(error Unsupported hardware target $(HW_TARGET))
endif


01_%-particles-inline-apply-funcs.mlir: %.mlir
	$(OPENPME_XDSL_OPT) \
	--print-op-generic \
	--passes \
	\
	particles-inline-apply-funcs \
	\
	$< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
	$(CAT_RESULT)

02_%-particles-dist-specialize-particles.mlir: 01_%-particles-inline-apply-funcs.mlir
	$(OPENPME_XDSL_OPT) \
	--print-op-generic \
	--passes \
	\
	particles-dist-specialize-particles \
	\
	$< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
	$(CAT_RESULT)


ifeq ($(LOOP_FUSION),YES)
03_%-particles-dist-fused.mlir: 02_%-particles-dist-specialize-particles.mlir
	$(OPENPME_XDSL_OPT) \
	--print-op-generic \
	--passes \
	\
	particles-dist-fuse-foreach-ops{aggression=1},\
	particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops{aggression=1},\
	particles-dist-fuse-foreach-ops{aggression=2},\
	particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops{aggression=2},\
	particles-dist-fuse-foreach-ops{aggression=3},\
	particles-dist-fuse-for-all-neighbors-ops-with-foreach-ops{aggression=3} \
	\
	$< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
	$(CAT_RESULT)
else
03_%-particles-dist-fused.mlir: 02_%-particles-dist-specialize-particles.mlir
	cp $< $@
endif


04_%-particles-dist-place-update-and-communication-ops.mlir: 03_%-particles-dist-fused.mlir
# The order is important here
# Need to convert the maybe ops to if-stale construct so that the consequent passes can
# place the required ops inside.
# E.g. place a (maybe-)map op in front of a ghost_get op
	$(OPENPME_XDSL_OPT) \
	--print-op-generic \
	--passes \
	\
	particles-dist-place-update-neighbor-list-ops,\
	particles-dist-convert-maybe-ops-to-if-stale,\
	canonicalize,\
	particles-dist-place-ghost-get-ops,\
	particles-dist-convert-maybe-ops-to-if-stale,\
	canonicalize,\
	particles-dist-place-map-ops,\
	particles-dist-convert-maybe-ops-to-if-stale,\
```

```
   canonicalize,\
   particles-dist-place-set-stale-ops \
   \
   $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
   $(CAT_RESULT)


runtime.h $(RUNTIME) 06_$(BASE)-particles-dist-generate-runtime.mlir: ⌋
        04_$(BASE)-particles-dist-place-update-and-communication-ops.mlir
   $(OPENPME_XDSL_OPT) \
   --print-op-generic \
   --passes \
   \
   particles-dist-place-update-internals-ops,\
   particles-dist-generate-runtime{header-file=\"runtime.h\"\ ⌋
                                   source-file=\"$(RUNTIME)\"\ ⌋
                                   header-file-include-path=\"runtime.h\"\ ⌋
                                   target=$(HW_TARGET)\ ⌋
                                   log-runtime-calls=$(LOG_RUNTIME_CALLS)} \
   \
   $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
   $(CAT_RESULT)


05-1_%-convert-operations.mlir: 06_%-particles-dist-generate-runtime.mlir
   $(OPENPME_XDSL_OPT) \
   --print-op-generic \
   --passes \
   \
   particles-dist-convert-ops{target=$(HW_TARGET)\ is-final-conversion=false},\
   local-domain-convert-ops,\
   cell-list-convert-ops-to-$(HW_TARGET),\
   particles-dist-convert-ops{target=$(HW_TARGET)\ is-final-conversion=true},\
   reconcile-unrealized-casts \
   \
   $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
   $(CAT_RESULT)

05-2_%-convert-types.mlir: 05-1_%-convert-operations.mlir
   $(OPENPME_XDSL_OPT) \
   --print-op-generic \
   --passes \
   \
   particles-dist-convert-types,\
   local-domain-convert-types,\
   cell-list-convert-types-to-$(HW_TARGET),\
   reconcile-unrealized-casts \
   \
   $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
   $(CAT_RESULT)

05-3_%-cse.mlir: 05-2_%-convert-types.mlir
# Previous pass usually introduces a lot of box.extract ops that can be eliminated with CSE
   $(OPENPME_MLIR_OPT) \
   -allow-unregistered-dialect \
   --mlir-print-op-generic \
   --canonicalize \
   --cse \
      $< -o $@ 2> >(tee $@.err >&2)
   $(CAT_RESULT)

06_%-box-convert-storage-to-ll.mlir: 05-3_%-cse.mlir
   $(OPENPME_XDSL_OPT) \
```

```
    --print-op-generic \
    --passes \
    \
    box-convert-storage-to-ll,\
    reconcile-unrealized-casts \
    \
    $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
    $(CAT_RESULT)

07-1_%-box-expand.mlir: 06_%-box-convert-storage-to-ll.mlir
    $(OPENPME_XDSL_OPT) \
    --print-op-generic \
    --passes \
    \
    box-expand-func,\
    box-expand-scf-for,\
    box-expand-scf-if,\
    box-expand-box-ops \
    \
    $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
    $(CAT_RESULT)

07-2_%-box-shortcut-extract-ops.mlir: 07-1_%-box-expand.mlir
    $(OPENPME_XDSL_OPT) \
    --print-op-generic \
    --passes \
    \
    box-shortcut-extract-ops,\
    canonicalize \
    \
    $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
    $(CAT_RESULT)

08_%-memwrap-eliminate-redundant-load-ops.mlir: 07-2_%-box-shortcut-extract-ops.mlir
    $(OPENPME_XDSL_OPT) \
    --print-op-generic \
    --passes \
    \
    memwrap-eliminate-redundant-load-ops,\
    canonicalize \
    \
    $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
    $(CAT_RESULT)


09_%-convert-memwrap-to-memref-vector.mlir: 08_%-memwrap-eliminate-redundant-load-ops.mlir
    $(OPENPME_XDSL_OPT) \
    --print-op-generic \
    --passes \
    \
    convert-memwrap-to-memref-vector \
    \
    $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
    $(CAT_RESULT)

21_%-target-hardware.mlir: 09_%-convert-memwrap-to-memref-vector.mlir
# Do some target specific passes before final optimizations and lowering steps.
ifeq ($(HW_TARGET), cpu)
# For cpu target: convert all scf.parallel to scf.for and host arith.addf since conversion
# from scf.parallel to cf seems to result in some very inefficient code
    $(OPENPME_XDSL_OPT) \
    --print-op-generic \
    --passes \
```

```
    \
    scf-convert-parallel-to-for{only_convert_annotated=false},\
    hoist-arith-addf-into-scf-if \
    \
    $< -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
    $(CAT_RESULT)
else ifeq ($(HW_TARGET), cuda)
# For cuda target: convert mapped scf.parallel ops to gpu.launch ops
# May also benefit from hoisting arith.addf into scf.if
    $(OPENPME_XDSL_OPT) \
    --print-op-generic \
    --passes \
    \
    hoist-arith-addf-into-scf-if \
    $< \
    | \
    $(OPENPME_MLIR_OPT) \
    --mlir-print-ir-after-failure \
    $(PRINT_IR_BEFORE_ALL) \
    --convert-parallel-loops-to-gpu \
    \
    -o $@ 2> >(tee $@.err >&2) || $(REMOVE_IF_FAILED)
    $(CAT_RESULT)
else
    $(error Unsupported hardware target $(HW_TARGET))
endif


# Target specific optimizations
ifeq ($(HW_TARGET), cpu)
# Nothing for now
TARGET_SPECIFIC_OPTIMIZATIONS =
else ifeq ($(HW_TARGET), cuda)
# Nothing for now
TARGET_SPECIFIC_OPTIMIZATIONS =
else
    $(error Unsupported hardware target $(HW_TARGET))
endif

22_%-optimized.mlir: 21_%-target-hardware.mlir
# Do some optimizations before lowering
    $(OPENPME_MLIR_OPT) \
    --mlir-print-ir-after-failure \
    $(PRINT_IR_BEFORE_ALL) \
    --canonicalize \
    --cse \
    --reconcile-unrealized-casts \
    $(TARGET_SPECIFIC_OPTIMIZATIONS) \
    --canonicalize \
    --cse \
    --reconcile-unrealized-casts \
    --symbol-dce \
    --math-uplift-to-fma \
    --buffer-loop-hoisting \
    --buffer-hoisting \
    --unroll-scf-for-by-attribute-or-default="annotate=true unroll-factor-default=1" \
    --canonicalize \
    --cse \
    --reconcile-unrealized-casts \
    $< -o $@ 2> >(tee $@.err >&2)
    $(CAT_RESULT)

# Target specific lowering passes
```

```
ifeq ($(HW_TARGET), cpu)
# For cpu target: Do nothing
TARGET_SPECIFIC_LOWERING =
BC_FILES =
else ifeq ($(HW_TARGET), cuda)
ifneq ($(GPU_MODULE_TO_BINARY_LINK),)
WHITESPACE = $(subst ,, )
COMMA = ,
BC_FILES_COMMA_SEPARATED = \
   runtime-cuda.bc,$(subst $(WHITESPACE),$(COMMA),$(GPU_MODULE_TO_BINARY_LINK))
BC_FILES = runtime-cuda.bc $(GPU_MODULE_TO_BINARY_LINK)
else
BC_FILES_COMMA_SEPARATED = runtime-cuda.bc
BC_FILES = runtime-cuda.bc
endif

# Outline kernels and lower gpu module ops to binary
# Very similar to "--gpu-lower-to-nvvm-pipeline" (GPUToNVVMPipeline.cpp),
# excluding the host pipeline which isn't necessary
TARGET_SPECIFIC_LOWERING = \
   --gpu-kernel-outlining \
   --nvvm-attach-target="O=3 chip=$(CUDA_ARCH) \
                         features=+ptx73 l=$(BC_FILES_COMMA_SEPARATED)" \
   --convert-gpu-to-nvvm \
   --canonicalize \
   --cse \
   --reconcile-unrealized-casts \
   --gpu-to-llvm \
   --gpu-module-to-binary \
   --canonicalize \
   --cse \
   --reconcile-unrealized-casts
else
   $(error Unsupported hardware target $(HW_TARGET))
endif

99_%.mlir: 22_%-optimized.mlir $(BC_FILES)
# Very similar to "--test-lower-to-llvm" (TestLowerToLLVM.cpp)
# (Removed linalg and affine conversion passes)
   $(MLIR_OPT) \
   --mlir-print-ir-after-failure \
   $(PRINT_IR_BEFORE_ALL) \
   --convert-vector-to-scf \
   --convert-scf-to-cf \
   --canonicalize \
   --cse \
   --convert-vector-to-llvm \
   --convert-math-to-llvm \
   --arith-expand \
   --expand-strided-metadata \
   --lower-affine \
   --finalize-memref-to-llvm \
   --convert-func-to-llvm \
   --convert-index-to-llvm \
   --reconcile-unrealized-casts \
   --canonicalize \
   --cse \
   $(TARGET_SPECIFIC_LOWERING) \
   $< -o $@ 2> >(tee $@.err >&2)
   $(CAT_RESULT)

clean-mlir-files:
   @if [ "$(BASE)" = "" ]; then \
```

```
    echo "ERROR: Expected BASE to be defined in Makefile" \
    exit 1; \
fi
rm -f [0-9][0-9]_$(BASE)*.mlir [0-9][0-9]_$(BASE)*.mlir.err
```