An extension of C++ with memory-centric specifications for HPC to reduce memory footprints and streamline MPI development

Pawel K. Radtke and Cristian G. Barrera-Hinojosa and Mladen Ivkovic and Tobias Weinzierl

The C++ programming language and its cousins lean towards a memory-inefficient storage of structs: The compiler inserts helper bits into the struct such that individual attributes align with bytes, and it adds additional bytes aligning attributes with cache lines, while it is not able to exploit knowledge about the range of integers, enums or bitsets to bring the memory footprint down. Furthermore, the language provides neither support for data exchange via MPI nor for arbitrary floating-point precision formats. If developers need to have a low memory footprint and MPI datatypes over structs which exchange only minimal data, they have to manipulate the data and to write MPI datatypes manually. We propose a C++ language extension based upon C++ attributes through which developers can guide the compiler what memory arrangements would be beneficial: Can multiple booleans be squeezed into one bit field, do floats hold fewer significant bits than in the IEEE standard, or does the code require a user-defined MPI datatype for certain subsets of attributes? The extension offers the opportunity to fall back to normal alignment and padding rules via plain C++ assignments, no dependencies upon external libraries are introduced, and the resulting code remains standard C++. Our work implements the language annotations within LLVM and demonstrates their potential impact, both upon the runtime and the memory footprint, through smoothed particle hydrodynamics (SPH) benchmarks. They uncover the potential gains in terms of performance and development productivity.

1. INTRODUCTION

Switching from a low-level (machine) programming language to a generic high-level language such as C, C++ or Fortran makes programming more efficient, increases the code performance, and it introduces machine-portability: Source code is not tied to one architecture's instruction set anymore as long as fitting compilers are available, while the translator can take over the lion's share of work to make the low-level code fast. This tuning includes proper memory alignment and padding. Some developers have the skills and knowledge to tweak the memory layout and, through this, to produce faster code than a compiler, but it is generally difficult to compete with a good compiler which has access to heuristics reflecting the internals of a machine.

One dominant high-level language family in scientific computing is C/C++ with its cousins CUDA and SYCL [Reinders et al. 2021]. Fortran remains the other prominent language to realise core software in high-performance computing (HPC). Our work focuses on C++ and starts from the identification of some shortcomings within C++ which adversely affect HPC developers.

First, the C++ language yields classes with a large memory footprint. Since we are interested in data arrangements, we use struct and class as synonym from hereon, assuming that a class is a struct with different default visibility constraints plus, in some cases, a pointer to a virtual function table [Hyde 2006]. A struct's members are aligned in memory by introducing padding bytes. Further to that, the smallest memory unit that can store a variable is a byte, which provides a poor information density for a boolean. As it can only hold true or false, one bit would be enough to encode its information. Enumerations suffer from this overprovision of memory, too.

Second, the C++ language lacks support for a "continuous" range of data precisions. It offers datatypes which are natively supported by hardware, yet does not allow programmers to express further knowledge about the value ranges of integers or the actual accuracy of a numerical datatypes (number of significant bits). This again affects the memory footprint of applications and makes programming for different datatypes (mixed precision programming) [Higham and Mary 2022] laborious.

Finally, the C++ language does not offer built-in support for distributed memory parallelisation through the Message-Passing Interface (MPI) [Gropp et al. 2014]. MPI remains the de-facto standard to program supercomputers. If developers want to map C++ structs onto MPI, they have to translate the struct's attributes manually into memory addresses and trigger some address arithmetics. This quickly becomes error-prone and time consuming, notably once we want to support different MPI types per struct which exchange different subsets of attributes.

Our work is driven by the hypothesis that these shortcomings of C++ often have a negative impact on the quality of scientific software design and its performance. In an era where the CPU–memory gap is widening [Dongarra et al. 2011], memory modesty gains importance. Codes with small memory footprint have reduced memory bandwidth requirements and are able to retain more data within the caches close to the core. They perform better. In an era where the memory per core is stagnating, weak scaling per node is constrained. Codes with small memory footprint can squeeze larger problems onto a single node and hence run into strong scaling saturation later. In an era where the energy consumption of computers—a metric determined by memory movements—gains importance, the science per moved byte, i.e. the information density, deserves particular attention. Codes should use every single bit to hold meaningful information. In an era where the interconnect bandwidth struggles to keep pace with the per-node performance, it is important to minimise the memory footprint per information exchanged between nodes.

We propose novel C++ annotations to address the language's shortcomings. We also prototype a LLVM modification supporting the new annotations. Our annotations, firstly, allow developers to mark booleans, enumerations or integers with constrained ranges to indicate that they should be packed into one large bitfield within the struct. Our compiler automatically supplements accesses to struct attributes with the required bit shift operations. Secondly, we propose that floating-point data are annotated with information density information: The struct's floating-point data are held in a compressed bit representation (smaller than built-in hardware datatypes) and mapped to and from native datatypes throughout computations. Finally, our compiler extension accepts MPI datatype annotations for those struct members that are to be exchanged via message passing. Different MPI views, i.e. subsets of attributes that are to be exchanged, can be specified straightforwardly.

C++ annotations allow users to stick to code which complies with the C++ standard. If a compiler does not "understand" the annotations, they are ignored. Annotations can be applied incrementally, i.e. do not require a code refactoring/rewrite to unfold their potential. We also do not introduce any dependencies on external libraries. Realised as additional code transformation pass, our language extensions play in a team with other compiler optimisations, while hiding how data are internally encoded from the user. This provides a "native" way for developers to toggle between various data representations. Simple assignments change from memory-optimised to default (performance-optimised) data representations which are subject to proper alignment, padding and mapping onto hardware-supported data formats. Any compiler will eliminate these assignments if our extensions are not supported.

C++ provides means to eliminate padding and to control alignment. They overwrite built-in compiler heuristics [Hyde 2006]. However, they do not operate on the bit-level,

require manual intervention and quickly break code. MPI provides the means to wrap C++ classes into bespoke MPI datatypes. However, no genuine C++ integration exists, i.e. defining MPI datatypes requires bit-level address manipulation on the developer side and introduces significant syntactic overhead. The C++ language offers a small number of floating-point data types. Symbolic, high-level programming environments such as Matlab or NumPy support generic, flexible precisions such that developers can focus on methodological challenges [Carson and Khan 2023; Higham and Mary 2022]. However, it remains unclear to which degree the developed algorithms translate one-to-one into production-ready C++ code. Most multi-precision codes therefore stick to built-in precisions, i.e. rely on "specialisations" of generic algorithmic building blocks for few hardware formats (cmp. for example [Abdelfattah et al. 2021; Langou et al. 2006; Carson and Khan 2023]). In C++, template meta programming provides a mechanism to write precision-generic realisations. However, it works only over types which offer all operators used within the templated code. Template meta programming also introduces syntactic overhead compared to plain realisations with built-in data types, and it increases compile times. Introducing templates typically requires major code rewrites and ripples through the implementation. Finally, we can use bespoke libraries to provide support for multifaceted or flexible precision [Lindstrom 2014]. However, switching to user-defined data types (bespoke classes with higher information density and non-native floating point formats) runs risk to make a code incompatible with third-party libraries if they are not prepared to utilise different data types, while any introduction of such a type can hinder the compiler to perform further optimising memory rearrangements [Hyde 2006]. Our approach has none of these disadvantages and indeed can be read as an embedded domain-specific language (DSL) or language extension, which streamlines the development of numerical HPC code development.

On the long term, we expect many supercomputing projects to benefit from our ideas. For the present paper, we illustrate the potential impact by means of a simple smoothed particle hydrodynamics (SPH) code inspired by [Schaller et al. 2016; Schaller et al. 2023]. SPH is one well-established method for simulating fluids of complex structure [Lind et al. 2020] or a vast dynamic range [Price 2012] with moving particles. They are typically administered within a dynamically adaptive mesh. Maintaining the dynamically adaptive mesh plus the particle-mesh relations induces an (integer) data overhead. As the particles move, SPH requires frequent spatial resorting of particles between MPI ranks [Oger et al. 2016], while particles interact between MPI ranks each and every time step. The resorting typically requires the migration of the whole particle, while the exchange of few particle properties suffices to realise the particle-particle interactions in most SPH steps. We need different MPI data views, i.e. exchange different attribute subsets depending on the algorithmic context. SPH often suffers from strong scaling limitations. We have to keep the particles' memory footprint low to allow for bigger simulations. In addition, application scientists face pressure in their domains for ever-increasing simulation sizes and resolutions, which directly translates to the total number of particles they are able to fit onto a machine. That number is currently in the order of hundreds of billions and growing [Schaye et al. 2023]. As such, keeping the particles' memory footprint as low as possible is of vital concern. In this context, empirical evidence suggests that SPH particles hold some floating-point quantities which do not require full single or double precision [Hosono and Furuichi 2024]. On the whole, we consider SPH as a prime example of an application that benefits from our proposed C++ annotations. Many other application domains face similar challenges.

Our work is organised as follows: We first present our SPH use case in Section 2. The rough algorithmic sketch of SPH principles highlights some fundamental challenges arising from such codes. In Section 3, we discuss properties of a direct translation of

the algorithmic steps into plain C++, its properties, and what a better-suited implementation would look like. This allows us to introduce our new C++ annotations as well as the underlying code transformations triggered by them. The manuscript continues with a discussion of how these code transformations are realised within LLVM (Section 4). We return to the SPH demonstrator in Section 5 for a review of the potential impact of the extensions, before we assess the observed impact in Section 6. A brief outlook and summary in Section 7 close the discussion.
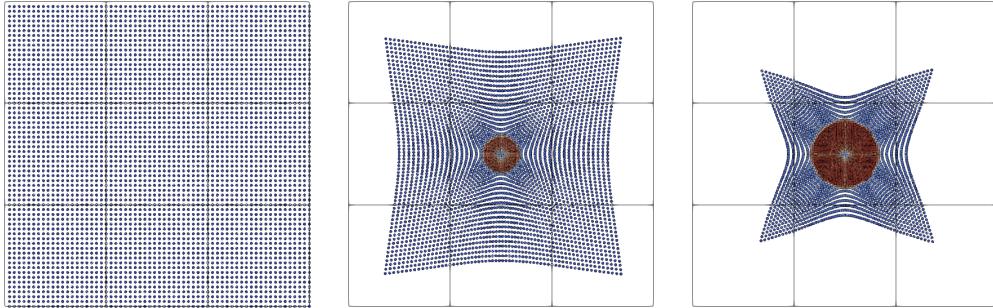
2. USE CASE: SMOOTHED PARTICLE HYDRODYNAMICS

Fig. 1. The Noh problem is a classic SPH benchmark: All fluid particles are initially positioned in a Cartesian layout (left) and are given a velocity towards the box center. The flow therefore develops a high-density region in the centre (middle), which eventually develops into a shock pushing fluid outwards again (right). Two-dimensional toy setup with 64^2 particles over a regular, time-invariant 3×3 grid.

Smoothed Particle Hydrodynamics (SPH) is used to model complex physical systems in a wide domain of computational sciences ranging from engineering to astrophysics [Gingold and Monaghan 1977; Monaghan 1992]. See [Lind et al. 2020] and [Price 2012], respectively, for state-of-the-art reviews on these two application domains. In SPH, a fluid of interest is discretised in terms of particles suspended in a computational domain, and the dynamics of the system are described by a set of equations for the interaction and evolution of these particles. SPH codes' physics are encapsulated within the implementations of the particle-particle interaction and the particles' evolution. As we have a finite number of particles, SPH discretises the continuous fluid flow into a finite number of equations. As each particle is equipped with a finite search radius and only particles within each others search radius interact, the arising discretised system of equations is sparse.

Conceptionally, SPH boils down to a temporal combination and arrangement of relatively simplistic steps per time step per particle:

(1) The density field of the fluid at the particle's position x_i is calculated based on the local distribution of particles. Around a given particle, only a compact set of particles, i.e. a neighbourhood, contributes to the value of the density $\rho(x_i) \equiv \rho_i$. Optionally, the size of the neighbourhood can be adjusted by solving a nonlinear implicit equation per particle that depends on the density.
(2) The particle's acceleration due to pressure gradients as well as the change in its internal energy are calculated. These calculations require information from the neighbours around each particle. Hence, their algorithmic intensity depends on the size of the neighbourhood.

(3) Finally, the particle's position, velocity, and internal energy are updated by integrating the equations of motion forward in time. Unlike the previous two steps, these updates do not require any exchange of information with any neighbour.

These three basic steps are typically complemented with some global reduction and broadcast phases, e.g. to identify the global admissible time step size.

Although only particle data structures are needed for SPH, most simulations use a grid—among other meta data such as Verlet lists or Cell Linked Lists [Domínguez et al. 2011]—as a helper structure to find neighbours efficiently. Binning the particles into a mesh allows us to search only through a small set of particles per time step for potential interaction partners: Two particles interact if and only if they are held within the same or two adjacent, i.e. vertex-connected, mesh cells. We use the grid as lookup mechanism. To make this work, the mesh cells have to have a size that is at least twice the maximal interaction radius of all particles held within the cell. Once the interaction radius, i.e. neighbourhood, changes, the grid should change, too, leading to the following additional algorithmic steps:

(5) As the particles move, we have to update the particle-mesh association. We have to resort.

(6) As the particles move and their density and interaction radius change, the mesh has to be adapted, i.e. refined and coarsened locally.

Our prime area of interest is cosmological simulations, where some particles move quickly, i.e. with a relative difference of several orders of magnitude compared to other particles contained in the simulation domain. These particles have to be resorted frequently. Therefore, we commit to an array of structs (SoA) data layout where the particles are administered within an adaptive Cartesian mesh.

The algorithmic sketch outlined above highlights that

— the adaptive mesh is a pure helper data structure lacking physical quantities. Storing, manipulating and maintaining these meta data are dominated by integers, enumerations and booleans.
— the attributes of a particle require various precisions. The cut-off radius determining the neighbourhood for example does not have to be very accurate, as it bounds the maximum particle-particle distance. We can equip it with a hard-coded safety factor. At the same time, domain experience suggests that the particles' position requires double precision, as the positions feed into complex non-linear evolution equations, while the densities vary by a factor of up to 10^{11} throughout the domain.
— we need at least three types of MPI data exchange: The density update requires us to exchange the density and neighbourhood search radii between ranks, the force calculation determines the acceleration and physical properties, and the actual particle update and resorting send whole particles comprising all fields around.

3. AN EXTENDED C++ LANGUAGE

C++ introduces an upper bound on the information density of structs that is significantly lower than the theoretical maximum. We work with minimal memory chunks of bytes. If a byte, an 8-bit entity, stores a boolean value which could be represented by one bit (on/off), the information density is only 12.5% (1/8). If that boolean is followed by another struct aligned at eight-byte addresses, it is attached an additional seven bytes, lowering its information density to 1.56% (1/64). We can make an analogous case for floating-point data were the actual number of meaningful, i.e. significant, bits known to the developer. The information density will likely stay under 100%, i.e. not all bits hold meaningful information.

Memory alignment and the padding are key to make code fast: They ensure that loads and stores hit memory entries exclusively, they ensure that the machine can work efficiently with cache lines, and they ensure that vector operations can load small vectors en bloc into the respective vector registers. However, many HPC codes are notoriously memory-bound, or they suffer from cache and memory latency due to scattered data access. A lower information density makes this situation worse.

The C++ language and compiler vendors offer ways to increase the information density of user-defined data structures. On the one hand, the language supports the notion of bit-fields with a user-defined number of bits. The `bitfield` class allows developers to pack multiple booleans together into one primitive datatype (integer). It tackles an extreme case of memory "waste". On the other hand, many compilers (gcc and Clang/LLVM, among others) support the `__attribute__((packed(N)))` and `__attribute__((aligned(N)))` syntax to allow users to manually control memory alignment, and newer C++ generations even support explicit alignment, e.g. through `alignas` and `std::aligned_alloc`. The solutions tackle special cases (sets of booleans) or provide byte-level control over some memory arrangements to the user.

We propose to extend C++ such that developers can squeeze out fill-in bits and bytes, exploit knowledge of potential value ranges of integers and inform the compiler of the required floating-point accuracy. This works per attribute. Further to that, we propose to add support for the explicit modelling of multiple MPI data views over each struct.

Design Decision 1. *Our extensions are language augmentations. They neither introduce dependencies on external libraries nor require rewrites of the underlying code.*

Design decision 1 assumes MPI to be omnipresent on an HPC system and does not count it in as an external library.

Design Decision 2. *Our extensions are semantics-preserving: As long as the assumptions expressed through an annotation remain valid, the extensions do not alter the code's behaviour.*

For integer data, Design decision 2 is strict. For floating-point data, we have to discuss the precise meaning of semantics-preserving.

Design Decision 3. *Our extensions are optional. If they are not supported by the C++ toolchain, they are ignored.*

We realise our language extensions through C++ attributes. If a compiler is unaware of particular attributes, they are simply ignored. We define additional compiler passes which map the attributes onto plain C++ instructions internally. No external libraries are required. As the extensions are prototypically implemented within LLVM, we make the attributes start with the `clang::` prefix.

Design Rationale 1. *The acceptance of C++ language extensions hinges upon the fact whether users can introduce and benefit from the extensions without (a) code rewrites and (b) tying their code to external libraries. Since the annotations preserve the semantics of the plain underlying code and as they are optional, developers can evolutionary augment their code in a trial-and-error fashion.*

Design Rationale 2. *Interpreting all extensions through additional compiler passes ensures that their realisation (a) benefits from all optimisation know-how within the compiler and (b) remains agnostic of the target architecture, i.e. back-end.*

Our objectives, i.e. an increased information density and reduced memory footprint, could also be achieved through a library hosting tailored C++ classes. However, this approach is not minimally invasive, i.e. entails major code rewrites compared to vanilla

C++ (cmp. Rationale 1). With template meta programming, developers can write type-generic algorithm realisations. However, this introduces additional syntactic (template) overhead, it prolongs compile times, and requires the library's high-density classes to offer each and every operand that might be used over built-in types. The templating also quickly ripples through the code base, as all functions used have either to be specialised or generalised to offer support for user-defined data types. As an example, a simple use of `std::min` quickly fails for bespoke user types, and becomes nasty once a user-defined class is used in combination with a built-in type. Finally, a realisation through a library implies that certain compiler optimisations become less straightforward. Compilers still struggle to optimise very large code blocks introduced by templated functions as optimisation spaces grow exponentially, and simple techniques such as attribute reordering [Hyde 2006] might become obstructed.

3.1. Memory compactification

C++ programmers pick a well-suited built-in datatype for their integers, i.e. a qualified variant of `int`. Once developers know the range of values encoded within an integer, we can argue whether a choice of a variant of `int` is valid, i.e. large enough to host all potential values' bit codes. Enums are symbolic identifiers for integer values. Consequently, we know the exact number of bits required to encode any element of an enum declaration. Booleans are integers with a value range from $\{0, 1\}$. They carry one meaningful bit.

Motivation. Whenever we work with integers of limited range, enumerations or small bit sets, the information density of plain C++ code is low. Scientific simulation software is often not integer-heavy. Yet, we note that integer arithmetics are used within meta data structures, e.g., trees, containers, search algorithms, lookup tables, which are performance critical. Data access latency penalties caused by memory footprints of structs that are larger than a cache line therefore are problematic.

Further to that, we note that integer data scattered among structs dominated by floating-point data have the potential to inflate struct encodings [Hyde 2006], as they insert padding bytes just before the floating-point numbers. In such a case, arrays of structs become hard to vectorise as gather and scatter operations start to span many different cache lines.

Reducing the memory footprint of integer-heavy structs hence is important. Since integers often feed into control logic, any reduction has to preserve all data bit-wisely.

Lossless compression. Let *packing* be a lossless compression, where multiple integer-valued variables are stored within one large bitfield. This bitfield has no bits without semantics.

C++ Annotation 1. *We introduce C++ attributes that label integer, boolean and enumeration members of structs as candidates for packing. For integers, a range of values, i.e. upper and lower limits can be specified.*

Given a set of labelled integer attributes with known range—this includes enumerations and booleans—a compiler can construct one large bitset which encodes this sequence of values with high information density. The attributes apply to scalars and multidimensional arrays with known array ranges (Algorithm 1).

— The `[[clang::pack]]` attribute applies to datatypes that can be packed automatically without any additional user-supplied information. These are booleans and enumerations as well as fixed-sized arrays of these. We implicitly know their underlying integer ranges.

Algorithm 1 Syntax of the C++ extensions for integers, enums and booleans in pseudo Backus–Naur form. C++ keywords are set bold, while non-bold names are examples of identifiers following the ISO C++ conventions. Entries (. | . | .) enlist alternatives, while double square brackets [[.]] embrace C++ annotations. The new attributes are called `pack` and `pack_range`. Uppercase identifiers L, M, N, MIN, MAX are to be replaced with compile-time constants in real code.

```
(struct | class) Data {
  [[ clang :: pack ]]
  (bool | enum) field1;

  [[ clang :: pack ]]
  (bool | enum) field2[M][N];

  [[ clang :: pack_range(MIN, MAX)]]
  ( (signed | unsigned) (char | short | int | long | long long) )  field3;

  [[ clang :: pack_range(MIN, MAX)]]
  ( (signed | unsigned) (char | short | int | long | long long) )  field4[L];
}
```

— The `[[clang::pack_range(MIN,MAX)]]` attribute controls packing of integer attributes and constant-sized arrays thereof. Since the compiler does not know a priori how many bits such a field uses, we ask users to manually provide a range of values that a field must be able to support.

Mapping onto plain C++. Our compiler determines the number of bits that are required to store packed values without losing any data. That is one bit for each boolean and $\lceil \log_2(n) \rceil$ bits for enumerations with n alternatives. Integer datatypes which are annotated with `[[clang::pack_range(MIN, MAX)]]` can be packed losslessly with a bit footprint of $\lceil \log_2(\text{MAX} - \text{MIN}) \rceil$.

Design Rationale 3. *The technical details how integers of limited range are packed into one big bitset have to be hidden (cmp. Rationale 1).*

Throughout the compilation, our compiler "removes" packed attributes from their struct, and it inserts one large bitfield that can accommodate all of their required bits. Code access to these values are wrapped into appropriate bit masking, i.e. we pick the right bits from the large bitset. Arrays of packed integers can be mapped onto sequences of entries within the bitset as long as the array size is known at compile time. If enumerator values are manually assigned yet do not span a continuous range, the compiler generates a lookup table to map them onto a compact range prior to packing.

C++ context. Packed values can coexist with unpacked values within one data structure. None of the characteristics of unpacked values are changed by the presence of packed values. Notably, only packed values lose their native memory alignment. However, the packing can permute the ordering of the struct attributes in the memory: The order of attributes of a struct in memory usually follows the order of their declaration. Consequently, performance guidebooks recommend to order the attributes large-to-small [Hyde 2006]. Our annotations make the compiler extract attributes from the struct. They are, eventually, inserted via a large bitset at the end of the struct's memory. We change the attribute ordering.

Taking pointers or references of packed fields is not possible and fails with a compilation error. An attempt to store a value that falls outside of the specified range of

a packed field is undefined, as the annotation's underlying assumptions are violated (cmp. Rationale 2).

Packing is only supported for built-in types. Structs within structs cannot be annotated with packing, although their built-in attributes in turn can be subject to packing.

Impact. The information density of a struct hosting a packed attributes is equal or higher than the struct realisation in plain C++. As the memory footprint is reduced, we expect memory-bound compute kernels to benefit from better cache utilisation. However, any access to packed data is subject to additional conversion effort. We assume that the underlying bit shifts and masking operations are fast on modern hardware. Yet, it is not clear a priori what performance impact the padding has.

Our approach facilitiates explicit unpacking and packing. A simple `int a = packedA` over a packed integer `packedA` will convert the bitset information into a native `int`. Subsequent accesses to such a variable `a` will not suffer from any conversion penalty. The synchronisation back into `packedA` however remains with the user, i.e. the user has to manually copy the updates value back. The corresponding `packedA = a` however will let the compiler automatically introduce all required packing operations.

3.2. Floating point storage precision

C++ programmers pick a well-suited built-in datatype for their floating-point numbers when they write numerical algorithms. Traditionally, this is either `float`, `double` or `long double`, though alternatives such as fixed width floating-point types or library-induced further types become increasingly popular. The choice is guided by forward/backward stability arguments and the precision required in the output.

Motivation. Supercomputers broke through the exascale wall twice: First in half precision and later in double. The higher throughput in half precision results from improved vector computing capabilities, but also from a reduced pressure on the memory subsystem due to a smaller memory footprint. As machines yield significantly higher performance for reduced precision, new (competing) floating-point formats become supported by hardware [Lindstrom et al. 2018], and scientists recast algorithms into mixed precision formulations, where as many computationally expensive steps as possible are rewritten with lower precision data types (cmp. [Bornemann et al. 2004; Carson and Higham 2018; Carson and Khan 2023; Higham and Mary 2022; Ichimura et al. 2018; Langou et al. 2006; Murray and Weinzierl 2020] and many others). Nevertheless, computationally intense compute kernels remain notoriously memory-bound, while we continue to work with overspecified data formats in many cases. The number of native floating-point formats within the language is too small to tailor the memory footprint of each variable precisely to its significant bits. We "over-invest" in bits.

At the same time, projects start to identify cases where data logically does not exhibit the information density provided by native floating-point formats: Some data arising in intermediate compute steps [Bungartz et al. 2008; Bungartz et al. 2010; Carson and Khan 2023] or streamed into post-processing [Diffenderfer et al. 2019; Lindstrom 2014] do not "need" all the significant bits, i.e. many bits carry no physical meaning [Abdelfattah et al. 2021; Tsai and Sanchez 2019].

Reducing the memory footprint of floating-point data beyond the few available formats made available by the hardware hence is timely and important to decrease the bandwidth requirements of codes further and to release stress from the last-level cache.

Lossy compression. Since we work with numerical approximations of real numbers, we leverage any user intelligence on the mantissa's information density.

C++ Annotation 2. *We introduce a C++ attribute that enables developers to specify the number of significant, i.e. relevant bits in the mantissa of a floating-point value within a struct. This qualifies the floating-point variable for packing.*

For floating-point variables with known significant bits, a compiler can extract these significant bits from the floating-point representation and store the bits within a bitset rather than the full `float` or `double`. In our C++ augmentation, the attribute `[[clang::truncate_mantissa(BITS)]]` specifies, for any native C++ floating-point type, that the actual mantissa can be stored with only `BITS` bits. The exponent and the bit for the sign are preserved with their original bit counts. Our attribute applies to scalars and multidimensional arrays with known array ranges (Algorithm 2).

Algorithm 2 Syntax of the C++ extension for the mantissa (exponent) truncation of floating-point numbers. `BITS`, `M` and `N` are integer constants known at compile time.

```
(struct | class) Data {
        [[clang::truncate_mantissa(BITS)]]
        (float | double | long double) field1 [, fieldArr[M][N]..., ...];
}
```

Mapping onto plain C++. The extracted mantissa bits plus the sign bits and the exponents are packed into a large bitset together with all the enums, booleans and integers which carry a `[[clang::pack]]` attribute. Our floating-point packing integrates seamlessly with the integer packing.

Design Rationale 4. *For performance reasons, calculations have to stick to built-in data formats (cmp. Rationale 2). However, developers often have expert insight how many significant bits their data really encode in-between calculations.*

We continue to run all calculations in native precision: Our extension specifies how data are stored, but these formats are converted back into the native C++ datatype prior to calculations. Therefore, our approach is lossy and realises a compress-decompress pattern.

Design Rationale 5. *Whenever external functions are invoked, the compressed data are automatically converted into native floating-point numbers.*

The compression therefore does not propagate through the code. Major rewrites do not ripple through the callstack.

C++ context. As we store all compressed floating-point values internally within bit fields, we inherit all properties of the packed integers, including the fact that referencing via pointers is not possible and fails with a compilation error. As we preserve the range of the exponent, it is impossible to create an additional overflow compared to the baseline code version ignoring the C++ attribute. However, the attribute can introduce additional underflows for very small quantities, and it can amplify the truncation error.

Further to the reduced precision and potential losses of significant bits, the packing has the potential to change the semantics of codes which employ logic over floating-point data: C++ guarantees $a \not< b \Rightarrow a \geq b$. Let $\hat{a}$ and $\hat{b}$ be compressed variants of a or b, respectively, and let

$$p(a,b) = \begin{cases} p_<(a,b) & a < b \\ p_\geq(a,b) & \text{if } a \geq b \end{cases}$$

be written down as C++ if-else statement. For $a < b$ reasonably close, we might preserve $\hat{a} < \hat{b}$ or end up in a situation where $\hat{a} = \hat{b}$ due to the truncation. The truncation shifts and reduces the representable data points of `double` and `float` within $\mathbb{R}$.

Impact. The language extension realises a lossy compression. Among such techniques, there are approaches which preserve all the bits of the exponent [Tagliavini et al. 2018], and approaches which also reduce the bits per mantissa (compare IEEE's half precision vs. single). Our approach preserves the exponent to be able to cover the same range as the original data format. We hence spread out the discrete data points within $\mathbb{R}$ that can be represented compared to the baseline type.

For selected problems, mainly from the linear algebra world, one can show that sophisticated rewrites with reduced sample accuracy over $\mathbb{R}$ do not compromise the solution [Bornemann et al. 2004; Carson and Higham 2018; Higham and Mary 2022]. For other problems, empirical data suggest that reduced precision is sufficient [Eckhardt et al. 2015; Fortin and Touche 2019; Weinzierl and Weinzierl 2018]. In general, stability and error propagation have to be studied carefully.

Similar to integer data packing, floating-point packing induces operations overhead. We have to unpack it from the input bitfield and befill the floating point registers prior to the actual computation. In particular, floating point sequences cannot be loaded "en bloc" from the memory into vector registers due to this conversation and the fact that we miss out on alignment or padding. We assume that savings in memory transfers and bandwidth have the potential to compensate for this penalty; notably if developers read from and write to packed structs carefully.

Vendors add support for reduced precision calculations to their chips. This is primarily driven by artificial intelligence [Agrawal et al. 2019]. Our extension does not advocate for reduced precision calculations, since it continues to work with standard C++ types for all calculations. It however works hand in hand with modifications of the core calculations or template meta programming to facilitate precision-generic codes.

3.3. MPI datatypes over structs

C++ developers pick a well-suited distributed programming model for their code manually, as C++ has no built-in support for this. MPI remains the de-facto standard for distributed memory codes in high-performance computing. It "natively" facilitates the exchange of scalars of built-in types, as well as arrays of these. For more complex data structures, manual work is required.

Motivation. Modern MPI supports user-defined datatypes [Gropp et al. 2014]. They cover structs hosting scalars and arrays of different types which are not contiguous in memory. We can also define an MPI datatype over a struct which covers only some of its attributes. This allows developers to exchange structs and arrays thereof partially, instead of serialising and exchanging all information independent of whether data is needed or not. It allows developers to maximise the information density on a communication stream.

Introducing user-defined datatypes requires developers to use low-level operations: We create an instance of the struct of interest, extract the relative addresses of the struct's attributes into a table, and commit the table to MPI as a new datatype. The datatypes encodes how a struct is serialised, i.e. how its bitstream is broken down into arrays of primitive types. Changes in a struct's number of attributes, types, their order

or the underlying type inheritance hierarchy require the maintenance of all "derived" MPI datatypes. It is laborious.

Creating and using bespoke MPI datatypes that only exchange required information is timely. Networks on supercomputers notoriously suffer from congestion and bandwidth restrictions, and hence throttle scientific codes.

Embedded MPI datatypes. We introduce a C++ attribute that enables developers to automatically create a factory method [Gamma et al. 1994] returning an MPI datatype. This datatype may encode an arbitrary subset of attributes of the struct.

C++ Annotation 3. *We introduce a C++ attribute that enables developers to annotate a member function of a struct to highlight that this function returns an MPI datatype. The function's existing implementation—if available—is replaced by a generated routine.*

Our C++ extension (Algorithm 3) streamlines the construction of MPI datatypes:

— A function annotated with `[[clang::map_mpi_datatype]]` has to return an `MPI_Datatype` and may not accept any arguments. It has to be `static`.
— If our compiler encounters a method annotated with this attribute, it generates an implementation of a factory method [Gamma et al. 1994]. Upon its first invocation, the routine constructs an MPI datatype. After that, it returns this datatype.
— An existing function implementation is replaced by the compiler.
— Via `[[clang::map_mpi_datatype("a","b",...)]]`, developers can instruct the compiler that the generated `MPI_Datatype` should cover only some object attributes a, b, ... The subtypes have to be primitive, i.e. have to be supported by MPI natively. Without these selectors, the factory method's return datatype comprises all attributes of a struct, i.e. it serialises the whole struct.
— If structs are contained within structs (nested), the default MPI datatype covers the whole conglomerate. Users however can pick subattributes of arbitrarily nested structs through `struct1.struct2.attribute`.
— If the function attribute enlists one packed integer or floating-point attribute, all packed attributes of the struct are subject to the generated MPI datatype.

Algorithm 3 Struct with MPI augmentation. The compiler generates the implementations of an augmented routines, replacing their user implementations. Both generated routines return an `MPI_Datatype` which can directly be used with `MPI_Send` or `MPI_Recv` or any MPI routine. The first datatype exchanges all attributes of `Data`, while the second datatype exchanges only two attributes of the struct.

```
struct Data {
  [[ clang :: map_mpi_datatype ]]
  static MPI_Datatype getMyFullMPIDatatype ();

  [[ clang :: map_mpi_datatype ( field1 , field2 . subfield1 )]]
  static MPI_Datatype getDatatypeForSubset ();
}
```

Mapping onto plain C++/MPI. The generated code takes care of all address arithmetics and the construction of helper data structures to describe the MPI datatype. Hiding the technical complexity behind MPI datatypes is not a new endeavour or idea, and there are different ways to achieve this: Boost for example supports data exchange of structs via byte code serialisation. Here, the serialisation is realised through

routines of a pre-defined name which are injected into the struct. This is an aspect-oriented approach. Our approach does not serialise the objects directly, but instead maps the struct attributes onto a MPI dataype, i.e. the actual serialisation is delegated to the MPI library.

We define the construction of the MPI datatypes to be lazy, i.e. they are generated upon the first invocation of the routine. This ensures that the MPI datatype construction does not precede any MPI initialisation. Even if the datatype is hosted within a library, its construction happens upon the first invocation of the factory method, i.e. after the code using the library has established the MPI environment. It remains the responsibility of the developer to clean-up (free) user-defined MPI datatypes created via our factory methods.

C++/MPI context. Since we extract the MPI datatypes from the source code at compile time, data format changes are automatically reflected within the MPI datatype generation. The extension implicitly flattens any inheritance hierarchy, although it does not support any polymorphism within MPI. The MPI datatype construction masks out the `vtable`, but it does not distinguish any particular subtypes. As a static routine, the resulting MPI datatype is tied to one particular class.

Our extension assumes that developers continue to work with MPI directly. Users have to know which datatype they send and receive in turn. The augmentation provides data types only and no other MPI features.

By default, all fields are included in the generated `MPI_Datatype` instance. However, developers can explicitly specify which fields should be included by listing them as attribute arguments. This enables developers to create multiple tailored `MPI_Datatypes` per struct, since we tie the datatype construction to a static member function rather than the struct itself. Developer can create multiple *views* over their structs:

Design Rationale 6. *To keep data consistent between ranks, many codes have to exchange some attributes of structs only. Which attributes to pick can depend on the context (algorithmic phase, e.g.).*

As the annotation triggers the compiler to replace any existing implementation of the annotated function, users can guarantee that their code continues to be correct even if the annotations are not supported (cmp. Rationale 2 and Rationale 3). For this, they have to provide a dummy realisation of the static function, e.g. serialising the whole struct independent of the view chosen.

Impact. The MPI annotation in first place is convenient for developers and can be used independently of the packing. While the ease argument is important in itself, it becomes particularly important once we support the previously introduced packed integer and floating-point data types, for which no MPI data type equivalent exists. It frees developers from the duty to care about the existence and implications of packing.

The concept of views aims to reduce the bandwidth pressure on the node interconnects, as we eliminate waste bytes or increase the information density of the exchanged data stream, respectively. Picking individual attributes that are to be exchanged from a struct or an array of structs means that MPI has to gather and scatter data from the memory. It is not clear how expensive these steps are.

However, our annotations wrap around MPI and do not interfere in any way with the standard. Therefore, any MPI optimisation carries over to our annotated code directly. Notably, concepts such as message compression [Filgueira et al. 2012; Ke et al. 2004] or sophisticated message buffering are not compromised by our techniques.

4. REALISATION WITHIN LLVM

LLVM is the baseline of many vendor-specific mainstream compilers (Intel, NVIDIA and AMD). Due to its clear separation-of-concerns, its explicit intermediate program representations, and a clear data flow through translation passes, Clang/LLVM is a natural candidate to realise our extensions.

We fork LLVM 13.0.0. Within this fork, all language extension are supported by default, though we can instruct the compiler to ignore them through `-fno-hpc-language-extensions`. Depending on the invocation, the compiler defines or undefines the symbols `__PACKED_ATTRIBUTES_LANGUAGE_EXTENSION__`, `__MPI_ATTRIBUTES_LANGUAGE_EXTENSION__` and `__AOSSOA_ATTRIBUTES_LANGUAGE_EXTENSION__` such that users can mask out code fragments through `ifdef` guards.

4.1. Extension architecture

LLVM is a modern compilation framework breaking down the translation into stages or phases. For our work, Clang serves as compiler frontend. It translates the (annotated) C++ source code into LLVM's intermediate language/representation (LLVM IR). This LLVM IR then is subject to optimisation passes and eventually streams into the (multi-target) machine code generation.

Many embedded DSLs add an additional level of abstraction on top of the generic programming language C++ and hence require front-end, i.e. lexer and parser, modifications. Our language extensions use C++'s annotations. We can therefore stick to an unmodified font-end to build up the abstract syntax tree (AST), and manipulate this AST before we lower it into plain LLVM IR.

Design Rationale 7. *Since we realise our extensions through an additional compile pass following the parsing, they become independent of both the IR optimisations and target-specific machine code production, as well as any C++ front-end modifications.*

Clang's high-level architecture follows a textbook compiler structure [Kruse 2021]. A SourceManager and FileManager handle file-related operations. The Preprocessor and Lexer run through the files' byte streams and produce tokens which are used to identify syntactic elements. They are handed over to the semantic analysis (Sema) which yields an abstract syntax tree (AST). The Sema's TreeTransform helper mechanism adds additional AST nodes besides those corresponding directly to parsed tokens: Each implicit template instantiation for example creates its own copy of the AST subtree into which it substitutes template parameters. We use an analogous mechanism to realise the transformations triggered by the annotations. If an attribute is marked as packed, we replace all follow-up accesses with the corresponding packing or unpacking code.

Yet, Clang favours forward propagation of information in line with LR(k) grammars and the C++ language which is static and strongly typed, i.e. requires all types and variables to be well-declared prior to their first usage. It is tied to single-pass translation. Consequently, Clang's tree transformations support localised alterations, such as changing AST nodes as they are created or unfolding of subtrees. Cross-references are eliminated in the tree generation phase by replicating information (such as datatype, type size or memory alignment) where required. Altering declarations in hindsight is not possible.

For our language modifications, we have to add or remove struct fields, or change types of declared variables. The exact bitfield layout, for example, is only known after we have parsed the whole underlying struct. At this point, we might already have processed (in-line) source code snippets. Our extensions potentially require non-local changes rippling through many data copies within the AST.

Our realisation therefore abandons the single-pass paradigm and instead uses in-memory pretty printing: In a preparatory phase, we traverse the tree and search for our domain-specific attributes. The set of attributes yields a source transformation plan, i.e. recipies which fragments of the underlying source code have to be changed. With these rules, our compiler extension reparses the code and builds the AST again. This time, it alters AST nodes that need to be changed immediately, and therefore propagates the changes to all replica of AST parts or code using the altered AST segments. This happens entirely in memory.

4.2. Packing realisation

We store packed integer data as sequence of values with a_i bits, where $a_i = \lceil (\log_2(n_i)) \rceil$. n_i is the number of potential values of each variable. The resulting memory footprint is $\sum_i \lceil (\log_2(n_i)) \rceil \leq \lceil \log_2 \prod_i (n_i) \rceil$. We refrain from "merging" the ranges of multiple variables, as this would introduce additional arithmetic overhead when we generate the data access operations. Without further assumptions, the information density within the packed bitset is therefore not optimal. We choose simplicity over the theoretical maximum of the information density.

Our floating-point annotations support the C++ floating-point formats `float`, `double` and `long double`. Besides scalar attributes, the compiler can handle constant-sized arrays of arbitrary dimensionality over these types. Bit-shifting and bit-masking over floating-point values are not natively supported by the C/C++ language. Our tool overcomes this obstacle by "dereference casting" of the floating-point values to and from integer representations. Throughout this process, mantissa bits are cut or added. The implementation of the conversion is realised as part of the same compiler pass that handles the `[[clang::pack]]` and `[[clang::pack_range(BITS)]]` attributes, i.e. floating point manipulations are directly forwarded into the logic handling integer packing.

Our conversion is a plain truncation, i.e. we chop the digits after the `BITS`th position off. When the truncated representation is retranslated into a native format, the previously truncated bits are set to 0 in the reconstructed value. Such a strongly biased conversion can lead to accumulation effects and make numerical implementations unstable. We recognise that techniques such as stochastic rounding [Croci et al. 2022] could mitigate this phenomenon [Fasi and Mikaitis 2021; Connolly et al. 2021] yet are out of scope here.

Design Rationale 8. *Numerical accuracy or stability considerations are out of scope for the present work, i.e. we solely rely on the user to keep track of these phenomena.*

Within the translation pipeline, any access to a packed variable a results in some implicitly generated conversion code from or to a's packed data representation. Any explicit copy b=a implies however that b is *not* packed anymore. A statement a++ over a packed variable hence unpacks and packs implicitly, i.e. synchronises the packed variable with its temporarily unpacked variant (cmp. Sections 3.1 and 3.2).

Design Rationale 9. *The packing attributes do not propagate through in the code, i.e. they do not apply to copied variables.*

The "do not propagate" policy allows developers to eliminate any packing from attributes explicitly by simply copying them into temporary variables. Compilers remove such helper copies if the new attributes are not supported, unless they are used within very large compilation units where the additional helper assignments push the total code size beyond the heuristics from which on the compiler stops optimising.

Implicit conversion for read and write accesses imply that our approach works seamlessly for third-party functions accepting built-in datatypes. A simple a = std::min(b,c) over packed floating-point values triggers two unpack and one pack

operation in the background, yet does not require any bespoke realisation of `std::min`. It also continues to work if either `a`, `b` or `c` are unpacked, native data. This makes our approach differ from a template-based solution, where `std::min` would have to be overloaded for packed specialisations and combinations of packed and unpacked data.

4.3. MPI datatype mapping

The MPI code generation triggered by `[[clang::map_mpi_datatype]]` invokes `MPI_Type_create_struct`. Prior to this, it gathers the block lengths, i.e. the continuous occurrences (array lengths) of a given type, relative offsets of these arrays over primitives within the memory, and the (MPI) types themselves into a map. To populate the map, we recursively traverse the AST, starting from the `CXXRecordDecl` node that describes the struct which declares the mapping method.

The `MPI_Datatype` created by the mapping methods is cached within the generated routine in a `static` local variable, such that the actual call to create the MPI datatype (`MPI_Type_create_struct`) and the corresponding `MPI_Type_commit` happen only once regardless of the number of invocations of the mapping method. Users have to ensure themselves that they invoke `MPI_Type_free` appropriately.

4.4. Overhead in machine code

Algorithm 4 Top: We label attributes as `pack`, while all remaining accesses to the attribute remain unchanged (compare left to right side). The compiler maps all packed attributes into one big bitset and automatically replaces access to these fields with the corresponding bit arithmetics. Bottom: Assembly instructions of baseline code (left) vs. the packed variant (right) as emitted by our compiler with the `-O3` flag.

```
struct Data {                            struct Data {
   bool b;                                  [[clang::pack]] bool b;
};                                       };
bool getB(Data &data) {                  bool getB(Data &data) {
   return data.b;                           return data.b;
}                                        }
void setB(Data &data, bool val) {        void setB(Data &data, bool val) {
   data.b = val;                            data.b = val;
}                                        }
void invertB(Data &data) {               void invertB(Data &data) {
   data.b = !data.b;                        data.b = !data.b;
}                                        }
```

```
getB(Data&):                             getB(Data&):
      mov     al, byte ptr [rdi]               mov     al, byte ptr [rdi]
      ret                                      and     al, 1
                                               ret
                                         setB(Data&, bool):
setB(Data&, bool):                             mov     al, byte ptr [rdi]
      mov     byte ptr [rdi], sil            and     al, -2
      ret                                    or      al, sil
                                               mov     byte ptr [rdi], al
                                               ret
invertB(Data&):                          invertB(Data&)
      xor     byte ptr [rdi], 1              xor     byte ptr [rdi], 1
      ret                                      ret
```

Packing and unpacking translate into few machine instructions that are inlined into the resulting code. In the case of a packed boolean, the overhead is just one x86_64

Algorithm 5 Packing and unpacking (top, left vs. right) introduce only few additional operations in the resulting machine code (bottom).

```cpp
struct Data {

    float f;
};
float getF(Data &data) {
    return data.f;
}
void setF(Data &data, float val) {
    data.f = val;
}
void add(Data &data, float val) {
    data.f += val;
}
```

```cpp
struct Data {
    [[clang::truncate_mantissa(7)]]
    float f;
};
float getF(Data &data) {
    return data.f;
}
void setF(Data &data, float val) {
    data.f = val;
}
void add(Data &data, float val) {
    data.f += val;
}
```

```asm
getF(Data&):
        movss    xmm0, dword ptr [rdi]
        ret

setF(Data&, float):
        movss    dword ptr [rdi], xmm0
        ret

add(Data&, float):
        addss    xmm0, dword ptr [rdi]
        movss    dword ptr [rdi], xmm0
        ret
```

```asm
getF(Data&):
        movzx    eax, word ptr [rdi]
        shl      eax, 16
        movd     xmm0, eax
        ret

setF(Data&, float):
        movd     eax, xmm0
        shr      eax, 16
        mov      word ptr [rdi], ax
        ret

add(Data&, float):
        movzx    eax, word ptr [rdi]
        cvtsi2ss         xmm1, eax
        addss    xmm1, xmm0
        cvttss2si        eax, xmm1
        mov      word ptr [rdi], ax
        ret
```

---

machine code instruction for reading (unpacking), and three instructions for writing (packing), and no overhead for in-place inversion (negation) (Algorithm 4).

For floating-point data, we obtain two extra x86_64 machine code instructions for either a read or a write operation, and three extra x86_64 instructions for an "in-place" arithmetic operation, i.e. a read immediately followed by a write (Algorithm 5).

5. THE SPH DEMONSTRATOR

5.1. Governing equations

The Lagrangian philosophy behind SPH—in which the fluid is mapped onto particles—recasts the partial differential equations governing the dynamics of the system into a set of coupled ordinary differential equations. They describe the interaction and evolution of these particles. In this Section, we present only the core equations of the numerical method we use, while the full description of the governing equations is presented in Appendix D.

At the heart of SPH is the smoothing operation which is used to estimate scalar fluid quantities such as the density ρ_i for each particle i. Giving each particle some (constant) mass m_i, the smoothed density is obtained via

$$\rho_i = \sum_j m_j W_{ij}(H_i) \tag{1}$$

where $W_{ij}(H_i) = W(\mathbf{x}_j - \mathbf{x}_i, H(h_i))$ is called the *kernel*. It's a smooth, differentiable, spherically symmetric, and monotonically decreasing function with compact support of radius H. In practice, kernels are computationally inexpensive polynomials. Although the sum in (1) runs, in principle, over all particles j in the domain, the finite $H(h)$ reduces it to a loop over neighbours around $\mathbf{x}_i$.

The smoothing length, h, plays a central role in SPH. It determines the compact support H and hence defines the concept of neighbours, i.e. it defines the group of particles which are close enough to contribute towards the value of the field, Hence it determines the number of neighbouring particles included in smoothing operations such as (1). Furthermore, it also specifies the spatial resolution of the simulation [Dehnen and Aly 2012].

For the present SPH demonstrator, we consider an inviscid fluid in the absence of gravity and external forces or energy sources. Hence, the individual particles tracking the fluid evolve according to the Euler equation,

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_j m_j \left[f_i \frac{P_i}{\rho_i^2} \nabla W_{ij}(H_i) + f_j \frac{P_j}{\rho_j^2} \nabla W_{ij}(H_j) \right] + \mathbf{a}_i^{\mathrm{AV}}, \tag{2}$$

while the thermodynamic internal energy per unit mass of the fluid, u_i, evolves according to

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = f_i \frac{P_i}{\rho_i^2} \sum_j m_j (\mathbf{v}_i - \mathbf{v}_j) \cdot \nabla W_{ij}(H_i) + \dot{u}_i^{\mathrm{AV}}. \tag{3}$$

$\mathbf{v}$ is the velocity field, P is the pressure and $\nabla \equiv \partial/\partial \mathbf{x}$ is the spatial gradient. The system is closed by specifying the equation of state of the fluid, $P = (\gamma - 1)u\rho$, in which γ is the adiabatic index.

The used equations include physical quantities of the fluid plus terms that are intrinsic to the SPH method. The scalar field

$$f_i = \left(1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right)^{-1} \quad \text{with} \quad \frac{\partial \rho_i}{\partial h_i} = \sum_j m_j \frac{\partial W_{ij}(H_i)}{\partial h_i} \tag{4}$$

represents the spatial fluctuations in the smoothing length $h(\mathbf{x})$ (typically known as 'grad-h' terms). They have to be taken into account whenever h is allowed to change over space and time. Formulations with such variable h are crucial in astrophysical applications, where the fluid can be strongly compressed (over a range of several orders of magnitudes).

Finally, following [Monaghan 1992; Balsara 1995], and [Price 2012], we add an artificial viscosity (AV) to the (physically inviscid) fluid in order to resolve potential discontinuities (e.g. due to shocks) that could develop in the fluid. In particular, we adopt the AV model used by the GADGET-2 code [Springel 2005]. The additional terms are reflected as $\mathbf{a}_i^{\mathrm{AV}}$ and $\dot{u}_i^{\mathrm{AV}}$ in eqns. (2) and (3), respectively, and are fully described in Appendix D.

5.2. Particle organisation within a dynamically adaptive Cartesian mesh

The elegance of SPH results from the localization of the interaction: Particles closeby exchange information, particles that are far away from each other do not. To exploit this in a code, it is crucial to evaluate neighbourhood queries (which particle is close) efficiently:

We employ a dynamically adaptive Cartesian mesh based upon a space-tree [Weinzierl 2019] as meta data to speed up the neighbourhood search. Our compu-

tational domain is embedded into a cube. We cut this cube into 27 subcubes. Per subcube, we decide recursively and independently whether to refine further. This yields a tree hierarchy of adaptive Cartesian meshes. Within this hierarchy, we make each particle belong to the finest cube resolution hierarchy with a cube length of at least $CH_i(t)$ with a hard-coded constant C, and assign it to the closest cube vertex [Weinzierl et al. 2015]. This assignment scheme yields a natural refinement and coarsening criterion: A cube is refined further if one of its vertices hosts at least K particles which would fit into the next finer resolution level, too. Cubes are removed if a set of 3^d children of one large cube host fewer than K particles. K is a tuning parameter.

Since our code hosts particles on the finest spacetree level which can accommodate their H_i, the smoothing kernel domain never spans more than two mesh cubes in any direction on the respective mesh level. To evaluate a sum over all neighbours of a particle it is hence sufficient to loop over all particles which are contained in the same cube as the particle of interest or in any vertex-connected neighbour cube. However, we also have to extend this argument recursively over coarser and finer mesh resolutions.

Table I. Attributes per grid vertex. Some additional vertex properties required for the parallelisation and data exchange are omitted from the table. Magic range constants can be change by user (default shown).

Property	Data type	Range
refinement status	enumeration	{refined, unrefined, will be refined, will be coarsened}
is vertex local	boolean	
particle pointers	pointers (via linked list, e.g.)	
level	int array $\mathbb{N}$	$\in (0, 63)$
local	boolean	
hanging	boolean	
neighbour ranks	int array $\mathbb{N}^{2d}$	$\in (0, 65536)$
...	...	...

Our implementation uses the spacetree as meta data structure to organise the particles. Once we linearise the tree along a space-filling curve, an enumeration per vertex is sufficient to encode the whole tree structure and to derive any spatial, geometric cell information. Said enumeration signals whether adjacent cubes within the spacetree are unrefined, refined, will be refined, or will be coarsened. Few additional bits and counters for the parallelisation plus pointers from and to particles supplement the vertex data type. The mesh data has a small memory footprint and hosts primarily enums, booleans and integers (Table I).

Hypothesis 1. *We assume that the mesh data resides in close cache and is used frequently as lookup mechanism. If the integer packing through Annotation 1 introduces algorithmic latency, it will slow down the code.*

5.3. Parallelisation

Our experiments employ a very simple domain decomposition method: The mesh is split up into equidistant chunks along the Peano space-filling curve. Each chunk is deployed to one rank, i.e. each rank gets a unique sequence of cubes from the spacetree. Each chunk furthermore is cut again into subchunks along the curve, such that each thread is given a chunk of its own. The particle distribution follows this non-overlapping domain decomposition of the tree: Each particle is owned, i.e. stored and updated, by the thread which owns the cube that overlaps with the particle centre.

To allow the individual chunks to update their particles independently, we supplement each mesh with ghost cubes. Due to the definition of H, one layer of ghost cubes on each spacetree level is sufficient. Particles falling into a ghost cube are replicated on

Table II. Core (physical) data per particle data model with known ranges and accuracies.

Property	Symbol	Data type	Range
mass	m	double	const.
smoothing length	h	double	$\in (h_{\min}, h_{\max}]$
position	$\mathbf{x}$	double array $\mathbb{R}^d$	$\in (0, 1]$
velocity	$\mathbf{v}$	double array $\mathbb{R}^d$	
acceleration	$\mathbf{a}$	double array $\mathbb{R}^d$	
density	ρ	double	$\in (0, \infty]$
pressure	P	double	$\in (0, \infty]$
internal energy	u	double	$\in (0, \infty]$
time derivative of u	$\dot{u}$	double	

neighbouring domain subpartitions. This requires synchronisation of data and yields a certain memory overhead, but it allows the individual threads to process their particle data without any sychronisation, as long as we ensure that all data (replica) are made consistent after each algorithmic step. More sophisticated, task-based formalisms [Schaller et al. 2016] exploiting shared memory exist, yet are out of scope here.

5.4. Data model and data access pattern

Our SPH implementation follows few well-trodden paths. As the particles may move in each and every simulation time step and hence have to be resorted into the space-tree frequently, we hold them as an array-of-structs (AoS). The particles' core data model (Table II) stores nine physical variables per particle and updates them along the following scheme:

First, the algorithm calculates the density and smoothing length per particle. The latter determines the shape of W_{ij}, i.e. the smoothing kernel W associated with i yet depending on the distance to particle j. For the underlying iterative scheme, the algorithm reads the particles' mass, density, position and smoothing length, and it updates their ρ and h iteratively. Note however that during this particle-particle interaction loop, for any given particle only the neighbouring particles' masses and positions are required to be read, but not their h.

Second, the algorithm "prepares" each particle to evaluate its acceleration and internal energy evolution using the updated values of h and ρ, i.e. it calculates and stores most of the terms going into the sum in the right-hand-side of (2) and (3), but the actual sum (loop) over j is calculated later. In particular, this step calculates f_i, P_i, as well as individual AV terms such as (15).

Third, a second loop is performed to calculate the acceleration that is exerted on each particle by its neighbours via (2) and the AV terms. The internal energy transfer among them via (3) is also performed. This step collects the terms evaluated and stored in the previous step.

The core algorithmic steps read and write different subsets of the particle properties. Prior to each algorithmic step, data read has to be sent from particles to their halo copies on other ranks.

Hypothesis 2. *The compute-intensive steps, i.e. the non-linear density solve and force calculation, should benefit from packing, as they can hold more data in closeby caches. However, the same packing might constrain the vector efficiency.*

Finally, the code integrates the equations of motion (2) and (3) to update the particles' position, velocity and internal energy. In this step, the spatial particle topology, i.e. the association of particles to mesh cells, can change, and particles can leave their subdomain, i.e. travel between cores and ranks.

Table III. Excerpt of additional attributes per particle which are required to keep the data consistent throughout the explicit time steps and the evaluation through multiple compute kernels.

Property	Symbol	Data type	Range
Variable smoothing length terms			
'grad-h' term	f	double	
density h-gradient	$\partial_h \rho$	double	
Artificial viscosity scheme			
Balsara switch	B	double	$\in (0, 1)$
signal velocity	v_{sig}	double	$\in (0, \infty]$
velocity curl	$\nabla \times \mathbf{v}$	double array $\mathbb{R}^d$	
velocity divergence	$\nabla \cdot \mathbf{v}$	double array $\mathbb{R}^d$	
Newton-Raphson iterative solver			
old smoothing length	h_{old}	double	$\in (h_{\text{min}}, R_{\text{cutoff}}]$
iteration count	N_{iter}	int	$\in (1, N_{\text{iter}}^{\text{max}}]$
has particle converged		bool	$\in \{0, 1\}$
Time integration			
CFL time-step size	Δt	double	$\in (0, \infty]$
has particle been kicked		bool	$\in \{0, 1\}$
Move state		enum	$\in \{0, 1, 2\}$
Parallel state		enum	$\in \{0, 1, 2\}$
New Parallel state		enum	$\in \{0, 1, 2\}$

Hypothesis 3. *While the spatial particle topology (spatial arrangement) remains invariant for most compute steps, particles can eventually travel between ranks and hence require the exchange of all of their attributes via MPI. Here, we expect the code to benefit from an reduction of the memory footprint as we stress the interconnect's bandwidth. For all other algorithm steps, we expect to benefit from the fact that we can define views on data types and exchange only some particle attributes.*

Particles hold predominantly floating-point data. Some attributes have temporal access character, i.e. are only used for some algorithm steps, while other properties such as the particle positions are needed in each and every algorithm step. We also store some secondary data such as gradients within each particle, i.e. quantities that are derived from other data yet cannot be recomputed quickly on-the-fly when we need them later on. This eliminates the need to reconstruct them expensively. We end up with a significant memory footprint per particle (Table III).

As we commit to AoS as storage format and as we deal with huge numbers of particles, we may assume that we have to read them from the main memory in each and every compute step.

Hypothesis 4. *We assume that the computationally cheap compute kernels suffer from bandwidth restrictions and hence benefit from the floating-point compression.*

5.5. Floating point accuracy

It is not clear in which precision different fields have to be stored: Even if we assume that double precision is required for primary, physical quantities, properties such as the smoothing length carry a lower information density: A difference in h in the order of floating-point accuracy most often does not include more particles into the underlying truncated sum, while even additional particles do not affect the algorithm outcome negatively. If in doubt, we can always make h slightly larger.

There is, to the best of our knowledge, no formal proof which accuracy is required for attributes which carry physical meaning. Empirical evidence and comparisons to other codes from the field suggest that we cannot make compromises on the particle positions and density which feed into non-linear follow-up calculations, but can compress other quantities to single precision or beyond.

Hypothesis 5. *Our baseline code is written over doubles, while other codes employ a mixture of double and single precision. Yet, not all variables might even require single precision.*

6. RESULTS

To assess the impact and potential of our language extensions, we rely on various benchmarks which highlight different extension properties. We run all benchmarks on several test platforms.

Durham's Hamilton 8 supercomputer is a cluster hosting AMD EPYC 7702 64-Core processors, i.e. the AMD K17 (Zen2) architecture, where the 2×64 cores per node are spread over two sockets. Each core has access to 32 kB exclusive L1 cache, and 512 kB L2 cache. The L3 cache is (physically) split into chunks of 16 MB associated with four cores. Infiniband HDR 200GB/s serves as interconnect. A second machine is an AMD EPYC 9654 (Genoa) testbed. It features 2×96 cores over 2×4 NUMA domains spread over two sockets, hosts an L2 cache of 1,024 KByte per core and a shared L3 cache with 384 MByte per socket. Our third system hosts an Intel Xeon Gold 6430 (Sapphire Rapid). It features 2×32 cores over two sockets. They form two NUMA domains with an L2 cache of 2,048 KByte per core and a shared L3 cache with 62 MByte per socket.

We use Intel MPI (version 2021.4) for the distributed memory parallelisation and realise all shared memory parallelism through OpenMP. The experiments rely on the most aggressive generic compiler optimisation level and code generation for the specific target instruction set. All results are conducted with the Peano AMR framework [Weinzierl 2019] handling all the meshing, domain decomposition and data handling, while the SPH compute kernels stem from the SWIFT software [Schaller et al. 2016; Schaller et al. 2023]. The particle administration within the mesh follows the particle-in-dual-tree concept [Weinzierl et al. 2015].

6.1. Lossless compression of integer data, enums and booleans

Fig. 2.   Earth and Sun orbit around each other. Our adaptive mesh zooms into and follows the two objects.

In order to study the impact of our compression on integers, enums, and booleans, we run a two-body problem where we disable all SPH-specific numerics, i.e. density and force calculations. Instead, we fix one object at the centre of the domain and send the second body on a stable orbit by hard-coding its centripetal force. We eliminate all compute load from the setup and instead focus on the meshing data structure which holds no floating point data.

Around the two objects, we resolve the computational domain with various maximum AMR depths: The coarsest mesh is fixed before we refine recursively for a fixed number of times around each particle. Since one particle moves, the mesh moves, too.

The vertex's booleans and enums are annotated with `[[clang::pack]]`, whereas integers are annotated with `[[clang::pack_range(MIN,MAX)]]` and make use of known value ranges (Table I).

Our studies focus on single-core runs only, and we keep track of the time-to-solution and number of instructions retired to assess the overhead introduced by the bit packing. Further to that, we measure the L2 cache miss rate, i.e. the number of cache misses vs. the number of instructions retired, as well as the L2 cache miss ratio, i.e. the number of cache misses vs. the number of cache requests. All data reported in Table IV are normalised against measurements from the unmodified code. They are hardware counters obtained through Likwid [Hager et al. 2010].

Table IV. Impact of the integers compression on system characteristics for the simulation of the orbiting particle. The higher the depth, the more accurate (finer) the dynamically adaptive mesh. We present baseline (uncompressed) data vs. data normalised against the baseline measurements from the EPYC 7702.

AMR depth	Instr. retired (normalised)	L2 cache miss rate (baseline)	L2 cache miss rate (normalised)	L2 cache miss ratio (normalised)	Runtime (normalised)
0	1.07	0.0002	0.92	0.93	1.08
1	1.12	0.0002	0.91	1.00	1.12
2	1.10	0.0002	0.92	0.97	1.12
3	1.12	0.0002	0.92	0.93	1.11

We work with a cache-oblivious AMR code [Weinzierl 2019] where the mesh code of tree is linearised into one big stream, while the total memory footprint of the setup is small. Once we encounter a reasonable number of refinements around the particles, the runtime increases by around 10% due to the compression (Table IV). This correlates directly to the number of instructions retired relative to the baseline code. The packing/unpacking introduces additional instructions. All the data resides within the L2 cache most of the time. The packing improves the cache access characteristics, but this effect is marginalised and cannot compensate for the additional instructions.

Our storage format modifications come not for free: They require the compiler to introduce additional bit shifts and bit masking. While these operations are cheap, they nevertheless increase the computational load of the generated code compared to the baseline and make the code slightly slower. This confirms Hypothesis 1 experimentally.

6.2. The impact of mantissa compression on the accuracy of the solution

In order to investigate how the accuracy of SPH is affected by the mantissa truncation, we run the Noh benchmark [Noh 1987] using various precisions (valid bits) for the particles' floating-point data. Analytical solutions for the radial profiles of the density and velocity fields are known for Noh. At $t = 0.1$, we expect a strong shock front at a circle of radius $r \approx 0.032$ around the centre and a rather smooth solution otherwise. These profiles are calculated as circular averages over the solution. SPH will yield oscillations around the shock and will deliver underestimated densities inside the shock region. Both are well-documented for traditional SPH schemes such as the one implemented in our code. Yet, it is not clear how the compiler's additional truncation amplifies or damps these numerical artefacts.

A systematic study of admissible precisions per attribute is beyond scope, as it would involve multiple long-term accuracy and stability studies. We also note that there are many different combinations of accuracies, as we can set the number of valid mantissa bits per particle attribute. This yields a large configuration space. Therefore, we initially pick the same number of valid bits for each and every floating point attribute. The only exception is the particle position x, which we always store in double precision. We present profiles for 52 mantissa bits (native double precision), 23 mantissa

bits (single precision) and 10 mantissa bits. The latter is equivalent to half precision. All calculations remain coded in double precision.
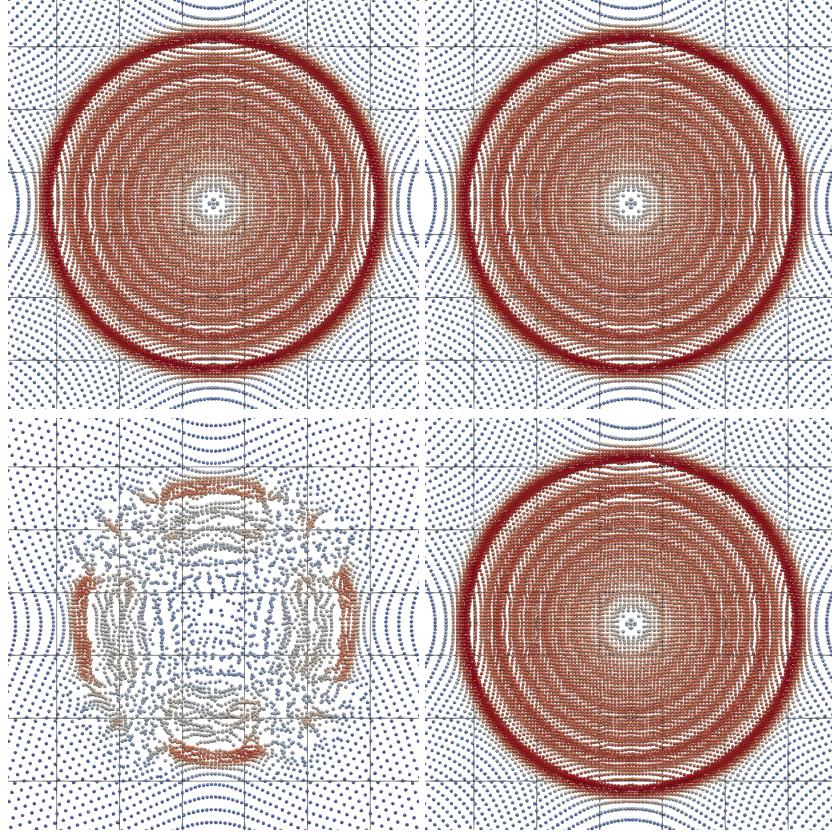

Fig. 3. Central region of the Noh problem at $t = 0.1$ for simulations with different mantissa size for the particle attributes. Lexicographically: 52 bits (double precision), 23 bits (single-precision equivalent), 10 bits (half-precision equivalent), and a mixed precision case, in which 23 bits are used for the "core" particle attributes shown in Table II, and 10 bits for all others. The colour map encodes the SPH density field values.

A visual comparison of single vs. double precision suggests that the compression has no impact at all (Figure 3). However, once we employ only 10 bits per mantissa, the solution is destroyed. The lack of precision is most noticeable on the diagonals, but we also see some loss of symmetry within the shock area. While some particles seem to outrun the shock, the vanilla SPH version overestimates the shock speed, while very strong compression yields a shock that propagates too slowly (Figure 4). In a comparison of the radial profiles of the density and radial velocity fields, the single and double precision case are indistinguishable. Besides the late shock arrival time for half precision, the solution becomes scattered which is reflected in the loss of symmetry in the plot, and the velocity outside the shock does not match the initial condition $v_r = -1$ closely anymore.

As a final test for the precision trials, we run a "mixed" case where we keep the core particle data from Table II in single precision and the rest of attributes in half precision. x remains in native double precision all the time.

Fig. 4. Radial profiles for the density field (left panel) and the radial component of the velocity field (right panel) for the central region of the setup in Figure 3.

Despite the aggressive compression of non-core particle data, the profiles continue to match full single precision or double precision calculations quite well (Figures 3 and 4). The data suggest the possibility of truncating the mantissa below the 23 bits for at least a subset of particle attributes, while we retain accurate and stable outcomes. At the same time, we recognise that a globally reduced precision is inappropriate (Hypothesis 5). Significant work on the numerical and experimental side is required to understand which attributes we can compress and by which ratio. Our annotations can streamline this development work; notably as we support user-defined compression on an attribute-by-attribute basis which can incrementally be introduced (Rationale 1).

6.3. Tailored MPI datatypes

To highlight the importance of minimalist, tailored MPI datatypes, we strip the code off any intra-node parallelisation and computation and solely focus on the data exchange between two MPI ranks. The ranks are deployed to two nodes and each sends a fixed number N of particles to their counterpart as we increase the local domains. This mimicks a ping-pong MPI test.

In a first run, we exchange whole particles, i.e. all attributes. We study the particle migration or resorting due to position updates or dynamic load balancing. In a second run, we exchange solely f, ρ and h in line with (12). These are the attributes required by the density update iterations. Third, we exchange all attributes besides the position and the f, ρ, h quantities. This would be an example of a typical attribute exchange used by the time integrator. The latter version can also be run with reduced floating-point precision. As we measure the total communication life span, all data comprise both latency and bandwidth effects. For small N, we expect latency to dominate, whereas bandwidth constraints take over for larger particle counts.

If very few particles are migrated or particles are exchanged individually—this happens for example when we sort them into cubes incrementally—the size of the particle plays close to no role (Table V). In some situations, picking a subset of attributes within the MPI implementation introduces a slight performance penalty. The more particles we transfer, the lower the cost per particle. Further to that, the size of the particle matters, i.e. exchanging only subattributes or compressed floating-point numbers reduces the runtime.

Table V. MPI ping-pong test for various particle versions, i.e. subsets or compression factors, with different byte footprint. Time $[t] = s$ per particle. N is the number of particles exchanged per MPI call, i.e. per boundary exchange. The table entries are coloured red or green if the time is bigger or smaller, respectively, compared to the column to their left, i.e. compared to the next lower level of compression.

N	288 Bytes	272 Bytes	168 Bytes	144 Bytes
1	$1.17 \cdot 10^{-3}$	$1.20 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	$1.18 \cdot 10^{-3}$
4	$3.61 \cdot 10^{-7}$	$5.31 \cdot 10^{-7}$	$2.76 \cdot 10^{-7}$	$2.08 \cdot 10^{-7}$
8	$2.89 \cdot 10^{-7}$	$1.44 \cdot 10^{-7}$	$1.83 \cdot 10^{-7}$	$2.07 \cdot 10^{-7}$
32	$3.19 \cdot 10^{-7}$	$2.55 \cdot 10^{-7}$	$8.77 \cdot 10^{-8}$	$8.71 \cdot 10^{-8}$
128	$2.04 \cdot 10^{-7}$	$3.57 \cdot 10^{-7}$	$2.86 \cdot 10^{-7}$	$2.80 \cdot 10^{-7}$
512	$1.70 \cdot 10^{-7}$	$1.21 \cdot 10^{-7}$	$8.23 \cdot 10^{-8}$	$7.40 \cdot 10^{-8}$
2,048	$1.66 \cdot 10^{-7}$	$1.17 \cdot 10^{-7}$	$7.57 \cdot 10^{-8}$	$6.84 \cdot 10^{-8}$
8,192	$1.41 \cdot 10^{-7}$	$1.14 \cdot 10^{-7}$	$6.73 \cdot 10^{-8}$	$6.15 \cdot 10^{-8}$
32,768	$1.36 \cdot 10^{-7}$	$1.24 \cdot 10^{-7}$	$6.31 \cdot 10^{-8}$	$5.40 \cdot 10^{-8}$

Picking a subset of attributes introduces some overhead. We assume that the MPI implementation internally has to gather and scatter some data. If the individual attributes become smaller due to floating-point compression, we again profit. This is likely a memory copy effect. Once we increase the particle count, the latency penalty is amortised over all particles, and bandwidth constraints kick in. Therefore, the particle footprint does matter. We approach an almost linear regime, where a halving the memory footprint almost yields a speedup of two.

6.4. Performance of the algoritmic phases

We finally assess the performance of the SPH compute phases. For this, we assess two degrees of freedom: the particle count and the number of threads. The threads are pinned to cores, and we use `numactl` with the `membind` option to ensure that all data stems from the used cores or NUMA domains respectively. That is, the cores use only cache and memory from one socket (Intel) or the number of NUMA domains employed (AMD). As each thread is pinned to one core, core and thread are used as synonyms.

Our measurements compare the uncompressed C++ version using the `double` data type everywhere with a version where we employ the integer, enum and bool packing plus annotations of floating point attributes where we know it is save to to do so. We only restrict the `double` to `float`'s mantissa precision, i.e. we stick to a regime which is known to be robust and do not study further gains resulting from below-`float` storage. Integer and floating-point compression together bring the particle's memory footprint down from 256 bytes to 152 bytes.

As we use a leapfrog time integrator, a time step (Section 2) decomposes into two kicks (acceleration update) and one drift (movement, i.e. position update), interfused by the density and force calculation. For the performance studies, we distill a benchmark (miniapp) running through this sequence. It allows us to mimick two realisation variants: In the first variant, we run through the sequence of the time step calculations one by one, always traversing all particles. This mirrors classic fork-join parallelism, i.e. one global parallel for loop per computational step. In the second variant, we work on one small chunk of particles at a time, i.e. run the calculations over this chunk several times before we continue with the next chunk. This mirrors a task-based approach [Schaller et al. 2023], where we traverse the task graph depth-first: If a set of particles tied to one vertex has drifted, we immediately kick again, update the density (with multiple iterations), compute forces, and so forth, all using minimal data exchange with other tasks handling spatially close particle sets. We try to complete as many steps on a small subset of the data as possible, i.e. we work very localised.

All setups are constructed such that the workload resembles the computational load that we obtain when we hold approximately 64 particles per cell. We balance these chunks of 64 equally among the involved threads using OpenMP's static partitioning. The benchmark clears all caches prior to the first kernel invocation assessed.

Two different memory access characteristics arise: For the sequence of steps, we stream the whole particle set into the cores per kernel invocation. The data has to run through the whole memory hierarchy once the total memory footprint of all particles is big enough, i.e. once the particles do not fit into a cache anymore. Otherwise, they reside within the L3 or L2 cache, respectively. For a task-like setup, we repeatedly work on the same small chunks of particles. They likely reside in cache.

Our miniapp breaks the runtime characteristics down per kernel. We discuss the kernels with linear internal cost separate from kernels with quadratic complexity, and use the drift as representative for the former while the force calculation represents the latter. Kicks and density iterations exhibit very similar characteristics as those discussed. For all kernels, we measure the throughput, i.e. number of particle updates per second. In the case of the density calculations, this corresponds to the cost for one non-linear iteration. For the force, it corresponds to the summation over all local neighbours which have an impact. In practice, we do not know how many iterations are required over a set of particles if we determine the density. Yet, the characteristics of many iterative updates are covered by the task-based miniapp execution pattern, i.e. if some particles trigger many updates, their memory access characteristics will start to resemble the task-based miniapp, even though we might globally work with a cascade of for loops over the individual algorithm phases.

6.4.1. Kernels with linear computational complexity on Sapphire Rapid. The more threads we use, the lower the throughput for tiny problem sizes (Figure 5). As we increase the particle count, the throughput increases. Any throughput curve for multiple threads eventually exceeds throughputs stemming from fewer threads. The break even point is roughly found around the L2 cache size. Once the problem size exceeds the L3 cache, the performance of the stream-like access pattern drops. If we access data multiple times however, falling out of L3 plays no observable role unless we put all threads to use. In this latter case, we pay a minor penalty. Overall, the throughput resembles a plateau.

The inverse scaling for very small particle numbers showcases that the OpenMP parallelisation overhead is not negligible. Consequently, this penalty is smaller relative to the runtime if we reuse the loaded data multiple times due to multiple kernel updates. As soon as we increase the number of particles sufficiently, adding more threads becomes advantageous. Each thread contributes its own L2 cache, and the L3 seems to be well-designed to serve all of the cores at the same time. If we access data repeatedly before we stream in the next chunk of work, we obtain a higher throughput. This is fundamentally a cache blocking effect. If the main memory serves a stream-like data access pattern, we suffer from its lower bandwidth relative to the L3 cache. If the main memory however is only hit occasionally, as we mainly work on in-cache data, we only pay the price for the latency, which really only introduces a penalty for very high particle counts.

The impact of the compression is best studied through the relative speedup compared to the uncompressed variant. Even if the problem overall fits into the L2 or L3 cache, we still have to stream it in from the main memory initially. We pay for the memory access latency. The compression increases this latency logically, as each data access first has to unpack the data and eventually pack it back. Therefore, compression does not pay off for small problems and stream-like data access where we already suffer from latency constraints. It only pays off once we stress the memory interconnect due
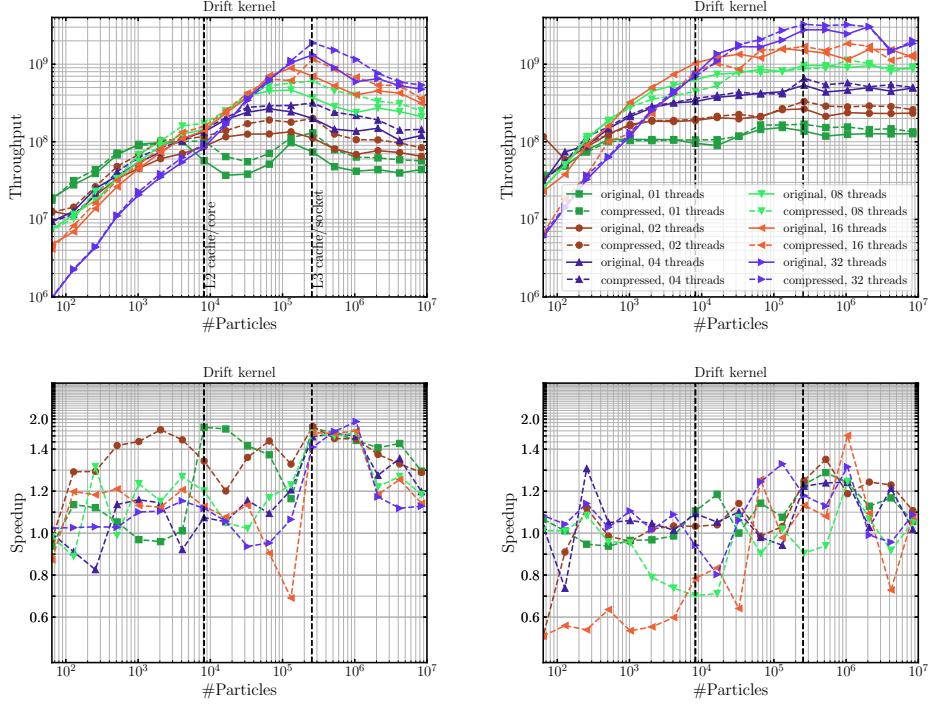
Fig. 5. Runtime behaviour for the drift step for various particle counts on one socket of the Sapphire Rapid. We compare profiles for a stream-like access (left) to the profiles resulting from a task-based realisation (right). We measure throughput, i.e. particle updates (top), but translate them into speedup of the compressed version over the uncompressed baseline (bottom).

to very large total problem sizes, and hence become bandwidth-bound. Consequently, using compression hardly ever is beneficial for the task-based access characteristics, where we are never bandwidth-bound.

We have to relativise our Hypothesis 4: Our computationally cheap kernels with linear compute characteristics are not automatically bandwidth-bound, and we therefore do not uniformly benefit from the compression. Instead, compression only pays off robustly for very large problem size, which is an insight that has to be taken into account by a software performance engineer.

6.4.2. Kernels with linear computational complexity on Genoa. The throughput on a single NUMA domain of the Genoa chip is more difficult to explain (Figure 6). For the task-like access patterns, we get qualitatively similar data to the Sapphire Rapid without any penalty once we fall out of the L3 cache. The stream access pattern is different. As long as we stick to very small problem sizes, we again observe that more threads yield smaller throughput initially, all throughputs increase as we increase the particle count, and the many threads' measurements catch up with the single threaded measurements. Yet, the throughputs all stagnate once we work within the L3 cache. They only fan out again for bigger setups when we fall out of L3.

Obviously, the memory controller is well-equipped to serve one NUMA domain. We never run into a bandwidth issue which would make the 24 thread access suffer. However, the L3 cache seems to be a bottleneck. It struggles to serve all cores concurrently. At the same time, once some of the memory accesses hit the main memory, we again

Fig. 6. Throughput on Genoa on a single NUMA domain for stream-like access (left) compared to a task-based realisation (right).

scale with the core count. The reason for this fan-out behaviour has to be buried within the chip architecture. It almost seems as if L3 cache misses allow the L3 to serve more cache hits while it waits for the main memory. Lacking in-depth insight into the reasons for this behaviour, we nevertheless can make statements on the impact of the compression:

Due to the L3 bottleneck, the compression is beneficial for all tiny problems fitting into the local L2 caches, as well as for problems that can be hosted completely in L3. The L2 cache per core can host more particles in total as we use compression. It hence reduces pressure on the L3. We see fewer L3 hits. This leads to improved throughput. In exchange, the compression does not help at all if we stream data all the time from the main memory, and it has no positive impact on the throughput for the task-like access pattern. In many cases, it introduces some overhead. Indeed, the speedup strays between 0.8 and 1.2 anarchically (not shown).

Once we run our benchmark over multiple NUMA domains (Figure 7), the throughput and the impact of the compression change character. We observe that the stream-like access runs into a plateau now as well, while the task-based access pattern yields a curve which drops once we leave the L3 cache, too. The latter localises all data accesses. Therefore, the drop has to be caused by the comparatively high latency of the main memory accesses.

As we stress the memory hierarchy on all three levels—compare the L2/L3 discussion for a single NUMA domain plus the latency observation above—compression pays off robustly for all setups, unless we are completely entering a streaming domain or hit the main memory frequently. Different to the Intel system, where compression pays off for the large particle counts only, we benefit from almost an inverted behaviour. This is reasonable given the vast L3 cache size of the system, but also the balancing between memory bandwidth of cores, the complex NUMA architecture and the total core count. In this context, it is important to note that all advantages of compression disappear if we scatter the threads over NUMA domains, i.e. use for example 24 threads distributed over two NUMA domains (not shown). In such a case, we do not stress the L3 anymore sufficiently.

For the Genoa system, Hypothesis 2 can be generalised: Also kernels with low computational load benefit from the compression, if the compression helps us to release pressure on a bottleneck further down the memory hierarchy. Again, our main argument is avoiding latency effects rather than bandwidth (Hypothesis 4) as we work on a cache architecture. The discouraging observation for developers here is that the two
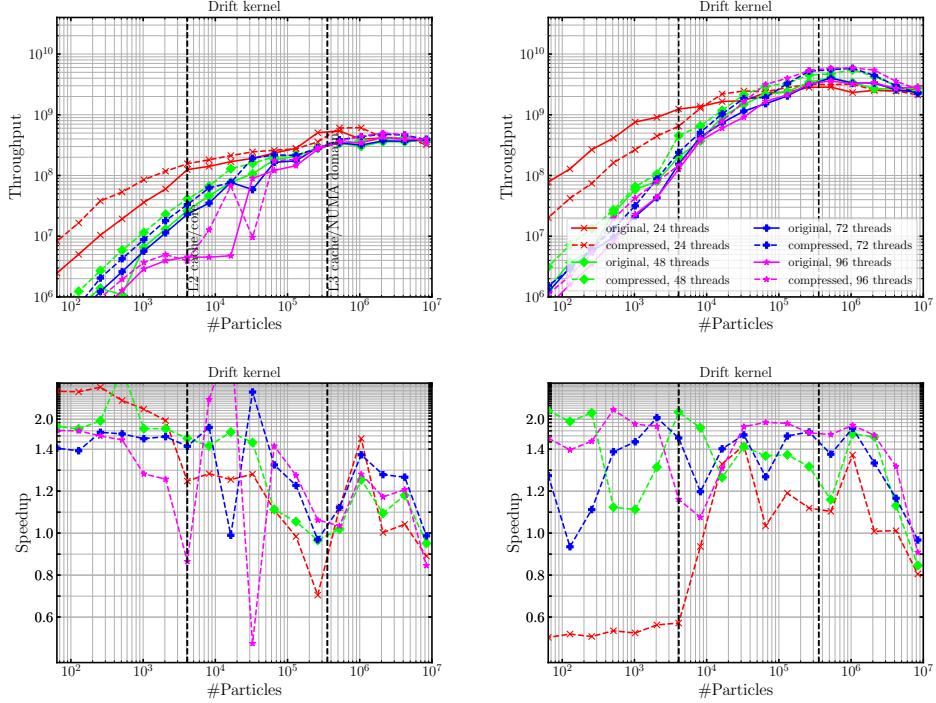
Fig. 7.   Throughput on Genoa on a multiple NUMA domains for stream-like access (left) compared to a task-based realisation (right). We present throughputs (top) as well as the speedups obtained through compression relative to the uncompressed version (bottom).

systems, though both x86-based, require completely different strategies regarding the compression. This is an important argument to "outsource" the actual compression decision to annotations and a compiler, rather than to realise it manually within source code through rewrites.

6.4.3. Kernels with quadratic computational complexity. For the force calculation, both testbed architectures yield qualitatively similar curves (Figure 8). More threads pay off, but we hit a plateau once our problem is too big to fit into the L3 cache anymore. The data for the Genoa is slightly more "noisy", which we can attribute to its more complex NUMA architecture. As we work with a computationally demanding kernel, the idea of task-based parallelism plays no significant role for the throughput: We are compute-bound.

The speedup curves for the Sapphire Rapid are rather erratic (Figure 9) and do not allow us to make robust statements if or when compression pays off or is detrimental. It seems that it is reasonably advantageous for the streaming-like kernel as long as we work within the L2/L3 caches, but if and only if we employ all threads. We may assume that this is again a cache effect.

On the Genoa system, the curve peaks are more pronounced, but they also are "less deterministic": A speedup for more than a factor of two can be obtained for some configurations, while the same thread choice can lead to a performance loss of up to 40% for a slightly different particle count.

Fig. 8. Measurements for the force calculation step for various particle counts. Throughput for stream-like access (left) vs. a task-based realisation (right) on Sapphire Rapid (top) and the Genoa testbed (bottom).

We hypothesise in Hypothesis 2 that the ability to hold more data in close caches is beneficial for compute-intense compute kernels. However, the data makes it clear that the overhead of the floating-point conversations is sometimes too high a price to pay. For some setups, we benefit from better cache utilisation, for others we pay too much algorithmic latency (conversion) overhead. It is a hit or miss.

For this particular type of kernel, a manual conversation from a packed representation prior to the kernel invocation hence seems to be a natural modification of the code, making the implementation robust without giving up on the performance advantages for the cheaper compute kernels. In this context, developers might consider to convert into SoA, as they have to copy anyway.

Reduced floating point precision is a widely applied technique in machine learning and successfully used in linear algebra setups [Ltaief et al. 2023]. In the context of a complex algorithmic code such as SPH, it however is no silver bullet. Its pros and cons have to be evaluated carefully, and our data suggests that the major impact of reduced precision results—for compute-intense applications—from the improved vector efficiency and not the sole bandwidth savings. Our compression approach targeting the memory footprint is clearly more relevant for compute kernels with low arithmetic intensity. Otherwise, it has to be used with care.

We reiterate our key statement that our technique allows for a further optimisation which is not assessed here: As we bring down the memory footprint, we can squeeze larger problems onto a node. Hence, we can weakly scale to a larger logical problem size, which is typically advantageous for the parallel efficiency and unfolds it full impact notably for compute-intense compute kernels.

Fig. 9.   Speedup data for the measurements from Figure 8.

7. CONCLUSION

Many scientific codes suffer from large memory footprints. Our annotations of the C++ language allow developers to specify and fine-tune the information density within a struct by altering the accuracy of floating-point numbers and ranges for integers as well as implicitly removing internal padding and alignment, and our implementation of these augmentations within LLVM uses the additional intelligence provided by the developers to reduce the memory footprint. Along the same lines, we offer a mechanism to develop MPI-based code more efficiently—at the moment, any change of data layout induces a tedious alteration of MPI datatypes. This extension enables developers to exchange only those attributes of a struct through MPI which actually change and makes the MPI data types benefit from our compression technologies, too.

Our experiments with an SPH code show that the extensions help to write more memory-modest code. This allows users to run bigger simulations on machines where memory is limited, i.e. to challenge classic strong scaling plateaus or performance degradations. With the trend to integrate faster yet overall limited High Bandwidth Memory into chips, this opportunity remains important even though bandwidth penalties might decrease, as we expect the average memory per core to shrink or stagnate.

The correlation of memory modesty with performance however is a nuanced, multi-faceted one: Our data suggests that cache-optimised and bandwidth-constrained codes benefit from the compression most, as we now can squeeze more data into existing caches close to the chip or transfer more logical data per cache line. Other codes will have to pay overheads and penalties for the savings in memory. Performance engi-

neering hence is not automated or made simpler with our approach, but we add an additional level of complexity.

In this context, we consider the seamless integration into ISO C++ to be pivotal for the realisation of our intention: Annotations can be ignored without breaking a code's semantics, simple assignments allow the programmer to convert packed and compressed data into native C++ datatypes which fit directly to machine instruction sets, and the realisation as additional compiler pass means that any compiler-internal optimisation further down the translation pipeline remains relevant. This way, memory optimisation can be implemented incrementally, and code remains standard-conform aka portable for different machines.

Many open questions remain: Future codes will run on strongly heterogeneous architectures, where heterogeneous means both heterogeneous memory as well as heterogeneous compute facilities such as CPU–GPU combinations or special-purpose compute entities such as large AVX "subprocessors" or their matrix extensions (tensor cores). While our code transformations reduce the memory footprint and help to write code with reduced bandwidth needs, they introduce additional computational work to convert the data representations into each other, and they do not exploit the reduced precision in any way for the actual computation. It not clear how work has to be distributed within heterogeneous systems: Should the conversions be deployed to a GPU if the computations run on the accelerator, could they be deployed to external smart compute units or networks once the data is expelled from the local caches, are the transformations en-bloc operations on all input data that precede the invocation of an offloaded compute kernel, or can they be triggered lazily on a stream while a compute unit already starts to process data, can we utilise AVX co-processors, and so forth? Further to that, it seems to appealing to use the knowledge about reduced precision to alter the underlying compute data type of variables: If the number of significant bits in the annotation is smaller than a `float`'s bits, it might seem to be convenient to use `float` as baseline type even though the variable might be modelled as `double`. Such considerations have to be subject of future work.

A second scientific challenge arises from flexible floating-point storage formats. Different to lossy compression that is applied only to data prior to post-processing ([Lindstrom 2014]), our code annotations work in-situ, i.e. on data potentially used by follow-up calculations. They are thus an excellent tool to study mixed-precision algorithms, and to introduce support for new reduced precision arithmetics in the hardware. However, our precision choices are static. In many applications, the actual information density within floating-point data changes over time [Weinzierl and Weinzierl 2018; Eckhardt et al. 2015; Murray and Weinzierl 2020], i.e. the number of significant bits has to be chosen dynamically. It is an open research question how our extensions can be generalised to support flexible precision choices.

Finally, we assume that our annotations yield a more significant speedup once they are combined with loop transformations: If floating point data are stored in reduced, user-defined precisions, our current compiler realisation wraps each data access into pack and unpack routines. This might be convenient for single access loops. It is likely a poor realisation whenever we work with loops accessing multiple particles' attributes multiple times. Here, we may assume that it is advantageous to unpack data once in a preamble to the loop and to convert it back once the loop has terminated. Such a prologue-epilogue transformation should be composable with on-the-fly AoS-to-SoA conversations [Radtke and Weinzierl 2024] and facilitate the usage of fully vectorised instruction streams including coalesced loads and stores. A price to pay is an increased temporary memory footprint. The elephant in the room is the question to which degree we can automate all of these transformations: Are there robust, reliable heuristics within a compiler that can guide the selection of a proper conversion realisation and

inform the translation when in the code to (optimistically) convert data representations?

Acknowledgements

REFERENCES

Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Alyson Fox, Mark Gates, Nicholas J Higham, Xiaoye S Li, Jennifer Loe, Piotr Luszczek, Srikara Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry F Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M Tsai, and Ulrike Meier Yang. 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. 35 (Jul 2021), 344–369. DOI:http://dx.doi.org/10.1177/10943420211003313

Ankur Agrawal, Silvia M. Mueller, Bruce M. Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan. 2019. DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, Kyoto, Japan, 92–95. DOI:http://dx.doi.org/10.1109/ARITH.2019.00023

Dinshaw S. Balsara. 1995. von Neumann stability analysis of smooth particle hydrodynamics–suggestions for optimal algorithms. *J. Comput. Phys.* 121, 2 (Jan. 1995), 357–372. DOI:http://dx.doi.org/10.1016/S0021-9991(95)90221-X

Folkmar Bornemann, Dirk Laurie, Stan Wagon, and Jörg Waldvogel. 2004. *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*. Society for Industrial and Applied Mathematics. DOI:http://dx.doi.org/10.1137/1.9780898717969

Hans-Joachim Bungartz, Wolfgang Eckhardt, Miriam Mehl, and Tobias Weinzierl. 2008. DaStGen—A Data Structure Generator for Parallel C++ HPC Software. In *Computational Science – ICCS 2008*, Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–222. DOI:http://dx.doi.org/10.1007/978-3-540-69389-5_25

Hans-Joachim Bungartz, Wolfgang Eckhardt, Tobias Weinzierl, and Christoph Zenger. 2010. A precompiler to reduce the memory footprint of multiscale PDE solvers in C++. *Future Generation Computer Systems* 26, 1 (2010), 175–182. DOI:http://dx.doi.org/10.1016/j.future.2009.05.011

Erin Carson and Nicholas J. Higham. 2018. Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions. *SIAM Journal on Scientific Computing* 40, 2 (Jan 2018), A817–A847. DOI:http://dx.doi.org/10.1137/17M1140819

Erin Carson and Noaman Khan. 2023. Mixed Precision Iterative Refinement with Sparse Approximate Inverse Preconditioning. *SIAM Journal on Scientific Computing* 45, 3 (2023), C131–C153.

Michael P. Connolly, Nicholas J. Higham, and Theo Mary. 2021. Stochastic Rounding and Its Probabilistic Backward Error Analysis. *SIAM Journal on Scientific Computing* 43, 1 (Jan. 2021), A566–A585. DOI:http://dx.doi.org/10.1137/20m1334796

Matteo Croci, Massimiliano Fasi, Nicholas J. Higham, Theo Mary, and Mantas Mikaitis. 2022. Stochastic Rounding: Implementation, Error Analysis and Applications. *Royal Society Open Science* 9, 3 (March 2022). DOI:http://dx.doi.org/10.1098/rsos.211631

Walter Dehnen and Hossam Aly. 2012. Improving Convergence in Smoothed Particle Hydrodynamics Simulations without Pairing Instability. *Monthly Notices of the Royal Astronomical Society* 425, 2 (Sept. 2012), 1068–1082. DOI:http://dx.doi.org/10.1111/j.1365-2966.2012.21439.x

James Diffenderfer, Alyson L. Fox, Jeffrey A. Hittinger, Geoffrey Sanders, and Peter G. Lindstrom. 2019. Error Analysis of ZFP Compression for Floating-Point Data. *SIAM Journal on Scientific Computing* 41, 3 (Jan 2019), A1867–A1898. DOI:http://dx.doi.org/10.1137/18M1168832

J. M. Domínguez, A. J. C. Crespo, M. Gómez-Gesteira, and J. C. Marongiu. 2011. Neighbour Lists in Smoothed Particle Hydrodynamics. *International Journal for Numerical Methods in Fluids* 67, 12 (2011), 2026–2042. DOI:http://dx.doi.org/10.1002/fld.2481

Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad van der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. 2011. The International Exascale Software Project roadmap. *The International Journal of High Performance Computing Applications* 25, 1 (Feb 2011), 3–60. DOI:http://dx.doi.org/10.1177/1094342010391989

Wolfgang Eckhardt, Robert Glas, Denys Korzh, Stefan Wallner, and Tobias Weinzierl. 2015. On-the-fly memory compression for multibody algorithms *(Advances in Parallel Computing—Volume 27: Parallel Computing: On the Road to Exascale)*, Vol. 27. 421–430.

Massimiliano Fasi and Mantas Mikaitis. 2021. Algorithms for Stochastically Rounded Elementary Arithmetic Operations in IEEE 754 Floating-Point Arithmetic. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (July 2021), 1451–1466. DOI:http://dx.doi.org/10.1109/tetc.2021.3069165

Rosa Filgueira, Malcolm Atkinson, Albert Nuñez, and Javier Fernández. 2012. An Adaptive, Scalable, and Portable Technique for Speeding Up MPI-Based Applications. In *Euro-Par 2012 Parallel Processing*, Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 729–740.

Pierre Fortin and Maxime Touche. 2019. Dual tree traversal on integrated GPUs for astrophysical N-body simulations. *The International Journal of High Performance Computing Applications* 33, 5 (Sep 2019), 960–972. DOI:http://dx.doi.org/10.1177/1094342019840806

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1994. *Design Patterns. Elements of Reusable Object-Oriented Software*. Prentice Hall.

Robert A. Gingold and Joseph J. Monaghan. 1977. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society* 181, 3 (1977), 375–389.

William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press.

Georg Hager, Gerhard Wellein, and Jan Treibig. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2012 41st International Conference on Parallel Processing Workshops*. IEEE Computer Society, Los Alamitos, CA, USA, 207–216. DOI:http://dx.doi.org/10.1109/ICPPW.2010.38

Nicholas J. Higham and Theo Mary. 2022. Mixed precision algorithms in numerical linear algebra. *Acta Numerica* 31 (2022), 347–414. DOI:http://dx.doi.org/10.1017/S0962492922000022

Natsuki Hosono and Mikito Furuichi. 2024. Efficient Implementation of Low-Order-Precision Smoothed Particle Hydrodynamics. *The International Journal of High Performance Computing Applications* 38, 3 (May 2024), 137–153. DOI:http://dx.doi.org/10.1177/10943420231201144

Randall Hyde. 2006. *Write great code. Volume 2, Writing high-level*. No Starch Press, San Francisco, USA.

Tsuyoshi Ichimura, Kohei Fujita, Takuma Yamaguchi, Akira Naruse, Jack C. Wells, Thomas C. Schulthess, Tjerk P. Straatsma, Christopher J. Zimmer, Maxime Martinasso, Kengo Nakajima, Muneo Hori, and Lalith Maddegedara. 2018. A Fast Scalable Implicit Solver for Nonlinear Time-Evolution Earthquake City Problem on Low-Ordered Unstructured Finite Elements with Artificial Intelligence and Transprecision Computing. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Dallas, TX, USA, 627–637. DOI:http://dx.doi.org/10.1109/SC.2018.00052

Jian Ke, M. Burtscher, and E. Speight. 2004. Runtime Compression of MPI Messanes to Improve the Performance and Scalability of Parallel Applications. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. 59–59.

Michael Kruse. 2021. Loop Transformations using Clang's Abstract Syntax Tree. In *50th International Conference on Parallel Processing Workshop*. ACM, Lemont IL, USA, 1–7. DOI:http://dx.doi.org/10.1145/3458744.3473359

Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. 2006. Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems). In *ACM/IEEE SC 2006 Conference (SC'06)*. IEEE, Tampa, FL, 50–68. DOI:http://dx.doi.org/10.1109/SC.2006.30

Steven J. Lind, Benedict D. Rogers, and Peter K. Stansby. 2020. Review of smoothed particle hydrodynamics: towards converged Lagrangian flow modelling. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 476, 2241 (2020), 20190801. DOI:http://dx.doi.org/10.1098/rspa.2019.0801

Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (Dec 2014), 2674–2683. DOI:http://dx.doi.org/10.1109/TVCG.2014.2346458

Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. 2018. Universal coding of the reals: alternatives to IEEE floating point. In *Proceedings of the Conference for Next Generation Arithmetic*. ACM, Singapore, 1–14. DOI:http://dx.doi.org/10.1145/3190339.3190344

Hatem Ltaief, Yuxi Hong, Leighton Wilson, Mathias Jacquelin, Matteo Ravasi, and David Elliot Keyes. 2023. Scaling the "Memory Wall" for Multi-Dimensional Seismic Processing with Algebraic Compression on Cerebras CS-2 Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, Dorian Arnold, Rosa M. Badia, and Kathryn M. Mohror (Eds.). ACM, 6:1–6:12. DOI:http://dx.doi.org/10.1145/3581784.3627042

Joseph J. Monaghan. 1992. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics* 30, 1 (1992), 543–574. DOI:http://dx.doi.org/10.1146/annurev.aa.30.090192.002551

Joseph J. Monaghan and John C. Lattanzio. 1985. A refined particle method for astrophysical problems. A&A 149, 1 (Aug. 1985), 135–143.

Charles D. Murray and Tobias Weinzierl. 2020. Lazy stencil integration in multigrid algorithms *(Parallel Processing and Applied Mathematics—13th International Conference on Parallel Processing and Applied Mathematics)*. 25–37.

W. F. Noh. 1987. Errors for calculations of strong shocks using an artificial viscosity and an artificial heat flux. *J. Comput. Phys.* 72, 1 (1987), 78–120. DOI:http://dx.doi.org/https://doi.org/10.1016/0021-9991(87)90074-X

Guillaume Oger, David Le Touzé, David Guibert, Matthieu de Leffe, John Biddiscombe, Jerome Soumagne, and Jg Piccinali. 2016. On distributed memory MPI-based parallelization of SPH codes in massive HPC context. *Computer Physics Communications* 200 (March 2016), 1–14. DOI:http://dx.doi.org/10.1016/j.cpc.2015.08.021

Daniel J. Price. 2012. Smoothed Particle Hydrodynamics and Magnetohydrodynamics. *J. Comput. Phys.* 231, 3 (feb 2012), 759–794. DOI:http://dx.doi.org/10.1016/j.jcp.2010.12.011

Pawel K. Radtke and Tobias Weinzierl. 2024. Compiler support for semi-manual AoS-to-SoA conversions with data views. (2024).

James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2021. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Apress open, New York.

Matthieu Schaller, Josh Borrow, Peter W. Draper, Mladen Ivkovic, Stuart McAlpine, Bert Vandenbroucke, Yannick Bahé, Evgenii Chaikin, Aidan B. G. Chalk, Tsang Keung Chan, Camila Correa, Marcel van Daalen, Willem Elbers, Pedro Gonnet, Loïc Hausammann, John Helly, Filip Huško, Jacob A. Kegerreis, Folkert S. J. Nobels, Sylvia Ploeckinger, Yves Revaz, William J. Roper, Sergio Ruiz-Bonilla, Thomas D. Sandnes, Yolan Uyttenhove, James S. Willis, and Zhen Xiang. 2023. Swift: A modern highly-parallel gravity and smoothed particle hydrodynamics solver for astrophysical and cosmological applications. (2023).

Matthieu Schaller, Pedro Gonnet, Aidan B. G. Chalk, and Peter W. Draper. 2016. SWIFT: Using Task-Based Parallelism, Fully Asynchronous Communication, and Graph Partition-Based Domain Decomposition for Strong Scaling on More than 100,000 Cores. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. DOI:http://dx.doi.org/10.1145/2929908.2929916

Joop Schaye, Roi Kugel, Matthieu Schaller, John C. Helly, Joey Braspenning, Willem Elbers, Ian G. Mc-Carthy, Marcel P. van Daalen, Bert Vandenbroucke, Carlos S. Frenk, Juliana Kwan, Jaime Salcido, Yannick M. Bahé, Josh Borrow, Evgenii Chaikin, Oliver Hahn, Filip Huško, Adrian Jenkins, Cedric G. Lacey, and Folkert S. J. Nobels. 2023. The FLAMINGO Project: Cosmological Hydrodynamical Simulations for Large-Scale Structure and Galaxy Cluster Surveys. *Monthly Notices of the Royal Astronomical Society* 526, 4 (Oct. 2023), 4978–5020. DOI:http://dx.doi.org/10.1093/mnras/stad2419

Volker Springel. 2005. The cosmological simulation code GADGET-2. MNRAS 364, 4 (Dec. 2005), 1105–1134. DOI:http://dx.doi.org/10.1111/j.1365-2966.2005.09655.x

Matthias Steinmetz and E. Mueller. 1993. On the capabilities and limits of smoothed particle hydrodynamics. A&A 268, 1 (Feb. 1993), 391–410.

Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2018. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Dresden, Germany, 1051–1056. DOI:http://dx.doi.org/10.23919/DATE.2018.8342167

Po-An Tsai and Daniel Sanchez. 2019. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI, USA, 229–242. DOI:http://dx.doi.org/10.1145/3297858.3304006

Marion Weinzierl and Tobias Weinzierl. 2018. Algebraic-geometric matrix-free multigrid on dynamically adaptive Cartesian meshes. *ACM Transactions on Mathematical Software (TOMS)* 44 (2018), 32:1–32:44. Issue 3.

Tobias Weinzierl. 2019. The Peano Software - Parallel, Automaton-based, Dynamically Adaptive Grid Traversals. *ACM Trans. Math. Software* 45, 2 (2019), 14:1–14:41. DOI:http://dx.doi.org/10.1145/3319797

Tobias Weinzierl, Bart Verleye, Pierre Henri, and Dirk Roose. 2015. Two Particle-in-Grid Realisations on Spacetrees. *Parallel Comput.* 52 (2015), 42–64. http://arxiv.org/abs/1508.02435

A. INSTALLATION OF MODIFIED LLVM VARIANT

The release of the compiler extension as LLVM patch is currently under preparation. A preview as used for the experiments here is available from https://github.com/pradt2/llvm-project.git.

We have built this compiler version with both Ubuntu 22.04 LTS and Fedora 37, relying only on relatively mature and old third-party software (Table 10). If the use of any of the new attributes leads to a compilation error, a common starting point for troubleshooting is to inspect the rewritten source code. To see the rewritten code, add `-fpostprocessing-output-dump` to the compilation flags. The flag causes the post-processed source code be written to the standard output.

Software package	Recommended version	Version on Ubuntu	Version on Fedora
Git	2+	2.34.1	2.39.1
GNU GCC (incl. C++ support)	10+	11.3.0	12.2.1
CMake	3+	3.22.1	3.25.2
GNU Make	4+	4.3	4.3
Python 3	3.6+	3.10.6	3.11.0
Perl (incl. core modules)	5+	5.34	5.36

Fig. 10. Minimum third-party software dependencies needed to compile Clang/LLVM.

B. REPRODUCIBILITY OF EXPERIMENTAL DATA

All experimental data has been produced with the Swift 2 code. It is shipped as part of the Peano 4 framework [Weinzierl 2019] available from https://gitlab.lrz.de/hpcsoftware/Peano.git. The repository provides a CMake and autotools build system. The autotools environment is set up via

```
libtoolize; aclocal; autoconf; autoheader
cp src/config.h.in .; automake --add-missing
./configure --with-multithreading=omp --enable-particles --enable-swift \
   --enable-loadbalancing --with-mpi=mpiicpc \
   CXXFLAGS="-O3■--std=c++20■-fopenmp■-g"
make
```

The `make` yields all libraries we need for our experiments. Besides the core libraries, we recommend to create all online documentation through Doxygen (`doxygen documentation/Doxyfile`) which also is available from https://hpcsoftware.pages.gitlab.lrz.de/Peano.

Each experiment is located within a repository subdirectory and contains a readme file which automatically is extracted into HTML documentation through Doxygen. Every single benchmark is produced through a Python script. It automatically picks up the compiler settings passed into autotools or CMake, respectively, and accepts arguments to configure the actual benchmark. The benchmark documentation plus the present experimental descriptions provide information on arguments used. Eventually, the Python scripts produce a stand-alone executable.

All benchmarks produce human-readable text files as outputs. Our repository ships Matplotlib scripts to convert them into figures, and the benchmarks' descriptions provide further information on these postprocessing scripts. The plots in the paper differ from Peano's vanilla benchmarking plots only by different layout choices and augmented annotations such as cache sizes.

— The *grid experiments* (Section 6.1) are produces through the benchmarks within `benchmarks/swift2/planet-orbit`.
— The *mantissa truncation impact* (Section 6.2) is studied through the Noh 2d benchmark as available through `benchmarks/swift2/hydro/noh-implosion-text`.
— The *MPI test case* (Section 6.3) is a simple ping-pong test using the data structures from the Noh 2d benchmark. Its integration into the test suite is work in progress.
— The *scalability* data (Section 6.4) are obtained through the benchmarks in `benchmarks/swift2/hydro/kernel-throughput`.

C. COMPREHENSIVE SCALABILITY DATA

Additional benchmark data for the Sapphire Rapid testbed are available from the Figures 11 and 12.

For the Genoa testbed, we collect data for one NUMA domain (Figures 13, 14), two (Figures 15, 16), three (Figures 17, 18), and four domains (Figures 19, 20). It is important to note that there are multiple benchmark curves for some thread choices: 24 threads for example can be distributed over one, two, three or four NUMA domains. A spread affinity policy yields significantly worse throughput overall.

Each test is run at least 16 times and data are averaged over these tests.

D. SMOOTHED PARTICHLE HYDRODYNAMICS: THE GOVERNING EQUATIONS

D.1. General Remarks

Smoothed Particle Hydrodynamics (SPH) is a class of meshless methods wherein the fluid is discretized using particles. Those particles are typically given some constant mass and are evolved in time using the Lagrangian equations of fluid dynamics. SPH is based on estimating the local fluid density (and other quantities) as a weighted sum over neighboring particles, where the weights are smoothly decreasing functions (kernels) such that the noise in the density estimate introduced by distant neighboring particles is reduced. More precisely, let $A(\mathbf{x})$ be a scalar field of interest of a partial
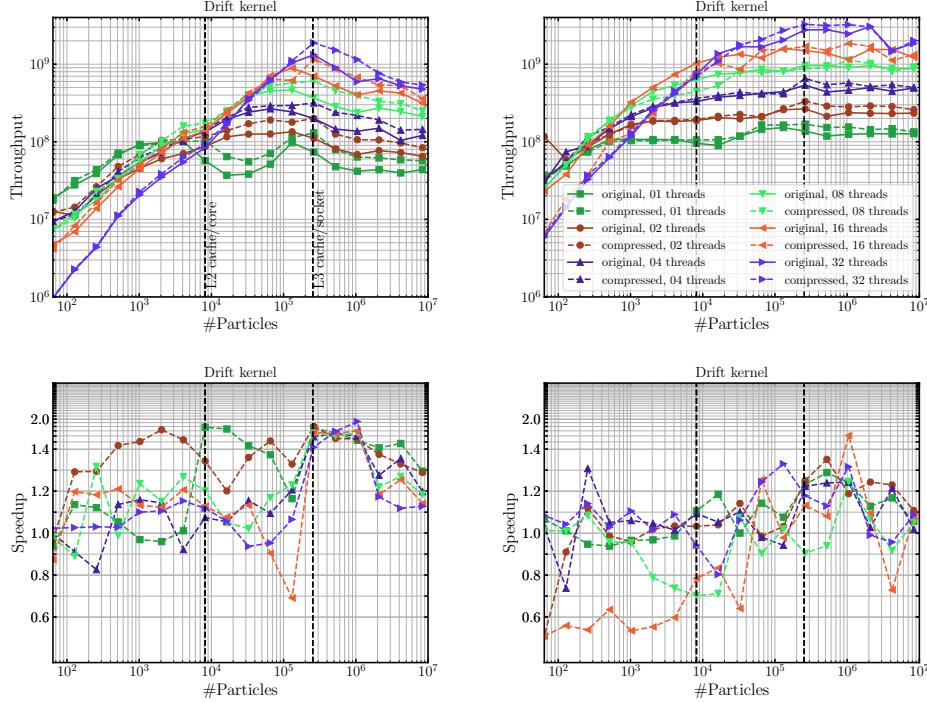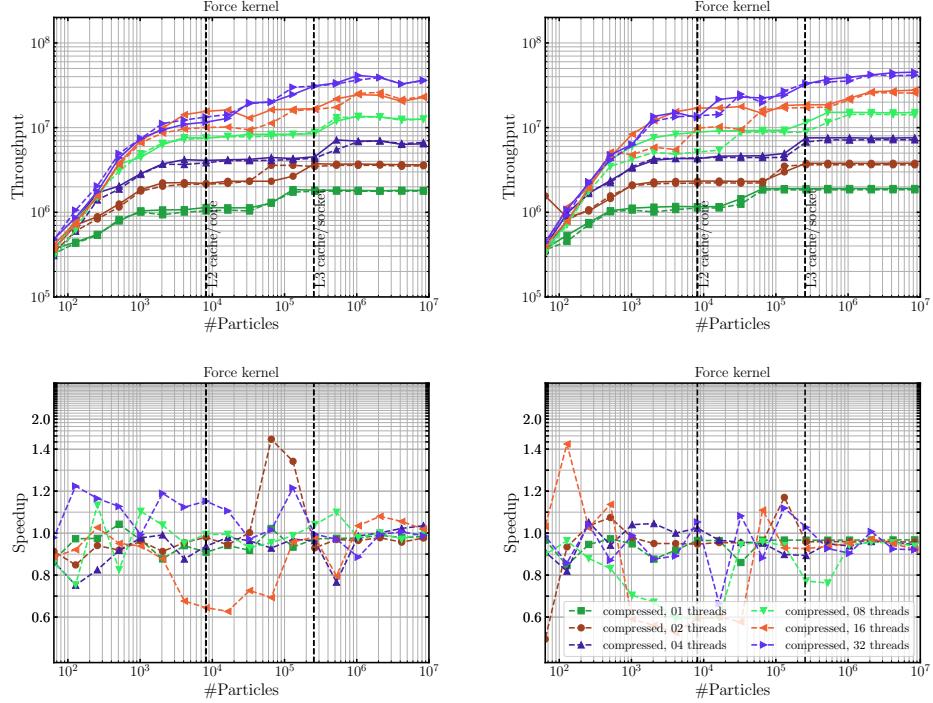
Fig. 11. Measurements for the drift kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Sapphire Rapid testbed for stream-like access (left) and task-based access characteristics (right).

differential equation. In an SPH description, we write down this quantity as the convolution

$$A(\mathbf{x}) = \int A(\mathbf{x}')\delta(\mathbf{x} - \mathbf{x}')\mathrm{d}^3\mathbf{x}' \approx \int A(\mathbf{x}')W(|\mathbf{x} - \mathbf{x}'|, H)\mathrm{d}^3\mathbf{x}' \tag{5}$$

$$\approx \sum_i \frac{m_i}{\rho_i} A(\mathbf{x}_i)W(|\mathbf{x} - \mathbf{x}_i|, H). \tag{6}$$

In (5), the Dirac distribution $\delta(\mathbf{x})$ is approximated by a smoothing kernel $W(\mathbf{x}, H)$ which is a smooth differentiable function with compact support H. While in principle (6) requires a sum over all particles i, the kernel's compact support reduces that problem to a sum over the local neighbourhood for which $W(|\mathbf{x}|, H) > 0$.

In this work, we use the quartic spline (M5) kernel [Monaghan and Lattanzio 1985]. Following the notation convention of [Dehnen and Aly 2012], the kernel in ν dimensions is given by

$$W(\mathbf{x}, H) = H^{-\nu}\, w(|\mathbf{x}|/H)$$

Fig. 12. Measurements for the force kernel kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Sapphire Rapids testbed for stream-like access (left) and task-based access characteristics (right).

with

$$w(r) = C_{\text{norm}} \times \begin{cases} (1-r)^4 - 5\left(\frac{3}{5}-r\right)^4 + 10\left(\frac{1}{5}-r\right)^4 & \text{if } 0 \leq r \leq \frac{1}{5} \\ (1-r)^4 - 5\left(\frac{3}{5}-r\right)^4 & \text{if } \frac{1}{5} \leq r \leq \frac{3}{5} \\ (1-r)^4 & \text{if } \frac{3}{5} < r < 1 \end{cases} \quad (7)$$

where $C_{\text{norm}} = \frac{5^5}{768}, \frac{5^6 3}{2398\pi} \frac{5^6}{512\pi}$ is a normalisation constant for $\nu = 1, 2, 3$ dimensions, respectively. Table 1 in [Dehnen and Aly 2012] lists other popular SPH kernel choices.

D.2. Determining the Density and Smoothing Lengths

The smoothing length, h, plays a central role in SPH. Following the definition of [Dehnen and Aly 2012], it is twice the standard deviation of the kernel and defines the spatial resolution of the numerical method: Its size determines the wavelength of acoustic waves that can be resolved.

Naturally, the compact support radius of a kernel and its smoothing length are related. The relation is typically given as $H = \Gamma_k h$. For the quartic spline kernel that we use, $\Gamma_k = 1.936492, 1.977173, 2.018932$ for $\nu = 1, 2, 3$, respectively.

Since the smoothing length determines both the spatial resolution and the number of neighbouring particles incorporated into the weighted sums, different applications demand different requirements. Hence the smoothing length can be specified via a free
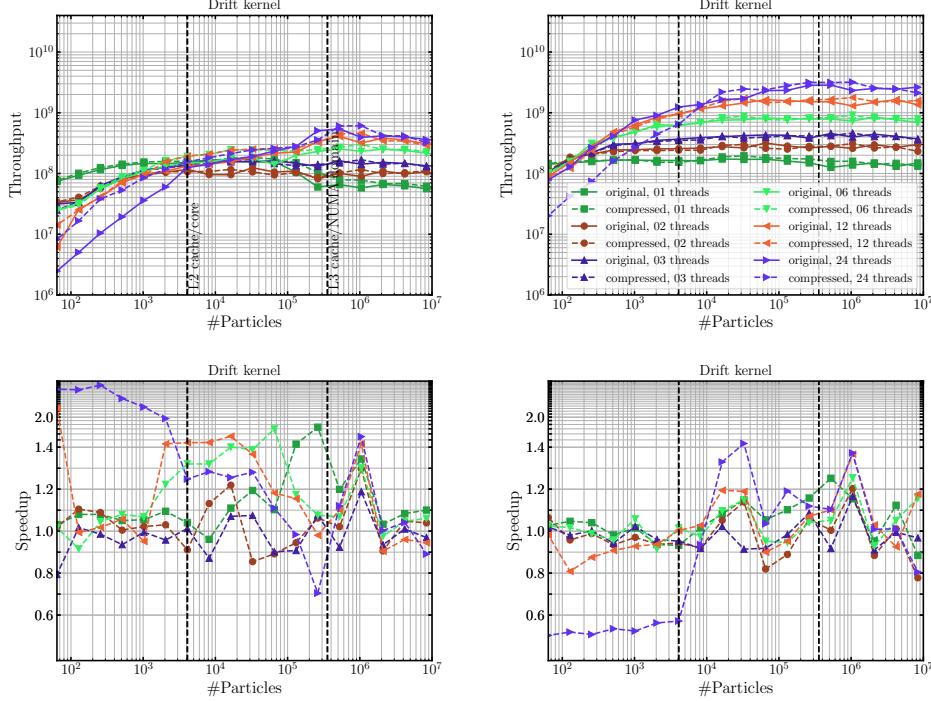
Fig. 13. Measurements for the drift kernel. Throughput (top) and speedup relative to uncompressed base-line version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use one NUMA domain.

parameter η:

$$h_i = \eta \left(\frac{m_i}{\rho_i} \right)^{1/\nu} \tag{8}$$

defining h in units of mean inter-particle distance. η is typically fixed in the range 1.2–1.5 [Steinmetz and Mueller 1993]. In our work, we use $\eta = 1.2348$.
However, since the particle density ρ_i is determined through the smoothing

$$\rho_i = \sum_j m_j W_{ij}(h_i) \tag{9}$$

with $W_{ij}(h_i) = W(\mathbf{x}_i - \mathbf{x}_j, H(h_i))$, we're left with a circular relation: ρ_i is required to determine h_i, while h_i is needed to estimate ρ_i. As a consequence, the algorithm runs a cascade of Picard Newton-Raphson iterations per particle to determine h_i and ρ_i. Since adaptive and variable smoothing lengths (in both space and time) are crucial for cosmological applications (given that the fluid can be compressed over a range spanning several orders of magnitude) this iteration needs to be performed each time step.

D.3. Equations of Motion
With the particles' smoothing lengths and densities determined, we can now turn to the equations of motion. For the present SPH demonstrator, we consider an inviscid fluid in the absence of gravity and external forces or energy sources. Hence, the indi-
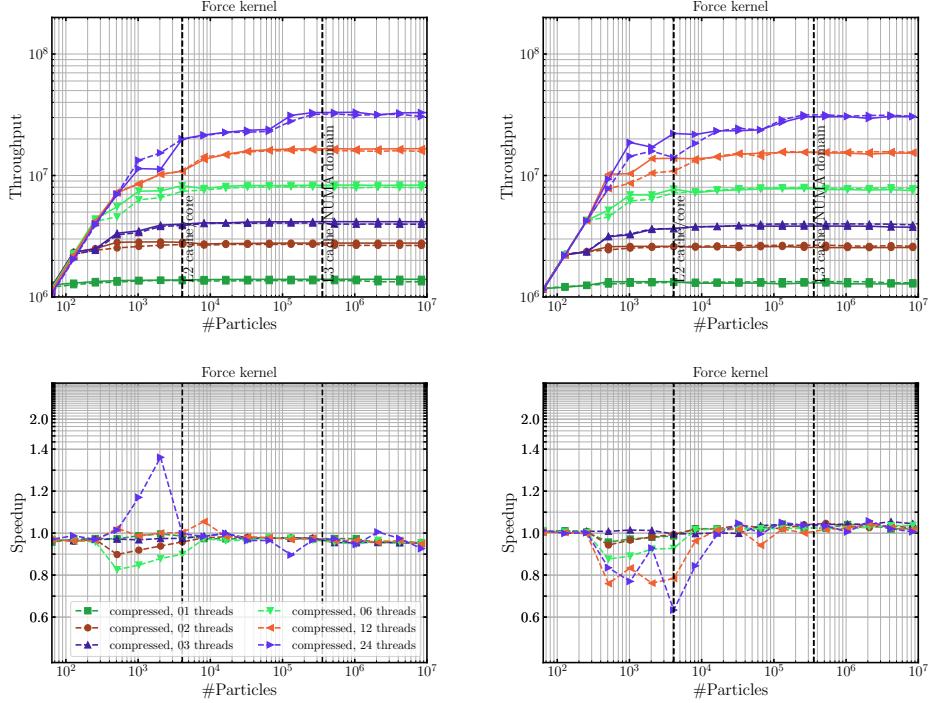
Fig. 14.   Measurements for the force kernel kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use one NUMA domain.

vidual particles tracking the fluid evolve according to the Euler equation,

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_j m_j \left[f_i \frac{P_i}{\rho_i^2} \nabla W_{ij}(h_i) + f_j \frac{P_j}{\rho_j^2} \nabla W_{ij}(h_j) \right] + \mathbf{a}_i^{\mathrm{AV}} \,, \tag{10}$$

while the thermodynamic internal energy per unit mass of the fluid, u_i, evolves according to

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = f_i \frac{P_i}{\rho_i^2} \sum_j m_j (\mathbf{v}_i - \mathbf{v}_j) \cdot \nabla W_{ij}(h_i) + \dot{u}_i^{\mathrm{AV}} \,. \tag{11}$$

$\mathbf{v}$ is the velocity field, P is the pressure and $\nabla \equiv \partial/\partial\mathbf{x}$ is the spatial gradient. The system is closed by specifying the equation of state of the fluid, $P = (\gamma - 1)u\rho$, in which γ is the adiabatic index.

The used equations include physical quantities of the fluid plus terms that are intrinsic to the SPH method. The scalar field

$$f_i = \left(1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right)^{-1} \quad \text{with} \quad \frac{\partial \rho_i}{\partial h_i} = \sum_j m_j \frac{\partial W_{ij}(h_i)}{\partial h_i} \tag{12}$$

represents the spatial fluctuations in the smoothing length $h(\mathbf{x})$ (typically known as 'grad-h' terms). They have to be taken into account whenever h is allowed to vary
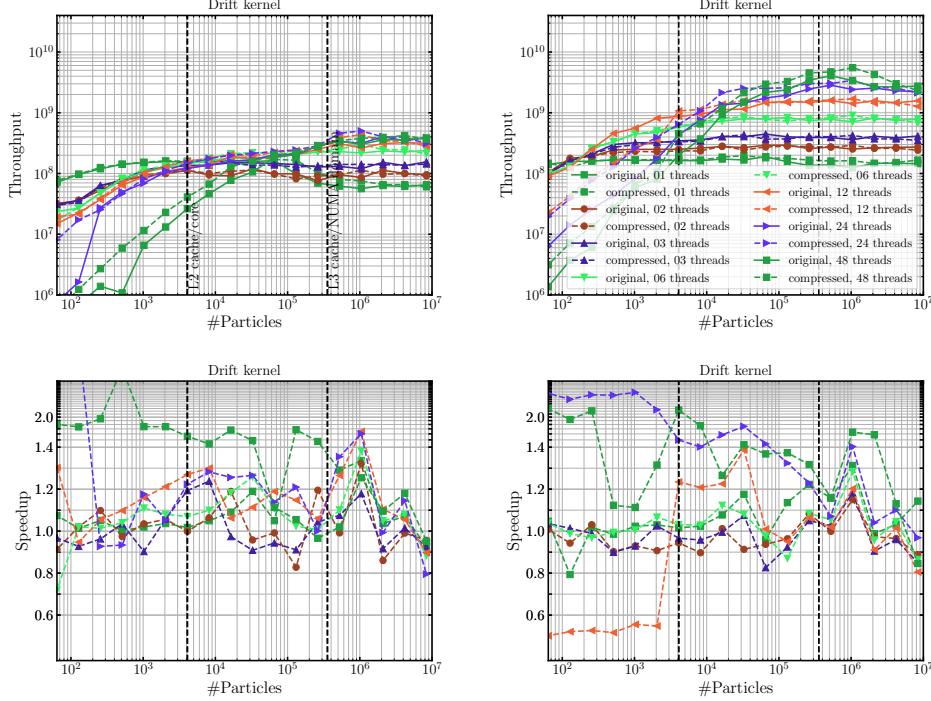
Fig. 15.   Measurements for the drift kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use two NUMA domains.

over space or time. Note that the sum in (12) can be collected simultaneously with the density field calculation (9).

Following [Monaghan 1992; Balsara 1995], and [Price 2012], we add an artificial viscosity (AV) to the (physically inviscid) fluid in order to resolve potential discontinuities (e.g. due to shocks) that could develop in the fluid. In particular, we adopt the AV model used by the GADGET-2 code [Springel 2005]. Its contribution to the acceleration in (10) is given by

$$\mathbf{a}_i^{\mathrm{AV}} = -\sum_j m_j \Pi_{ij} \nabla \overline{W}_{ij} \,. \tag{13}$$

We pick $\overline{W}_{ij} \equiv [W_{ij}(h_i) + W_{ij}(h_j)] / 2$, whereas Π_{ij} is the artificial viscosity tensor

$$\Pi_{ij} = -\frac{\alpha^{\mathrm{AV}}}{2} \frac{v_{ij}^{\mathrm{sig}} \mu_{ij}}{(\rho_i + \rho_j)/2} \frac{(B_i + B_j)}{2} \,. \tag{14}$$

In (14), $\alpha^{\mathrm{AV}} = 1$ is a (free) artificial viscosity parameter, $v_{ij}^{\mathrm{sig}} = c_{s,i} + c_{s,j} - \beta^{\mathrm{AV}} \mu_{ij}$ is the signal velocity with $c_{s,i} = \sqrt{\gamma P_i / \rho_i}$ the sound speed of the fluid at position $\mathbf{x}_i$, and $\beta^{\mathrm{AV}} = 3$ is the second viscosity parameter in this model. μ_{ij} is given by

$$\mu_{ij} = \begin{cases} \mathbf{v}_{ij} \cdot \hat{\mathbf{x}}_{ij} & \text{if } \mathbf{v}_{ij} \cdot \hat{\mathbf{x}}_{ij} < 0 \\ 0 & \text{otherwise} \end{cases}$$
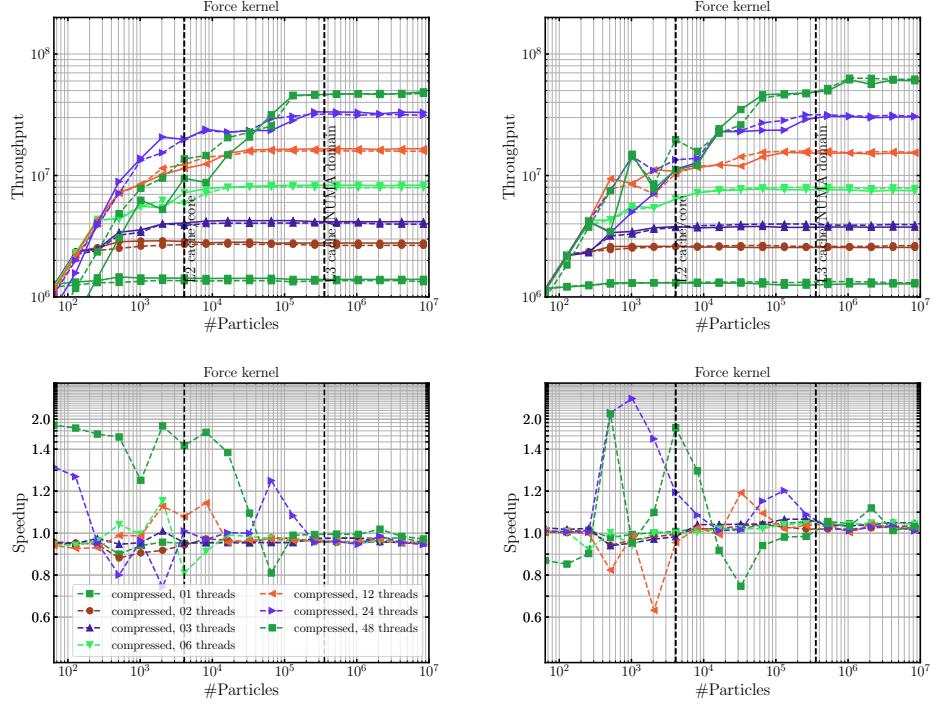
Fig. 16. Measurements for the force kernel kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use two NUMA domains.

where $\hat{\mathbf{x}}_{ij}$ is the unit position vector separating particles i and j and $\mathbf{v}_{ij} = \mathbf{v}_j - \mathbf{v}_i$. The term switches the viscous tensor (14) on whenever two particles approach each other. Lastly, the Balsara switch B_i [Balsara 1995] is modelled as

$$B_i = \frac{|\nabla \cdot \mathbf{v}_i|}{|\nabla \cdot \mathbf{v}_i| + |\nabla \times \mathbf{v}_i| + 10^{-4} c_{s,i}/h_i} . \tag{15}$$

The divergence and curl of the velocity field are computed using the standard SPH expressions:

$$\nabla \cdot \mathbf{v}_i = \frac{1}{\rho_i} \sum_j m_j \mathbf{v}_{ij} \cdot \nabla W(\mathbf{x}_{ij}, h_i),$$

$$\nabla \times \mathbf{v}_i = \frac{1}{\rho_i} \sum_j m_j \mathbf{v}_{ij} \times \nabla W(\mathbf{x}_{ij}, h_i).$$

Likewise, the AV diffusion term for the evolution equation of the internal energy (11) is

$$u_i^{\mathrm{AV}} = -\frac{1}{2} \sum_j m_j \Pi_{ij} \mathbf{v}_{ij} \cdot \nabla \overline{W}_{ij} . \tag{16}$$
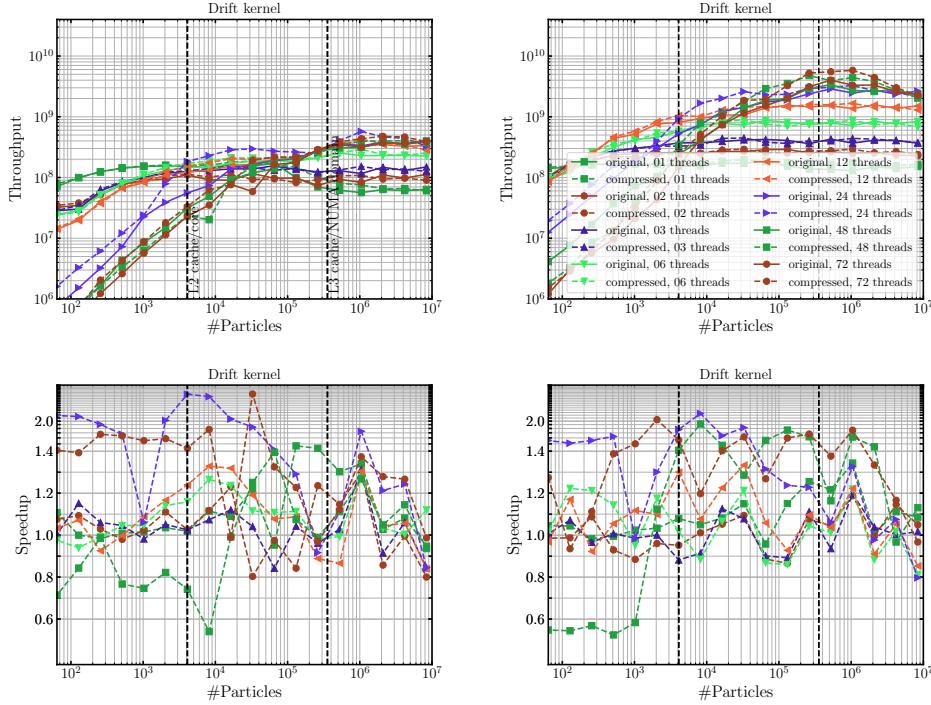
Fig. 17. Measurements for the drift kernel. Throughput (top) and speedup relative to uncompressed base-line version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use three NUMA domains.

D.4. Time Integration

Finally, the evolution equations (10) and (11) are supplemented with a well-suited time integrator. We use a kick-drift-kick leapfrog in a velocity-Verlet form:

$$\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^n + \mathbf{a}_i^n \tfrac{\Delta t}{2} \qquad \text{kick}$$

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2} \Delta t \qquad \text{drift}$$

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \mathbf{a}_i^{n+1} \tfrac{\Delta t}{2} \qquad \text{kick}$$

The evaluation of the updated acceleration $\mathbf{a}_i^{n+1} = \frac{d\mathbf{v}_i}{dt}$ according to (10) as well as the thermodynamic update (11) are applied in the midpoint in time of the integration step after the drift operation. The maximally permissible time step size Δt is determined using the CFL condition

$$\Delta t_i = 2 C_{\text{cfl}} \frac{H_i}{\max_j v_{ij}^{\text{sig}}} \qquad (17)$$

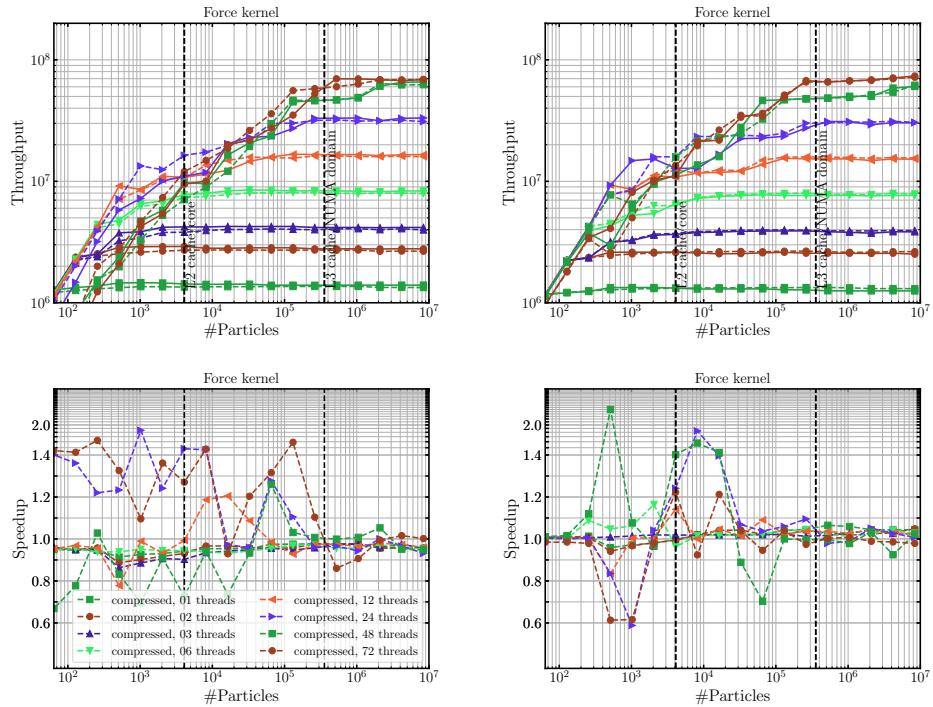where $0 < C_{\text{cfl}} \le 1$ is a free parameter, typically set to be 0.1.

Fig. 18. Measurements for the force kernel kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use three NUMA domains.
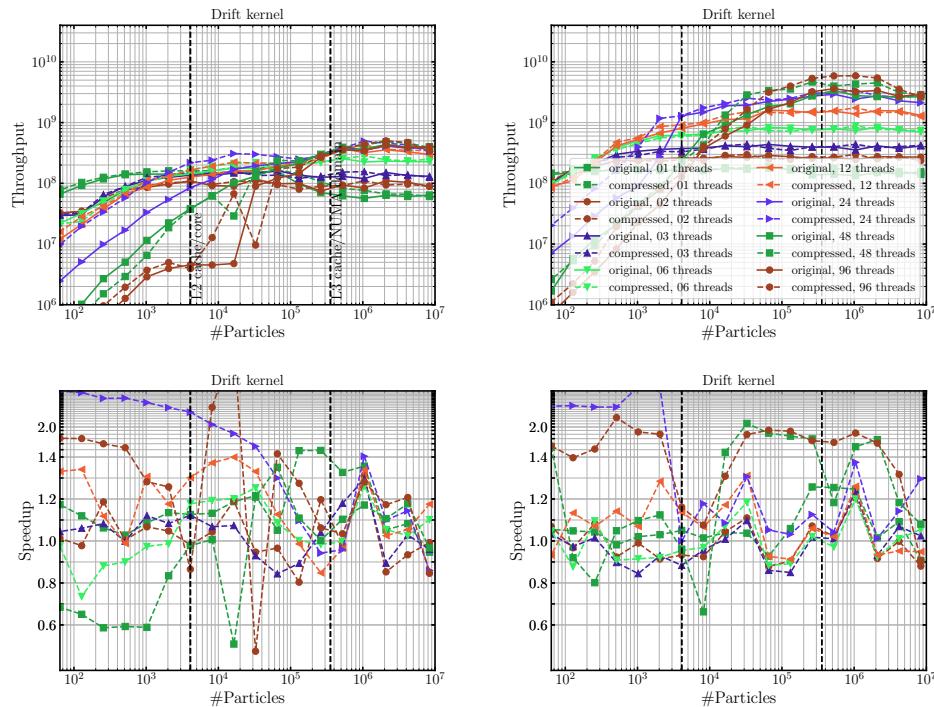
Fig. 19. Measurements for the drift kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use four NUMA domains.
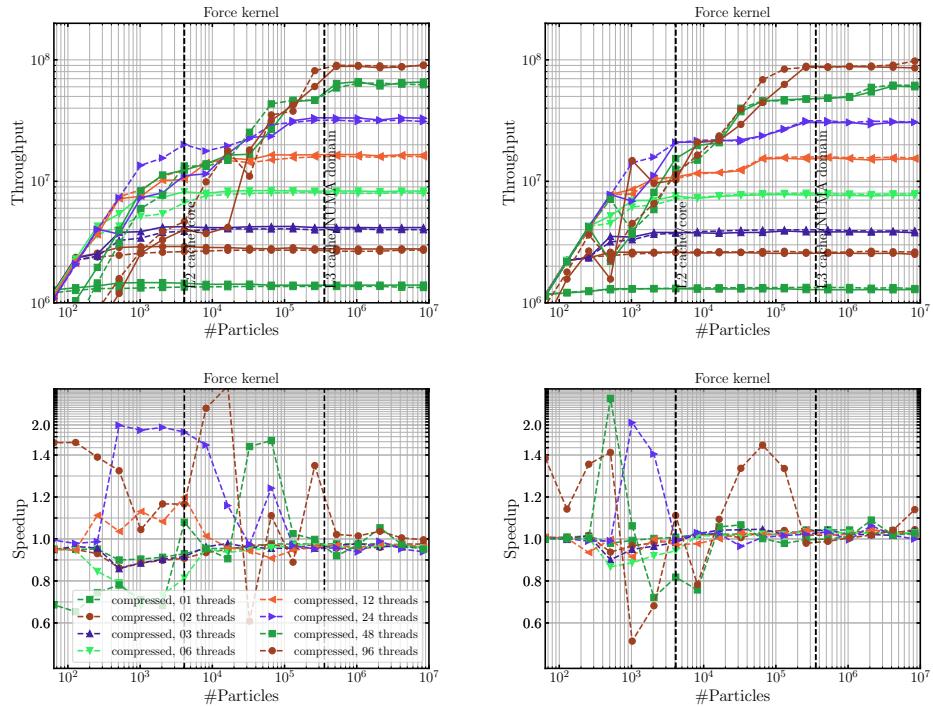
Fig. 20. Measurements for the force kernel kernel. Throughput (top) and speedup relative to uncompressed baseline version (bottom) on the Genoa testbed for stream-like access (left) and task-based access characteristics (right). We use four NUMA domains.