Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition

Stephen Nicholas Swatman*
University of Amsterdam
Amsterdam, The Netherlands
s.n.swatman@uva.nl

Ana-Lucia Varbanescu
University of Twente
Enschede, The Netherlands
a.l.varbanescu@utwente.nl

Andy Pimentel
University of Amsterdam
Amsterdam, The Netherlands
a.d.pimentel@uva.nl

Andreas Salzburger
European Organization for Nuclear Research
Geneva, Switzerland
andreas.salzburger@cern.ch

Attila Krasznahorkay
European Organization for Nuclear Research
Geneva, Switzerland
attila.krasznahorkay@cern.ch

ABSTRACT

We present a novel benchmark suite for implementations of vector fields in high-performance computing environments to aid developers in quantifying and ranking their performance. We decompose the design space of such benchmarks into *access patterns* and *storage backends*, the latter of which can be further decomposed into components with different functional and non-functional properties. Through compile-time meta-programming, we generate a large number of benchmarks with minimal effort and ensure the extensibility of our suite. Our empirical analysis, based on real-world applications in high-energy physics, demonstrates the feasibility of our approach on CPU and GPU platforms, and highlights that our suite is able to evaluate performance-critical design choices. Finally, we propose that our work towards composing vector fields from elementary components is not only useful for the purposes of benchmarking, but that it naturally gives rise to a novel library for implementing such fields in domain applications.

CCS CONCEPTS

• **Software and its engineering → Software performance**; *Software libraries and repositories*; *Abstraction, modeling and modularity*.

KEYWORDS

high-performance computing, benchmarking, vector fields, composition, meta-programming, CUDA

ACM Reference Format:
Stephen Nicholas Swatman, Ana-Lucia Varbanescu, Andy Pimentel, Andreas Salzburger, and Attila Krasznahorkay. 2023. Systematically Exploring High-Performance Representations of Vector Fields Through Compile-Time Composition. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578244.3583723

*Also with European Organization for Nuclear Research.

ICPE '23, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0068-2/23/04.
https://doi.org/10.1145/3578244.3583723

1 INTRODUCTION

Vector fields are ubiquitous in a variety of domains sciences such as meteorology [34], oceanography [26], and high-energy physics [22]. When developing applications which rely on vector fields, finding efficient data structures for storing and methods for accessing such fields can be paramount to achieving high performance. Unfortunately, there is no universal solution—let alone a performant one—for representing vector fields in software: the design space is far too large and the requirements are far too varied. In terms of functional requirements [14], for example, some applications might require two-dimensional fields while others might require three-dimensional data. Non-functionally, applications may exhibit different access patterns which can significantly affect the performance of a given implementation. Finally, the landscape of hardware on which domain applications are executed has become more complicated than ever: traditional homogeneous computing systems now compete with heterogeneous systems equipped with a variety of accelerators [5]. Thus, domain scientists must find methods of storing and accessing vector fields in heterogeneous environments which guarantee high performance in specific applications.

Currently, selecting representations of vector fields is an ad-hoc process based on developer experience and trial-and-error, neither of which provides any guarantees in finding the best-performing solutions. As far as we are aware, there are no comprehensive benchmark suites that can be used to systematically quantify and rank the performance of different vector field representations for a given application. In this paper, we introduce a systematic benchmarking approach that aims to cover the design space of vector field representations, to expose performance-relevant elements of this space, and to be easily extendable. To do so, we explicitly decompose the aforementioned design space into *access patterns* which model a field's *usage*, and *storage backends* which model the field's *implementation*. We then use compile-time meta-programming to generate benchmarks across the entire design space with far less effort than would be required by a conventional trial-and-error approach. Finally, we enable developers to directly apply the results of our benchmark suite through a novel library which exposes the same domain decomposition used by our suite for use in domain applications.

Our current design space covers five families of access patterns—each of which can be extensively configured—and nineteen components for constructing storage backends, which can be composed arbitrarily. We support benchmarks for both C++-compatible CPU platforms as well as CUDA-based general-purpose GPU (GPGPU) platforms, and we make it easy for users to add new access patterns and storage backends—including for new platforms.

In short, this paper makes the following contributions:

- We decompose domain applications of vector fields into access patterns and storage backends, thus constructing a vector field representation design space based on these two dimensions (Sections 3, 4, and 5);
- We propose a framework for the automated generation of a large number of benchmarks using the aforementioned decomposition, thus facilitating the exploration of the design space (Section 6);
- We enable users to leverage our benchmark suite in real-world applications by presenting a novel library that corresponds directly to the implementation of our benchmarks (Section 7).

2 BACKGROUND

In a formal mathematical sense, a vector field is a vector-valued mapping $\vec{f}$ that assigns to every element of some set S a vector in a vector space F, such that $\vec{f} : S \to F$ [13]. For the purposes of this work, we have opted to restrict the codomains of our vector fields to coordinate spaces, which consist of coordinates of arbitrary dimensionality over a given algebraic field. This restriction imparts additional structure upon our vector fields such that they map more naturally onto the design of modern computer systems, and while it excludes more exotic varieties of vector spaces such as function spaces, it still allows us to model many of the vector fields that are encountered in scientific computing.

2.1 Related Work

The representation of vector fields—and multi-dimensional data in a more general sense—has been the subject of intense study for many years. For example, Thiyagalingam et al. [39] investigate the performance of different layout schemes for data, but their work is limited to two-dimensional data and CPU-based platforms. Nocentino and Rhodes [30] evaluate the performance of Morton curve layouts on GPU platforms but they, too, consider only two-dimensional data and study a single application. Chatterjee et al. [7] study the performance of data layout schemes in depth, but their analysis is restricted to the application of matrix multiplication. Sarawagi and Stonebraker [36] evaluate different storage methods for large amounts of scientific data, potentially including vector fields, but their analysis is tailored specifically towards secondary and tertiary storage devices, while our work focuses on in-memory arrays. Edwards and Sunderland [10] introduce *Kokkos*, a library which supports the storage of data in heterogeneous environments using a variety of representations; while this is a very versatile and useful method for storing multi-dimensional arrays, we are not aware of any functionality in Kokkos that allows users to evaluate the performance of different representations.

In this work, we do not propose new access patterns or storage methods vector fields, instead, we focus on defining and systematically exploring a *design space* for the representation of vector fields that is as large as it is precisely due to the amount of existing research into the topic. We aim to provide developers of scientific applications with a comprehensive way of comparing all these choices and their effects on application performance.

2.2 Notation

Throughout this manuscript, we adhere to a common system of notation. When describing the types of vector fields, we use the syntax of dependent types [18]. For example, the statement $\prod_{n:\mathbb{N}} \mathbb{R}^n \to \mathbb{R}^n$ is used to mean that such a vector field exists *for all* natural values of n. We use double bracket notation to denote inclusive integer intervals, such that $[\![1, 3]\!]$ is equivalent to $\{1, 2, 3\}$. The symbol $\mathfrak{S}_n$ is used to denote the set of all permutations of $[\![1, n]\!]$. We denote vector values and vector-valued functions using overhead arrows, such as $\vec{f}$. In the context of types, we use $a + b$ to denote a sum type (a term of type a or a term of type b), and $a \times b$ to denote a product type (both a term of type a as well as a term of type b); we also use a^n to denote an n-tuple of type a.

3 DESIGN SPACE EXPLORATION

Our goal is to select the most appropriate representation for a given vector field. To this end, we need an approach which combines comprehensive coverage of the design options (i.e., the *design space*) with a systematic *exploration* of that space. In this section, we describe and motivate our envisioned design space, as well as our method for exploring it through the automated generation of benchmarks.

3.1 Design Space Dimensions

The performance of software using vector fields is an inextricable combination of the field's *application*, which determines how it is *used* (i.e., what the access pattern is) and its *implementation*, which determines how it is *built* (i.e., what computation is required to produce a result). Indeed, different implementations of vector fields—even if they produce the same result—can provide wildly different performance for the same application, and it is not necessarily obvious which implementations will provide the best performance in real-world scenarios. Thankfully, the fact that application and implementation are so intertwined when it comes to performance leads us naturally to a two-dimensional decomposition of the design space for such programs into *access patterns* and *storage backends*.

In this framework of thinking, an *access pattern* is an abstract model of a real-world application. Access patterns impose functional requirements on storage backends, such as the dimensionality of their vectors, and determine the locality of reference—both spatial and temporal—of vectors retrieved from a given field. In addition, access patterns are bound to specific programming platforms and, as a result, require storage backends to be compatible with a given platform. We detail the access patterns supported by our benchmark suite in Section 4.

A *storage backend*, then, fulfils the functional requirements imposed by a given access pattern, and introduces additional non-functional properties; in particular, storage backends model how

much performance is lost by adding functionality—such as, for example, the interpolation of vectors—to a program. In sampled vector fields, storage backends also map indices in a high-level coordinate space onto the memory of the system; a well-chosen storage backend should be capable of translating locality of reference in the high-level coordinate space (determined by the access pattern) into locality of reference in the system's memory, such that caches can be most effective. We explore storage backends in more depth in Section 5.

3.2 Exploration through Automated Benchmarking

Once the design space of vector field representations is defined, selecting the best performing solution for a specific application is a matter of exploring this space. Thus, we propose design space exploration through selective automated benchmarking: we define a benchmarking suite, from which we select and benchmark all representations feasible for the target application, and select the best performing one. For our benchmarking suite to be useful, we posit that it must meet three requirements. Firstly, it must be *comprehensive*: the suite must be able to approximate a large variety of real-world applications and methods of storing vector fields. Secondly, the benchmarks in our suite must be *specific*: they must be capable of identifying performance-relevant design choices and allow the user to evaluate their non-functional effects in depth and at a fine level of granularity. Finally, the suite should be *applicable*: the results of our benchmark suite should allow application developers to easily apply the results of our benchmarks to the development of their applications.

A naive solution to tackle *comprehensiveness* would be to write such a large number of benchmarks that most real-world applications would be represented by sheer chance. However, this would only shift the time-consuming exploration of the design space from the application developers to the authors of the suite. Instead, we rely on an automated exploration of the design space. In particular, we compute—at compile-time—the Cartesian product of available access patterns and storage backends, and generate a benchmark for every viable pair. This approach requires far less code to be written by hand. Additionally, our benchmarking suite is easily *extensible*: developers who find a particular application missing from the existing repository of access patterns can simply add it, and our suite will automatically generate benchmarks that combine the newly-added access pattern or application with all compatible storage backends. Correspondingly, users implementing new storage backends can easily benchmark their implementation against a number of existing access patterns. Our suite is implemented in C++ and makes extensive use of the Boost Mp11 meta-programming library [8] to perform type-level computation.

4 ACCESS PATTERNS

In this section, we describe the access patterns included in our benchmark suite; we present five families of access patterns, encompassing a total of sixteen variants: ten for CPU-based platforms and six for CUDA-based platforms. Most of these variants can be further distinguished though the use of compile-time parameters, which are shown in Table 1. For example, the Euler pattern can be compiled to operate on vector fields of any dimensionality, and it can be compiled with both single- and double-precision floating point numbers. Expanding the additional compile-time parameters of these access patterns for two- and three-dimensional cases results in a total of 208 access patterns which are distinct at compile-time. Finally, each access pattern can be configured with a series of run-time parameters which may further impact performance.

Table 1: List of Supported Access Patterns

Name	Variants	Compile-time parameters[1]
Scan	1 CPU / 1 CUDA	$\prod_{d,d':\mathbb{N}} \prod_{S,T:\mathcal{V}} S^{d+d'} \to T$
Random	1 CPU / 1 CUDA	$\prod_{d:\mathbb{N}} \prod_{S,T:\mathcal{V}} S^{d} \to T$
Euler†	2 CPU / 1 CUDA	$\prod_{d:\mathbb{N}} \mathbb{R}^d \to \mathbb{R}^d$
RK4†	2 CPU / 1 CUDA	$\prod_{d:\mathbb{N}} \mathbb{R}^d \to \mathbb{R}^d$
Lorentz†	4 CPU / 2 CUDA	$\mathbb{R}^3 \to \mathbb{R}^3$

† Access pattern is data-dependent.
[1] $\mathcal{V}$ denotes the family of finite-dimensional coordinate spaces.

4.1 Scan

The Scan pattern—given two parameters d and d'—iterates over a d'-dimensional slice of a $(d + d')$-dimensional vector field along each of the axes, visiting equidistant points in lexicographic order. The remaining d dimensions are static, and the indices in these dimensions are given by a d-dimensional coordinate. It follows that setting $d = 0$ simply scans the entire vector field. The Scan access pattern can operate on input vectors of finite dimensionality over any totally ordered monoid, but in this work we restrict ourselves to the real, natural, and integral numbers. The CPU-based implementation of the Scan access pattern iterates over the axes in order. The CUDA-based implementation is slightly more complex, as it uses multi-dimensional blocks to iterate over the vector field slice. Since CUDA supports only one, two, and three-dimensional kernels [32], d' is limited to $[\![1, 3]\!]$ when using this access pattern on a GPU. The shape of the kernel execution blocks and grid are performance-relevant parameters, as they affect the locality of vector field accesses.

4.2 Random

The Random access pattern generates random accesses in real- or integer-valued vector fields. Given a number of points m and two coordinates describing opposite corners of a hyper-box r, we generate m uniformly random coordinates $\vec{p_1}, \ldots, \vec{p_m} \in r$ and retrieve the value of the vector field at those positions. The Random access pattern is implemented both for CPU-based platforms as well as for CUDA-based GPUs.

4.3 Euler

Unlike the other access patterns mentioned so far, the Euler family of patterns introduces a dependency between the way a vector field is accessed and the contents of that vector field. Parameterized

by a number of agents m and a hyper-box r, this access patterns generates agents at uniformly random initial positions $\vec{p_1}, \ldots, \vec{p_m}$ in the volume r in exactly the same fashion as the RANDOM pattern. Then, a total of s steps of the Euler method [6] are used to find an approximate solution to the system of initial value problems given by the randomly generated initial positions, with the derivative function given by the vector field.

We implement three variants of the EULER access pattern. The first, EULER⟨DEEP⟩, processes agents in parallel, completing all required steps for each agent sequentially. The EULER⟨WIDE⟩ method instead processes one step for each agent before repeating the process until all agents have taken all required steps; we distinguish between these access patterns because they exhibit meaningfully different locality of reference. For GPGPU platforms, a variant of the EULER⟨DEEP⟩ is available.

4.4 RK4

The RK4 access pattern performs—in essence—the same function as the EULER access pattern, except that it uses a fourth-order Runge–Kutta method [35] rather than an Euler method (which is, in itself, a first-order Runge–Kutta method). The difference between the Euler method and the fourth-order Runge–Kutta method is that the latter makes four sub-steps in close vicinity to each other at every step; as such, these sub-steps naturally exhibit spatial locality. The RK4 access pattern has similar variants to the EULER access pattern.

4.5 LORENTZ

Finally, the LORENTZ access pattern is inspired by the application of vector fields in high energy physics, where they are used to model magnetic fields that alter the trajectory of charged particles according to the Lorentz force [15], such that the force $\vec{F}$ applied to a particle with charge q and momentum $\vec{v}$ at position $\vec{p}$ under the influence of two vector fields, the electric field $\vec{E}$ and the magnetic field $\vec{B}$, is given as follows:

$$\vec{F}(\vec{v}, \vec{p}; \vec{E}, \vec{B}) = q\big(\vec{E}(\vec{p}) + \vec{v} \times \vec{B}(\vec{p})\big) \tag{1}$$

In contrast to the EULER and RK4 patterns, the LORENTZ pattern does not initialise its agents at random positions, but at a given origin $\vec{o}$; in order to ensure that agents diverge, each agent is assigned a velocity vector of a given length i in a random direction. It is worth noting that the movement of agents away from the origin leads to shifts in access locality as the simulation progresses. The setup of this access pattern gives rise to an initial value problem where agents move through the vector field according to Equation 1, and we find approximate solutions to the final positions of the agents using both an Euler method and a fourth-order Runge–Kutta method. Since each of these methods has three distinct variants (see Sections 4.3 and 4.4), the LORENTZ pattern has a total of six variants: four variants for CPU-based platforms and two variants for GPU-based platforms. Note that this access pattern is not a full-fledged physics simulation, but rather an approximation of the access pattern such a simulation may exhibit; we simplify our access pattern by assuming identical charge and unit mass for all particles, by assuming the electric field to be zero, and by wrapping particles around when they exit the boundaries of the vector field.

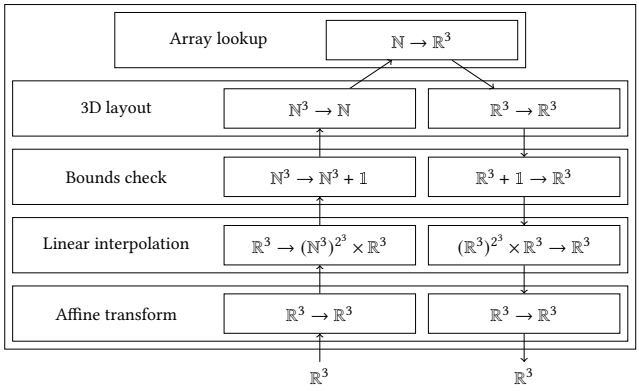


Figure 1: The internal structure of a storage backend, incorporating an affine transformation, trilinear interpolation, a boundary checking mechanism, and a three-dimensional layout scheme.

5 STORAGE BACKENDS

Where the aforementioned access patterns describe how vector fields are used, storage backends describe the inner workings of the vector fields, which impact both their functional and non-functional properties. In the context of vector fields, functional properties include—but are not limited to—the types of the input and output vectors and the way out-of-bounds accesses are sampled. Conversely, the non-functional properties of vector fields include access latency, throughput, and energy usage, which are determined by factors such as the layout of samples in memory and the use of hardware acceleration.

Even if we decompose the design space of vector field benchmarks as we have described in Section 3, the design space for storage backends remains dauntingly large. We posit that in order to capture this space, we need to continue our decomposition further such that we do not only describe benchmarks in terms of access patterns and storage backends, but that we additional break the latter down into their constituent components.

5.1 Backend Composition

Within our benchmark suite, complex vector field implementations are constructed through the composition of simple components, of which we define two distinct classes: *primitive backends* and *transformers*. Primitive backends provide behaviour that cannot be meaningfully deconstructed into smaller components; examples of such backends include the array backend which simply looks up a vector in the one-dimensional memory of the machine, and the constant vector backend which always outputs the same vector. Primitive backends are in principle usable as stand-alone vector field backends, but they lack—by design—much of the functionality that is desired in real-world applications. This functionality is correspondingly provided by backend *transformers*. Transformers are not fully-fledged storage backends by themselves, but can be *applied* to existing backends to imbue them with additional functional and non-functional properties.

To define the structure of backend transformers, we identify three distinct requirements of such transformers, the first of which is that transformers must be able to manipulate *input* coordinates before passing them to an underlying storage backend. We might consider, as an example, a backend transformer that imparts a two-dimensional layout onto an existing one-dimensional array of vectors in $\mathbb{R}^2$. The function of such a transformer is to convert a coordinate of type $\mathbb{N}^2$ into a coordinate of type $\mathbb{N}^1$. In other words, the transformer is equipped with a function of type $\mathbb{N}^2 \to \mathbb{N}^1$ which is applied to a coordinate *before* it is passed to the underlying vector field of type $\mathbb{N}^1 \to \mathbb{R}^2$, such that the entire composite backend has type $\mathbb{N}^2 \to \mathbb{R}^2$. We refer to this as the *contravariant* component of the backend transformer, in accordance with the idea that the profunctorial function arrow $\to$ is a contravariant functor in its first argument [28]. Secondly, a backend transformer must be able to modify the *output* of its underlying backend. For instance, a transformer which discards the second component of a two-dimensional real-valued vector applies a function of type $\mathbb{R}^2 \to \mathbb{R}^1$ *after* querying the underlying storage backend. If we consider the same $\mathbb{N}^1 \to \mathbb{R}^2$ backend as before, this results in a new backend of type $\mathbb{N}^1 \to \mathbb{R}^1$. We refer to this as the *covariant* component of the transformer.

Finally, a backend transformer must be able to communicate information between its contravariant and covariant components, and it must be able to apply simple underlying storage backends to complex structures. A prime example of these requirements is given by bilinear interpolation which, in its contravariant component, computes four integer-valued coordinates from a single real-valued coordinate as well as two weights which are used to linearly interpolate the output vector. Because the computation of the weights happens in the contravariant component of the transformer while the weighting happens in the covariant component, the transformer must somehow communicate these weights. In addition, the transformer must request *four* vectors from the underlying vector field, rather than one. Within the design we have proposed so far, both of these tasks are impossible. However, we can resolve both problems elegantly by applying a functor to the output of the contravariant component and the input of the covariant component of the transformer; in the case of bilinear interpolation, we might consider the functor $F(a) = a^4 \times \mathbb{R}^2$. This allows us to transparently lift the underlying storage backend such that communication becomes possible by embellishing our types with additional data, and we can use existing backends without the need to adapt them to more complex data types.

Definition 1. Given a functor F and types a_1, a_2, b_1, b_2, a backend transformer is a term of type $(a_2 \to F(a_1)) \times (F(b_1) \to b_2)$, such that a_1 and a_2 describe the contravariant component of the transformer, and the remaining types b_1 and b_2 describe the covariant component.

Definition 2. Composition of backend transformers is an operation

$$\begin{aligned} \circ_T : (a_3 \to F(a_2)) \times (F(b_2) \to b_3) \to \\ (a_2 \to G(a_1)) \times (G(b_1) \to b_2) \to \\ (a_3 \to (F \circ G)(a_1)) \times ((F \circ G)(b_1) \to b_3) \end{aligned} \tag{2}$$

such that

Table 2: List of Supported Primitive Storage Backends

Name	Platform	Field type
Array	CPU	$\prod_{T:\mathcal{V}} \mathbb{N} \to T$
CudaArray	CUDA	$\prod_{T:\mathcal{V}} \mathbb{N} \to T$
CudaPitch	CUDA	$\prod_{d:[\![1,3]\!]} \prod_{T:\mathcal{V}} \mathbb{N}^d \to T$
CudaTex	CUDA	$\prod_{d:[\![1,3]\!]} \prod_{d':[\![1,4]\!]} \mathbb{R}^d \to \mathbb{R}^{d'}$
Analytic	CPU	$\prod_{S,T:\mathcal{V}} S \to T$
Constant	Any	$\prod_{S,T:\mathcal{V}} S \to T$

$$(f_1, g_1) \circ_T (f_2, g_2) = (F(f_2) \circ f_1, g_1 \circ F(g_2)) \tag{3}$$

Definition 3. Application of a backend transformer to a storage backend is an operation

$$\begin{aligned} \$_T : (a_2 \to F(a_1)) \times (F(b_1) \to b_2) \to \\ (a_1 \to b_1) \to (a_2 \to b_2) \end{aligned} \tag{4}$$

such that

$$(f, g)\ \$_T\ h = g \circ F(h) \circ f \tag{5}$$

Storage backend transformers can be applied in any order as long as their types match, and arbitrarily many transformers can be composed. In Figure 1, we show an example of what the internal structure of a storage backend may look like in practice.

5.2 Primitive Backends

Table 2 presents a summary of the primitive backends currently supported by our suite. We describe them briefly in the following paragraphs.

5.2.1 Array. The Array backend represents perhaps the most trivial in-memory vector field: a one-dimensional array of vectors. While such one-dimensional vector fields are of little use in practice, they serve as the basis for virtually all sampled vector field backends that we can construct in CPU-accessible memory.

5.2.2 CudaArray. The CudaArray backend functions similarly to the Array backend, except that its contents inhabit the host-inaccessible memory of a CUDA device. Note that this backend does *not* use the opaque arrays which NVIDIA refers to as "CUDA arrays" [32]; for a backend based on these opaque structures, we provide the CudaTex backend (Section 5.2.4).

5.2.3 CudaPitch. As an alternative to ordinary CUDA device memory, we provide a primitive backend based on pitched memory allocated using the `cudaMalloc3D` API[1] [32]. The advantage of using this method is that the CUDA runtime may pad the allocation size to ensure performance-favourable data alignment. Because pitched CUDA memory opaquely handles multi-dimensional accesses, this backend primitively supports multi-dimensional coordinates.

[1]This function is capable of allocating both one- and two-dimensional arrays in addition to three-dimensional ones.

Table 3: List of Supported Backend Transformers

Name	Type
Pitched†	$\prod_{n:\mathbb{N}} \prod_{T\ \mathrm{Type}} (\mathbb{N}^n \to \mathbb{N}) \times (T \to T)$
Morton†	$\prod_{n:\mathbb{N}} \prod_{T\ \mathrm{Type}} (\mathbb{N}^n \to \mathbb{N}) \times (T \to T)$
Hilbert	$\prod_{T\ \mathrm{Type}} (\mathbb{N}^2 \to \mathbb{N}) \times (T \to T)$
Shuffle	$\prod_{n:\mathbb{N}} \prod_{p:\mathfrak{S}_n} \prod_{S,T\ \mathrm{Type}} (S^n \to S^n) \times (T \to T)$
Default	$\prod_{S,T\ \mathrm{Type}} (S \to S + \mathbb{1}) \times (T + \mathbb{1} \to T)$
Wrap†	$\prod_{S,T\ \mathrm{Type}} (S \to S) \times (T \to T)$
Nearest	$\prod_{n:\mathbb{N}} \prod_{T\ \mathrm{Type}} (\mathbb{R}^n \to \mathbb{N}^n) \times (T \to T)$
Linear	$\prod_{n:\mathbb{N}} (\mathbb{R}^n \to (\mathbb{N}^n)^{2^n} \times \mathbb{R}^n) \times ((\mathbb{R}^n)^{2^n} \times \mathbb{R}^3 \to \mathbb{R}^n)$
Affine	$\prod_{n:\mathbb{N}} \prod_{T\ \mathrm{Type}} (\mathbb{R}^n \to \mathbb{R}^n) \times (T \to T)$

† Backend transformer with additional variants.

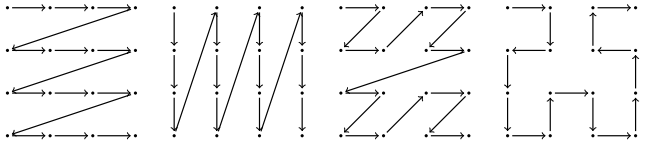
5.2.4 CudaTex. CUDA textures are useful for representing vector fields [33] because they are nominally used to store texels which can be represented using one, two, three, or four-dimensional vectors of floating point numbers analogously to how pixels can be represented by greyscale values or by red, green, blue, and optionally alpha values. The advantage of using textures to represent vector fields in a broader sense is that they feature hardware-accelerated interpolation (both nearest-neighbour and linear, up to nine bits of precision [32]), various boundary checking methods, and cache-friendly storage layouts [40]. CUDA textures support real-valued inputs up to three dimensions and real-valued outputs up to four dimensions.

5.2.5 Analytic. In some applications, vector fields can be described entirely analytically, removing the need to store the field in memory. Our benchmark suite provides a general analytic vector field, wrapping an arbitrary user-provided vector-valued function.

5.2.6 Constant. A constant-valued vector field returns the same vector regardless of its input, performing only the bare minimum computation necessary in order to model a vector field. Such fields provide performance that is both very high and very predictable, making them useful for establishing upper bounds for the performance that can be achieved under a given access patterns.

5.3 Backend Transformers

The primitive backends described in Section 5.2 are, by design, too simple for most real-world use cases; rather, they are designed to be used in conjunction with backend transformers which imbue them with additional functional properties. Of course, most of these properties do not come for free, and virtually any behaviour we can add to a storage backend comes at the cost of performance. The advantage of our approach, even if it may seem unnecessarily granular, is that it is *specific*: it allows us to pick—and pay for—only the functionality we need. In addition, it makes it trivial to exchange transformers in order to compare their performance. In this section, we describe the different transformers that we provide as part of our benchmark in addition to a brief summary in Table 3.



(a) x-major **(b) y-major** **(c) Morton** **(d) Hilbert**

Figure 2: Examples of storage orders for two-dimensional arrays (of size 4×4), with arrows indicating the sequence of indices in the underlying one-dimensional memory.

5.3.1 Pitched. In most modern computer systems, memory is presented in a one-dimensional fashion; a request is made for the n-th byte in the address space, and the corresponding byte (or, more commonly, set of bytes) is returned. In order to represent multi-dimensional arrays in a fundamentally one-dimensional address space, bijective indexing functions translate multi-dimensional coordinates into one-dimensional ones. Perhaps the most ubiquitous method for laying out multi-dimensional data is in a *pitched* (or lexicographic) fashion. Examples of two-dimensional pitched layouts in which the x and y axes are major are shown in Figures 2a and 2b, respectively.

5.3.2 Morton. Pitched storage orders provide, informally, maximal spatial locality in one dimension at the expense of locality in all other dimensions [20]. While this can be a desirable property, many real-world applications (modelled, for example, by the Euler and Lorentz access patterns from Sections 4.3 and 4.5) exhibit locality in more than a single dimension. Such applications may benefit from laying data out according to a space-filling curve [24], which provides a compromise between locality in multiple dimensions [27]. A common example of a space-filling curve is the Morton curve, shown in Figure 2c. Using such a curve, a multi-dimensional index is converted to a one-dimensional index by interleaving the digits of the binary representations of the input coordinates as in the following example:

$$f(5,3,4) = f(101_2, 011_2, 100_2) = 101010110_2 = 342_{10}$$

The downside of Morton curve layouts is that the calculation of indices requires a significant amount of bit-manipulation, which may negate the benefits of the improved locality: d-dimensional coordinates with scalar types of width n bits require at least $3dn$ bit-wise operations: one bit-wise disjunction, one conjunction, and one barrel shift for each bit in each dimension. It must be noted, however, that the calculation of Morton curve indices can be accelerated greatly on x86 architectures equipped with the BMI2 instruction set extension [23], which introduces the PDEP instruction[2]. Assuming that the necessary deposition masks are computed at compile time, index calculations can be performed with as few as $2d$ operations: one bit-wise disjunction and one bit-deposition per scalar in the coordinate. Thus, indices based on Morton curves can be computed with latency and throughput similar to more traditional

[2] On most recent architectures, this instruction executes with the same latency as other bit-wise instructions, but it is emulated in microcode in pre-Zen 3 AMD processors, which may degrade performance significantly [11].

```
imul    0x8(%rsi),%rdx
add     %rcx,%rdx
imul    0x10(%rsi),%rdx
add     %rdx,%r8
mov     0x18(%rsi),%rdx
lea     (%r8,%r8,2),%rax
lea     (%rdx,%rax,8),%rax
vmovdqu (%rax),%xmm0
mov     0x10(%rax),%rax
vmovdqu %xmm0,(%rdi)
mov     %rax,0x10(%rdi)
mov     %rdi,%rax
ret
```

(a) Without coordinate shuffling.

```
imul    0x8(%rsi),%rcx
add     %r8,%rcx
imul    0x10(%rsi),%rcx
lea     (%rcx,%rdx,1),%rax
mov     0x18(%rsi),%rdx
lea     (%rax,%rax,2),%rax
lea     (%rdx,%rax,8),%rax
vmovdqu (%rax),%xmm0
mov     0x10(%rax),%rax
vmovdqu %xmm0,(%rdi)
mov     %rax,0x10(%rdi)
mov     %rdi,%rax
ret
```

(b) With $(2, 3, 1)$-shuffling.

Listing 1: Comparison of x86-64 assembly generated by *gcc* 11.2 with the -O3 flag for a backend consisting of an array lookup and a three-dimensional pitched layout, with and without shuffling; although Listing (b) is generated using an additional transformer, the volume of assembly does not increase.

pitched layouts. It is worth noting that Morton curve layouts without intermediate lookup tables require the size of the array in each dimension to be a power of two. In order to represent arrays with other sizes, the array must be padded and space must therefore be wasted. Representing a d-dimensional array in such a way requires at most $2^d - 1$ times more memory than an optimal layout.

5.3.3 Hilbert. Another space filling curve is due to Hilbert [17]. Because this curve provides good locality (see Figure 2d for an example), its application to the storage of multi-dimensional data has been intensively studied [12, 25, 29]. The Hilbert curve is well-understood in two dimensions, but can also be generalised to three dimensions in a large number of ways [16]. Currently, we support layouts based on Hilbert curves only in two-dimensions.

5.3.4 Shuffle. All multi-dimensional layouts in our suite treat the first coordinate as the major coordinate, followed by the second, and so forth. In the two-dimensional pitched case, this is often referred to as a *row-major* layout; in order to extend this concept to an arbitrary number of dimensions, we will instead refer to it somewhat more systematically as a $(1, 2)$-permuted layout. In a more general n-dimensional sense, all of our layouts are $(1, 2, \ldots, n-1, n)$-permuted. Of course, such layouts may not always be optimal: in cases where multi-dimensional array accesses are anisotropically distributed along the axes, alternate permutations may provide higher performance due to caching [21].

The number of possible permutations grows factorially with the number of dimensions; the one-dimensional case has a single permutation, the two-dimensional case has two (namely, row-major and column-major), the three-dimensional case has as many as six possible permutations, and so forth. Due to this rapid rate of growth, implementing separate storage layouts for every possible permutation is not feasible. Rather, we implement a simple family of transformers which re-order coordinates before passing them to an underlying backend. As an example, a $(2, 1)$-permuted shuffling transformer swaps the values of a two-dimensional coordinate, causing an underlying row-major layout to behave externally like a column-major layout. Such coordinate shuffling transformers receive their permutations as compile-time parameters, allowing them to be optimised away; a simple example of a compiler's ability to do so is demonstrated in Listing 1.

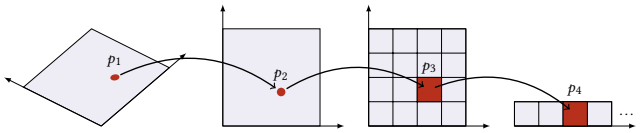


Figure 3: An example of how a storage backend takes a coordinate p_1 in a geometrically meaningful coordinate space to a coordinate space that maps onto a two-dimensional array (p_2 and p_3), and finally onto an address p_4 in memory.

5.3.5 Default. None of the backend transformers provided by our work incorporate boundary checking, as doing so would violate the separation of concerns between backends, and could affect performance. Of course, this means that invalid accesses invariably lead to undefined behaviour or segmentation violations. In order to alleviate this problem, boundary checking behaviour is provided by backend transformers, including the Default transformer which—if the requested coordinate lies outside a given set of bounds—returns a user-provided default value.

5.3.6 Wrap. An alternative to the Default transformer is given by extending a vector field to infinite size, which we refer to as wrapping. We currently support clamping (extending the edges of the vector field to infinity), tiling (repeating the vector field throughout the entire space), and mirrored tiling as wrapping methods.

5.3.7 Nearest. Most of the transformations hitherto described deal with integer-valued domains. While such vector fields are useful in domains such as image processing, many real-world use cases demand real-valued coordinates. In order to transform a vector field with integral-valued domain into one with a real-valued domain, we support nearest-neighbour interpolation, where the value of a real-valued coordinate is equal to the closest (by rectilinear distance) integer-valued coordinate in the underlying field [4].

5.3.8 Linear. By providing a weighted linear combination of multiple neighbouring points, linear interpolation can provide more accurate results than a nearest-neighbour method at the cost of additional computation [4]. Notably, linear interpolation in d dimensions requires 2^d accesses into the underlying vector field, all of which are guaranteed to be corners of a unit-sided hyper-cube. As a result, there exists a strong and unbiased spatial locality between the points necessary to perform such interpolation.

5.3.9 Affine. Interpolation methods provide a way to convert index spaces addressed by positive integers to spaces that can be accessed by real numbers, but they fail to impart geometric meaning upon their input. For most real-world applications, such coordinates need to be transformed such that they map onto a more meaningful coordinate space, as shown in Figure 3. Currently, our benchmark suite supports arbitrary affine transformations such as translation, scaling, rotation, and shearing through a user-supplied transformation matrix. The impact of such a transformation on the performance of a vector field is potentially significant, as it requires a $(d + 1)$-dimensional matrix-vector multiplication.

5.4 Performance Considerations

Although the compositional design of storage backends in our suite helps to define and explore the design space, we must ensure that this approach is not detrimental to performance; if it were, the results of our benchmarks would not be applicable to non-composite real-world vector field representations. Our approach of composing benchmarks at compile time guards us against performance degradation as a result of (1) dynamic function dispatching due to run-time polymorphism; and (2) reductions in the optimisation space afforded to the compiler.

Indeed, our approach entirely avoids the overhead of dispatch table look-ups which are necessary when employing run-time polymorphism [9]; because the entire code path is known at compile-time, there is no need to dynamically dispatch function calls. While such look-ups are often not performance-critical in larger applications, we envision vector fields as very-high throughput structures where such overhead would be undesirable.

Secondly, a compositional approach risks reducing the optimisation space of the compiler by enabling—or, depending on the design, requiring—the compiler to emit separate symbols for all possible transformers. During the subsequent composition phase, the compiler may be unable to optimise the code *across* different components of the storage backend, thereby degrading performance. Through the use of compile-time composition, we afford the compiler a complete view of all the components of a storage backend, allowing it to optimise across function boundaries. An example of this was shown previously in Listing 1, where a compiler was able to completely eliminate an additional backend transformation, incorporating additional behaviour into the vector field without incurring overhead.

6 CASE STUDY

In this section, we demonstrate the use of our benchmark suite in a real-world scenario: the propagation of charged particles through magnetic fields, a common problem in the field of high-energy physics.

6.1 Experimental Setup

To test our benchmarks on data that is representative of real-world scenarios, we use a solenoidal magnetic field generated by the ACTS software package [2]; this vector field is representative of real-world high-energy physics applications and uniformly samples a three-dimensional magnetic field in a range of $-10\,000$ mm to $10\,000$ mm in the x- and y-axes, and a range of $-15\,000$ mm to $15\,000$ mm in the z-axis. The data is sampled at a resolution of 100 mm, resulting in a total of $201 \times 201 \times 301$ samples. We store this data using single-precision IEEE 754 floating-point numbers, resulting in a total data size of 145.9 MB.

We execute our benchmarks on six distinct devices. Five of these devices, namely an AMD EPYC 7402P CPU of the *Zen 2* microarchitecture [1, 37] as well as NVIDIA A2, A4000, A6000, and A100[3] GPUs [31] of the *Ampere* architecture, are part of the DAS-6 cluster [3]. The final device, an Intel Xeon E5-2630 v3 CPU [19] of the *Haswell* microarchitecture, is part of the DAS-5 cluster [3].

[3]The PCI-e model with 40 GB of HBM2e memory.

```
benchmark::register_product_bm<
    boost::mp11::mp_list<
        Lorentz<Euler>,
        Lorentz<RungeKutta4>,
        RungeKutta4Pattern,
        EulerPattern,
        Random,
        Scan>,
    boost::mp11::mp_list<
        FieldConstant,
        FieldTex<TexInterpolateLin>,
        FieldTex<TexInterpolateNN>,
        Field<InterpolateNN, LayoutStride>,
        Field<InterpolateNN, LayoutMortonNaive>,
        Field<InterpolateLin, LayoutStride>,
        Field<InterpolateLin, LayoutMortonNaive>>>();
```

Listing 2: An example of C++ code used to generate our CUDA benchmarks. Given a compile-time list of six access patterns and a list of seven storage backends, we generate forty-two benchmarks.

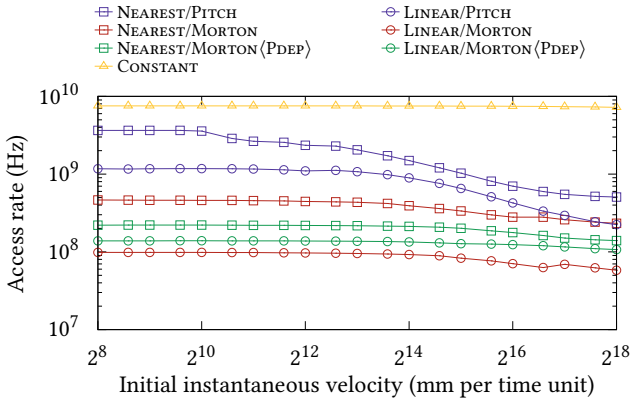
The code for our CPU-based platforms was compiled with gcc 11.2, and code for the NVIDIA GPU was compiled using nvcc 11.5 (targeting PTX versions 8.0 and 8.6).

6.2 Benchmarking Method

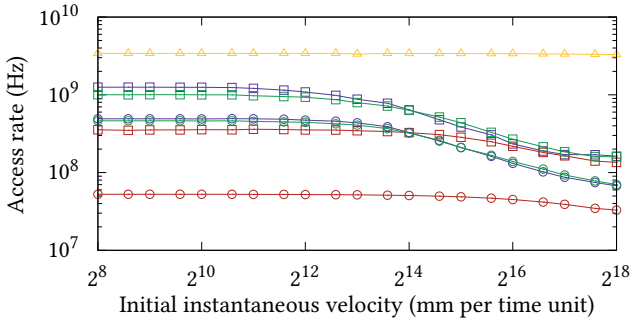
The first step in applying our benchmarking-based design space exploration approach is to select an access pattern that approximates the targeted application. In this case, we use the Lorentz access pattern, designed to closely represent the domain-specific application we are investigating (see Section 4.5). For the sake of brevity, we test the variants of this access pattern using only the Euler method—in a depth-first fashion—although our compositional approach to generating benchmarks makes it trivial to repeat this analysis for other access patterns: Listing 2 shows an example of how a much broader range of benchmarks can be generated using the compile-time meta-programming techniques described in Section 3.2.

Next, we construct a variety of storage backends with the necessary functional properties, namely (1) three-dimensional indexing; (2) real-valued indexing; and (3) appropriate transformations from the field's real-world geometry. This leads us to the composition of fourteen storage backends; of these, seven are suited for CPU-based experiments and seven are suited for GPU-based experiments. These backends include constant-valued vector fields and various combinations of primitive backends, multi-dimensional layouts, and interpolation schemes. In our experiments, the constant-valued vector fields serve as an upper bound for the performance of a certain access pattern on a given device, capturing the maximum throughput of the computation inherent to the application rather than the access to the vector field. It is worth noting that we only investigate a subset of storage backends that could be constructed for the application under investigation; indeed, additional functional behaviour such as boundary checking could be incorporated, but we chose to omit this due to the combinatorial growth of the number of resulting experiments.

The selection of our six devices with seven storage backends each yields a collection of forty-two benchmarks. Each of these benchmarks has a number of run-time parameters, which can additionally be varied. For this case study, we set the number of steps to



(a) AMD EPYC 7402P (48 threads)



(b) Intel Xeon E5-2630 v3 (2 sockets, 32 threads)

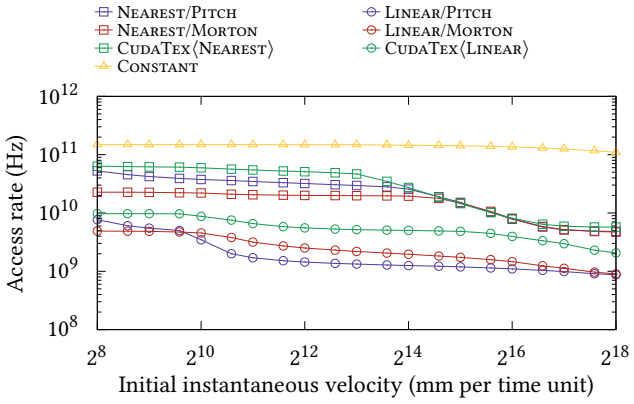
Figure 4: Vector field access throughput for the Lorentz access pattern with a depth-first Euler method for two different CPUs.

8192, the step size to 10^{-3} time units, and the number of agents to 65 536. This leaves us with a final parameter, the magnitude of the initial velocity vector, which we vary between 2^8 and 2^{18} millimeters per time unit. As we grow the initial velocity of the agents in our simulation, particles take larger steps through the vector field resulting in reduced locality, and understanding the magnitude of this effect is the goal of our analysis.

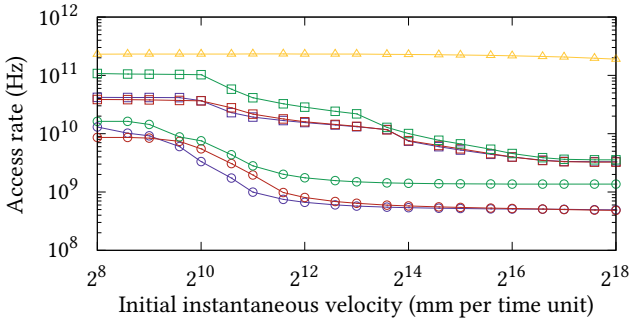
With our collections of benchmarks now selected and parameterised, we obtain a total of 882 distinct experiments. For each of these experiments, we measure the throughput—in accesses per second—from the vector field by dividing the total number of accesses by the total wall-clock runtime of the benchmark. Each experiment is repeated fifty times, and we report the mean throughput of these runs. Our benchmark suite automatically gathers additional descriptive statistics such as the minimal and maximal throughput as well as the variance between runs, but we have omitted these metrics from this section because the variance of our results is very small.

6.3 Results

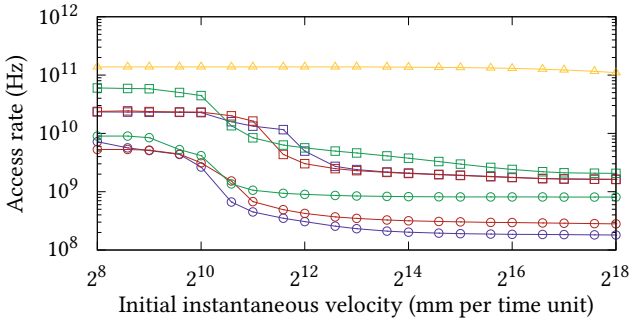
The results of our experiments are shown in Figure 4 for the CPU platforms and in Figure 5 for the GPU platforms. From the perspective of a developer, inspection of these data provides valuable



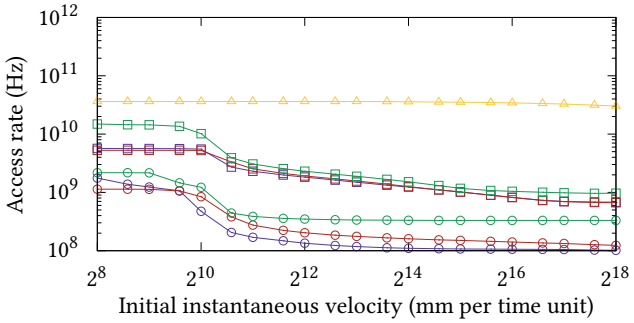
(a) NVIDIA A100



(b) NVIDIA A6000



(c) NVIDIA A4000



(d) NVIDIA A2

Figure 5: Vector field access throughput for the Lorentz access pattern using 128 threads per block for four different NVIDIA GPUs.

insights that motivate application development. We analyse a few such insights in the following paragraphs.

Firstly, the representation of vector fields in these modelled applications is an important design decision: the difference in throughput between different storage backends can be an order of magnitude or larger. Furthermore, our benchmark provides a clear ranking between different backends at each point in the chosen parameter space: given a specific device and parameterisation of the access pattern, the storage backend with the highest throughput can be easily selected. The inclusion of constant-valued vector fields provides additional insights into the upper bounds of storage backends, thus allowing us to compare other backends against this bound and calculate the fraction of maximally achievable performance obtained with each backend.

Secondly, our results show that the performance of vector fields is also strongly dependent on the properties of the access pattern. Even though all of our results are based on the Lorentz access pattern, performance behaviour varies significantly with the initial velocity of the simulation agents, which correlates with the locality of reference in the logical three-dimensional space. Unsurprisingly, all storage backends across all devices perform worse as the initial velocity increases and locality decreases, but the degree at which performance is lost varies significantly between storage backends. This is apparent, for example, in Figure 5a: the storage backend incorporating linear interpolation and Morton curve indexing performs relatively poorly for lower initial velocities, but outperforms the more traditional pitched layout at higher initial velocities due to the increased preservation of locality afforded by the Morton layout. In Figure 5c, we observe three distinct regimes: at low and high initial velocities, the storage backend based on texture memory outperform other backends with nearest neighbours, but at intermediate velocities the pitched layout and Morton layouts provide the best performance.

Finally, our results indicate strong sensitivity of our benchmarks to the properties and features of the hardware on which it is executed. Figure 5 shows the results of our benchmark when executed on four NVIDIA GPGPUs of the same architecture; beyond constant-factor performance differences due to differences in memory bandwidth and arithmetic performance, the devices under investigation exhibit different behaviour across the range of parameters tested. For example, Figure 5b shows that Morton curve layouts for linearly interpreted fields outperform pitched layouts only in a specific regime of initial velocities—between approximately 1024 and 4096 millimeters per unit of time—whereas this regime extends indefinitely in Figure 5c. Although we do not fully understand the source of this discrepancy, these results indicate that our benchmark can also be used to motivate the choice of processor or accelerator for an application.

Although a complete analysis of the data presented in this chapter is beyond the scope of this paper, we find the results presented in this section to be strong indicators of the *comprehensiveness* of our benchmarking methodology: we were able to gather results using a variety of storage backends through little more effort than selecting the benchmarks at run-time. Furthermore, our results are *specific* as they show clear differences in performance between different storage backends: even relatively similar backend which differ by only a single component exhibit significantly performance patterns, indicating that our suite is capable of comparing different representations at fine levels of granularity.

7 APPLICABILITY, GENERALISATION, AND LIMITATIONS

The primary purpose of our benchmark-based design space exploration is to guide developers in the selection of appropriate vector fields representations for their target applications. Once the selection is made, developers are free to implement their own vector fields from scratch. However, our benchmark implementation—by design—allows direct code re-use in the target application. In fact, our implementation that works primarily as a benchmark suite can be re-used as a C++ library which allows users to compose vector field implementations from the same components that we use in the benchmark. In essence, our benchmark suite and library are co-designed such that the library allows for the construction of benchmarks and real-world applications, while the benchmark allows insight into the performance of the different vector fields that can be constructed using the library. Our library provides features such as loading and storing vector fields to disk, and converting vector fields between different storage backends (including between CPU- and GPU-based backends, which automatically moves memory between devices wherever necessary). The source code for both our benchmark suite and our library are available as free software under the MPL 2.0 license [38].

We are confident that both the idea of composing *applications* and *implementations* of data structures at compile time can be generalised to other classes problems using similar techniques with the ones proposed in this work. Vector fields are excellent candidates for such an approach due to the wide range of access patterns and storage backends. Applications and data structures for which the design space is smaller might benefit less from the generation of benchmarks (as these benchmarks would be fewer), but the benefits of systematic exploration and automation remain relevant. Similarly, the decomposition of storage backends is transferable to other data structures, and it would be especially interesting for other kinds of multi-dimensional data. However, the dimensions of the design space could be less broad, because functionality such as interpolation and spatial transformations do not apply to all multi-dimensional structures. In a nutshell, the generalisation of the approach to other applications is easy to achieve, but its urgency is determined by the complexity of the application domain.

Finally, it is worth noting that our benchmark suite has limitations of which users should be aware. Chiefly, our suite models applications at an abstract level that omits non-functional effects that might be present in real-world scenarios. For example, the access patterns in our suite assume that the vector field is the only data structure being used by the application at a given time. In real-world applications, multiple regions of memory may be accessed at the same time by one or more threads. This may significantly alter the cache behaviour and thereby the performance of an application when compared to our benchmarking suite. Furthermore, our suite does not currently support accessing data stored on persistent media (such as through `mmap`) which may be relevant for applications using vector fields too large to fit entirely in the main memory of a given machine. Finally, we do not currently support fields of

heterogeneous vectors such as fields that combine vectors of both single and double precision floating point numbers, or vectors of different dimensionalities.

8 CONCLUSION

Vector fields are performance-critical components of scientific applications, but we lack the tools to quantify and rank their performance. In this work, we introduce a comprehensive benchmarking suite that enables developers of applications using vector fields to select appropriate, high-performance vector field representations through systematic design-space exploration. To create a comprehensive, specific, and applicable benchmark suite, we decompose vector field representations into access patterns and storage backends. These can be combined at compile-time to construct hundreds of unique benchmarks, each with additional run-time parameters. We further decompose storage backends into primitive backends and backend transformers, which can be used to add functional and non-functional properties to vector fields, both in a benchmarking setting as well as in domain-scientific use cases. We show that the use of template meta-programming allows us to automatically generate a large number of high-performance benchmarks, providing software that compromises neither performance nor usability.

In order to evaluate the efficacy of our benchmarking suite, we have applied it to analyse the performance of a range of vector field storage backends in a real-world problem in the domain of high-energy physics. Thereby, we have shown that the implementation of vector fields can play an important role in achieving high performance in real-world applications, and we have demonstrated that our benchmarking suite is capable of capturing the design space for such implementations.

In the future, we aim to expand our benchmarking suite and library to support a broader variety of functional and non-functional behaviour. For example, we aim to add support for generalised multi-dimensional Hilbert curves to lay out multi-dimensional data, as well as cubic interpolation.

REFERENCES

[1] Advanced Micro Devices, Inc. 2019. *2nd Gen AMD EPYC 7402P.* Advanced Micro Devices, Inc. https://www.amd.com/en/products/cpu/amd-epyc-7402p

[2] Xiaocong Ai, Corentin Allaire, Noemi Calace, Angéla Czirkos, Markus Elsing, Irina Ene, Ralf Farkas, Louis-Guillaume Gagnon, Rocky Garg, Paul Gessinger, Hadrien Grasland, Heather M. Gray, Christian Gumpert, Julia Hrdinka, Benjamin Huth, Moritz Kiehn, Fabian Klimpel, Bernadette Kolbinger, Attila Krasznahorkay, Robert Langenberg, Charles Leggett, Georgiana Mania, Edward Moyse, Joana Niermann, Joseph D. Osborn, David Rousseau, Andreas Salzburger, Bastian Schlag, Lauren Tompkins, Tomohiro Yamazaki, Beomki Yeo, and Jin Zhang. 2022. A Common Tracking Software Project. *Computing and Software for Big Science* 6, 1 (13 Apr 2022), 8. https://doi.org/10.1007/s41781-021-00078-8

[3] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 05 (5 2016), 54–63. https://doi.org/10.1109/MC.2016.127

[4] Paul Bourke. 1999. Interpolation methods. (Dec. 1999).

[5] Rainer Buchty, Vincent Heuveline, Wolfgang Karl, and Jan-Philipp Weiss. 2012. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurrency and Computation: Practice and Experience* 24, 7 (2012), 663–675. https://doi.org/10.1002/cpe.1904

[6] John Charles Butcher. 2016. *Numerical Differential Equation Methods.* John Wiley & Sons, Ltd, Chichester, West Sussex, England, Chapter 2, 55–142. https://doi.org/10.1002/9781119121534.ch2

[7] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. 2002. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems* 13, 11 (2002), 1105–1123. https://doi.org/10.1109/TPDS.2002.1058095

[8] Peter Dimov. 2017. *Boost.Mp11: A C++11 metaprogramming library.* The Boost Organization. https://www.boost.org/doc/libs/master/libs/mp11/doc/html/mp11.html

[9] Karel Driesen and Urs Hölzle. 1996. The Direct Cost of Virtual Function Calls in C++. *SIGPLAN Not.* 31, 10 (10 1996), 306–323. https://doi.org/10.1145/236338.236369

[10] H. Carter Edwards and Daniel Sunderland. 2012. Kokkos Array Performance-Portable Manycore Programming Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores* (New Orleans, Louisiana) *(PMAM '12)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2141702.2141703

[11] Agner Fog et al. 2011. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering* 93 (2011), 110.

[12] Patrick Franco, Giap Nguyen, Remy Mullot, and Jean-Marc Ogier. 2018. Alternative patterns of the multidimensional Hilbert curve. *Multimedia Tools and Applications* 77, 7 (2018), 8419–8440. https://doi.org/10.1007/s11042-017-4744-4

[13] Antonio Galbis and Manuel Maestre. 2012. *Vectors and Vector Fields.* Springer US, Boston, MA, 1–17. https://doi.org/10.1007/978-1-4614-2200-6_1

[14] Martin Glinz. 2007. On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)* (Delhi, India). IEEE, New York, NY, USA, 21–26. https://doi.org/10.1109/RE.2007.45

[15] David Jeffrey Griffiths. 1999. *Introduction to Electrodynamics.* Prentice Hall, Upper Saddle River, New Jersey, United States of America.

[16] Herman Haverkort. 2011. An inventory of three-dimensional Hilbert space-filling curves. https://doi.org/10.48550/ARXIV.1109.2323

[17] David Hilbert. 1891. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.* 38, 3 (1891), 459–460. https://doi.org/10.1007/BF01199431

[18] Martin Hofmann. 1997. *Syntax and semantics of dependent types.* Springer London, London, 13–54. https://doi.org/10.1007/978-1-4471-0963-1_2

[19] Intel Corporation. 2014. *Intel Xeon Processor E5-2630 v3* . Intel Corporation. https://ark.intel.com/content/www/us/en/ark/products/83356/intel-xeon-processor-e52630-v3-20m-cache-2-40-ghz.html

[20] Mahmut Taylan Kandemir, Alok Choudhary, J. Ramanujam, N. Shenoy, and Prithviraj Banerjee. 1998. Enhancing spatial locality via data layout optimizations. In *Euro-Par'98 Parallel Processing*, David Pritchard and Jeff Reeve (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 422–434. https://doi.org/10.1007/BFb0057885

[21] Mahmut Taylan Kandemir, J. Ramanujam, and Alok Choudhary. 1999. Improving cache locality by a combination of loop and data transformations. *IEEE Trans. Comput.* 48, 2 (1999), 159–167. https://doi.org/10.1109/12.752657

[22] Fabian Klimpel. 2020. Track propagation for different detector and magnetic field setups in Acts. *Journal of Physics: Conference Series* 1525, 1 (4 2020), 012080. https://doi.org/10.1088/1742-6596/1525/1/012080

[23] Daniel Kusswurm. 2014. Advanced Vector Extensions (AVX). In *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX.* Apress, Berkeley, CA, 327–349. https://doi.org/10.1007/978-1-4842-0064-3_12

[24] Jonathan K. Lawder. 2000. *The application of space-filling curves to the storage and retrieval of multi-dimensional data.* Ph. D. Dissertation. University of London.

[25] Jonathan K. Lawder and Peter J. H. King. 2000. Using Space-Filling Curves for Multi-dimensional Indexing. In *Advances in Databases*, Brian Lings and Keith Jeffery (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35. https://doi.org/10.1007/3-540-45033-5_3

[26] Ki Myung Brian Lee, Chanyeol Yoo, Ben Hollings, Stuart Anstee, Shoudong Huang, and Robert Fitch. 2019. Online Estimation of Ocean Current from Sparse GPS Data for Underwater Vehicles. In *2019 International Conference on Robotics and Automation (ICRA)* (Montreal, QC, Canada). IEEE, New York, NY, USA, 3443–3449. https://doi.org/10.1109/ICRA.2019.8794308

[27] K. Patrick Lorton and David S. Wise. 2007. Analyzing Block Locality in Morton-Order and Morton-Hybrid Matrices. *SIGARCH Comput. Archit. News* 35, 4 (9 2007), 6–12. https://doi.org/10.1145/1327312.1327315

[28] Bartosz Milewski. 2019. Category theory for programmers.

[29] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel Saltz. 2001. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering* 13, 1 (2001), 124–141. https://doi.org/10.1109/69.908985

[30] Anthony E. Nocentino and Philip J. Rhodes. 2010. Optimizing Memory Access on GPUs Using Morton Order Indexing. In *Proceedings of the 48th Annual Southeast Regional Conference* (Oxford, Mississippi) *(ACM SE '10)*. Association for Computing Machinery, New York, NY, USA, Article 18, 4 pages. https://doi.org/10.1145/1900008.1900035

[31] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture.* NVIDIA Corporation. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[32] NVIDIA Corporation. 2021. *CUDA C++ Programming Guide.* NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[33] Takayuki Okimura, Teruyoshi Sasayama, Norio Takahashi, and Soichiro Ikuno. 2013. Parallelization of Finite Element Analysis of Nonlinear Magnetic Fields

Using GPU. *IEEE Transactions on Magnetics* 49, 5 (2013), 1557–1560. https://doi.org/10.1109/TMAG.2013.2244062

[34] Steffen Raach, David Schlipf, Florian Haizmann, and Po Wen Cheng. 2014. Three Dimensional Dynamic Model Based Wind Field Reconstruction from Lidar Data. *Journal of Physics: Conference Series* 524 (6 2014), 012005. https://doi.org/10.1088/1742-6596/524/1/012005

[35] Carl Runge. 1895. Über die numerische Auflösung von Differentialgleichungen. *Math. Ann.* 46, 2 (1895), 167–178. https://doi.org/10.1007/BF01446807

[36] Sunita Sarawagi and Michael Stonebraker. 1994. Efficient organization of large multidimensional arrays. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering* (Houston, TX, USA). IEEE, New York, NY, USA, 328–336. https://doi.org/10.1109/ICDE.1994.283048

[37] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40, 2 (2020), 45–52. https://doi.org/10.1109/MM.2020.2974217

[38] Stephen Nicholas Swatman. 2022. Covfie: A Compositional Vector Field Library. https://github.com/acts-project/covfie.

[39] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly. 2006. Is Morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience* 18, 11 (2006), 1509–1539. https://doi.org/10.1002/cpe.1018

[40] Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David Kaeli. 2015. NUPAR: A Benchmark Suite for Modern GPU Architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2668930.2688046