The Continuous Tensor Abstraction: Where Indices Are Real

JAEYEON WON, Massachusetts Institute of Technology, USA
WILLOW AHRENS, Georgia Institute of Technology, USA
TEODORO FIELDS COLLIN, Massachusetts Institute of Technology, USA
JOEL S. EMER, Massachusetts Institute of Technology, USA and NVIDIA, USA
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA

This paper introduces the continuous tensor abstraction, allowing indices to take real-number values (e.g., `A[3.14]`). It also presents continuous tensor algebra expressions, such as $C_{x,y} = A_{x,y} * B_{x,y}$, where indices are defined over a continuous domain. This work expands the traditional tensor model to include continuous tensors. Our implementation supports piecewise-constant tensors, on which infinite domains can be processed in finite time. We also introduce a new tensor format for efficient storage and a code generation technique for automatic kernel generation. For the first time, our abstraction expresses domains like computational geometry and computer graphics in the language of tensor programming. Our approach demonstrates competitive or better performance to hand-optimized kernels in leading libraries across diverse applications. Compared to hand-implemented libraries on a CPU, our compiler-based implementation achieves an average speedup of 9.20× on 2D radius search with ~60× fewer lines of code (LoC), 1.22× on genomic interval overlapping queries (with ~18× LoC saving), and 1.69× on trilinear interpolation in Neural Radiance Field (with ~6× LoC saving).

CCS Concepts: • **Software and its engineering → Compilers**; **Domain specific languages**; • **Mathematics of computing** → *Mathematical software.*

Additional Key Words and Phrases: Sparse Tensor Compiler, Domain Specific Language

ACM Reference Format:
Jaeyeon Won, Willow Ahrens, Teodoro Fields Collin, Joel S. Emer, and Saman Amarasinghe. 2025. The Continuous Tensor Abstraction: Where Indices Are Real. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 368 (October 2025), 29 pages. https://doi.org/10.1145/3763146

1 Introduction

Array programming has been a cornerstone in the history of computing, with its origin dating back to FORTRAN's introduction in 1957 [6]. It remains integral to imperative languages like C/C++ [67], Java [5], Julia [11], and Python [62], as well as specialized array-focused languages and frameworks including APL [41], MATLAB [40], TensorFlow [1], and PyTorch [56]. The array abstraction is simple, consistent across languages, and widely familiar from introductory programming courses. Many domains rely on multidimensional arrays and share a common set of tools built around it.

Traditional dense tensor programming[1] is guided by the fundamental requirement that all indices are integers. While traditional tensor programming relies on integer indices, it is insufficient for

[1]Currently, "tensor programming" is often used interchangeably with "array programming," where "tensor" denotes a multidimensional array. For this paper, we adopt the term "tensor programming."

Authors' Contact Information: Jaeyeon Won, Massachusetts Institute of Technology, Cambridge, USA, jaeyeon@csail.mit.edu; Willow Ahrens, Georgia Institute of Technology, Atlanta, USA, ahrens@gatech.edu; Teodoro Fields Collin, Massachusetts Institute of Technology, Cambridge, USA, Teoc@csail.mit.edu; Joel S. Emer, Massachusetts Institute of Technology, Cambridge, USA and NVIDIA, Westford, USA, emer@csail.mit.edu; Saman Amarasinghe, Massachusetts Institute of Technology, Cambridge, USA, saman@csail.mit.edu.

ACM 2475-1421/2025/10-ART368
https://doi.org/10.1145/3763146

many real-world data, which often involves real number coordinates—common in domains such as 3D deep learning [17, 39], computer graphics [29, 52], and spatial databases [30, 31]. Unfortunately, without a common abstraction, support for such data has fragmented across domains, with each domain relying on its own tools. We believe that continuous domain programs can be unified under the umbrella of tensor programming. A simple, uniform abstraction would lower the learning curve for programmers and amortize the development cost across a wide range of applications.

In this work, we challenge three assumptions that restrict tensor programming to integer coordinates and show it can support real coordinates. While recent works have implicitly questioned the first two assumptions, our work challenges all three. To our knowledge, we are the first to explicitly address the third assumption, leading to our main contribution.

Conventional Assumption 1: Computations are performed at all points in the *iteration space*—the set of all index tuples within the tensor's shape. This includes both *effectual computations* (those contributing to the final output) and *ineffectual computations* (those that do not affect the output, e.g., `out+=in*0`). Like other sparse compilers [3, 12, 45], we break this assumption by computing only at effectual points and skipping ineffectual points without sacrificing correctness.

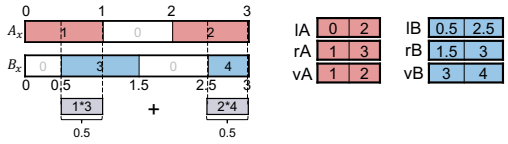
Conventional Assumption 2: The iteration space is finite because every point in the iteration space must project into coordinates in the tensors. Traditionally, computations at out-of-bounds indices were considered invalid or caused errors. By conceptualizing tensors as implicitly holding zeros at out-of-bound indices, we can extend the iteration space—even to infinity—simplifying programming models and avoiding out-of-bounds errors. This is true because computations at these indices are ineffectual and can be skipped. Some prior tensor compilers, such as Halide [59] and Finch [3], implicitly break this assumption. This work also relies on breaking this assumption.

Conventional Assumption 3: The iteration space is an integer lattice. Traditionally, the points in an N-dimensional iteration space are N-tuples of discrete integer coordinates. As an iteration space can be infinite, the domain of the space do not have to be integers either. Thus, we could have real-number coordinates. This allows us to envision the iteration space as continuous rather than discrete, enabling computations at real-valued indices. We generalize computations to operate over real domains, focusing on effectual computations that occur at real-number indices.

In this paper, we extend the tensor programming model by expanding coordinate points from the discrete space of integers to the continuous space of real numbers. Users can write tensor algebra expressions such as $C_x = A_x + B_x$ and $C_{x,y} = A_{x,z} * B_{z,y}$, using real-numbered indices and iterating over a continuous domain. By leveraging a piecewise-constant assumption, we propose implementation methods for storing continuous tensors in memory, performing reduction operations over continuous iteration spaces, and generating efficient code for continuous tensor programs. By expressing applications using the tensor programming model that previously required domain-specific codes, we bring the simplicity of a universal abstraction to those domains.

To our knowledge, this paper is the first to **extend tensor programming to real-numbered indices with continuous iteration space.** This paper includes the following contributions:

- We extend the tensor programming model to iterate over a continuous domain under a piecewise-constant assumption.
- We introduce reduction operations specifically designed for the continuous iteration space.
- We introduce an efficient code generation mechanism for computations on continuous tensors by extending the fibertree abstraction [68] and Looplets [2, 3].
- We unify a diverse range of applications across various fields using the continuous tensor abstraction, including bioinformatics, geospatial applications, point cloud processing, and Neural Radiance Fields (NeRF). Writing applications in the continuous tensor abstraction is straightforward and intuitive, requiring ~18× fewer lines of code in bioinformatics, ~62×



(a) Visualization of interval dot product along with the memory representation of the coordinates (lA, rA, lB, rB) and values (vA, vB).

```
# Format Spec of Continuous Tensors
A = Interval(Element(vA),lA,rA)
B = Interval(Element(vB),rA,rB)
# Continuous Einsum
@einsum val = A[x] * B[x] * d(x,μλ)
```

(b) Interval dot product in continuous tensor abstraction, where $d(x, \mu_\lambda)$ denotes integration over x with Lebesgue measure μ_λ.

```
field#1(1)[] A, B;
strand dotprod(){
real t = 0;
real tmax = 3;
real step = 0.01;
real val = 0;
update {
  val += A(t) * B(t) * step;
  t += step;
  if (t > tmax){stabilize;}}}
```

(c) Interval dot product *approximately* written in Sci-Vis language Diderot [44]

```
SELECT
 SUM(
  ST_Length(
   ST_Intersection(A.geom,B.geom)
  )
  * A.value * B.value
 )
FROM A
JOIN B
  ON ST_Intersects(A.geom,B.geom)
GROUP BY A.id
```

(d) Interval dot product written in spatial SQL.

```
pA, pB = 0, 0
while (pA<2 and pB<2):
 lA,rA,vA = A[pA]
 lB,rB,vB = B[pB]
 lAB = max(lA,lB)
 rAB = min(rA,rB)
 if lAB <= rAB:
  sum += (rAB-lAB)*vA*vB
 pA += (rA==rAB)
 pB += (rB==rAB)
```

(e) Interval dot product written in array programming with integer indices.

Fig. 1. Illustration of an interval dot product written in four different languages, highlighting how our continuous tensor abstraction provides a concise and straightforward implementation compared to others.

fewer lines of code in geospatial queries, ~101× fewer lines of code in 3D point cloud convolution, and ~6× fewer lines of code in trilinear interpolation in NeRF.

- Compared to hand-implemented libraries, our compiler-based implementation achieves an average speedup of 9.20× on radius search queries, 1.22× on genomic interval overlapping queries, and 1.69× on trilinear interpolation in NeRF.

2 Motivation

Programming over continuous domains is challenging in current tools. In contrast, our abstraction is simple, intuitive, and concise, while still producing code that matches or outperforms existing approaches. This advantage is especially clear in applications, such as scientific visualization [18, 44] and spatial deep learning [29, 39, 52], which require both (1) geometric operations and (2) numerical computation on geometry-tied values, a combination rarely supported by current systems.

Figure 1 shows the interval dot product, which sums the product of values over the lengths of overlapping intervals, requiring both intersection computation and numerical integration. This operation is useful in many areas, such as bioinformatics. For example, when comparing two genomic intervals A and B, it calculates how much of gene A's position overlaps with gene B's position on the chromosome. More operations on genomic intervals are discussed in Section 7.2. Existing models—scientific visualization languages [18, 44], spatial databases [30, 31], and tensor frameworks [36, 67]—typically focus only one of two aspects, making such tasks difficult to express.

SciVis: Languages like Diderot [44] support pointwise evaluations on continuous fields, but lack mechanisms for expressing geometric operations like intersections. As shown in Figure 1c, Diderot allows querying field values at specific points on real domain, but cannot compute field intersections directly, requiring users to approximate the interval dot product via Riemann integration.

$$
\begin{aligned}
\langle einsum \rangle &::= \langle stmt \rangle \\
&\quad | \ \langle stmt \rangle * d(\langle rIndex \rangle ...; \langle measure \rangle ...) \\
\langle stmt \rangle &::= \langle tensorName \rangle [\langle index \rangle \dots] = \langle expr \rangle \\
\langle expr \rangle &::= \langle val \rangle \\
&\quad | \ \langle affineIndex \rangle \\
&\quad | \ \langle tensorName \rangle [\langle affineIndex \rangle \dots] \\
&\quad | \ (\langle expr \rangle) \\
&\quad | \ \langle expr \rangle \ \langle op \rangle \ \langle expr \rangle
\end{aligned}
\qquad
\begin{aligned}
\langle affineIndex \rangle &::= \langle affineTerm \rangle \\
&\quad | \ \langle affineIndex \rangle + \langle affineTerm \rangle \\
\langle affineTerm \rangle &::= \langle val \rangle \mid \langle index \rangle \mid \langle val \rangle * \langle index \rangle \\
\langle tensorName \rangle &::= \langle identifier \rangle \\
\langle index \rangle &::= \langle identifier \rangle \\
\langle rIndex \rangle &::= \langle identifier \rangle \\
\langle op \rangle &::= + \mid - \mid * \mid \text{AND} \mid \text{OR} \dots \\
\langle measure \rangle &::= \mu_{\wedge\vee} \mid \mu_{\#} \mid \mu_{\lambda} \mid \dots \\
\langle val \rangle &::= \langle \text{Real Number} \rangle \\
\langle identifier \rangle &::= [a\text{–}zA\text{–}Z]
\end{aligned}
$$

Fig. 2. Grammar of our continuous Einsum.

Spatial Database: Spatial databases [24, 30] support geometric queries and efficient spatial structures like R-trees [47], but are limited in expressing numerical computations. As shown in Figure 1d, spatial SQL easily handles intersections and lengths, but becomes cumbersome for complex computations on multidimensional tensors.

Tensor Programming: Tensor frameworks excel at high-performance numerical computation but assume dense, integer indexing, making them poorly suited for continuous domains. Supporting real-valued coordinates requires custom data structures (e.g., COO formats, interval trees [4, 19, 27]) and significant engineering to implement geometric operations like intersection. As shown in Figure 1e, low-level implementations can perform efficient intersection and weighted sums but must be re-written for each new computation.

To bridge this gap, we propose a continuous tensor abstraction that unifies the strengths of existing models. Tensors are defined over real coordinates and have nonzero values on geometrically meaningful regions (e.g., boxes, lines). We generalize tensor indexing and extend Einsums [25, 36, 53] to continuous domains, enabling concise, expressive geometric and numerical computation.

Continuous Tensor Abstraction: As shown in Figure 1b, our abstraction allows users to define data structures using a *level format abstraction* [19, 68] and write computations using an *extended Einsum notation* with real-valued indices. The resulting code is compiled into efficient low-level implementations, equivalent to those shown in Figure 1e. Users simply write a familiar Einsum expression and format specification, yielding a unified programming model for continuous domain that combines both geometric and numerical computation.

3 Continuous Tensor Expression Language

3.1 Background

Tensor compilers [16, 36, 45, 53, 56, 59] provide a syntax similar to Einsums (Einstein summation notation [25]), a concise way to describe operations like matrix multiply ($C_{x,y} = \sum_r A_{x,r} * B_{r,y}$).

An Einsum can be understood as a traversal over all coordinates of the index variables. During this traversal, the specified tensor expression is evaluated at each point and the result is stored in the output. The reduction operator is used when the same output coordinate is referenced more than once. In other words, reductions occur along index variables that do not appear in the output. This typically implies summation over those dimensions, removing the need for explicit summation symbols, as in matrix multiplication ($C_{x,y} = A_{x,r} * B_{r,y}$). Reductions, however, are not limited to summation; they can also involve operations like min, max, and other aggregations.

Einsum

$$[\![\texttt{tensorName[index}_1,...,\texttt{index}_n\texttt{]}]\!]^{\text{Env}} = (x \to [\![\texttt{expr}]\!]^{\text{Env}\cup\{\texttt{index}_1,...,\texttt{index}_n \mapsto x\}} : x \in \mathbb{R}^n)$$

$$[\![\texttt{tensorName[index}_1,...,\texttt{index}_n\texttt{]} = \texttt{expr} * d(\texttt{rIndex}_1,...,\texttt{rIndex}_k;\ \texttt{measure}_1,...,\texttt{measure}_k)]\!]^{\text{Env}} =$$
$$\text{Let } f(x) : \mathbb{R}^{n+k} \to [\![\texttt{expr}]\!]^{\text{Env}\cup\{\texttt{index}_1,...,\texttt{index}_n,\texttt{rIndex}_1,...,\texttt{rIndex}_k \mapsto x\}}$$
$$\text{in Let } g(x) : \mathbb{R}^n \to \int^{\oplus}_{\texttt{rIndex}_{1...k}} f(\texttt{index}_{1,...,n}, \texttt{rIndex}_{1,...,k})\, d\mu_1(\texttt{rIndex}_1)...d\mu_k(\texttt{rIndex}_k) \text{ in } g$$

Expression

$$[\![\texttt{val}]\!]^{\text{Env}} = \texttt{val}$$

$$[\![\texttt{tensorName[affineIndex}\ldots\texttt{]}]\!]^{\text{Env}} = \text{Env}[\texttt{tensorName}]([\![\texttt{affineIndex}]\!]^{\text{Env}}\ldots)$$

$$[\![\texttt{(expr)}]\!]^{\text{Env}} = [\![\texttt{expr}]\!]^{\text{Env}}$$

$$[\![\texttt{expr}_1\ \texttt{op}\ \texttt{expr}_2]\!]^{\text{Env}} = [\![\texttt{op}]\!]\left([\![\texttt{expr}_1]\!]^{\text{Env}}, [\![\texttt{expr}_2]\!]^{\text{Env}}\right)$$

Affine Index

$$[\![\texttt{index}]\!]^{\text{Env}} = \text{Env}[\texttt{index}]$$

$$[\![\texttt{val} * \texttt{index}]\!]^{\text{Env}} = [\![\texttt{val}]\!]^{\text{Env}} * [\![\texttt{index}]\!]^{\text{Env}}$$

$$[\![\texttt{affineIndex}_1 + \texttt{affineIndex}_2]\!]^{\text{Env}} = [\![\texttt{affineIndex}_1]\!]^{\text{Env}} + [\![\texttt{affineIndex}_2]\!]^{\text{Env}}$$

Fig. 3. Denotational semantics of our language. We assume an environment Env providing: **(1)** a mapping from each tensor name to a function $\mathbb{R}^n \to \mathbb{S}$ (where $\mathbb{S}$ is a semiring domain), **(2)** a mapping from each index name to a real number, and **(3)** a mapping from each measure name to a corresponding semiring-valued measure μ_i. Each μ_i is referenced in the syntax as $\texttt{measure}_i$ and, when combined with $\texttt{rIndex}_i$, enables integrating over the specified reduction indices.

3.2 Syntax and Denotational Semantics

In Figure 2 and 3, we present the syntax and denotational semantics of our language. It closely resembles existing Einsum notation. One notable difference from existing Einsums is that the reduction over continuous domain may sometimes correspond to discrete operations (e.g., summation) and other times to continuous operations (e.g., integration). To handle this, the user must specify a reduction *measure* for each reduction index with `d(<rIndex>..., <measure>...)`.

Specifically, Einsum statements in our language can take one of the following forms:

$$C_{x_1,...,x_n} = P_{x_1,...,x_n} \quad \text{if there is no reduction}$$

$$C_{x_1,...,x_n} = P_{x_1,...,x_n,rx_1,...,rx_k} * d(rx_1,\ldots,rx_k;\mu_1,\ldots,\mu_k) \quad \text{otherwise}$$

Here, C is the output tensor, whose values are computed over the indices $x_1,\ldots,x_n,rx_1,\ldots,rx_k$ defined on a real domain. The indices $(rx_1,\ldots,rx_k) \in D$ represent the reduction indices (where D is a reduction domain). $P_{x_1,...,x_n}$ denotes a general expression that may involve one or more tensors and depends on the indices $x_1,\ldots,x_n$. We adopt subscript notation for consistency with standard tensor indexing; however, this notation does not imply that P is necessarily a single tensor. In fact, P may incorporate multiple tensor accesses and semiring operations [33, 64, 71]. In the denotational semantics presented in Figure 3, a statement involving reductions is interpreted as follows:

$$C_{x_1,...,x_n} = \int^{\oplus}_{rx_1,...,rx_k} P_{x_1,...,x_n,rx_1,...,rx_k}\, d\mu_1(rx_1)\ldots d\mu_k(rx_k)$$

The operator $\int^{\oplus}$ represents an integral using semiring addition $\oplus$, combining the values of P across the reduction indices. This reduction is weighted by the *semiring-valued measures* $\mu_1, \ldots, \mu_k$ for each reduction index $rx_1, \ldots, rx_k$, respectively.

Calculations are performed within a semiring $\mathbb{S} = (R, \oplus, \otimes, 0, 1)$, which is defined by an addition operation $\oplus$ (associative, commutative, with identity 0) and a multiplication operation $\otimes$ (associative, with identity 1, distributing over $\oplus$). Common examples include the real number semiring $(\mathbb{R}, +, \times, 0, 1)$ and the boolean semiring $(\{T, F\}, \vee, \wedge, F, T)$. To formally define the reduction process, a measure-theoretic approach [23, 61, 65] is used. Given a sigma-algebra over the reduction indices $V \subset \mathbb{P}(D)$ (a power set of D), a *semiring-valued measure* $\mu : V \to \mathbb{S}$ is defined on subsets of D, satisfying $\mu(\emptyset) = 0$ and disjoint additivity: $\mu(A \cup B) = \mu(A) \oplus \mu(B)$ for disjoint subsets A and B.

Using semiring-valued measures $\mu_1, \ldots, \mu_k$ for each reduction index $rx_1, \ldots, rx_k$, the reduction statement can be also rewritten as:

$$C_{x_1,\ldots,x_n} = \bigoplus_{m \in \mathbb{S}} \left[(\mu_1 \otimes \cdots \otimes \mu_k)(\{(rx_1, \ldots, rx_k) \in D : P_{x_1,\ldots,x_n,rx_1,\ldots,rx_k} = m\}) \right] \otimes m.$$

This expression defines each output $C_{x_1,\ldots,x_n}$ by summing over all possible values $m \in \mathbb{S}$ that P can take. Each value m is weighted by the product of semiring-valued measures associated with the reduction indices. Specifically, since $m \in \mathbb{S}$, it may represent any element in the semiring: for example, any real number in the real semiring, or true and false in the boolean semiring. The measure μ provides these weights, determining how much each m contributes to $C_{x_1,\ldots,x_n}$. Finally, the semiring addition $\oplus$ is used to combine these weighted contributions into a single value.

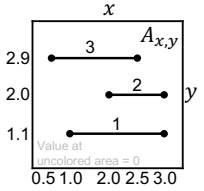
$\oplus$ and $\otimes$ could represent real addition and multiplication or the logical OR and AND. While this subsection focuses on the real number semiring and the Boolean semiring, the framework can accommodate additional semirings, allowing the language to support various reduction semantics.

1. Real number semiring $(R, +, *, 0, 1)$ *:* ***Lebesgue measure*** μ_λ ***and Counting measure*** $\mu_\#$. For addition ($\oplus = +$), users specify a measure μ over D, choosing either the Lebesgue measure μ_λ or the counting measure $\mu_\#$ [60]. The Lebesgue measure, suitable for continuous intervals, is defined as $\mu_\lambda([a, b]) = b - a$. The counting measure, suitable for discrete points, is defined as follows: for isolated pinpoints $[a, a]$, $\mu_\#([a, a]) = 1$; for intervals like $[a, b]$ with $a < b$, $\mu_\#([a, b]) = \infty$. Users can flexibly model both continuous and discrete accumulations by selecting the proper measure.

Given a function f_P representing P evaluated at $(x_1, \ldots, x_n, rx_1, \ldots, rx_k)$, we define:

$$C_{x_1,\ldots,x_n} = \int_{rx_1 \ldots rx_k} f_P(x_1, \ldots, x_n, rx_1, \ldots, rx_k)\, d\mu_1(rx_1) \ldots d\mu_k(rx_k)$$

In the right example, the Einsum $s = A_{x,y} * d(x, y; \mu_\lambda, \mu_\#)$ evaluates $s = \int_y \int_x f_A(x, y)\, d\mu_\lambda(x)\, d\mu_\#(y) = 1 * (3.0 - 1.0) + 2 * (3.0 - 2.0) + 3 * (2.5 - 0.5)$, where $f_A(x, y)$ is the function evaluating the tensor $A_{x,y}$. Another Einsum, $s = A_{x,y} * d(x, y; \mu_\lambda, \mu_\lambda)$, yields zero because the 2D xy area over a 1D x interval is zero. An easy analogy is that the measure μ_λ acts as an integral over intervals, while the measure $\mu_\#$ resembles a discrete sum over points.



2. Boolean semiring $(\{T, F\}, \vee, \wedge, F, T)$ *:* ***Logical measure*** $\mu_{\wedge\vee}$. For logical disjunction, we use a function $f_P : (x_1, \ldots, x_n, rx_1, \ldots, rx_k) \to \{\mathrm{F}, \mathrm{T}\}$. The logical reduction is defined by

$$C_{x_1,\ldots,x_n} = \bigvee_{rx_1 \ldots rx_k} f_P(x_1, \ldots, x_n, rx_1, \ldots, rx_k) \wedge \mu_{\wedge\vee}(rx_1) \wedge \ldots \mu_{\wedge\vee}(rx_k)$$

where $C_{x_1,\ldots,x_n} = T$ if $f_P(x_1, \ldots, x_n, rx_1, \ldots, rx_k) = T$ for any $rx_1 \ldots rx_k$; otherwise, $C_{x_1,\ldots,x_n} = F$. In this case, the boolean measure $\mu_{\wedge\vee}$ assigns T to any non-empty set, and F to the empty set.

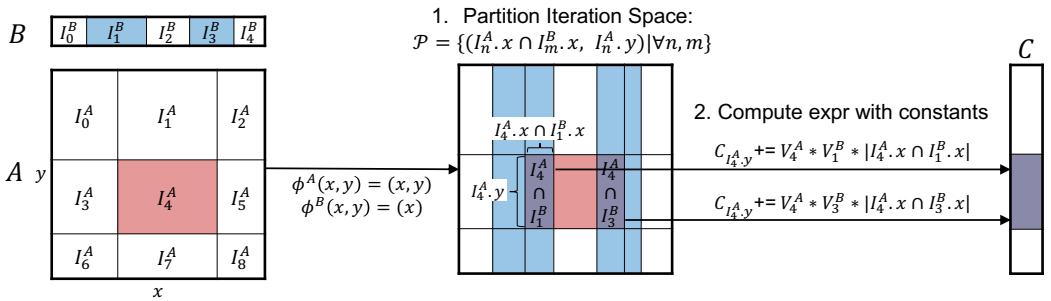


Fig. 4. Evaluation of $\boldsymbol{C_y = A_{x,y} * B_x * d(x, \mu_\lambda)}$, where $d(x, \mu_\lambda)$ indicates an lebesgue integral on dimension x. The computation iterates over all partitions formed by the intersection of the piecewise-constant intervals of $A_{x,y}$ and B_x, where non-colored regions represent zero values, and evaluates the expression within each partition. The two right arrows indicate computations involving non-zero elements of the tensors, highlighting only the interactions that contribute non-zero values to the result.

4 Piecewise-constant Specialization

Continuous tensor programs cannot reuse the operational definition from traditional Einsums [53], as traversing all real-valued coordinates is not computable. To make our approach practical, we restrict our focus for the remainder of this paper to **piecewise-constant** tensors, where tensor values remain constant over specific intervals. This restriction enables efficient computation while still supporting a wide range of useful applications. In this subsection, we present a new evaluation semantics for continuous tensor programs under the piecewise-constant assumption. We then describe the validity conditions and explain how to verify whether a given program satisfies them. Finally, we discuss extensions of our system to support piecewise-non-constant functions.

4.1 Piecewise-constant Tensor

We introduce the piecewise-constant restriction to create a practical and implementable solution. In the rest of the paper, we assume: ***All continuous tensors must be piecewise-constant.***

We define a **tensor** A as a concrete data structure in memory whose value at real-valued indices $x_1, \ldots, x_n$ can be obtained by evaluating $\mathtt{A}[x_1, \ldots, x_n]$. We can express its values as a mathematical function $f(x_1, \ldots, x_n) = \mathtt{A}[x_1, \ldots, x_n]$. A tensor is said to be **piecewise constant** when its values are constant over specific regions of its domain and can be expressed as $f(x_1, \ldots, x_n) = \sum_m V_m \times [\![(x_1, \ldots, x_n) \in I_m]\!]$, where: V_m is the constant value in the m-th piece; I_m is an N-dimensional interval (a hyperrectangle) representing the domain of the m-th piece; $[\![\,]\!]$ denotes the Iverson bracket, defined as $[\![cond]\!] = 1$ if the $cond$ is true, and 0 otherwise.

In this definition, the N-dimensional interval I_m must satisfy the following rules:

1. **Disjoint Intervals**: The intervals are pairwise disjoint, $\forall\, m_1 \neq m_2,\ I_{m_1} \cap I_{m_2} = \emptyset$.
2. **Complete Coverage**: The union of all intervals covers the entire domain, $\bigcup_m I_m = \mathbb{R}^N$.
3. **Axis-Aligned Intervals**: Each interval I_m is axis-aligned, meaning it is a hyperrectangle aligned with the coordinate axes (i.e., no diagonal or curved boundaries).

4.2 Piecewise-constant Evaluation Semantics

In tensor computations involving piecewise constant tensors, we first partition the iteration space into regions where each tensor maintains a constant value. Rather than traversing all real-numbered coordinates in the entire space, we traverse each partition and evaluate the tensor expression using the constant values for that partition. We formalize an evaluation semantics as follows :

1. Partitioning the Iteration Space: The goal is to partition the iteration space into regions

where all tensors used in the expression remain constant. To achieve this, we first map piecewise-constant intervals of each tensor into the iteration space using indexing functions. An indexing function explains how each index is used to access the tensor. By applying the inverse of this indexing function, each tensor's constant interval is converted into a corresponding region in the iteration space (illustrated as Step 1a in Figure 5). Next, the iteration space is partitioned by intersecting these regions from all tensors. Each resulting intersection indicates a unique region in which every tensor in the expression remains simultaneously constant. (illustrated as Step 1b)

For instance, in Figure 4, the tensor expression $C_y = A_{x,y} * B_x * d(x, \mu_\lambda)$ is partitioned as follows. The indexing functions are $\phi^A(x,y) = (x,y)$ and $\phi^B(x,y) = (x)$, with their respective pre-images:

$$\phi^{A^{-1}}(I^A) = \{(x,y) \in \mathcal{S} \mid (x,y) \in I^A\}, \quad \phi^{B^{-1}}(I^B) = \{(x,y) \in \mathcal{S} \mid x \in I^B\},$$

where I^A and I^B are intervals of tensor A and B, and $\mathcal{S}$ denotes the iteration space of indices (x,y). Note that the pre-image of an interval in tensor B spans the entire region along the y-axis. The partitioned iteration space $\mathcal{P}$ is then defined as an intersection of the pre-image of:

$$\mathcal{P} = \left\{ \phi^{A^{-1}}(I_n^A) \cap \phi^{B^{-1}}(I_m^B) \,\middle|\, \forall n, m \right\},$$

where n and m range over all intervals of A and B. In each partition, both A and B are constant.

2. Compute Expression with Constants: After partitioning, each disjoint partition contains constant values for all tensors involved in the expression. We traverse each partition and evaluate the expression using these constants. When the expression involves a reduction operation, we aggregate the computation results from each partition into the corresponding output region. If there is a reduction with Lebesgue measure over an axis, we multiply the result by the length of the partition along that axis to account for the integration over that interval. Step 2 in Figure 5 illustrates this process. Note that the partition is an n-D axis-aligned rectangle $\texttt{partition} = I_1 \times I_2 \times \cdots \times I_n$, where n is the number of indices in the expression. Let $R \subseteq \{1, \ldots, n\}$ be the set of reduced axes, each with a semiring-valued measure μ_i : Intervals $\longrightarrow \mathbb{S}$. In lines 25-26, the reduction contributes

$$\texttt{out\_constant} \otimes \mu\left(\prod_{i \in R} I_i\right) = \texttt{out\_constant} \otimes \prod_{i \in R} \mu_i(I_i),$$

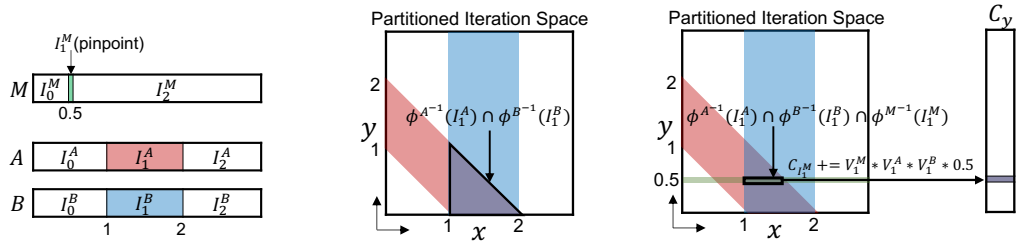
```
# Step 1a: Map tensor's constant intervals into
# iteration space with inverse of indexing func
inverse_pieces = {}
for access in einsum:
  tensor = access.tensor
  indexingfunc = access.indexing
  preimage = inverse(indexingfunc)
  inverse_pieces[tensor] = [
      {'interval': preimage(piece.interval),
       'constant': piece.constant}
      for piece in tensor
  ]
```

(a) Step 1a

```
# Step 1b: Partition iteration space into regions
#          where tensors remain constant
for piece1 in inverse_pieces[tensor1]:
  ...
  for pieceN in inverse_pieces[tensorN]:
    intervals = [piece1.interval, ...,
                 pieceN.interval]
    partition = intersect(intervals)
    if partition.is_empty(): continue
    # Step 2: Evaluate expression using constants
    weight = 1
    if reduction exists: # if d(x,μ) exists
      for (axis, measure) in einsum.d_exprs:
        weight ⊗= measure(partition.axis)
    out_constant = weight ⊗ compute_expr(
      piece1.constant, ..., pieceN.constant)
    out_interval = out.indexing(partition)
    out[out_interval] ⊕= out_constant
```

(b) Steps 1b & 2

Fig. 5. Pseudocode of the piecewise-constant evaluation semantics of our language.



(a) 1D Continuous Tensors

(b) Invalid Program: $C_y = A_{y+x} * B_x * d(x, \mu_\lambda)$

(c) Valid Program: $C_y = M_y * A_{y+x} * B_x * d(x, \mu_\lambda)$

Fig. 6. Illustrations of partitioned iteration spaces and program validity. (a) shows 1D continuous tensors where non-colored regions have values of zero. (b) demonstrates an invalid program because the intersected partition is a triangular region, leading to a non-piecewise-constant output. (c) shows a valid program where the pinpoint I_1^M in tensor M adjusts the partition to form axis-aligned intervals.

which follows from the product measure for axis-aligned rectangles: by Fubini's and Tonelli's theorems [60], the measure over the product equals the product of the individual measures.

To support an arbitrary semiring $(R, \oplus, \otimes, 0, 1)$ with arbitrary measures, users only need to implement lines 23-30 according to their chosen semiring and measure. To support a custom semiring, users initialize the output with the additive identity $0 \in R$, the measure weight with the multiplicative identity $1 \in R$, and define the implementations of $\oplus$ (addition) and $\otimes$ (multiplication). Users may also provide their own measure function $\mu : Interval \to R$ over intervals. For example, the *Lebesgue measure* in the real-number semiring is defined as $\mu_\lambda([a, b]) = b - a$. As a concrete example, consider the *min-product* semiring $(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, *, \infty, 1)$, where $\oplus = \min$ and $\otimes = *$. To use this semiring, one would initialize the output to ∞, and may implement an idempotent measure such as $\mu_{idem}([a, b]) = 1$, assigning uniform weight to all partitions regardless of size.

In Figure 4, we illustrate this process by traversing all partitions. For illustrative purposes, we only depict effectual computations involving non-zeros. For each partition, we perform the following operation with constants: $C_{I_y^{\text{Partition}}} \mathrel{+}= V_n^A \times V_m^B \times \text{length}(I_x^{\text{Partition}})$.

4.3 Program Validity

Not all programs are valid under the piecewise-constant assumption. To ensure correctness, the output must also be piecewise-constant—that is, the program must be *closed under piecewise-constant assumption*. We establish the following conditions to guarantee this closure after partitioning:

(1) **Each *effectual partition* must be an axis-aligned hyperrectangular interval.**
(2) **Expressions within each effectual partition must remain constant, except when the partition reduces to a pinpoint.**

An *effectual partition* refers to a partition where effectual computations—those that contribute to the final output, such as multiplying non-zero constants—take place. A *pinpoint* refers to an N-D interval that collapses to a single point, meaning its end points are identical along all dimensions.

For the first criterion, the definition of a piecewise constant tensor requires partitions to maintain an axis-aligned shape to ensure the output remains piecewise constant. As illustrated in Figure 6b, a standard convolution operation on intervals creates triangular partitions, resulting in a non-constant output. As shown in Figure 6c, incorporating a pinpoint tensor M allows for the adjustment of effectual partitions into axis-aligned shapes, ensuring validity. This masking guarantees that the output tensor retains its piecewise constant property. Figure 7 presents an algorithm for validity checking, commented with the example in Figure 6c, using an SMT solver. This algorithm

```
# Step 1: Construct a Symbolic Partition
# For example, in the expression
# C[y] = M[y] * A[x + y] * B[x],
# where M has pinpoints / A and B has intervals,
# the partition over (x, y) satisfies :
# partition = [
#    y == p1,           # pinpoint M[y]
#    p2 <= x + y <= p3, # interval A[x+y]
#    p4 <= x <= p5      # interval B[x]
# ]
partition = []
indices = [x, y, z, ...]   # Index symbols
for access in einsum:
  for tensordim in access:
    indexExpr = tensordim.indexing(indices)
    if tensordim.pinpoint:
      partition.add(
        indexExpr == FreshRealSymbol()
      )
    elif tensordim.interval:
      partition.add(
        FreshRealSymbol() <= indexExpr,
        indexExpr <= FreshRealSymbol()
      )
```

(a) Step 1 defines the shape of the partition by formulating linear inequalities and equalities of indices based on tensor accesses.

```
# Step 2: Construct a Symbolic Hyperrectangle
# In the example:
# hyperrectangle = [
#    p6 <= x <= p7,
#    p8 <= y <= p9,
# ]
hyperrectangle = []
for index in indices:
  hyperrectangle.add(
    FreshRealSymbol() <= index,
    index <= FreshRealSymbol()
  )

# Step 3: Check whether the partition
#         always forms a hyperrectangle.
#
# In the example:
# Z3.check(ForAll([p1, p2, p3, p4, p5],
#          Exists([p6, p7, p8, p9],
#          ForAll([x, y], partition == hyperrectangle))))
Check1 = Exists(hyperrectangle.RealSymbols,
         ForAll(indices, partition == hyperrectangle))
Check2 = ForAll(partition.RealSymbols, Check1)
Z3.check(Check2)
```

(b) Step 2 defines the hyperrectangular region over indices, and Step 3 checks whether the partition always forms a hyperrectangular shape.

Fig. 7. An algorithm for verifying hyperrectangular partitions based on pinpoint/interval specifications.

determines whether effectual partitions form hyperrectangles in a given program with information about whether a tensor dimension has nonzero pinpoints or intervals.

For the second criterion, expressions within each partition must remain constant to ensure that the computation remains constant throughout the partition. For example, the program $C_x = A_x * (x+1)$ with a piecewise-constant vector A is invalid because the term $(x+1)$ varies within any partition unless all effectual partitions are pinpoints. This check is only required for index expressions outside of tensor access. Such expressions are allowed only when the partition is guaranteed to be a pinpoint. To verify this, we use the same code structure as the hyperrectangularity check, but replace the hyperrectangle shape with a pinpoint to determine whether the partition is pinpoint.

4.4 Arbitrary Piecewise Functions Beyond Constants

Our piecewise-constant abstraction sits at one extreme of a spectrum of richer piecewise function classes—polynomials, rationals, splines, and beyond. To extend our compiler to any such piecewise-X class, we reuse the same partitioning logic from piecewise-constant evaluation semantics, but each tensor piece carries a symbolic description (e.g., polynomial coefficients) rather than a single scalar. In the constant case, the reduction over a product measure admits the shortcut $c \otimes \prod_i \mu_i(I_i)$ but with non-constant integrands the compiler must instead evaluate the full nested integral: $\int_{I_1} \cdots \int_{I_n} g \; d\mu_n \cdots d\mu_1$ using closed-form anti-derivatives when available. Crucially, any chosen function class must be closed under every operations in piecewise evaluation semantics:

(1) **Algebraic operations.** Adding or multiplying two pieces must stay within the class.
(2) **Integration.** Integrating over any subinterval must either admit a closed-form solution or be handled by a well-defined approximation strategy.

Polynomials satisfy all properties exactly. Most other classes fail at least one step—for example, rational integrals introduce logarithms, transcendental functions often lack closed forms, and fixed-degree splines increase in degree under multiplication. In those cases, the compiler must

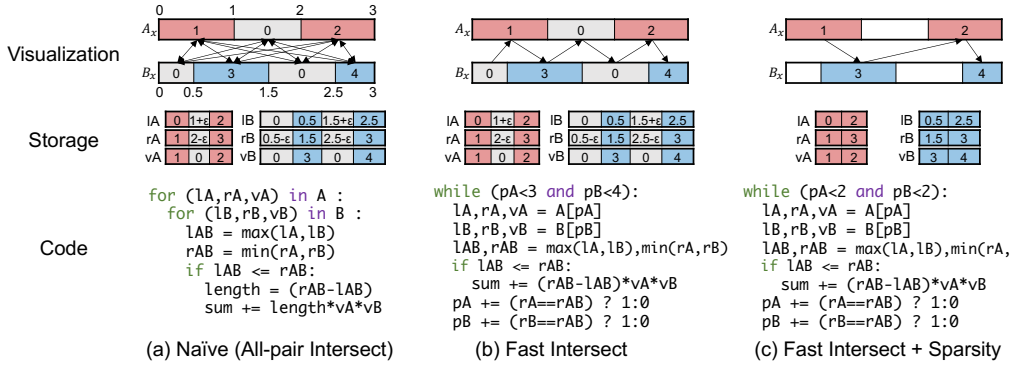


Fig. 8. Three strategies for partitioning the iteration space for $\boldsymbol{sum = A_x * B_x * d(x, \mu_\lambda)}$. lT, rT, and vT represent the left endpoints, right endpoints, and values of the interval of the tensor T, respectively. **(a) Naïve approach**: Computes all intersections using exhaustive pairwise comparisons. **(b) Fast Intersect**: Speeds up intersection using a two-pointer technique on sorted arrays. **(c) Fast Intersect + Sparsity**: Further improves (b) by leveraging sparsity to skip ineffectual partitions. Our compiler generates this final code.

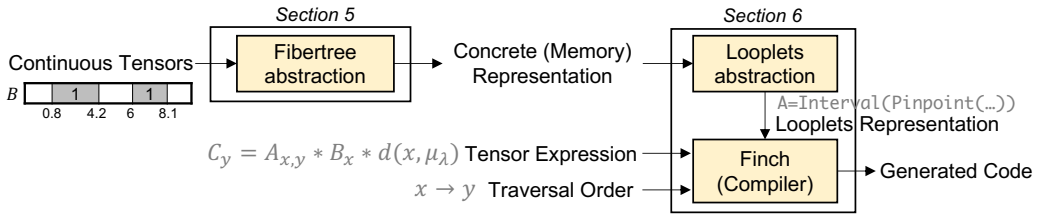


Fig. 9. Overview of our compiler implementation. The generated code efficiently partitions the iteration space into constant regions and computes the tensor expression within each partition.

detect the loss of closure, apply targeted approximation (e.g., numeric quadrature or projection back into the class), and propagate rigorous error bounds to guarantee overall accuracy. Supporting fully general piecewise-X functions requires careful design with incorporating a symbolic engine like SymPy [51], and is left to future work.

4.5 Implementation

While the piecewise-constant evaluation semantics describe partitioning the iteration space via all-pair intersections, we found that this process can be significantly optimized in practice. Figure 8 shows two key optimizations which make our generated code more efficient. First, when partitioning the iteration space, it is unnecessary to compute all pairwise intersections between every interval of each tensor. Second, we can skip computations for partitions where ineffectual operations occur, such as $a \times 0 = 0$, by only storing nonzero intervals. Section 6 details how we generate efficient code to partition the iteration space and exploit sparsity using the Looplets [3] abstraction.

5 Continuous Tensor Storage

Our compiler implementation partitions the iteration space into constant regions efficiently and computes expressions within each constant partition, leveraging the fibertree [68] and Looplets [3]. Figure 9 provides an overview of our compiler implementation. This section will explain how the piecewise-constant continuous tensor is stored in memory using a fibertree abstraction.

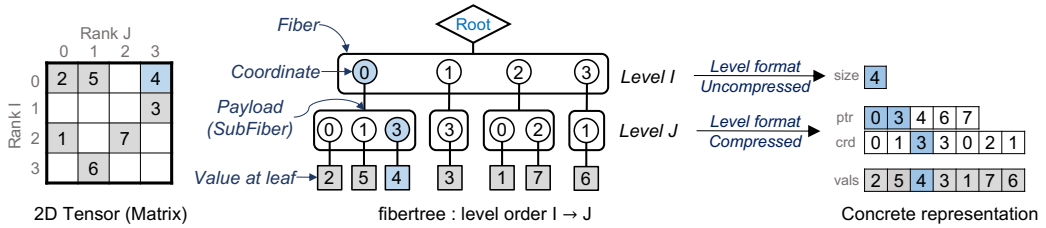


Fig. 10. A traditional sparse matrix, its fibertree abstraction and concrete representation stored in memory.

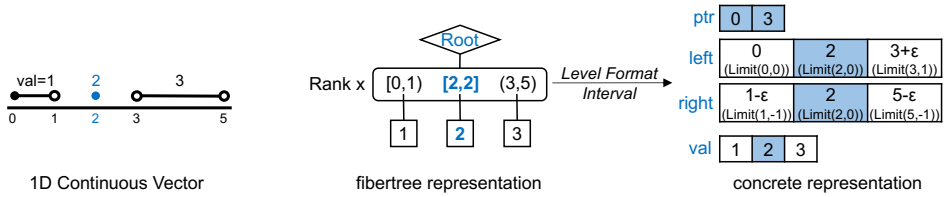


Fig. 11. A 1D continuous vector, its interval-typed fibertree abstraction and concrete representation.

5.1 Background on fibertrees

This subsection provides background information on a format abstraction that provides an approach to storing traditional dense/sparse tensors in memory. Some of these abstractions are grounded in the coordinate tree concept, which was initially introduced in the context of the format abstraction [19] within TACO and subsequently refined and formalized as the *fibertree* abstraction [68].

Figure 10 illustrates how the fibertree abstraction depicts a 2D matrix. A *tensor* is characterized as a multidimensional array with N *ranks* (dimensions). The fibertree abstraction envisions a tensor as a tree structure, where each *level* corresponds to a specific rank in the tensor. Each level comprises one or more fibers, representing sets of elements that share coordinates in the higher levels of the tree. Elements in a fiber are coordinate/*payload* pairs, with the payload taking the form of either a (sub)fiber at the next level or a value located at the leaf of the tree.

A *level format* defines the physical storage used to store the fibers at that level. The two most prevalent level formats for integer coordinates are the *Uncompressed* and *Compressed* level formats. The Uncompressed level format encodes a dense integer coordinates within the range of [0, N). In contrast, the Compressed level format exclusively encodes non-zero integer coordinates within the fiber by explicitly storing their coordinates with offset pointers. In Figure 10 (right), the concrete representation denotes that the matrix is stored in the $I \rightarrow J$ layout (row-major), with level formats assigned as Uncompressed for I and Compressed for J. This specific representation corresponds to the Compressed Sparse Row (CSR) format. For more details on fibertrees, refer to [53, 68].

5.2 Interval Coordinates in fibertrees

We introduce interval-typed coordinates in a fibertree to effectively capture piecewise-constant properties. Figure 11 illustrates an interval-typed fibertree for a continuous tensor `A`. *Pinpoint coordinates* are a special case, treated as intervals with both endpoints equal, collapsing to a single point; a pinpoint coordinate `2` is equivalent to the closed interval `[2,2]`. We account for the inclusiveness (open or closed) of each endpoint, resulting in four distinct interval subtypes. Similar to how traditional level formats operate on fibers with integer coordinates, we designed several level formats with interval coordinates, with the *Interval level format* being assigned for rank x in this example. Interval level format encodes the number and inclusiveness pair with `Limit` type.

Table 1. Four inclusiveness types of intervals. In our implementation, we treat these as a single closed interval representation with the use of the infinitesimal number ϵ stored in `Limit` type.

Name	Notation	Definition	Treated as	Implemented as
Closed interval	$[a, b]$	$\{x \in \mathbb{R} \mid a \leq x \leq b\}$	$[a, b]$	`[Limit(a,0), Limit(b,0)]`
Right half-open interval	$[a, b)$	$\{x \in \mathbb{R} \mid a \leq x < b\}$	$[a, b - \epsilon]$	`[Limit(a,0), Limit(b,-1)]`
Left half-open interval	$(a, b]$	$\{x \in \mathbb{R} \mid a < x \leq b\}$	$[a + \epsilon, b]$	`[Limit(a,+1), Limit(b,0)]`
Open interval	(a, b)	$\{x \in \mathbb{R} \mid a < x < b\}$	$[a + \epsilon, b - \epsilon]$	`[Limit(a,+1), Limit(b,-1)]`

```
# Definition of Limit.
struct Limit<T>
    val::T    # Numeric type T (e.g., Int, Float32) chosen by the user
    eps::Int8 # (+ϵ, 0, -ϵ) = (+1, 0, -1)
end
# Example Operations defined on Limit.
(+)(x::Limit, y::Limit)::Limit = Limit(x.val + y.val, min(max(x.eps + y.eps, -1), +1))
(-)(x::Limit, y::Limit)::Limit = Limit(x.val - y.val, min(max(x.eps - y.eps, -1), +1))
(<)(x::Limit, y::Limit)::Bool = x.val < y.val || (x.val == y.val && x.eps < y.eps)
```

Fig. 12. Implementation of `Limit` type. Infinitesimal number is represented by `eps` field stored in `Int8` type.

5.2.1 Interval Level Format with `Limit` Type. To represent the interval inclusiveness (open and closed endpoints), we introduce a new number type called `Limit` for encoding interval endpoints. The `Limit` type incorporates an infinitesimal number, denoted by ϵ, which is smaller than any positive real number but not zero. This approach allows us to treat all intervals as closed by default. We emulate open endpoints by adding or subtracting ϵ as needed, transforming open intervals into closed ones. Table 1 illustrates how this representation works for each inclusiveness category.

Figure 12 depicts the definition of `Limit` and arithmetic operations implemented using this numeric type. `Limit` serves as an augmented number type for real numbers, essentially a struct comprising a regular numerical value (`val::T`) and the infinitesimal value (`eps::Int8`).

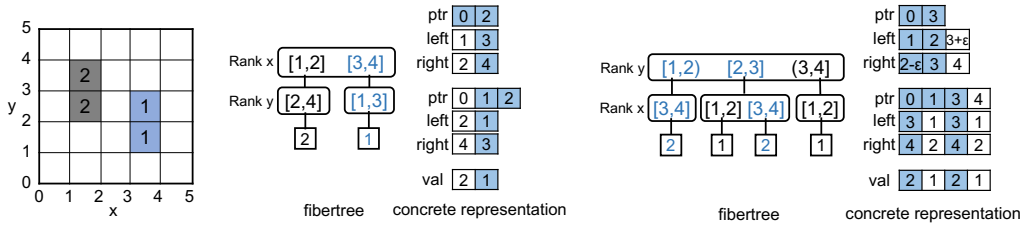
Although our compiler generates symbolic code that operates on real coordinates or intervals, actual computations are executed using finite-precision computer number types. This finite precision inevitably introduces representation and rounding errors, distinguishing practical computations from idealized continuous real-number arithmetic. The `Limit<T>` type supports various numeric types `T` (e.g., `Float32`, `Rational`, or `BigFloat` for arbitrary precision), allowing users to select the precision level that best fits their application's requirements. For consistent comparison, we matched the numeric type for real indices to each baseline in our case studies (Section 7).

5.3 Optimized Representation

This subsection outlines optimized representations for storing tensors with specific interval patterns. Choosing appropriate level order and level formats results in notable benefits in terms of memory efficiency and the complexity of the generated code.

5.3.1 Impact of Level Order. The careful selection of the level order is crucial for optimizing tensor operations in both memory consumption and performance [73]. As shown in Figure 13, the level order directly affects the structure of the fibertree and concrete representation. For instance, the order $x \rightarrow y$ and $y \rightarrow x$ results in different layouts. An optimal level order reduces redundancy in the storage format, minimizing memory usage by eliminating unnecessary index pointers. It also improves computation efficiency by reducing the overhead of irregular memory accesses.

5.3.2 Impact of Level Format. Figure 14 shows three tensors stored in various level formats. Depending on the pattern, certain tensors can benefit from a more optimized representation than storing every endpoint in the `Limit` type.



(a) Continuous tensor (b) Level order: $x \rightarrow y$ (c) Level order: $y \rightarrow x$

Fig. 13. Two concrete representations based on different level orders. With the level order $x \rightarrow y$ and $y \rightarrow x$, the alternative representation illustrates how memory usage can be reduced depending on the chosen order.

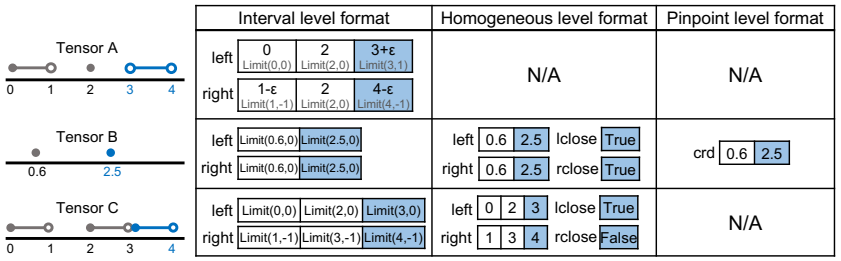


Fig. 14. Level formats across three distinct 1D continuous tensors. Instead of storing every pair of (number, inclusiveness) in the `Limit` type, certain tensors benefit from optimized level formats.

Homogeneous Level Format. The need to store pairs of (number, inclusiveness) for every endpoint may vary, depending on interval inclusiveness. When all intervals within a level share the same inclusiveness, we refer to them as *homogeneous* intervals. Conversely, when intervals have inconsistent inclusiveness, they are categorized as *heterogeneous*. Tensor B and C in Figure 14 are homogeneous but tensor A is heterogeneous. In the case of homogeneous intervals, there is no need to store every endpoint in the `Limit` type. Instead, the level format can store endpoints in regular numeric types while keeping inclusiveness details separate using `lclose` and `rclose`.

Pinpoint Level Format. If the level contains only pinpoint coordinates, there is no need to store both endpoints for each coordinate, as they share the same endpoints. Instead, pinpoints can be stored as single coordinates using the regular number type, as depicted in Tensor B in Figure 14.

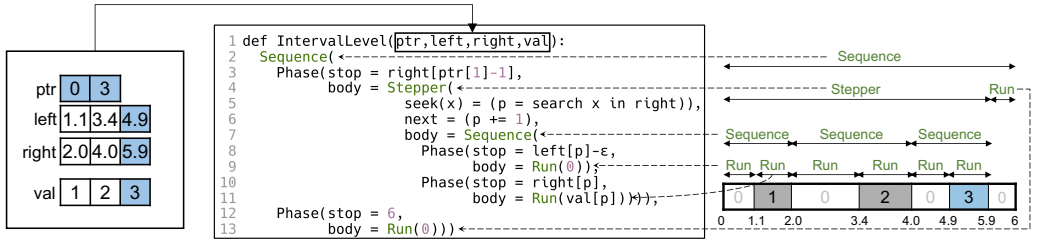
6 Code Generation

In this section, we describe how our compiler transforms continuous loops into efficient, executable code by extending Finch [2], which was originally designed for sparse tensor computations in the integer domain using Looplets [3]. We leverage Looplets to efficiently partition the iteration space, extending them into the continuous domain. At each step, repeated rewriting and optimization passes incorporate mathematical properties, such as sparsity, to enhance efficiency.

Our compiler takes two key inputs: (1) a continuous Einsum expression with index ordering, and (2) Looplet descriptions for each tensor. Using these inputs, the compiler generates executable Julia code. A key aspect of the compiler is its mechanism for lowering Looplets into loops.

6.1 Background on Looplets

While fibertrees describe tensors as a tree of levels, Looplets describe each fiber in a level as a tree of ranges, enabling more detailed and structured access patterns. The hierarchical decomposition



(a) Concrete representation (b) Looplets representation of Interval level format and its visualization

Fig. 15. (a) Concrete representation derived from fibertree using Interval level format. (b) Looplets representation describing the pattern of Interval level format using concrete representations. Here, ϵ is `Limit(0,1)`.

provided by Looplets allows the compiler to resolve interactions, such as intersections or unions, between different structural formats, such as pinpoint levels and interval levels. Originally developed in the integer domain, but we found that Looplets can be extended naturally to the continuous domain with minimal modification. The following provides background information on four core types of Looplets designed for describing value patterns:

- **Run**: Describes repeated instances of the same payload, scalar or sub-fiber.
- **Sequence**: Concatenate multiple child Looplets into one sequence.
- **Stepper**: Repeats a single child Looplet a variable number of times.
- **Phase**: Marks the interval spanned by a child Looplet.

A useful analogy can be drawn between Looplets and regular expressions (Regex). A Run Looplet is similar to a single character in Regex, while a Sequence Looplet functions like concatenation (e.g., RS for sub-Regex R and S). A Stepper Looplet corresponds to the Kleene star (R^*), allowing for repeated patterns. For a comprehensive overview of Looplets, please refer to the original paper [3].

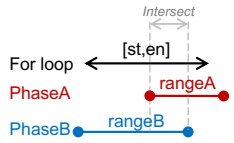
Looplets provide a mechanism to interpret the concrete memory representations of tensors as full (which may include zeros) structures. Figure 15 illustrates this process using an Interval Level format and how it can be mapped to a Looplet representation. In Figure 15(a), the concrete representation of Interval level format is derived from fibertree. Figure 15(b) visualizes how this concrete Interval Level format is transformed into a Looplet representation. The Looplets representation decomposes the entire level into structured patterns. Inside the top Sequence, a Stepper Looplet iterates over the Sequence of zero and non-zero Runs. For non-zero Runs, the corresponding value (`val[p]`) is used, where `p` is the current position in the Stepper. Phases mark the boundaries of each interval using the `left` and `right` arrays. In this way, Looplets provide a hierarchical approach to understanding the concrete memory representation of continuous tensors.

6.2 Compiler Pass for Looplets

In Finch, each Looplet type defined within a `for` loop statement is lowered by a corresponding compiler pass. We found that most of the Looplet passes can be applied to continuous space without modification, with the exception of the Stepper. Figure 16 provides background information through a visualization of how each pass lowers the Looplets within a continuous loop.

Phase (Figure 16a, 16b, and 16c) : This pass lowers continuous loops associated with Phase Looplets . It intersects Phase ranges with the loop's range, generating code to check for intersections with a length ≥ 0 (Lines 3-4 in Figure 16c).

Run (Figure 16d, 16e, and 16f) : This pass lowers Run Looplets by replacing with a constant.



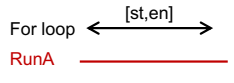
(a) Phase: Visualization.

```
# bodyA/B = child Looplet
A = Phase(rangeA, bodyA)
B = Phase(rangeB, bodyB)
for x = st:en
  s += A[x] * B[x]
```

(b) Phase: Source program.

```
# A ∩ B = [max(A.start, B.start),
#          min(A.stop, B.stop)]
intersect = [st,en] ∩ rangeA ∩ rangeB
if intersect.start <= intersect.stop
  for x = intersect.start:intersect.stop
    s += bodyA[x] * bodyB[x]
```

(c) Phase: Lowered program.



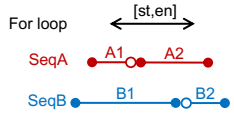
(d) Run: Visualization

```
A = Run(scalarA)
B = Run(scalarB)
for x = st:en
  s += A[x] * B[x]
```

(e) Run: Source program.

```
for x = st:en
  s += scalarA * scalarB
```

(f) Run: Lowered program.



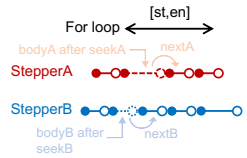
(g) Seq: Visualization.

```
A = Sequence(PhaseA1, PhaseA2)
B = Sequence(PhaseB1, PhaseB2)
for x = st:en
  s += A[x] * B[x]
```

(h) Seq: Source program.

```
for x=st:en { s += PhaseA1[x]*PhaseB1[x] }
for x=st:en { s += PhaseA1[x]*PhaseB2[x] }
for x=st:en { s += PhaseA2[x]*PhaseB1[x] }
for x=st:en { s += PhaseA2[x]*PhaseB2[x] }
```

(i) Sequence: Lowered program.



(j) Stepper: Visualization.

```
A = Stepper(seekA,bodyA,nextA)
B = Stepper(seekB,bodyB,nextB)
for x = st:en
  s += A[x] * B[x]
```

(k) Stepper: Source program.

```
call seekA(st) # find bodyA contains st
call seekB(st) # find bodyB contains st
x = st
while x <= en
  curr = [x,en] ∩ bodyA.range ∩ bodyB.range
  for x2 = curr.start:curr.stop
    s += bodyA[x2] * bodyB[x2]
  if (curr.stop==bodyA.stop) call nextA
  if (curr.stop==bodyB.stop) call nextB
  x = curr.stop + ϵ
```

(l) Stepper: Lowered program.

Fig. 16. Compiler Pass for Looplets. Continuous loops are highlighted in purple. Non-purple code in right represents generated Julia code. The compiler recursively lowers nested Looplets until no purple code remains.

Sequence (Figure 16g, 16h, and 16i): This pass lowers the continuous loop associated with Sequence Looplets, concatenating multiple child Phases. This pass generates all combinations of child Phases within each Sequence: `(A1,B1), (A1,B2), (A2,B1), (A2,B2)`.

Stepper (Figure 16j, 16k, and 16l) : This pass lowers Stepper Looplets, which repeat the child Looplet step by step. Each Stepper maintains a current body Looplet and performs computations on the intersected range. It then advances to the next body Looplet when the current one is complete. The `seek` field contains code to fast-forward the stepper to the first body containing the start (`st`) of the for loop (Lines 1-2 in Figure 16l). Within the while loop, computations occur in the intersected range between current bodies (Lines 5-7). When the current body is complete, the Stepper advances to the next body (Lines 8-9). Finally, the next starting point `x` is set to the end of the intersection incremented by ϵ (`x = curr.stop + 1`). This ϵ is essentially implemented as `Limit(0, Int8(1))`. This increment changes a closed ending point to an open starting point, and vice-versa.

The process of compiling continuous loops into executable code entails a recursive lowering of these loops until none of them are present in the code (i.e., until there is no purple code). A specific compiler pass is selected based on the type of Looplet, with a focus on lowering the outermost first in nested Looplets. In scenarios where different tensors feature varying outermost Looplet types, tiebreaking rules establish the priority order as `Run > Phase > Sequence > Stepper`.

6.3 Compiler Pass for Simplifying Program

```
# Rule for rewriting continuous reduction
rewrite_rule(for idx in interval; body) => begin
  apply_rule((op=)(lhs,rhs*d(idx,measure)) => begin
    if rhs is constant with respect to idx then
      reduce(idx, interval, lhs, op, rhs, measure)
    end
  end)(body)
end
```

```
# Limit(value, eps)
st = Limit(3.0,+1) #3.0 + ϵ
en = Limit(4.2,0)  #4.2
for x = st:en
  s += Va * Vb * d(x, Lebesgue)
↓↓↓↓↓↓↓↓↓↓↓↓↓
s += Va * Vb * drop_eps(en-st) # s += Va*Vb*1.2
```

(a) Continuous reduction rule and an example.

```
# Function to drop epsilon (e.g., 3 + ϵ => 3)
drop_eps(x::Limit) = x.value

# Lebesgue measure
reduce(idx, interval, lhs, +=, rhs, Lebesgue) =
  interval_length = length(interval)
  lhs += rhs * drop_eps(interval_length)

# Counting measure
reduce(idx, interval, lhs, +=, rhs, Counting) =
  if length(interval) == 0 then
    lhs += rhs
  end

# Boolean measure
reduce(idx, interval, lhs, ||=, rhs, Boolean) =
  lhs = lhs || rhs
```

(b) Collapsing terms based on operator and context.

Fig. 18. Rewriting rule for continuous reduction. When the rule identifies that the assignment is reducible with respect to a loop, it substitutes the loop into the collapsed expression.

```
+(a..., 0, b...) => +(a..., b...)
*(a..., 0, b...) => 0
&&(a..., true, b...) => &&(a..., b...)
&&(a..., false, b...) => false
a[i...] += 0 => emptyblock()
a[i...] *= 1 => emptyblock()
a[i...] &= true => emptyblock()
a[i...] |= false => emptyblock()
if(true, a) => a
if(false, a) => emptyblock()
for x=a:b; emptyblock() => emptyblock()
```

Fig. 17. Example Rewrite Rules in Finch

Background. Finch's simplify pass plays a crucial role in optimization. It operates after each Looplet Pass, simplifying the program through predefined rewriting rules that account for mathematical properties. Examples of such rules are illustrated in Figure 17. These rules extend beyond basic optimizations like constant propagation; some also impact control flow, such as `for` or `if` statements.While the existing rules were initially designed for the integer domain, they are equally applicable to continuous loops. By leveraging annihilators in arbitrary semirings, Finch's rewriting system can optimize continuous-domain programs as well. The current implementation supports diverse semirings such as the number, Boolean, tropical, min-max, min-product, and max-product semirings. Users can also extend the system to support custom semirings by adding new rules.

Additional Simplifications for Continuous Reduction. The main challenge in lowering continuous loops lies in managing infinite operations. However, with piecewise-constant tensors, we can simplify by focusing on constant regions, distinguishing reduction modes based on measure definition as described in Section 4. Idempotent reductions like `max` and `min` are straightforward, as they behave consistently across both points and intervals. For the `+` reduction, we define two measures: summation (interpreted via the counting measure $\mu_\#$) and integration (interpreted via the Lebesgue measure μ_λ). Summation aggregates over discrete points, while integration accumulates values across intervals by multiplying by interval length.

Continuous reduction is achieved through the addition of rewriting rules in the simplify pass, as depicted in Figure 18a. When this rule detects a `for` loop, it substitutes all applicable assignments into the collapsed expression, provided that the assignment is reducible with respect to the loop. Figure 18b shows three collapsed expressions—two measures in the real-number semiring, and a boolean measure in the boolean semiring. As discussed in the piecewise-constant evaluation semantics, users can define custom measures μ: Interval $\to R$ through rewrite rules.

In Lebesgue measure, we utilize the `drop_eps` function on the `Limit` type to remove epsilon from the length of the loop interval. In Figure 18b, `drop_eps` extracts the number(`x.value`) from `Limit` type. This is done because integration with Lebesgue measure yields the same result regardless of

Table 2. Lines of code comparison between the baseline and ours, along with our compilation times.

Applications	Baseline	Ours	LoC Saving	Ours Compilation Time
Radius Search Query	501 lines	8 lines	**62**×	16s
Genomic Interval Overlapping Query	206 lines	11 lines	**18**×	18s
Trilinear Interpolation in NeRF	82 lines	13 lines	**6**×	28s
Point Cloud Convolution	2,330 lines	23 lines	**101**×	280s

the inclusiveness of the interval (i.e., $\int_{[0,1]} f(x)dx = \int_{(0,1)} f(x)dx$). Figure 18a below provides an example of how a continuous for loop is reduced using Lebesgue measure. In summation mode using counting measure, we emit an additional condition to check if the interval is pinpoint (i.e., the length of the interval is zero). This ensures that summation only operates on pinpoint pieces.

Simplifying Interval Operations with Z3. Our compiler frequently generates code that handles interval operations, such as intersection checks or length computations. To optimize these operations, we utilize compile-time information about interval relationships within nested Looplets. We then employ the Z3 solver [22] to prove certain interval conditions statically, allowing us to eliminate unnecessary computations. A typical example occurs when checking if the intersection between two intervals (A ∩ B) has zero length, especially when one interval is a pinpoint where `B.start == B.end`. Using Z3, we verify that pinpoint intersections always have zero length. Therefore, explicit length checks become unnecessary, reducing overhead and improving performance, especially in computations involving numerous pinpoint intersections.

```
ABStart = min(A.end, B.end)
ABEnd = max(A.start, B.start)
if ABEnd >= ABStart:
    Length = ABEnd - ABStart
    if Length == 0:
        Sum += val
```

(a) Original Code

```
ABStart = min(A.end, B.end)
ABEnd = max(A.start, B.start)
if ABEnd >= ABStart:
    Sum += val
```

(b) Simplified Code

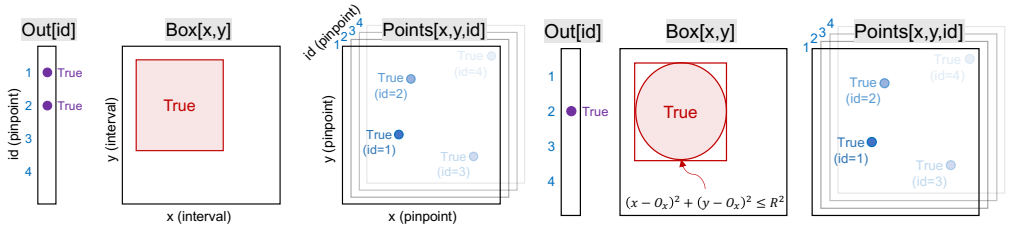
Fig. 19. Optimizing pinpoint interval intersection checks using Z3.

7 Case Studies

In this section, we explore diverse applications under the continuous tensor abstraction across four domains: (1) Geospatial search, (2) Genomic interval operations in Bioinformatics, (3) Interpolation in Neural Radiance Field, and (4) 3D point cloud convolution. Our goal is to show the simplicity and clarity with which these applications can be expressed in our abstraction, particularly compared to challenges in existing tensor programming (Table 2). Additionally, we report compilation time. Our extension for continuous tensors adds little overhead. Most of the compile time comes from the number of indices in Einsum, as each index is lowered into its own looplet, leading to deeper loop nests and more IR to generate and optimize. In practice this is a one-time cost, since the compiled kernel can be reused for any inputs. We assess the performance of the code generated by our compiler by comparing it with hand-written libraries. All experiments are conducted through single-threaded execution on a Macbook Pro M2 Max with 32GB of memory.

7.1 Geospatial Search

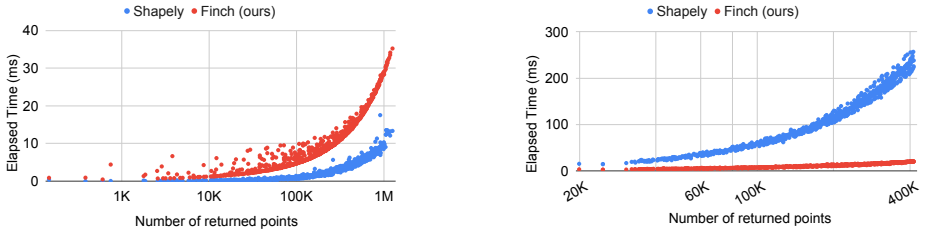
The first application we have explored is the spatial search query on 2D points, a widely used technique in applications such as geographical information systems (GIS) [15], computer-aided design (CAD) [10], and spatial databases [34]. In our study, we focused on two commonly employed queries: (1) the box search and (2) the radius search. In a box search, given a set of 2D points, this



(a) **Box search query**:
$Out_{id} = (Box_{x,y}$ && $Points_{x,y,id})$ * $d(x,y;\mu_{\wedge\vee},\mu_{\wedge\vee})$

(b) **Radius search query**:
$Out_{id} = Box_{x,y}$ && $Points_{x,y,id}$ && $(x-O_x)^2+(y-O_y)^2 \le R^2$

Fig. 20. Illustration of two spatial searches. Points are represented as a 3D tensor `Points[x, y, id]`, with each point assigned a unique ID. `Out` retrieves the `id` of points intersecting a specified box or circle. Measure on the radius search query omitted for brevity; it also uses boolean measures $d(x,y;\mu_{\wedge\vee},\mu_{\wedge\vee})$.



(a) Box search query with increasing box size.

(b) Radius search query with increasing radius.

Fig. 21. Experimental result of spatial queries: lower values indicate better performance.

query retrieves all points within a specified box. A radius search query retrieves all points within a circle centered at (O_x, O_y) with a radius of R.

Figure 20 demonstrates how spatial search queries are represented in continuous tensor abstraction. We transform 2D points into a 3D continuous tensor, denoted as $Points_{x,y,id}$, by assigning a unique ID to each point (x, y). This approach accommodates multiple points that may share the same coordinates (x, y) depending on the dataset. Figure 20a illustrates a box query, which outputs IDs of points intersecting with $Box_{x,y}$. Similarly, Figure 20b presents a radius search query where the circle is defined with $(x - O_x)^2 + (y - O_y)^2 \le R^2$.

Figure 21 presents the performance of our generated code in comparison to Shapely [31], a Python wrapper for GEOS [30], a well-known C++ library widely used in GIS for performing operations on two-dimensional geometries. We employed a synthetic dataset that uniformly distributed 10 million points in the range [0,10000] × [0,10000]. In both experiments, we increased the size of the query shape along the x-axis to augment the number of returned output points.

Figure 21a shows that Shapely outperforms our generated code on the box search query, achieving a geometric mean speedup of 4.7×. This advantage is mainly due to Shapely's use of an advanced spatial data structure, STRtree [47], which accelerates search operations. In contrast, our code does not employ any spatial data structures. However, Figure 21b shows that our code surpasses Shapely on the radius query by a geometric mean of 9.2×. This improvement ironically stems from Shapely's reliance on STRtree, which only supports axis-aligned bounding boxes. For a circular query, Shapely retrieves all points within the bounding box of a circle using STRtree and then performs a linear scan to check if each point lies within the radius R. In contrast, our generated code directly iterates within the circular region (Figure 20b), avoiding the inefficiency of Shapely's two-step approach.

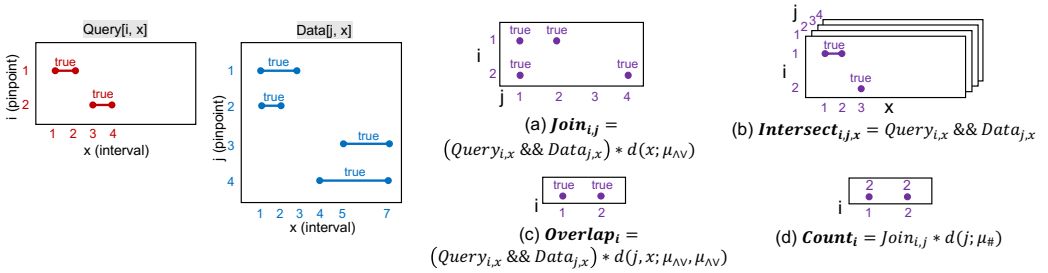


Fig. 22. Four different genomic interval operations in continuous tensor abstraction. Genomic intervals are represented as 2D continuous tensor ($Query$ and $Data$) where white regions indicate 'false' boolean values.

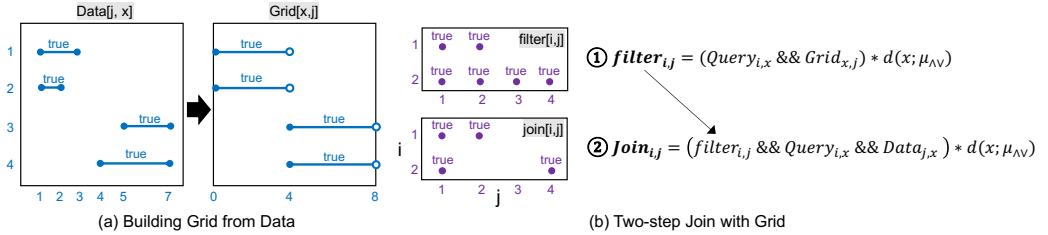


Fig. 23. The uniform grid splits the x-dimension of 'Data' into two intervals, $[0, 4)$ and $[4, 8)$, to streamline intersection tests by eliminating irrelevant data points. (a) illustrates the construction of the grid. (b) demonstrates a two-step join operation using the grid: the first step ($filter_{i,j}$) prunes unnecessary pairs (i, j) with the grid structure, while the second step ($Join_{i,j}$) performs the actual join operation only on filtered pairs.

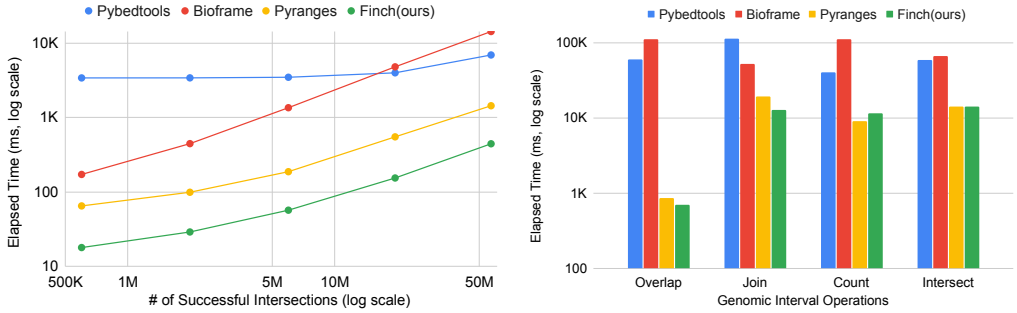
7.2 Genomic Interval Operations

The second application we've explored is genomic interval operations using our continuous abstraction. In Bioinformatics, performing operations on genomic sequences [58] and applying boolean operations to genomic interval data can be computationally intensive.

In Figure 22, genomic interval data is depicted in a 2D continuous tensor, denoted as the interval database $Data_{j,x}$ and query intervals $Query_{i,x}$, each identified by a unique interval ID. Right side of the figure illustrates four genomic operations written in the continuous tensor abstraction. For example, $Count_i$ counts the number of intervals in $Data$ intersecting with a query interval for each query ID i. These programs are expressed clearly and succinctly in the continuous tensor abstraction. However, the naive version involves pairwise comparisons ($N \times M$) between all query and data intervals (i, j), incurring substantial computational costs when handling numerous intervals.

In practice, previous studies [4, 27, 28, 48, 58] have used an interval data structure to skip unnecessary comparisons. We demonstrated a uniform grid [28] using our abstraction to partition the continuous dimension x into exclusive partitions, encompassing the entire dimension x. The uniform grid, represented in a 2D continuous tensor as $Grid_{x,j}$, is illustrated in Figure 23. It divides the x domain of $Data_{j,x}$ into two halves: [0,4) and [4,8), with interval IDs j allocated to the respective partitions. The right side of the figure shows an example on a Join operation using the uniform grid. First, the uniform grid filters out irrelevant intervals in Data, reducing the need for pairwise comparisons. In the second stage, it further eliminates false positives to refine the final results.

Figure 24 presents a performance comparison between our generated code using uniform grid and three baseline implementations: Pybedtools [21], Bioframe [55], and Pyranges [66]. Pybedtools is a Python wrapper of Bedtools [58], a C library which utilizes a hierarchical binning data structure



(a) Sensitivity test regarding the count of successful intersections in a synthetic dataset.

(b) Performance comparison using realistic dataset.

Fig. 24. Experimental result of genomic interval operations: lower values indicate better performance.

internally, Bioframe leverages the Pandas [50] framework for genomic interval operations, and Pyranges employs a Nested Containment List [4], a variation of the segment tree written in C. Index building time was not measured in these experiments.

In Figure 24a, a sensitivity test examines the number of successful intersections using a synthetic dataset. The intervals are uniformly distributed, maintaining a total of 100,000 intervals in both `Data` and `Query`. As the x-axis increases, the length of intervals in both `Data` and `Query` is extended to increase the number of intersections between intervals. Figure 24b presents a performance comparison on a realistic dataset [48], with `Data` containing 8,942,869 intervals and `Query` containing 1,193,657 intervals. Our generated code demonstrates superior or comparable performance in both synthetic and realistic datasets, with the advantage of being implemented in just 11 lines of code for the Overlap operation, while Bedtools implementation require 206 lines of code.

7.3 Trilinear Interpolation in Neural Radiance Field

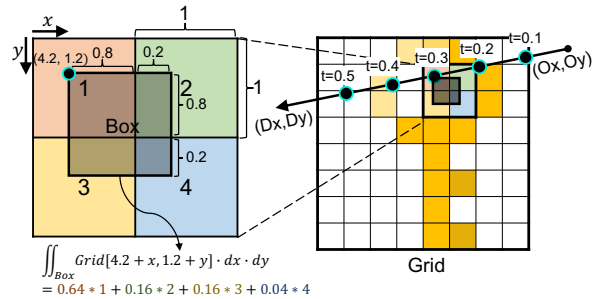
The third application we explored is a Neural Radiance Field (NeRF) [52] in 3D deep learning. NeRF is a widely used machine learning model in computer graphics and computer vision that generates detailed 3D reconstructions from 2D images. Many NeRF models [29, 49, 69] use trilinear interpolation on 3D sparse voxel grids to efficiently represent the 3D scene. In NeRF, rendering a 2D image involves casting rays, sampling points along each ray, performing trilinear interpolation on the sparse voxel grid for each sampled point, and combining these results to compute the final RGB color. Our focus here is on optimizing trilinear interpolation during ray sampling in Plenoxel [29].

For explanatory purposes, we illustrate a 2D bilinear interpolation in Figure 25a, even though the actual computations are in 3D. This interpolation is applied at each sampled point along the ray by calculating the intersected area using integral reduction:

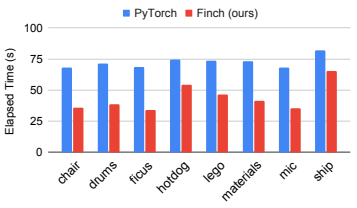
$$Out_t = Time_t * Grid_{O_x+D_x*t+x, O_y+D_y*t+y} * Box_{x,y} * d(x, y; \mu_\lambda, \mu_\lambda),$$

where (O_x, O_y) and (D_x, D_y) are the ray's origin and direction, respectively. $Box_{x,y}$ is a binary tensor defining the interpolating region, and $Time_t$ is a binary tensor marking the specific time step for sampling along the ray. In practice, these calculations occur in 3D space.

Figure 25b presents a performance comparison between our generated code and a PyTorch implementation of trilinear interpolation in Plenoxel. The evaluation was conducted while rendering a 256×256 image using the NeRF Synthetic dataset [52]. Our code achieves a speedup of 1.3× to 2.0× over the baseline. This improvement is mainly due to our efficient handling of sparse voxel grids, as we store only non-zero voxels and avoid unnecessary computations. In contrast, the



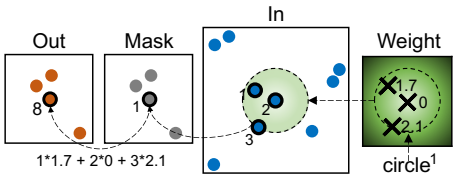
(a) Interpolation at sampled points on a sparse grid.



(b) Performance comparison.

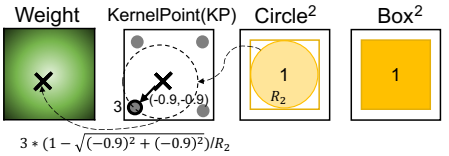
$$Out_t = Time_t * Grid_{O_x+D_x*t+x, O_y+D_y*t+y} * Box_{x,y} * d(x, y; \mu_\lambda, \mu_\lambda)$$

Fig. 25. (a) Interpolation at sampled points during ray sampling on a sparse grid in 2D for illustrative purposes. (b) Performance comparison between PyTorch and our implementation on various 3D objects in the NeRF Synthetic dataset [52], where lower values indicate better performance.



$$Circle^1_{r,s} = Box^1_{r,s} * (r^2 + s^2 \le R_1^2)$$
$$Out_{x,y} = Mask_{x,y} * In_{x+r,y+s} * (Weight_{r,s} * Circle^1_{r,s})$$

(a) Kernel Points Convolution (KpConv) [70]



$$Circle^2_{p,q} = Box^2_{p,q} * (p^2 + q^2 \le R_2^2)$$
$$Weight_{r,s} = KP_{r+p,s+q} * \left(\left(1 - \sqrt{p^2 + q^2}/R_2\right) * Circle^2_{p,q}\right)$$

(b) Definition of $Weight$. $Weight$ interpolates the points in $KernelPoint$ within radius R_1.

Fig. 26. Kernel Points Convolution on input point clouds written in continuous tensor abstraction. Measure omitted for brevity; all reductions use counting measures.

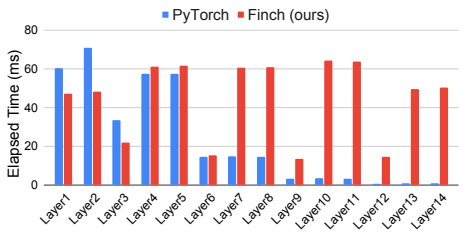
PyTorch implementation uses a fully dense voxel grid, storing even empty voxels and performing calculations regardless of voxel occupancy.

7.4 3D Point Cloud Convolution

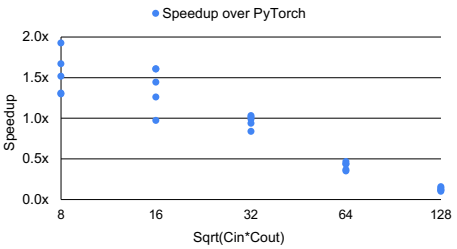
The last application we explored involves 3D point cloud convolution [20, 32, 72]. A 3D point cloud is a set of points represented by xyz coordinates in 3D space, capturing the structure of an object. In 3D deep learning, convolutions are adapted to operate on point clouds rather than grid-like image data, necessitating specialized techniques. KPConv [70] is a notable example of such a technique, and we demonstrate its implementation using our continuous tensor abstraction.

Figure 26 shows how KPConv, represented in 2D for simplicity, can be formulated within continuous tensor expression. In Figure 26a, KPConv performs convolutions selectively on specific points, marked by a binary mask, $Mask$. For each masked point, the neighboring input points, $In_{x+r,y+s}$, within a radius R_1 are gathered and convolved with a continuously varying weight, $Weight_{r,s}$, which depends on the relative position of each neighboring point. The radius-constrained region is indicated by $Circle_{r,s}$, which determines which neighboring points are gathered.

As shown in Figure 26b, the continuous weight $Weight_{r,s}$ is defined by interpolating values at fixed set of "kernel points" within a radius R_2. This interpolation is distance-based, relying only



(a) Performance comparison of convolutional layers in the network.



(b) Speedup and slowdown of our generated code over PyTorch with increasing channel size.

Fig. 27. Experimental results of 3D point cloud convolution: (a) Lower is better, (b) Higher is better.

on kernel points within the radius. Notably, the interpolated weight $Weight_{r,s}$ is not a piecewise-constant, but is evaluated only on pinpoints selected by the mask $Mask_{x,y}$ in the main convolution.

By using continuous tensor abstraction, complex point cloud convolutions that are challenging to implement in traditional tensor programming models can be expressed in a simplified manner. In contrast, the original PyTorch implementation of KPConv requires 2,330 lines of code, including dependencies for neighborhood search libraries [13]. With our abstraction, users can specify the whole KPConv operation as a single Einsum:

$$Out_{x,y,m} = Mask_{x,y} * In_{x+r,y+s,c} * (KP_{r+p,s+q,m,c} * (1 - \sqrt{p^2 + q^2}/R_2^2) * Circle^2_{p,q}) * Circle^1_{r,s} * d(r, s, c; \mu_\#, \mu_\#, \mu_\#)$$

where m and c are indices for the output and input channels, respectively, along with the traversal order of indices and the choice of storage format.

Figure 27 presents the experimental results comparing our generated code with the PyTorch implementation of KPConv. In Figure 27a, the elapsed time of each KPConv layer in the 3D shape classification model architecture using the ModelNet40 [74] dataset is shown. In the initial layers (layers 1-6), our code either outperforms or matches the performance of the PyTorch implementation. However, starting from layer 7, PyTorch begins to outperform our generated code.

We found that the later layers have larger channel sizes, resulting in dense computations. PyTorch leverages highly optimized dense BLAS routines for this large tensor processing. Figure 27b further illustrates this difference by showing the speedup over PyTorch as channel size increases. In the initial layers, where the channel sizes are small ($\sqrt{C_{in} \cdot C_{out}} \leq 32$), our generated code performs better. However, in the later layers with larger channel sizes, where dense computation becomes more significant, PyTorch's implementation excels. Consequently, our code performs better in the early layers, where the radius query is the primary computational component, while the PyTorch implementation dominates in the later layers, where dense computation is the primary workload.

8 Related Works

Dense tensor programming models. Tensor programming, rooted in Fortran's array data structure [6], has provided a foundation for diverse applications. In recent years, the machine learning community has introduced frameworks like TensorFlow [1], Jax [14], and PyTorch [56], inspired by tensor-focused languages like Matlab [40] and NumPy [36]. These frameworks are instrumental in developing machine learning models, which heavily rely on tensor operations. The latest advancements in scheduling languages [16, 59] have played a pivotal role in enhancing the performance of tensor-based programs. They separate tensor programs into what to compute (algorithm) and how to compute (schedule), simplifying the creation of high-performance tensor programs and the exploration of various loop transformations.

Sparse tensor programming models. Many of our designs take inspiration from existing sparse tensor programming models. Sparse tensors provide multiple storage formats. The level format abstraction, originally introduced in TACO [19], explains the diversity of sparse formats by introducing the concepts of a coordinate hierarchy and level format. This abstraction has further evolved into the fibertree abstraction [68], which serves as a format abstraction for our work.

Sparse tensor programming often entails complex code to co-iterate over multiple sparse tensors, each stored in a different format. Numerous compiler projects have dedicated their efforts to generating efficient code for accommodating these diverse formats. Projects like Taichi [38], MLIR sparse dialect [12], TACO [45], SparseTIR [75], and Finch [2, 3] can generate efficient code from sparsity-agnostic definitions of computation. The TACO project [19, 37, 45, 63] introduces the "merge lattice" concept to efficiently generate code for sparse tensor algebras, even when the tensors are stored in different sparse formats. In a recent development, the Finch project [3] introduces the innovative concept of "Looplets," simplifying the generation of sparse code on integer domain through the use of rewriting rules and enhancing extensibility. Looplets support various element types, not limited to numeric types, and a wide range of operators, expanding their versatility.

Continuous programming models. Several tools, such as Chebfun [7] and Sympy [51], offer an intuitive way to manipulate continuous functions in numerical computing. In addition to working with piecewise constant functions, they offer the capability to handle a wider range of function types beyond constant functions. However, their primary focus is not on performance or the tensor programming model. Chebfun focuses on Chebyshev polynomials, a computation class commonly used in numerical computing, which is entirely different from our focus. Both Sympy and Chebfun do not account for sparsity or interval intersections, resulting in the need to compare all pairs of pieces. However, our framework allows the creation of more efficient code that operates only on intersecting pieces, eliminating the need to compare all pairs.

Computer graphics and scientific image visualization languages, such as Diderot [44] and Vivaldi [18], also use continuous fields, enabling them to define functions beyond piecewise-constant, similar to systems like Chebfun or Sympy. However, their primary application often revolves around operations like texture sampling [57], which fundamentally rely on pointwise evaluations of these continuous fields and their derivatives. Consequently, their design does not prioritize complex geometric operations, and they typically do not support computations on regions beyond pinpoints, such as intervals or N-dimensional shapes. For instance, while Vivaldi supports neighbor queries, these operations are restricted to neighboring points, not continuous intervals. Thus, they lack support for complex geometric operations like interval intersections. In contrast, while our continuous tensor representation is currently restricted to piecewise-constant, our approach enables computations beyond pinpoint evaluation. This allows us to express various geometric operations, such as point cloud convolution and spatial search.

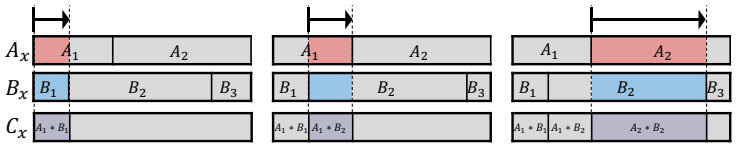
Spatial Database and Spatial Join. Spatial databases [24, 30, 31, 34] are designed to efficiently handle geometric data. A key operation is the spatial join [42], which identifies all pairs of geometries from two datasets that satisfy a given spatial predicate (e.g., intersects, contains). This operation is challenging due to the expensive cost of geometric checks and the quadratic number of pairings. Consequently, spatial join has been extensively studied, and various methods have been developed to make it more efficient than exhaustive search. Spatial join is also closely related to partitioning the iteration space in piecewise evaluation semantics. When computing with piecewise tensors, it is essential to efficiently identify pairs of pieces that intersect which is analogous to spatial join.

A common approach to spatial join is to use spatial indexes, such as interval trees [4, 26, 27, 48], KD-trees [9], R-trees [35, 47] or uniform grids [8, 28], to reduce the number of candidate pairs. Tree-based joins traverse hierarchical partition of the space or hierarchical bounding boxes of the

geometries [46] to find potentially intersecting pairs. Grid-based joins assign geometries to uniform grid cells, limiting comparisons to objects within the same cell.

Alternatively, some methods avoid spatial indexing by using plane sweep (also known as line sweep) algorithms [42, 43, 54]. In this approach, all geometries from both inputs are combined and sorted along a chosen axis. A virtual sweep line then moves across space, maintaining an active set of potentially intersecting intervals. By leveraging this sorted order, plane sweep eliminates the overhead of building spatial indexes and avoids exhaustive pairwise comparisons.

We observe that continuous iteration over real-valued indices in continuous Einsums, together with Looplet-generated code, generalizes the plane-sweep spatial join when co-iterating the piecewise tensors. As shown in the figure, computing $C_x = A_x * B_x$ effectively sweeps a real-valued index x along the x-axis of tensors A and B. In higher-dimensional tensors, the storage level order determines the sweeping order of the high-dimensional iteration space.



(a) $C_x = A_x * B_x$, computed by sweeping a virtual line over the x-axis.

```
pA, pB, pC = 0, 0, 0
while (pA<2 and pB<3):
 lA,rA,vA = A[pA]
 lB,rB,vB = B[pB]
 lAB,rAB = max(lA,lB),min(rA,rB)
 if lAB <= rAB:
  C[pC] = (lAB, rAB, vA*vB)
  pC += 1
 pA,pB += (rA==rAB),(rB==rAB)
```

(b) Generated code

Unlike the classical plane sweep, which aggregates all geometries, globally sorts them once, and then sweeps—each tensor here keeps its intervals already sorted, and the generated code performs an on-the-fly merge of these sorted streams. Consequently, our sorted storage functions as a lightweight spatial index, and iteration over real indices proceeds in a plane sweep like fashion.

Combining spatial indices with the level format abstraction is a promising direction. Grid-based spatial indices, such as uniform and sparse grids, align naturally with our model. Each grid cell can be seen as an N-D interval, allowing the entire grid to be represented as a continuous tensor. We demonstrated this in our case studies by modeling uniform grids as 2D tensors (Figure 23) and implementing sparse grids in the NeRF example (Figure 25a). Tree-based spatial indices like R-trees and KD-trees, are more complex. One way to represent them is by treating the tree as an N-D tensor, where each tree level corresponds to a tensor dimension (e.g., a depth-3 tree as a 3D tensor). Alternatively, the entire tree can represent a single logical dimension by linearizing its nodes (e.g., using pre-order traversal) and wrapping the structure as a single level.

9 Conclusion

In this paper, we have introduced the continuous tensor abstraction, which extends tensor indices to real numbers. Our approach, based on piecewise-constant tensors, provides both a novel format abstraction for storage and an efficient code generation technique for continuous tensor expressions. This abstraction enables diverse applications including genomic interval operations, spatial searches, point cloud convolution, and trilinear interpolation on sparse voxel grids. It offers a fresh perspective on these applications, exploring domains largely untouched by traditional tensor programming models. We believe this work opens new possibilities in tensor programming.

Acknowledgments

We thank reviewers for their valuable feedback. Supported by Intel; NSF (CCF-2217064, CCF-2107244, CCF-2217099); DARPA (PROWESS HR0011-23-C-0101, SBIR HR001123C0139); DoE PSAAP (DE-NA0003965).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and Structured Tensor Programming with Control Flow. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 117 (April 2025), 31 pages. https://doi.org/10.1145/3720473

[3] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 41–54.

[4] Alexander V Alekseyenko and Christopher J Lee. 2007. Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. *Bioinformatics* 23, 11 (2007), 1386–1393.

[5] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java programming language*. Addison Wesley Professional.

[6] John Backus. 1978. The history of Fortran I, II, and III. *ACM Sigplan Notices* 13, 8 (1978), 165–180.

[7] Zachary Battles and Lloyd N Trefethen. 2004. An extension of MATLAB to continuous functions and operators. *SIAM Journal on Scientific Computing* 25, 5 (2004), 1743–1770.

[8] Ludger Becker, Klaus Hinrichs, and Ulrich Finke. 1993. A New Algorithm for Computing Joins with Grid Files. In *Proceedings of the Ninth International Conference on Data Engineering*. IEEE Computer Society, USA, 190–197.

[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[10] Stefan Berchtold, Christian Böhm, Daniel A Keim, and Hans-Peter Kriegel. 1997. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 78–86.

[11] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. 2012. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145* (2012).

[12] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler support for sparse tensor computations in MLIR. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 4 (2022), 1–25.

[13] Jose Luis Blanco and Pranjal Kumar Rai. 2014. nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann.

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[15] Peter A Burrough, Rachael A McDonnell, and Christopher D Lloyd. 2015. *Principles of geographical information systems*. Oxford University Press, USA.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[17] Yukang Chen, Jianhui Liu, Xiangyu Zhang, Xiaojuan Qi, and Jiaya Jia. 2023. Largekernel3d: Scaling up kernels in 3d sparse cnns. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 13488–13498.

[18] Hyungsuk Choi, Woohyuk Choi, Tran Minh Quan, David G. C. Hildebrand, Hanspeter Pfister, and Won-Ki Jeong. 2014. Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2407–2416. https://doi.org/10.1109/TVCG.2014.2346322

[19] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[20] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3075–3084.

[21] Ryan K Dale, Brent S Pedersen, and Aaron R Quinlan. 2011. Pybedtools: a flexible Python library for manipulating genomic datasets and annotations. *Bioinformatics* 27, 24 (2011), 3423–3424.

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[23] Joseph Diestel and B Faires. 1974. On vector measures. *Trans. Amer. Math. Soc.* 198 (1974), 253–271.

[24] M. J. Egenhofer. 1994. Spatial SQL: A Query and Presentation Language. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 86–95. https://doi.org/10.1109/69.273029

[25] Albert Einstein. 1916. The foundation of the general theory of relativity. *Annalen Phys.* 49, 7 (1916), 769–822. https://doi.org/10.1002/andp.19163540702

[26] Jost Enderle, Matthias Hampel, and Thomas Seidl. 2004. Joining interval data in relational databases. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 683–694. https://doi.org/10.1145/1007568.1007645

[27] Jianglin Feng, Aakrosh Ratan, and Nathan C Sheffield. 2019. Augmented Interval List: a novel data structure for efficient genomic interval search. *Bioinformatics* 35, 23 (2019), 4907–4911.

[28] Jianglin Feng and Nathan C Sheffield. 2021. IGD: high-performance search for large-scale genomic interval datasets. *Bioinformatics* 37, 1 (2021), 118–120.

[29] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5501–5510.

[30] GEOS contributors. 2021. *GEOS coordinate transformation software library*. Open Source Geospatial Foundation. https://libgeos.org/

[31] Sean Gillies, Casper van der Wel, Joris Van den Bossche, Mike W. Taves, Joshua Arnott, Brendan C. Ward, and others. 2023. *Shapely*. https://doi.org/10.5281/zenodo.5597138

[32] Benjamin Graham, Martin Engelcke, and Laurens Van Der Maaten. 2018. 3d semantic segmentation with submanifold sparse convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 9224–9232.

[33] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.

[34] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *the VLDB Journal* 3 (1994), 357–399.

[35] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.* 14, 2 (June 1984), 47–57. https://doi.org/10.1145/971697.602266

[36] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[37] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[38] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.

[39] Binh-Son Hua, Minh-Khoi Tran, and Sai-Kit Yeung. 2018. Pointwise convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 984–993.

[40] The MathWorks Inc. 2022. *MATLAB version: 9.13.0 (R2022b)*. Natick, Massachusetts, United States. https://www.mathworks.com

[41] Kenneth E Iverson. 1962. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*. 345–351.

[42] Edwin Jacox and Hanan Samet. 2007. Spatial Join Techniques. *ACM Trans. Database Syst.* 32 (03 2007), 7. https://doi.org/10.1145/1206049.1206056

[43] Edwin H. Jacox and Hanan Samet. 2003. Iterative spatial join. *ACM Trans. Database Syst.* 28, 3 (Sept. 2003), 230–256. https://doi.org/10.1145/937598.937600

[44] Gordon Kindlmann, Charisee Chiw, Nicholas Seltzer, Lamont Samuels, and John Reppy. 2016. Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 867–876. https://doi.org/10.1109/TVCG.2015.2467449

[45] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[46] James T Klosowski, Martin Held, Joseph SB Mitchell, Henry Sowizral, and Karel Zikan. 1998. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE transactions on Visualization and Computer Graphics* 4, 1 (1998), 21–36.

[47] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings 13th international conference on data engineering*. IEEE, 497–506.

[48] Heng Li. 2020. *Biofast: A small benchmark for evaluating the performance of programming languages and implementations on a few common tasks in the field of Bioinformatics*. https://github.com/lh3/biofast

[49] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. *Advances in Neural Information Processing Systems* 33 (2020), 15651–15663.

[50] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

[51] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103.

[52] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2021. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1 (2021), 99–106.

[53] Nandeeka Nayak, Toluwanimi O Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. 2023. TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators. *arXiv preprint arXiv:2304.07931* (2023).

[54] J. Nievergelt and F. P. Preparata. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM* 25, 10 (Oct. 1982), 739–747. https://doi.org/10.1145/358656.358681

[55] Open2C, Nezar Abdennur, Geoffrey Fudenberg, Ilya Flyamer, Aleksandra A Galitsyna, Anton Goloborodko, Maxim Imakaev, and Sergey V Venev. 2022. Bioframe: operations on genomic intervals in pandas dataframes. *bioRxiv* (2022), 2022–02.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[57] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann.

[58] Aaron R Quinlan and Ira M Hall. 2010. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics* 26, 6 (2010), 841–842.

[59] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[60] Halsey Lawrence Royden and Patrick Fitzpatrick. 1968. *Real analysis.* Vol. 2. Macmillan New York.

[61] Walter Rudin. 1986. *Real and Complex Analysis* (3 ed.). McGraw-Hill Professional, New York, NY.

[62] Michel F Sanner et al. 1999. Python: a programming language for software integration and development. *J Mol Graph Model* 17, 1 (1999), 57–61.

[63] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[64] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (April 2022), 33 pages. https://doi.org/10.1145/3527333

[65] Maurice Sion. 2006. *A theory of semigroup valued measures.* Vol. 355. Springer.

[66] Endre Bakken Stovner and Pål Sætrom. 2020. PyRanges: efficient comparison of genomic intervals in Python. *Bioinformatics* 36, 3 (2020), 918–919.

[67] Bjarne Stroustrup. 2013. *The C++ programming language.* Pearson Education.

[68] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2020. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture* 15, 2 (2020), 1–341.

[69] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. 2021. Neural geometric level of detail: Real-time rendering with implicit 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 11358–11367.

[70] Hugues Thomas, Charles R Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette, and Leonidas J Guibas. 2019. Kpconv: Flexible and deformable convolution for point clouds. In *Proceedings of the IEEE/CVF international conference on computer vision.* 6411–6420.

[71] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. https://doi.org/10.14778/3407790.3407799

[72] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. 2023. Unified Convolution Framework: A compiler-based approach to support sparse convolutions. *Proceedings of Machine Learning and Systems* 5 (2023), 666–679.

[73] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. 2023. WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 920–934. https://doi.org/10.1145/3575693.3575742

[74] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 2015. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and*

pattern recognition. 1912–1920.

[75] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678.

Received 2025-03-21; accepted 2025-08-12