The Continuous Tensor Abstraction: Where Indices are Real

JAEYEON WON, MIT CSAIL, USA

WILLOW AHRENS, MIT CSAIL, USA

JOEL S. EMER, MIT CSAIL / NVIDIA, USA

SAMAN AMARASINGHE, MIT CSAIL, USA

This paper introduces the continuous tensor abstraction, allowing indices to take real-number values (e.g., A[3.14]), and provides a continuous loop construct that iterates over the infinitely large set of real numbers. This paper expands the existing tensor abstraction to include continuous tensors that exhibit a piecewise-constant property, enabling the transformation of an infinite amount of computation into a finite amount. Additionally, we present a new tensor format abstraction for storing continuous tensors and a code generation technique that automatically generates kernels for the continuous tensor abstraction. Our approach introduces a novel method for loop-level reasoning in domains like computational geometry and computer graphics, traditionally unexplored in tensor programming models. Our approach demonstrates comparable 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 ~100× fewer lines of code (LoC), 1.22× on genomic interval overlapping queries (with ~26× LoC saving), and 1.69× on trilinear interpolation in Neural Radiance Field (with ~9× LoC saving).

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++ [53], Java [5], Julia [10], and Python [50], as well as specialized array-focused languages including APL [33], MATLAB [32], TensorFlow [1], and PyTorch [45]. This paradigm forms the basis for compiler analyses, polyhedral transformation [13], and loop vectorization [17, 36] aimed at optimizing array-based programs. Array programming also plays a vital role in performance engineering, employing techniques like loop tiling and reordering for enhanced software efficiency [48].

Array programming traditionally requires values at every integer coordinate, whereas recent advancements in the sparse tensor programming model[1] efficiently manage sparse data with values at only a few specific integer coordinates. This innovation leverages the ineffectual computations introduced by multiplication by zero ($a * 0 = 0$) to boost program efficiency. Sparse tensor programming excels in handling such data by exclusively processing non-zero points. Users can write sparse tensor programs with a dense-like approach, simplifying sparse data management and maintaining the illusion of working with large tensors. Specialized DSLs such as TACO [34], SparseTIR [59], and Finch [2] facilitate this transition by optimizing sparse tensor formats and non-zero data processing, all while using discrete integer indices. Dense tensors stored data at every integer grid point, but sparse tensor programming eliminated this need by storing only relevant grid points. This work goes further, removing the requirement for stored points to be integers.

Existing tensor programming models are well-suited for representing data at discrete integer grid coordinates, making them a natural choice for operations like matrix multiplication. However, they face challenges in handling domains dealing with continuous data which do not seamlessly fit into the current tensor programming model. For instance, writing programs in fields like computational geometry and computer graphics becomes challenging because programmers cannot find a direct

---

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

---

way to map geometries into array data structures. Consequently, they have devised myriad formats for these domains, such as interval trees [46] and Bounding Volume Hierarchies [35], to efficiently store and iterate over geometries. This requires complex data structures and even more complicated control flow structures than the straightforward loop nest structures and simple indexed tensors.

In this paper, we expand the boundaries of the tensor programming model by extending coordinate points from finite ranges of integers to the infinite realm of real numbers. This expansion allows data points to have real numbered coordinates, offering a wide range of new possibilities. Users can access tensors using real-numbered indices (e.g., A[3.14]) and iterate through them via a continuous for loop (e.g., for i = 0.0:3.14; B[i] = A[i]). In the Figure 1, we present a 2D radius search query that counts the number of points in A within a distance of 1.7 from (2.2, 3.9), expressed using the continuous tensor abstraction. The syntax aligns with the ubiquitous tensor programming model, but now the for loop iterates continuously over the real number domain. This query serves as a commonly employed operation in applications such as spatial databases or geometric programs. Remarkably, our implementation requires ~100× fewer lines of code and performs better compared to the manually-written library [26], underscoring its brevity and efficiency.


```
count = 0
for dx=-1.7:1.7  # continuous
 for dy=-1.7:1.7 # continuous
  if dx*dx+dy*dy <= 1.7*1.7
   count += A[2.2+dx,3.9+dy]
# count = 3
```

Fig. 1. Radius search query in continuous tensor abstraction.

Using a piecewise-constant assumption, we introduce novel methods for storing continuous tensors in memory, evaluating reduction operations in continuous loops (for i = 0.0:3.14; sum += A[i]), and generating efficient code for continuous tensor programs. We believe that the continuous tensor abstraction marks a new era of loop-level reasoning, offering the potential to unify diverse applications across various domains that were unexplored in traditional tensor programming models.

To the best of our knowledge, this paper is the first to **extend tensor programming to real-numbered indices with continuous loops.** In addition, this paper includes the following contributions:

- We show how to iterate over continuous tensors with loops.
- We demonstrate how to represent an infinite number of values in a continuous tensor under the piecewise-constant assumption.
- We introduce reduction operations specifically designed for the continuous loop.
- We introduce an efficient code generation mechanism for continuous loops by extending the fibertree abstraction [54] and Finch [4].
- We show how to use a number type reflecting the infinitesimal number ϵ, Limit, to implement the inclusiveness of intervals.
- 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 ~26× fewer lines of code in bioinformatics, ~100× fewer lines of code in geospatial queries, ~145× fewer lines of code in 3D point cloud convolution, and ~9× 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.

(a) Sparse vector x_i and y_i.
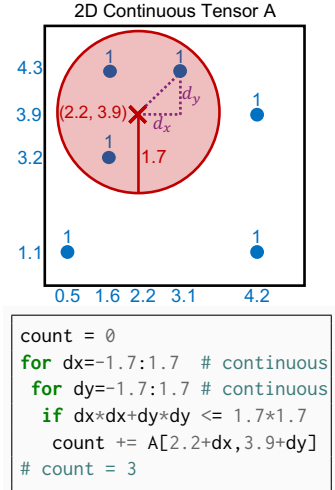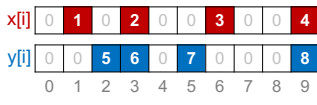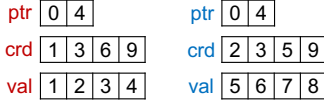
(b) Physical storage of (a).

(c) Dot product between sparse vectors, $s = \sum_{i=0\ldots9} x_i * y_i$.

```
#loop iterates discretely
for i = 0:9
    s += x[i] * y[i]
end # s = 44
```

(d) Continuous vectors x_i and y_i with pinpoint coordinates.

(e) Physical storage of (d).

(f) Dot product between continuous vectors.

```
#loop iterates continuously
for i = 0.0:9.0
    s += x[i] * y[i]
end # s = 44
```

(g) Continuous vector x_i and y_i with interval coordinates.

(h) Physical storage of (g).

(i) Dot product between continuous vectors, $s = \int_{0.0}^{9.0} x_i * y_i * di$

```
#loop iterates continuously
for i = 0.0:9.0
    s += x[i] * y[i] * d(i)
end # s = 2.4
```

Fig. 2. Motivating examples: **(a,b,c)** A dot product between two sparse vectors. **(d,e,f)** A dot product between two continuous vectors with pinpoint coordinates. **(g,h,i)** A dot product between two continuous vectors with interval coordinates. Non-colored coordinates (in the white region) in continuous vectors (d, g) have values of zero. Our codes (f, i) give users the illusion of iterating over a continuous domain.

2 MOTIVATING EXAMPLES

In this section, we explain the fundamental design principles behind the continuous tensor abstraction, clarified through illustrative examples. Our exploration begins with a common operation in the existing sparse tensor programming world: the dot product between sparse vectors ($s = \sum_i A_i * B_i$). We then make an extension of the dot product into the continuous domain.

Figure 2 illustrates this extension, depicting a dot product between sparse vectors on the top (2a,2b,2c) and a dot product between continuous vectors within our continuous tensor abstraction in the middle (2d,2e,2f). Notably, the syntax of the two versions of our dot product is identical, yet their semantics differ. The for loop in 2c iterates over a discrete (integer) domain, while the for loop in 2f traverses a continuous (real) domain. In 2d, the non-zero data is positioned at specific spots (i = 1.0, 3.0, 4.1, 5.1 for x[i]) within the continuous domain. We term these specific spots as *pinpoint* coordinates, and the corresponding continuous tensor is referred to as a pinpoint tensor. Pinpoint tensors with real coordinates naturally extend existing sparse tensors into the continuous domain. Their physical storage structures remain largely consistent, with the primary distinction being the storing of real numbers in the crd array. Although there are infinitely many real numbers between [0.0,9.0], the pinpoint coordinates where effectual computation takes place, i.e., multiplication between non-zeros, are finite (e.g., i=3.0 and i=5.1). Let's further extend this concept to scenarios where data is situated within intervals, rather than at pinpoint coordinates.

Figures 2g, 2h, and 2i present another variation of a continuous dot product. The key distinction in this example, in comparison to the previous one, lies in the placement of non-zero data at *interval* coordinates. Within an interval, there are infinitely many pinpoint coordinates, making it impractical to directly sum up the infinitely many values. In such cases, we define a reduction

operation as an integral $\int$ (indicated by d(i) in Figure 2i); $s = \int_{0.0}^{9.0} x_i \cdot y_i \cdot di$. This example can be employed in temporal database queries, such as "What is the total duration during which both hotel rooms x and y are booked within the time range [0.0, 9.0]?"

3 PIECEWISE-CONSTANT TENSOR

Naively iterating over a continuous domain is impossible as there are infinitely many real numbers. To process an infinite amount of data and computation with finite resources, we introduce the fundamental assumption within our abstraction.

All continuous tensors must satisfy a piecewise-constant property.

We say that a **tensor** A is a concrete data structure in memory whose value at index i can be obtained by evaluating the program A[i]. Note that this definition does not preclude real-valued indices. The values of our tensor A at each index i can be expressed as a mathematical function $f(i) = $ A[i]. We say that a tensor is **piecewise constant** when its values are piecewise-constant over i, meaning that the value of a tensor can be expressed as $f(x) = \sum_n V_n * [\![x \in I_n]\!]$, where I_n and V_n represent the interval and constant of the nth piece, respectively. $[\![x]\!]$ is the Iverson bracket; if x is true, then $[\![x]\!] = 1$, otherwise $[\![x]\!] = 0$. In this definition, the intervals I_n must satisfy two rules:

R1 Intervals are pairwise disjoint ($\forall_{n \neq m}, I_n \cap I_m = \emptyset$).
R2 The union of intervals covers the entire real domain ($\bigcup_n I_n = \mathbb{R}$).

In Figure 2g, the value of a tensor $x[i]$ is expressed as a piecewise-constant function $f_x(i) = $

$$0 * [\![i \in [-\infty, 1)]\!] + 1 * [\![i \in [1, 3]]\!] + 0 * [\![i \in (3, 4.1)]\!] + 2 * [\![i \in [4.1, 5.1]]\!] + 0 * [\![i \in (5.1, \infty)]\!] = \begin{cases} 0 & -\infty \le x < 1 \\ 1 & 1 \le x \le 3 \\ 0 & 3 < x < 4.1 \\ 2 & 4.1 \le x \le 5.1 \\ 0 & 5.1 < x < \infty \end{cases}$$

Traditional tensors on the integer domain define a discrete realm of computation. A natural extension of traditional tensors into the continuous domain is pinpoint tensors (Figure 2d) wherein each non-zero value is anchored to a real coordinate. This can be further extended to interval tensors (Figure 2g), where non-zero values span across continuous intervals. We categorize both pinpoint and interval tensors as piecewise-constant tensors.

Def1 A **pinpoint tensor** can be represented as a piecewise-constant function $f(x) = \sum_n V_n \cdot [\![x \in I_n]\!]$, subject to the condition that $\forall n : V_n \neq 0 \rightarrow |I_n| = 0$, where every piece with a non-zero value spans a closed interval with identical endpoints.
Def2 An **interval tensor** can be represented as a piecewise-constant function $f(x) = \sum_n V_n \cdot [\![x \in I_n]\!]$, subject to the condition that $\exists n : V_n \neq 0 \rightarrow |I_n| > 0$. In this representation, at least one piece with a non-zero value spans an interval of length greater than zero.

While functions represent abstract mappings between mathematical sets, tensors are concrete data structures in memory with well-designed implementations. While many numerical libraries have been studied to support piecewise-constant functions [8, 28, 41], to the best of our knowledge, this paper is the first to introduce the piecewise-constant property into the tensor programming world. Furthermore, we can generate efficient code for continuous tensors. Since tensor programming is already widespread and employs familiar paradigms, users can easily write programs and explore diverse loop scheduling strategies for performance enhancement. Moreover, our abstraction facilitates intuitive reasoning about programs that were previously challenging in traditional tensor programming. We demonstrate the utility of piecewise-constant tensors in representing various applications involving geometries or genomic operations in Section 8.

4 CORE LANGUAGE

Our language maintains the common syntax of standard tensor and loop abstractions. Notably, this work expands upon this syntax to enable loops over the continuous domain and access with real indices. Specifically, we extend Finch's language [4] to accommodate real indices. While Finch has additional features, our focus in this paper remains on foundational syntax elements such as for, if, let, and assignment (=, +=) statements, as well as tensor access (A[]) and index expressions.

Our program comprises imperfectly nested loops, with the restriction that it includes only a single assignment statement (=,+=). This statement allows tensors to appear exclusively on either the left-hand side or the right-hand side, but not both simultaneously. Within such a loop nest, an assignment statement is enclosed by one or more for and if statements, defining the domain where the function will be updated.

```
1  for i = -∞:∞
2    if Mask[i] # Pinpoint Tensor
3      for j = -∞:∞
4        Z[i] += A[i+j]*B[j]*d(j)
```

Fig. 3. Example Code. Continuous loops colored in purple.

We represent the every continuous tensor A[i] as the piecewise-constant function $f_A(i)$. We define the semantics of continuous loops by partitioning loops into constant regions. As a walkthrough example in the next several sections, we will use the program in Figure 3, which performs 1D convolution only on pinpoint regions specified by Mask.

4.1 Validity of the program

We have identified that not all programs in our language are valid. Although the next few sections define the semantics of a program precisely, we summarize here:

- A continuous loop is executable if, after partitioning the loop into a finite number of piecewise regions, all expressions in each region must be *rewritable* to constant expressions with respect to the loop index, or each region must have a width of 0 (representing a pinpoint $[x, x]$).

We can draw several observations from this property. One implication is that:

1. Any index expression $g(i)$ used in tensor access A[g(i)] must have a well-defined inverse function g^{-1} such that the intervals $g^{-1}(I_n)$ preserve piecewise properties (**R1** and **R2**). Violation of this rule (e.g., A[i^2] or A[$\sin(i)$]) prevents the partitioning of the iteration space into a finite number of disjoint regions.

Another implication is that

2. Index expressions outside of tensor access must be evaluated at a finite number of pinpoint coordinates. Violation of this rule (e.g., for i = 0.0:10.0; A[i] += i; end) results in evaluating index expressions over infinitely many pinpoints within a continuous loop.

We note the distinction between what could be executed in theory and in practice due to the limitations of our term rewriting implementation. For example, for i = 0.0:10.0; if i*i-2*i+1==0: A[i] = 1; end end) can theoretically be evaluated exclusively at a pinpoint $i = 1.0$ but may result in evaluation over infinitely many pinpoints within a continuous loop if our implementation cannot rewrite the equation to solve for the root.

4.2 Piecewise Loop Transformation

The primary challenge lies in converting the infinite iteration space into a finite one to execute on hardware. The iteration space is a set of all index points that the loops traverse in the program. This iteration space spans real numbers, constituting an infinite set where not every real index can be enumerated. The iteration space of Figure 3 is defined as $IS = \{(i, j) | -\infty \leq i, j < \infty, i, j \in \mathbb{R}\}$.

We establish the program semantics by partitioning the iteration space into the smallest possible number of disjoint regions, ensuring that all right-hand side accesses remain constant within each

region. This process is formalized as the **piecewise loop transformation**, which allows us to assign meaning to the continuous loop. To construct a region for each index i, we compute the intersection of every n-tuple in the Cartesian product over n sets of pieces, where n represents the number of tensor dimensions accessed by index i. This approach is valid for partitioning into disjoint regions while covering the entire iteration space, as it guarantees that all pieces in input tensors are already disjoint and cover the entire space, as defined by (**R1**) and (**R2**).

When handling an arbitrary index expression $g(i)$ accessing the tensor, we replace `A[g(i)]` with a new access using the pure index `A'[i]`. The value of this new access can be expressed as a piecewise function with a transformed interval $g^{-1}(I_n)$: $f_{A'}(x) = \sum_a V_a^A \cdot [\![x \in g^{-1}(I_a^A)]\!]$ where the piecewise function of original access `A[g(i)]` is represented as $f_A(g(x)) = \sum_a V_a^A \cdot [\![g(x) \in I_a^A]\!]$.

```
1  for m = 0:n_Mask
2    for a = 0:n_A
3      for b = 0:n_B
4        # Disjoint loop partitioning.
5        # RHS access replaced with constants.
6        for i ∈ Mask.itvl[m]
7          if Mask.val[m]
8            for j ∈ (A.itvl[a]-i) ∩ B.itvl[b]
9              Z[i] += A.val[a]*B.val[b]*d(j)
```

Fig. 4. After piecewise loop transformation. Continuous loops are colored in purple.

In cases where multiple indices access the same dimension, like `A[i+j]` in Figure 3, we prioritize the index (`j`) in the deepest loop nest, as other indices (`i`) can be treated as constants once evaluated in outer loops. Figure 4 illustrates the partitioned iteration space of the example program. Note that every right-hand side access is replaced with the constant value corresponding to its piece. In line 8, we intersect two intervals $g^{-1}(I_a^A) \cap I_b^B$ that an index j accesses, where g^{-1} shifts an interval by i.

4.3 Pinpoint Specialization

Even after substituting tensor accesses with constant values, index expressions such as `A.itvl[a]-i` in line 8 of Figure 4 may still persist in the program, rendering them uncomputable within a continuous loop. To address this issue, we further partition the disjoint regions according to pinpoint tensors into two cases by its definition (**Def1**): (1) pinpoint coordinates with non-zero values, and (2) interval coordinates with a value of zero.

```
1  for m = 0:n_Mask
2    for a = 0:n_A
3      for b = 0:n_B
4
5        if Mask.itvl[m] is pinpoint:
6          let i = Mask.itvl[m].left
7            if Mask.val[m]
8              for j ∈ (A.itvl[a]-i) ∩ B.itvl[b]
9                Z[i] += A.val[a]*B.val[b]*d(j)
10
11       # Can be dead code eliminated
12       elseif Mask.itvl[m] is interval:
13         for i ∈ Mask.itvl[m]
14           if 0 # Always false
15             for j ∈ (A.itvl[a]-i) ∩ B.itvl[b]
16               Z[i] += A.val[a]*B.val[b]*d(j)
```

Fig. 5. After pinpoint specialization on Mask, the interval case can be eliminated as dead code. Continuous loops are highlighted in purple.

In the pinpoint case, where the loop exclusively contains pinpoint coordinates, we replace the continuous loop with a `let` statement, enabling the evaluation of remaining index expressions on a single pinpoint. Conversely, in the interval case, we may able to dead-code eliminate or zero-annihilate the continuous loops, hoping that the program's structure can facilitate the deletion of continuous loops. Thus, the example program would be transformed into Figure 5 after specializing over the pinpoint tensor Mask. The interval case can be eliminated because the loop body won't execute due to a false `if` condition.

4.4 Converting Continuous Loops

For the remaining continuous loops, we categorize them based on the left-hand side (LHS) access to the loop index. If the LHS includes the continuous loop index and each access statement follows the format `for i ∈ Interval; A[i] = constexpr` with respect to `i`, we can eliminate the continuous loop and substitute index with the corresponding ranges from the loops, assigning the interval and constant of the output piece as `A[Interval] = constexpr`.
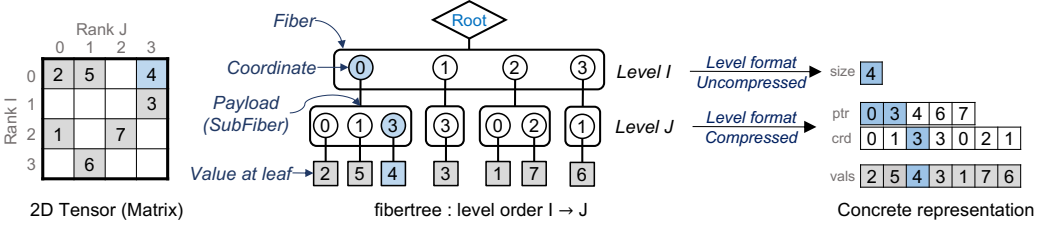
Fig. 7. A sparse matrix with dimensions I and J, its fibertree abstraction and concrete representations.

On the other hand, if the LHS does not include the loop index, an important consideration arises regarding reduction operations within a continuous domain, which requires careful attention to the semantics of reduction operators. We introduce two distinct modes for the += operator within our continuous tensor abstraction: summation ($\sum$) and integration ($\int$).

Continuous summation ($\sum$) mirrors traditional tensor programming models, aggregating values from pinpoint coordinates during iteration. However, summation over intervals yields an infinite value as there are infinitely many pinpoints. The integration ($\int$) mode is designed for interval tensors, enabling integration across all values within an interval. This integration is essentially the product of the right-hand side (RHS) value multiplied by the length of the interval, since the RHS remains constant. Syntactically, our language distinguishes between two reduction modes based on the presence or absence of d(i) in the expression, where i represents the loop index. We provide further elaboration on implementation details and reduction operators beyond += in Section 7.

```
1  for m = 0:n_Mask
2    for a = 0:n_A
3      for b = 0:n_B
4        if Mask.itvl[m] is pinpoint
5          let i = Mask.itvl[m].left
6            if Mask.val[m]
7              Z[i] = A.val[a] * B.val[b] *
8                length((A.itvl[a]-i) ∩ B.itvl[b])
```

Fig. 6. After removing continuous loop based on integration reduction: No continuous loops.

Figure 6 illustrates our final program by demonstrating the conversion of the last continuous loop into integration reduction mode.

Although Figure 6 provides executable code, it still includes numerous intervals involving redundant computations. To mitigate this, we avoid traversing all tuplewise pieces ($O(n_{Mask} * n_A * n_B)$), optimizing the program's efficiency in practice. In the remaining sections of this paper, we will address two key implementation challenges: (1) How to efficiently store continuous piecewise-constant tensors in memory (detailed in Section 5), and (2) How to efficiently iterate over an infinite number of real indices using a continuous loop (detailed in Sections 6 and 7). In Section 8, we further illustrate how this assumption's versatility extends to modeling a wide range of applications within an infinite space.

5 CONTINUOUS TENSOR STORAGE

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 [18] within TACO and subsequently refined and formalized as the *fibertree* abstraction [54].

Figure 7 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. The order of the levels signifies the data layout, such as row-major or column-major.
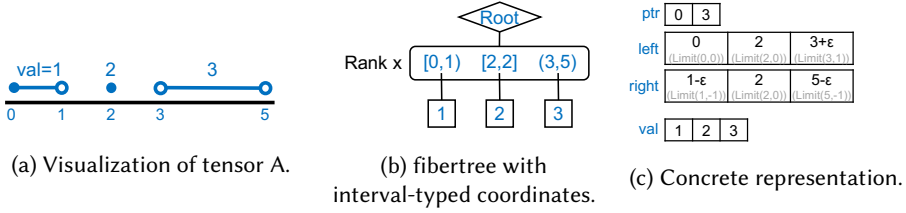
(a) Visualization of tensor A.

(b) fibertree with interval-typed coordinates.

(c) Concrete representation.

Fig. 8. fibertree with an interval-typed coordinate of piecewise-constant continuous tensor A

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)]`

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.

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 coordinate interval 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. In Figure 7 (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 a more detailed explanation of fibertrees, refer to [43, 54].

5.2 Interval Coordinates in fibertrees

We introduce interval-typed coordinates in a fibertree to effectively capture piecewise-constant properties. Figure 8 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 $[2, 2]$, as depicted in Figure 8b. We account for the inclusiveness (open or closed) of each endpoint within the interval type, leading to four distinct interval subtypes. Similar to how traditional level formats operate on fibers with integer coordinates, we designed several level formats for fibers with interval coordinates.

5.3 `Limit`: A Number Type to Represent Inclusiveness of Endpoints with ϵ.

To represent the four distinct inclusiveness categories for intervals, we propose a new number type `Limit` to encode interval endpoints. The `Limit` type integrates the concept of an infinitesimal number, denoted as ϵ, which is smaller than any standard positive real number but nonzero in size. This allows us to treat all intervals as closed, regardless of inclusiveness, as we can emulate an open endpoint with a closed interval by adding or subtracting ϵ as appropriate, as shown in Table 1.

Figure 9 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`). Note that ϵ is not to be confused with machine epsilon in floating point representations. While real numbers $\in \mathbb{R}$ are infinite and continuous, computer number systems are finite and discrete. Consequently, representation errors, leading to roundoff errors, are inherent in computer number systems.
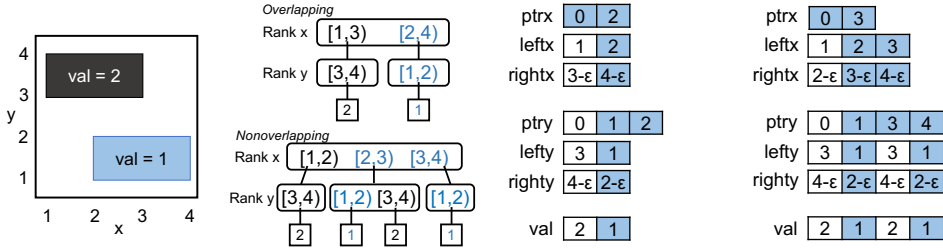
In Figure 8c, we present a concrete representation of the fibertree with interval coordinates, encapsulating the number and inclusiveness pair using the `Limit` type. This representation stores each endpoint within the `left` and `right` arrays, assuming closed intervals for all interval coordinates.

```
1   # Definition of Limit.
2   struct Limit{T}
3       val::T
4       eps::Int8 # (+ε, 0, -ε) = (+1, 0, -1)
5   end
6
7   # Operations between two Limits.
8   (+)(x::Limit, y::Limit)::Limit = Limit(x.val + y.val, min(max(x.eps + y.eps, -1), +1))
9   (-)(x::Limit, y::Limit)::Limit = Limit(x.val - y.val, min(max(x.eps - y.eps, -1), +1))
10  (<)(x::Limit, y::Limit)::Bool = x.val < y.val || (x.val == y.val && x.eps < y.eps)
11  (<=)(x::Limit, y::Limit)::Bool = x.val < y.val || (x.val == y.val && x.eps <= y.eps)
12  (==)(x::Limit, y::Limit)::Bool = x.val == y.val && x.eps == y.eps
13
14  # Operations between normal number and Limit.
15  numeric_types = [Int8, Int16, Int32, Int64, Float32, Float64]
16  for S in numeric_types
17      (+)(x::Limit, y::S)::Limit = x + Limit(y, 0)
18      (-)(x::Limit, y::S)::Limit = x - Limit(y, 0)
19      (<)(x::Limit, y::S)::Bool = x < Limit(y, 0)
20      (<=)(x::Limit, y::S)::Bool = x <= Limit(y, 0)
21      (==)(x::Limit, y::S)::Bool = x == Limit(y, 0)
22  end
```

Fig. 9. Implementation of `Limit` type. Infinitesimal number is represented by eps field stored in `Int8` type.

(a) Visualization of tensor A. (b) fibertree with overlapping and nonoverlapping. (c) Concrete representation with overlapping. (d) Concrete representation with nonoverlapping.
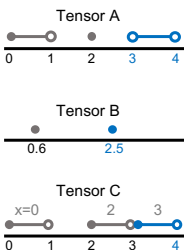
Fig. 10. **(b)** fibertree representation of a 2D continuous tensor $A_{x,y}$. **(c)** The level format on dimension x allows overlapping intervals ([1,3] and [2,4]). **(d)** Dimension x is stored in a non-overlapping manner.

Our compilation process generates a code for interval arithmetics, such as interval intersection, at a symbolic level. When executing the generated code, all interval arithmetics are performed on the `Limit{T}` type, which can introduce numeric rounding errors. Our `Limit{T}` type is designed to accommodate various numeric types T, such as `Limit{Float64}`, to encode endpoints of intervals. The choice of an appropriate numeric type T (e.g `Float32` and `BigFloat` for arbitrary-precision) depends on the desired precision for the application. This flexibility allows users to choose the most suitable computer number system for their specific applications. To ensure a fair comparison, we used the same numeric type for real indices as each baseline in the case studies (Section 8).

5.4 Overlap and Ordering of Intervals.

Overlapping Intervals. When considering the concrete representation of piecewise-constant tensors, it's essential to account for the potential existence of overlapping intervals. While overlap is not a major concern in the 1D case, it becomes a significant consideration in multi-dimensional piecewise-constant tensors.

Figure 10a provides a visual depiction of a 2D continuous tensor, denoted as $A_{x,y}$, comprising two distinct pieces, each defined within box-shaped ranges. Figures 10c and 10d show two different concrete representations originating from the same fibertree, with a specified level order of $x \rightarrow$

	Limit level format	Homogeneous level format	Pinpoint lvlformat	Regular level format
Tensor A left	0 Limit(0,0) / 2 Limit(2,0) / 3+ε Limit(3,1)	N/A	N/A	N/A
right	1-ε Limit(1,-1) / 2 Limit(2,0) / 4-ε Limit(4,-1)			
Tensor B left	0.6 Limit(0.6,0) / 2.5 Limit(2.5,0)	left 0.6 2.5 lclose True	crd 0.6 2.5	lclose True rclose True
right	0.6 Limit(0.6,0) / 2.5 Limit(2.5,0)	right 0.6 2.5 rclose True		stride 1 len 0
				x 0.6 2.5
Tensor C left	0 Limit(0,0) / 2 Limit(2,0) / 3 Limit(3,0)	left 0 2 3 lclose True	N/A	lclose True rclose False
right	1-ε Limit(1,-1) / 3-ε Limit(3,-1) / 4-ε Limit(4,-1)	right 1 3 4 rclose False		stride 1 len 1
				x 0 2 3

Fig. 11. Level formats across three distinct 1D continuous tensors.

y. It's important to note that, although the concrete representations in 10c and 10d differ, the two fibertrees in 10b are semantically equivalent. We've represented them differently for visual clarity and ease of understanding. The choice of level format applied to rank *x* distinguishes these representations. In 10c, the level format allows intervals in dimension *x* to overlap in memory ([1,3) and [2,4)), whereas in 10d, overlap is not permitted, resulting in disjoint intervals ([1,2), [2,3), and [3,4)). Permitting overlapping intervals offers advantages in terms of memory space conservation. However, when achieving a continuous loop along rank *x*, the generated code becomes more complex compared to iterating over a non-overlapping representation. Figures 10c and 10d are valid representations, but our current compiler only accepts non-overlapping level formats.

Ordered Intervals. A level format is ordered if the intervals within a fiber are stored in a specific order. The endpoints of intervals can be ordered in one of three ways: (1) sorted by the left endpoint, (2) sorted by the right endpoint, or (3) sorted lexicographically by both. Note that when a level is not overlapped, these three orders result in identical representations. Our compiler implementation currently only accepts ordered levels and does not support unordered ones.

5.5 Optimized Representation

This subsection outlines optimized representations for storing tensors with specific patterns, including pinpoint coordinates, intervals with consistent inclusiveness, and regular intervals. Such patterns allow fibers to be represented with alternative level formats, resulting in notable benefits in terms of memory efficiency and the complexity of the generated code. Figure 11 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.

Homogeneous Intervals. The need to store pairs of (number, inclusiveness) for every endpoint may vary, depending on the inclusiveness of these intervals. 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* intervals. Tensor B and C in Figure 11 are homogeneous but a 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, as illustrated in lclose and rclose in Figure 11.

Pinpoint Coordinates. If the level only comprises 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 11.

Fig. 12. Describing a pattern with Looplets: Single interval on the left, densely packed intervals on the right.

Regular Intervals. A set of intervals is considered *regular* if all endpoints of intervals in the set satisfy a specific relation: $[stride \cdot x, stride \cdot x + len)$ where *stride* and *len* are constants. Such patterns are commonly used, for example, in a uniform grid or 3D voxels. The far-right column in Figure 11 illustrates how we can optimize the representation for such regular intervals in tensors B and C. In tensor C, all intervals have the same length and are distributed regularly, precisely represented by the right half-open interval $[1 \cdot x, 1 \cdot x + 1)$. Pinpoint coordinates in tensor B can also be represented in a regular level format, as all pinpoints can be expressed using the closed interval $[1 \cdot x, 1 \cdot x + 0]$. For regular intervals, we can store only an array of x with metadata instead of storing all endpoints.

6 LOOPLETS

In this section, we provide background information on Looplets, which serve as the core intermediate representation for code generation in the Finch compiler [2]. While fibertrees describe tensors as a tree of ranks, Looplets describe each fiber in a rank as a tree of ranges. The hierarchical decomposition allows the compiler to resolve interactions between different structures, such as banded matrices, run-length encoded images, or most critically, pinpoint levels and interval levels. For example, one such interaction is an intersection. Looplets provide a mechanism for interpreting the concrete memory representation as the full tensor (i.e. including zeros). The compiler uses progressive lowering to traverse and interact tensors with each other as represented by Looplets. Repeated rewriting and optimization passes at each step accounts for mathematical properties such as sparsity.

6.1 Background on Looplets

Although Looplets were originally conceived within the integer domain, extending the concept to the continuous domain is a natural transition that requires little alteration. The following provides background information on four core types of Looplets designed for describing value patterns, enabling precise descriptions of continuous patterns:

- **Run** : The Run Looplet describes a run of the same payload (scalars or subfibers).
- **Sequence** : The Sequence Looplet concatenates a constant number of child Looplets, enabling the creation of intricate patterns by joining multiple Looplets together. Sequences are analogous to blocks of code, with each child Looplet a statement in the block.
- **Stepper** : The Stepper Looplet concatenates a variable number of instances of the same child Looplet, allowing for more complex repetitive patterns. Steppers are analogous to for-loops, where the child looplet is the body of the loop.
- **Phase** : The Phase Looplet indicates the range spanned by a child Looplet, used to indicate the boundaries imposed by Steppers and Sequences.

To aid in understanding Looplets, one can draw connections between Looplets and Regular Expressions (Regex). In this analogy, a Run Looplet corresponds to a character in Regex, a Sequence

Looplet corresponds to Regex concatenation (RS) of subRegex R and S, and a Stepper Looplet corresponds to the Kleene star R^* in Regex, allowing for a more intuitive grasp of the Looplets.

6.2 Describing Continuous Tensors with Looplets

Figure 12 illustrates two examples of pattern descriptions when Looplets are applied to the continuous domain. In the left figure, you can see a piece-wise constant vector with a single interval. The Looplets divide the entire space into three chunks: runs of zero, runs of one, and runs of zero. Each run is wrapped by a Phase Looplet, indicating the range it spans. Then, a Sequence Looplet concatenates these three child Looplets, creating the desired pattern. In the right figure, we have a pattern where uniform intervals are tightly distributed. To capture the repeating non-zero intervals, a Stepper Looplet is employed. The Sequence Looplet is then used to concatenate a phase of zero, the Stepper Looplet, and another phase of zero, forming the complete pattern.

Looplets must represent all tensor values, not just the stored or nonzero ones. This is because even zero regions of the tensor may interact with programs or tensors to affect the output. Since Looplets must cover the entire space completely, continuous looplets may need to use an infinitesimal value ϵ to define continuous intervals with open endpoints. This implies that the range of intervals filled with zeros (depicted in gray) begins and ends at points infinitesimally close to the boundaries of non-zero intervals (depicted in blue). For example, in the left figure, the leftmost zero interval terminates at $3.1 - \epsilon$, which is slightly before the start of the adjacent non-zero interval.

We complete the description of our continuous tensor formats by representing each level format with Looplets. In Figure 13, we present a step-by-step process of how our abstraction interprets a 2D continuous tensor as a fibertree, level format, and Looplet nest. In this case, the 2D continuous tensor $A_{x,y}$ features pinpoint coordinates on rank y and interval coordinates on rank x, with the fibertree's layout structured as $y \rightarrow x$. Figure 13a visually demonstrates the derivation of the concrete representation by applying a level format for each rank in the fibertree.

Actual Looplets need to include user-specified iteration code in order to relate concrete memory to theoretical representations. Figure 13b presents the complete Looplet nest for rank y. The color coding corresponds to the rank y in the visualization in left of Figure 13a. It begins by taking a pos as input, representing the position of the parent fiber (Root), which is always zero. Each field within the Looplet is populated with the concrete representation of rank y. Notably, a stepper contains two additional fields aside from the body, namely, seek (Line 6) and next (Line 15). The seek field initializes variables for a starting index 'j', while the next field advances the state to the next Looplet within the stepper. It's essential to observe that, as rank y comprises pinpoint coordinates, the stopping point of the first phase in the sequence (Line 10) is adjusted by subtracting ϵ from the pinpoint coordinate crd[p]. Similarly, Figure 13c defines the Looplet for interval patterns on rank x, completing the level description for this rank. In summary, the compiler uses Looplets to interpret concrete memory representations in each level as complete tensors.

7 CODE GENERATION

In this section, we describe how our compiler transforms continuous loops into efficient and executable code. Our compiler is built upon Finch [2], which was originally designed for sparse tensor computations in the integer domain using Looplets. We provide insight into Finch's background and detail the adaptations we made to extend its capabilities into the continuous domain. Our compiler takes two key inputs: (1) a user's code[2] written using continuous tensor abstractions, and (2) Looplet descriptions for each tensor. With these inputs, the compiler generates executable Julia code. A key aspect of compiler is the mechanism for lowering the Looplet on a loop.

---

[2]We use Finch's input tensor programming language as described in [4]

(a) (1) 2D continuous tensor $A_{x,y}$ and its Looplets, (2) its fibertree, and (3) concrete representation.

```
1  # pos = position of Root fiber (0)
2  Sequence(
3    Phase(
4      stop = crd[ptr[pos+1]-1],
5      body = Stepper(
6        seek(j) = (p = search(j, crd,
7                        ptr[pos], ptr[pos+1])),
8        body = Sequence(
9          Phase(
10           stop = crd[p]-ε,
11           body = Run(0)),
12         Phase( # pinpoint [crd[p],crd[p]]
13           stop = crd[p],
14           body = Run(getPayload(p)))),
15       next = (p += 1))),
16   Phase(
17     stop = +Inf,
18     body = Run(0)))
```

```
1  # pos = position of fiber on Rank x
2  Sequence(
3    Phase(
4      stop = right[ptr[pos+1]-1],
5      body = Stepper(
6        seek(j) = (p = search(j, right,
7                        ptr[pos], ptr[pos+1])),
8        body = Sequence(
9          Phase(
10           stop = left[p]-ε,
11           body = Run(0)),
12         Phase( # interval [left[p],right[p]]
13           stop = right[p],
14           body = Run(val[p]))),
15       next = (p += 1))),
16   Phase(
17     stop = +Inf,
18     body = Run(0)))
```

(b) **Rank y** : Looplets of a **pinpoint level** on a concrete representation using `crd` array. The color coding corresponds to the rank y in the **visualization in left of (a)-(1)**.

(c) **Rank x** : Looplets of a **interval level** on a concrete representation using `left` and `right` array. The color coding corresponds to the rank x in the **visualization in top of (a)-(1)**.

Fig. 13. Example on 2D continuous tensor storage. (a) A concrete representation derived from fibertree using level formats. (b,c) Complete Looplets description on rank y and x using their concrete representations above.

7.1 Compiler Pass for Looplets

Background. In Finch, each Looplet type defined within a `for` loop statement is lowered by a corresponding compiler pass. Most of the compiler passes can be applied to continuous Looplets without modification, with the exception of the Stepper. Figure 14 provides background information through a visualization of how each pass reduces the Looplets within a continuous loop.

- **Phase** : This pass lowers continuous loops associated with Phase Looplets (Figure 14a, 14b, and 14c). It intersects Phase ranges with the loop's range, generating code to check for intersections with a length $\geq$ 0 (Lines 3-4 in Figure 14c). Our compiler emits symbolic expressions of intersection with max and min, though we've used ∩ for brevity (lines 1-2).
- **Run** : This pass lowers Run Looplets, as illustrated in Figure 14d, 14e, and 14f. In this case, the Run Looplet is essentially replaced with a constant scalar value.
- **Sequence** : This pass lowers the continuous loop associated with Sequence Looplets, concatenating multiple child Phases. As the Sequence's exact range is determined at runtime, this pass generates all feasible combinations of child Phases within each Sequence. In Figure 14g, 14h,

(a) Phase: Visualization.

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

(b) Phase: Source program.

```
1  # A ∩ B = [max(A.start, B.start),
2  #           min(A.stop, B.stop)]
3  intersect = [st,en] ∩ rangeA ∩ rangeB
4  if intersect.start <= intersect.stop
5    for i = intersect.start:intersect.stop
6      s += bodyA[i] * bodyB[i]
```

(c) Phase: Lowered program.

(d) Run: Visualization

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

(e) Run: Source program.

```
1  for i = st:en
2    s += scalarA * scalarB
```

(f) Run: Lowered program.

(g) Sequence: Visualization.

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

(h) Sequence: Source program.

```
1  for i = st:en
2    s += PhaseA1[i] * PhaseB1[i]
3  for i = st:en
4    s += PhaseA1[i] * PhaseB2[i]
5  for i = st:en
6    s += PhaseA2[i] * PhaseB1[i]
7  for i = st:en
8    s += PhaseA2[i] * PhaseB2[i]
```

(i) Sequence: Lowered program.

(j) Stepper: Visualization.

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

(k) Stepper: Source program.

```
1   call seekA(st) # find bodyA contains st
2   call seekB(st) # find bodyB contains st
3   i = st
4   while i <= en
5     curr = [i,en] ∩ bodyA.range ∩ bodyB.range
6     for i2 = curr.start:curr.stop
7       s += bodyA[i2] * bodyB[i2]
8     if (curr.stop==bodyA.stop) call nextA
9     if (curr.stop==bodyB.stop) call nextB
10    i = curr.stop + ϵ
11  end
```

(l) Stepper: Lowered program.

Fig. 14. 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.

and 14i, both tensor A and B have two Phases, resulting in optimized code for all four possible pairs, e.g., (A1, B1), (A1, B2), (A2, B1), (A2, B2).

- **Stepper** : This pass lowers Stepper Looplets, which repeat the child Looplet pattern 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. Figure 14j, 14k, and 14l illustrate this process. The seek field contains code to fast-forward the stepper to the first body containing the start (st) of the for loop range (Lines 1-2 in Figure 14l). The while loop continues until it reaches the end (en) of the continuous for loop where the variable i tracks the current start (Lines 3-4). Within the 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, i, indicating the next starting point, is set to the end of the intersected region (curr.stop) incremented by ϵ (Line 10).

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 in Figure 14). A specific compiler pass is selected based on the type of Looplet, with a focus on lowering the

(a) Visualization.

```
1  a = Sequence(Phase(La-ε, Run(0)),
2                Phase(Ra, Run(Va)))
3  b = Sequence(Phase(Lb-ε, Run(0)),
4                Phase(Rb, Run(Vb)))
5  for i = st:en
6    s += a[i] * b[i] * d(i)
```

(b) Continuous Loop: $s = \int_{st}^{en} a_i * b_i * di$

```
1  A1B1 = [st,en] ∩ [-∞,La-ε] ∩ [-∞,Lb-ε]
2  if A1B1.start <= A1B1.stop
3    for i = A1B1.start:A1B1.stop
4      s += 0 * 0 * d(i)
5  A1B2 = [st,en] ∩ [-∞,La-ε] ∩ [Lb,Rb]
6  if A1B2.start <= A1B2.stop
7    for i = A1B2.start:A1B2.stop
8      s += 0 * Vb * d(i)
9  A2B1 = [st,en] ∩ [La,Ra] ∩ [-∞,Lb-ε]
10 if A2B1.start <= A2B1.stop
11   for i = A2B1.start:A2B1.stop
12     s += Va * 0 * d(i)
13 A2B2 = [st,en] ∩ [La,Ra] ∩ [Lb,Rb]
14 if A2B2.start <= A2B2.stop
15   for i = A2B2.start:A2B2.stop
16     s += Va * Vb * d(i)
```

(c) Code after Sequence and Phase Pass.

```
1  A1B1 = [st,en] ∩ [-∞,La-ε] ∩ [-∞,Lb-ε]
2  block() #Empty
3
4  A1B2 = [st,en] ∩ [-∞,La-ε] ∩ [Lb,Rb]
5  block() #Empty
6
7  A2B1 = [st,en] ∩ [La,Ra] ∩ [-∞,Lb-ε]
8  block() #Empty
9
10 A2B2 = [st,en] ∩ [La,Ra] ∩ [Lb,Rb]
11 if A2B2.start <= A2B2.stop
12   for i = A2B2.start:A2B2.stop
13     s += Va * Vb * d(i)
```

(d) Code after Simplify Pass.
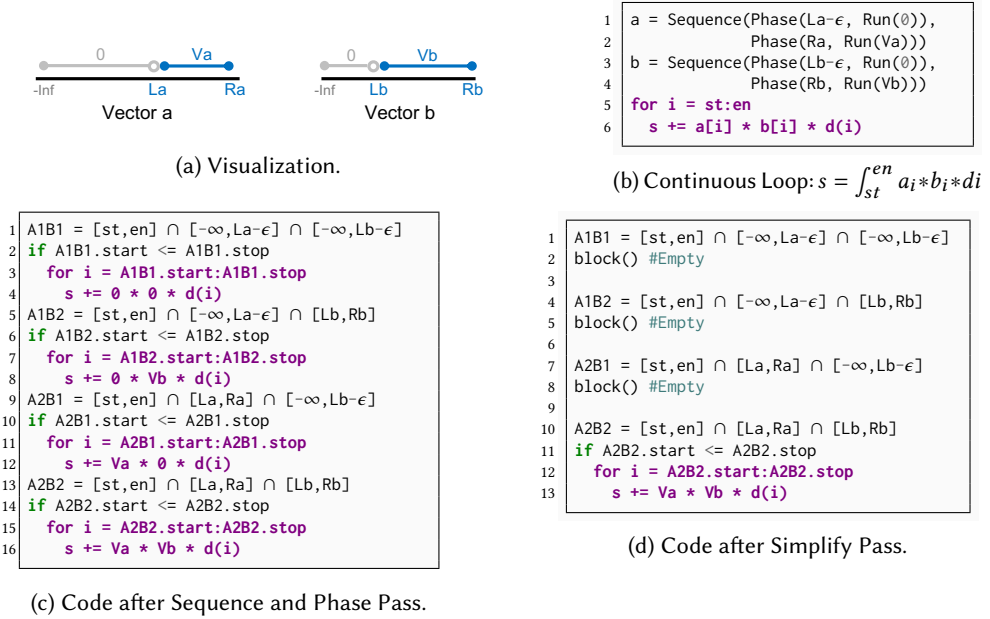
Fig. 16. Code after undergoing multiple compiler passes is depicted below. Continuous loops are highlighted in purple, while the remaining non-purple code represents generated Julia code.

outermost Looplet when nested Looplets are involved. In scenarios where different tensors feature varying outermost Looplet types, tiebreaking rules establish the priority order as Run > Phase > Sequence > Stepper. Any Looplets that remain unprocessed are deferred for subsequent passes.

Extending Looplet lowering to Continuous Space. While most of the Looplet passes were already compatible with continuous loops, a minor adjustment was made in the Stepper pass, where we set i = curr.stop + ϵ instead of the integer domain's i = curr.stop + 1. This ϵ is essentially implemented as Limit(0, Int8(1)) in Figure 9. This increment changes a closed ending point to an open starting point, and vice-versa.

7.2 Compiler Pass for Simplifying Program

```
+(a..., 0, b...) => +(a..., b...)
*(a..., 1, b...) => *(a..., b...)
&&(a..., true, b...) => &&(a..., b...)
||(a..., false, b...) => ||(a..., b...)
a[i...] += 0 => block()
a[i...] *= 1 => block()
a[i...] &= true => block()
a[i...] |= false => block()

*(a..., 0, b...) => 0
&&(a..., false, b...) => false
||(a..., true, b...) => true

if(true, a) => a
if(false, a) => block()
for i=st:en; block() => block()
if(a, block()) => block()
```

Fig. 15. Rewriting Rules in Finch

Background. Finch's simplify pass plays a crucial role in program 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 15. These rules extend beyond basic optimizations like zero-annihilation and 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. Utilizing Finch's rewriting rules, we can leverage the presence of annihilators (e.g., zero in multiplication) within the continuous domain to enhance program efficiency.

Figure 16 offers an illustrative example. When computing $s = \int_{st}^{en} a_i * b_i * di$ with a sequence of phases, the first phase holds a value of zero. After lowering the Looplet following the "Sequence" and "Phase" compiler

passes, the compiler generates four pairs of continuous loops, as shown in purple codes in Figure 16c. Notably, the first three pairs involve multiplying with zero, leaving only the final pair with effective computation. The simplify pass iteratively applies rewriting rules until the program converges, as depicted in Figure 16d. Here, the simplify pass removes the first three pairs, exemplifying the power of leveraging sparsity in continuous domain.

Additional Simplifications Required for Continuous Space. The fundamental challenge behind lowering continuous loops is that it is impossible to execute an infinite number of statements. However, because our tensors are piecewise constant, we can usually simplify the constant regions. For example, repeatedly assigning the same value to a contiguous region of output can be accomplished in one operation by setting that value along an interval. Repeated addition can be accomplished in one operation using integration rules. We add rewrite rules to the simplification pass to accomplish these tasks. However, this also introduces an essential consideration of reduction operator semantics. When the reduction operator is idempotent, like max=, the semantics are clear. However, we must introduce two distinct reduction modes for the += operator within our continuous tensor abstraction: summation ($\sum$) and integral ($\int$) reductions.

Continuous summation ($\sum$) shares a similar semantic with summation in traditional tensor programming models. The summation mode operates exclusively on tensors with pinpoint coordinates, aggregating values from each real (pinpoint) coordinate during the iteration. Consequently, when the piece under consideration is not a pinpoint (i.e., it has a length greater than zero), summation results in an infinite value. In contrast, the integration ($\int$) mode is tailored for any piecewise-constant tensor, allowing for integration across all values within a specified interval. It's important to note that applying integration mode to pinpoint tensors leads to a result of zero, as pinpoints inherently possess a length of zero. In mathematics, particularly in measure theory, summation mode can be thought of as an integral using the counting measure, while integration mode corresponds to an integral using the Lebesgue measure [49].

From a syntactical perspective, our compiler distinguishes between integration and summation reductions based on the presence or absence of d(i) within the expression, with i serving as the index of loop. The presence of d(i) indicates the use of an integral operator (e.g., s += A[i] * d(i)), while its absence indicates the use of summation to pinpoints (e.g., s += A[i]).

In addition to the += operator, we have defined several other reduction operators, including max=, min=, &=, and |= for continuous domain. These operators share a similar semantic with their counterparts in traditional tensor programming models. Unlike +=, they have a consistent semantic on both pinpoints and intervals. These operators concentrate only on the "value" of a piece, as their semantics are independent of the piece's length.

Continuous reduction is achieved through the addition of rewriting rules in the simplify pass, as depicted in Figure 17b. 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 17a presents a list of collapsed expressions categorized by operators.

In integral mode, we utilize the drop_eps function on the Limit type to remove epsilon from the length of the loop interval. In Figure 17a, drop_eps extracts the number(x.val) from Limit type. This is done because integration with Lebesgue measure yields the same result regardless of the inclusiveness of the interval (i.e., $\int_{[0,1]} f(x)dx = \int_{(0,1)} f(x)dx$). Figure 17b below provides an example of how a continuous for loop, using integral mode, is reduced.

In summation mode, 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.

```
# Drop epsilon (e.g., 3+ϵ => 3)
drop_eps(x::Limit) = x.val
drop_eps(x::Number) = x

# Integral mode
collapsed(idx, interval, lhs, +, rhs::∈(d(idx)))=
  # e.g., 3*a[i]*d(i) => 3*a[i]
  newrhs = remove d(idx) from rhs
  newlen = drop_eps(length(interval))
  return (+=)(lhs, (*)(newlen, newrhs))

# Summation mode
collapsed(idx, interval, lhs, +, rhs::∉(d(idx)))=
  return if((==)(length(interval), 0),
            (+=)(lhs, rhs))

# |=, &=, max=, min=
collapsed(idx, interval, lhs, |, rhs) =
  return (|=)(lhs, rhs)
collapsed(idx, interval, lhs, &, rhs) =
  return (&=)(lhs, rhs)
collapsed(idx, interval, lhs, max, rhs) =
  return (max=)(lhs, rhs)
collapsed(idx, interval, lhs, min, rhs) =
  return (min=)(lhs, rhs)
```

(a) Collapsing terms based on operator and context.

```
# Rewriting rule for continuous reduction
(@rule (for idx ∈ interval; body) => begin
  # Every assignments in body
  Rewrite(@rule ((op=)(lhs, rhs)) => begin
    # If assignment is reducible
    if (lhs is reducible &&
        rhs is constant w.r.t loop)
      # Collapse the loop to reduced expression
      collapsed(idx, interval, lhs, op, rhs)
    end
  end)(body)
end)
```

```
# Example from Figure13-(d)
A2B2 = [st,en] ∩ [La,Ra] ∩ [Lb,Rb]
if A2B2.start <= A2B2.stop
  for i = A2B2.start:A2B2.stop
    s += Va * Vb * d(i)

↓↓↓↓↓↓↓↓↓↓↓↓↓

# After rewriting with reduction rule
A2B2 = [st,en] ∩ [La,Ra] ∩ [Lb,Rb]
if A2B2.start <= A2B2.stop
  s += drop_eps(A2B2.stop - A2B2.start) * Va * Vb
```

(b) Continuous reduction rule and an example.

Fig. 17. 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.

```
1. (-∞ <= La-ϵ)
2. (La <= Ra)
3. (-∞ <= Lb-ϵ)
4. (Lb <= Rb)
```

(a) Relationships collected from Looplets in Figure 16a.

```
#A2B2 = [-∞,+∞] ∩ [La,Ra] ∩ [Lb,Rb]
A2B2.start = max(-∞, La, Lb)  # Query1
A2B2.stop = min(+∞, Ra, Rb)   # Query2
if A2B2.start <= A2B2.stop    # Query3
  len = drop_eps(A2B2.stop - A2B2.start)
  s += len * Va * Vb
```

(b) Example code from Figure 17b with iteration domain [-∞,+∞].

```
A2B2.start = max(La, Lb)
A2B2.stop = min(Ra, Rb)
len = drop_eps(A2B2.stop - A2B2.start)
s += len * Va * Vb
```

(c) Optimized code after asking three queries: 1. max(-∞, La, Lb) == max(La, Lb), 2. min(+∞, Ra, Rb) == min(Ra, Rb), and 3. max(La, Lb) <= min(Ra, Rb).

Fig. 18. (a) Information gathered from Looplets. (b,c) Code optimization using bounds query testing with Z3.

7.3 Optimzation: Bound Analysis

Our code generation heavily relies on interval arithmetic, which involves tasks like computing interval intersections or verifying if interval lengths are zero. To boost the efficiency of these interval operations, we leverage compile-time information obtained from the relationships between interval endpoints within nested Looplets. For instance, child Looplets of a Phase Looplet are always bounded by the Phase's range. In a Sequence Looplet, the child Looplet that precedes others also precedes subsequent child Looplets in the sequence. Similarly, in a Stepper Looplet, the Stepper's body always precedes the next body of the Stepper. After gathering this information, we extended Finch to use Z3 [21] to statically prove the validity of specific interval relationships.

This process is illustrated in Figure 18. In the example code before optimization, shown in Figure 18b, we demonstrate how it can be simplified to the form in Figure 18c by testing bounds queries using Z3. Initially, we collect information from nested Looplets (Figure 18a), resulting in

four relationships derived from descriptions of Looplets of vector a and b. Subsequently, when calculating the endpoints of intersection, we reduce the number of comparisons by formulating two queries: (1) `max(-∞, La, Lb) == max(La, Lb)` and (2) `min(+∞, Ra, Rb) == min(Ra, Rb)`. Finally, we ask Z3 (3) `max(La, Lb) <= min(Ra, Rb)` to determine that `A2B2.start <= A2B2.stop` is always true, enabling the compiler to eliminate the need for the `if` condition. This approach streamlines our code and enhances the efficiency of interval arithmetic operations.

8 CASE STUDIES

In this section, we explore diverse applications under the continuous tensor abstraction across four domains: (1) Geospatial search, (2) 3D point cloud convolution, (3) Interpolation in Neural Radiance Field, and (4) Genomic interval operations in Bioinformatics. 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 assess the performance of the code generated by our compiler by comparing it with expertly hand-written libraries. Note that our generated code may differ from its actual algorithm used in the baseline. We provide specific details in each subsection. All experiments are conducted through single-threaded execution on a Macbook Pro M2 Max with 32GB of memory.

Applications	Baseline	Ours	LoC Saving
Radius Search Query	501 lines	5 lines	**100×**
Point Cloud Convolution	2,330 lines	16 lines	**145×**
Trilinear Interpolation in NeRF	82 lines	9 lines	**9×**
Genomic Interval Overlapping Query	206 lines	8 lines	**26×**

Table 2. Comparison of lines of code between the baseline and the program expressed in our abstraction.

8.1 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) [9], and spatial databases [27]. 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 query retrieves all points within a specified box. Conversely, a radius search retrieves all points within a circle centered at (O_x, O_y) with a radius of R.

Figure 19 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 19a illustrates a box query, which outputs IDs of points intersecting with `Box[x, y]`. Figure 19b presents the box and radius search query written in continuous tensor abstraction. In radius search query, it iterates through all points within the radius R (i.e., `Points[Ox+r, Oy+s, id]` where $(r^2 + s^2) \leq R^2$).

Figure 20 presents the performance of our generated code in comparison to Shapely [26], a Python wrapper for GEOS [25], 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 $[0,10000] \times [0,10000]$. In both experiments, we increased the size of the query shape along the x-axis to augment the number of returned output points.

Figure 20a demonstrates that Shapely outperforms our generated code with a geomean of 4.7× on the box search query. This is primarily attributed to Shapely's utilization of an advanced spatial data structure known as STRtree [37], which accelerates search operations. In contrast, our code does

(a) Illustration of box search query described in continuous tensor abstraction. `Out[id]` retrieves the IDs of points that intersect with a specified box.

```
# Box Search Query
for x=-∞:∞   # continuous
 for y=-∞:∞  # continuous
  for id=1:N # discrete
   Out[id] |= Box[x,y] && Points[x,y,id]

# Radius Search Query (Center=(Ox,Oy))
for r=-R:R  # continuous
 for s=-R:R  # continuous
  if (r*r + s*s <= R*R) #within radius R
   for id=1:N # discrete
    Out[id] |= Points[Ox+r,Oy+s,id]
```
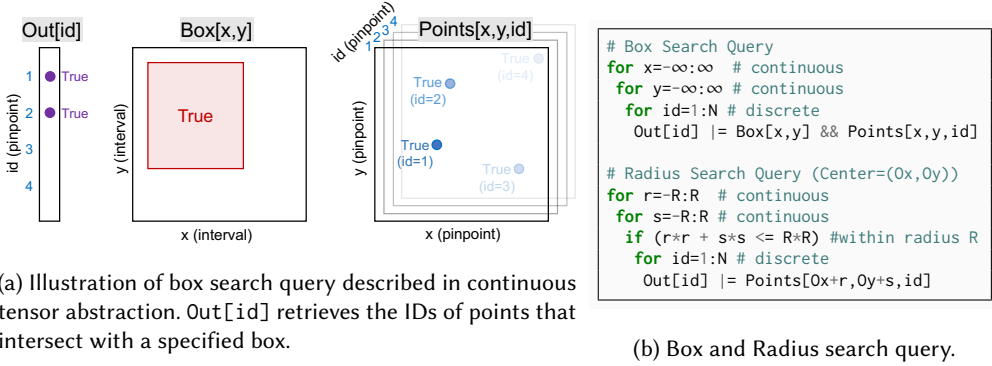
(b) Box and Radius search query.

Fig. 19. Spatial Search Queries in Continuous Tensor Abstraction. (a) Illustration of a box search query. 2D points are represented as a 3D tensor `Points[x,y,id]`, with each point assigned a unique ID. (b) Code for box and radius search query, where the center of the circle is (O_x, O_y) and the radius is R.

(a) Performance comparison of box search query with increasing box size.

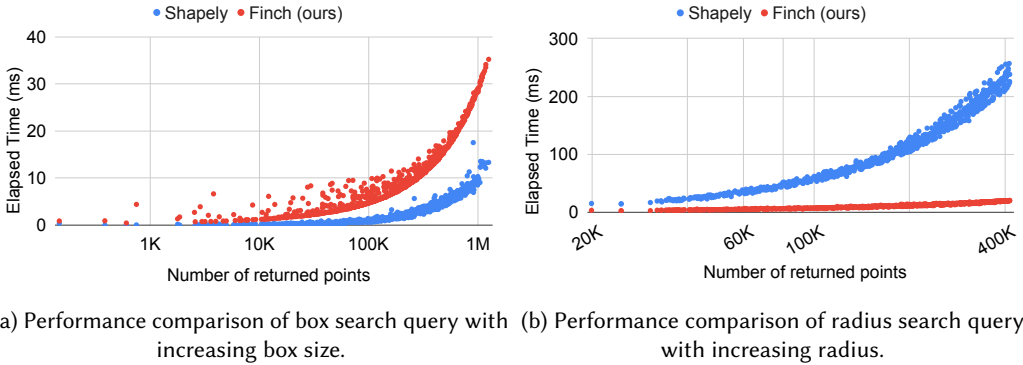(b) Performance comparison of radius search query with increasing radius.

Fig. 20. Experimental result of spatial queries: lower values indicate better performance.

not employ any spatial data structure. However, Figure 20b reveals that our code surpasses Shapely by 9.2× in terms of the geomean. This superiority arises from Shapely's reliance on STRTree, which operates solely on axis-aligned rectangle boxes. Shapely retrieves all the points within the bounding box of a circle ([-R,R] × [-R,R]), and then linearly scan the retrieved points to check whether the distance between each point and the center falls within the radius R. In contrast, our generated code flexibly iterates within the circular region on the fly (Figure 19b), eliminating the inefficiency of the two-step process in Shapely. Our implementation, accomplished in just 4-5 lines of code, offers a considerable advantage compared to Shapely implementations, which involve 501 lines of code for STR tree. Considerable potential exists for improving spatial query performance by incorporating a 2D spatial structure into the continuous tensor abstraction.

8.2 3D Point Cloud Convolution

The second application we explored involves 3D point cloud convolution. A 3D point cloud comprises a set of XYZ coordinates in 3D space, representing a 3D shape or object. In 3D deep learning, convolutional neural networks operate on point clouds rather than images, necessitating adaptations to accommodate these points. Various works in this field exist, and we chose KPConv [56], a

```
1   # Compute convolution only on designated pinpoints in Mask
2   for x=-∞:∞; for y=-∞:∞; for z=-∞:∞; # continuous
3     if Mask[x,y,z]
4
5       # Iterate Input points (x+r,y+s,z+t) within a ball centered at (x,y,z) with radius R
6       for r=-R:R; for s=-R:R; for t=-R:R; # continuous
7         if ((r*r+s*s+t*t) <= R*R)
8
9           # Iterate Kernel points (xk,yk,zk) within a ball centered at (r,s,t) with radius σ
10          for xk=-∞:∞; for yk=-∞:∞; for zk=-∞:∞; # continuous
11            if ((xk-r)*(xk-r)+(yk-s)*(yk-s)+(zk-t)*(zk-t) <= Sigma*Sigma)
12
13              # Compute convolution with Input and Kernel features
14              intpl_weight = (1 - sqrt((xk-r)*(xk-r)+(yk-s)*(yk-s)+(zk-t)*(zk-t))/Sigma)
15              for m=0:Cout-1; for c=0:Cin-1 # discrete integer iteration
16                Out[x,y,z,m] += intpl_weight * Kernel[xk,yk,zk,m,c] * Input[x+r,y+s,z+t,c]
```

Fig. 21. KPConv on 3D point clouds written in continuous tensor abstraction.

representative example, to illustrate its implementation within our continuous tensor abstraction. In this context, let $p_i \in \mathbb{R}^3$ denote the coordinates from a point cloud ($\mathbb{R}^{N \times 3}$), and $f_i \in \mathbb{R}^{C_{in}}$ represents the corresponding features from $F \in \mathbb{R}^{N \times C_{in}}$, where N is the number of points in the point cloud. The convolution of a point cloud *Input* by a filter *Kernel* at a point $x \in \mathbb{R}^3$ is defined as follows:

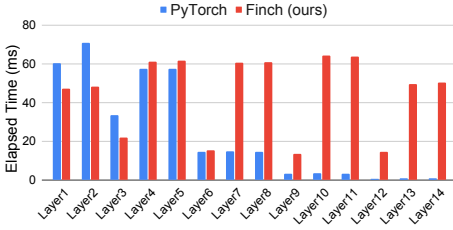$$(Input * Kernel)(x) = \sum_{p_i \in B_{x,R}} Kernel(p_i - x) \cdot f_i \qquad (1)$$

Here, our neighborhoods are defined within a ball $B_{x,R} = \{p_i \in \mathbb{R}^3 | \|p_i - x\| \leq R\}$ centered at x with a radius of R. It aggregates all the neighbors' feature f_i within the ball using a weighted sum based on the kernel weight. KPConv defines a continuous kernel $Kernel(p_i - x)$ using interpolation with predefined points $\tilde{p}_k \in \mathbb{R}^3$ from kernel points $\tilde{P} \in \mathbb{R}^{M \times 3}$, where M is the number of kernel points, and $W_k \in \mathbb{R}^{C_{in} \times C_{out}}$ represents their corresponding features.

$$(Input * Kernel)(x) = \sum_{p_i \in B_{x,R}} \left(\sum_{\tilde{p}_k \in B(p_i - x), \sigma} \left(1 - \frac{\|(p_i - x) - \tilde{p}_k\|}{\sigma} \right) \cdot W_k \right) \cdot f_i \qquad (2)$$
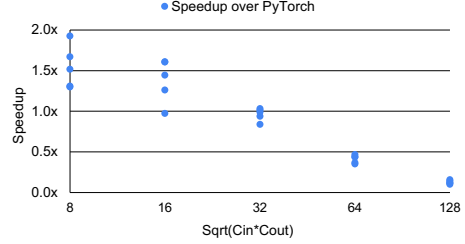
KPConv selectively interpolates with kernel points $\tilde{p}_k$ within a ball $B_{(p_i - x), \sigma}$ centered at $(p_i - x)$, the position of input point neighbors centered on x, with a radius of σ. It then employs linear correlation as an interpolation weight, denoted as $\left(1 - \frac{\|(p_i - x) - \tilde{p}_k\|}{\sigma} \right)$.

Figure 21 provides an implementation of KPConv described within the continuous tensor abstraction. The Input is a 4-dimensional tensor, with the first three dimensions containing the positions of points p_i, and the fourth dimension corresponding to the features $f_i \in \mathbb{R}^{C_{in}}$. Similarly, the 5-dimensional continuous tensor Kernel includes three dimensions for the positions of kernel points $\tilde{p}_k$ and the remaining two dimensions for features $W_k \in \mathbb{R}^{C_{in} \times C_{out}}$.

Lines 2-3 iterate solely over a region of interest, encompassing the pinpoints in Mask for which convolution is desired. Lines 6-7 and Lines 10-11 involve the iteration over input points and kernel points within the neighborhoods $B_{x,R}$ and $B_{(p_i - x), \sigma}$, respectively. Finally, Lines 14-16 perform the convolution between input and kernel features with interpolation weights. In Line 15 (c and m), both input and kernel features are iterated discretely, like in the traditional dense tensor programming model. With our continuous tensor abstraction, KPConv is succinctly and clearly described in 16 lines of code, while PyTorch requires 2,330 lines of code, including the ball query search library [12].

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

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

Fig. 22. Experimental results of 3D point cloud convolution: (a) Lower is better, (b) Higher is better.

Figure 22 presents the experimental results comparing our generated code with the PyTorch implementation of KPConv. In Figure 22a, the elapsed time of each KPConv layer in the 3D shape classification model architecture using the ModelNet40 [58] 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, the PyTorch implementation outperforms our generated code. In contrast to our code in Figure 21, where we perform (1) ball query searches (Lines 6-11) and (2) dense feature multiplication on the fly (Lines 14-16) within a single nested for loop, the PyTorch implementation separates these two components. It first creates a large dense tensor, which comprises all the neighbor features collected through a ball search query library, and subsequently applies well-optimized dense BLAS routines to this large dense tensor.

Figure 22b further confirms the difference by demonstrating the speedup over PyTorch with increasing channel size. The initial layers have small channel sizes ($\sqrt{C_{in} \cdot C_{out}} \leq 32$), but the later layers have larger channels that emphasize dense computation. As a result, our generated code performs better in the first half of the layers, where the ball query plays a dominant role in the overall computation. In contrast, the PyTorch implementation excels in the latter half, where dense computation is the dominant component.

8.3 Trilinear Interpolation in Neural Radiance Field

The third application we've explored is a Neural Radiance Field (NeRF) [42] in 3D deep learning. NeRF is a widely used machine learning model in computer graphics and computer vision, creating detailed 3D reconstructions from 2D images. Many NeRF models [24, 39, 55] utilize trilinear interpolation on 3D sparse voxel grids to efficiently represent the 3D scene. In the context of NeRF, generating a 2D image from a new viewpoint involves creating rays, sampling points along each ray, performing trilinear interpolation on the sparse voxel grid for each sampled point, and combining the outcomes to calculate the final RGB color using the volume rendering. Our specific focus in this context is on trilinear interpolation during the ray sampling phase within Plenoxel [24].

While the actual computations take place in 3D, we illustrate a 2D bilinear interpolation for explanatory purposes in Figure 23a. It highlights how the interpolated value at the desired point (depicted in black) is equal to the sum of the products of the values at each corner point and the corresponding partial areas diagonally opposite the corners (indicated by different colors). In Figure 23b, we represent this concept in continuous tensor abstraction by expanding each corner point and the desired point (x, y) into squares. We then compute the interpolation by calculating intersected area using integral reduction as $\int_0^1 \int_0^1 Grid_{x+i, y+j} \cdot di \cdot dj$. This interpolation is performed on the sparse 2D grid at every sampled point along the ray.

Figure 23c demonstrates the continuous tensor implementation of 3D trilinear interpolation at sampled points along the ray. In Lines 1-4, it samples a series of points by incrementing the

(a) Visualization of bi-linear interpolation.

(b) Bilinear interpolation at sampled points along the ray on a 2D sparse grid.

```
1  for t=0:T-1       # sampling on discrete timestep
2    x = Ox + Dx*t # O : ray origin, D : ray direction
3    y = Oy + Dy*t
4    z = Oz + Dz*t
5    for i=0.0:1.0     # continuous
6      for j=0.0:1.0   # continuous
7        for k=0.0:1.0 # continuous
8          for c=0:27  # interpolating 28 discrete features
9            Out[t,c] += Grid[x+i,y+j,z+k,c]*d(i)*d(j)*d(k)
```

(c) Trilinear interpolation of sampled ray point in a 3D sparse voxel grid written in our abstraction.

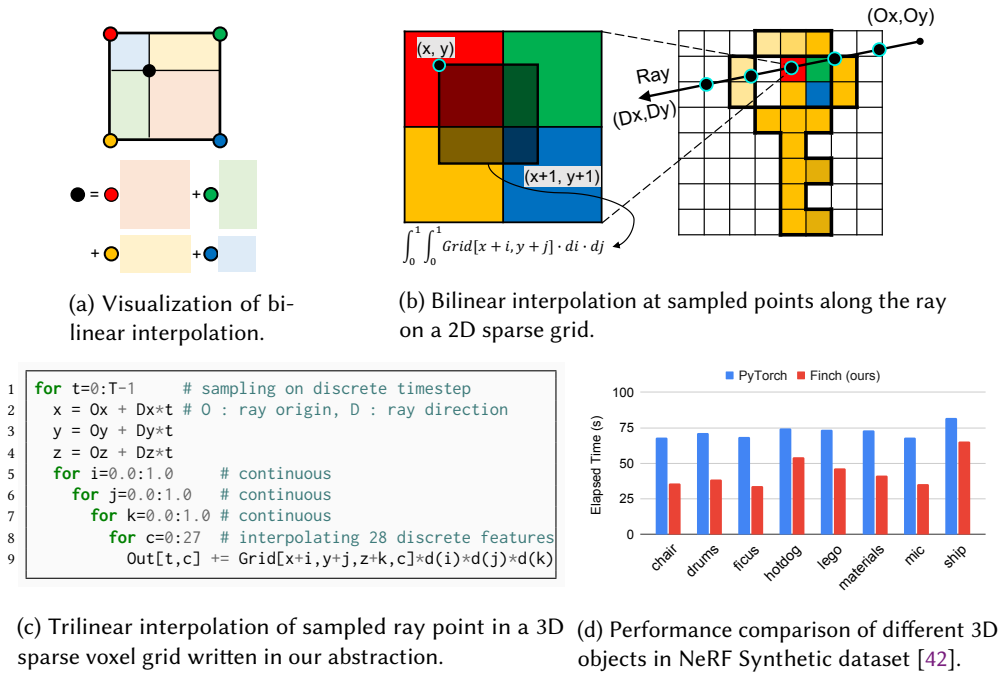(d) Performance comparison of different 3D objects in NeRF Synthetic dataset [42].

Fig. 23. Illustration of (a,b) bilinear interpolation and (c) trilinear interpolation during ray sampling in sparse voxel grids. Note that (a) and (b) are for visualization purposes, while (c) the actual computations occur in 3D.
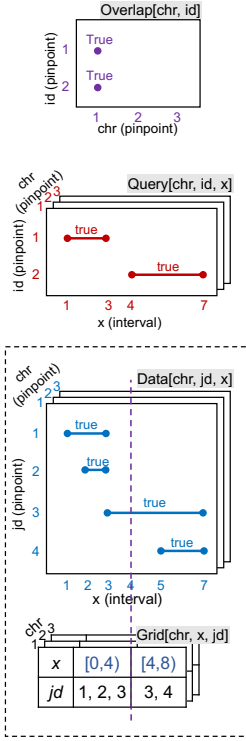
integer time step regularly along the ray. Lines 5-9 subsequently perform trilinear interpolation at each of these points using integral reduction. It's important to note that in the actual Plenoxel model, the implementation interpolates a 28-dimensional feature vector, with Line 8 reflecting this. Our implementation is remarkably simple and intuitive, providing a clear understanding of the program's behavior in 3D continuous space. It requires only 9 lines of code, while the PyTorch implementation involves 82 lines of code within traditional tensor programming model.

Figure 23d presents a performance comparison between our generated code and a PyTorch implementation for trilinear interpolation of Plenoxel. We conducted this evaluation while rendering a 256×256 image using the NeRF Synthetic dataset [42]. Our generated code surpasses the baseline implementation by a factor of 1.3× to 2.0×. This improvement can be primarily attributed to our efficient utilization of voxel grid sparsity, as we only store non-zero voxels and avoid unnecessary computations. In contrast, the PyTorch implementation employs a fully dense voxel grid, storing even empty voxels, and performing computations regardless of voxel occupancy.

8.4 Genomic Interval Operations

The last application we've explored is genomic interval operations using our continuous abstraction. In Bioinformatics, performing operations on genomic sequences and applying boolean operations to genomic interval data can be computationally intensive [47]. Figure 24 illustrates genomic interval operations presented within the continuous tensor abstraction.

In Figure 24a, genomic interval data is depicted in a 3D continuous tensor, denoted as the interval database Data[chr, jd, x] and query intervals Query[chr, id, x]. Each chromosome (chr) contains a 2D subtensor (e.g., Data[chr, :, :]), with multiple intervals across a continuous

(a) Genomic data is represented as a continuous tensor. `Grid` divides the `Data`'s x dimension in half, serving to accelerate intersection tests by filtering out unnecessary data points.

```
# Naive description of Intersect query
for chr=1:23        # discrete
  for id=1:N        # discrete
    for jd=1:M      # discrete
      for x=-∞:+∞   # continuous
        Intersect[chr,id,jd,x] |= Query[chr,id,x] && Data[chr,jd,x]

# Naive description of Overlap query
for chr=1:23        # discrete
  for id=1:N        # discrete
    for jd=1:M      # discrete
      for x=-∞:+∞   # continuous
        Overlap[chr,id] |= Query[chr,id,x] && Data[chr,jd,x]

# Naive description of Join query
for chr=1:23        # discrete
  for id=1:N        # discrete
    for jd=1:M      # discrete
      for x=-∞:+∞   # continuous
        Join[chr,id,jd] |= Query[chr,id,x] && Data[chr,jd,x]

# Naive description of Count query
for chr=1:23        # discrete
  for id=1:N        # discrete
    for jd=1:M      # discrete
      for x=-∞:+∞   # continuous
        if (Query[chr,id,x] && Data[chr,jd,x])
          Count[chr,id] += 1
```

```
# Accelerating Overlap query with Grid
for chr=1:23             # discrete
  for id=1:N             # discrete
    for x1=-∞:+∞         # continuous
      for jd=1:M         # discrete
        if (Query[chr,id,x1] && Grid[chr,x1,jd]) # Filtering with Grids
          for x=-∞:+∞   # continuous
            Overlap[chr,id] |= Query[chr,id,x] && Data[chr,jd,x]
```
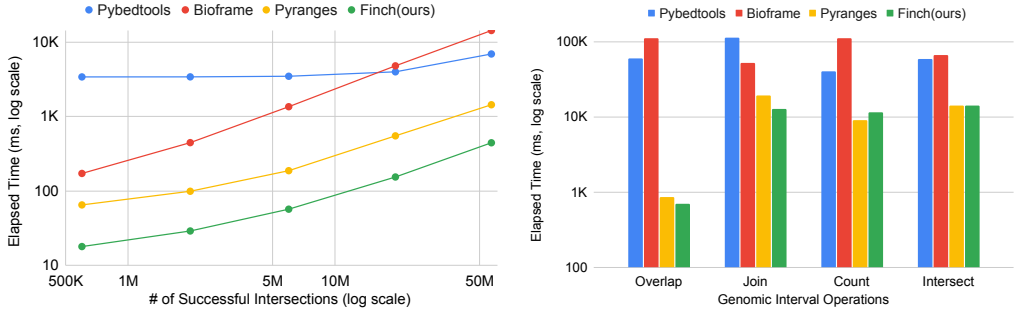
(b) Continuous loops for four distinct genomic operations: Intersect, Overlap, Join, and Count. The code above demonstrates the naive version, while the code below illustrates the Overlap query using grid partitioning.

Fig. 24. Genomic interval operations in continuous tensor abstraction. (a) Genomic data is represented as 3D continuous tensor where white regions indicate 'false' boolean values. Below is a table that represents a 1D grid index of Data[chr, jd, x] partitioned by half.

domain (x), each identified by a unique interval ID (`jd`). The `chr` and `jd` dimensions represent pinpoint coordinates, while the `x` dimension signifies interval coordinates.

Figure 24b (top) illustrates four distinct genomic operations within our continuous tensor abstraction. For example, `Count[chr, id]` counts the number of intervals in `Data` intersecting with a query interval (`Query[chr, id, :]`). 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 intervals (`id`) and data intervals (`jd`) within each chromosome, incurring substantial computational costs when handling numerous intervals per chromosome.

Thus, in practice, previous studies [3, 22, 23, 38, 47] have used an interval data structure to skip unnecessary comparisons. We demonstrated a 1D grid [23] using our abstraction to partition the continuous domain along dimension `x` into exclusive partitions, encompassing the entire dimension `x`. The 1D grid index, represented in a 3D continuous tensor as `Grid[chr, x, jd]`, is illustrated in

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

(b) Performance comparison using realistic dataset.

Fig. 25. Experimental result of genomic interval operations: lower values indicate better performance. Finch(ours) uses 1D-grid partitioning data structure.

Figure 24a (bottom table). It divides the x domain into two halves: [0,4) and [4,8), with interval IDs (jd) allocated to the respective partitions.

Figure 24b (bottom) depicts the Overlap query using this 1D grid index within our abstraction. The grid filters out unnecessary intervals in Data, avoiding pairwise comparisons. For instance, consider a query interval Query[1,2,:] spanning [4,7]. It intersects only with Data[1,3,:] and Data[1,4,:]. As this query interval [4,7] exclusively intersects with the right half ([4,8)) of the Grid, computations are only processed for jd=3, 4.

Figure 25 presents a performance comparison between our generated code using grid partitioning and three baseline implementations: Pybedtools [20], Bioframe [44], and Pyranges [52]. Pybedtools is a Python wrapper of Bedtools [47], a C library which utilizes a hierarchical binning data structure internally, Bioframe leverages the Pandas [40] framework for genomic interval operations, and Pyranges employs a Nested Containment List [3], a variation of the segment tree written in C. Index building time was not measured in these experiments.

In Figure 25a, 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 25b presents a performance comparison on a realistic dataset [38], 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 8 lines of code, while Bedtools implementation require 206 lines of code.

9 RELATED WORKS

Dense tensor programming models. Tensor programming, rooted in Fortran's array data structure [6], has provided a foundation for diverse applications. Numerous compiler works aim to improve code generation for such tensor-based programs. Techniques like loop vectorization [17, 36] and parallelization [19] enhance tensor access optimization and hardware resource utilization. The polyhedral compilation model [7, 13], based on integer linear programming (ILP), optimizes programs by treating nested loop iterations as lattice points within polyhedra and applying affine loop transformations like tiling or skewing.

In recent years, the machine learning community has introduced frameworks like TensorFlow [1], Jax [14], and PyTorch [45], inspired by tensor-focused languages like Matlab [32] and NumPy [28]. These frameworks are instrumental in developing machine learning models, which heavily rely on tensor operations. The latest advancements in scheduling languages [16, 48] 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, unlike their dense counterparts, provide multiple storage format options. The level format abstraction, originally introduced in TACO [18, 34], explains the diversity of existing sparse formats by introducing the concepts of a coordinate hierarchy and level format. This abstraction has further evolved into the fibertree abstraction [54], which serves as a foundational format abstraction for our continuous tensors.

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 [31], MLIR sparse dialect [11], TACO [34], SparseTIR [59], and Finch [2] can generate efficient code from sparsity-agnostic definitions of computation. The TACO project [18, 30, 34, 51] 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 [2] 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 [8] and Sympy [41], offer an intuitive way to manipulate continuous functions in numerical computing. They enable operations like differentiation, integration, and root-finding for functions defined over specific intervals. In addition to working with piecewise constant functions, they offer the capability to handle a wider range of function types beyond constant functions by leveraging a symbolic computation engine. However, their primary focus is not on performance or the tensor programming model with loops. Chebfun focuses on Chebyshev polynomials, a computation class commonly used in numerical computing, which is entirely different from our focus. While Chebfun supports 1D piecewise functions, it is limited to 2D and 3D support [29, 57] and lacks general N-D piecewise capabilities. 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 of intervals.

10 CONCLUSION

In this paper, we've introduced the continuous tensor abstraction, extending the domain of indices to real numbers. Our approach is grounded in piecewise-constant tensors, offering a unique format abstraction for storing these tensors and a code generation technique to produce efficient code from continuous loops. This novel abstraction enables the creation of diverse applications, spanning genomic interval operations, spatial searches, point cloud convolution, and trilinear interpolation on sparse voxel grids. It introduces a fresh perspective to loop-level reasoning for these applications, which has remained largely unexplored within traditional tensor programming models. The resulting compiler efficiently generates code, achieving competitive performance levels with leading libraries in some domains on the CPU, all while requiring significantly less code. We believe that this paper marks the onset of a new era in tensor programming paradigms, opening up exciting possibilities for the future.

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, 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.

[3] 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.

[4] Anonymous. 2024. Finch: Sparse and Structured Array Programming with Control Flow. *under submission at OOPSLA round 2* (2024).

[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] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[8] 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.

[9] 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.

[10] 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).

[11] 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.

[12] 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.

[13] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[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] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All you need is superword-level parallelism: systematic control-flow vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 301–315.

[18] 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.

[19] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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

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

[28] 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.

[29] Behnam Hashemi and Lloyd N Trefethen. 2017. Chebfun in three dimensions. *SIAM Journal on Scientific Computing* 39, 5 (2017), C341–C363.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. *Acm Sigplan Notices* 35, 5 (2000), 145–156.

[37] 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.

[38] 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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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).

[44] 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.

[45] 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).

[46] Franco P Preparata and Michael I Shamos. 2012. *Computational geometry: an introduction.* Springer Science & Business Media.

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

[48] 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.

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

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

[51] 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.

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

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

[54] 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.

[55] 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.

[56] 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.

[57] Alex Townsend and Lloyd N Trefethen. 2013. An extension of Chebfun to two dimensions. *SIAM Journal on Scientific Computing* 35, 6 (2013), C495–C518.

[58] 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.

[59] 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.