Cinder: Design and Implementation of a Compact CUDA Tensor Library

Jiarui Guo
College of Computer Science, Nankai University
Tianjin, China
guojiarui@mail.nankai.edu.cn

ABSTRACT

Modern tensor software is a quiet systems boundary: Python objects, mathematical array semantics, native implementations, accelerator memory, and kernel execution meet behind a compact user interface. Cinder is a small CUDA-backed dense tensor library that exposes one Python value type over a C++ and CUDA implementation. This report documents Cinder as an engineering artifact rather than a research contribution or performance competitor. Its current scope is deliberately narrow: dense `float32` tensors, runtime-rank shapes, row-major storage, explicit host/device transfer, eager materialized operations, and CUDA kernels for shape and tensor-algebra primitives. The report explains the distinction between mathematical tensors and finite array tensors, then traces how Cinder keeps shape reasoning, storage ownership, device metadata, and kernel planning separate enough to remain inspectable. The result is a concrete design record for a compact Python/C++/CUDA tensor library, including its implementation structure, current limits, and risks for future extension.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; • **Computing methodologies** → *Parallel computing methodologies.*

KEYWORDS

tensor library, CUDA, dense arrays, Python extension, GPU kernels, tensor semantics

ACM Reference Format:
Jiarui Guo. 2026. Cinder: Design and Implementation of a Compact CUDA Tensor Library. In *Cinder Technical Report, June 2026, Tianjin, China.* ACM, New York, NY, USA, 15 pages.

1 INTRODUCTION

The AI era has made tensor software both invisible and decisive. To most users, the visible object is a model, an assistant, an embedding, or a generated image. Underneath that interface, however, modern AI systems are built from repeated tensor programs running across accelerator memory hierarchies, kernel launch boundaries, compiler stacks, and distributed runtimes. The Transformer made attention-heavy tensor computation central to modern deep learning [39]; foundation models turned that computation into a general-purpose platform layer [3]; and recent AI Index reporting describes a field whose capability, investment, and deployment pressure are advancing faster than many evaluation and governance mechanisms [35]. In this setting, tensor libraries are not merely convenience APIs for numerical arrays. They are the small, load-bearing joints between mathematical notation, user-facing programming models, native system software, and physical accelerator hardware.

That shift changes what is worth explaining in a small tensor project. The interesting question is not whether a compact library can imitate PyTorch, TensorFlow, NumPy, cuTENSOR, or a tensor compiler. It cannot, and Cinder does not claim otherwise. The useful question is more architectural: when a tensor value crosses from Python into C++ and then into CUDA, which meanings must stay explicit so that the system does not dissolve into operation-by-operation special cases? Shape, layout, ownership, metadata placement, error reporting, and launch planning may look like implementation details, but in AI workloads they are exactly where cost and correctness meet. Work such as FlashAttention shows this at a larger scale: the algorithmic idea matters, but so do memory traffic, tiling, and the concrete path through the GPU memory hierarchy [7].

Cinder is a compact CUDA-backed dense tensor library exposed to Python through a native C++ extension. Its current public scope is deliberately narrow: dense `float32` tensors, runtime-rank shapes, row-major storage, explicit host-to-device and device-to-host movement, and eager tensor operations implemented by CUDA kernels. The supported operation set includes construction, arithmetic, reshaping, broadcasting, slicing, concatenation, transpose, tensor product, contraction, mode multiplication, inner product, and norm. This is not the scope of a production numerical framework. It is a small engineering artifact whose value lies in being inspectable: the reader can see where tensor semantics end, where storage begins, and where host-side planning becomes device-side work.

This report therefore uses Cinder as a concrete design record, not as a research claim. Cinder introduces no new tensor algebra, no new GPU programming model, no automatic differentiation system, no graph compiler, no distributed runtime, and no performance result that would threaten mature libraries. Its engineering stance is intentionally more modest and, for that reason, easier to audit. Public tensors are dense row-major values. Shape errors are host errors. Results are materialized rather than exposed as lazy views. Rank-zero results remain tensors rather than escaping through a separate scalar channel. CUDA remains the execution substrate [21]; Cinder's concern is how to keep the Python object, C++ owner, packed device allocation, and CUDA kernel plan from being collapsed into one ambiguous object.

Cinder, June 2026, Tianjin, China
2026.

The report first separates mathematical tensors from array tensors, because the engineering object used by Cinder is a finite dense array with explicit shape, scalar type, and storage convention rather than a basis-independent multilinear object. It then describes Cinder's dense runtime-rank tensor model, its value-producing public operations, and the execution discipline that keeps shape reasoning on the host while leaving dense value computation to CUDA. After that, it walks through the native implementation: typed C++ products, policy-based layout, packed CUDA allocations, host-side kernel plans, and a pybind11 boundary that exposes one Python value type. Finally, it discusses current limits and risks so that future extensions can be evaluated as engineering changes rather than absorbed as incidental complexity.

The remainder of the report is organized as follows. Section 2 develops the mathematical and computational principles behind the tensor abstraction. Section 3 presents Cinder's design objectives, semantic core, operation protocol, layering, metadata strategy, and public API boundary. Section 4 describes the implementation in C++23, CUDA C++, pybind11, and scikit-build-core, with particular attention to storage ownership, kernel planning, launch behavior, transfer economy, and the Python extension ABI. Section 5 discusses current limits, design risks, and an incremental roadmap for measurement, descriptors, testing, views, dtype support, and optimized dispatch.

2 PRINCIPLE

The principle of tensor computation can be stated as a hierarchy of mathematical objects: the intrinsic tensorial object, its coordinate table, the finite index domain on which an array is defined, and the linearization map that assigns offsets to coordinates. These levels are often collapsed in numerical discourse, but keeping them separate prevents a category error: a mathematical tensor is an invariant multilinear object, whereas an n-dimensional array is a finite function with a chosen coordinate and linearization convention.

2.1 From Linear Algebra to Tensor Algebra

The historical development from linear algebra to tensor algebra is a movement from calculation with coordinates to laws that survive changes of coordinates. Classical linear algebra began with linear equations, determinants, and matrices. Grassmann's *Ausdehnungslehre* introduced an abstract calculus of extension and exterior products, making it possible to reason about vectorial magnitudes beyond their coordinate tuples [10]. Cayley's theory of matrices made linear transformations algebraic objects in their own right [4]. Ricci and Levi-Civita's absolute differential calculus then supplied systematic coordinate transformation rules for higher-order quantities [33]; Einstein's general relativity made those rules central to physics by requiring geometric laws to be coordinate-independent [9].

The central abstraction is not "many indices", but *controlled behavior under linear change*. Modern tensor theory packages this into three equivalent or complementary definitions: transformation laws, multilinear maps, and elements of tensor products [20]. The tensor product view is the most compact for finite-dimensional algebra. Let $V_0, \dots, V_{r-1}$ be vector spaces over a field $\mathbb{K}$. The tensor product

$$V_0 \otimes V_1 \otimes \cdots \otimes V_{r-1}$$

is characterized by the universal factorization property: every r-linear map

$$\Phi : V_0 \times V_1 \times \cdots \times V_{r-1} \to W$$

to a vector space W factors uniquely through a linear map on the tensor-product space,

$$\Phi = F_\Phi \circ \sigma_\otimes, \qquad \sigma_\otimes(v_0, \dots, v_{r-1}) = v_0 \otimes \cdots \otimes v_{r-1}.$$

Here $v_a \in V_a$, $\sigma_\otimes$ is the canonical multilinear map, and F_Φ is the unique linear map determined by Φ. This property says that tensor products isolate the multilinear part of every multilinear construction; once the tensor product has been formed, the remaining operation is linear.

Property 1: multilinear universality. A tensor product is universal for multilinear expressions. If a quantity is genuinely tensorial, then substituting a linear combination into any input slot distributes linearly through that slot while all other slots remain fixed. This is stronger than saying that its components sit in a rectangular table. A table records coefficients; the tensor product explains why multilinear expressions can be represented by those coefficients in the first place.

Property 2: basis covariance. Let

$$T \in V_0 \otimes \cdots \otimes V_{r-1}, \qquad T = \sum_{\mathbf{i}} T_{i_0,\dots,i_{r-1}}\, b_{0,i_0} \otimes \cdots \otimes b_{r-1,i_{r-1}}.$$

For each a, change the basis of V_a by

$$\tilde{b}_{a,j} = \sum_i b_{a,i} P^{(a)}_{ij}, \qquad P^{(a)} \in GL(V_a).$$

Then the same tensor T has new components

$$\tilde{T}_{j_0,\dots,j_{r-1}} = \sum_{\mathbf{i}} \big((P^{(0)})^{-1}\big)_{j_0 i_0} \cdots \big((P^{(r-1)})^{-1}\big)_{j_{r-1} i_{r-1}} T_{i_0,\dots,i_{r-1}}.$$

The component array changes, but the underlying tensor does not. A tensorial statement is therefore not a statement about one table of numbers; it is a statement compatible with every admissible basis.

Property 3: variance and duality. With dual spaces included, one obtains mixed tensors. A tensor of type (p, q) over a finite-dimensional vector space V is an element of

$$V^{\otimes p} \otimes (V^*)^{\otimes q},$$

or equivalently a multilinear map with p covector slots and q vector slots, depending on the chosen convention. Vector slots and covector slots transform in opposite ways. This variance information is invisible in a bare multidimensional array: two arrays with the same shape may represent tensors with different transformation laws.

Property 4: functoriality. If $f_a : V_a \to W_a$ are linear maps, then they induce a canonical linear map

$$f_0 \otimes \cdots \otimes f_{r-1} : V_0 \otimes \cdots \otimes V_{r-1} \to W_0 \otimes \cdots \otimes W_{r-1}.$$

This functorial behavior is why tensor notation composes cleanly with linear maps, change of coordinates, and products of spaces.

The operation acts on the spaces and only secondarily on components.

Property 5: invariant contractions. The pairing $V^* \times V \to \mathbb{K}$ induces coordinate-independent contractions. In components, contraction looks like a summation over a repeated index; intrinsically, it is evaluation of a covector slot against a vector slot. The summation symbol is therefore not the source of invariance. The source is the dual pairing that remains well-defined under basis change.

Together these properties give a useful test: a tensor is not merely a rectangular collection of scalars. It is a coordinate-dependent representation of an object whose defining operations commute with admissible linear changes of basis.

2.2 Array Tensor as a Finite-Mapping Approximation

The computational convention uses the word "tensor" for a related but weaker object. An order-r array tensor with shape

$$\mathbf{e} = (e_0, e_1, \ldots, e_{r-1})$$

is a total map

$$A : D_{\mathbf{e}} \to S, \qquad D_{\mathbf{e}} = \prod_{a=0}^{r-1} \{0, 1, \ldots, e_a - 1\},$$

where S is a homogeneous element set such as $\mathbb{R}$ or $\mathbb{C}$. The integer r is the order or number of axes; the extent e_a is the cardinality of axis a. The dense cardinality is

$$|D_{\mathbf{e}}| = \prod_{a=0}^{r-1} e_a.$$

The empty product is 1, so an order-zero array tensor has exactly one coordinate, the empty tuple. If some extent is zero, the corresponding axis set is empty and the whole domain is empty.

This definition is purely mathematical: it describes a finite rectangular domain and a value assigned to every coordinate. Shape transformations are maps between such domains. Elementwise operators compose A with scalar operations on S. Reductions quotient a domain by summing over fibers. Linearization enters only after this semantic object has been defined. The approximation is that this finite map is treated as though it were the tensor, even though it is only a coordinate table after additional structure has been fixed.

A conditional embedding. When $S = \mathbb{K}$, vector spaces V_a with $\dim V_a = e_a$ are selected, and ordered bases

$$\mathcal{B}_a = (b_{a,0}, b_{a,1}, \ldots, b_{a,e_a-1})$$

are fixed, the array map A determines a mathematical tensor

$$T_A = \sum_{\mathbf{i} \in D_{\mathbf{e}}} A(\mathbf{i})\, b_{0,i_0} \otimes b_{1,i_1} \otimes \cdots \otimes b_{r-1,i_{r-1}}.$$

For those fixed bases, the correspondence

$$A \longleftrightarrow T_A$$

is a linear isomorphism between $\mathbb{K}^{e_0 \times \cdots \times e_{r-1}}$ and $V_0 \otimes \cdots \otimes V_{r-1}$. The isomorphism is coordinate-dependent. Under a basis change, the component table changes by the appropriate product of change-of-basis matrices, while the tensor T_A remains invariant.

Proposition 1: the embedding is not canonical. The same finite map can represent different mathematical tensors if the bases are changed but the table is held fixed.

Proof. It is enough to consider order 1. Let $A(0) = 1$, $A(1) = 0$, so the table is $(1, 0)$. In basis (b_0, b_1), it represents b_0. In another basis $(\tilde{b}_0, \tilde{b}_1)$, the same table represents $\tilde{b}_0$. Unless $\tilde{b}_0 = b_0$, these are different vectors. Thus the finite map alone does not determine a basis-independent tensor. □

Proposition 2: the finite map has no intrinsic covariance law. A mathematical tensor carries a prescribed action of each basis change on its components. A bare map $A : D_{\mathbf{e}} \to S$ does not.

Proof. A basis change is an element of $GL(V_a)$ for some vector space V_a. The finite map specifies only an index set and values in S; it does not specify the vector spaces V_a, the bases of those spaces, or whether each axis is covariant or contravariant. Therefore there is no intrinsic object on which $GL(V_a)$ can act. One may add a rule externally, but that rule is additional structure, not a consequence of the finite map. □

Proposition 3: array rank is not tensor rank. The order r of an array tensor counts axes. Tensor rank in multilinear algebra measures the minimal number of simple tensors needed to express an element:

$$\operatorname{rank}(T) = \min \left\{ m : T = \sum_{\ell=1}^{m} v_{0,\ell} \otimes \cdots \otimes v_{r-1,\ell} \right\}.$$

For matrices, this recovers ordinary matrix rank; for higher-order tensors, it is a subtler decomposition invariant [17]. Thus the engineering phrase "rank-r tensor" means order r, not algebraic tensor rank. The terminology is convenient, but it collapses two distinct invariants.

Proposition 4: many array transformations are not tensorial maps. Some array operations are tensorial under extra interpretation, but many are only operations on coordinate tables. Reshape, for example, is defined by a chosen linearization:

$$B(\mathbf{j}) = A\left(L_{\mathbf{e}}^{-1}(L_{\mathbf{f}}(\mathbf{j}))\right).$$

This rule can map a table of shape $(2, 3)$ to a table of shape $(3, 2)$, but the result is not the canonical factor swap $\mathbb{K}^2 \otimes \mathbb{K}^3 \to \mathbb{K}^3 \otimes \mathbb{K}^2$. It is an enumeration-dependent bijection of coordinate tables. The linearization convention, not tensor algebra, determines the result.

Broadcasting illustrates a different failure mode. It duplicates values along an axis of extent 1. The map is well-defined on finite domains, but duplication is not a basis-covariant tensor operation unless one supplies a distinguished diagonal, copy map, or comonoid structure. Such a structure is natural for arrays and unnatural for arbitrary vector spaces. This is the conceptual price paid by the engineering approximation: it gains operational convenience by fixing more coordinate structure than tensor algebra itself provides.

What the approximation preserves. The finite-mapping model is nevertheless faithful within a fixed computational gauge. A gauge here means a fixed choice of vector spaces, ordered bases, variance convention, scalar semantics, and linearization convention. Inside

that gauge, the component table is a complete coordinate representation of the corresponding tensor, and coordinate formulas for tensor product, contraction, mode multiplication, and inner product reproduce the usual multilinear algebra. Outside that gauge, the same table should be read more cautiously: it is a finite map that approximates tensorial behavior where the missing geometric structure is irrelevant or intentionally frozen.

Boundary of the identification. The array tensor and the mathematical tensor coincide only after four pieces of structure have been fixed: vector spaces, bases, variance, and transformation law. Without these, an n-dimensional array is simply a finite function. This distinction matters for at least four reasons. First, arrays may store samples, pixels, probabilities, or learned parameters with no intended tensorial equivariance. Second, changing an array layout or axis order need not implement a basis change; it may merely choose another enumeration of the same finite map. Third, many useful array operations exploit copying, broadcasting, or linearization conventions that have no basis-free tensor analogue. Fourth, exact tensor identities are stated over fields or modules, while numerical arrays may be evaluated in finite arithmetic, where associativity and distributivity are only approximate.

2.3 Linearization and Offset Calculus

Any dense array tensor can be enumerated by a linearization map

$$L : D_{\mathbf{e}} \to \{0, 1, \ldots, |D_{\mathbf{e}}| - 1\}.$$

The purpose of L is to impose a total order on a rectangular index domain. The offset calculus of multidimensional arrays is the study of such maps. A contiguous linearization uses a bijective L; a view-like linearization may use an affine map into a larger coordinate space. In a byte-addressed model, the corresponding address formula is

$$\mathrm{addr}(\mathbf{i}) = b + \omega L(\mathbf{i}),$$

where b is the base address and ω is the item size. The mathematical content is the same even if one ignores bytes and treats L as an element offset.

Row-major order. Row-major order, also called C-order, makes the last axis vary fastest. Define the row-major stride of axis a by

$$s_a^{\mathrm{row}} = \prod_{b=a+1}^{r-1} e_b.$$

Then

$$L_{\mathrm{row}}(\mathbf{i}) = \sum_{a=0}^{r-1} i_a s_a^{\mathrm{row}}.$$

Equivalently, row-major linearization is mixed-radix Horner evaluation:

$$L_{\mathrm{row}}(\mathbf{i}) = (((i_0 e_1 + i_1) e_2 + i_2) \cdots e_{r-1}) + i_{r-1}.$$

Column-major order. Column-major order, also called Fortran-order, makes the first axis vary fastest. Its strides are

$$s_a^{\mathrm{col}} = \prod_{b=0}^{a-1} e_b,$$

and the offset is

$$L_{\mathrm{col}}(\mathbf{i}) = \sum_{a=0}^{r-1} i_a s_a^{\mathrm{col}}.$$

Row-major and column-major differ only by the priority order in which axes are enumerated; neither is more tensorial in the mathematical sense.

Permutation-major order. Let $\rho = (\rho_0, \rho_1, \ldots, \rho_{r-1})$ be a permutation of axes ordered from slowest varying to fastest varying. The stride assigned to axis ρ_k is

$$s_{\rho_k}^{\rho} = \prod_{\ell=k+1}^{r-1} e_{\rho_\ell},$$

and the corresponding linearization is

$$L_{\rho}(\mathbf{i}) = \sum_{a=0}^{r-1} i_a s_a^{\rho}.$$

Row-major is the special case $\rho = (0, 1, \ldots, r-1)$; column-major is the special case $\rho = (r-1, \ldots, 1, 0)$.

General strided order. The general affine strided map is

$$L_{\mathrm{stride}}(\mathbf{i}) = o + \sum_{a=0}^{r-1} i_a s_a,$$

where o is a base element offset and s_a is the stride of axis a. This formula is the conceptual core of many array systems: a multidimensional array is represented by data, shape, and strides [12, 38]. Row-major and column-major are special stride choices. Slicing changes o and possibly the extents; transposition permutes strides; broadcasting is expressible by setting some strides to zero.

Blocked, space-filling, packed, and sparse orders. More specialized layouts use structured linearization rather than a single global affine formula. In a blocked layout with tile sizes $\mathbf{t} = (t_0, \ldots, t_{r-1})$, each coordinate decomposes as

$$i_a = q_a t_a + r_a, \qquad 0 \le r_a < t_a,$$

so the offset combines an outer linearization of tile coordinates $\mathbf{q}$ and an inner linearization of in-tile coordinates $\mathbf{r}$. Morton order, or Z-order, interleaves coordinate bits and is useful when spatial locality is more important than axis-major iteration. Packed triangular and symmetric layouts restrict the domain, e.g. $0 \le j \le i < n$, and use offsets such as $i(i+1)/2 + j$ for one lower-triangular row-packed convention. Sparse layouts replace dense linearization by an index data structure; the semantic map is then no longer a total dense map to stored elements but a finite map with an implicit default value, usually zero.

2.4 Operator Principles

The operators considered here can be expressed as coordinate semantics. Let $\mathbf{i}$ and $\mathbf{j}$ denote input and output coordinates. The essential question for each operator is one of the following: which input coordinate is pulled back from an output coordinate, which scalar operation is applied at matching coordinates, or over which fiber is a sum taken?

Table 1: Offset strategies as mathematical linearization maps.

Strategy	Domain model	Linearization principle
Row-major	Rectangular dense	Last axis varies fastest
Column-major	Rectangular dense	First axis varies fastest
Permutation-major	Rectangular dense	Axis priority ρ determines strides
General strided	Affine view of a rectangular domain	$o + \sum_a i_a s_a$
Broadcast strided	Rectangular domain with repeated fibers	Some strides are zero
Blocked/tiled	Rectangular domain partitioned into tiles	Outer tile order plus inner tile order
Morton/Z-order	Rectangular grid	Bit interleaving of coordinates
Packed triangular	Non-rectangular domain	Closed-form enumeration of valid pairs
Sparse	Finite support in a larger domain	Lookup in coordinate/index structure

Elementwise tensor and scalar operations. For tensors $A, B : D_{\mathbf{e}} \to \mathbb{R}$ with identical shape and for $\oplus \in \{+, -, \times, /\}$,

$$C(\mathbf{i}) = A(\mathbf{i}) \oplus B(\mathbf{i}).$$

For a scalar $\alpha \in \mathbb{R}$,

$$C(\mathbf{i}) = A(\mathbf{i}) \oplus \alpha \qquad \text{or} \qquad C(\mathbf{i}) = \alpha \oplus A(\mathbf{i}),$$

depending on operand order. Addition and multiplication are commutative over $\mathbb{R}$; subtraction and division are not, so reverse scalar operations are semantically distinct.

Reshape. Let $A : D_{\mathbf{e}} \to S$ and let $\mathbf{f} = (f_0, \ldots, f_{q-1})$ satisfy

$$\prod_{a=0}^{r-1} e_a = \prod_{b=0}^{q-1} f_b.$$

Reshape preserves a chosen linearization:

$$B(\mathbf{j}) = A\Big(L_{\mathbf{e}}^{-1}(L_{\mathbf{f}}(\mathbf{j}))\Big).$$

Thus reshape is not a tensorial basis change. It is a reinterpretation of the same ordered sequence of values under a different rectangular domain.

Broadcast. Broadcasting is pullback along a projection. Suppose shape $\mathbf{e}$ is aligned to the trailing axes of target shape $\mathbf{f}$, and let $d = |\mathbf{f}| - |\mathbf{e}|$. For output coordinate $\mathbf{j}$, define

$$\pi(\mathbf{j})_a = \begin{cases} 0, & e_a = 1, \\ j_{a+d}, & e_a = f_{a+d}. \end{cases}$$

The broadcast result is

$$B(\mathbf{j}) = A(\pi(\mathbf{j})).$$

The condition $e_a = 1$ means an entire output fiber is pulled back to a single input coordinate.

Slice. Given a start vector $\mathbf{s}$ and output shape $\mathbf{f}$, slicing is the affine pullback

$$B(\mathbf{j}) = A(\mathbf{s} + \mathbf{j}), \qquad 0 \le j_a < f_a, \quad s_a + f_a \le e_a.$$

The constraints ensure that the translated output domain remains inside the input domain.

Concatenation. Let $A^{(0)}, \ldots, A^{(m-1)}$ have equal order and equal extents except on axis c. Define prefix sums

$$p_0 = 0, \qquad p_{\ell+1} = p_\ell + e_c^{(\ell)}.$$

The output extent on axis c is p_m. For output coordinate $\mathbf{j}$, choose the unique ℓ such that

$$p_\ell \le j_c < p_{\ell+1}.$$

Then

$$B(\mathbf{j}) = A^{(\ell)}(j_0, \ldots, j_{c-1}, j_c - p_\ell, j_{c+1}, \ldots, j_{r-1}).$$

Concatenation is therefore a piecewise affine pullback determined by a partition of one axis.

Transpose and axis permutation. For a permutation π interpreted as mapping output axes to input axes, the output shape is $f_k = e_{\pi(k)}$. The transposed tensor satisfies

$$B(j_0, \ldots, j_{r-1}) = A(i_0, \ldots, i_{r-1}), \qquad i_{\pi(k)} = j_k.$$

Ordinary matrix transpose is only the rank-two instance. In higher order, the operation is an arbitrary action of the symmetric group on axes.

Tensor product. For $A : D_{\mathbf{e}} \to \mathbb{R}$ and $B : D_{\mathbf{f}} \to \mathbb{R}$, the coordinate tensor product has domain $D_{\mathbf{e}} \times D_{\mathbf{f}}$ and

$$C(\mathbf{i}, \mathbf{j}) = A(\mathbf{i})B(\mathbf{j}).$$

After bases are fixed, this is the component form of the algebraic tensor product. Without bases, it is best understood simply as an outer product of two finite maps.

Contraction. Contraction sums over paired axes. Let $P = (p_0, \ldots, p_{c-1})$ be axes of A, $Q = (q_0, \ldots, q_{c-1})$ axes of B, and assume

$$e_{p_t} = f_{q_t} = g_t \qquad \text{for } 0 \le t < c.$$

Let $\mathbf{u}$ denote the free coordinates of A, $\mathbf{v}$ the free coordinates of B, and

$$\mathbf{t} \in \prod_{h=0}^{c-1} \{0, 1, \ldots, g_h - 1\}$$

the contracted coordinate. Then

$$C(\mathbf{u}, \mathbf{v}) = \sum_{\mathbf{t}} A(\mathrm{merge}_P(\mathbf{u}, \mathbf{t}))B(\mathrm{merge}_Q(\mathbf{v}, \mathbf{t})).$$

Here merge_P inserts the contracted coordinates into the axes P, and merge_Q does the analogous insertion for Q. Matrix multiplication is the special case

$$C_{ij} = \sum_{t=0}^{k-1} A_{it} B_{tj}.$$

Mode multiplication. The m-mode matrix product multiplies every fiber parallel to axis m by a matrix [17]. Let

$$A \in \mathbb{R}^{e_0 \times \cdots \times e_{r-1}}, \qquad U \in \mathbb{R}^{h \times e_m}.$$

The output has shape

$$(e_0, \ldots, e_{m-1}, h, e_{m+1}, \ldots, e_{r-1}),$$

and components

$$C_{i_0,\ldots,i_{m-1},j,i_{m+1},\ldots,i_{r-1}} = \sum_{t=0}^{e_m-1} A_{i_0,\ldots,i_{m-1},t,i_{m+1},\ldots,i_{r-1}} U_{j,t}.$$

It is ordinary matrix-vector multiplication applied uniformly to all mode-m fibers.

Inner product, dot product, and norm. For two same-shaped tensors,

$$\langle A, B \rangle = \sum_{\mathbf{i} \in D_{\mathbf{e}}} A(\mathbf{i}) B(\mathbf{i}).$$

In full-array semantics, the dot product is this same total inner product. The L_2 norm is

$$\|A\|_2 = \sqrt{\sum_{\mathbf{i} \in D_{\mathbf{e}}} A(\mathbf{i})^2}.$$

For an empty domain, the summation convention gives the empty sum as 0; therefore the corresponding norm is 0.

2.5 Synthesis

The mathematical tensor is basis-independent; the array tensor is a finite map. The bridge between them is a choice of vector spaces, bases, variance, and transformation law. The bridge between an array tensor and a one-dimensional address space is a linearization map. Once these distinctions are fixed, the operator theory becomes uniform: shape operators are coordinate pullbacks, elementwise operators are pointwise scalar maps, tensor products are products over product domains, and contractions and norms are sums over fibers.

3 DESIGN

Cinder's design problem is deliberately narrower than that of a production tensor framework. It does not attempt to optimize every common numerical workload, dispatch across devices, or support a large dtype lattice. Instead, the system asks a smaller systems question: what is the minimum structure needed to present tensor algebra as a stable Python object while executing every operation as explicit CUDA work? This constraint is productive. It keeps the design space small enough that the main trade-offs can be stated precisely.

The central design choice is to make all public operations value-producing. A method such as `reshape`, `transpose`, or `broadcast` returns a new dense tensor rather than a lazy view. This rejects a large class of optimization opportunities, but it also rejects the aliasing, lifetime, and non-contiguous-stride cases that dominate mature array systems. Cinder therefore prioritizes semantic regularity over peak generality: every public tensor owns a dense row-major device allocation, and every result can be passed to the next operation without consulting hidden view state.

3.1 Design Objectives

The implementation is organized around four objectives.

O1: a small and stable public surface. The Python API exposes one primary type, `Tensor`. Internal concepts such as mappings, device buffers, views, kernel plans, and CUDA launch details remain native implementation machinery. This follows the array-programming tradition in which users manipulate high-level array values while layout and execution stay behind the object boundary.

O2: explicit tensor semantics. Each operation is specified by shape validity, output shape, and value rule. Shape errors are detected before kernels launch. The device code is therefore not responsible for discovering semantic invalidity; it only evaluates a plan that has already been checked.

O3: dense ownership as the normal case. Cinder does not expose non-owning Python views, strided tensors, sparse formats, or in-place mutation. Those features are useful, but they introduce a second semantic layer in which a tensor value is no longer simply a shape plus dense row-major storage. The current design keeps that layer absent.

O4: host planning, device filling. The host computes compact plans: output extents, row-major strides, axis maps, and reduction sizes. CUDA kernels then fill dense output buffers using data-parallel loops. This matches CUDA's host/device programming model while keeping exception-based validation on the host.

3.2 Semantic Core

Let a Cinder tensor be modeled as a pair

$$T = (\mathbf{e}, x), \qquad \mathbf{e} = (e_0, \ldots, e_{r-1}) \in \mathbb{N}^r, \qquad x \in \mathbb{R}^{N(\mathbf{e})},$$

where r is the runtime rank, e_a is the extent of axis a, and

$$N(\mathbf{e}) = \prod_{a=0}^{r-1} e_a$$

is the dense element count. The rank-zero shape $()$ has $N(()) = 1$, so scalar results are ordinary tensors. If any extent is zero, $N(\mathbf{e}) = 0$, and the tensor is an empty dense value with preserved shape.

Coordinates live in the finite domain

$$D_{\mathbf{e}} = \prod_{a=0}^{r-1} \{0, \ldots, e_a - 1\}.$$

For row-major storage, the linearization map is

$$\lambda_{\mathbf{e}}(i_0, \ldots, i_{r-1}) = \sum_{a=0}^{r-1} i_a \prod_{b=a+1}^{r-1} e_b.$$

Here i_a is the coordinate on axis a. The empty coordinate maps to offset 0. This single convention is used by construction, host transfer, shape operations, tensor algebra, and reductions.

Every public operation is a partial function on these dense values:

$$f : \mathcal{T}^m \rightharpoonup \mathcal{T}.$$

It is undefined when ranks, extents, axes, or value counts violate the operation contract. When defined, it decomposes into two functions:

$$\text{shape}_f : (\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(m)}) \mapsto \mathbf{o},$$

and

$$\text{value}_f : D_{\mathbf{o}} \to \mathbb{R}.$$

For non-reduction operations, value_f is a coordinate remapping plus a scalar operation. For contraction-like operations, it is a summation over an auxiliary finite domain. The point of the formalism is not to invent new mathematics, but to make the implementation boundary crisp: shape logic is planned once, while value logic is evaluated many times in parallel.

3.3 Execution Protocol

The same protocol is used across the library:

$$\textsc{Validate} \to \textsc{Plan} \to \textsc{Allocate} \to \textsc{Fill}.$$

Validate checks the public contract. Plan builds the output shape and the small metadata object needed by a kernel. Allocate creates a dense output tensor with packed device storage. Fill writes metadata and data, either by launching a CUDA kernel or by performing a device-to-device copy for operations whose value rule preserves linear order.

This protocol is the main reason the codebase remains simple despite supporting several operation families. The special cases that do exist become ordinary instances of the same protocol: a rank-zero result is just an output shape $()$; a zero-extent input launches enough work to write metadata but no data values; a scalar operand is a kernel parameter rather than a synthetic tensor.

3.4 Layering

Cinder separates the system into four layers, shown in Table 3. The layers are intentionally asymmetric: the Python layer is small, while the native layer carries most of the mechanical responsibility. This is a good trade for a Python extension because it keeps the user-visible contract compact and lets C++ express ownership, layout, and launch invariants directly.

The storage/view split is especially important. `Tensor` owns device memory. `Shape` is a non-owning rank-and-extents view. `DenseRowMajorMapping` is a stateless mapping policy. `TensorView` combines a data pointer with a shape and mapping policy without owning memory. Thus ownership, shape interpretation, and coordinate linearization are not fused into one object.

3.5 Metadata and Kernel Plans

Each tensor allocation stores a header, extents, padding, and the data region in one CUDA allocation. A device-buffer descriptor contains only the allocation base and the offset of the data region. Kernels can reconstruct shape and data pointers from this descriptor without receiving separate metadata pointers.

This design has three useful consequences. First, tensor arguments are compact: inputs and outputs can be passed to kernels by value as descriptors. Second, device metadata travels with device data, so chained operations can consume kernel-produced tensors without an intermediate host query. Third, the host still keeps a copy of extents, making validation and output-shape derivation ordinary C++ code.

The cost is that metadata is duplicated between host and device. For a small runtime-rank library, that duplication is acceptable. It avoids a worse alternative: making every shape query require a device-to-host copy, or forcing kernels to depend on host-only shape objects. The design spends a few words of metadata to preserve a clean boundary between planning and execution.

Table 2: Operation semantics and design trade-offs.

Family	Semantic rule	Design trade-off
Elementwise	equal shapes, pointwise value rule	no implicit broadcast in arithmetic
Tensor-scalar	input shape, scalar parameter	avoids synthetic scalar tensors
Reshape	same $N(\mathbf{e})$, same linear order	materialized copy, not aliasing view
Broadcast	trailing-axis expansion	materialized output, not stride-zero view
Slice	coordinate window copied to dense output	simple ownership, higher copy cost
Concat	equal non-axis extents, axis sum	bounded compact multi-input plan
Transpose	axis permutation	physical reorder, not strided metadata
Tensor product	output shape concatenation	direct kernel, no vendor primitive
Contraction	sum over paired axes	generic rule, not GEMM-specialized
Mode multiply	matrix action along one axis	direct fiber loop, simple rank handling
Reduction	finite sum to rank-zero tensor	uniform tensor result, no host scalar path

Table 3: Design layers and their responsibilities.

Layer	Responsibility	Hidden from users
Python facade	constructors, methods, operators	native helper functions
Semantic planner	validation and output shape derivation	kernel-specific plans
Storage/view core	owned buffers, shapes, mappings, views	packed metadata layout
CUDA kernels	dense data-parallel evaluation	launch geometry and device pointers

Table 4: Key representation choices.

Choice	Benefit	Cost
Runtime rank	one API for all ranks	bounded plan arrays
Dense row-major	simple linearization	no layout polymorphism
Packed allocation	compact kernel arguments	duplicated metadata
Value semantics	predictable chaining	more materialization
Host-side plans	clear validation path	more launch-time work
Rank-zero scalars	uniform result type	scalar extraction is explicit

3.6 Public API Boundary

The public API is intentionally not a mirror of the C++ namespace. Python users see `Tensor`, constructors from a shape or dense row-major host values, metadata properties, explicit `to_list` transfer, arithmetic operators, shape operations, tensor-algebra operations, and reductions. They do not see free functions, mapping classes, views, launch helpers, or device-buffer descriptors.

This boundary makes the API more conservative than the implementation. The C++ code already contains a column-major mapping policy and non-owning views, but the Python contract does not promise layout polymorphism or view semantics. Keeping those tools internal prevents accidental user dependence on mechanisms that are not yet complete design commitments.

3.7 Why Not Lazy Views?

The most obvious missing feature is lazy view semantics. In many array systems, `reshape`, `transpose`, `slice`, and `broadcast` can be represented by modifying shape and stride metadata instead of copying data. Cinder chooses the opposite default: materialize dense results.

This is not because views are unimportant. It is because they change the system invariants. With lazy views, a tensor may be non-contiguous, may alias another tensor, may carry stride-zero axes, may outlive the base object in Python, and may require kernels to support arbitrary stride expressions. A single operation then has two meanings: a semantic value transformation and a storage-view transformation. Cinder currently keeps only the first meaning.

The trade-off is visible:

$$\text{materialization cost} \quad \leftrightarrow \quad \text{dense postcondition}.$$

After any public operation, the result is again a dense row-major tensor. That postcondition makes later kernels simpler and is the reason the system can support a moderately rich operation set without a complicated dispatcher.

3.8 Design Invariants

The design is held together by the following invariants.

I1: public tensors are dense row-major values. No public tensor represents a strided view, sparse object, symbolic expression, or deferred computation.

I2: shape errors are host errors. Invalid axes, incompatible extents, and value-count mismatches are rejected before device execution. Kernels assume a checked plan.

I3: every operation returns a tensor. Even inner products and norms return rank-zero tensors. This avoids a separate scalar result channel and keeps chaining uniform.

I4: metadata is sufficient on both sides of the boundary. The host has extents for planning; the device allocation has enough metadata for kernels to reconstruct views.

I5: extensions should preserve the protocol. Future dtype support, layout support, or view support should not be added by breaking the execution protocol defined above. If an extension cannot fit that protocol, it is not a local feature; it is a redesign of the system's semantic contract.

4 IMPLEMENTATION

Cinder instantiates the preceding design as a CUDA-backed Python extension written in C++23, CUDA C++, pybind11, and scikit-build-core. The public module, `cinder.core`, exports a single value type, `Tensor`; the native implementation supplies the hidden machinery for shape validation, device allocation, kernel planning, and CUDA launch. The implementation is deliberately not a thin binding over a collection of ad hoc kernels. It is a typed execution system in which C++ concepts define the legal static vocabulary, small algebraic objects encode runtime plans, and RAII binds CUDA resources to C++ object lifetimes.

4.1 Typed Representation and Policy-Based Layout

The native core is organized around typed products rather than a single catch-all descriptor:

$$\begin{aligned} \mathsf{Tensor} &= \mathsf{HostMeta} \times \mathsf{DeviceOwner}, \\ \mathsf{DeviceOwner} &= \mathsf{Storage} \times \mathsf{MappingType}, \\ \mathsf{TensorView}\langle E, M\rangle &= \mathsf{Ptr}\langle E\rangle \times \mathsf{Shape}, \\ E &\models \mathsf{TensorElement}, \qquad M \models \mathsf{TensorMapping}. \end{aligned}$$

Here E is the element type and M is a mapping policy. The public `Tensor` owns CUDA storage and fixes the current public interpretation to dense row-major `float32`. `Shape` and `TensorView` are non-owning products of small values. `DenseRowMajorMapping` and `DenseColumnMajorMapping` are empty mapping policies that implement coordinate-to-offset functions without storing per-view state. This means that layout is not an afterthought attached to a view at runtime; it is part of the native type vocabulary used to interpret a pointer and a shape.

This representation borrows the discipline of algebraic data types: compound objects are explicit products, alternatives are represented by small closed operation codes, and illegal combinations are excluded either by construction or by a checked host-side plan. C++ does not give Cinder a single ML-style datatype declaration, but the implementation follows the same modeling rule: the shape of each runtime value is visible in its type and in a compact constructor payload. Examples include

`TensorStorageHeader`, `TensorDeviceBuffer`, and the operation-specific plans for broadcast, slice, concat, transpose, contraction, and mode multiplication. This is the systems-programming analogue of the type-theoretic account of products and sums in typed programming languages [27].

C++ concepts make these representations executable contracts rather than comments. `TensorElement` restricts tensor values to object types; `IntegralIndex` restricts index parameters to integer types; `TensorShape` requires `rank()` and `extent(axis)`; `TensorMapping` requires an empty default-constructible policy; and `TensorMappingFor` requires that a policy map a shape and index tuple to a linear offset. This mirrors the purpose of concepts in generic C++: template requirements are stated as compile-time predicates over uses, not merely recovered from template instantiation failures [8].

Cinder uses policy-based design [2] at the exact point where dynamic dispatch would otherwise leak into every element access: layout. `TensorView<Element, Mapping>` is parameterized by a mapping policy, and element access is compiled as

$$\texttt{data[map}_M(\textit{shape}, \textit{indices})\texttt{]}.$$

Because the policy is empty, `TensorView` stores no policy object. The layout rule is selected by the type `Mapping`, so there is no virtual table, no runtime branch on layout, and no additional field in the view. The current public `Tensor` fixes dense row-major layout, but the native layer already demonstrates that layout is an internal policy dimension rather than a property fused with ownership.

Keeping the typed representation and layout policy together is more than editorial tidiness. The mapping policy is the static half of the representation; the runtime shape is the dynamic half. Splitting them would create a false boundary between "what object is this?" and "how is this object read?" even though every element access uses both. This is template metaprogramming in the narrow, performance-oriented sense: the compiler specializes the view for the selected mapping, while the runtime rank and extents remain ordinary data. Cinder therefore avoids two extremes. It does not encode every tensor rank in the C++ type, which would make the Python boundary impractical; it also does not erase layout behind an abstract base class. The resulting split is simple: static policy for the interpretation rule, runtime metadata for the tensor instance.

4.2 Host-Owned Storage and Planning Protocol

Cinder's public tensor is *host-owned* but *device-resident*. The owning C++ object lives in the CPython process, carries the host copy of the shape, and releases CUDA memory through RAII. The value payload and a compact metadata mirror live on the device. Each tensor is stored in one CUDA allocation:

$$B = \mathsf{Header} \parallel \mathsf{Extents} \parallel \mathsf{Padding} \parallel \mathsf{Data}.$$

The header stores rank and dense element count. The extents array stores the runtime shape. The data region is aligned to `alignof(float)`:

$$\mathsf{data\_offset} = \mathrm{align\_up}\big(|\mathsf{Header}| + r \cdot |\texttt{size\_t}|, \mathrm{alignof}(\texttt{float})\big),$$

where r is the tensor rank. A `TensorDeviceBuffer` contains only the allocation base and the data offset; kernels reconstruct the header, extent pointer, data pointer, shape view, and tensor view from that compact descriptor. The host-side `Tensor` keeps its own `std::vector` of extents for validation, shape queries, and output planning, so Python metadata access does not require a device-to-host query.

The constructor stages the header, extents, padding, and optional dense data in a host byte buffer, then performs one host-to-device copy for the entire packed allocation. This is intentionally aligned with CUDA performance guidance: host/device transfers should be minimized and batched when possible, because the host/device link is much slower than device memory bandwidth and per-transfer overhead is observable [11, 22]. The same principle appears throughout the implementation: scalar operands are passed by value, kernel plans are passed by value, concat passes input buffer descriptors inside the plan, and intermediate rank-zero reduction results remain device tensors until the user explicitly calls `to_vector()`.

Ownership follows RAII. `Tensor` calls `cudaMalloc` during construction and `cudaFree` in the destructor; copy construction performs a deep device-to-device copy of the packed allocation; copy assignment uses copy-and-swap; move construction and move assignment transfer the allocation pointer without copying data. This gives the Python-exposed value a simple contract: a live `Tensor` either owns exactly one packed allocation or is empty, and releasing a Python object cannot leak device memory. The design follows the C++ resource-management model in which object lifetime is the resource lifetime [36].

The same host ownership relation also explains the operation protocol described in Section 3. Every nontrivial operation rejects invalid shapes, axes, ranks, and sizes on the host; computes output extents, row-major strides, axis maps, and small fixed-size metadata arrays; creates an uninitialized packed output tensor; and finally writes metadata and data by a CUDA kernel or by a direct device-to-device copy when the value rule preserves dense linear order.

This protocol is the main device/host boundary inside the native implementation. Host code may throw C++ exceptions, manipulate vectors, select operation paths, and own CUDA allocation lifetimes. Device code receives a checked plan and runs a predictable data-parallel loop. Most plans are bounded by a maximum rank of 32 and are passed by value to the kernel, avoiding temporary device allocations for axis permutations, strides, slice starts, broadcast strides, or concat input tables. Table 5 summarizes the plan shapes.

Special cases are absorbed into the same protocol rather than creating a second semantic path. A rank-zero tensor is an ordinary tensor whose extent array is empty and whose element count is one. A zero-element tensor still launches at least one work item so thread zero can write output metadata. A reshape with a different shape initializes only output metadata and copies the dense data region device-to-device. A full slice or single-input concat degenerates into a deep tensor copy. These cases are special in cost, not in representation.

Table 5: Kernel plans used by operation families.

Family	Plan payload	Avoided runtime work
Broadcast	output extents, output count, input strides	synthetic stride-zero views on device
Slice	starts, output extents, input strides	per-thread bounds checking
Concat	input descriptors, axis offsets, output extents	device pointer-table allocation
Transpose	output extents, input strides by output axis	runtime axis search in the kernel
Contraction	free extents, contraction extents, free and contracted strides	device-side axis validation
Mode multiply	mode axis, input mode extent, input strides, output extents	reconstructing fibers from host data

4.3 CUDA Thread Assignment

Cinder's dense kernels use a one-dimensional launch geometry. For an operation with N output elements, the host computes

$$W = \max(N, 1), \qquad G = \left\lceil \frac{W}{256} \right\rceil,$$

where W is the number of work items and G is the number of thread blocks. Each block has 256 threads. In the kernel, each thread computes its global linear index

$$t = \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}.$$

If $t = 0$, the thread writes the output header and extents. If $t < N$, the thread computes one output element. Data threads do not read output metadata; they use input metadata and the by-value plan. Therefore, the metadata write does not require a grid-wide synchronization, which CUDA does not provide inside a normal kernel. This organization follows CUDA's grid/block/thread model, in which independent blocks are scheduled across streaming multiprocessors and threads identify their work using `blockIdx`, `blockDim`, and `threadIdx` [21, 23].

The per-thread value rule depends on the operation family:

$$\begin{aligned}
&\text{elementwise:} && y_t = \phi(x_t^{(1)}, x_t^{(2)}),\\
&\text{tensor product:} && y_t = x^{(1)}_{\lfloor t/|x^{(2)}| \rfloor} x^{(2)}_{t \bmod |x^{(2)}|},\\
&\text{transpose/shape:} && y_t = x_{\rho(t;\pi)},\\
&\text{contraction:} && y_t = \textstyle\sum_{k=0}^{K-1} x^{(1)}_{\rho_1(t,k)} x^{(2)}_{\rho_2(t,k)},\\
&\text{mode multiply:} && y_t = \textstyle\sum_{j=0}^{m-1} x_{\rho(t,j)} A_{\alpha(t),j}.
\end{aligned}$$

Here t is the output linear index, ϕ is the scalar operator, ρ functions are offset maps derived from a kernel plan, K is the number of contracted coordinate tuples, m is the selected input mode extent, and $\alpha(t)$ is the matrix row selected by the output coordinate on the mode axis.

Reductions use a different assignment. For small reductions, one block scans the input by thread-stride, stores one partial sum per thread in shared memory, and reduces the block with `__syncthreads()`. For larger reductions, the first kernel launches up to 4096 blocks; each thread traverses the input with a grid-stride loop

$$i = t,\ t + \texttt{gridDim.x} \cdot \texttt{blockDim.x},\ \ldots$$

and writes one partial sum per block. A second kernel reduces those partials into a rank-zero tensor. The grid-stride pattern preserves unit-stride memory access within warps while allowing a bounded grid to cover arbitrary input sizes [13]. For the L2 norm, the final square root is performed in the final reduction kernel, avoiding a host scalar round trip.

4.4 Launch and Transfer Economy

Cinder treats launch and transfer overhead as a first-class part of the algorithm, not as bookkeeping around it. In a heterogeneous runtime, the expensive mistake is to measure only the arithmetic in the kernel while ignoring the fixed boundary costs paid to enter the device, construct device-side metadata, or bring intermediate values back to the host. Prior GPU performance work makes this point sharply: the location of the data can dominate the apparent CPU/GPU comparison once transfer time is included [11]. CUDA's own optimization guidance gives the same engineering rule: minimize host/device transfer and batch small transfers whenever possible [22].

The implementation therefore optimizes for *device residency*. A tensor that enters Cinder should remain a CUDA-resident value across native operations, and metadata should move through kernel parameters or the packed allocation rather than through separate allocations and copies. For a public operation f, the implementation tries to realize the cost shape

$$C(f) \approx L_f + B_f^{\mathrm{H2D}} + B_f^{\mathrm{D2H}} + K_f(N),$$

where L_f is launch overhead, B_f^{H2D} and B_f^{D2H} are host-to-device and device-to-host transfer costs, and $K_f(N)$ is the device work over N dense output elements. Cinder cannot make $K_f(N)$ disappear, but it can keep the other three terms close to their minimum: one launch for most materializing operations, one packed H2D copy at construction, and zero D2H transfers unless the user explicitly asks for a Python value.

This choice creates a concrete set of negative space in the implementation: there is no separate metadata kernel, no temporary device array for a broadcast or transpose plan, no device tensor allocated merely to hold a scalar, no implicit host scalar extraction for rank-zero reductions, and no host round trip between chained tensor operations. Plans are ordinary by-value kernel arguments; concat embeds input buffer descriptors in the plan; scalar operands are passed as immediate values; output metadata is written by thread zero in the same kernel that writes output data. The design is small, but it is small in a specific direction: every removed launch and every avoided copy prevents the Python object model from reappearing as a performance tax at the CUDA boundary.

The resulting policy is deliberately more surgical than a general fusion compiler. Kernel fusion across arbitrary Python expressions could reduce launches further, but it would require deferred execution, alias analysis, graph-level scheduling, and a substantially larger semantic contract. Cinder instead performs *intra-operation fusion*: validation, layout planning, and metadata construction are host work; metadata emission and value computation are

Table 6: Launch and copy behavior of major implementation paths.

Path	Device work	Transfer behavior
Tensor construction	no kernel	one packed H2D copy for metadata and optional data
Elementwise/scalar	one kernel	scalar and operation code passed by value
Broadcast, slice, concat	one kernel	plan passed by value; concat embeds input descriptors
Transpose	one kernel	axis and stride metadata passed by value
Tensor product	one kernel	output metadata written by thread zero
Contract/mode	one kernel	summation performed inside output-element threads
Reshape	no kernel	metadata H2D plus data D2D copy
Inner product/norm	one or two kernels	rank-zero result remains on device until explicit D2H

co-located in the minimum number of device operations for that tensor primitive. This is a modest claim, but a systems-relevant one. The library does not pretend that a small tensor framework can win by raw FLOP/s alone; it wins, when it wins, by refusing to pay unnecessary boundary costs.

4.5 Python, FFI, and Build Boundary

The Python boundary is the outer heterogeneous boundary, not just a convenient wrapper. In CUDA C++, a source file can contain both host code and device code, but `nvcc` separates those paths during compilation: host code is compiled for the CPU, while device code is compiled to GPU code and embedded or linked into the resulting binary [23]. CPython can call only functions that are present as host-callable native symbols inside the loaded extension module. It cannot directly invoke a `__global__` kernel or a `__device__` helper as a Python function. Therefore the actual call chain is

Python
↓
pybind11 Host Wrapper
↓
C++ Host Plan/Launch
↓
CUDA Kernel

Cinder's `bindings.cpp` reflects that chain exactly. The pybind11 module entry registers the Python-visible `Tensor` methods; each method calls a host C++ member or free function; that host function validates, plans, allocates, and only then launches a CUDA kernel or performs a CUDA copy. Device kernels remain implementation details, not Python-callable API.

At the foreign-function interface level, `cinder.core` is a dynamically loaded CPython extension module. CPython's extension contract is concrete: an extension is a shared library loadable into the Python process, named with an interpreter-supported extension suffix, and exporting a module initialization function [31]. pybind11 supplies the boilerplate for this contract: `PYBIND11_MODULE` creates the import-time entry point, and `pybind11_add_module` builds a Python extension target with the platform- and interpreter-specific naming and linker

Table 7: Boundary chain from Python import to CUDA execution.

Layer	Contract and Cinder artifact
Python import	find a correctly suffixed `cinder.core` extension and its initialization symbol
CPython C API	create the module object through the pybind11-generated initializer
pybind11 wrapper	convert Python objects to C++ values and translate exceptions around the `Tensor` method bindings
Native host layer	validate, plan, allocate, and dispatch through `Tensor` methods and operation-specific host helpers
CUDA runtime/device	execute `.cu` kernels over packed device buffers through `CUDA::cudart`
Build system	make the Python, CMake, pybind11, CUDA, and wheel choices reproducible

details [28, 29]. Beneath CPython, the operating system dynamic loader resolves the extension and its dependent shared libraries; that is the ordinary linker/loader problem of binding symbolic references to loaded code and data [19]. For Cinder, the dependencies are not merely C++ runtime details: the CUDA runtime library must also be available, and the embedded or linked GPU code must match the target GPU policy selected at build time.

The result is an ABI contract, not source-level magic. Cinder does not use the CPython limited API or `abi3`; pybind11 exposes a normal CPython extension module whose compatibility is tied to the selected Python interpreter, compiler, platform, and dependent native libraries. CPython's Stable ABI exists precisely to reduce some of those version-coupling problems, but using it is an explicit design and build choice, with API and performance trade-offs [30]. In the current implementation the safer statement is narrower: the public Python ABI is the small `Tensor` surface, while the binary compatibility of the wheel remains a build artifact.

The build files are therefore part of the implementation, not packaging decoration. The project metadata selects scikit-build-core as the PEP 517 build backend and declares pybind11 as a build dependency. scikit-build-core bridges Python packaging to CMake, so a Python build frontend can configure a native CMake project and assemble the resulting extension into a wheel [34].

Cinder's CMake then finds the selected Python interpreter and module-development component. It asks that interpreter where pybind11's CMake package lives, builds the `core` module with pybind11's CMake helper, and installs it under the `cinder` package. When CUDA is enabled, the same target appends `.cu` translation units, defines the CUDA feature macro, finds the CUDA Toolkit, and links the CUDA runtime target.

That coupling is especially important in heterogeneous systems. A pure C++ extension mainly needs a host compiler, Python headers, and a loadable shared object. A CUDA-backed extension additionally needs a CUDA compiler path, CUDA runtime linkage, GPU architecture selection, and sometimes device-code linking. CMake's CUDA properties expose those choices explicitly: projects are encouraged to set CUDA architectures rather than inherit compiler-version defaults, and separable device compilation becomes necessary when device code must link across translation units [14, 15, 23]. For Cinder today, keeping kernels and their device helpers close to the operation source files avoids a separate device-linking contract. If future versions introduce

cross-translation-unit device libraries, precompiled GPU kernels, or multiple architecture wheels, that is not a local Python binding change; it is a build-system and ABI change.

This boundary is important for evolution. The implementation can add more mapping policies, dtypes, execution plans, or kernel strategies without promising them as Python surface area. Conversely, the user-visible `Tensor` remains a stable dense value: ownership is deterministic, transfers are explicit, and every operation returns another tensor that can immediately feed the next CUDA-backed operation. The only explicit public device-to-host value transfer is `to_list()` through `Tensor::to_vector()`; rank-zero reductions remain CUDA-resident tensors until the user asks for a Python list.

5 DISCUSSION

Cinder should be evaluated as a compact tensor-system prototype, not as a replacement for mature array frameworks. The important question is therefore comparative: which parts of the implementation already resemble durable systems structure, and which parts remain intentionally below the level expected from industrial libraries or recent tensor-compiler research? This section answers that question by separating current limits, design risks, and realistic next steps.

5.1 Current Limits

Implementation summary. The current implementation is eager, dense, CUDA-resident, and value-producing. A public `Tensor` owns one packed device allocation containing a header, runtime extents, padding, and a dense `float32` data region. The host keeps shape metadata for validation and planning, while kernels recover enough metadata from the packed descriptor to construct device-side views. Operations use the checked host-planning discipline defined earlier: plans are small by-value objects containing extents, row-major strides, axis maps, input descriptors, or reduction parameters. Shape operations and tensor algebra operations materialize dense row-major outputs rather than exposing lazy views. The Python surface is deliberately smaller than the native machinery: users see one type, `Tensor`, and do not see mapping policies, device buffers, kernel plans, streams, or storage descriptors.

This makes Cinder easy to reason about, but it also defines the main boundary of the current artifact. Table 8 places Cinder next to three stronger reference classes: mature Python array systems, specialized GPU tensor libraries, and compiler-oriented tensor systems.

Against mature array libraries. NumPy's central abstraction is not merely an n-dimensional buffer; it is an array object with shape, dtype, strides, broadcasting rules, views, and an ecosystem contract [12, 38]. PyTorch and TensorFlow add accelerator backends, automatic differentiation, graph or runtime execution machinery, and integration with model-building workflows [1, 26]. The Python Array API standard exists because library compatibility itself has become a design object [6]. Cinder has none of that breadth. Its public type does not represent a dtype lattice, a device object, a stream, a view, or a cross-library memory capsule. Its operators do not implement NumPy-compatible indexing, implicit arithmetic broadcasting, promotion rules, or in-place update.

That gap is not a single missing feature. It is the absence of a general array descriptor. In Cinder today, "tensor" means a host-owned dense row-major `float32` value on CUDA. In industrial libraries, "tensor" means a family of values parameterized by dtype, layout, device, aliasing relation, and sometimes gradient state. The former is a clean prototype invariant; the latter is the minimum vocabulary needed for real user programs.

Against specialized tensor libraries. cuTENSOR is a useful lower bound for what a production CUDA tensor primitive library considers ordinary: mixed precision, complex and real combinations, JIT compilation, support for up to 64-dimensional tensors, arbitrary layouts, contractions, reductions, permutations, plan caches, and multi-GPU variants [24]. Cinder's contractions and mode multiplications are semantically general, but their kernels are direct per-output loops rather than tiling, tensor-core, or vendor-library implementations. This is acceptable for an inspectable system study, but it means Cinder currently proves semantic coverage more than performance competitiveness.

The most important practical consequence is that Cinder cannot yet separate *what* operation is requested from *how* the implementation should be selected. Mature systems dispatch among kernels, vendor libraries, generated code, precision modes, and workspace choices. Cinder dispatches mostly by operation family and then launches a fixed kernel. The design is legible, but the optimization space is mostly hard-coded.

Against academic tensor-compiler practice. Recent systems research pushes the boundary in a different direction. Halide separates the algorithm from its schedule, making locality, parallelism, and recomputation explicit optimization dimensions [32]. TVM extends this idea to deep learning operators with graph-level optimization, operator scheduling, autotuning, and hardware-specific cost models [5]. TACO targets dense and sparse tensor algebra by compiling index notation into kernels, addressing the combinatorial explosion of tensor formats and operations [16]. MLIR treats compiler infrastructure itself as a reusable multi-level substrate for domain-specific computation [18], while Triton gives GPU programmers a tile-level abstraction for high-performance custom kernels [37].

Cinder intentionally stops before this compiler layer. Its host plans are small algebraic objects, but they are not an intermediate representation. They cannot be rewritten, fused, tiled, autotuned, serialized, or lowered to multiple backends. This is the core current limit from a research perspective: Cinder has a good local execution protocol, but it does not yet have a global optimization representation.

Limit conclusion. The current implementation is therefore best characterized as a semantically clean eager dense tensor library. Its strengths are value semantics, explicit shape planning, compact device metadata, and a small Python ABI. Its limits are equally clear: no dtype generality, no public view model, no sparse formats, no automatic differentiation, no graph/fusion layer, no backend abstraction, no interoperability protocol, no vendor-kernel dispatch, and no performance evidence. The positive reading is that these

Table 8: Cinder compared with industrial and research tensor systems.

Dimension	Mature systems	Cinder today
Data model	dtype families, strided views, device objects, standardizing APIs [6, 12, 38]	dense row-major `float32`; one public tensor type
Execution	eager runtimes, dataflow graphs, automatic differentiation, distributed execution [1, 26]	eager one-operation-at-a-time CUDA kernels; no graph or autograd
GPU library depth	mixed precision, arbitrary layouts, JIT plans, plan caches, multi-GPU paths [24]	handwritten kernels with fixed compact plans and a single CUDA device
Compilation	scheduling languages, autotuning, tensor IRs, sparse/dense code generation [5, 16, 18, 32, 37]	operation-specific host plans; no reusable IR or schedule representation
Interoperability	large ecosystems, serialization, packaging, backend conventions	pybind11 module with explicit `to_list()` transfer

omissions are visible and structured. The negative reading is that each omission corresponds to an entire subsystem in mature tensor software.

5.2 Design Risks

The main design risks are software-engineering risks rather than mathematical ones. Cinder's choices are mostly defensible tradeoffs; the danger is that future features could be added in ways that erase the clean boundaries that made the prototype understandable. Parnas's classical criterion for modular decomposition is still the right lens here: hide design decisions likely to change behind stable interfaces [25]. In Cinder, the likely-to-change decisions are dtype, layout, device, aliasing, dispatch, and kernel selection.

R1: descriptor drift. The packed allocation currently assumes a fixed element type and a dense row-major interpretation. If dtype or layout support is added by threading extra fields through every operation-specific plan, the representation will become brittle. The better boundary is a single tensor descriptor that owns element type, item size, alignment, layout, device identity, and possibly stream or allocator identity. The public `Tensor` can remain simple, but the native descriptor should become the one place where storage interpretation is defined.

R2: duplicated planning logic. The implementation already repeats small utilities: checked element-count products, row-major stride construction, fixed rank limits, work-item rounding, and metadata writes. This is not yet a serious problem because the operation set is small. It becomes a maintenance risk when new dtypes, layouts, or backends arrive: one missed overflow check or one inconsistent zero-element path would produce a semantic split that tests may not catch. The practical fix is modest: shared plan-building helpers and common launch utilities, not a full compiler rewrite.

R3: view semantics as a compatibility trap. Materialization is Cinder's strongest simplifier. After every public operation, the result is again a dense row-major owner. Lazy views would improve memory traffic for reshape, transpose, slice, and broadcast, but they introduce aliasing, base-object lifetime, non-contiguous access, stride-zero axes, and mutation questions. The risk is not that views are bad; the risk is exposing a partial view model and then being forced to preserve it. A safer path is to add internal view descriptors first, use them only inside kernels or fused operations, and delay any public view promise until lifetime and mutation rules are settled.

R4: performance special cases leaking into semantics. The current code handles special cases by keeping the representation uniform: rank-zero tensors are tensors, zero-element outputs still receive metadata, and scalar reductions return rank-zero device values. This taste should be preserved. Future performance work will tempt the codebase to add separate paths for matrices, vectors, scalars, contiguous copies, tiny tensors, and vendor-library calls. Those paths are often necessary, but they should be implementation choices under the same semantic contract. Once users can observe different error behavior, dtype behavior, synchronization behavior, or ownership behavior across optimized paths, the library has created accidental API surface.

R5: testing that proves examples rather than invariants. The current tests exercise many operations with reference computations, which is the right start. They are not yet enough for an extensible tensor library. The future risk is that tests stay example-based while the implementation grows descriptor dimensions. Shape algebra, broadcasting, slicing, contraction axes, zero extents, rank-zero values, and overflow boundaries are all better expressed as invariants and differential tests against a trusted reference such as NumPy. Performance tests should be separate from correctness tests, but both must exist before claims about maturity are credible.

Risk conclusion. The deepest maintainability risk is coupling by convenience: copying a small plan structure here, adding a dtype branch there, special-casing a fast path elsewhere. None of those changes is disastrous alone. Together they would make the current execution discipline decorative instead of governing. Cinder should therefore treat checked host planning followed by dense device filling as an architectural invariant, and any extension that cannot fit it should be recognized as a subsystem redesign.

5.3 Future Work

Future work should be incremental and evidence-driven. The next version does not need to become PyTorch, TVM, or cuTENSOR. It needs to strengthen the parts of Cinder that are already promising while avoiding commitments that would make later correction expensive.

F1: build a benchmark and profiling harness. The first priority is measurement. Cinder should report operation latency, effective bandwidth, launch count, H2D/D2H traffic, and device residency for a fixed matrix of shapes. The baseline set should include construction, `to_list`, elementwise arithmetic, broadcast, slice, concat, transpose, tensor product, contraction, mode multiplication, inner product, and norm. CUDA events can measure

device time; wall-clock measurements should include transfer and Python boundary costs. Comparisons should be honest: NumPy for CPU semantics, PyTorch or CuPy for common GPU behavior, and cuTENSOR for contraction, permutation, and reduction where applicable.

F2: introduce an internal tensor descriptor. Before adding new public features, the native layer should factor out a compact `TensorDescriptor`. It should record rank, extents, dtype, element size, layout kind, strides or dense mapping, device identity, and data offset. The first descriptor can still describe only dense row-major `float32`; the point is to move from implicit assumptions to one explicit object. This makes later dtype and layout support a local extension rather than a cross-cutting edit.

F3: add differential and property-based correctness tests. The current example tests should be complemented with generated shapes and axis sets. For each operation, Cinder can compare against NumPy reference semantics on small tensors, including rank-zero tensors, zero extents, singleton broadcast axes, repeated invalid axes, overflow-adjacent shapes, and empty concat inputs. The goal is not only to find bugs, but to make semantic contracts executable.

F4: add internal views before public views. A realistic next step is an internal strided view descriptor used by kernels and planning code. Reshape can become metadata-only internally; transpose and broadcast can be represented as stride transformations; materialization can remain the public postcondition. This gives Cinder a way to reduce copies inside compound operations without immediately promising NumPy-like aliasing to Python users.

F5: route selected operations through vendor primitives. Contractions and mode multiplications should keep the generic fallback kernel, but special cases that reduce to matrix multiplication or supported cuTENSOR contractions should dispatch to vendor routines. This is more practical than inventing a compiler first. The important constraint is semantic equivalence: the optimized path must return the same dense tensor contract, preserve the same error behavior, and be covered by the same tests.

F6: expand dtype support conservatively. Once the descriptor exists, Cinder can add a small dtype set: `float64` for numerical comparison, `int32` for indexing-like workloads, and possibly `float16`/`bfloat16` only when accumulation and conversion rules are specified. Dtype promotion should not be improvised. Either Cinder implements a documented small promotion table, or it keeps operations same-dtype-only until it can follow a broader array standard.

F7: document the API contract and version it. The report currently explains the design, but the package should also expose a short user-facing contract: supported dtype, device model, shape rules, transfer rules, error classes, synchronization behavior, and compatibility promises. This matters because even a small Python library develops userspace. Once users depend on behavior, changing it is a regression unless the contract warned them otherwise.

Future-work conclusion. The grounded roadmap is therefore measurement first, descriptor second, tests throughout, and only then wider dtype, view, and dispatch support. A compiler IR, graph fusion, automatic differentiation, distributed execution, or a public NumPy-compatible view model may eventually be valuable, but they are not the next honest step. Cinder's immediate engineering value is sharper: it can become a small but well-measured CUDA tensor library whose architecture is strong enough to explain exactly when a feature is local and when it is a redesign.

ACKNOWLEDGMENTS

I thank the pybind11 project for making the Python/C++ boundary feel much smaller than it otherwise would have been. My initial plan was to compile a native shared object manually and then build a separate Python wrapper around `ctypes`. pybind11's header-only design, CMake integration, and direct use of the CPython extension interface turned that plan into a much lighter and more maintainable binding layer.

I also acknowledge OpenAI Codex, in particular the GPT-5.5 xhigh model configuration, for substantial assistance in implementing Cinder and drafting this report. The collaboration was productive, though not always graceful: the model occasionally misunderstood the intended design badly enough to require firm correction, and the repository survived three `git reset --hard HEAD` recoveries along the way. The final code and report should therefore be read as an AI-assisted artifact with a human still serving as designer, reviewer, and occasional emergency rollback mechanism. Evidently, even in an era anxious about AI replacing programmers, a human can still provide some residual utility to the machine.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, USA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Andrei Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, Boston, MA, USA. https://books.google.com/books/about/Modern_C%2B%2B_Design.html?id=aJ1av7UFBPwC

[3] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, et al. 2021. On the Opportunities and Risks of Foundation Models. arXiv:2108.07258 [cs.LG] https://arxiv.org/abs/2108.07258

[4] Arthur Cayley. 1858. A Memoir on the Theory of Matrices. *Philosophical Transactions of the Royal Society of London* 148 (1858), 17–37. https://doi.org/10.1098/rstl.1858.0002

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA, USA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[6] Consortium for Python Data API Standards. 2024. *Python Array API Standard*. Consortium for Python Data API Standards. https://data-apis.org/array-api/latest/ Version 2024.12; accessed 2026-06-09.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing*

Systems 35. Curran Associates, Inc., New Orleans, LA, USA, 16344–16359. https://proceedings.neurips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

[8] Gabriel Dos Reis and Bjarne Stroustrup. 2006. Specifying C++ Concepts. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/1111037.1111064

[9] Albert Einstein. 1916. Die Grundlage der allgemeinen Relativitätstheorie. *Annalen der Physik* 354, 7 (1916), 769–822. https://doi.org/10.1002/andp.19163540702

[10] Hermann Günther Grassmann. 1844. *Die lineale Ausdehnungslehre, ein neuer Zweig der Mathematik*. Otto Wigand, Leipzig, Germany.

[11] Chris Gregg and Kim Hazelwood. 2011. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. IEEE, Piscataway, NJ, USA, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[12] 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, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array Programming with NumPy. *Nature* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[13] Mark Harris. 2013. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. NVIDIA Technical Blog. https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

[14] Kitware, Inc. 2026. *CMAKE_CUDA_ARCHITECTURES*. CMake. https://cmake.org/cmake/help/latest/variable/CMAKE_CUDA_ARCHITECTURES.html CMake 4.3 documentation; accessed 2026-06-09.

[15] Kitware, Inc. 2026. *CUDA_SEPARABLE_COMPILATION*. CMake. https://cmake.org/cmake/help/latest/prop_tgt/CUDA_SEPARABLE_COMPILATION.html CMake 4.3 documentation; accessed 2026-06-09.

[16] 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, Article 77 (2017), 29 pages. https://doi.org/10.1145/3133901

[17] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500. https://doi.org/10.1137/07070111X

[18] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*. IEEE Press, Seoul, Republic of Korea, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[19] John R. Levine. 2000. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, USA. https://openlibrary.org/books/OL8606474M/Linkers_and_Loaders

[20] Lek-Heng Lim. 2021. Tensors in Computations. *Acta Numerica* 30 (2021), 555–764. https://doi.org/10.1017/S0962492921000076

[21] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[22] NVIDIA Corporation. 2026. *CUDA C++ Best Practices Guide*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html Accessed 2026-06-08.

[23] NVIDIA Corporation. 2026. *CUDA C++ Programming Guide*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Accessed 2026-06-08.

[24] NVIDIA Corporation. 2026. *cuTENSOR: A High-Performance CUDA Library for Tensor Primitives*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cutensor/latest/ Accessed 2026-06-09.

[25] David L. Parnas. 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (1972), 1053–1058. https://doi.org/10.1145/361598.361623

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., Vancouver, BC, Canada, 8024–8035. https://papers.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[27] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA. https://mitpress.mit.edu/9780262162098/types-and-programming-languages/

[28] pybind11 Development Team. 2026. *Build Systems*. pybind11. https://pybind11.readthedocs.io/en/stable/compiling.html Accessed 2026-06-09.

[29] pybind11 Development Team. 2026. *pybind11 Reference*. pybind11. https://pybind11.readthedocs.io/en/stable/reference.html Accessed 2026-06-09.

[30] Python Software Foundation. 2026. *C API Stability*. Python Software Foundation. https://docs.python.org/3/c-api/stable.html Python 3.14 documentation; accessed 2026-06-09.

[31] Python Software Foundation. 2026. *Defining Extension Modules*. Python Software Foundation. https://docs.python.org/3/c-api/extension-modules.html Python 3.14 documentation; accessed 2026-06-09.

[32] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[33] Gregorio Ricci-Curbastro and Tullio Levi-Civita. 1901. Méthodes de calcul différentiel absolu et leurs applications. *Math. Ann.* 54 (1901), 125–201. http://eudml.org/doc/157997

[34] Henry Schreiner, Jean-Christophe Fillion-Robin, and Matt McCormick. 2024. Scikit-build-core: A Modern Build-Backend for CPython C/C++/Fortran/Cython Extensions. Proceedings of the 23rd Python in Science Conference. https://doi.org/10.25080/FMKR8387

[35] Stanford Institute for Human-Centered Artificial Intelligence. 2026. *The 2026 AI Index Report*. Technical Report. Stanford University. https://hai.stanford.edu/assets/files/ai_index_report_2026.pdf Accessed 2026-06-09.

[36] Bjarne Stroustrup. 2013. *The C++ Programming Language* (4 ed.). Addison-Wesley Professional, Boston, MA, USA. https://www.stroustrup.com/4th.html

[37] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[38] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30. https://doi.org/10.1109/MCSE.2011.37

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., Long Beach, CA, USA, 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html