LOOM: A GPU COMPILER FOR SCALAR FUNCTIONS OVER FINITE INDEX MAPS

Anonymous authors
Paper under double-blind review

ABSTRACT

AI for science applications often mix two kinds of computations: dense matrix operations that benefit greatly from GPU architecture, and application of small scientific formula on an irregular, indexed data structure, that is a poor candidate for the modern GPU if implemented naively. During the evaluation of the scientific formula, separate kernels for indexed read, arithmetic and reductions may cost more time moving the intermediate arrays than the actual compute. We introduce Loom, a GPU compiler for scalar functions over finite index maps. The user supplies a fixed-shape JAX function and a description of where each array element lives; Loom fuses the computation into GPU kernels specialized around one contiguous axis of independent lanes for vectorization. It accepts declared algebraic constants such as $\sqrt{3}/2$ and rewrites polynomial arithmetic so that it can apply exact cancellations. Loom lowers programs, including those built by JAX automatic differentiation, through StableHLO, generates many PTX implementations and selects one based on timing. We compare Loom to existing domain-specific libraries: on NVIDIA B200 and A100 GPUs, benchmarked for two sparse equivariant operations that are widely used in machine-learned interatomic potentials, Loom-generated hot kernels are up to approximately 2× faster. By lowering the optimization to the compiler level, Loom trades a costly first call, paid once per program, for fast kernels specific to the shapes of a given problem. Loom shows that specialized compilers can complement performance-driven domain-specific AI for science libraries.

1 INTRODUCTION

A significant portion of scientific computation can be characterized as the application of small algebraic formulas to large, irregular, indexed structures. This is because many of these formulas were refined by practitioners over decades from symmetry, group theory, conservation laws, and similar foundational principles, which keeps them small yet generally applicable. This is also why their coefficients are often exactly representable, as in square roots of rationals. In the computational applications of these formulas, functions are often applied over big, sparse, data-dependent structures, such as atomic coordinates, connectivity described on a mesh, lists of bonds, angles, dihedrals. The structures where the function is applied also often change from call to call. For example, in molecular dynamics, the interaction neighbourhood of the atom may change over time, while the formula for interaction stays the same.

Loom compiles these small, fixed-shape functions into GPU kernels. By specifying the formula separately from the placement of its irregular indexed data, it enables fusion, which reduces the overhead of multiple kernel launches and intermediate arrays. Loom applies wherever computation has small enough fixed internal shapes and a wide uniform, independent axis to vectorize over.

Machine-learned interatomic potentials (MLIPs) have both properties: small fixed internal shapes such as tensor products and an independent axis for vectorization such as channels, and serve as an appropriate evaluation workload for Loom. They predict atomic energies and obtain forces by differentiation, approximating costly electronic-structure calculations. Early MLIPs used designed local descriptors (Behler and Parrinello, 2007); newer models learn geometric features that transform correctly under rotations, reflections, and translations (Batatia et al., 2022; Batzner et al., 2022; Deringer et al., 2019). These models combine short feature blocks with sparse, fixed coefficients, gather neighbouring features, and reduce into destination atoms. XLA delegates dense matrix multiplication to mature libraries (OpenXLA Authors, n.d.-b), but for small tensor products, polynomial

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

contractions, indexed reads, and reductions, separate kernels can cost more in launches and intermediate arrays than in arithmetic.

Several systems avoid this cost by specializing around known operators. cuEquivariance provides CUDA kernels for segmented polynomials (NVIDIA Corporation, n.d.-a); e3nn describes equivariant tensor products as sparse paths and generates a contraction graph (Geiger and Smidt, 2022); OpenEquivariance produces CUDA or HIP kernels for Clebsch–Gordan tensor products and can fuse graph convolution (Bharadwaj et al., 2025); FlashTP fuses sparse, path-aggregated tensor products with graph reduction (Lee et al., 2025); Sobek streams radial projection, tensor products, and reduction through forward and derivative kernels (Chorošajev and Bény, 2026). Their interfaces, however, do not accept arbitrary JAX code or freely combine scalar operations with gathers and reductions.

General-purpose compilers cover parts of this problem. KeOps compiles symbolic formulas reduced over two index variables (Charlier et al., 2021). TACO and SparseTIR compile tensor-algebra expressions over declared sparse formats (Kjolstad et al., 2017; Ye et al., 2023). TVM and Halide separate an algorithm from the hardware-specific series of code transformations that define its execution (Chen et al., 2018; Ragan-Kelley et al., 2013). Triton and Pallas compile tile kernels that the programmer writes (JAX Authors, n.d.; Tillet et al., 2019), and Taichi and Dr.Jit fuse traced scalar programs with gathers and atomic scatters (Hu et al., 2019; Jakob et al., 2022). GNN compilers fuse message passing over one sparse graph (Hu et al., 2020; Wu et al., 2021; Xie et al., 2022; Zhou et al., 2023), and PyTorch Geometric reduces edges into nodes by segment (Fey and Lenssen, 2019). Unlike these systems, Loom derives both the indexed reads and the reductions from a declared graph of index maps, for every program JAX differentiation produces.

Loom takes a JAX function, lowered through StableHLO, together with this graph, and emits NVIDIA PTX kernels that fuse supported fixed-shape formulas with their indexed reads and sums, specialized around one contiguous axis of independent lanes. Before scheduling, it rewrites polynomial arithmetic with exact coefficients to expose cancellations, and bounded equality saturation (Tate et al., 2009; Willsey et al., 2021) keeps several equivalent forms instead of committing to the first rewrite, leaving the choice to measurement.

The main contributions of this work are: (i) a reusable compiler contract, rather than a new model operation, for fixed-shape formulas with any number of inputs and outputs over finite domains linked by index maps; (ii) reductions derived from the declared maps, with a domain protocol that can replace contended scatter updates with ordered walks over incoming entries (*fiber walks*); (iii) a path from JAX and StableHLO to scalar GPU programs, including programs created by JAX automatic differentiation; (iv) polynomial rewriting over the reals followed by register-aware partitioning and data-driven tuning; and (v) a performance evaluation on MLIP operators alongside cuEquivariance, OpenEquivariance, and FlashTP on NVIDIA B200 and A100 GPUs.

2 The Loom contract

The user writes a *lane function*, a JAX function for one logical position and one hidden lane; a domain graph of index maps relates its inputs and outputs to physical data.

In the call below, `owners[s]` is the group owning sample s. Loom gathers group data, forms a scaled outer product per sample and lane, and sums the results into groups; `exact` declares the value $\sqrt{3}/2$ to Loom's optimizer under the real-arithmetic contract below. The rest of this section defines each piece.

```
from fractions import Fraction

import jax
import jax.numpy as jnp
import numpy as np
from algebraic_real import pow
from algebraic_real.jax import exact
import loom

_d = loom.domains
n_group, n_sample, u = 2, 5, 32
owners = jnp.array([0, 1, 0, 1, 0], dtype=jnp.int32)
rng = np.random.default_rng(0)
group_values = rng.standard_normal((n_group, 16, u), dtype=np.float32)
sample_vectors = rng.standard_normal((n_sample, 3, u), dtype=np.float32)
```

```
graph = loom.DomainGraph(
    {"sample": n_sample, "group": n_group},
    {("sample", "group"): loom.IndexMap(owners, may_have_unmapped=False)},
)

def transform(operands):
    group_value, sample_vector = operands
    scale = pow(3, Fraction(1, 2)) / 2
    exact_scale = exact(scale, dtype=jnp.float32)
    return (jnp.outer(sample_vector, group_value) * exact_scale).reshape(-1)

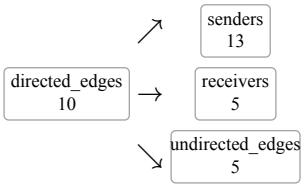
aggregated = loom.call(
    transform,
    (group_values, sample_vectors),
    domain_graph=graph,
    input_domains=(_d["group"], _d["sample"]),
    output_domains=_d["group"],
    u=u,
)

assert aggregated.shape == (n_group, 48, u)
reference = jnp.zeros_like(aggregated).at[owners].add(
    jax.vmap(jax.vmap(transform, in_axes=1, out_axes=-1))(
        (group_values[owners], sample_vectors)
    )
)
assert jnp.allclose(aggregated, reference, rtol=1e-6, atol=1e-6)
```

On the B200, this block's relative error $\max \frac{|A-R|}{\max}|R|$ against reference R is 1.15×10^{-7} in `float32`.

Domains, index maps, and fibers. A domain is a named, nonempty finite set declared by its size. An index map is a possibly partial function $\rho : A \to B$, stored as a `target_indices` array with one entry per element of A: its index in B, or -1 if it is unmapped. The example below has ten directed-edge slots, two of them padding. Three maps give each valid edge's sender, receiver, and undirected edge; opposite orientations share an `undirected_edges` index. In the sketch, each arrow points from a map's source domain to its target domain.

```
maps = {
    ("directed_edges", "senders"): loom.IndexMap(
        jnp.array([0, 2, 4, 6, 8, 10, 12, 1, -1, -1]),
        is_injective=True,
    ),
    ("directed_edges", "receivers"): loom.IndexMap(
        jnp.array([0, 0, 0, 0, 1, 2, 2, 2, -1, -1]),
        is_sorted=True,
    ),
    ("directed_edges", "undirected_edges"): loom.IndexMap(
        jnp.array([0, 0, 1, 1, 2, 2, 3, 3, -1, -1]),
        is_sorted=True,
    ),
}
graph = loom.DomainGraph(
    {"senders": 13, "receivers": 5,
     "directed_edges": 10, "undirected_edges": 5},
    maps,
)
```



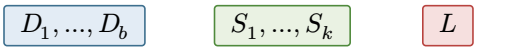
By default, an `IndexMap` may leave elements unmapped and is neither injective nor sorted. `is_injective=True` promises distinct targets for distinct mapped sources, `is_sorted=True` nondecreasing valid targets followed by any -1 padding, and `may_have_unmapped=False` a total map. These promises are trusted, not checked (Appendix D.1 prices a check).

The maps must form a directed acyclic graph with at most one directed path between two domains and at most one map per ordered pair, so senders and receivers are separate domains even when they enumerate the same physical set. `DomainGraph` checks sizes and topology and precomputes a `Preimage` for every direct map and every multi-hop path a call crosses. Its `source_indices` group mapped sources by target, and the fiber $\rho^{-1}(b)$ of target b, the set of sources mapped to it, is the range $[\text{offsets}[b], \text{offsets}[b+1])$; in the example, the fiber of receiver 0 holds directed edges 0–3.

When several maps share a source domain, `aligned_target_indices` stores the other targets in the same source order.

Instead of atomically adding each contribution at its target, Loom can give each target one worker that walks its fiber contiguously, accumulates locally, and writes once: the owner-computes aggregation that PyTorch Geometric performs by segment (Fey and Lenssen, 2019). This avoids contention but is not guaranteed: traversal, workspace limits, or partitioning may require another strategy.

Lane functions and array layout. Every input or output leaf has three consecutive groups of axes: the domain axes, the internal axes, and the lane axis.



All leaves share the same number b of domain axes, each governed by a domain or left as an unindexed singleton, and the lane extent is $L \in \{1, u\}$. Extent-one lane inputs broadcast over u; an output keeps all u lanes or, with `reduce_lanes=True`, sums them into one, the transpose of that broadcast. Domain placement likewise decides whether an output is written directly or reduced across domain positions. Lanes share control flow and indexing and never depend on one another, so their contiguity permits packing and vectorization without dependence analysis, and block widths can be chosen independently of the formula. Internal axes carry no such guarantee: the lane function may manipulate them as small fixed-size arrays, and Loom scalarizes them instead, exposing constants and cancellations across operation boundaries. It therefore targets short fixed shapes and leaves large dense matrix products to XLA and vendor libraries.

Meaning of a call. Loom iterates one domain per domain axis a. Let U_a hold the domains that operands declare there: an empty U_a gives a singleton axis, a single domain is iterated directly, and otherwise Loom iterates the unique most-specific domain that reaches every member of U_a, preferring a member of U_a, or rejects the call. Each operand is then addressed along the unique path from the iterated domain, by the identity, or at index zero on an unindexed axis (Appendix A).

Each output is then summed over the fiber of iterated positions that reach it. With $x_i(q, \ell)$ the value input i presents at iterated position q and lane ℓ, $\sigma_j^{-1}(p)$ the fiber of destination p, and $\Lambda_j(\ell')$ either $\{\ell'\}$ or, when output j sums its lanes, all u lanes,

$$Y_j(p, s, \ell') = \sum_{\ell \in \Lambda_j(\ell')} \sum_{q \in \sigma_j^{-1}(p)} \left[f_j\big((x_i(q, \ell))_i \big) \right]_s .$$

This single additive rule accounts for every reduction Loom performs, whether over a map fiber, over an iterated axis the output leaves unindexed, or over the u lanes; with the addressing above it also covers gathers, lane-local computation, and broadcasting (Appendix A). The opening call of this section is one instance: its only map sends each sample to its group, so the input `group_values` is gathered along it and the output is summed over each group's fiber.

A call takes the function, physical arrays, domain graph, the domain placement of every input and output, and optional compilation settings; inputs and outputs may be arbitrary JAX pytrees. Loom traces the function to infer output structure and internal shapes, and infers u from the last extents that are not one, which must agree; if all are one, u must be given. Captured constants become shared values.

Accepted operations. JAX traces the function at its internal shapes and lowers it to StableHLO, a portable operation set for machine-learning compilers (OpenXLA Authors, n.d.-a), which Loom translates into its own intermediate representation, XIR. The supported operations cover the intended fixed-shape formulas:

- elementwise arithmetic, comparisons, `select`, and `clamp`;
- `reshape`, `broadcast`, `transpose`, static slicing, concatenation, and padding;
- reductions, `dot_general`, `gather`, and `scatter`; and
- `sort` and `while` regions only when static lowering can eliminate them.

Scalar types are Boolean, signed and unsigned integers, `float16`, `bfloat16`, `float32`, and `float64`. Convolution, FFT, random-number, collective, dynamic-shape, and arbitrary custom-call operations, unknown composites, and loops with undetermined trip counts are rejected with their StableHLO name and shapes, never silently passed to another compiler. The supported StableHLO subset, not JAX tracing, is therefore the boundary of Loom's input language.

Derivatives. JAX automatic differentiation (AD) determines derivatives and saved primal values (Bradbury et al., 2018) Loom compiles the resulting value, tangent, and cotangent programs under the same domain and lane contract, through StableHLO and the same compiler, including under repeated AD. For example, JAX turns $y = x\sin(x)$ into $\dot{y} = (\sin(x) + x\cos(x))\dot{x}$. Reverse mode swaps gathers and reductions: the cotangent of an input read along a map is a sum over that map's fibers. Differentiation therefore carries domain placement explicitly, exchanging input and output placements where needed, rather than reconstructing it from array shapes. Integer, Boolean, and other non-differentiable leaves remain auxiliary data.

Arithmetic over the reals. An algebraic real number is a real root of a nonzero polynomial with integer coefficients, such as a rational or $\sqrt{2}$. A *polynomial region* is a maximal connected subgraph of additions, subtractions, multiplications, and negations. Values entering it from loads, conversions, comparisons, transcendental functions, or control flow are its variables, and its constants are its coefficients, so each of its outputs is a polynomial with algebraic real coefficients. A finite floating-point constant denotes the rational number its bits encode. Irrational constants such as $\sqrt{2}$ have no floating-point form, so `algebraic_real.jax.exact` gives JAX a rounded value while declaring the intended value to Loom's optimizer. The example combines a declared value with multiplication by zero under Loom's real-arithmetic contract; its only input axis is the lane axis, with $u = 8$.

```
from fractions import Fraction

import jax.numpy as jnp
from algebraic_real import pow
from algebraic_real.jax import exact
import loom

x = jnp.array([jnp.inf, 1, 2, 3, 4, 5, 6, 7], dtype=jnp.float32)

def identity_residual(value):
    sqrt2 = exact(pow(2, Fraction(1, 2)), dtype=value.dtype)
    return value * 0 + (sqrt2 * sqrt2 - 2)

jax_result = identity_residual(x)
loom_result = loom.call(identity_residual, x)

assert jnp.array_equal(
    jax_result, jnp.array([jnp.nan] + [-1.1920929e-7] * 7), equal_nan=True
)
assert jnp.array_equal(
    loom_result, jnp.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
)
```

JAX follows IEEE-754: at $x = \infty$, multiplication by zero is NaN, and the other seven values keep a residual from the rounded square root. Loom simplifies $x \times 0$ to 0 and $\sqrt{2}\sqrt{2}$ to 2, so all eight outputs are exactly zero. Every rewrite Loom applies to a polynomial region is exact and happens at compile time (Section 3 lists them); kernels still compute in the requested floating-point type. This is the real-arithmetic contract: Loom evaluates an expression equal to the program over the reals, not the IEEE-754 evaluation order. Special values do not survive: $x \times 0$ and $x - x$ are 0 even for an infinity or a NaN, and signed zeros do not survive reassociation. Last bits may differ from the IEEE-754 order, sometimes for the better: $(10^{-8} + 1) - 1$ returns 10^{-8}, not 0. Loom therefore targets finite inputs; since a NaN can cancel inside a polynomial region, divergence checks belong outside the call. The second assertion specifies the CUDA back end; the non-CUDA reference lowering returns the IEEE-754 result of JAX.

3 Compilation and Execution

We follow the call that opens Section 2 through Loom, from the moment JAX meets it to the moment its kernel runs. Loom is a thin Python package over a Rust core. When the user invokes `loom.call`, the Python side traces `transform` with `jax.make_jaxpr` on per-lane shapes, turns closed-over arrays into extra inputs, and binds a JAX primitive whose parameters hold this lane Jaxpr, the domain layout, and the compile options. JAX then treats the call like any other operation, with abstract evaluation, a batching rule for `vmap`, and a rule that prunes unused outputs. The derivative rules let JAX alone define derivatives: they apply `jax.jvp` or `jax.linear_transpose` to the lane function and bind a new Loom primitive on the traced result, so Loom compiles a derivative like any other program.

StableHLO, as JAX lowers `transform`

```
%cst = stablehlo.constant dense<0.866025388> ...
%0 = stablehlo.composite
       "algebraic_real.exact_constant" %cst {...}
%5 = stablehlo.multiply %3, %4 : tensor<3x16xf32>
%8 = stablehlo.reshape %7 : ... -> tensor<48xf32>
```

XIR, one lane

```
func @main(%0: f32[16], %1: f32[3]) -> f32[48] {
  %3 = constant {exact=[√3/2]} : f32[]
  %8 = multiply %6, %7 : f32[3,16]
  %10 = multiply %8, %9 : f32[3,16]
  %11 = reshape %10 {out_dims=[48]} : f32[48]
```

Device IR, fiber walk of the winner

```
op *140 step loop s32 per-thread [%58 %62 %63 ...]
op *60 operand offset [preimage_source_indices_0 _k] [%95]
op *61 operand load lanes 1 nc false ... [%95] [%96]
op *76 step bind [%110] [loom_v2_0]
op *80 operand bin add [loom_fiber_acc_0_0 loom_v2_0] [%115]
op *81 step bind [%115] [%116] into [loom_fiber_acc_0_0]
```

PTX, sm_120

```
$Lloop_5: ...
ld.global.nc.f32 %f31, [%rd26];
add.rn.f32 %f1, %f1, %f31;
$Ldone_6: ...
ld.global.nc.f32 %f33, [%rd36];
mul.rn.f32 %f34, %f33, %f1;
mul.rn.f32 %f35, 0f3F5DB3D7, %f34;
st.global.L1::evict_first.f32 [%rd39], %f35;
```

Listing 1: The call of Section 2 at four stages; ... marks a cut. XIR holds $\sqrt{3}/2$ exactly; the winner's device IR, shown with its value names, adds each sample value of the fiber into a register, and its PTX multiplies by the group value and the rounded constant only after the walk.

Nothing is compiled until JAX lowers the enclosing program. On a CUDA device, the lowering rule turns the lane Jaxpr into StableHLO; elsewhere it emits the pure-JAX reference. Loom reads StableHLO rather than the Jaxpr because it is a small, versioned operation set in which JAX has already reduced its many array APIs to a few operations such as `dot_general` and `gather`. A walker built on JAX's MLIR Python bindings visits the module and builds XIR through PyO3 bindings. In Listing 1, `jnp.outer` becomes a broadcast and a multiplication over a 3×16 tile, and `exact` becomes a composite that carries $\sqrt{3}/2$ beside its `float32` rounding, so XIR holds the algebraic number itself. The walker also accepts `sort`, which becomes a stable compare-and-swap network, and `while`, which is unrolled when constant propagation reads its trip count.

From here on, everything runs in Rust. AI agents wrote the implementation (see the disclosure of AI use), and Rust's static types suited this: type errors and non-exhaustive matches caught many agent mistakes at compile time, though numerical correctness still rests on tests and the oracle. The core first lifts the lane program over the domains: the read of `group_values` through `owners` becomes a symbolic gather along the index map, and the output placement becomes a sum over each group's fiber. Scalarization then unrolls every fixed internal array into scalar values, while the lane axis stays whole. Our call becomes 48 scalar outputs per lane, each a fiber sum of one sample component times one group component times $\sqrt{3}/2$.

Rewriting comes next, in one typed e-graph (Tate et al., 2009; Willsey et al., 2021) held by egglog, an equality-saturation library written in Rust (Zhang et al., 2023). Its nodes are exact algebraic coefficients, arithmetic, selections, gathers along index maps, and fiber sums. Saturation is bounded and driven by proposals: in each round, Rust code reads the classes and proposes equalities, such as polynomial factorizations or the law that a gather constant on each fiber leaves the fiber sum, and egglog merges each admitted proof and maintains congruence. Rounds stop at a fixed point, after eight rounds, or after 16,384 new nodes. Extraction keeps up to 16 alternatives on the Pareto front of two costs: work before accumulation, paid per sample, and work after it, paid per group. Here the front spans the written form, which spends 15,360 multiplications on the five samples, and the factored form $\sum_{s\in\rho^{-1}(g)}(v_s \otimes w_g)c = \left(\left(\sum_{s\in\rho^{-1}(g)} v_s\right) \otimes w_g\right)c$, with sample vectors v_s, group values w_g, and $c = \sqrt{3}/2$, which spends 6,144 on the two groups. Which is faster depends on the fiber sizes, so Loom keeps both and lets measurement decide. TENSAT applies a related bounded method to tensor graphs (Yang et al., 2021). Herbie also rewrites floating-point expressions with e-graphs, but for accuracy rather than cost (Panchekha et al., 2015) none of the general-purpose compilers discussed in the introduction rewrites polynomial arithmetic over the reals to prove cancellations before rounding.

Each alternative then leaves the symbolic world. Gathers and fiber sums become index arithmetic over the index-map arrays, and the promises remove work: a sorted map avoids index loads in fiber walks, an injective map proves writes collision-free, and a total map removes -1 guards. A deterministic pass pipeline follows. It applies neutral and absorbing elements under the real-arithmetic contract (Section 2), and ends with common-subexpression and dead-code elimination, since one proven zero can disconnect much of the graph. In a function with more than 65,536 polynomial operations or a chain deeper than 512, it also expands each polynomial region into exact terms and

rebuilds it as an evaluation tree, keeping the tree only if it saves more than a third of the weighted cost. The options `max_algebraic_work` and `max_algebraic_expression_work` bound all this symbolic work without changing semantics; an expression that exhausts its budget keeps its last valid form.

Loom now builds kernels. When thousands of statements remain, it first re-rolls wide sums over identical terms into loops over operand tables, unless a loop would hide a provable zero that lets a kernel skip work. Maximum fusion can spill registers into slow local memory, so the partitioner splits the scalar program by estimated statements, live values, and memory footprint, within `max_workspace_bytes` and `max_kernel_statements`. Every output is then a fold by key. In our call, each sample contributes to one key, its group: a key that no two samples share would be stored directly, and a shared key would be combined with `red.add` atomics, optionally spread over buckets that a second kernel sums. Each accumulation follows its last dependency, which shortens live ranges. Our call instead gets one kernel with a fiber walk. Loom stores each map's preimage as the samples sorted by group with one offset per group; one warp per group reads its samples through these offsets, sums them in registers, and writes each output once, without atomics.

Each plan is rendered many times, because the best launch shape depends on the GPU and on the data. Candidates range from 32 to 256 threads per block, and may split a group's 48 outputs across grid rows, so that several blocks walk the same fiber; others add vector loads, zero-skip guards, or register caps. Loom emits PTX (NVIDIA Corporation, n.d.-b) rather than CUDA C or Triton because its kernels are long straight-line programs without tiles, whose instruction selection and register limits Loom sets itself. A typed device IR is verified and rendered to PTX directly, with packed `f32x2` arithmetic where the GPU has it and register caps as `.maxnreg` directives. Our call has 16 alternatives, two statement orders each, and 25 kernels per pair: 800 candidates.

All candidates enter one *recipe*, a versioned binary description shaped like a CUDA graph: kernel launches, memsets, and copies with their dependencies, buffer sizes, and resource bounds. A *choice* node holds the candidates for one step, and the runtime keeps one of them. Loom itself runs nothing on the GPU. It freezes a shortlist in advance: it estimates each candidate's work per warp that the GPU can keep resident, ranks candidates by this estimate with a fixed-seed random permutation, and admits them until their predicted time, the larger of assembly time over three workers and 4 ms of tuning each, reaches `shortlist_budget_ms`. Measurements never feed back, so one program yields the same shortlist on every machine; ours keeps all 800 candidates, for a predicted 8.4 s.

The recipe returns to JAX as an XLA foreign-function call through `jax.ffi`, with a workspace buffer as an extra output, so XLA allocates every device buffer. On the first execution, the runtime assembles the PTX with nvJitLink, `ptxas`, or the driver JIT, through a disk cache and a worker pool, and rejects candidates that exceed their declared register or local-memory bounds. It times the others on the call's actual buffers with CUDA events, taking the median of five batches after three warm-up runs and dropping candidates far slower than the best. It then loads the winner with `cuLibraryLoadData`, captures the recipe as one CUDA graph, and replays it on every later call.

On an RTX 5080, our call takes 3.3 s from tracing to result. The factored form wins, with 128 threads per block and the outputs split across two grid rows; Listing 1 shows its walk, where `0f3F5DB3D7` is the `float32` rounding of $\sqrt{3}/2$. Inside `loom.aot`, the tuned recipe is cached under the program, the options, the GPU, and a workload identity that the user states, since the winner depends on the input distribution. Hot timings exclude tracing, compilation, assembly, and tuning.

4 Evaluation

Setup. We evaluate two sparse equivariant MLIP operators on one NVIDIA B200. The symmetric contraction (SC) has 10,000 atoms, ten species, and 192 channels; the edge product has 50,000 edges, 10,000 atoms, and 192 channels, and gathers sender features and sums edge contributions onto receivers. Each has four programs: Forward, Feature VJP over the features, Full VJP over all primal inputs, and Second VJP of the Full VJP. Index arrays stay fixed, and inputs are deterministic `float32`. Table 1 reports the median aggregate CUPTI (NVIDIA Corporation, n.d.-c) kernel time over thirty hot calls and the normalized maximum error, in parts per million, against an independent `float64` oracle on the same GPU (Appendix B.5); Appendix B gives inputs, software, provenance, and protocol, and Appendix C the complete results. Loom runs every case with 300 s each for source search, predicted shortlist realization, and runtime tuning (Section 3). Each baseline library contributes its fastest validated implementation from a fixed list. The ablations ran on B200 GPUs, and the changing graphs, the other compilers, the model layer, and the finite-element operator on one RTX 5080, all with a later Loom build (Appendix D.1).

Workload	Program	Loom		cuEq		OEQ		FlashTP	
		ms	ppm	ms	ppm	ms	ppm	ms	ppm
Symmetric contraction	Forward	**0.121**	**0.31**	0.216	1.02	16.219	**0.31**	—	—
	Feature VJP	**0.310**	**0.29**	0.517	1.46	42.016	0.30	—	—
	Full VJP	**0.450**	**0.28**	0.831	1.74	47.302	1.14	—	—
	Second VJP	**2.032**	0.35	3.354	2.34	130.004	**0.32**	—	—
Edge product	Forward	**0.207**	**0.21**	0.405	0.46	1.052	**0.21**	1.802	0.27
	Feature VJP	**0.171**	0.20	0.434	**0.16**	1.524	0.18	3.472	0.17
	Full VJP	0.614	0.18	**0.569**	0.20	1.539	0.24	3.472	**0.17**
	Second VJP	**1.408**	**0.24**	1.491	0.41	4.866	**0.24**	5.827	0.26

Table 1: Hot execution on one B200: median aggregate CUPTI kernel time over thirty calls in ms and normalized maximum oracle error in ppm. Bold marks the best time and the best error of each row. All rows come from one campaign on one GPU. A dash is unsupported.

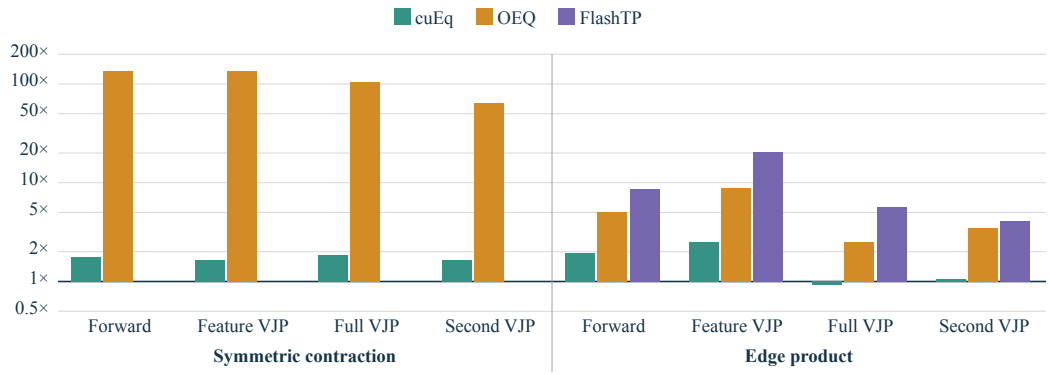


Figure 1: Speedup of Loom over each baseline on one B200: ratio of the median hot kernel times in Table 1, on a logarithmic axis. A bar above 1× means Loom is faster. FlashTP has no symmetric-contraction path.

Results. Against cuEq, Loom's speedup spans 0.93–2.53× and exceeds one in 7 of 8 cases. Loom is 2.5–135.6× faster than OEQ and 4.1–20.3× faster than FlashTP wherever they apply. Loom's largest error is 0.35 ppm; cuEq, OEQ, and FlashTP reach 2.34, 1.14, and 0.27 ppm. On one A100 under the same protocol, Loom is 1.16–3.11× faster than cuEq in 8 of 8 cases, 2.1–182.9× faster than OEQ, and 3.2–16.0× faster than FlashTP (Appendix C.4). On the B200, Loom loses only the edge-product Full VJP, where its B200 search chose cuEq's own edge-parallel design with atomic adds (Appendix D.1).

Compilation cost. In a fresh process with empty caches, Loom's first call takes 37–535 s, against 2–96 s for cuEq, 6–36 s for OEQ, and 6–7 s for FlashTP (Appendix C.1). The baselines choose their implementation beforehand: timing their variants took 52–1062 s per library and case. Figure 2 shows that 6 of 8 Loom cases reach their final candidate within 30 s of tuning. Tripling every budget to 900 s improves Loom's hot kernel time by at most 1.2% and lengthens its first call by up to 614 s (Appendix C.3).

Where the speed comes from. Table 2 removes one mechanism at a time on the B200. Atomic adds instead of fiber walks make the edge product 3.7× and 2.5× slower. Limiting polynomial rewriting slows the memory-bound edge product by at most 1.1× but makes the compute-bound SC Forward 8.7× slower, with a 14× larger error. Rounded constants change no hot time here but enlarge the search (Appendix D.2), and skipping runtime tuning costs up to 2.2×.

Changing graphs. Molecular dynamics rebuilds neighbour lists. New index arrays of the same shape reuse the recipe, and building the `DomainGraph` with its fibers takes 0.073 ms inside `jit`. With a fixed edge capacity padded by −1, as in JAX MD (Schoenholz and Cubuk, 2020), 60, 80, and 100% valid edges run as fast as programs compiled at those exact sizes (Appendix D.1).

Arm	Edge Forward	Edge Feature VJP	SC Forward
Loom, ms	0.205	0.174	0.133
Atomic adds instead of fiber walks	3.65×	2.53×	—
Polynomial rewriting limited to one step	1.06×	1.10×	8.67×
Rounded instead of declared values	1.00×	1.01×	—
No runtime tuning	2.17×	1.87×	1.11×

Table 2: One mechanism removed at a time on B200 GPUs: Loom's median hot kernel time, then each arm's time relative to it. Edge-product arms use 300 s budgets, SC arms and the limited-rewriting arms 120 s. A dash marks an arm that does not apply: SC has no reduction along a map, and its rounded-constant ablation is in Appendix D.2.

Beyond MLIPs and general-purpose compilers. The same interface compiles a matrix-free P1 finite-element operator on 384,000 tetrahedra, with 32 right-hand sides as lanes, into one kernel: 0.180 ms, against 0.211 ms for GPU reassembly then cuSPARSE and 0.592 ms under `jax.jit`, though with a fixed matrix cuSPARSE wins at 0.088 ms. The same Forward programs run 6–100× slower under plain `jax.jit` or `torch.compile` (Ansel et al., 2024) with Triton, and Loom's kernels take about a quarter of a MACE-shaped layer's forward pass (Appendices D.1 and D.3). Dr.Jit, Taichi, and Pallas, which add into receivers atomically, run the edge-product Forward about five times slower than Loom (Appendix D.4).

Limits. The evidence covers two polynomial workloads at one size, one deterministic input distribution, and one search per case. The benchmark index maps are arithmetic sequences, Loom's best case: on a random and a receiver-sorted radius graph of the same shape on the B200, Loom stays 1.03–1.68× faster than cuEq on Forward and Feature VJP but reaches only 0.75–0.91× of its speed on the two higher derivatives (Appendix C.5). Aggregate kernel time is not end-to-end latency, and thirty samples give a median, not a confidence interval.

5 Conclusion

Scientific computation mixes dense matrix operations, which GPUs and vendor libraries already serve well, with small formulas evaluated over large, irregular, indexed structures, which they do not. Loom compiles the second kind. Separating the formula from the placement of its data lets one compiler fuse indexed reads, arithmetic, and reductions: fiber walks replace atomic accumulation, a contiguous lane axis gives regular memory access, scalarization and exact polynomial rewriting expose cancellations, and register-aware partitioning with micro-benchmarks selects a recipe for the shapes at hand. Loom trades a first, lengthy call, paid once per program and GPU, for hot kernels that match or exceed library performance in most benchmark cases, though on realistic graphs it trails on the two higher edge-product derivatives. The requirements for Loom to be advantageous are small fixed internal shapes and a wide axis of independent lanes. These are not specific to equivariant operators and can benefit other domains, for example a matrix-free finite-element operator compiles through the same interface. As Loom is written by AI agents under author direction, similar to recent acceleration efforts by others (Anthropic, 2026), its development suggests that building such specialized compilers per domain is becoming practical and can aid domain-specific libraries in acceleration efforts.

Reproducibility statement

Appendix A states the semantic contract of a Loom call. Appendix B specifies the benchmark inputs, index maps, derivative programs, software versions, and cold and hot measurement protocol, reproduces the complete forward programs, and defines the error metric and its `float64` oracle (Appendix B.5). Every number in the evaluation is computed by the manuscript source from the campaign summary files. The `loom` (the compiler), `algebraic_real` (exact real algebraic arithmetic), and `e3` (exact representations and equivariant tensor bases) packages and the measurement code are intended for release under the Apache License 2.0 (Apache Software Foundation, n.d.).

Disclosure of AI use

Under the authors' guidance and direction, large language models carried out the research and writing, including literature search and synthesis, development of the compiler's concepts, formulation of the semantic contract and its mathematical statements, experimental design and execution, generation of synthetic benchmark inputs, implementation, testing, benchmarking, and interpretation of results. They generated all code developed for this work and the entire manuscript, including its figures and tables.

The authors guided the AI systems and reviewed the manuscript text, portions of the code, test results, and benchmark results. This review was not a complete manual audit of the code or an independent verification of every mathematical claim. The authors take full responsibility for the final manuscript, code, results, and claims.

References

J. Ansel et al. (2024). PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 2, pp. 929–947. doi:10.1145/3620665.3640366

Anthropic (2026). How Claude Is Uplifting Biomolecular Modeling. www.anthropic.com/research/claude-uplifts-biomolecular-modeling

Apache Software Foundation (n.d.). Apache License, Version 2.0. www.apache.org/licenses/LICENSE-2.0

I. Batatia, D. P. Kovács, G. N. C. Simm, C. Ortner, and G. Csányi (2022). MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields. In *Advances in Neural Information Processing Systems*, pp. 11423–11436. proceedings.neurips.cc/paper_files/paper/2022/hash/4a36c3c51af11ed9f34615b81edb5bbc-Abstract-Conference.html

S. Batzner et al. (2022). E(3)-Equivariant Graph Neural Networks for Data-Efficient and Accurate Interatomic Potentials. *Nature Communications*, vol. 13, no. 1, p. 2453. doi:10.1038/s41467-022-29939-5

J. Behler and M. Parrinello (2007). Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Physical Review Letters*, vol. 98, no. 14, p. 146401. doi:10.1103/PhysRevLett.98.146401

V. Bharadwaj, A. Glover, A. Buluç, and J. Demmel (2025). An Efficient Sparse Kernel Generator for O(3)-Equivariant Deep Networks. In *Proceedings of the 2025 SIAM Conference on Applied and Computational Discrete Algorithms*, pp. 32–46. doi:10.1137/1.9781611978759.3

J. Bradbury et al. (2018). JAX: Composable Transformations of Python+NumPy Programs. github.com/jax-ml/jax

B. Charlier, J. Feydy, J. A. Glaunès, F.-D. Collin, and G. Durif (2021). Kernel Operations on the GPU, with Autodiff, without Memory Overflows. *Journal of Machine Learning Research*, vol. 22, no. 74, pp. 1–6. jmlr.org/papers/v22/20-275.html

T. Chen et al. (2018). TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pp. 578–594. www.usenix.org/conference/osdi18/presentation/chen

V. Chorošajev and C. Bény (2026). Sobek: Streaming Equivariant Tensor Product Convolutions. arXiv:2607.18074. arxiv.org/abs/2607.18074

V. L. Deringer, M. A. Caro, and G. Csányi (2019). Machine Learning Interatomic Potentials as Emerging Tools for Materials Science. *Advanced Materials*, vol. 31, no. 46, p. 1902765. doi:10.1002/adma.201902765

M. Fey and J. E. Lenssen (2019). Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*. arXiv:1903.02428

M. Geiger and T. Smidt (2022). e3nn: Euclidean Neural Networks. doi:10.48550/arXiv.2207.09453

Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand (2019). Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Transactions on Graphics*, vol. 38, no. 6, article 201. doi:10.1145/3355089.3356506

Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang (2020). FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13. doi:10.1109/SC41405.2020.00075

JAX Authors (n.d.). Pallas: A JAX Kernel Language. docs.jax.dev/en/latest/pallas/index.html

W. Jakob, S. Speierer, N. Roussel, and D. Vicini (2022). Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering. *ACM Transactions on Graphics*, vol. 41, no. 4. doi:10.1145/3528223.3530099

F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe (2017). The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages*, vol. 1, OOPSLA, article 77. doi:10.1145/3133901

S. Y. Lee et al. (2025). FlashTP: Fused, Sparsity-Aware Tensor Product for Machine Learning Interatomic Potentials. In *Proceedings of the 42nd International Conference on Machine Learning*, PMLR 267, pp. 33143–33156. proceedings.mlr.press/v267/lee25l.html

NVIDIA Corporation (n.d.-a). cuEquivariance. docs.nvidia.com/cuda/cuequivariance/

NVIDIA Corporation (n.d.-b). Parallel Thread Execution ISA. docs.nvidia.com/cuda/parallel-thread-execution/

NVIDIA Corporation (n.d.-c). CUPTI: CUDA Profiling Tools Interface. docs.nvidia.com/cupti/

OpenXLA Authors (n.d.-a). StableHLO. openxla.org/stablehlo

OpenXLA Authors (n.d.-b). XLA. openxla.org/xla

P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock (2015). Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–11. doi:10.1145/2737924.2737959

J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe (2013). Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of PLDI*, pp. 519–530. doi:10.1145/2491956.2462176

S. S. Schoenholz and E. D. Cubuk (2020). JAX, M.D.: A Framework for Differentiable Physics. *Advances in Neural Information Processing Systems*, vol. 33. arXiv:1912.04232

R. Tate, M. Stepp, Z. Tatlock, and S. Lerner (2009). Equality Saturation: A New Approach to Optimization. In *Proceedings of POPL*, pp. 264–276. doi:10.1145/1480881.1480915

P. Tillet, H. T. Kung, and D. 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, pp. 10–19. doi:10.1145/3315508.3329973

M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha (2021). egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages*, vol. 5, POPL, pp. 1–29. doi:10.1145/3434304

Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu (2021). Seastar: Vertex-Centric Programming for Graph Neural Networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 359–375. doi:10.1145/3447786.3456247

Z. Xie, M. Wang, Z. Ye, Z. Zhang, and R. Fan (2022). Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. In *Proceedings of Machine Learning and Systems*, vol. 4, pp. 515–528. proceedings.mlsys.org/paper_files/paper/2022/hash/a1126573153ad7e9f44ba80e99316482-Abstract.html

Y. Yang, P. M. Phothilimthana, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar (2021). Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, pp. 255–268. proceedings.mlsys.org/paper_files/paper/2021/hash/cc427d934a7f6c0663e5923f49eba531-Abstract.html

Z. Ye, R. Lai, J. Shao, T. Chen, and L. 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*, vol. 3, pp. 660–678. doi:10.1145/3582016.3582047

Y. Zhang et al. (2023). Better Together: Unifying Datalog and Equality Saturation. *Proceedings of the ACM on Programming Languages*, vol. 7, PLDI, pp. 468–492. doi:10.1145/3591239

Y. Zhou et al. (2023). uGrapher: High-Performance Graph Operator Computation via Unified Abstraction for Graph Neural Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 2, pp. 878–891. doi:10.1145/3575693.3575723

APPENDICES

A SEMANTIC CONTRACT

A call declares b domain axes. Every operand has the same b leading domain axes, then its fixed internal axes, then one lane axis of extent 1 or u. Write D_o^a for the domain declared by operand o on axis a, and $\star$ when the axis is unindexed. A declared index map $\rho : A \to B$ maps each element of A to an index of B or to -1. The maps form a DAG with at most one directed path between two domains. Say that c reaches d when that path exists or $c = d$.

A.1 ITERATION DOMAINS

For each axis, define

$$U_a = \{D_o^a : o \in \text{operands} \wedge D_o^a \neq \star\}.$$

Declared domains U_a	Iteration domain I_a	Extent
Empty	Unindexed singleton $\star$	1
One domain d	d	$\lvert d\rvert$
Several; a member of U_a reaches all	Unique most-specific such member	$\lvert I_a\rvert$
Several; no member reaches all	Unique most-specific domain in the graph that reaches all	$\lvert I_a\rvert$
No candidate or no unique choice	Rejected	—

Table 3: The independent iteration-domain choice on one domain axis. "Most-specific" means that no other candidate in the selected pool reaches it.

For example, sibling domains `sender` and `receiver` select their common source `edge`; `edge` and `sender` select `edge`. Choosing a declared domain before an undeclared common source keeps the iteration no finer than needed. Ambiguity and absence are errors rather than traversal heuristics.

A.2 ADDRESSES, PARTIAL MAPS, AND REDUCTIONS

Let $q = (q_1, ..., q_b)$ range over $I_1 \times ... \times I_b$. The index of operand o on axis a is:

Declaration D_o^a	Index	If a composed map gives -1
$\star$	0	Cannot occur
I_a	q_a	Cannot occur
Reached from I_a	Composition along the unique path	Input reads zero; output contribution is dropped
Not reached from I_a	Rejected	—

Table 4: Physical addressing on one domain axis. An unindexed axis has extent one and broadcasts on input.

The per-axis indices together form $\pi_i(q)$ for input i and $\sigma_j(q)$ for output j. A partial-map padding slot therefore needs no guard in user code. An unmapped input behaves as zero; an unmapped write is discarded.

Write X_i for the physical array of input i, and set $\lambda_i(\ell) = 0$ when input i has a single lane and $\lambda_i(\ell) = \ell$ otherwise. With $x_i(q, \ell) = X_i(\pi_i(q), \cdot, \lambda_i(\ell))$, every output obeys

$$Y_j(p, s, \ell') = \sum_{\ell \in \Lambda_j(\ell')} \sum_{q \in \sigma_j^{-1}(p)} \left[f_j\big((x_i(q, \ell))_i\big) \right]_s .$$

The fiber includes every iterated position mapped to p. Thus one additive rule covers reduction over a map fiber, over an iterated axis that the output leaves unindexed, and, when requested, over all u lanes. Nothing else is reduced. Retained lanes use $\Lambda_j(\ell') = \{\ell'\}$; a lane-reduced output uses $\Lambda_j(\ell') = \{0, ..., u - 1\}$.

A.3 Defaults and rejections

Situation	Meaning or result
`domain_graph` omitted	An empty graph
A domain-spec leaf is `None`	`loom.domains[...]`, filling the already-resolved `num_axes`
Every spec omitted	Zero domain axes; `loom.call(fn, inputs)` is a pure lane map
`:` in a spec	One unindexed domain axis
`...` in a spec	All remaining axes are unindexed; at most one ellipsis per spec
Every spec has `...`, `num_axes` absent	Rejected because the domain-axis count cannot be inferred
Exact axis counts conflict	Rejected
Non-singleton lane extents conflict	Rejected
Every lane extent is one, u absent	Rejected because u cannot be inferred
Unknown domain or wrong physical extent	Rejected
Boolean output needs a nontrivial domain or lane reduction	Rejected because additive Boolean reduction is undefined

Table 5: Essential defaults and static errors. Omitted pytree leaves are expanded before the ranks are checked.

`output_dtypes` defaults to the dtypes returned by the lane function and may narrow them after reduction. `reduce_lanes` broadcasts over the output pytree. The `is_injective`, `is_sorted`, and `may_have_unmapped` properties of an index map are trusted promises and are not checked at run time.

B Benchmarks and protocol

We evaluate eight cases on NVIDIA B200, and on A100 in Appendix C.4: two sparse polynomial operators from equivariant MLIPs, each with four programs. The tables compare Loom, cuEquivariance (cuEq), OpenEquivariance (OEQ), and FlashTP using one hot-execution metric. These are fixed-size operator tests, not end-to-end model benchmarks.

B.1 Eight benchmark cases

The **symmetric contraction** (SC) has 10,000 atoms, ten species, and 192 channels. It combines polynomial degrees one through three. The **edge product** has 50,000 edges, 10,000 atoms, and 192 channels; it gathers sender features and reduces edge contributions onto receivers. Both use deterministic `float32` inputs and sparse equivariant coefficients.

Forward evaluates the operator. **Feature VJP** differentiates only the feature input. **Full VJP** differentiates every primal input, including weights and, for the edge product, harmonics. **Second VJP** differentiates the full-VJP program with respect to both the original primal inputs and the cotangent supplied to the first VJP. Index arrays stay fixed.

Let $F(p)$ denote the forward program with indexing arrays fixed: $p = (w, x)$ for SC and $p = (w, x, a)$ for the edge product, where a contains harmonics. Feature VJP is $D_x F(p)^T g$, and full VJP is $B(p, g) = D_p F(p)^T g$. Second VJP is $D_{(p,g)} B(p, g)^T h$. Its two blocks are the Hessian–vector product $\nabla_p^2(g^T F(p))h$ and tangent $D_p F(p)h$. The cotangents g and h are dynamic all-ones arguments, not constants folded into the executable.

Loom compiles these programs with their gathers and reductions. The baselines use their native interfaces: FlashTP and OEQ fused edge convolutions, and, for SC, OEQ 0.7.0′s undocumented `symmetric_contraction` module, which implements MACE's dense algorithm, through a basis adapter whose dynamic weight transform remains timed. FlashTP has no SC path.

The figure and tables use Forward = `fwd`, Feature VJP = `x-bwd`, Full VJP = `wx-bwd`, and Second VJP = `wx-bwd2`. Edge product denotes `channelwise`; Symmetric contraction denotes `symmetric_contraction`.

B.2 Inputs and descriptors

Before conversion to `float32`, entry (r, c) of input operand k is

$$0.35\sin(0.0007r + 0.019c + 0.31k)\cos(0.011c - 0.0013r).$$

For edge e and $A = 10,000$ atoms, `sender` is $(37e + 11) \bmod A$ and `receiver` is $(53e + 17) \bmod A$, incremented modulo A when it equals the sender. Receiver r uses species $r \bmod 10$. Every map is total.

The channelwise tensor product and symmetric contraction use flattened scalar coefficient modes. Their segment-lane layouts are `(13,192) (9,192) (16,1) -> (43,192)` and `(49,192) (16,192) -> (9,192)`, with 125 and 5,520 paths. Their flat inputs are `(50000,2496) (10000,1728) (50000,16)` and `(10,9408) (10000,3072)`. The coefficient-recovery gaps are 4.4×10^{-16} and 3.3×10^{-16}.

SC input features have degrees zero through three and outputs zero through two; polynomial degrees are one, two, and three. Edge features have degrees zero through two and harmonics zero through three. Each descriptor coefficient is recovered as the nearest signed square root of a rational whose denominator is at most 10^7. The recovery gaps above are below one `float32` unit in the last place.

B.3 Forward programs

The complete standalone Loom program for both forward passes, the edge product and the symmetric contraction, is reproduced below. Its random inputs illustrate the physical shapes; the timed campaign instead uses the deterministic inputs specified above.

```
import e3
import e3.jax
import jax
import loom

NUM_ATOMS = 10_000
NUM_EDGES = 50_000
NUM_SPECIES = 10
NUM_CHANNELS = 192
IRREPS_L2 = (e3.O3("0e"), e3.O3("1o"), e3.O3("2e"))
IRREPS_L3 = (*IRREPS_L2, e3.O3("3o"))

keys = iter(jax.random.split(jax.random.key(0), 32))
def randn(*shape):
    return jax.random.normal(next(keys), shape)

sender, receiver = jax.random.randint(next(keys), (2, NUM_EDGES), 0, NUM_ATOMS)
species = jax.random.randint(next(keys), (NUM_ATOMS,), 0, NUM_SPECIES)

graph = loom.DomainGraph(
    {"edge": NUM_EDGES, "sender": NUM_ATOMS, "receiver": NUM_ATOMS, "species":
NUM_SPECIES},
    {
        ("edge", "sender"): loom.IndexMap(sender, may_have_unmapped=False),
        ("edge", "receiver"): loom.IndexMap(receiver, may_have_unmapped=False),
        ("receiver", "species"): loom.IndexMap(species, may_have_unmapped=False),
    },
)


# Edge product: weighted x[sender] ⊗ Y per edge, summed onto receivers.
def edge_product(values):
    weights, features, harmonics = values
    paths = e3.jax.tensor_product(features, harmonics, keep=IRREPS_L2)
    return {irrep: (coupled * weights[irrep][None, :]).T for irrep, coupled in
paths.items()}


edge_inputs = (
    {
        irrep: randn(NUM_EDGES, paths, NUM_CHANNELS)
        for irrep, paths in zip(IRREPS_L2, (3, 5, 5), strict=True)
    },
    {irrep: randn(NUM_ATOMS, irrep.dim, NUM_CHANNELS) for irrep in IRREPS_L2},
    {irrep: randn(NUM_EDGES, irrep.dim, 1) for irrep in IRREPS_L3},
)

edge_output = loom.call(
```

```
    edge_product,
    edge_inputs,
    u=NUM_CHANNELS,
    domain_graph=graph,
    input_domains=(loom.domains["edge"], loom.domains["sender"],
loom.domains["edge"]),
    output_domains=loom.domains["receiver"],
)


# Symmetric contraction: degree 1-3 symmetric powers of x, weighted per species.
def symmetric_contraction(values):
    weights_by_degree, features = values
    return e3.jax.symmetric_contraction(features, weights_by_degree)


symmetric_inputs = (
    {
        degree: {
            irrep: randn(NUM_SPECIES, paths, NUM_CHANNELS)
            for irrep, paths in zip(IRREPS_L2, counts, strict=True)
        }
        for degree, counts in {1: (1, 1, 1), 2: (4, 3, 5), 3: (8, 12, 14)}.items()
    },
    {irrep: randn(NUM_ATOMS, irrep.dim, NUM_CHANNELS) for irrep in IRREPS_L3},
)

symmetric_output = loom.call(
    symmetric_contraction,
    symmetric_inputs,
    u=NUM_CHANNELS,
    domain_graph=graph,
    input_domains=(loom.domains["species"], loom.domains["receiver"]),
    output_domains=loom.domains["receiver"],
)
```

B.4 Software and measurement protocol

All measurements use one NVIDIA B200 with driver 615.71.09, Python 3.12.13, JAX and jaxlib 0.11.1, PyTorch 2.11.0, cuEquivariance 0.11.1, OpenEquivariance 0.7.0, FlashTP 0.1.0, and the `loom`, `e3`, and `algebraic-real` 0.1.0 packages. Inputs are `float32`, TF32 is disabled, and the canonical inputs and index arrays are on the GPU before any timing starts.

A cold measurement starts a fresh process with empty private compilation caches, including Loom's compile cache, and ends at the first synchronized canonical output. For Loom it contains source search, shortlisting, assembly, and tuning. The hot measurement follows in the same process: three warm-up calls, then thirty samples of aggregate CUPTI kernel time, whose median the tables report. Loom sets `search_budget_ms`, `shortlist_budget_ms`, and `scheduling_time_limit_ms` to 300000 and uses eight compiler threads. Each case runs once with `LOOM_PROFILE=1`, which records the event stream behind Figure 2. An unprofiled repeat of symmetric-contraction Feature VJP differs by 1.0% in cold time and 0.08% in hot kernel time.

All execution tables report the **median aggregate GPU kernel duration over thirty hot executions**, measured with CUPTI. Compilation, tuning, host dispatch, and synchronization are excluded; value-dependent layout transforms are included. Kernel durations are summed without subtracting overlap between streams, so this measures GPU work rather than end-to-end latency. Every B200 row comes from one campaign on one B200 and is judged against the `float64` GPU oracle of Appendix B.5.

Each baseline library first timed a fixed list of implementations per case, covering its JAX and PyTorch front ends, graph compilation, and library-specific kernel options, and kept the fastest validated one by the median of five CUPTI samples. The twenty winners were then rerun cold and hot on the measuring GPU against the same oracle as Loom.

Two earlier runs on another B200 ended in GPU faults; their partial results enter no table.

B.5 Oracle error

For output leaf j, the reported normalized maximum error is

$$e_j = \frac{\max_i |A_{j,i} - T_{j,i}|}{\max\left(\max_i |T_{j,i}|, 10^{-300}\right)},$$

and a record reports $\max_j e_j$. Here T is produced by an independent term-list evaluator of the descriptor, written in pure JAX without Loom or cuEq code and run in `float64` on the measuring GPU, in its own process before the campaign. Forward and both derivative orders use the same evaluator after symbolic monomial differentiation. On six full-size problems, its truth differs from a CPU evaluator accumulating in x87 extended precision by at most 1.6×10^{-14} relative to $\max_i |T_{j,i}|$; symmetric-contraction Full and Second VJP are compared only at reduced size. All tables express e in parts per million. This is not the maximum elementwise relative error.

As a secondary sanity check, each coordinate is also compared with the a-priori bound $\gamma_{n-1+d+2} \sum_i |t_i|$ for n terms of degree d, where $\gamma_k = k\frac{u}{1-ku}$ and $u = 2^{-24}$, plus coefficient-recovery slack. Every completed arm has finite outputs and no coordinate above its bound. The bound is a sanity check, not an accuracy target; error magnitude is the accuracy measurement.

The executable listings were run on the B200 with the Loom build used for the campaign. That build predates the split into the `loom`, `algebraic_real`, and `e3` packages, so the listings' imports were adapted to its layout; all assertions pass. The opening example as printed, with normal samples, was rerun on a B200 with the later build of Appendix D.1; its relative error is 1.15×10^{-7}.

C Complete results

This appendix extends Table 1 with the cold first calls, the search behind them, longer budgets, a second GPU, and other graphs.

C.1 Cold first calls

Workload	Program	Loom	cuEq	OEQ	FlashTP
Symmetric contraction	Forward	58.4	**2.8**	23.4	—
	Feature VJP	37.2	**10.0**	26.3	—
	Full VJP	127.7	30.6	**28.6**	—
	Second VJP	228.0	96.0	**36.0**	—
Edge product	Forward	164.8	**1.8**	6.8	6.2
	Feature VJP	136.7	**1.8**	6.0	6.7
	Full VJP	535.0	**2.3**	6.0	6.8
	Second VJP	452.7	12.6	9.6	**6.9**

Table 6: Cold first call on one B200, in seconds: a fresh process with empty caches, through the first synchronized output. Loom's call includes its search and tuning under 300 s budgets; each baseline runs its fixed winner, whose selection is not included. Bold marks the fastest library of each row.

C.2 Search and tuning

Figure 2 shows when each profiled cold call finds faster complete candidates. Search emits 2175023 sources across the eight cases, and 41 complete root timings survive pruning and infeasibility. The longest call ends 530 s after its first profile event. One search per case under fixed budgets does not establish a global optimum.

Figure 2 reduces the event stream of each profiled cold call in Appendix C.1. Search, recipe construction, setup, and tuning share one process and one clock, whose origin is the first profile event; lowering before it lies outside the plot. The boundaries are the search-stopped and recipe-built marks and the first and last tuning events.

Each staircase uses complete depth-zero candidate timings at their visit-completion times. Only strict improvements create steps, and the last value is held to the final tuning event. A lone shortlisted candidate is kept without measurement and has no staircase. Latencies are the tuner's own measurements of complete candidates, not CUPTI aggregate kernel samples, and never enter the tables. Source ticks mark every candidate emission; fixed-width ticks are unioned in logarithmic display coordinates without bins, and events before one second lie outside the view.

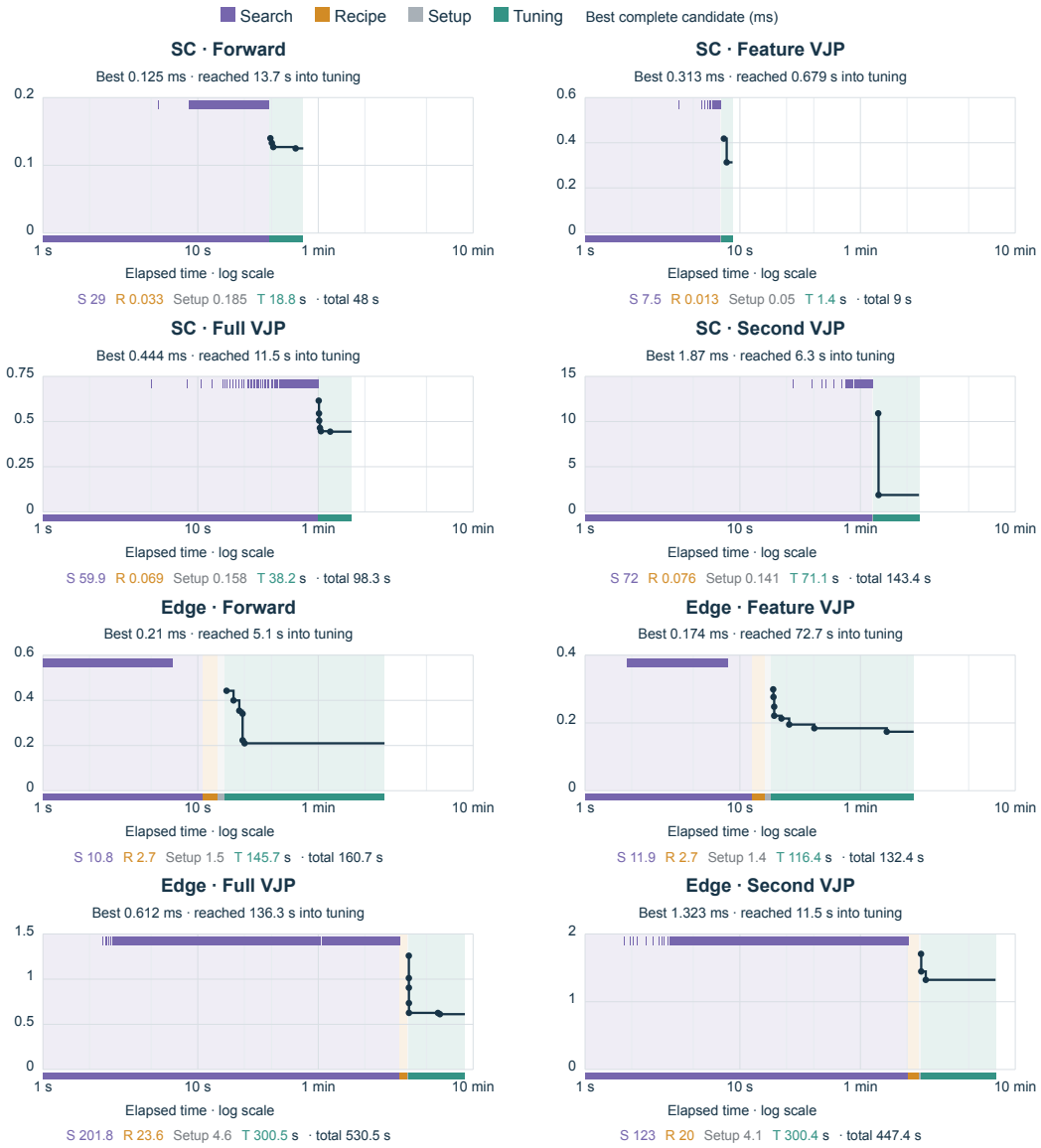


Figure 2: Best-so-far complete-candidate latency during the eight profiled B200 cold calls behind Table 1, with 300 s budgets. All panels use a logarithmic 1 s–10 min axis from the first profile event; latency axes are linear with independent ranges. Colors mark search, recipe construction, setup, and tuning. Purple ticks mark source emissions; dense ticks overlap. Dots mark strict improvements in the tuner's measurements, not CUPTI hot medians.

C.3 Longer search budgets

Two further runs of the eight Loom cases set all three budgets to 900,000 ms, on the same GPU, against the same oracle, and with the same protocol. Table 7 compares them with the 300 s run of Table 1. Symmetric-contraction searches end before 300 s, so their cold calls are unchanged. The longer budgets change hot kernel time by −1.2% to 0.5% and cold time by −1 to 614 s. The two 900 s runs also bound run-to-run variation of one search.

Workload	Program	300 s		900 s, run 1		900 s, run 2	
		ms	cold s	ms	cold s	ms	cold s
Symmetric contraction	Forward	0.1210	**58**	**0.1196**	**58**	0.1201	59
	Feature VJP	**0.3098**	**37**	**0.3098**	**37**	0.3099	**37**
	Full VJP	**0.4503**	128	0.4523	128	0.4504	**127**
	Second VJP	**2.0316**	**228**	2.0349	**228**	2.0381	234
Edge product	Forward	0.2065	**165**	0.2065	198	**0.2043**	194
	Feature VJP	**0.1712**	**137**	**0.1712**	177	0.1713	178
	Full VJP	0.6140	**535**	0.6141	1148	**0.6138**	1148
	Second VJP	1.4085	**453**	1.3969	1067	**1.3921**	1066

Table 7: Loom under 300 s and 900 s budgets on one B200: median hot kernel time in ms and cold first call in s. Bold marks the lowest value of each row and column kind.

C.4 A100 CAMPAIGN

We repeated the eight cases on one NVIDIA A100-SXM4-80GB with the same Loom, `e3`, and `algebraic_real` builds, the same measured sources, and the same 300 s budgets. The Loom searches and the baseline implementation grid ran as two separate jobs on the same GPU, each judged against `float64` GPU oracles computed in that job; the Loom job's GPU oracles agree with the CPU extended-precision oracles to within 1.6×10^{-14} of the largest output magnitude, as on the B200. Because the grid itself was judged against the GPU oracle, each baseline reports its winner's confirmation measurement rather than a separate rerun. Loom's first call takes 58–639 s on the A100; its largest error is 0.35 ppm, against 2.34 ppm for cuEq.

Workload	Program	Loom		cuEq		OEQ		FlashTP	
		ms	ppm	ms	ppm	ms	ppm	ms	ppm
Symmetric contraction	Forward	**0.421**	**0.31**	0.822	1.02	36.984	**0.31**	—	—
	Feature VJP	**0.987**	**0.29**	1.583	1.46	140.884	0.30	—	—
	Full VJP	**1.214**	**0.27**	2.392	1.95	222.033	1.15	—	—
	Second VJP	**5.076**	0.35	9.768	2.34	493.729	**0.32**	—	—
Edge product	Forward	**0.568**	**0.21**	1.204	0.46	2.767	0.24	4.605	0.27
	Feature VJP	**0.593**	0.20	1.847	**0.16**	3.714	0.19	9.485	0.17
	Full VJP	**1.766**	0.18	2.331	0.20	3.667	0.24	8.941	**0.17**
	Second VJP	**4.451**	**0.24**	5.172	0.40	10.668	0.26	14.334	0.26

Table 8: Hot execution on one A100-SXM4-80GB: median aggregate CUPTI kernel time over thirty calls in ms and normalized maximum oracle error in ppm. Bold marks the best time and the best error of each row. A dash is unsupported.

C.5 OTHER EDGE-PRODUCT GRAPHS

The benchmark maps send edge e to sender $(37e + 11) \bmod N$ and receiver $(53e + 17) \bmod N$. Both depend only on $e \bmod N$, so the five edges of each sender share one receiver. We reran the edge product on another B200 of the same cluster, with driver 610.57.04, the same builds, budgets, and protocol, and two other graphs with $N = 10{,}000$ atoms and $E = 50{,}000$ edges. The **random** graph draws uniform senders and sets each receiver to the sender plus a uniform nonzero offset, unsorted. The **radius** graph places atoms uniformly in a periodic unit cube in Morton order and keeps the directed pairs closer than the distance quantile that yields 50,000 edges, sorted by receiver as MD neighbour lists are. Only the index arrays of the measured inputs and the oracles change; every case agrees with its `float64` GPU oracle. cuEq, OEQ, and FlashTP rerun their B200 winners without reselection. A fourth run declares the radius graph's receiver map sorted.

Program	Benchmark	Random	Radius	Radius, sorted
Forward	0.207 (1.96×)	0.254 (1.68×)	0.244 (1.20×)	0.240 (1.22×)
Feature VJP	0.171 (2.53×)	0.284 (1.33×)	0.221 (1.03×)	0.221 (1.03×)
Full VJP	0.614 (0.93×)	0.584 (0.91×)	0.468 (0.82×)	0.468 (0.82×)
Second VJP	1.408 (1.06×)	1.753 (0.81×)	1.409 (0.75×)	1.408 (0.75×)

Table 9: Edge product on one B200 over four index maps: Loom's median hot kernel time in ms and, in parentheses, its speedup over cuEq. All runs use 300 s budgets. The Benchmark column repeats Table 1 (driver 615.71.09); the other columns come from a second B200 (driver 610.57.04).

cuEq gains most from locality; on the benchmark and random graphs its Forward also regroups its indices on every call. Loom keeps an edge-parallel recipe with atomic adds for the Full VJP on every graph, and declaring the receivers sorted changes no recipe. OEQ and FlashTP remain slower than both.

C.6 Limits

The study covers one size and one deterministic full-size input distribution per workload. Thirty samples describe a median, not a confidence interval. Each Loom case contains one search under fixed budgets; the scheduling clock stops opening candidates but lets launched work finish. Oracle agreement on these inputs does not prove accuracy for every supported program.

D Ablations and local studies

The ablations ran on NVIDIA B200 GPUs, one arm per GPU and process. The changing graphs, the finite-element operator, the other compilers, and the model layer ran on one NVIDIA RTX 5080 16 GB.

D.1 Mechanisms, changing graphs, and a finite-element operator

These checks ran with a later Loom build whose polynomial rewriting is unchanged, JAX 0.11.2, and cuEq 0.11.1, through the same benchmark harness. Each value is the median CUPTI kernel time of thirty hot calls, one call per sample, with fresh compile caches for every arm. Every arm agrees with its `float64` oracle to 3.1×10^{-7} relative error or better, except the throttled SC arms of Table 2.

Ablation arms. The atomic arm uses a development switch that disables the promotion turning a reduction along a map into a fiber walk, so the recipe must clear the output and scatter with atomic adds. The limited-rewriting arm sets `max_algebraic_work` and `max_algebraic_expression_work` to one. Its SC search then grows past 12 GB of host memory under the default search-memory estimate, so both SC arms cap that estimate at 2 GiB; the unthrottled SC result is the same with and without the cap. Throttling keeps the edge-product kernels within 10% of their times but lengthens them from 372 to 413 statements; on SC it multiplies statements by about seven. Rounded edge-product constants give the same kernel after a 454,380-candidate search instead of 90,876, and a 351 s instead of 109 s first call. `scheduling_time_limit_ms=0` selects a feasible recipe without measurement, with a first call under one second.

Changing graphs. Building the edge product's `DomainGraph`, two maps with their fibers by sorting, counting, and prefix sums, takes 0.073 ms of GPU time inside `jit`, or 0.044 ms when receivers are declared sorted. A compiled program called with new index arrays of the same shape is neither retraced nor recompiled. With a capacity of 50,000 edges on a random graph and -1 padding, 60, 80, and 100% valid edges take 0.985, 1.161, and 1.336 ms, against 0.985, 1.161, and 1.338 ms for programs compiled at those exact sizes. Sorting the index arrays by receiver takes 0.028 ms; Loom does not require it. Checking `is_sorted`, `is_injective`, and total maps on both edge maps takes 0.018 ms of GPU time together and 0.073 ms with the host readback needed to raise an error.

The edge-product Full VJP. On the B200, Loom's recipe clears the input cotangent and runs one edge-parallel kernel with atomic adds (0.599 ms), the design of cuEq's winning kernel (0.571 ms). Its search stopped at the 16 GiB search-memory estimate after 1,022,192 candidates, so whether it reached a walk over senders is unknown. On the RTX 5080, Loom selects that walk: 1.82 ms, 1.45 GB of DRAM traffic, the compulsory amount, and no atomics. Forced to the edge-parallel design, Loom takes 4.37 ms and cuEq 4.33 ms, both with 3.68 GB of traffic and 2.7 million atomic

reductions. On a random graph of the same shape, the walk takes 3.17 ms, cuEq 4.04 ms, and the edge-parallel recipe 4.07 ms.

Finite-element operator. The mesh divides 40^3 jittered cubes into six tetrahedra each: 384,000 elements, 68,921 nodes, and 993,961 nonzeros in K. The Loom lane function computes, for each element, the four products of its local stiffness matrix with the gathered corner values, and scatters them through four element-to-node maps; a JAX addition sums the four outputs. An equivalent formulation over the 6,144,000 element-corner pairs takes 0.600 ms, so the formulation still matters. The cuSPARSE arms use JAX's BCSR product, with K assembled once on the host, or reassembled on the GPU at every call by a segment sum into a fixed sparsity pattern. All arms agree with a SciPy `float64` product to 2×10^{-7} relative error. Loom's first call takes 75 s under 60 s budgets.

D.2 Declared constant values

We compiled the full-size symmetric contraction (10,000 receivers, ten species, $u = 192$, `float32`) on one NVIDIA RTX 5080 16 GB twice, under 300 s search and shortlist budgets: with every coefficient passed through `exact`, and with the same coefficients rounded to `float32`, which Loom then reads as exact rationals, so products such as $\sqrt{2}\sqrt{2}$ no longer cancel. The main effect is on search: with declared values, the Forward and Feature VJP programs exhaust their source space after 968 and 24 candidates, in 23 s of lowering, while the rounded ones generate 984,183 candidates before the search-memory budget stops them, and 36,612, in 86 s and 59 s. Median CUPTI times over 30 samples are 0.224 and 0.417 ms with declared values and 0.226 and 0.462 ms with rounded ones; a repeated Feature VJP pair reproduced both within 0.2%. Every selected recipe launches one kernel. Oracle errors are 0.31 and 0.29 ppm with declared values and 0.26 and 0.24 ppm with rounded ones, all admissible. Declaring the values thus buys a much smaller search and a 10% faster Feature VJP, not a smaller error, in this small experiment on one workload and one local GPU.

D.3 XLA, Triton, and a model layer

We ran both Forward programs at their full sizes, with JAX 0.11.2, cuEq 0.11.1, and PyTorch 2.13.0 with Triton 3.7.1. The XLA arm applies `jax.jit` to the same lane function, with an explicit gather, `vmap` over lanes, and `segment_sum`. The Triton arm applies `torch.compile(fullgraph=True, dynamic=False)` to the same term list; Inductor lowers it to Triton kernels. Each value is the median aggregate CUPTI kernel time of thirty hot calls, taken one call per sample. For SC Forward, Loom takes 0.225 ms, cuEq 0.278 ms, XLA 22.5 ms, and Triton 5.11 ms. For edge Forward, they take 1.08, 1.21, 6.64, and 9.45 ms. XLA emits 19 kernels for each program and needs 3.9 GB and 387 MB of temporary storage. The first call, including Loom's search and tuning, takes 36 s and 159 s with Loom and an empty compile cache, 42 s and 0.5 s with XLA, and 165 s and 6 s with Inductor. Triton outputs agree with a `float64` evaluation of the same terms within 4.7 ppm.

To place these operators in a model step, we composed one synthetic MACE-shaped layer at the same sizes, with random untrained weights: a radial MLP with widths 8, 64, 64, 64, and 2,496; spherical harmonics up to degree three; the edge product; per-irrep channel mixing with a skip connection; the symmetric contraction; and an output linear map. Its forward pass takes 5.22 ms of kernel time with Loom, 5.38 ms with cuEq, and 32.9 ms with the XLA arms. The two Loom kernels take 1.32 ms, a quarter of the Loom step; dense products and layout copies take most of the rest. The three variants agree in output RMS to seven digits. This is one forward layer on a consumer GPU, not a trained model, a force evaluation, or a molecular-dynamics step.

D.4 General-purpose GPU compilers

We wrote both Forward programs in four general-purpose systems and ran them beside Loom: Dr.Jit 1.5.0, Taichi 1.7.4, KeOps 2.3 with PyTorch 2.14.0+cu130, and Pallas with its Triton lowering in JAX 0.11.2. Every system receives the same `float32` inputs, index maps, and term list, grouped by output segment in the order Loom's rounded-constant lane program uses. Each reads every input segment once per lane and multiplies the rounded coefficients in term order; none receives declared constant values, factorizations, or a kernel partition. Loom compiles its usual lane program with declared constant values.

Each system uses its own programming model. Dr.Jit traces gathers, products, and a `scatter_reduce` addition over one flat (row, lane) index and compiles the trace itself. Taichi runs one generated kernel over (row, lane) with `atomic_add` into receivers. Pallas runs a Triton tile kernel of rows × lanes with `atomic_add`; we timed tiles of 8 or 16 rows by 32 or 64 lanes with two or four warps and report

the fastest, 8 × 64 for the edge product and 8 × 32 for SC. For SC we timed only the 32-lane tiles, because `ptxas` exceeded our 12 GB host-memory cap on 64-lane tiles. KeOps variables are indexed by i, by j, or by nothing, so a value indexed by both a row and a lane must be flattened into i, and an indexed read or a sum along a map cannot enter a formula: PyTorch gathers the inputs, KeOps evaluates one formula per (row, lane), and `index_add_` sums rows into receivers.

We measure each system in a fresh process with empty CUDA and KeOps caches. Table 10 reports the median, over thirty hot calls, of the summed CUPTI kernel time of one call, recorded by the PyTorch profiler across all CUDA contexts in the process. It includes every fill, gather, and scatter kernel a system launches and excludes memory copies. The first call includes tracing, compilation, and, for Loom, search and tuning. For Pallas it follows the tile search, which compiled and timed the tiles in 7 s for the edge product and 5 min for SC. Errors are normalized maximum differences, in ppm, from a `float64` evaluation of the same terms, over all receivers for the edge product and over the first 256 receivers for SC.

	Edge Forward				SC Forward			
System	**ms**	**kernels**	**first s**	**ppm**	**ms**	**kernels**	**first s**	**ppm**
Loom	1.084	2	157.1	0.19	0.223	1	35.3	0.41
Dr.Jit	5.313	1	0.1	0.22	1.934	9	0.4	4.54
Taichi	5.199	2	0.2	0.20	1.748	2	10.5	0.85
Pallas	5.506	2	0.6	0.23	24.784	1	49.4	4.54
KeOps	15.959	7	0.5	0.21	352.520	4	1167.7	0.27

Table 10: Forward programs on one RTX 5080: median aggregate CUPTI kernel time over thirty hot calls in ms, kernels per call, first-call wall time in s, and normalized maximum error against `float64` terms in ppm.

On the edge product, Dr.Jit, Taichi, and Pallas each fuse the computation into one kernel that adds into receivers atomically, Taichi and Pallas after one fill of the output, and all three take about five times Loom's time. None of the kernels Loom compiled for these two programs contains an atomic instruction; Loom walks each receiver's fiber instead. We have not isolated this choice from Loom's other optimizations. KeOps spends 6.06 ms in its formula kernel and the rest in PyTorch gathers and `index_add_`. SC has no reduction along a map, so its gap comes from the scalar program and its code generation: Loom's polynomial rewriting, partitioning, and tuned PTX against direct evaluation of 5,520 terms. Dr.Jit splits SC into one kernel per output segment. KeOps turns each of the 5,520 coefficients into a parameter variable; generating and compiling that formula takes most of its 19-minute first call.

We did not run TACO, SparseTIR, TVM, or Halide. TACO and SparseTIR would need the lane program restated as tensor algebra over incidence matrices, and TVM and Halide would need a hand-written or searched schedule for each program; the comparison would then measure our schedules rather than the systems.

E Glossary

Algebraic real number. A real root of a nonzero polynomial with integer coefficients. Rational numbers and square roots of rational numbers are examples.

Atomic add. A GPU instruction that safely adds to one memory location when several workers may update it. Heavy contention serializes the updates.

Automatic differentiation (AD). A program transformation that constructs derivative programs by applying local calculus rules to a value program.

Candidate. A recipe under consideration for a Loom program, fixing its partition and GPU launch settings.

Clebsch–Gordan tensor product. An operation that couples features transforming under rotations using fixed sparse coefficients from representation theory.

Cold observation. One measurement of variant construction, lowering, compilation, and first execution for one case: a workload, a program, and an implementation.

Contiguous axis. An array axis whose adjacent logical entries occupy adjacent memory locations.

Cotangent. A value propagated backward by reverse-mode AD. Paired with an output perturbation, it produces input gradients.

Descriptor. A declarative account of an operator's segments, sparse paths, coefficients, and contractions.

Domain. A named, nonempty finite set used to identify the leading axes of an array.

Domain axis. A leading physical array axis governed by a domain or left as an unindexed singleton.

Domain graph. The finite domains and direct index maps supplied to one Loom call, together with their fibers, aligned target indices, and any unique compositions required by the call.

e-graph. A data structure that stores expressions in equivalence classes while sharing common parts.

Equality saturation. Optimization that adds equivalent expressions to an e-graph before selecting a low-cost representative.

Equivariance. The property that a change of input coordinate frame produces a known corresponding transformation of the output.

Fiber. The preimage of one target element under an index map: all mapped source elements whose target index selects it.

Gather. Reading values through an index map, such as reading the source feature for each edge.

Hot execution. A run after tracing, compilation, assembly, and tuning have completed and caches are populated.

IEEE-754. The standard for binary floating-point values and operations, including rounding, NaNs, infinities, and signed zero.

Index map. A possibly partial function from a source domain to a target domain. Loom stores it as one target index per source element.

Internal axes. Fixed-size axes visible to the user function and scalarized by Loom.

Iteration domain. The domain Loom walks on one domain axis, chosen from the domains declared by the operands on that axis.

Jacobian–vector product (JVP). A forward-mode AD operation that maps an input tangent through a function's Jacobian.

Jaxpr. JAX's typed, functional intermediate representation of a traced program.

Just-in-time (JIT) compilation. Compilation performed when a program or a new input signature is first used.

Lane axis. Loom's last, contiguous physical axis. It has shared extent u or singleton extent one, and its lanes are independent.

Lane function. The user's JAX function for one domain position and one hidden lane. It may manipulate fixed internal arrays.

Machine-learned interatomic potential (MLIP). A learned function that predicts atomistic energy and usually obtains forces by differentiation.

Polynomial region. A maximal connected subgraph of additions, subtractions, multiplications, and negations. Its incoming values are its variables, and its constants are its coefficients.

PTX. NVIDIA's documented virtual instruction set for GPUs. A driver or assembler converts it to target machine code.

Real-arithmetic contract. Loom may replace a polynomial region by any expression equal to it over the real numbers, rather than follow every exceptional rule of IEEE-754 floating point.

Recipe. Loom's complete executable plan: PTX modules, buffer sizes, launch dependencies, ports, and resource bounds.

Reference lowering. Loom's portable non-CUDA implementation of the same domain and lane contract in plain JAX, without polynomial rewriting.

Register spilling. Moving live values that do not fit in GPU registers to slower thread-local memory.

Reverse mode. AD that propagates cotangents from outputs to selected inputs. It is efficient when there are few scalar objectives.

Scalarization. Replacing a fixed-size internal array program with explicit scalar dependencies while retaining the parallel lane axis.

Scatter. Writing through an index map. A scatter-add combines values that select the same destination.

Segmented polynomial. A sparse polynomial description whose operand dimensions are split into named or sized segments connected by paths.

StableHLO. A versioned, portable operation set for machine-learning computations and the input language of Loom's lowerer.

Target-index array. The array representing an index map. Each source element stores its target index, or -1 when it is unmapped.

Temporary memory / scratch. Intermediate storage requested by a compiled executable, excluding arguments and final outputs in this paper.

Vector–Jacobian product (VJP). A reverse-mode operation that multiplies output cotangents by a function's Jacobian to produce input cotangents.

XIR. Loom's typed intermediate representation after StableHLO lowering and before GPU planning.

XLA. The general optimizing array compiler used by JAX. It delegates dense matrix multiplication to tuned libraries.